CSGraph2Vec: Distributed Graph-based Representation Learning for Assembly Functions

Wael J. Alhashemi
McGill University
Montreal, Canada
wael.alhashemi@mail.mcgill.ca

Benjamin C. M. Fung
McGill University
Montreal, Canada
ben.fung@mcgill.ca

Adel Abusitta
Polytechnique Montréal
Montreal, Canada
adel.abu-sitta@polymtl.ca

Claude Fachkha
University of Dubai
Dubai, United Arab Emirates
cfachkha@ud.ac.ae

***Abstract*—Software reverse engineering is an essential but time-consuming undertaking in identifying malware, software vulnerabilities, and plagiarism, especially when access to the source code is limited. The extraction of abstract characteristics that represent malware and work as classifier inputs in traditional machine learning approaches requires feature engineering. The calibre of the features that are extracted has a major impact on how well these algorithms perform. In contrast, end-to-end learning solutions do not require hand-designed features and instead attempt to determine if an executable is harmful or not. However, a certain level of preprocessing remains essential in order to present malware content in a manner that the machine learning algorithm can comprehend. Due to the development of machine learning and deep learning, automating the construction of vector embeddings has become more feasible. This research introduces *CSGraph2Vec*, a distributed and automated deep learning approach that produces representations of assembly functions. Using the power of the *Electra* pre-trained language model, as well as *message-passing neural networks*, *CSGraph2Vec* efficiently incorporates control flow and semantic information from assembly code. Our model successfully learns significant features that distinguish benign from malicious functions. Through extensive experimentation and evaluation of the malware classification task, we show that our model performs better than several alternative approaches.**

***Index Terms*—Cybersecurity, representation learning, distributed learning, spark, reverse engineering**

I. Introduction

Machine learning technology has gained traction to detect malware thanks to recent advancements, including the availability of labeled malware datasets for research, the decreasing cost of computational power, and the progress made in machine learning techniques. This approach is attractive because it possesses the ability to comprehend intricate relationships within the input data and make informed decisions without relying on specific signatures.

The extraction of abstract characteristics that represent malware and work as classifier inputs in traditional machine learning approaches requires feature engineering. The calibre of the features that are extracted has a major impact on how well these algorithms perform. In contrast, end-to-end learning solutions do not require hand-designed features and instead attempt to determine if an executable is harmful or not. However, a certain level of preprocessing remains essential in order to present malware content in a manner that the machine learning algorithm can comprehend [1]–[3].

This research is supported by NSERC Discovery Grants (RGPIN-2024-04087) and Canada Research Chairs Program (950-232791).

Reverse engineering proves to be a complex and time-consuming process that requires significant expertise and knowledge in the field [4]. This technique is employed by professionals to gain a comprehensive understanding of software in cyber-physical systems, particularly in situations where access to the source code is not available. As a result, reverse engineering serves as a critical tool in detecting malware and is a fundamental component in modern cybersecurity efforts.

Previous work pertaining to representation learning specifically targeting the assembly code level has been limited [5], [6]. Although these studies have shown promising outcomes, they primarily focus on capturing the semantic aspects of binaries by embedding the relationships among assembly instruction tokens. However, they overlook the significant information embedded within the hierarchical structure of the assembly code. In order to bridge the aforementioned gap, we present an innovative model that incorporates both the semantic and hierarchical structures of the assembly code. Our model aims to offer a more comprehensive representation of assembly functions. Furthermore, to optimize efficiency and scalability, we suggest employing Spark, a distributed computing framework, for generating distributed vector representations of assembly functions. Using Spark, we can enhance the computational performance and scalability of our model.

In this paper, our research mainly revolves around the assembly code level, where we conduct experimental analysis. Our main focus is on exploring a novel approach to representation learning using graph neural networks [7]. Specifically, we delve into the utilization of *message-passing neural networks (MPNN)* [8], which have shown remarkable performance in various tasks. By leveraging MPNN, we aim to generate enhanced representations of assembly code by effectively aggregating messages from neighboring nodes. To ensure that semantically similar instructions are closely embedded, we harness the power of a pre-trained Transformer model called Electra [9]. In particular, our work marks the first utilization of the Electra model to represent assembly instructions. The model's workflow is structured in the following manner: 1) Construction of control flow graphs (CFGs) for assembly functions, 2) Generation of initial embeddings for basic blocks using Electra, 3) Utilization of message-passing neural net-

work for generating comprehensive vector embeddings of the assembly function, and 4) Utilization of these embeddings for malware classification tasks, such as distinguishing between benign and malicious samples. In addition, we evaluate the effectiveness of our model by comparing it with alternative approaches already in use. The crux of our study revolves around addressing the representation learning challenge and devising effective and efficient representations for assembly code. Furthermore, we are focusing on the task of classifying malware. It is worth mentioning that these vector representations hold potential for broader applications, including binary clone detection [10], thus offering promising avenues for further research.

The main contributions of this paper can be summarized as follows.

- We introduce a new method for learning representations of assembly code. What sets our approach apart is its innovative utilization of a hybrid semantic and structural distributed representation learning technique, which harnesses the power of Apache Spark for parallel processing. This combination enables us to effectively capture both the semantic aspects and the structural characteristics of the assembly code, resulting in enhanced representations. Notably, our study marks the first instance of incorporating Apache Spark into assembly code representation learning, making it a pioneering contribution in the field.
- We reduce the execution time for generating vector embeddings for assembly functions by employing multiple Spark workers to execute the pipeline concurrently.
- By leveraging datasets that are publicly accessible, we demonstrate the effectiveness of our model, *CS-Graph2Vec*, through rigorous experimentation. In our approach, we leverage a language model to learn the embeddings of assembly code, which allows us to capture their semantic properties. This is achieved by considering the relationships between tokens within the language model. Additionally, we present the benefits of utilizing CFGs with MPNN, since it helps to produce improved vector embeddings. Taking the aforementioned components, along with the utilization of Apache Spark for parallel processing, our model surpasses the performance of various other models in the task of assembly code-level malware classification. The combination of semantic understanding, structural analysis, and parallel processing provides our model with a competitive advantage and contributes to its superior performance.

II. Related Work

A. Malware Classification

As machine learning continues to progress, it has become crucial to evaluate its potential within the realm of cybersecurity [11]. *Sethi et al.* [12] introduced an innovative framework in the field of machine learning, aiming to detect and categorize malware. The authors first use *Cuckoo Sandbox*[1] to retrieve the static and dynamic analysis report of an executable file. The analysis report then serves the purpose of extracting pertinent characteristics, which are subsequently employed to identify and classify the most important features for the detection and categorization of malicious software. In another work, *Gulmez et al.* [13] focus on using graph-based techniques to detect malware based on opcode sequences. They disassemble the executable files to obtain the opcode sequences. Then, the opcode sequences are transformed into weighted and directed graphs, where each node represents an opcode, and the transitions between opcodes are represented as edges. In order to capture important features and differentiate between malware and benign files, sub-graphs are created by eliminating connections between distinct opcodes. Finally, The sub-graphs are used to generate a histogram file containing the node degrees. The histogram depicts the degree distribution of each opcode in the graph, where the degree refers to the number of edges connected to a node. The generated histogram files of the degrees of the nodes are used as input features for classification.

B. Assembly Code Representation

Researchers have recognized the shared characteristics between assembly code and natural text, leading them to apply natural language processing (NLP) models to analyze these programs [14], [15]. To embed assembly instructions, *Redmond et al.* [16] used the *Word2Vec* [17] model. Their work examined NLP methods and modified them for cross-architectural binary code analysis, including multilingual word embedding [18]. *Instruction2Vec*, a framework created expressly to model assembly code, was introduced by *Lee et al.* [6]. It is an improved *Word2Vec* model that takes into account the syntax of the assembly language. The authors created a lookup table with each instruction represented as a fixed-dimension vector made up of an opcode and two operands using *Word2Vec*. Then, they used *Text-CNN* [19] to find software flaws. Incorporating extensive semantic information between tokens, *Ding et al.* [5] suggested an approach to representing assembly code based on the *PV-DM* model [20]. However, it should be noted that the main emphasis of these approaches lies in the semantics of the code rather than explicitly taking into account the real-time execution sequence.

Researchers have increasingly employed graph embedding networks to acquire representations of assembly functions. In their work, *Feng et al.* [21] developed *Genius*, a scalable graph-based bug search model designed for firmware images. Their approach involves converting the control flow graphs (CFGs) of executable functions into numeric feature vectors at a higher level, followed by using cutting-edge hashing techniques for efficient searching. Moreover, *Yan et al.* [22] suggested a malware classification model that takes advantage of the capabilities of the *deep graph convolutional neural network (DGCNN)* to embed structural information extracted from a CFG. Initially, they converted the CFG into an at-

[1]https://cuckoosandbox.org/

tributed control flow graph (ACFG), where each vertex was associated with specific attributes at the block level. Subsequently, DGCNN is applied to the graph data, transforming it to facilitate classification tasks. In their research, *Xu et al.* [23] introduced a neural-based graph embedding model called *Gemini* for detecting similarity in cross-platform binary code. Their model used the control flow graph and associated attributes with each node to represent the code. To transform the graph into an embedding suitable for detecting similarities, they employed *Structure2Vec* [24] as a graph embedding network. By integrating the graph embedding network into a Siamese network [25], they effectively captured the objective of bringing similar graph embeddings closer to each other. In contrast, our work explores the usage of *Electra* [9] to generate the initial node embedding. Subsequently, a message-passing neural network is employed for further analysis.

III. Problem Definition

This section defines the research problem as well as the notation used. Our model takes as input an assembly function, denoted as f. We initiate the process by parsing f into a control flow graph (CFG). The CFG is produced by establishing connections, which are the edges, between the basic blocks that invoke each other. Each basic block, represented as v, is characterized by a vector embedding, x_v. To generate x_v, we map the set of assembly instructions in a basic block, b_v, to their corresponding vector embedding x_v using an embedding function $f_v(b_v) = x_v$.

Graph $G = (V, E)$ serves as the message-passing neural network's (MPNN) input, where V represents the set of initial basic block representation in the CFG, and E denotes the set of edges connecting the basic blocks. After obtaining the initial block embedding (x_v), a graph neural network, denoted f_g is used to improve the quality of block embeddings, resulting in enhanced representations, x'_v. Subsequently, we derive the graph embedding, θ_f, for the CFG, which serves as the input for the malware classification task. The classification model assigns a label, $\hat{y}$, to the assembly function, with $\hat{y}$ taking values from the set $\{0, 1\}$ to indicate whether the function is classified as benign or malicious: $f_g(G) = \hat{y}$.

The research problem of our malware classification task can be defined as follows, taking into account the depicted workflow of our *CSGraph2Vec* model in Figure 1.

Definition 1 (Malware Classification): Consider a set of assembly functions F, accompanied by their corresponding labels Y, indicating whether each assembly function is classified as malicious or benign. Let f be an unknown assembly function such that $f \notin F$. The malware classification problem involves constructing a classifier M, which is based on F and Y. This model M is designed to enable the identification of whether the assembly function, f, is malicious ($\hat{y}$ has the value 1) or benign ($\hat{y}$ has the value 0). ■

IV. Graph-based Representation of Assembly Code

We divide our pipeline into three steps. Step 1 is to generate the control flow graph. Step 2 is to generate the initial node embedding and enhance them. Step 3 is to store the final graph embedded in a local disk. Before explaining our approach, we first discuss the preliminaries needed to understand our model.

A. Preliminaries

Assembly functions can be effectively represented using control flow graphs (CFGs), which visually depict the various paths through which a program can be executed [26]. A CFG provides a graphical representation of the control flow within a program, illustrating the decision points, loops, and branches that influence the execution flow. By mapping the control flow of a program into a graphical format, a CFG facilitates understanding of the program's structure and the possible execution paths it can take. In a CFG, the basic blocks (v) serve as nodes and represent groups of sequential instructions without any branching or jumping. These basic blocks are connected by the CFG's edges, which indicate the direction in which execution might proceed. These edges capture the transitions between basic blocks that can occur during program execution. Fig. 2 shows how the CFG of an assembly function is generated. This representation is helpful for malware classification because it helps us determine the behaviour of program execution.

To obtain vector representations of assembly instructions, we utilize message-passing neural network (MPNN) [8]. The MPNN is a valuable choice because it accumulates messages from neighboring nodes, incorporating hidden layer representations, and combines them to improve the node embeddings. This allows us to capture the intricate relationships and dependencies among the assembly instructions within the CFG. Moreover, we must encode each basic block b_v's instruction sequence as embeddings in order to make it easier to integrate the CFG with the MPNN. Motivated by the accomplishments of pre-trained language models in problems involving natural language processing [27], a pre-trained language model f_v is used to obtain the initial block embeddings x_v. These embeddings effectively capture the semantic aspects of each assembly instruction by considering the interrelationships between different tokens and preserving crucial information such as the grammar and the associations between the operation and its operands. Our primary objective is to preserve the assembly instructions' semantic information throughout the representation learning process. By incorporating the MPNN and harnessing the capabilities of pre-trained language models, we can generate comprehensive and meaningful vector representations for assembly code.

B. Apache Spark Framework

In recent times, there has been a noticeable increase in the expansion of data both in terms of volume and variety. This, in turn, has spurred the development of sophisticated big-data technologies aimed at efficiently managing and processing these data. Among the prominent distributed systems for handling big data, one that has gained significant attention is *Apache Spark*[2]. *Apache Spark* [28], an in-memory cluster com-

[2]https://spark.apache.org/

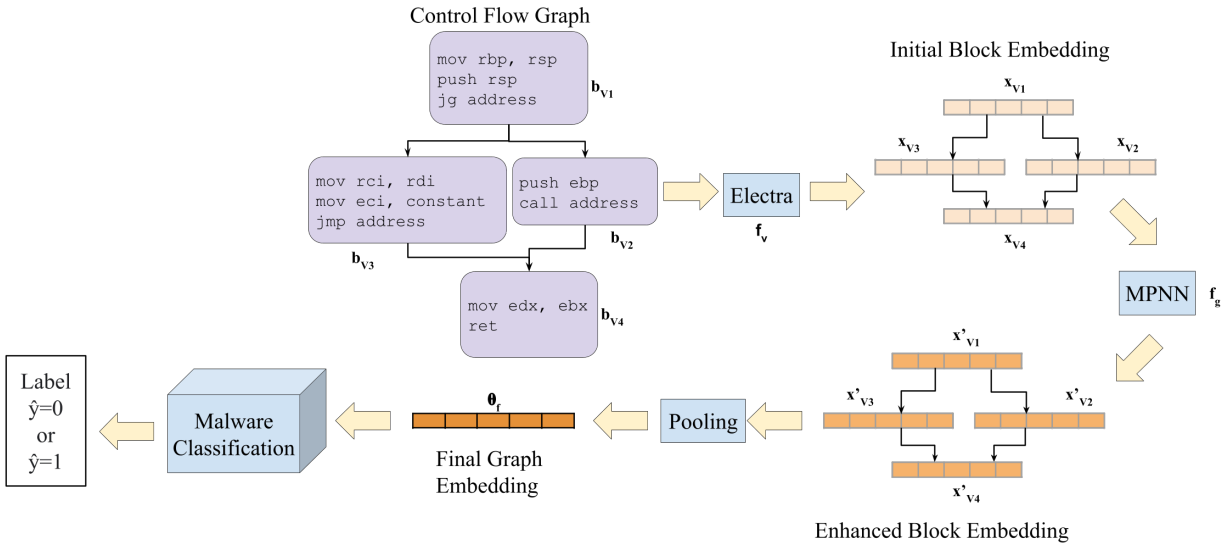

Fig. 1. Workflow of CSGraph2Vec model

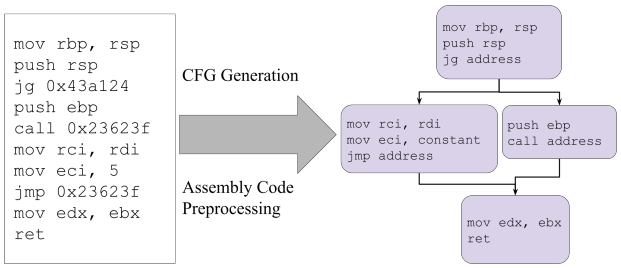

Fig. 2. Control flow graph generation and preprocessing.

puting system, has emerged as a result of extensive research and widespread adoption. Spark employs a data abstraction known as Resilient Distributed Datasets (RDDs) [29], which enables the system to handle large datasets by dividing them into smaller partitions across multiple machines. This approach facilitates rapid data processing while ensuring fault tolerance. Furthermore, in the event of a partition loss, the RDD can be reconstructed as the system retains knowledge of its derivation from parent RDDs.

RDDs can be generated through two different approaches: by utilizing data stored in stable storage or by employing other RDDs. These RDDs support two main categories of operations. The first category is known as *transformations*, which involve creating a fresh dataset from a pre-existing dataset. Examples of transformation operations include *map* and *filter*. The second category is referred to as *actions*, which execute computations on the dataset and return a value to the driver program. Examples of action operations include *count* and *collect*. In *Apache Spark*, transformations are designed to be lazy, which means that the actual results are not computed immediately. Instead, they store information about the applied transformation in a base dataset, such as a file. Only when an action requires the delivery of a result to the driver program, the transformations are executed. Additionally, *Spark* offers a *persist* method, allowing users to retain an RDD in the cluster for faster accessibility during subsequent queries. By utilizing *Spark*, we leverage its ability to parallelize pipeline execution by evenly distributing the data set among multiple workers. This approach enables efficient and concurrent processing of the data.

C. Message-Passing Neural Network

Message-passing neural network (*MPNN*) [8] is a framework that can operate on graph G with node and edge features, x_v and e_{vw}, respectively. It is based on the notion of enhancing node embeddings by combining the node embeddings from neighboring nodes. Figure 3 illustrates the overview of message aggregation. The message passing phase, which is used to update the node embeddings, and the readout phase make up MPNN's architecture. The vertex update functions and message sending phases are defined for the message passing phase, which lasts for T time steps. M_t and U_t respectively. During the message passing phase, at every time step t, every node possesses a hidden state, h_v^t that gets updated according to the messages m_v^{t+1} received from neighboring nodes.

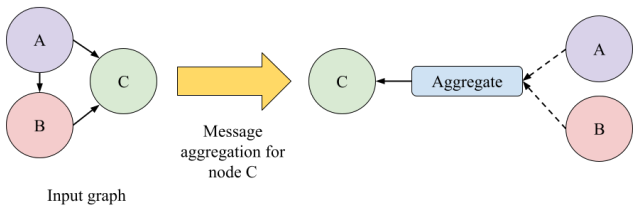

Fig. 3. Illustration of message aggregation in message-passing neural network.

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \quad (1)$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}) \quad (2)$$

A feature vector pertaining to the complete graph is generated in the final stage, also referred to as the readout phase, using a predetermined readout function abbreviated R.

$$\hat{y} = R(\{h_v^T | v \in G\}) \quad (3)$$

It is vital to remember that the learnt, differentiable functions (M_t, U_t, and R) for message update, node update, and readout are all used. This flexibility allows for the incorporation of various message and update functions within the framework, providing robustness and adaptability to different scenarios.

D. Electra

Electra [9] is a method employed for self-supervised language representation learning, allowing the pre-training of transformer text encoders [30]. These encoders can be subsequently fine-tuned on particular tasks using limited computational requirements. The primary objective of this approach is to distinguish between input tokens classified as either "real" or "fake," which are generated by an additional neural network, akin to the discriminator component in *generative adversarial nets (GAN)* [31].

The *Electra* methodology involves the training of two neural networks: a *generator* and a *discriminator*. The purpose of the *generator* is to perform masked language modelling (MLM), where a portion of tokens within an input are masked, and the goal is to predict the original vocabulary ID of the masked token. Typically, around 15% of the tokens are selected for masking, and the generator learns to accurately predict their original identities. Conversely, the role of the *discriminator* is to tell apart between tokens in the original data and tokens that have been substituted by the samples generated by the *generator*.

We utilize *Electra* due to its comparable performance with *RoBERTa* as well as *XLNet*, while requiring less than one-fourth of their computational resources. Moreover, *Electra* outperforms these models when the same amount of compute is employed.

E. Proposed Method: CSGraph2Vec

By integrating the aforementioned concepts and knowledge, we develop our model, called *CSGraph2Vec*, which aims to convert assembly functions (f) into graph embeddings (θ_f). These embeddings can be utilized for malware classification by employing a graph neural network on CFGs and producing corresponding embeddings. *CSGraph2Vec* effectively captures both the semantic and structural elements of assembly code by utilizing a graph structure, specifically CFG. Each basic block within the CFG is represented by a vector representation derived from a pre-trained language model. These elements are then combined using a graph neural network that aggregates messages from neighboring nodes, ultimately producing an enhanced embedding for the graph. Lastly, these enhanced vectors are utilized for the purpose of malware classification.

To begin with, we generate the CFG of the assembly function by establishing connections between the basic blocks that call one another. Each assembly instruction is treated as a word while training our language model, and the complete basic block including assembly instructions is treated as a sentence. For learning the initial block embeddings, we employ *Electra* and calculate the basic block's overall instruction embedding average. We choose the mean approach since each basic block can contain a varying number of assembly instructions, and the number of assembly code instructions that can be contained in a basic block is not restricted in any way.

In order to train the *Electra* model to generate the initial node embeddings, we built a text collection containing 3,890,170 x86 assembly instructions with optimization levels varying from O0 to O3. We train *Electra* on the *Masked Language Modelling* (MLM) task, and we generate the block embedding by taking the average of the instruction embeddings in that block. Training our language model took 39 hours, 41 minutes and 19 seconds with an Intel i9-9980XE 18 core 3.00GHz CPU, 128GB RAM, and two NVIDIA GeForce RTX 2080 Ti graphic cards.

To create the relationship between the embeddings of each block (nodes) and the blocks themselves (edges), we utilize them as input for our MPNN. The MPNN framework efficiently gathers data from nearby blocks and uses it to produce improved block embeddings. One of the advantages that our architecture has is scalability in the sense that any deep neural network model can be used in place of our MPNN as long as the deep neural network takes as input a graph. To generate the final embedding of the function, we employ a pooling layer. During our experimentation, we evaluate $average$ as well as add pooling layers, and find that the *average* readout layer yields superior results. Subsequently, we employ the embeddings to perform malware classification as part of the downstream task

We utilize *Spark* framework for parallel processing. Each deployed worker will be given an equal share of the dataset to execute steps 1 through 4. The concurrent execution of the pipeline from each worker allows for a faster runtime and more data handling.

V. Experiments

The objective of the experiment is to evaluate the efficiency and scalability of the generation of embedding functions in the assembly function. However, it is also important to ensure that the quality of the learned embedding is high. To assess the quality of the embeddings, we compare *CSGraph2Vec* with various methods in the task of malware classification. *PyTorch* [32] and *PyTorch Geometric* [33] are used for our model's implementation, and our experiments are run on an Intel Xeon E5-1650 v4 6 core 3.60GHz CPU, and 32GB RAM.

A. Dataset

We use a subset of the dataset used by *Asm2Vec* [5] as benign software, consisting of four widely used utility and numerical calculation libraries. As for malicious software, we collect 10 different classes of malware. The entire dataset has a total of 9,371 assembly functions. Our analysis involves benchmarking the performance of our model in the software. Furthermore, we observe from Figures 4 and 5 that each software has a different number of nodes and edges. More detailed statistics of the dataset are listed in Tables I and II.

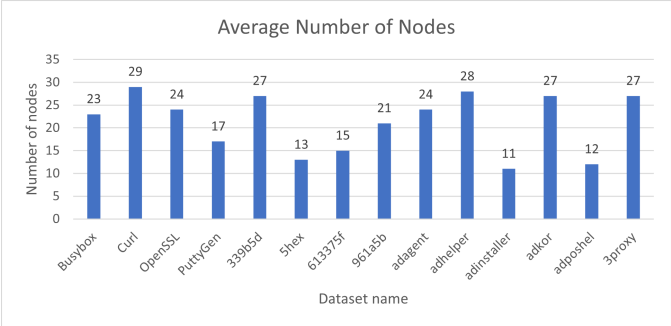

Fig. 4. The average number of nodes in the dataset.

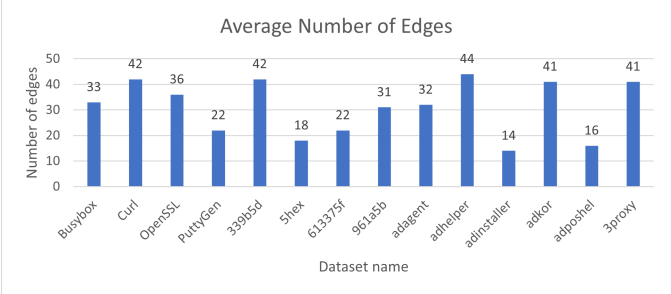

Fig. 5. The average number of edges in the dataset.

TABLE I
STATISTICS OF THE BENIGN DATASET. A TOTAL OF 5,195 FUNCTIONS.

Library	Number of Functions
Busybox	1389
Curl	468
OpenSSL	3124
PuttyGen	214

In the preprocessing step, we begin by converting all instructions to lowercase. To avoid the need to learn distinct representations for every unique hexadecimal address and constant value, we opt to replace hexadecimal addresses with the token `address` and numerical constants with the token `constant`. This substitution enhances the quality of the vector embeddings.

B. Experimental Settings

For the efficiency evaluation, we execute the entire pipeline but with varying the number of *Spark* workers being employed. Efficiency experiments are conducted using {1, 2, 4, 6, 8} workers. For each of these experiments, we utilize the whole 9,371 functions. Further, we split the dataset equally between the workers. However, for the 1 worker case, it will be given the entire dataset as that particular experiment is considered a nonparallel experiment.

TABLE II
STATISTICS OF THE MALICIOUS DATASET. A TOTAL OF 4,176 FUNCTIONS.

Library	Number of Functions
339b5d	56
5hex	1924
613375f	72
961a5b	67
adagent	1076
adhelper	57
adinstaller	197
adkor	58
adposhel	269
3proxy	400

As for evaluating the scalability, we set the number of *Spark* workers to 4. For each experiment, we execute the pipeline while varying the dataset's size. We conduct the scalability experiments using {3,000, 6,000, 9,371} functions. Again, the dataset is split equally between the 4 *Spark* workers for each experiment.

For evaluating the accuracy of our model, we perform malware classification experiments. To showcase the effectiveness of our semantic and structural components, we conduct a comparative analysis of the performance of our model against various methods. In the first method, we adopt a methodology that relies on manually crafting features to generate our basic block embeddings [23]. We craft each basic block's features by counting the number of instructions, transfer instructions, function calls, arithmetic instructions, logical operations, constants, and strings in the block. However, this representation would cause us to miss the crucial information expressed in the assembly instruction. In a separate approach described in [6], the authors utilize *Instruction2Vec* to embed the assembly instructions, and then employ *TextCNN* as a classification method to classify the samples. We also experiment with using *Word2Vec* [17] for node representation and GCN for the message-passing element. Ultimately, we assess our model *CSGraph2Vec*, using the following variations: 1) Using *manually crafted features with GCN* ***(MC-GCN)***. 2) Employing *Instruction2Vec along with TextCNN* ***(I2V-TCNN)***. 3) Utilizing *Word2Vec along with GCN* ***(W2V-GCN)***.

Furthermore, we conduct a performance comparison between our model and *Asm2Vec* [5], *CACompare* [34], and *Genius* [21] with regard to their accuracy on the task of clone search. Given the functions' embedding, we calculate the cosine similarity between pairs of functions. We evaluate the performance of our model using precision at 1 (P@1) and evaluate it in comparison with other methods. For clone search, We specifically focus on assembly functions that consist of a minimum of five basic blocks so that the functions have

semantically more meaning and graph structure. The dataset used for this experiment is the benign software mentioned in V-A. We conduct experiments on the compiler optimization level O0 as it is the default optimization level as well as no optimization is performed.

C. Results

Figures 6 show that using more than one *Apache Spark* worker in *CSGraph2Vec* reduces the time to generate the graph embedding of assembly functions. This is due to the fact that more workers are executing the pipeline in parallel as the dataset is split equally amongst them. We also observe that using 4 workers is the optimal number. Setting it to more than 4 would result in increasing the running time slightly and then plateaus. We believe that this is due to the overhead of the workers. Moreover, given the scale of our dataset, using more than 4 workers is overkill. Hence, there is a fine line between the size of the dataset and the number of workers. When comparing *CSGraph2Vec*'s performance with *Asm2Vec*'s, Figure 7 shows that *Asm2Vec* is faster than our model. This is due to *CSGraph2Vec*'s MPNN being computationally expensive as messages are being exchanged between nodes. Hence, the larger the graph and edges between nodes, the longer it will take to exchange messages.

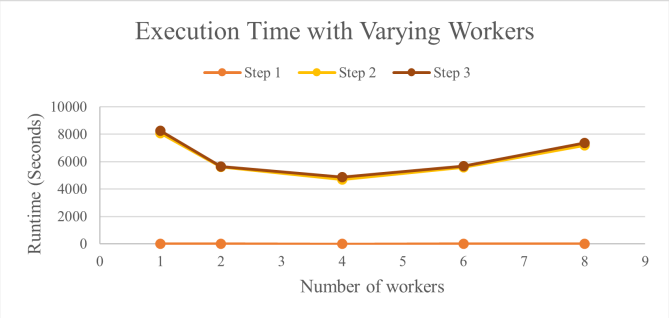

Fig. 6. Efficiency of *CSGraph2Vec* on 9,371 functions. It depicts the runtime to generate the assembly functions embeddings.

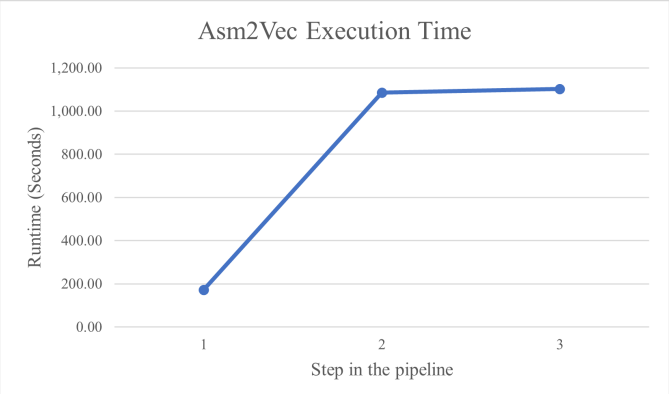

Fig. 7. Efficiency of *Asm2vec* with only 1 worker on 9,371 functions.

Regarding scalability, Figure 8 that as the number of functions increases, the time it takes to generate the embeddings

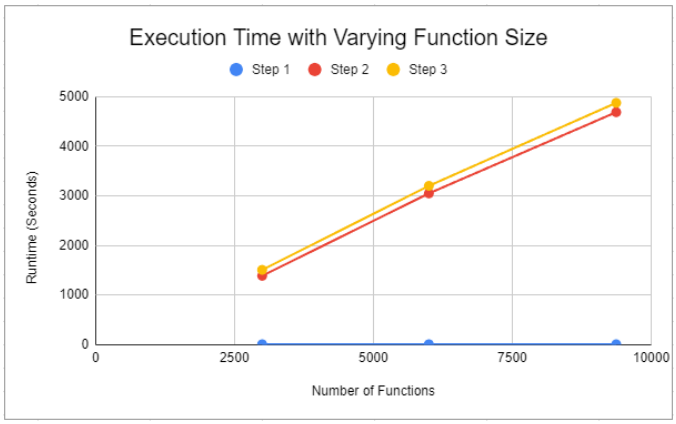

Fig. 8. Scalability of *CSGraph2Vec* with 4 workers.

increases almost linearly. This is due to Spark splitting the dataset equally to each worker so that no worker is being overworked.

For malware classification, Table III shows that *CSGraph2Vec* performs better than the other models. We observe that the manually crafted features do not adequately capture the essence of the assembly instructions. Therefore, in order to ensure high-quality representation, it is imperative to integrate meaningful contextual representations of the assembly instructions.

TABLE III
MALWARE CLASSIFICATION RESULTS.

Malware Classification Results					
Model	F1-Score	Accuracy	Recall	Precision	AUC-ROC
MC-GCN	63.85	68.92	62.03	66.24	68.29
W2V-GCN	92.62	93.45	92.16	93.22	93.32
I2V-TCNN	99.57	99.62	99.15	99.64	99.58
CSGraph2Vec*	**99.81**	**99.83**	**99.81**	**99.81**	**99.83**

For clone search, Table IV shows that our model is able to attain comparable results to other methods when searching using same compiler optimization level. This is due to the target function having identical graph structure as the ground truth, thus precision at 1 will always return the same function as the target function.

TABLE IV
CLONE SEARCH RESULTS OF COMPILER OPTIMIZATION LEVEL O0 USING THE PRECISION AT POSITION 1 (PRECISION@1) METRIC.

O0 Compiler Optimization Clone Search					
Baseline	Busybox	Curl	OpenSSL	PuttyGen	Avg.
CACompare [34]	**1**	∅	**1**	**1**	**1**
Genius [21]	**1**	∅	**1**	**1**	**1**
Asm2Vec [5]	**1**	**1**	**1**	**1**	**1**
CSGraph2Vec*	**1**	**1**	**1**	**1**	**1**

VI. CONCLUSION

In this paper, we propose a distributed representation learning method for malware classification named *CSGraph2Vec*.

To create efficient graph embeddings, we use message-passing neural networks in conjunction with the Electra model. We evaluate the performance of *CSGraph2Vec* by conducting extensive experiments on the malware classification task. We also evaluate the efficiency of generating the graph embeddings by employing *Apache Spark* for parallel processing. We demonstrate that *CSGraph2Vec* is able to efficiently generate the embeddings with more *Apache Spark* workers and outperforms various methods in the malware classification task considering the semantic and structural hierarchy of the assembly code. The message-passing neural network provides a more thorough representation by collecting messages from all neighbors, while the control flow graph aids in identifying the malicious execution routes.

REFERENCES

[1] A. Abusitta, M. Q. Li, and B. C. M. Fung, "Malware classification and composition analysis: A survey of recent developments," *Journal of Information Security and Applications (JISA)*, vol. 59, no. 102828, pp. 1–17, June 2021.

[2] M. Saqib, B. C. M. Fung, P. Charland, and A. Walenstein, "GAGE: Genetic algorithm-based graph explainer for malware analysis," in *Proc. of the 40th IEEE International Conference on Data Engineering (ICDE)*. Utrecht, Netherlands: IEEE Computer Society, May 2024, pp. 2258–2270.

[3] T. Bilot, N. El Madhoun, K. Al Agha, and A. Zouaoui, "A survey on malware detection with graph representation learning," *ACM Computing Surveys*, vol. 56, pp. 1–36, June 2024.

[4] E. Eilam, *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[5] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 472–489, 2019.

[6] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park, "Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn," *Applied Sciences*, vol. 9, no. 19, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/19/4086

[7] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[8] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," 2017. [Online]. Available: https://arxiv.org/abs/1704.01212

[9] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," 2020. [Online]. Available: https://arxiv.org/abs/2003.10555

[10] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[11] M. Q. Li, B. C. M. Fung, and P. Charland, "Dyadvdefender: An instance-based online machine learning model for perturbation-trial-based black-box adversarial defense," *Information Sciences (INS)*, vol. 601, pp. 357–373, July 2022.

[12] K. Sethi, R. Kumar, L. Sethi, P. Bera, and P. K. Patra, "A novel machine learning based malware detection and classification framework," in *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, 2019, pp. 1–4.

[13] S. Gülmez and I. Sogukpinar, "Graph-based malware detection using opcode sequences," in *2021 9th International Symposium on Digital Forensics and Security (ISDFS)*, 2021, pp. 1–5.

[14] M. Q. Li, B. C. M. Fung, P. Charland, and S. H. H. Ding, "I-MAD: Interpretable malware detector using Galaxy Transformers," *Computers Security (COSE)*, vol. 108, no. 102371, pp. 1–15, September 2021.

[15] L. Li, S. H. H. Ding, Y. Tian, B. C. M. Fung, P. Charland, W. Ou, L. Song, and C. Chen, "VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution," *ACM Transactions on Privacy and Security (TOPS)*, vol. 26, no. 28, pp. 1–25, August 2023.

[16] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," *CoRR*, vol. abs/1812.09652, 2018. [Online]. Available: http://arxiv.org/abs/1812.09652

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[18] T. Luong, H. Pham, and C. D. Manning, "Bilingual word representations with monolingual quality in mind," in *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing*. Denver, Colorado: Association for Computational Linguistics, Jun. 2015, pp. 151–159. [Online]. Available: https://aclanthology.org/W15-1521

[19] Y. Kim, "Convolutional neural networks for sentence classification," *CoRR*, vol. abs/1408.5882, 2014. [Online]. Available: http://arxiv.org/abs/1408.5882

[20] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014. [Online]. Available: http://arxiv.org/abs/1405.4053

[21] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[22] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 52–63.

[23] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," *CoRR*, vol. abs/1708.06525, 2017. [Online]. Available: http://arxiv.org/abs/1708.06525

[24] L. Song, "Structure2vec: Deep learning for security analytics over graphs." Atlanta, GA: USENIX Association, May 2018.

[25] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in *Advances in Neural Information Processing Systems*, J. Cowan, G. Tesauro, and J. Alspector, Eds., vol. 6. Morgan-Kaufmann, 1993.

[26] K. D. Cooper, T. J. Harvey, and T. Waterman, "Building a control-flow graph from scheduled assembly code," Tech. Rep., 2002.

[27] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *CoRR*, vol. abs/2003.08271, 2020. [Online]. Available: https://arxiv.org/abs/2003.08271

[28] M. A. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud*, 2010.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," 04 2012, pp. 2–2.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[31] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014. [Online]. Available: https://arxiv.org/abs/1406.2661

[32] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS 2017 Workshop on Autodiff*, 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ

[33] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *CoRR*, vol. abs/1903.02428, 2019. [Online]. Available: http://arxiv.org/abs/1903.02428

[34] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 88–98.