Approach for the Optimization of Machine Learning Models for Calculating Binary Function Similarity

Suguru Horimoto[1,2], Keane Lucas[3], and Lujo Bauer[3]

[1] National Police Agency, Tokyo, Japan
[2] Visiting Researcher at CyLab Security and Privacy Institute of Carnegie Mellon University, USA shorimot@andrew.cmu.edu
[3] Carnegie Mellon University, USA {kjlucas,lbauer}@andrew.cmu.edu

Abstract. Binary function similarity comparison is essential in a variety of security fields, such as software vulnerability detection and malware analysis, because it enables engineers to accelerate otherwise time-consuming tasks. While various approaches for binary function similarity comparison have been proposed, in an experiment of previous work to fairly evaluate existing methods, a method combining graph neural network (GNN) and bag-of-words (BoW) exhibited the highest performance. In this method, each basic block (BB) in a function is embedded into a vector by BoW. As a result, the function vector is derived from sparse vectors. In this paper, we propose a method combining a GNN with fastText, instead of BoW. Furthermore, in order to optimize machine learning models for calculating binary function similarity, we apply early stopping based on mean reciprocal rank (MRR) to our machine learning training. Our method outperformed the previous method combining GNN and BoW by up to 2% in AUC, up to 9% in Recall@1 and up to 7% in MRR10 in a certain case. Additionally, through a function search case study in malware analysis, our method has been found to be applicable for finding distinctive functions present in LockBit Ransomware.

Keywords: Malware analysis · Graph learning · Similarity

1 Introduction

Binary function similarity comparison is essential in a variety of security fields, such as software vulnerability detection and malware analysis. For example, in software vulnerability detection, the similarity comparison helps engineers quickly find the same vulnerability in different binaries [7,18]. However, binaries compiled with different compilers and to different architectures make binary function similarity comparison very challenging. For instance, each function in a binary consists of a control flow graph (CFG) and basic blocks (BBs), where each BB consists of assembly instructions. When binaries are compiled from the same source code with different compilers, the structure of a CFG and assembly instructions of BBs are drastically different from each of the binaries because

the structure of a CFG and assembly instructions of BBs in the binaries rely on their respective compiler [9,13].

To tackle this issue, many researchers in security fields have proposed methods to compare binary function similarity with machine learning [13,18,25]. While numerous methods for calculating function similarity have been proposed, Marcelli et al. [17] conducted an evaluation to perform a fair comparison of these methods. Their evaluation found that Graph Neural Network (GNN) and Graph Matching Network (GMN) models [13] outperformed other methods in terms of binary function similarity comparison. In the GNN and GMN models Marcelli et al. [17] prepared, however, function vectors are generated using CFGs and Bag-of-Words (BoW)-based BB vectors. In the BoW approach, a BB vector is generated from the counts of opcode occurrences within the top 200 opcodes selected by Marcelli et al. [17]. As a result, the function vector is derived from sparse vectors. Besides, the effectiveness of the GNN and GMN models in malware analysis remains unclear because the search performance of the GNN and GMN models in malware analysis has not been evaluated by Marcelli et al. [17].

In this paper, we propose a new method for binary function similarity comparison by using *fastText* [4] (Sect. 3.1). fastText is an open-source natural language processing (NLP) tool developed by Facebook, well-suited for various NLP tasks, including word embeddings generation. One of the features of fastText is to split a word into subwords, which are parts of letters within a word, and learn embeddings for these subwords. This approach enhances the handling of out-of-vocabulary words and language variations, allowing for the consideration of fine-grained language details. In our research, we obtain embeddings of opcodes using fastText and use these embeddings to generate BB vectors. Even across different architectures, there are multiple opcodes that share partial common subwords. For instance, opcodes for copying data, such as "mov" or "movzx", contain the partially common subword "mov". Therefore, using fastText to generate embeddings of opcodes appears to intuitively be a more effective approach.

Furthermore, we apply early stopping [20] based on Mean Reciprocal Rank (MRR) in training. Early stopping is a methodology used to halt the learning process when evaluation metrics no longer exhibit continuous improvement during the training. MRR is an evaluation metric for ranking accuracy and is used for evaluating binary function comparison methods [17,27]. Early stopping based on MRR enables to enhance the search performance of our method (Sect. 3.3).

In the evaluation of search performance in malware analysis (Sect. 4.7), we conducted a search for the functions of LockBit Ransomware [5] obtained from Vx Underground [1] using our method. Our method has been found to be applicable for finding distinctive functions present in LockBit Ransomware.

In a nutshell, the contributions of this paper are as follows:

- We improve the performance of GNN and GMN models by leveraging fastText as a method of embedding BBs (Sect. 3.1). We published the source code for training the models, some trained multi-architecture models and the testing dataset for the multi-architecture models.[1]

[1] https://github.com/sgr-ht/mam-for-cbfs

– We evaluate the effectiveness of early stopping based on MRR in training to enhance the search performance of the GNN and GMN models (Sect. 4).
– We evaluate a method combining the GNN model and fastText for its effectiveness in calculating the similarity of binary functions in malware analysis (Sect. 4.7).

2 Related Work

Creating *embeddings* is a popular method for binary function similarity comparison. The embeddings refer to vectors in a low-dimensional space where semantically similar inputs are associated with points that are close to each other, irrespective of the differences in their original representations [17]. Two methods commonly employed for creating embeddings are code embeddings and graph embeddings [17]. Code embeddings is an approach to create embeddings using NLP by considering assembly code as text. Graph embeddings is another approach to create embeddings using machine-learning models which compute embeddings using a CFG and features in each function.

2.1 Code Embeddings

SAFE [18] can convert a function into a vector using a self-attentive network [14], which comes from the seq2seq model, a commonly used NLP structure. In SAFE, assembly instructions in a function are embedded into vectors and then each vector of assembly instruction in a function is fed into the self-attentive network to embed a function vector.

2.2 Graph Embeddings

Gemini The CFG and BBs of a function are first transformed into an annotated CFG, which is a graph containing manually selected features such as the number of assembly instructions and the number of string constants in Gemini [25]. After that, Gemini creates a function vector using structure2vec [6].

GNN and GMN Models Li et al. [13] proposed GNN and GMN models, which can create a function vector from the CFG and BBs of a function and calculate the similarity between pairs of functions. GNN and GMN models achieve the best results in the experiments of prior work [17], which performs a comparison among existing approaches that are publicly available, such as Gemini and SAFE.

3 Proposed Method

Fig. 1 gives an overview of the process of converting a function into a vector in order to compare binary function similarity. The process that we use is based on Li et al. [13]. In this process, using fastText [4] as a BB converter and the

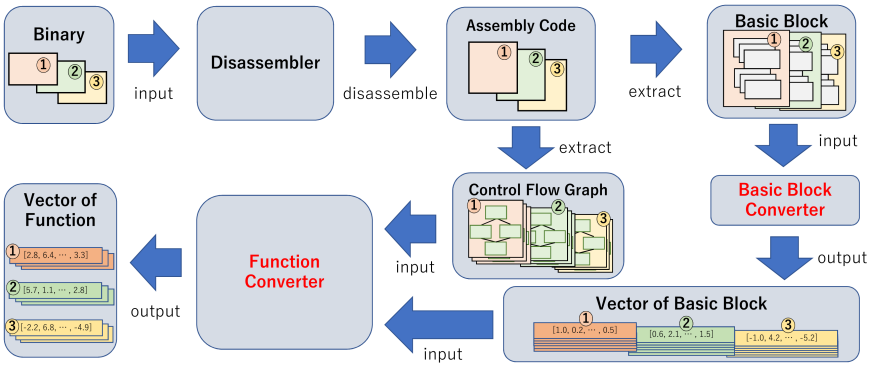

Fig. 1. The overview of the process of converting functions into vectors.

training of a function converter with early stopping based on MRR are new. We first input binaries as a dataset to be disassembled. We then extract BBs and CFGs from the assembly code. After that, the BBs are fed into the BB converter, which converts a BB into a vector. Finally, the BB vectors and the CFGs are fed into the function converter, which embeds a function vector. Each function vector has the embedded meaning of the BBs and the CFG in each function. By repeating these processes to each function of each binary, it enables us to transform all functions of a dataset into vectors.

3.1 BB Converter

As the BB converter we use fastText, which is an NLP tool developed by Facebook used to generate word embeddings. Other work has shown that fastText outperforms word2vec [19], which is an NLP tool to generate word embeddings [4]. Our results presented here suggest that using fastText for vectorizing BBs is a better approach for calculating binary function similarity, particularly because of fastText's utilization of opcode embeddings to vectorize BBs. In terms of similar improvements in other applications, other work has shown that fastText can also be beneficial for vectorizing assembly instructions, which can then be used to vectorize a function [2,24]. Other work has also shown that fastText can lead to improvements in type inference [11].

In our research, we obtain embeddings of opcodes using fastText and use these embeddings to generate BB vectors. The advantage of fastText is to transform unseen opcodes not included in a training dataset into vectors. If there is a BB containing only unseen opcodes, i.e., opcodes not included in the training dataset for BoW and fastText, then BoW would be incapable of extracting any features. On the other hand, fastText is likely to extract at least some features owing to the benefits of subword embeddings. For example, in one of our test datasets, there is a BB containing only one "mov.s", which is an opcode of MIPS (e.g., mov.s dest, src), and "mov.s" is not included in the training dataset. BoW

cannot extract any features of the BB, but fastText can extract a feature thanks to the subword embedding of "mov". In examining our dataset, we found at least 60 other BBs where fastText has this advantage for "mov.s", and many other BBs for other unseen opcodes.

The benefit of subword embedding leads to another advantage of fastText, which is to transform BBs with similar meanings into vectors that are spatially close to each other. In BoW, BBs with similar meanings do not necessarily become vectors that are spatially close. This is because BoW is based on the frequency of opcode occurrence, and does not take into account the semantic aspects of opcodes. On the other hand, in fastText, opcodes are vectorized considering the context information of the surrounding opcodes, and semantically similar opcodes tend to become more spatially close vectors. For example, comparing the similarity between a BB consisting of only "mov" and a BB consisting of only "movzx" using cosine similarity, fastText can capture semantics of the BBs better compared to BoW. "mov" and "movzx" are opcodes related to data movement. In our dataset, we found the similarity of fastText is 0.515 and the similarity of BoW is 0 and many BBs consist of only "mov" or "movzx". Therefore, these two advantages help inter-node GNN's classify better.

In order to train fastText, we first preprocess all assembly instructions in each function to reduce the vocabulary size. The reduction of the vocabulary size enables us to train the BB converter faster and to generate assembly instruction vectors which capture the meaning of the assembly instructions more accurately [18,29]. Based on Marcelli et al. [17], we extract opcodes from all assembly instructions in each function. For example, "mov" is extracted from the following instruction.

```
mov eax, 800
```

After preprocessing, opcodes and BBs are considered as words and sentences, respectively. This methodology regarding contents in BBs as words and BBs as sentences is commonly used to create BB vectors [10,28,29]. Then, opcodes in each BB are fed into fastText. In order to generate opcode vectors, each opcode is divided into subwords by N-gram. N-gram is a representation method of contiguous sequences of N characters in a word [4]. For example, when N equals to 3, the mov opcode is divided into as follows.

```
<mo, mov, ov>
```

The < and > are boundary symbols at the beginning and the end of a word [4]. fastText generates opcode vectors using the subword embeddings. After that, we create a BB vector by averaging of opcode vectors in the BB.

3.2 Function Converter

As the function converter, we use GNN and GMN models [13]. The difference between the GNN model and the GMN model lies in their approach to generating vectors for functions. In the GNN model, each function vector is generated based only on its respective BB vector and CFG. On the other hand, in the GMN model each generated function vector is influenced by the BB and CFG of its paired counterpart, which is another function from a pair of functions (Sect. 2.2). We

use and tweak GNN and GMN models based on prior work [17], which provides detailed instructions on how to use and tweak them.

3.3 Early Stopping Based on MRR

In order to optimize GNN and GMN models for calculating function similarity, we leverage early stopping [20] based on MRR. Early stopping is a methodology used in training to halt the learning process when evaluation metrics, such as the area under curve (AUC) of the receiver operating characteristic (ROC) curve, no longer exhibit continuous improvement during the training. The primary objective of applying early stopping in training is to prevent overfitting. Overfitting is a problem in which a model is trained extremely to match trends in the training dataset, such that the model achieves high accuracy on the training dataset, but fails to generalize well to data outside the training dataset.

MRR is an evaluation metric for ranking accuracy and is used for evaluating binary function comparison methods [17,27]. The computation of MRR involves calculating the reciprocal of the highest rank position for the correct answer for each search query and then taking the average of these reciprocals. The value of MRR falls within the range of 0 to 1, with the value closer to 1 indicating superior ranking of search performance. For instance, MRR10 assesses search performance within the top 10 positions, and if, for a specific search query, the correct answer is ranked second, the MRR10 will be 0.5.

In Sect. 4, we show that applying early stopping based on MRR to training enhances the search performance of our method.

4 Evaluation

Our evaluation is based on the comparison performed by Marcelli et al. [17]. We trained and tested GNN and GMN models [13] as multi-architecture models and single-architecture models. Functions of ARM, x86 and MIPS are fed into multi-architecture models for training. The functions come from ELF format binaries. We trained and tested multi-architecture models on an Amazon EC2 P3 instance (p3.2xlarge), which is equipped with Ubuntu 20.04, Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz and 61 GB memory.

In regard to single-architecture models, we fed the models functions of x86 32-bit binaries in PE format. We trained and tested single-architecture models on a workstation equipped with Ubuntu 18.04 on Windows 10 Pro 64-bit with WSL, Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz and 64 GB memory.

For preparing datasets, we used Python programs which are open to the public and were provided by prior work [17]. Some of these Python programs were originally created to work with IDA Pro [8] 7.3, and we worked on rewriting them to work with IDA Pro 7.7. Thus, some of the dataset preprocessing, such as disassembling binaries and counting BBs in a function, depends on IDA Pro.

We implemented multi-architecture models and single-architecture models in Tensorflow 1.14 and conducted training based on Python programs publicly available via prior work [17].

4.1 Dataset

We prepared two datasets for multi-architecture models, and one dataset for single-architecture models.

Dataset-1 `Dataset-1` comprises seven open source projects: `ClamAV`, `Curl`, `Nmap`, `OpenSSL`, `Unrar`, `Z3` and `Zlib` [17]. These open source projects are compiled for ARM, x86, MIPS in 32- and 64-bit versions and are compiled by Clang and GCC with 4 different versions each and with 5 optimizations (O0, O1, O2, O3 and Os) [17]. 5,489 binaries of Dataset-1 are in ELF format and are used for the training and the evaluation of multi-architecture models. We obtained Dataset-1 preprocessed by Marcelli et al. [17]. Prior work removed duplicate functions based on their names and the hash values of their instructions [17].

In total, Dataset-1 comprises approximately 790k functions, each of which has more than four BBs. Dataset-1 is split into training, validation and testing datasets. The training dataset comprises approximately 260k functions from ClamAV, Curl, OpenSSL and Unrar. The validation dataset comprises approximately 10k functions from Zlib. The testing dataset comprises approximately 520k functions from Nmap and Z3.

Dataset-BINKIT We obtained `Normal-dataset` from prior work [12] and selected some binaries of Normal-dataset as another testing dataset for the evaluation of multi-architecture models. We call this testing dataset Dataset-BINKIT. In order to investigate the impact of the number of software used to a testing dataset on performance, we minimized the differences between Dataset-1 and Dataset-BINKIT as much as possible, excluding the number of software. We selected 7,200 binaries in ELF format, which are compiled for ARM, x86, MIPS in 32- and 64-bit versions and are compiled by Clang and GCC with 2 different versions each and with 4 optimizations (O0, O1, O2 and O3). We preprocessed Dataset-BINKIT using prior work's code [17], which enables us to remove duplicate functions based on their names and the hash values of their opcodes. We also removed functions in Dataset-BINKIT which are the same as functions in Dataset-1 in terms of the hash values of their opcodes. In total, Dataset-BINKIT comprises approximately 510k functions which have more than four BBs. The selected binaries in Dataset-BINKIT are the following GNU software packages; `gsl`, `gss`, `gzip`, `hello`, `inetutils`, `libiconv`, `libidn`, `libmicrohttpd`, `libtasn1`, `libtool`, `libunistring`, `lightning`, `macchanger`, `nettle`, `osip`, `patch`, `plotutils`, `readline`, `recutils`, `sed`, `sharutils`, `spell`, `tar`, `texinfo`, `time`, `units`, `wdiff`, `which` and `xorriso`.

Dataset-Win32 We first collected six open source project source codes: `Curl`, `FFmpeg`, `OpenSSL`, `PuTTY`, `SQLite` and `Zlib`. Then, we compiled them for x86 (32-bit) and by Clang and GCC with 2 different versions each and with 5 optimizations (O0, O1, O2, O3 and Os). 480 binaries of Dataset-Win32 are in PE

format and are used for the training and the evaluation of single-architecture models. We preprocessed Dataset-Win32 using prior work's code [17]. In total, Dataset-Win32 comprises approximately 260k functions which have more than four BBs. In the evaluation of single-architecture model, we performed 5-fold cross validation because Dataset-Win32 is smaller than the datasets for multi-architecture models. For convenience, we represent each separate cross validation fold as cv1, ..., cv5. In each cross validation fold, the training dataset comprises approximately 210k functions and the validation dataset comprises approximately 50k functions.

4.2 Machine Learning Models

For multi-architecture models, we evaluated the models using holdout validation to compare them with the GNN model [17]. When data imbalance occurs upon splitting datasets using the holdout validation, the data imbalance can introduce potential bias in the evaluation of a model [22]. Also, other work has shown that k-fold cross validation tends to give better results compared to holdout validation [26]. To mitigate the bias, we employed five different random seeds to conduct the evaluation and obtained the mean results of the all random seeds. Altering the random seeds leads to changes in the training of GNN and GMN models, such as model initialization and variations of function pairs used for training. We first employed random seed 11 because the GNN model [17] to which we compare our results appeared to be trained on random seed 11, according to prior work [17]. We also tried random seeds 12 through 15, with similar results.

In regard to the evaluation of single-architecture models, we evaluated the models using cv1, ..., cv5 and random seed 11. Before training the models, We modified the training process to ensure that the function pairs were not duplicated in each epoch. The rationale behind this is to avoid the duplication of function pairs due to the lower abundance of compiler versions and architectures compared to the training dataset for the multi-architecture models.

Baseline In the evaluation of multi-architecture models, we prepared three baseline models. One of them is a GNN model which is created by Marcelli et al. [17] and is open to the public. The GNN model uses BB vectors created by BoW. The others are a GNN model and a GMN model which use BB vectors created by BoW and are trained by early stopping based on MRR. As a BB converter, BoW serves to convert BBs into BB vectors, which are generated from the counts of opcode occurrences within the top 200 opcodes in all BBs of the training dataset from Dataset-1. We selected the top 200 opcodes using prior work's code [17]. The other training configuration for the GNN model and the GMN model, such as the dimension of function vector (128-dimension), is the same as Marcelli et al. [17].

In regard to the evaluation of single-architecture models, we prepared a GNN model and a GMN model which uses BB vectors created by BoW as baseline models. BB vectors are generated from the counts of opcode occurrences within

the top 200 opcodes in all BBs of the training dataset from cv1, ..., cv5. We selected the top 200 opcodes using prior work's code [17]. The top 200 opcodes vary depending on cv1, ..., cv5. The baseline models are trained by early stopping based on MRR. The other training configuration for the GNN model and the GMN model is the same as Marcelli et al. [17].

Our Proposed Model In the evaluation of multi-architecture models, we prepared a GNN model and a GMN model which use BB vectors, which are the average of the embeddings of opcodes in each BB. The embeddings are created by a fastText [4] model. The fastText model is trained using BBs of the training dataset from Dataset-1. The training configuration for the fastText model includes skipgram, 200 iterations, minn 3, maxn 6, window size 5, minCount 1, and a BB vector dimension of 200. After training, the fastText model takes each of the BBs of Dataset-1 and Dataset-BINKIT as input and generates corresponding BB vectors for each of the BBs. Our proposed models are trained by early stopping based on MRR. The other training configuration is the same as Marcelli et al. [17].

In regard to the evaluation of single-architecture models, we prepared a GNN model and a GMN model which use BB vectors, which are the average of the embeddings of opcodes in each BB. The embeddings are created by a fastText model. The training configuration for the fastText model includes skipgram, 200 iterations, minn 3, maxn 3, window size 5, minCount 1, and a BB vector dimension of 128. The fastText model is trained using BBs of the training dataset from Dataset-Win32. In total, we created five fastText models from the training datasets in cv1, ..., cv5. The other training configuration is the same as the evaluation of multi-architecture models.

4.3 Measures of Performance

Based on Marcelli et al. [17], in order to evaluate the performance of multi-architecture models, we prepared four different tasks to evaluate: (1) XA: two functions of a pair come from different architectures and bitness, but the same compiler, compiler version, and optimization. (2) XC: two functions of a pair come from different compiler, compiler versions, and optimizations, but the same architecture and bitness. (3) XC+XB: two functions of a pair come from different compiler, compiler versions, optimizations, and bitness, but the same architecture. (4) XM: two functions of a pair come from randomly selected architectures, bitness, compiler, compiler versions, and optimizations. We also prepared two different tests based on Marcelli et al. [17]: (i) AUC, which is one of the metrics to evaluate classifiers. (ii) MRR10 and the recall at different K thresholds (Recall@K), which are common ranking metrics. Ranking metrics are useful to assess the search performance, particularly in applications requiring the exploration of candidate functions within a large database [17].

Pair Creation We created positive pairs and negative pairs based on function names and tasks. In the case of training, we created them based on function

names. For instance, when two functions have the same function name although the two functions are compiled by different configurations, such as optimization, the two functions are regarded as a positive pair. On the other hand, when two functions have different function names, the two functions are regarded as a negative pair regardless of the compilation settings. We created positive pairs and negative pairs using prior work's code [17]. Additionally, two functions of a positive pair for AUC test, the ranking test and early stopping come from binaries with the same name but compiled by different configurations.

Calculation of AUC For the first test, we prepared two datasets of 50k positive pairs and 50k negative pairs for each task from Dataset-1 and Dataset-BINKIT using prior work's code [17]. As for Dataset-1, we first tried to use two datasets from Marcelli et al. [17] to replicate their experiment. However, we found duplicate pairs from the positive pairs from Dataset-1 and some positive pairs for XA, XC, and XC+XB from Dataset-1 partially do not satisfy the constraint of each task. We also fixed a bug in prior work's AUC code, which caused an error of approximately 0.3% in the AUC calculation on Dataset-1. We performed the first test in the evaluation of multi-architecture models.

Calculation of MRR10 and Recall@K For the ranking test of the XM task, we randomly selected 20k positive pairs and 2,000k negative pairs, that is 100 negative pairs for each positive one.

In order to evaluate the performance of single-architecture models, we prepared the XM task. The XM task is evaluated according to two commonly used ranking metrics, MRR10 and Recall@K. For the ranking test of the XM task, we randomly selected 20k positive pairs and 6,000k negative pairs, that is 300 negative pairs for each positive one.

Configuration of Early Stopping Based on MRR10 We prepared four different patience values to stop training, early stopping patience (EP) 5, EP10, EP15 and EP20. For example, training on EP5 would be halted when the value of MRR10 does not improve for five consecutive epochs. In the evaluation of multi-architecture models, MRR10 in each epoch is calculated by 500 positive pairs and 50k negative pairs of the validation dataset from Dataset-1 for XM task. In regard to the evaluation of single-architecture models, MRR10 in each epoch is calculated by 500 positive pairs and 150k negative pairs of the validation dataset from Dataset-Win32 for XM task. The positive and negative pairs in early stopping and in the ranking test are not duplicated.

4.4 Results of Multi-architecture Models

From this point onward, we shall refer to the GNN model and the GMN model of our proposed models as GNN+fastText and GMN+fastText respectively. Likewise, we shall refer to a GNN model created by Marcelli et al. [17] as the GNN+

Table 1. AUC, Recall@1 and MRR10 comparison between our proposed models and the baseline models on Dataset-1. Each value is the average from random seed 11 to 15. Using fastText [4] improves the baseline performance in almost every case.

Model	XA	XC	XC+XB	XM	Recall@1(XM)	MRR10(XM)
GNN+BoW	0.98	0.82	0.82	0.88	**0.48**	0.55
GNN+fastText	0.98	**0.86**	**0.86**	**0.90**	**0.48**	**0.56**
GMN+BoW	0.98	0.79	0.78	0.85	0.43	0.51
GMN+fastText	**0.99**	0.83	0.83	0.88	0.44	0.53

BoW epoch10 [17], and a GNN model and a GMN model which use BB vectors created by BoW and are trained by early stopping based on MRR as GNN+ BoW and GMN+BoW.

Results of Training Fig. 2 shows the results of training with random seed 11. On EP15, GNN+fastText is better than the other models according to MRR10 although GNN+fastText necessitates a greater number of epochs in comparison to the other models. The MRR10 of GNN+fastText is 0.814 at epoch 75, whereas the MRR10 of GNN+BoW is 0.797 at epoch 24. Also, the MRR10 of GMN+ fastText is 0.738 at epoch 20, whereas the MRR10 of GMN+BoW is 0.745 at epoch 12. For your reference, we calculated AUC in each epoch using 10k positive pairs and 10k negative pairs for the only XM task, which come from prior work [17]. Recall@1 is also calculated by the same pairs as the ones to calculate MRR10. In comparison to MRR10, AUC converges more quickly after 20 iterations rather than 60 in the result of GNN+fastText. The similar trend observed in training with random seed 11 was confirmed for the other random seeds as well. In the evaluation of Multi-Architecture Models, GNN+fastText and GNN+ BoW are trained on EP15 for the evaluation. We confirmed that the result of GNN+BoW on EP20 with random seed 11 was slightly worse than on EP15 in Dataset-1, and GNN+BoW on EP20 with random seed 12 and 15 were slightly worse than on EP15 in Dataset-BINKIT. To mitigate the impact of overfitting, we evaluated GNN+fastText and GNN+BoW which are trained on EP15. On the other hand, GMN+fastText and GMN+BoW are trained on EP20.

Results of the Testing Dataset of Dataset-1 Table 1 and Table 2 show the AUC, Recall@1 and MRR10 of multi-architecture models on Dataset-1. Each value in Table 1 is the average from random seed 11 to 15, and each value in Table 2 is the result of random seed 11. Also, Table 2 shows the comparison between GNN+fastText, GNN+BoW and the GNN+BoW epoch10 [17]. In Table 1, the AUC values of 0.86, 0.86 and 0.90 for XC, XC+XB and XM respectively, the Recall@1 value of 0.48 and the MRR10 value of 0.56 show that GNN+fastText is better than the other models. In Table 2, GNN+fastText outperformed the GNN+BoW epoch10 [17] by up to 3% in AUC, 5% in Recall@1 and 5% in MRR10.

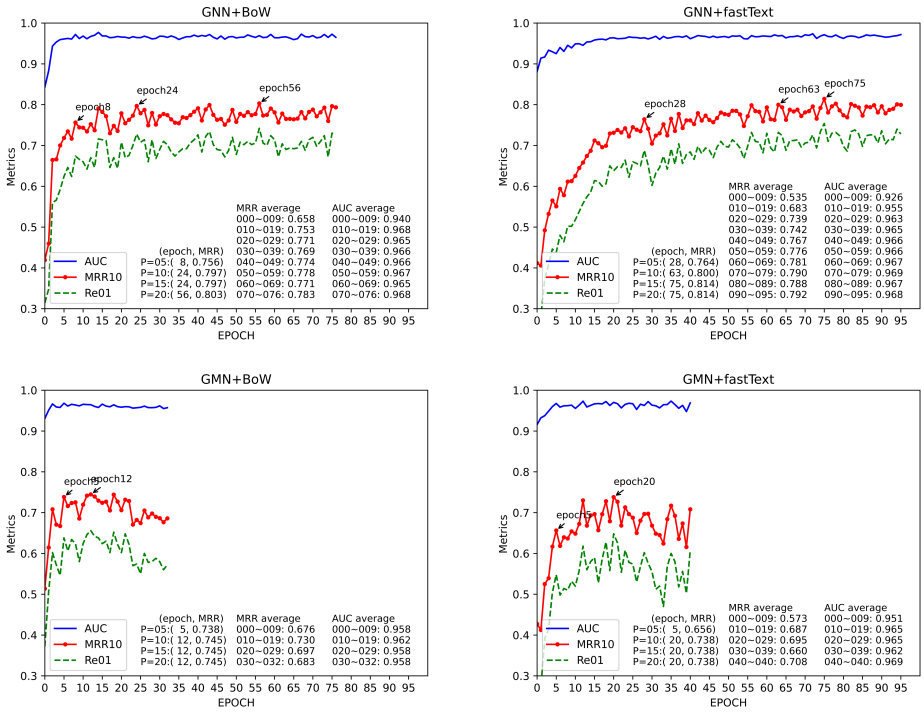

Fig. 2. The training results of GNN+fastText, GMN+fastText, GNN+BoW and GMN+BoW with random seed 11 on Dataset-1. Re01 denotes Recall@1 and P denotes early stopping patience (EP). Epoch indices start at zero. In the results of training on EP15, the MRR10 value of 0.814 shows that GNN+fastText is better than the other models.

Results of Dataset-BINKIT Table 3 and Table 4 show the AUC, Recall@1 and MRR10 of multi-architecture models on Dataset-BINKIT. Each value in Table 3 is the average from random seed 11 to 15, and each value in Table 4 is the result of random seed 11. In a manner akin to Table 2, Table 4 shows the comparison between GNN+fastText, GNN+BoW and the GNN+BoW epoch10 [17]. In Table 3, the AUC values of 0.98, 0.95 and 0.96 for XA, XC+XB and XM respectively, the Recall@1 value of 0.64 and the MRR10 value of 0.72 show that GNN+fastText is better than the other models. In Table 4, GNN+fastText outperformed the GNN+BoW epoch10 [17] by up to 2% in AUC, 9% in Recall@1 and 7% in MRR10.

4.5 Case Study of Vulnerability Search Using Multi-architecture Models

In order to validate the effectiveness of early stopping based on MRR in enhancing the search performance of multi-architecture models, we replicated the

Table 2. AUC, Recall@1 and MRR10 comparison between our proposed model and the baseline models on Dataset-1. Each model is trained with random seed 11. Using fastText [4] improves the baseline performance in almost every case.

Model	XA	XC	XC+XB	XM	Recall@1(XM)	MRR10(XM)
GNN+BoW	**0.98**	0.81	0.81	0.88	**0.49**	**0.56**
GNN+fastText	**0.98**	**0.85**	**0.84**	**0.90**	0.48	**0.56**
GNN+BoW epoch10 [17]	0.97	0.82	0.82	0.88	0.43	0.51

Table 3. AUC, Recall@1 and MRR10 comparison between our proposed models and the baseline models on Dataset-BINKIT. Each value of AUC, MRR10 and Recall@1 is the average from random seed 11 to 15. Using fastText [4] improves the baseline performance in almost every case.

Model	XA	XC	XC+XB	XM	Recall@1(XM)	MRR10(XM)
GNN+BoW	0.97	0.95	0.94	0.95	0.62	0.70
GNN+fastText	**0.98**	0.95	**0.95**	**0.96**	**0.64**	**0.72**
GMN+BoW	0.97	0.95	0.93	0.94	0.59	0.68
GMN+fastText	0.97	**0.96**	0.94	0.95	0.58	0.67

case study of vulnerability search, using a dataset from Marcelli et al. [17]. The dataset includes preprocessed functions from OpenSSL1.0.2d, which is compiled for x86, x64, ARM 32-bit and MIPS 32-bit architectures, and from two firmware images: Netgear R7000(ARM 32-bit) and TP-Link Deco M4(MIPS 32-bit). OpenSSL1.0.2d compiled for the four architectures contain ten vulnerable functions, Netgear R7000 contains four of them and TP-Link Deco M4 contains nine of them [17]. We searched for the four functions of Netgear R7000 and the nine functions of TP-Link Deco M4 using the ten functions of OpenSSL1.0.2d. We also calculated MRR10 from these search results using prior work's code [17].

Table 5 shows the MRR10 of GNN+fastText, GMN+fastText, GNN+BoW and GMN+BoW and the results of Marcelli et al. [17] on the case study. In this case study, we used GNN+fastText and GNN+BoW trained on EP15 with random seed 11 and GMN+fastText and GMN+BoW trained on EP20 with random seed 11. Also, GNN+BoW [17] and GMN+BoW [17] in Table 5 denote the results of Marcelli et al. [17]. In regard to Netgear R7000, the MRR10 values of 0.88, 1.00, 1.00 and 0.80 for x86, x64, ARM32 and MIPS32 respectively show that GMN+fastText is better than the other models in all architectures. As for TP-Link Deco M4, the MRR10 values of 0.79 and 0.75 for x86 and x64 respectively show that GMN+fastText is better than the other models in x86 and x64. Also, the MRR10 values of 0.78 and 0.78 for ARM32 and MIPS32 respectively show that GMN+BoW are better than the other models in ARM 32-bit and MIPS 32-bit.

4.6 Results of Single-architecture Models

Fig. 3 shows that the results of training with random seed 11 in cv1. On EP15, GNN+fastText is better than the other models according to MRR10 although GNN+fastText necessitates a greater number of epochs in comparison to the

Table 4. AUC, Recall@1 and MRR10 comparison between our proposed model and the baseline models on Dataset-BINKIT. Each model is trained with random seed 11. Using fastText [4] improves the baseline performance.

Model	XA	XC	XC+XB	XM	Recall@1(XM)	MRR10(XM)
GNN+BoW	0.96	0.95	0.93	0.95	0.62	0.70
GNN+fastText	**0.98**	**0.96**	**0.95**	**0.96**	**0.66**	**0.74**
GNN+BoW epoch10 [17]	0.97	0.95	0.93	0.95	0.57	0.67

Table 5. MRR10 comparison between our proposed models and the baseline models on the case study of vulnerability search of Netgear R7000 and TP-Link Deco M4. (11) represents the result of random seed 11 and [17] denotes the result of prior work [17].

	Netgear R7000				TP-Link Deco M4			
Model	x86	x64	ARM32	MIPS32	x86	x64	ARM32	MIPS32
GNN+BoW (11)	0.60	0.58	0.75	0.58	0.33	0.53	0.50	0.40
GNN+fastText (11)	0.75	0.63	0.78	0.75	0.51	0.51	0.38	0.64
GMN+BoW (11)	0.67	0.50	0.56	0.38	0.56	0.67	**0.78**	**0.78**
GMN+fastText (11)	**0.88**	**1.00**	**1.00**	**0.80**	**0.79**	**0.75**	0.52	0.73
GNN+BoW [17]	0.33	0.32	0.56	0.30	0.49	0.56	0.36	0.61
GMN+BoW [17]	**0.88**	0.54	**1.00**	0.79	0.67	0.73	0.70	**0.78**

other models. The MRR10 of GNN+fastText is 0.819 at epoch 82, whereas the MRR10 of GNN+BoW is 0.813 at epoch 42. Also, the MRR10 of GMN+fastText is 0.778 at epoch 32, whereas the MRR10 of GMN+BoW is 0.715 at epoch 3. For your reference, we calculated AUC and Recall@1 in each epoch during training. Each AUC is calculated by 40k positive pairs and 40k negative pairs of the validation dataset from Dataset-Win32 for XM task. Each Recall@1 is calculated by the same pairs to calculate MRR10. In comparison to MRR10, AUC converges more quickly after 10 iterations rather than 40 in the result of GNN+fastText. The similar trend observed in training in cv1 was confirmed for cv2, ..., cv5 as well. To mitigate the impact of overfitting, we evaluated GNN+fastText and GNN+BoW on EP15 for the evaluation. This is because the result of GNN+BoW in cv2 and of GNN+fastText in cv3 on EP20 with random seed 11 were slightly worse than on EP15. On the other hand, GMN+fastText and GMN+BoW are trained on EP20.

Table 6 shows Recall@1 and MRR10 of GNN+fastText, GMN+fastText, GNN+BoW and GMN+BoW on Dataset-Win32. Each value in Table 6 is the average from cv1 to cv5. In Table 6, the Recall@1 value of 0.73 and the MRR10 value of 0.79 show that GNN+fastText is better than the other models.

4.7 Case Study of Function Search Using GNN+fastText

In order to validate the search performance of our proposed model in malware analysis, using the samples of LockBit version 3.0 (LockBit 3.0) and a LockBit 3.0 builder obtained from Vx Underground [1] we conducted a search for the functions of LockBit 3.0 using the GNN+fastText, which is trained in cv1 with random seed 11 on EP15. LockBit is a variant of ransomware that was prominent in 2022, and a builder for LockBit 3.0 was leaked [5], making it easier to

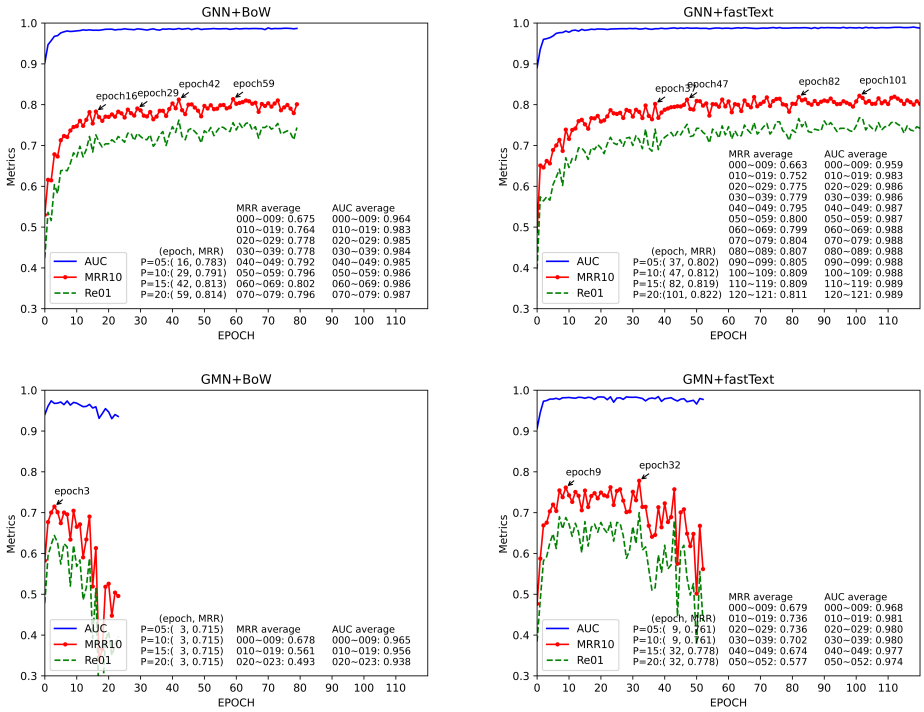

Fig. 3. The training results of GNN+fastText, GMN+fastText, GNN+BoW and GMN+BoW with random seed 11 on cv1. Re01 denotes Recall@1 and P denotes early stopping patience (EP). Epoch indices start at zero. In the results of training on EP15, the MRR10 value of 0.819 shows that GNN+fastText is better than the other models.

analyze. Analysis of LockBit 3.0 [23] revealed that there was a variant of LockBit 3.0 which has four distinct phases "Unpack Sections", "Reconstruct IAT", "Escalate Privilege", and "Ransom Main". The variant was also found to call corresponding functions for each of these phases within the `.itext` section [23]. For experimental purposes, using the LockBit 3.0 builder which is uploaded on Vx Underground [1], we created a variant (LockBit 3.0 encryptor), which was unpacked and included each phase excluding the Unpack Sections.

In light of this, using four functions each from the Reconstruct IAT phase (`Reconstruct IAT, create_heap_API_jmp_table, hash_api_address_resolve, gen_random`), the Escalate Privilege phase (`Escalate Privilege, Load Configuration, System Language Discovery, Set Icon`), and the Ransom Main phase (`Ransom Main, Drop Ransom Note, Set Wallpaper, Wipe Recycle Bin`), we conducted the case study of function search, which is based on the case study of vulnerability search by Marcelli et al. [17]. As for the twelve function names,`create_heap_API_jmp_table, hash_api_address_resolve, gen_random` were chosen based by prior work [21], and the rest were chosen based by

Table 6. MRR10 and Recall@1 comparison between our proposed models and the baseline models on Dataset-Win32. Each value is the average from cv1 to cv5.

Model	Recall@1(XM)	MRR10(XM)
GNN+BoW	0.72	0.78
GNN+fastText	**0.73**	**0.79**
GMN+BoW	0.64	0.71
GMN+fastText	0.66	0.74

prior work [23]. We prepared three targets for the case study, which are obtained from Vx Underground [1]. The SHA-256 hash values of the targets are as follows.

`Sample-58260:58260a6687486e39dc46461270b391280b7d59997d84b6639230d95e3bdfca23`

`Sample-87b76:87b76f35740262abb8da224b94779ff56eb6346318b4f9fb1988a59a72a4e6c9`

`Sample-a56b4:a56b41a6023f828cccaaef470874571d169fdb8f683a75edd430fbd31a2c3f6e`

In our function search, we confirmed that 12 out of 12 functions are detected in each target. In this case study, "detected" means that the result of a function search, using a specific function as a query for a given target, ranked first. We also observed that 31 out of 36 functions in the targets matched those of the LockBit 3.0 encryptor when comparing functions by hashopcode. The hashopcode, in this context, is a hash value of opcodes of a function. We computed the hashopcode of each function using the Python program from prior work [17]. In terms of hashopcode, `gen_random` in each target, `Reconstruct IAT` and `create_heap_API_jmp_table` in Sample-58260 are functions which did not match the corresponding ones of the LockBit 3.0 encryptor.

5 Discussion

5.1 Multi-architecture Models

GNN+fastText outperformed the GNN+BoW epoch10 [17] by up to 2% in AUC, 9% in Recall@1 and 7% in MRR10 in the evaluation of multi-architecture models with random seed 11 on Dataset-BINKIT. In regard to the results of AUC, GNN+fastText is higher than that of the other models in almost every case. On the other hand, in the average results of random seed 11 to 15 in the ranking test (Table 1 and Table 3), a difference of up to 2% in MRR10 and up to 2% in Recall@1 was observed between GNN+fastText and GNN+BoW. From these results, it is considered that the superior performance of GNN+fastText and GNN+BoW over the GNN+BoW epoch10 [17] on Dataset-1 and Dataset-BINKIT can be attributed to the effectiveness of early stopping based on MRR.

Additionally, it is necessary to prepare multiple seeds for the evaluation because altering the random seeds will change the search performance. For example, in the results of GNN+fastText on Dataset-BINKIT (Table 3 and Table 4), there was a 2% difference in Recall@1 and MRR10 between the average results and those obtained with random seed11.

In terms of training time, a tendency was observed that GNN+fastText required more time for training compared to GNN+BoW (Sect. 4.4). This is attributed to the denser BB vectors created by fastText, which is believed to

have contributed to the increased time requirement in comparison to those created by BoW. On the other hand, no significant difference was observed between GMN+fastText and GMN+BoW. The GMN models (GMN+fastText and GMN+BoW) conclude their training relatively quickly compared to GNN models (GNN+fastText and GNN+BoW). However, the GMN models exhibit more pronounced fluctuations in MRR10 during training, making it challenging to enhance the search performance of the GMN models.

When comparing the performance of the GNN models and the GMN models, it was observed that the GNN models outperformed the GMN models on Dataset-1 and Dataset-BINKIT (Sect. 4.4), whereas in the case study of vulnerability search, the GMN models yielded better results (Sect. 4.5). Notably, the case study involved searches for four or nine functions, while Dataset-1 and Dataset-BINKIT encompassed searches across 20k functions. Therefore, the results of Dataset-1 and Dataset-BINKIT are more reliable in terms of evaluating the search performance of each model compared to the case study. Furthermore, the function vectors generated by the GMN models exhibit slight variations depending on the pairs [13]. This results in an increased number of searches when searching for a specific function. For instance, when searching for one function similar to a particular function among 100 functions, the GMN models require creating 100 function pairs, necessitating 200 conversions of functions into vectors. In contrast, the GNN models only require 101 conversions of functions into vectors, making it nearly twice as fast in terms of function-to-vector conversion speed. In the realm of malware analysis, where new malware is being developed at a rate of 4.2 samples every second [3], the speed of analysis is important. Therefore, leveraging the GNN models in malware analysis are highly desirable. However, it cannot be discounted that with more meticulous hyperparameter tuning and the addition of training data, the performance of the GMN models could potentially surpass that of the GNN models.

We conducted a further experiment because we noticed that there were duplicate functions in terms of the hash values of their opcodes in Dataset-1 we obtained. Although prior work removed duplicate functions based on their names and the hash values of their instructions [17], we hypothesized that removing duplicate functions based on the hash values of their opcodes would be appropriate for machine learning models that use features of opcodes, not instructions. In the further experiment, we removed approximately 12% and 25% of functions from the training dataset and validation dataset, respectively, based on for their names and the hash values of their opcodes based on prior work's code [17]. Using the revised Dataset-1, we recreated the top 200 opcodes and a fastText model, and trained GNN+BoW and GNN+fastText with random seed 11 to 15. After the training, we tested the models on EP15 using the same testing datasets from Dataset-BINKIT in Table 3 and Table 4. As a result, we observed on average from random seed 11 to 15 that GNN+fastText outperformed GNN+BoW on the XM task for AUC and MRR10 while GNN+BoW outperformed GNN+fastText on the XC task for AUC. As for the comparison between GNN+BoW epoch10 [17] and GNN+fastText with random seed 11, GNN+fastText outper-

formed GNN+BoW epoch10 [17] by up to 1% in AUC, 5% in Recall@1 and 3% in MRR10.

5.2 Single-architecture Models

In terms of AUC Recall@1 and MRR10, the single-architecture models exhibited a similar trend to the multi-architecture models. It is difficult to assess the effectiveness of early stopping based on MRR compared to the evaluation of multi-architecture models. However, because single-architecture models can detect functions of LockBit 3.0 accurately (as shown in the case study), they are considered to be sufficiently optimized for search performance.

5.3 Future Work

To further enhance the search performance in malware analysis using our proposed models, we plan on implementing data augmentation utilizing Malware Makeover [15,16]. In existing methods for calculating binary function similarity, function pairs based on function name are required to train the existing methods. However, the function name of malware typically has been stripped, making it exceedingly challenging to conduct training using functions of malware. Moreover, even if some functions have the same function name, there is no guarantee that they possess the same functionality. Therefore, we leverage Malware Makeover to enhance the search performance in malware analysis. Malware Makeover can create variants from malware, which retain the functionality of original malware. By conducting additional training using Malware Makeover, we aim to achieve further improvement in the search performance of our proposed models. We also plan to compare binary program similarity using a GNN model [13]. We hypothesize that the GNN model may perform well for calculating binary program similarity because the GNN model can convert a call-tree into a vector.

6 Conclusion

In this paper we proposed the method combining GNN and GMN models [13] and fastText [4] to transform functions in a binary into vectors for binary function similarity comparison. GNN+fastText outperformed the GNN+BoW epoch10 [17] by up to 2% in AUC, up to 9% in Recall@1 and up to 7% in MRR10 in a certain case when training multi-architecture models with random seed 11 (Sect. 4.4). In the case study of vulnerability search (Sect. 4.5), specific values of MRR10 in Table 5 show that GMN+fastText are better than the baseline models. Furthermore, our findings suggest that early stopping based on MRR enables to improve the search performance of GNN+fastText and GNN+BoW. In the case study of single-architecture models (Sect. 4.7), our work shows that GNN+fastText is effective to search for malicious functions of LockBit 3.0 because GNN+fastText was able to find 12 out of 12 functions which we prepared

for the case study. In future work, to further enhance the search performance in malware analysis using our proposed models, we plan to implement data augmentation utilizing Malware Makeover [15,16] (Sect. 5.3).

Acknowledgments This work was supported in part by the U.S.\ Army Research Office under MURI grant W911NF-21-1-0317; by Carnegie Mellon CyLab; and by National Police Agency of Japan.

References

1. Vx Underground. https://www.vx-underground.org/, last accessed 12 Dec 2023
2. Ahmad, I., Luo, L.: Unsupervised binary code translation with application to code clone detection and vulnerability discovery. In: Findings of the Association for Computational Linguistics: EMNLP 2023. pp. 14581–14592 (2023)
3. AV-TEST: SECURITY REPORT 2019/2020. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2019-2020.pdf, last accessed 10 Nov 2023
4. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T.: Enriching word vectors with subword information. Transactions of the Association for Computational Linguistics **5**, 135–146 (2017)
5. Cybersecurity and Infrastructure Security Agency: Understanding Ransomware Threat Actors: LockBit. https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-165a/, last accessed 09 Nov 2023
6. Dai, H., Dai, B., Song, L.: Discriminative embeddings of latent variable models for structured data. In: Proceedings of The 33rd International Conference on Machine Learning. pp. 2702–2711. PMLR (2016)
7. Gao, J., Yang, X., Fu, Y., Jiang, Y., Sun, J.: Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 896–899 (2018)
8. Hex-Rays: IDA Pro. https://hex-rays.com/ida-pro/, last accessed 15 Oct 2023
9. Hu, Y., Wang, H., Zhang, Y., Li, B., Gu, D.: A semantics-based hybrid approach on binary code similarity comparison. IEEE Transactions on Software Engineering **47**(6), 1241–1258 (2021)
10. Ito, N., Hashimoto, M., Otsuka, A.: Feature extraction methods for binary code similarity detection using neural machine translation models. IEEE Access **11**, 102796–102805 (2023)
11. Ivanov., V., Romanov., V., Succi., G.: Predicting type annotations for python using embeddings from graph neural networks. In: Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS. pp. 548–556. SciTePress (2021)
12. Kim, D., Kim, E., Cha, S.K., Son, S., Kim, Y.: Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. IEEE Transactions on Software Engineering **49**(4), 1661–1682 (2023), https://doi.org/10.1109/TSE.2022.3187689
13. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph matching networks for learning the similarity of graph structured objects. In: Proceedings of the 36th International Conference on Machine Learning. pp. 3835–3845. PMLR (2019)

14. Lin, Z., Feng, M., Santos, C.N.d., Yu, M., Xiang, B., Zhou, B., Bengio, Y.: A structured self-attentive sentence embedding. arXiv preprint arXiv:1703.03130 (2017)
15. Lucas, K., Pai, S., Lin, W., Bauer, L., Reiter, M.K., Sharif, M.: Adversarial training for raw-binary malware classifiers. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 1163–1180 (2023)
16. Lucas, K., Sharif, M., Bauer, L., Reiter, M.K., Shintre, S.: Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. pp. 744–758 (2021)
17. Marcelli, A., Graziano, M., Ugarte-Pedrero, X., Fratantonio, Y., Mansouri, M., Balzarotti, D.: How machine learning is solving the binary function similarity problem. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2099–2116. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli
18. Massarelli, L., Di Luna, G.A., Petroni, F., Baldoni, R., Querzoni, L.: SAFE: Self-attentive function embeddings for binary similarity. In: Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 309–329. Springer International Publishing (2019)
19. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
20. Shen, R., Gao, L., Ma, Y.A.: On optimal early stopping: Over-informative versus under-informative parametrization. arXiv preprint arXiv:2202.09885 (2022)
21. Tanaka, Y.: https://jp.security.ntt/tech_blog/102hz18 (2022), last accessed 21 Oct 2023
22. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: An empirical comparison of model validation techniques for defect prediction models. IEEE Transactions on Software Engineering **43**(1), 1–18 (2017)
23. TXOne Networks: LockBit 3.0 Analysis: How to Enhance Ransomware and Malware Protection. https://www.txone.com/blog/malware-analysis-lockbit-3-0/, last accessed 21 Oct 2023
24. Wang, M., Interrante-Grant, A., Whelan, R., Leek, T.: COBRA-GCN: Contrastive learning to optimize binary representation analysis with graph convolutional networks. In: Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 53–74. Springer International Publishing (2022)
25. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 363–376 (2017)
26. Yadav, S., Shukla, S.: Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification. In: 2016 IEEE 6th International conference on advanced computing (IACC). pp. 78–83 (2016)
27. Yang, S., Dong, C., Xiao, Y., Cheng, Y., Shi, Z., Li, Z., Sun, L.: Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge. ACM Trans. Softw. Eng. Methodol. **33**(1), 1–40 (2023)
28. Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., Wu, S.: Order matters: Semantic-aware neural networks for binary code similarity detection. Proceedings of the AAAI Conference on Artificial Intelligence **34**(01), 1145–1152 (2020)
29. Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., Zhang, Z.: Neural machine translation inspired binary code similarity comparison beyond function pairs. arXiv preprint arXiv:1808.04706 (2018)