The Journal of Supercomputing (2023) 79:15390–15414
https://doi.org/10.1007/s11227-023-05243-x

BejaGNN: behavior-based Java malware detection via graph neural network

Pengbin Feng[1] · Li Yang[2] · Di Lu[2] · Ning Xi[1] · Jianfeng Ma[1]

Accepted: 29 March 2023 / Published online: 17 April 2023

Abstract

As a popular platform-independent language, Java is widely used in enterprise applications. In the past few years, language vulnerabilities exploited by Java malware have become increasingly prevalent, which cause threats for multi-platform. Security researchers continuously propose various approaches for fighting against Java malware programs. The low code path coverage and poor execution efficiency of dynamic analysis limit the large-scale application of dynamic Java malware detection methods. Therefore, researchers turn to extracting abundant static features to implement efficient malware detection. In this paper, we explore the direction of capturing malware semantic information by using graph learning algorithms and present BejaGNN (Behavior-based Java malware detection via Graph Neural Network), a novel behavior-based Java malware detection method using static analysis, word embedding technique, and graph neural network. Specifically, BejaGNN leverages static analysis techniques to extract ICFGs (Inter-procedural Control Flow Graph) from Java program files and then prunes these ICFGs to remove noisy instructions. Then, word embedding techniques are adopted to learn semantic representations for Java bytecode instructions. Finally, BejaGNN builds a graph neural network classifier to determine the maliciousness of Java programs. Experimental results on a public Java bytecode benchmark demonstrate that BejaGNN achieves high $F1$ 98.8% and is superior to existing Java malware detection approaches, which verifies the promise of graph neural network in Java malware detection.

Keywords Java malware detection · Graph neural network · ICFG · Word embedding

✉ Pengbin Feng
pbfeng@xidian.edu.cn

Extended author information available on the last page of the article

1 Introduction

Java, the most popular development language used in enterprise application [1], continues to be an attractive target for attackers [2]. According to `Java.com`, 89% of Desktops, 3 billion mobile phones, and 97% of Enterprise Desktops in the USA run Java. Therefore, malicious programs, vulnerabilities and exploits in Java have become increasingly prevalent in the past few years. MITRE's CVE dataset recorded nearly 700 new Java vulnerabilities [3].

Recent research [2] points out that Java malware is mainly spread via malicious attachments or phishing emails. These malware files are organized in JAR (Java Archive) compression format, which can run on any infected system with Java Runtime Environment (JRE). In consideration of cross-platform convenience, the JAR file aggregates all Java class files, resources, and associated metadata into one archive. With everyone's focus on COVID-19, attackers recently leverage the popular COVID-19 update maps to infect computers via Java malware [4] silently. In this attack, malicious Java applications leverage the real-time data map component to camouflage as benign and contain a payload to steal sensitive information.

Despite receiving less attention from the research community compared to Android, Windows, or IoT (Internet of things) malware, Java malware causes great harm to the Java ecosystem [5]. Anti-malware industry or anti-virus software widely adopts signature-based malware detection approaches [6]. These approaches rely on the built signatures from huge known malware samples to match similar malware files. The signature matching mechanism could be easily bypassed by code transformation techniques [7]. To overcome this issue, researchers have proposed more precise detection approaches via building advanced machine learning models on a set of meaningful static or dynamic features [8–10].

With the ability to resist code obfuscation via execution within sandbox environments, dynamic analysis techniques could still be evaded by environment-aware and event-trigger malware [11]. In addition, dynamic analysis techniques can hardly cover all execution paths and cause more time and resource consumption for the execution environment, which limits the practicality of these methods. Traditional static analysis methods mainly extract syntax features such as strings or imports to identify malware samples. While syntax features achieve efficient malware detection, these static approaches cannot capture sufficient semantic information from malware behaviors [12]. Recently, researchers have explored image-based malware detection approaches which leverage the booms of ML algorithms in computer vision [13–15]. These approaches usually covert raw malware binaries into pixel-based images via various transformation methods. Then, ML algorithms can directly leverage image features to classify malware binaries rather than behavior or signature features. Most of these approaches treat machine code as the pixel values and convert the binary files into grayscale images [13]. Jadeite [15] adopts an image transformation method on ICFG of Java bytecode, which improves the accuracy of image-based malware detection methods. Nevertheless, these image transformation methods directly use image processing techniques on bytecode files or pruned ICFGs, which limits the capture of informative Jimple [16] instruction semantic.

One emerging approach in ML for cybersecurity tasks is to use graph learning to capture critical information directly from program representation graph [17–19]. In these approaches, researchers use multiple graph representation structures, including CFG (Control Flow Graph), CDG (Control Dependence Graph), DDG (Data Dependence Graph) or even fusion graph CPG (Code Property Graph) [20], to denote semantic information. GNN (Graph Neural Network) algorithms directly use these graph structures to perform malware vulnerability detection rather than sequence-based or tree-based approaches [21]. Most of these approaches adopt various GNN algorithms to capture attack characteristics directly from informative graph structures without manually constructing features via expert knowledge. With the innate ability to handle graph structures, GNN algorithms are promising in the next-generation cyber security systems [22].

In this paper, motivated by the advantages of graph-based learning algorithms, we explore the direction of capturing malware semantic information and propose a novel behavior-based Java malware detection method via graph neural network. Stand apart from existing Java malware classification methods using image-based approaches or traditional ML algorithms, we use the ICFG at the bytecode level to represent the behavior of Java programs and capture critical relational patterns via GNN algorithms. Previous studies have proved that ICFG is one of the most compelling features to capture malicious program behaviors [23–25]. After generating ICFGs, we generate node embeddings of ICFGs and leverage advanced ML algorithms in the graph classification domain. In particular, we perform static analysis to extract ICFG from Java bytecode files. After obtaining these graphs, we prune these graphs to retain instructions with meaningful information. Next, we leverage program representation methods to capture semantic information from basic blocks inside ICFG. Finally, we use GNN to classify the generated ICFGs to identify the maliciousness within the original Java bytecode programs. With the help of handling complex graph structure of the GNN algorithm, BejaGNN achieves superior detection performance compared to existing approaches. Experimental results show that GNN is a promising technique in the Java malware detection domain.

In summary, we make the following contributions:

- We propose a novel Java bytecode classification framework, namely BejaGNN, that utilizes code embedding techniques and graph neural networks to identify the maliciousness of Java bytecode programs. We directly use ICFG to capture behavioral features of Java bytecode programs and then build the detection model from these features to leverage the remarkable success in the graph learning domain.
- We adopt multiple code embedding techniques and GNN algorithms to verify the effectiveness and explore the best performance of the proposed detection framework.
- We evaluate our approach on a public Java bytecode benchmark, which consists of around 4k malicious and benign samples. Our proposed method achieves 99.1%, which outperforms existing Java bytecode classification methods.

The rest of the paper is organized as follows. Section 2 summarizes recent work on Java malware detection. Section 3 describes the system design of BejaGNN. Section 4 reports the experimental settings and evaluates the detection performance of BejaGNN. Section 5 outlines the limitations of BejaGNN. Section 6 concludes this paper.

2 Related work

In this section, we review the malware detection methods in general and related to Java malware.

2.1 Traditional malware detection methods

With malware continuing to compromise cyberspace security, malware detection has received much attention from industry and academia. Conventional anti-virus software leverages signature-based methods to perform efficient malware detection [6]. However, these methods heavily rely on the collected known malware dataset, which could hardly detect new malware or malware variants generated via evading techniques [26] such as code obfuscation, packing, and encryption mechanism. At an early age, researchers propose to extract CFG by using dynamic analysis and leverage subgraph isomorphism to combat malware variants [27]. However, malware could insert junk function calls and reshape the structure of the malware to evade isomorphism mechanism [28].

2.2 Machine learning-based malware detection methods

Recent research leverages advanced machine-learning approaches to automatically infer critical malicious behavior properties, which could identify newly generated malware variants. The machine learning methods use the behavior features of the malware samples to perform malware classification and detection. These features are typically extracted via static, dynamic, or hybrid analysis [29]. Static analysis extracts statistical or string features from assembly or intermediate representative code [30, 31]. Although static extracted features are effective in malware detection, these methods still struggle against code obfuscation [32, 33]. On the other hand, dynamic analysis approaches can collect runtime temporal and behavioral information, such as a sequence of system calls and process states, to recognize malware [34]. Younghee et al. [34] proposed a classical common behavior graph representation method that achieve high malware detection rates. MtNet [9] adopts a multi-task learning method to improve the accuracy of dynamic malware detection. The designed neural network structure could implement malware detection and family classification at the same time. However, the inherent pitfalls of dynamic analysis are the requirement of time and resources consumed in executing malware and the limited coverage of the execution path, which may miss the critical part of the

malware code. These limitations hinder the preferred application of dynamic-based approaches.

2.3 Image-based malware detection methods

Image-based malware detection methods leverage static analysis techniques to convert malware binaries into pix-based images, which owns the ability to mitigate code obfuscation and encoding issues [35–40]. These images can be fed into mature image classification techniques for identifying malicious samples. Pratikkumar et al. [40] conduct a comprehensive empirical evaluation on various famous image-based learning techniques for malware classification. Cui et al. [36] transform malicious binary files into a two-dimensional array of a specific length. Next, a CNN classifier is used to classify the grayscale images mapped from the two-dimensional arrays. Jadeite [15] proposes to generate grayscale images from ICFGs of Java bytecode files, which considers the program's semantics information and improves the detection performance. Nevertheless, the image processing-based approach, achieving extraordinary results in malware detection, encodes malware into an abstract image, which limits the exploration of Jimple instruction semantics and increases the difficulty in providing explainable results.

2.4 Java malware detection methods

In this paper, we focus on malware detection on the Java platform as the widespread application of Java in enterprise platforms and the rapidly increasing malware in the wild [41]. Android applications, written in Java, have received much attention for their malicious detection [42]. Several approaches adopt similar image transformation approaches to perform Android malware detection [43–45]. For example, Ding et al. [43] proposes a bytecode image-based malware detection approach. It extracts bytecode files from the files of Android applications and treats each byte in this file as pixel values. Then, the bytecode files are transformed into grayscale images. Finally, the CNN classifier is used to identify maliciousness from the generated images. Vasan et al. [39] leverages a color-mapping mechanism to convert the raw Android applications into images and then perform malware detection and family classification on these images. These approaches treat the raw binaries as images for using advanced computer vision algorithms but miss the semantic execution information of these malware files.

Although being developed by Java language, Android applications are different from Java programs from the following aspects: a) organize in different file formats (`.class` vs `.dex`); b) execute in different virtual machine system (Dalvik Virtual Machine vs Java Virtual Machine); c) Android mainly focuses on the mobile platform; meanwhile, Java is widely used in the enterprise; d) Java programs only have one main method, while Android applications are based on complex component and configuration mechanism. These differences hinder the Android malware detection methods from applying to the Java malware programs. Few works in the literature investigate Java malware detection [46]. Jarhead [47] performs simple static

analysis to extract string features and trains machine learning algorithms for Java malware detection. Gassen et al. [48] propose a dynamic analysis-based method to combat common obfuscation techniques. JMD [49] combines symbolic execution and instrumentation techniques with dynamic analysis to improve malware detection. Kumar et al. [50] design a lexical analysis-based approach to identify code obfuscation. Recently, researchers have proposed to perform Java malware program detection by using static and dynamic analysis [11, 51]. Apart from these methods mentioned above, we design a novel Java malware detection approach, BejaGNN, which combines word embedding techniques and GNN algorithms to capture critical Jimple semantic information from the ICFG representations of Java programs.

3 System design

The goal of BejaGNN is to directly learn critical behavior features from the ICFG structure for malicious Java application detection. Figure 1 presents the overall architecture of our malware detection system, BejaGNN, which consists of three main components, namely *ICFG Extraction*, *Node Embedding Generation* and *GNN Classification* ICFG extraction firstly leverages static analysis to transform a Java bytecode file, in the form of JAR format, into an ICFG structure. Then, node embedding generation captures semantic information from basic blocks inside ICFG by using program representation methods. Finally, GNN classification takes the ICFG structure as input and trains a GNN classifier to perform malicious bytecode file identification. In the following, we introduce the details of each component.

3.1 ICFG extraction

In this component, BejaGNN extracts the Inter-Procedural Control Flow Graph (ICFG) from the Java bytecode file via Soot [52] analysis framework. ICFG is a direct graph that covers the execution order of all instructions within a program. In the ICFG graph, the nodes represent sequentially executed program instructions, and the edges denote execution orders between nodes.

The JAR file is packed in a compressed format and consists of one or more class (Java bytecode) files compiled from Java source code. BejaGNN first leverage Soot to convert the Java bytecode included in the JAR file into Jimple intermediate representation. Soot is a popular analysis framework and provides a convenient

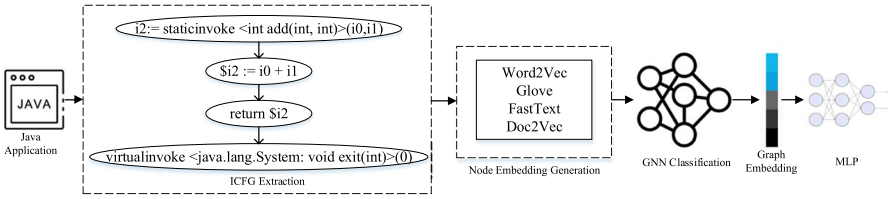

Fig. 1 Architecture of BejaGNN system

intermediate representation for customized static analysis. Thus, it is widely used for various analysis purposes, including security problem [53], bug finds [54] and privacy issues [55], etc. Then, BejaGNN uses Heros [56] to build the ICFG from Jimple representation of the whole Java program. Heros is built on top of Soot and owns the ability to perform inter-procedural dataflow analysis.

The Soot analysis framework provides the functionality of generating ICFG in DOT format, which is easy to visualize and handle. Figure 2 presents a simple example to illustrate the ICFG extraction process. Figure 2 presents a simple Java program with two functions `func1` and `add`. In this program, `func1()` invokes `add()`, and each function contains several instructions. The right part of Fig. 2 shows a graph representation of Jimple ICFG without meaningless variable definitions and initializations. In the graph, the nodes represent Jimple instructions, and the edges between nodes denote the control flow between instructions.

Considering all unique nodes from ICFG graphs of our entire Jimple instructions as features would influence the effectiveness and efficiency of Java malware detection via importing noises and increasing training time. Therefore, BejaGNN removes meaningless nodes in the generated ICFG. Specifically, we remove meaningless instructions, such as variable definitions and initializations, which are necessary for the correct execution of Java programs but hardly contribute to understanding program execution semantics. For the remaining instructions, BejaGNN leverages *Node Embedding Generation* and *GNN Classification* components to extract critical information.

3.2 Node embedding generation

To convert ICFG into structures that are applicable to graph neural networks, we need to extract numerical vectors that summarize the semantic information of each node. We treat Jimple instructions as words to generate node embeddings that capture semantic information via the state-of-the-art NLP technique Word2Vec. The NLP embedding technique has been proven to be effective in capturing informative semantics from malware [57, 58]. Before generating node embeddings, we normalize the Jimple instructions to remove meaningless code differences. In this paper, we refer to meaningless code differences as opcodes within the same category, various

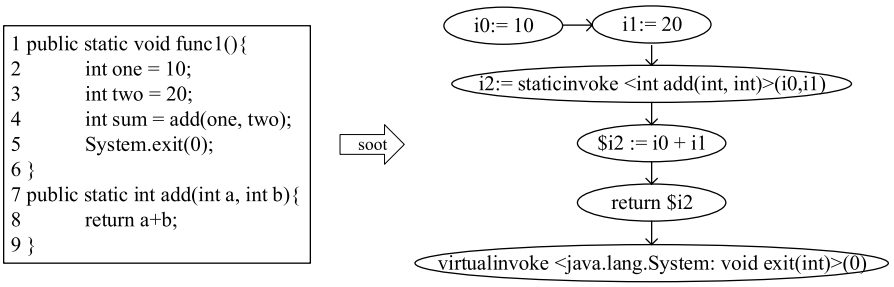

Fig. 2 Simple Java program with corresponding Jimple ICFG

numeric constant values, and local variable definitions. Opcodes within the same category may denote operations toward different types of objects but cause similar effects to the program. Numeric values are infinite, and their semantics are difficult to capture. Same local variables from Java programs usually represent different objects. The normalized process could reduce the possibility of entering instructions with a low occurrence in the training data, or are absent from the training data, and further mitigate the OOV (out-of-vocabulary) [59] problems. The generation of node embedding consists of normalization and Word2Vec.

3.2.1 Normalization

In this step, the Jimple instructions are scanned one by one along the execution sequence of the Java program. Before further processing, we narrow down the diversity of the Jimple instructions by removing meaningless instructions that are greatly affected by the programmers' coding habits. In addition, a majority of Jimple instructions share similar semantic information (e.g., `specialinvoke` and `virtualinvoke`), such as operand types and definition forms. Although diverse Jimple instructions provide detailed information for Java programs, they significantly increase the burdens of security analysis. Instruction normalization is necessary means to improve efficiency in the security research domain, such as binary diffing [60], patch detection [61], and even malware detection [62]. According to our observation on all extracted Jimple instructions, BejaGNN performs instruction normalization via the following rules: (1) replacing all variable definitions as unified "*var*" to reduce differences introduced by huge variable names; (2) replacing all numeric constant values with "*num*" to reduce wide range of changes; (3) replacing the Jimple instructions within one category as one simplified representation to focus on the semantic information of each instruction.

Figure 3 shows an example of Jimple normalization process. In Fig. 3, the Jimple instruction `virtualinvoke`, `store.i` and `load.i` respectively belong to category `invoke`, `store` and `load`. Thus, the original instructions

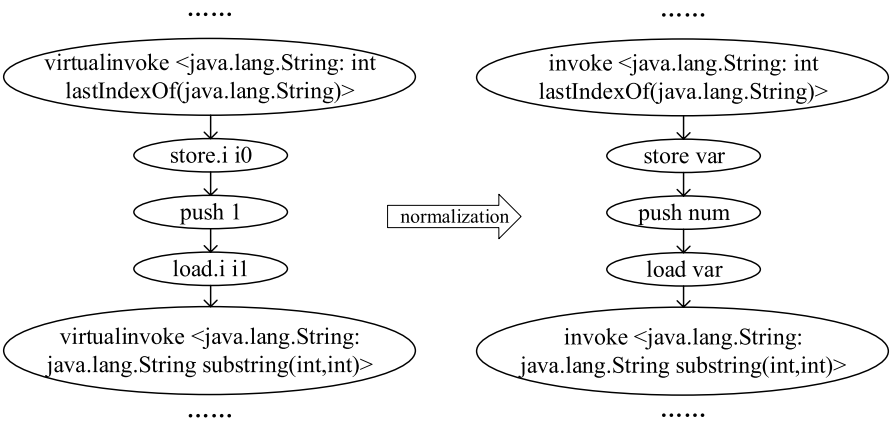

Fig. 3 Normalization process of Jimple Instructions

are replaced with the instruction categories according to the normalization rules. The load variables `i0` and `i1` are replaced with *var*, and the constant number is replaced with *num*. Other meta-information within instructions, such as constant string, class name, class name, and fields, are reserved to ensure integrity.

3.2.2 Word2Vec

In this phase, BejaGNN represents each normalized Jimple instruction with a dense and real-valued vector, which covers semantic information according to its surrounding context. The context represents neighbor instructions within one function. Figure 4 shows the workflow of Word2Vec, which consists of instruction grouping, Word2Vec modeling, and embedding generation.

Instruction grouping. BejaGNN firstly builds Jimple instruction vocabulary extracted from all Java bytecode files. Then, it constructs instruction groups by matching each instruction with its local context. Let J_i denotes the ith instructions, and c represents half of the window size. For each instruction J_i, its local context $\{J_{i-c}, J_{i-c+1}, ..., J_{i+c-1}, J_{i+c}\}$ consists of ahead and behind neighborhood instructions within one function. Accordingly, BejaGNN generates $2c$ instruction groups by matching J_i with each instruction in its context.

Word2Vec Modeling. After generating instruction groups, BejaGNN leverages the Word2Vec algorithm to train an instruction embedding model by taking each instruction group as a training instance. This embedding model only needs training once because it is shared by all Jimple instructions. During the training process, each instruction is firstly represented in the one-hot encoding form. In the Continuous Bag of Words (CBOW) model, the Word2Vec matches the target instructions with context instructions. It takes J_i as output and context $\{J_{i-c}, J_{i-c+1}, ..., J_{i+c-1}, J_{i+c}\}$ as input and learns an embedding space which maximizes the similarity between the instruction and its context instructions. On the contrary, in the Skip-gram model, the Word2Vec predicts the context instructions with the target instructions and turns around the input and output. The objective of Word2Vec is presented as follows:

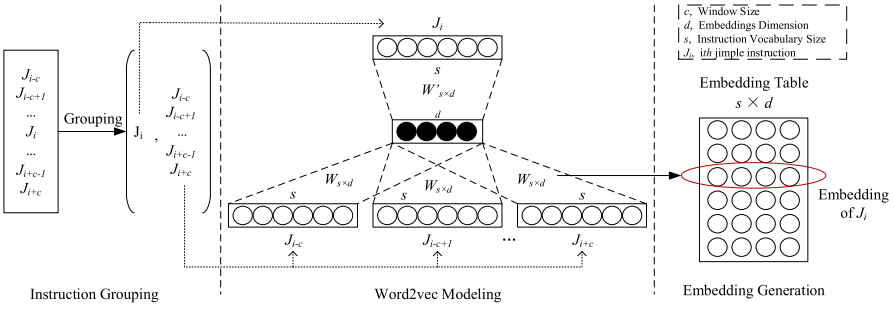

Fig. 4 Workflow of Word2Vec

$$\pounds(J) = \frac{1}{s}\sum_{i=1}^{s}\sum_{c\in C} p(J_c \mid J_i) \tag{1}$$

where C represents context instructions of each target instruction J_i. $p(J_c \mid J_i)$ is defined as

$$p(J_c \mid J_i) = \frac{\exp\left(v_{J_c} \cdot v_{J_i}\right)}{\sum c \in C \exp\left(v_{J_c} \cdot v_{J_i}\right)}, \tag{2}$$

where v_{J_c} and v_{J_i} are the embedding representations of instruction J_c and J_i.

Embedding Generation. After training the Word2Vec modeling with the matched Jimple instruction groups, BejaGNN generates two $s \times d$ weight matrices W and W', where s denotes the size of Jimple instruction vocabulary and d represents the dimension of embeddings. In particular, BejaGNN chooses the default matrix W to represent the embeddings of all Jimple instructions. For example, the instruction embedding of J_i is located at the ith row of W. Thus, the matrix W serves as the embedding search table for target Jimple instruction.

3.2.3 Alternative embedding methods

BejaGNN adopted multiple embedding methods: Word2Vec, GloVe, FastText and Doc2Vec, and compared their performance to select the most effective method to embed Jimple intermediate code.

GloVe [63] owns the ability to preserve the co-occurrence probability of instructions. The inner product between the two instruction embeddings by GloVe represents the cosine similarity and the probability of co-occurrence. Thus, GloVe takes into consideration of the statistical information within the entire instruction vocabulary, which makes it easier to capture the semantic information of each instruction. GloVe defines the instruction–instruction co-occurrence matrix X, whose entries $X_{i,k}$ represent the number of times instruction J_k occurs in the context of instruction J_i, and $X_i = \sum_k X_{ik}$ denotes the number of times any instruction appears in the context of instruction J_i. The cost function of this model is shown as follows:

$$\pounds(J) = \sum_{i,k}^{s} f(X_{i,k})(v_i^T v_k + b_i + b_k - \log(X_{i,k}))^2, \tag{3}$$

where v_i and v_k represent the embeddings of instructions J_i and J_k, b_i and b_k denote the additional biases, and $f(X_{i,k})$ is a weighting function generalized the similarity between instructions.

FastText [64] adopts similar training methods as the Word2Vec model but represents and divides the instructions as sub-instructions. Each instruction is expressed as a bag of character n-grams. In the form of this sub-instruction representation, the number of vectors for each instruction is increased, and the ability to capture semantic information is improved. For the instructions with a low appearance rate, FastText could increase the number of reference cases via the nature of embedding with n-grams. Given an instruction J_i, the set of n-grams appearing in J_i is represented

as Z_{J_i}. An instruction is represented by the sum of the vector representations of its n-grams:

$$\sum_{z \in Z_{J_i}} v_z^T v_c. \tag{4}$$

Doc2Vec [65] regards the bytecode file id as a single instruction. The file id has positional coordinates in the semantic space. Subsequently, from all the snapshots, the context vector is created by taking the average of the position coordinates of the other instructions. The remaining operation is the same as that of Word2Vec. That is, Doc2Vec updates the file embedding such that the file id and the instruction appearing in each bytecode file approach each other. In this way, the file embedding is capable of constructing representations of instructions sequences of variable length. Thus, even if the instructions are different, the embeddings of each file become similar because the embedding vectors of the instructions are similar. The Doc2Vec owns two models: Distributed Memory Model of Paragraph Vectors (PV-DM) and Distributed Bag of Words version of the Paragraph Vector (PV-DBOW), which takes the file id as input or output, respectively.

3.3 GNN classification

With the ability to capture comprehensive information from non-Euclidean data structures, GNN has recently received a lot of attention via the generation of graph embeddings [66]. In this paper, we explore the ability of GNN in Java malware detection. Therefore, BejaGNN adopts GCN, GAT and GIN, and their performance is compared to identify the most effective method for capturing malicious behavior patterns.

After the processing of the *Node Embedding Generation*, we obtain a number of ICFGs with corresponding node attributes. Specifically, we treat each Jimple instruction as one node to highlight the importance of semantic information. We define an ICFG as $G = (N, E)$, in which N represents the node set and E represents the edge set. All node embeddings are combined as a new matrix X in which the i-th row x_i represents the embedding of Jimple instructions J_i. We define A as the adjacent matrix of the ICFG G. Then, the GNN transforms the tuple (A, X) into graph embedding $h_G \in \mathbb{R}^d$, where d is the predefined embedding dimension of the graph. Finally, the MLP classifies the input h_G into the category malware or benign.

After obtaining the tuple (A, X), GNN would capture the embedding representation of the graph G. During the graph learning process, each node J is associated with a set of hidden representations $\{\dots, h_J^t, \dots\}$, where t denotes the t-th GNN layer within the graph learning model and the initial representation is denoted by the node embedding, $h_J^0 = v_J$. At layer $t+1$, h_J^t aggregates the t-layer hidden representations from its neighbor nodes, which could be generalized as follows:

$$h_J^{t+1} = M(h_J^t, \{h_u^t, \forall u \in N_{(J)}\}), \tag{5}$$

where M represents an aggregation function, which varies in different GNN models. After iterative processing by T GNN layers, we finally obtain the hidden representations for each node h_J^T. During this iterative procedure, the node information is propagated deeper and deeper. Thus, the final hidden representations could capture far-away neighborhood information. The embedding vector of the whole graph G is formalized as:

$$z_G = \frac{\sum h_{J_i}^T,\ i \in \{1, \ldots s\}}{s}. \tag{6}$$

The GNN model aims to generate the whole graph embeddings [67], encoding the entire graph information into low-dimensional space by passing all node hidden representations through the readout function. Similarly, the node embedding is transformed from a graph node with local information into low-dimensional space. Different GNN models adopt multiple aggregation mechanisms. GCN is one representative GNN, which computes node hidden representations via the following formulas:

$$H^{t+1} = \sigma(\hat{A}H^t W^t) \tag{7}$$

where H^t denotes the representations at the t-layer for all the graph nodes, and H^0 is the initial embeddings generated by the *Node Embedding Generation* component for all nodes. W^t is the trainable weight matrix of the t-layer GCN. σ is an activation function that is usually set as ReLU. $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$, where $\tilde{D}$ is the degree matrix, and $\tilde{A} = A + I_s$. I_s denotes the identity matrix.

GAT makes some improvement on the propagation rule and assumes the neighbors' contribution is imbalanced for different importance to the central vertex. Thus, GAT adopts the attention mechanism to calculate the relative weights between two connected nodes before the neighbor information propagation. The output features for every node can be formalized as:

$$h_{J_k} = \sigma\left(\sum_{J_i \in N(J_k)} a_{ki} W h_{J_i}\right), \tag{8}$$

where a_{ki} represents the attention coefficient computed by a shared attention mechanism and indicates the importance of node J_k's features to node J_i.

Similarly, GIN adopts one powerful message aggregation function, which is shown below:

$$h_J^{t+1} = \mathrm{MLP}^{t+1}\left(\left(1 + \epsilon^t\right)h_J^t + \sum_{u \in N(J)} k_u^t\right), \tag{9}$$

where ϵ^t is a scalar learnable parameter and MLP stands for a multi-layer perception, which could aggregate comprehensive information.

4 Experiments and evaluation

According to the above design, we implement a novel behavior-based Java malware detection tool, called BejaGNN, based on Soot [52], python natural language processing Gensim module and graph learning library DGL. In this section, we discuss the evaluation of our proposed system BejaGNN. We first describe the experiment settings used in BejaGNN. Then, we discuss the results of our experiments.

4.1 Experimental settings and dataset

The machine we used to extract ICFG from Java programs was a workstation with Intel (R) Xeon (R) E5-2620 CPU (15 M Cache, 2 GHz) and 24 GB of RAM. Meanwhile, BejaGNN is evaluated on a PC equipped with Intel (R) Core (TM) i7 CPU (6 M Cache, 2.5 GHZ), 16 GB of RAM, and NVIDIA GTX 850 M. We randomly shuffle the dataset and split 70% for the training, 15% for validation, and the rest 15% for test.

We evaluate the performance of BejaGNN by using the dataset provided by Jadeite [15], which was collected from multiple public resources and is representative of Java malware in the wild. The original dataset consists of 1816 benign and 2223 malicious Java programs. With 139 programs unable to extract effective ICFG on our workstation, we finally obtained a benchmark dataset with 3901 Java programs.

4.2 Measure metrics

We utilized five widely used metrics: precision, recall, true negative rate, accuracy and $f1$-score, to evaluate the performance of our Java malware detection approach. These metrics are calculated based on true positive (TP), true negative (TN), false positive (FP) and false negative (FN). In the malware detection scene, a true positive (TP) indicates the count of detected malware programs that are truly malicious, and a true negative (TN) indicates the count of correctly identified benign programs. A false positive (FP) indicates the count of detected malware programs that are actually benign, and a false negative (FN) indicates the count of undetected malware programs. The detailed measure metrics are as follows:

- **Precision** The ratio of true positive programs to the total programs that are detected as malware. Precision $= \frac{\text{TP}}{\text{TP+FP}}$
- **Recall** The ratio of true positive programs to the total count of malware programs. Recall $= \frac{\text{TP}}{\text{TP+FN}}$
- **True negative rate (TNR)** The ratio of true negative programs to the total benign programs. TNR $= \frac{\text{TN}}{\text{TN+FP}}$
- **Accuracy (Acc)** The ratio of the sum of true positive and true negative programs to the total count of all programs. Acc $= \frac{\text{TP+TN}}{\text{TP+FP+TN+FN}}$

- **F1-score (F1)** The overall effectiveness denotes the harmonic mean of precision and recall. $F1 = \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$

For precision, recall, true negative rate, accuracy, and $f1$-score, the closer to 1, the better the detection performance. In particular, accuracy and $f1$-score are indicators of overall malware detection performance.

4.3 Comparison of embedding methods

In the following, we evaluate the performance of BejaGNN with different node embedding techniques.

Firstly, we set the graph learning algorithm as GCN to select the best embedding techniques in our Java malware detection scene. Table 1 shows the performance comparison of word-embedding algorithms. In the initial experiment, for GCN, the number of layers is 2, and the batch size and hidden dim are both 8. Except for embedding size, all word embedding algorithms are in the default setting. From Table 1, we observe that FastText achieves the best performance by taking advantage of subword learning, which is suitable for our dataset with limited tokens. The two models of Word2vec and Doc2vec show different results, which illustrates that the semantic information is influenced by the input–output format of context. The Glove achieves the worst performance in our case, which may be caused by the built co-occurrence matrix cannot accurately capture the similarity between instructions. Generally, Doc2Vec is known to show good performance in large corpus and increases the performance with dataset size, which does not work well in our limited tokens scene. Accordingly, we recommend applying FastText when designing a malware detection system with a limited corpus.

To compare the word embedding algorithm performance according to embedding size, we experimented by increasing the embedding size by 10 units from 20 to 120 for each algorithm with different models. The comparison results are shown in Fig. 5. From Fig. 5, we observe that the detection performance increases with the embedding size and starts to decrease when reaching the optimal performance. A large embedding size could carry more semantic information and further increase the detection performance. After obtaining the optimal performance, the model becomes overfitting and decreases performance with the embedding

Table 1 Detection performance comparison with different node embedding techniques

Node embedding	Precision	Recall	TNR	Acc	$F1$
Word2Vec (CBOW)	0.9230	0.9489	0.8965	0.9235	0.9326
Word2Vec (Skip-gram)	0.9153	**0.9752**	0.8963	0.9354	0.9402
GloVe	0.8915	0.9438	0.8522	0.8997	0.9120
FastText	**0.9305**	0.9746	**0.9109**	**0.9439**	**0.9497**
Doc2Vec (PV-DM)	0.9140	0.9727	0.8868	0.9303	0.9387
Doc2Vec (PV-DBOW)	0.8998	0.9717	0.8655	0.9218	0.9312

Bold values represent the best detection performance for each method

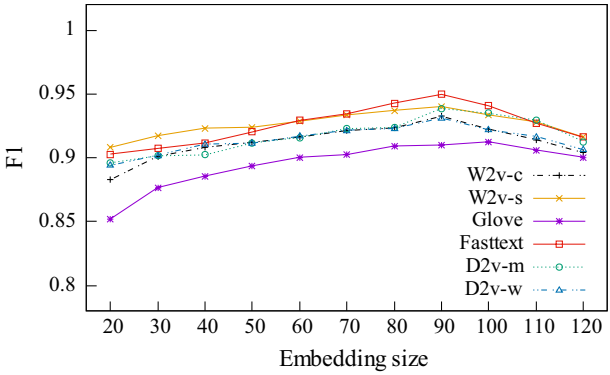

Fig. 5 Detection performance comparison with different embedding sizes. 'W2v-c' denotes Word2Vec in CBOW model. 'W2v-s' denotes Word2Vec in skip-gram model. 'D2v-m' denotes Doc2Vec in PV-DM model. 'D2v-W' denotes Doc2Vec in PV-DBOW model

size increase. We also infer that almost all word embedding algorithms follow a similar law of change, and all algorithms show the optimal performance at the embedding size 90, except for Glove. Accordingly, we recommend selecting optimal word embedding algorithms according to corpus characteristics within the dataset.

The performance of the word embedding algorithm is also affected by the context instruction size, which directly determines the information quantity within instruction embeddings. As FastText achieves the optimal performance, we then explore its optimal context size and conduct an experiment by increasing the context size by 2 units from 3 to 13. The detection performance results with various context sizes are shown in Fig. 6. From Fig. 6, we can infer that the overall detection performance increase with the context size and slightly changes after the context size reach 7. The larger context size will carry more information and improve the detection performance while increasing the time consumption

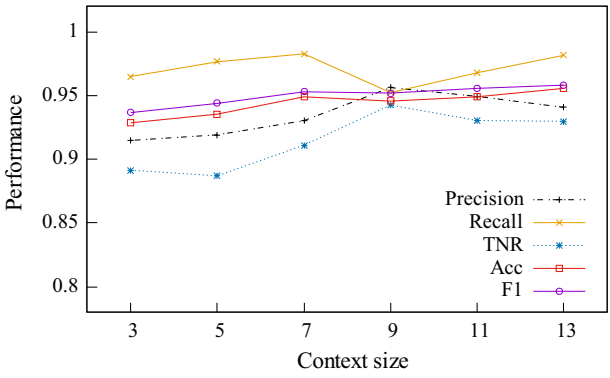

Fig. 6 Detection performance variation with different context sizes

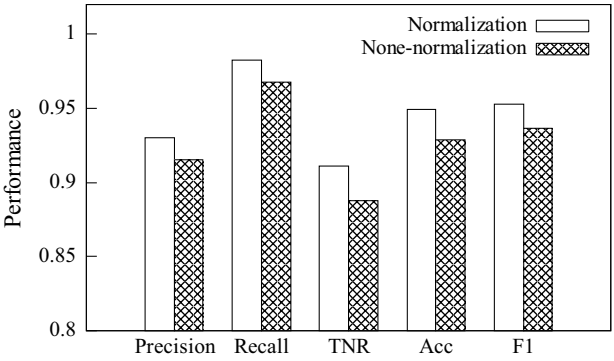

Fig. 7 Detection performance comparison with instruction normalization

Table 2 Search range of hyper-parameters and optimal values for graph neural network models

Hyper-parameters	GCN	GAT	GIN	Search Range
Learning Rate	0.01	0.01	0.01	{0.1, 0.01, 0.001}
Batch Size	256	256	256	{32, 128, 256, 512}
Epochs	100	100	100	{40, 60, 80, 100, 200, 300}
Hidden Layers	2	3	3	{2, 3, 4, 5}
Hidden dim	16	10	8	{8, 16, 32, 64} for GCN and GIN
Attention heads	–	10	–	6, 8, 10, 12, 14

in node embedding generation. Therefore, in our Java malware detection scene, we set the optimal context size as 7 to take a trade-off between performance and efficiency.

As described in Sect. 3.2, BejaGNN uses a normalization process to remove useless Jimple instructions, which contributes less to the semantics of the Java program. We experiment to explore the effectiveness of our instruction normalization process. The instruction normalization with non-normalization comparison results is shown in Fig. 7. From Fig. 7, we can observe that the instruction normalization process improves the detection performance from all measure metrics. This is caused by that our normalization process removes noisy and meaningless information from instructions and further improves the representation accuracy of instruction embedding.

4.4 Comparison of GNN algorithms

After obtaining the optimal node embedding structure, we start to explore the influence of multiple GNN algorithms.

We focused on tuning hyper-parameters which significantly affect the detection performance based on the expert knowledge from the deep learning community. The search range of hyper-parameters and optimal values of GCN, GAT and GIN are

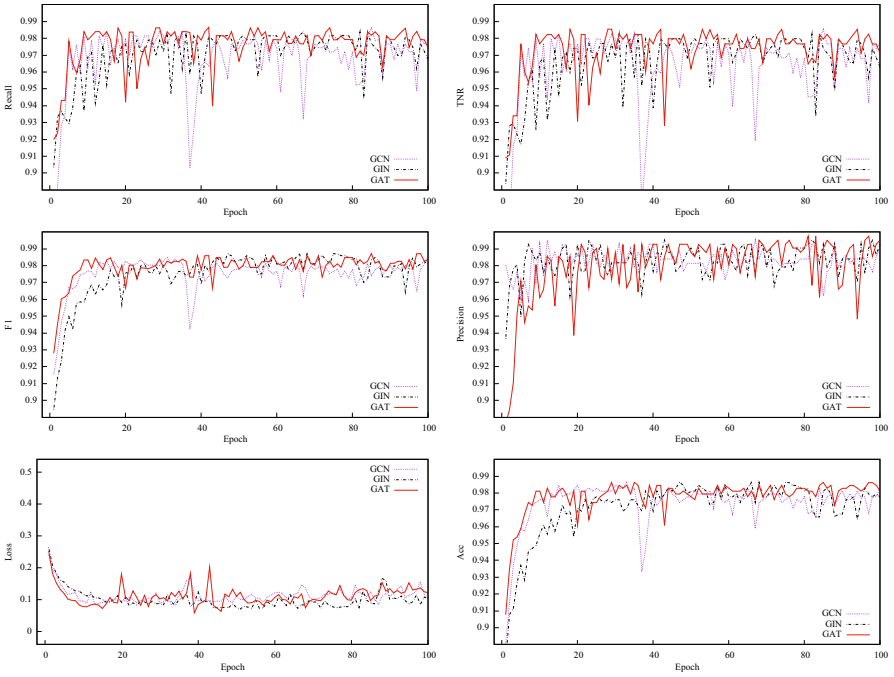

Fig. 8 Detection performance comparison of GNN algorithms with epochs

Table 3 Detection performance comparison with different graph neural network algorithms

Algorithms	Precision	Recall	TNR	Acc	*F*1
GCN	0.9701	0.9832	0.9591	0.9728	0.9767
GIN	0.9862	0.9835	0.9822	0.9813	0.9848
GAT	**0.9908**	**0.9858**	**0.9881**	**0.9864**	**0.9882**

Bold values represent the best detection performance for each method

shown in Table 2. We chose the default values implemented in the DGL library for other hyper-parameters. From Table 2, we can observe that all three graph neural network algorithms achieve the best performance at 100 epochs with a batch size of 256. The number of neural network layers generates the best performance in multi-layers. The neural networks' hidden dimensions achieve the best performance when GCN is 16, GIN is 8, and GAT is 10.

We compare the detection performance of BejaGNN by using three GNN algorithms: GCN, GIN and GAT. The optimal detection performance comparison is shown in Table 3. The detection performance variation with epochs is shown in Fig. 8. From Fig. 8, we can observe that our proposed Java malware detection framework achieves good performance on all three GNN algorithms, which illustrates that the effectiveness of our design and GNN is promising in Java malware detection. From Table 3, we can observe that GAT provides superior detection performance. The adaptation of

the powerful message aggregation function improves the detection performance of GIN when compared with GCN, and the experiment results of this study show the increased performance of GIN over GCN. In addition, the GAT algorithm utilizes an attention mechanism to filter important nodes during the neighbor information propagation process. Meanwhile, it owns the ability to handle directed graphs, which is more suitable for our Java ICFG scene. Therefore, the experiment results of BejaGNN show that GAT achieves the best performance when compared with GIN and GCN.

4.5 Comparison of other approaches

In the following, we compare BejaGNN with existing Java malware detection approaches, including Jarhead [47], BIN2PNG [68] and Jadeite on the public dataset to further verify the effectiveness of our proposed detection framework.

Jarhead is a classical Java malware detection method based on static analysis and machine learning and supports the JAR file classification. It manually extracts 42 dimension features, consisting of statistical information of bytecode, obfuscation-related invocation and well-known malicious operations, to represent the behavior of the Java program. Then, a decision tree classifier is built to detect malicious programs. BIN2PNG proposes an image processing-based malware detection method, which can be used to convert the JAR files into grayscale images. Next, these grayscale images are input to CNN-based classifier to build the detection model. The recently published method, Jadeite, proposes a new image-based Java malware detection method. It chooses to build grayscale images from the Java bytecode ICFG instead of the binary files and selects the most informative 20 features from Jarhead. Then, it designs a complex CNN-based fusion structure to jointly build a detection model based on the above image and selected features.

The final comparison results are shown in Table 4. In Table 4, Jadeite (IL) represents the detection model based only on the grayscale image, Jadeite (IL) denotes detection based on informative 20 features and flattened image matrix, and Jadeite (DL) indicates the final fusion model. From Table 4, we observe that our BejaGNN achieves the highest detection performance when compared with existing approaches and even outperforms Jadeite in fusion model form. Jarhead achieves good malware detection performance, but enters a bottleneck. This is caused by past manually selected features that cannot handle recent complex Java malware and require expert knowledge to refine new representative features. The Jadeite (IL) outperforms BIN2PNG, which illustrates that the grayscale images converted from ICFGs contain more information than those images converted from binary files. This phenomenon indicates that the ICFG represents one informative behavior feature. In this paper, we design a GNN-based architecture to detect Java malware directly from ICFG effectively. Therefore, GNN-based detection framework is promising in Java malware detection.

Table 4 Detection performance comparison with existing approaches

Approaches	Behavior feature	Model	Precision	Recall	Acc	*F*1
Jarhead	42 Manually selected features	Decision Tree	0.95	0.95	0.948	0.95
BIN2PNG	Grayscale image converted from binary file	CNN	0.84	0.84	0.84	0.839
Jadeite (IL)	Grayscale image converted from ICFG	CNN	0.95	0.95	0.95	0.95
Jadeite (SL)	20 Informative features concatenated with flatten grayscale image	CNN	0.93	0.93	0.928	0.93
Jadeite (DL)	Aggregation of the above two feature spaces	CNN based fusion model	0.98	0.98	0.984	0.98
BejaGNN	**Only ICFG**	**GNN**	**0.9908**	**0.9858**	**0.9864**	**0.9882**

Bold values represent the best detection performance for each method

5 Discussion

Although BejaGNN is useful and effective for detecting Java malware, there are still limitations in current BejaGNN systems. BejaGNN proposes a behavior-based Java malware detection approach by using static analysis and graph neural network algorithm to detect Java malware, which inherits the main limitations from static analysis. Java malware may employ active code obfuscation techniques, such as polymorphism mechanism, reflection invocation and dynamic exploit loading, to modify their structure and behavior at runtime. This would hinder the correctness and integrity of the generated ICFG, and further threaten the detection performance of BejaGNN. The obfuscation issues may be mitigated by using dynamic CFG recovery approaches to construct accurate ICFG for malware files. On the other hand, BejaGNN owns the ability to handle Java malware programs with trivial obfuscation techniques, such as identifier renaming, data encoding, junk code insertion, etc. In addition, when applying deep learning models in security domains, almost all deep learning-based approaches suffer from black box issues. This hinders security researchers from obtaining critical malicious features and exploring the evolution of malicious behavior. To solve this problem, explainable GNN models [69] can be applied to extract critical malicious patterns from Java malware programs.

6 Conclusion and future work

Due to the pervasive and convenient nature of the Java platform, its malware continues to be a significant threat to enterprise security. In this paper, we propose a novel approach, namely BejaGNN, to classifying Java bytecode programs by using static analysis, code semantic representation techniques and graph neural networks. BejaGNN leverages static analysis to extract Jimple inter-procedural CFG, which is directly used to represent the behavior of Java programs. Then, the word embedding technique is leveraged to capture the semantic information from Jimple instructions. Finally, we adopt a GNN classifier to classify vectorized ICFG to detect maliciousness in original programs. Experimental results on a pubic dataset demonstrated that our BejaGNN achieves superior detection performance than existing Java malware detection approaches. In addition, the combination of code semantic representation techniques and graph neural network algorithms is promising in Java malware detection.

In the current prototype, we only used ICFG to represent the behavior of Java bytecode files, which miss control dependence and data transformation information. We plan to leverage the comprehensive fusion graph CPG to sufficiently represent the behavior of Java programs, which would enhance the applicability of BejaGNN. Current BejaGNN only utilizes three basic GNN algorithms, which can be improved by self-supervised learning-based GNN models [70] to capture more critical structure information. In addition, we plan to adopt advanced

program language learning models, such as HMM2Vec, ELMo, and BERT [58], to replace current word embedding techniques, which could precisely capture the semantic information from Jimple instructions. As combing advanced program language learning models with graph learning algorithms would increase the complexity, resource and time consumption of detection models, which limits the large-scale deployment of detection methods, we plan to explore which combination mechanism is more efficient and effective to combine advanced language models with simple learning algorithms or combine simple word embedding models with complex learning algorithms like GNN.

Author contributions The authors confirm their contribution to the paper as follows: PF involved in methodology, investigation and Design and original manuscript writing; LY involved in data collection and prototype implementation; DL involved in experiments, visualization and result analysis; NX involved in manuscript writing—review and editing and supervision; JM involved in manuscript writing—review and editing, supervision and funding acquisition. All authors reviewed the results and approved the final version of the manuscript.

Funding This research was funded by the Natural Science Basic Research Program of Shaanxi (Program No.2023-JC-QN-0759).

Availability of data and materials The data that were utilized to substantiate the study's findings are included in the article.

Declarations

Conflict of interest The authors declare that there are no conflicts of interest regarding the publication of this article. All authors have contributed to, read, and approved this submitted manuscript in its current form. The authors declare that they have no competing financial interests.

Ethical approval Not applicable.

References

1. Java.com, Learn About Java Technology. https://www.java.com/en/
2. Balan G, Popescu AS (2018) Detecting java compiled malware using machine learning techniques. In: 2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). IEEE, pp 435–439
3. CVE Details, J, The Ultimate Security Vulnerability Datasource. https://www.cvedetails.com/product/19116/Oracle-JDK.html?vendor_id=93
4. Krebson Security, C, Live Coronavirus Map Used to Spread Malware. https://krebsonsecurity.com/2020/03/live-coronavirus-map-used-to-spread-malware/
5. Coker Z, Maass M, Ding T, Le Goues C, Sunshine J (2015) Evaluating the flexibility of the java sandbox. In: Proceedings of the 31st Annual Computer Security Applications Conference, pp 1–10
6. Ye Y, Li T, Adjeroh D, Iyengar SS (2017) A survey on malware detection using data mining techniques. ACM Comput Surv (CSUR) 50(3):1–40
7. You I, Yim K (2010) Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications. IEEE, pp 297–300
8. Dahl GE, Stokes JW, Deng L, Yu D (2013) Large-scale malware classification using random projections and neural networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, pp 3422–3426

9. Huang W, Stokes JW (2016) Mtnet: a multi-task neural network for dynamic malware classification. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp 399–418
10. Kolosnjaji B, Zarras A, Webster G, Eckert C (2016) Deep learning for classification of malware system call sequences. In: Australasian Joint Conference on Artificial Intelligence. Springer, pp 137–149
11. Jha PK, Shankar P, Sujadevi V, Prabhaharan P (2018) Deepmal4j: Java malware detection employing deep learning. In: International Symposium on Security in Computing and Communication. Springer, pp 389–402
12. Shalaginov A, Banin S, Dehghantanha A, Franke K (2018) Machine learning aided static malware analysis: a survey and tutorial. In: Cyber threat intelligence, pp 7–45
13. Le Q, Boydell O, Mac Namee B, Scanlon M (2018) Deep learning at the shallow end: Malware classification for non-domain experts. Digit Investig 26:118–126
14. Jian Y, Kuang H, Ren C, Ma Z, Wang H (2021) A novel framework for image-based malware detection with a deep neural network. Comput Secur 109:102400
15. Obaidat I, Sridhar M, Pham KM, Phung PH (2022) Jadeite: a novel image-behavior-based approach for java malware detection using deep learning. Comput Secur 113:102547
16. Vallee-Rai R, Hendren LJ (1998) Jimple: simplifying java bytecode for analyses and transformations. Technical report, McGill University
17. Yu Z, Cao R, Tang Q, Nie S, Huang J, Wu S (2020) Order matters: semantic-aware neural networks for binary code similarity detection. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol 34, pp 1145–1152
18. Gao H, Cheng S, Zhang W (2021) Gdroid: android malware detection and classification with graph convolutional network. Comput Secur 106:102264
19. Sun Q, Abdukhamidov E, Abuhmed T, Abuhamad M (2022) Leveraging spectral representations of control flow graphs for efficient analysis of windows malware. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, pp 1240–1242
20. Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp 590–604
21. Siow JK, Liu S, Xie X, Meng G, Liu Y (2022) Learning program semantics with code representations: an empirical study. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE
22. Bowman B, Huang HH (2021) Towards next-generation cybersecurity with graph ai. ACM SIGOPS Oper Syst Rev 55(1):61–67
23. Yang W, Kong D, Xie T, Gunter CA (2017) Malware detection in adversarial settings: exploiting feature evolutions and confusions in android apps. In: Proceedings of the 33rd Annual Computer Security Applications Conference, pp 288–302
24. Narayanan A, Chandramohan M, Chen L, Liu Y (2018) A multi-view context-aware approach to android malware detection and malicious code localization. Empir Softw Eng 23(3):1222–1274
25. Ou F, Xu J (2022) S3feature: a static sensitive subgraph-based feature for android malware detection. Comput Secur 112:102513
26. Anderson HS, Kharkar A, Filar B, Roth P (2017) Evading machine learning malware detection. black Hat **2017**
27. Macedo HD, Touili T (2013) Mining malware specifications through static reachability analysis. In: European Symposium on Research in Computer Security. Springer, pp 517–535
28. Osorio FCC, Qiu H, Arrott A (2015) Segmented sandboxing-a novel approach to malware polymorphism detection. In: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE). IEEE, pp 59–68
29. Damodaran A, Troia FD, Visaggio CA, Austin TH, Stamp M (2017) A comparison of static, dynamic, and hybrid analysis for malware detection. J Comput Virol Hacking Tech 13:1–12
30. Hardy W, Chen L, Hou S, Ye Y, Li X (2016) Dl4md: a deep learning framework for intelligent malware detection. In: Proceedings of the International Conference on Data Science (ICDATA). The Steering Committee of The World Congress in Computer Science, Computer, p 61
31. Athiwaratkun B, Stokes JW (2017) Malware classification with lstm and gru language models and a character-level cnn. In: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, pp 2482–2486

32. Lakhotia A, Preda MD, Giacobazzi R (2013) Fast location of similar code fragments using semantic'juice'. In: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, pp 1–6
33. Fass A, Backes M, Stock B (2019) Jstap: a static pre-filter for malicious javascript detection. In: Proceedings of the 35th Annual Computer Security Applications Conference, pp 257–269
34. Park YH, Reeves DS, Stamp M (2013) Deriving common malware behavior through graph clustering. Comput Secur 39:419–430
35. Yajamanam S, Selvin VRS, Di Troia F, Stamp M (2018) Deep learning versus gist descriptors for image-based malware classification. In: 2nd International Workshop on Formal Methods for Security Engineering (ForSE 2018), pp 553–561
36. Cui Z, Du L, Wang P, Cai X, Zhang W (2019) Malicious code detection based on cnns and multi-objective algorithm. J Parallel Distrib Comput 129:50–58
37. Cho M, Kim J-S, Shin J, Shin I (2020) Mal2d: 2d based deep learning model for malware detection using black and white binary image. IEICE Trans Inf Syst 103(4):896–900
38. Nisa M, Shah JH, Kanwal S, Raza M, Khan MA, Damaševičius R, Blažauskas T (2020) Hybrid malware classification method using segmentation-based fractal texture analysis and deep convolution neural network features. Appl Sci 10(14):4966
39. Vasan D, Alazab M, Wassan S, Naeem H, Safaei B, Zheng Q (2020) Imcfn: image-based malware classification using fine-tuned convolutional neural network architecture. Comput Netw 171:107138
40. Prajapati P, Stamp M (2021) An empirical analysis of image-based learning techniques for malware classification. In: Malware analysis using artificial intelligence and deep learning, pp 411–435
41. Acar A, Lu L, Uluagac AS, Kirda E (2019) An analysis of malware trends in enterprise networks. In: International Conference on Information Security. Springer, pp 360–380
42. Qiu J, Zhang J, Luo W, Pan L, Nepal S, Xiang Y (2020) A survey of android malware detection with deep neural models. ACM Comput Surv (CSUR) 53(6):1–36
43. Ding Y, Wu R, Xue F (2018) Detecting android malware using bytecode image. In: International Conference on Cognitive Computing. Springer, pp 164–169
44. Xiao X, Yang S (2019) An image-inspired and cnn-based android malware detection approach. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 1259–1261
45. Yadav P, Menon N, Ravi V, Vishvanathan S, Pham TD (2022) Efficientnet convolutional neural networks-based android malware detection. Comput Secur 115:102622
46. Pizzolotto D, Fellin R, Ceccato M (2019) Oblive: seamless code obfuscation for java programs and android apps. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp 629–633
47. Schlumberger J, Kruegel C, Vigna G (2012) Jarhead analysis and detection of malicious java applets. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp 249–257
48. Gassen J, Chapman JP (2014) Honeyagent: detecting malicious java applets by using dynamic analysis. In: 2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE). IEEE, pp 109–117
49. Herrera A, Cheney B (2015) Jmd: a hybrid approach for detecting java malware. In: Proceedings of the 13th Australasian Information Security Conference (AISC 2015). vol 27, p 30
50. Kumar R, Vaishakh ARE (2016) Detection of obfuscation in java malware. Procedia Comput Sci 78:521–529
51. Pinheiro R, Lima S, Fernandes S, Albuquerque E, Medeiros S, Souza D, Monteiro T, Lopes P, Lima R, Oliveira J et al. (2019) Next generation antivirus applied to jar malware detection based on runtime behaviors using neural networks. In: 2019 IEEE 23rd International Conference on Computer Supported Cooperative Work in Design (CSCWD). IEEE, pp 28–32
52. Lam P, Bodden E, Lhoták O, Hendren L (2011) The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011). vol 15
53. Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Not 49(6):259–269
54. Nistor A, Song L, Marinov D, Lu S (2013) Toddler: detecting performance problems via similar memory-access patterns. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, pp 562–571

55. Holzinger P, Hermann B, Lerch J, Bodden E, Mezini M (2017) Hardening java's access control by abolishing implicit privilege elevation. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE, pp 1027–1040
56. Bodden E (2012) Inter-procedural data-flow analysis with ifds/ide and soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, pp 3–8
57. Chandak A, Lee W, Stamp M (2021) A comparison of word2vec, hmm2vec, and pca2vec for malware classification. In: Malware analysis using artificial intelligence and deep learning, pp 287–320
58. Kale AS, Pandya V, Di Troia F, Stamp M (2022) Malware classification with word2vec, hmm2vec, bert, and elmo. J Comput Virol Hacking Tech 19:1–16
59. Kwon O, Kim D, Lee S-R, Choi J, Lee S (2021) Handling out-of-vocabulary problem in hangeul word embeddings. In: Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, pp 3213–3221
60. Duan Y, Li X, Wang J, Yin H (2020) Deepbindiff: learning program-wide code representations for binary diffing. In: Network and Distributed System Security Symposium
61. Xu Y, Xu Z, Chen B, Song F, Liu Y, Liu T (2020) Patch based vulnerability matching for binary programs. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 376–387
62. Xu K, Li Y, Deng RH, Chen K (2018) Deeprefiner: multi-layer android malware detection system applying deep neural networks. In: 2018 IEEE European Symposium on Security and Privacy (EuroS &P). IEEE, pp 473–487
63. Pennington J, Socher R, Manning CD (2014) Glove: global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp 1532–1543
64. Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. Trans Assoc Comput Linguist 5:135–146
65. Le Q, Mikolov T (2014) Distributed representations of sentences and documents. In: International Conference on Machine Learning. PMLR, pp 1188–1196
66. Zhou J, Cui G, Hu S, Zhang Z, Yang C, Liu Z, Wang L, Li C, Sun M (2020) Graph neural networks: a review of methods and applications. AI Open 1:57–81
67. Cai H, Zheng VW, Chang KC-C (2018) A comprehensive survey of graph embedding: Problems, techniques, and applications. IEEE Trans Knowl Data Eng 30(9):1616–1637
68. Mercaldo F, Santone A (2020) Deep learning for image-based mobile malware detection. J Comput Virol Hacking Tech 16(2):157–171
69. Yuan H, Yu H, Gui S, Ji S (2022) Explainability in graph neural networks: a taxonomic survey. IEEE Trans Pattern Anal Mach Intell 45(5):5782–5799
70. Xie Y, Xu Z, Zhang J, Wang Z, Ji S (2022) Self-supervised learning of graph neural networks: a unified review. IEEE Trans Pattern Anal Mach Intell 45(2):2412–2429

Authors and Affiliations

Pengbin Feng[1] · Li Yang[2] · Di Lu[2] · Ning Xi[1] · Jianfeng Ma[1]

Li Yang
yangli@xidian.edu.cn

Di Lu
dlu@xidian.edu.cn

Ning Xi
nxi@xidian.edu.cn

Jianfeng Ma
jfma@mail.xidian.edu.cn

[1] School of Cyber Engineering, Xidian University, Xi'an 710071, Shaanxi, China

[2] School of Computer Science & Technology, Xidian University, Xi'an 710071, Shaanxi, China