Journal of Universal Computer Science, vol. 28, no. 7 (2022), 758-775
submitted: 14/10/2021, accepted: 11/5/2022, appeared: 28/7/2022

Disassemble Byte Sequence Using Graph Attention Network

Jing Qiu
(Zhejiang A&F University, Hangzhou, China
https://orcid.org/0000-0003-3264-1681, qiujing@zafu.edu.cn)

Feng Dong
(Harbin University of Science and Technology, Harbin, China
https://orcid.org/0000-0002-3496-4305, fengdong97@qq.com)

Guanglu Sun
(Harbin University of Science and Technology, Harbin, China
https://orcid.org/0000-0003-2589-1164, sunguanglu@hrbust.edu.cn)

Abstract: Disassembly is the basis of static analysis of binary code and is used in malicious code detection, vulnerability mining, software optimization, etc. Disassembly of arbitrary suspicious code blocks (e.g., for suspicious traffic packets intercepted by the network) is a difficult task. Traditional disassembly methods require manual specification of the starting address and cannot automate the disassembly of arbitrary code blocks. In this paper, we propose a disassembly method based on code extension selection network by combining traditional linear sweep and recursive traversal methods. First, each byte of a code block is used as the disassembly start address, and all disassembly results (control flow graphs) are combined into a single flow graph. Then a graph attention network is trained to pick the correct subgraph (control flow graph) as the final result. In the experiment, the compiler-generated executable file, as well as the executable file generated by hand-written assembly code, the data file and the byte sequence intercepted by the code segment were tested, and the disassembly accuracy was 93%, which can effectively distinguish the code from the data.

Keywords: Graph neural network, disassembly, function identification, reverse engineering, binary code analysis
Categories: I.2.1, D.2.7, L.4.0
DOI: 10.3897/jucs.76528

1 Introduction

In software research and analysis, the source code of most software is not available or accessible, and the software can only be analyzed effectively through binary code analysis (BCA) [Liu et al., 2013]. In this case, BCA is an important tool for software analysis, such as malicious code analysis, malware detection, vulnerability mining and analysis, etc [Song et al., 2008]. For many software, vulnerabilities are hidden and not easily detected, and BCA can be used to better discover vulnerabilities and fix them [Djoudi and Bardin, 2015]. In the process of computer use, malware is hidden in the normal application software to cause damage to the computer. Malware is often not open source. Through BCA, hidden malware can be detected [Ma et al., 2020]. For malicious code hidden in normal programs, such as advertising pop-ups, malicious access to software usage, downloading irrelevant applications and other malicious code can greatly damage user privacy [Cui et al., 2018]. By BCA, they can be detected. In short, for the analysis of malicious code, BCA is a key technical tool [Cui et al., 2019].

For BCA, the usual method is to disassemble the binary code, then divide the functions and construct a control flow graph (CFG) for each function, so as to analyze the function and infer the role of the whole software accordingly[Anand et al., 2013, Besson et al., 2001]. In the IA32 architecture executable binary code, there is a mixture of code and data [Wartell et al., 2011]. Static analysis cannot distinguish between code and data [Kruegel et al., 2004]. Direct disassembly of binary code containing data will disassemble the data into assembly code, making it impossible to analyze the functionality of the binary code. Therefore, distinguishing between code and data without any debugging information is a challenging problem.

Existing BCA approaches can usually only be performed on a complete program [Rosenblum et al., 2008]. In network transmissions, the intercepted traffic packets may only be part of a program [Zhang et al., 2007]. For a sequence of bytes, the disassembly results starting from different locations are completely different. For example, "8BFF558BEC", the disassembly result from 8B is "mov edi,edi/ push ebp/ mov ebp,esp", the disassembly result from "FF" is "call dword ptr [ebp - 0x75]/ in al, dx", and the disassembly result from "55" is "push ebp/ mov ebp, esp". For a sequence of bytes, the first few bytes may be part of the previous instruction or the sequence may start with partial data. Thus, it is not possible to determine the starting byte position of the binary sequence. Usually, it can only be determined manually after disassembling from different bytes, which is very inefficient and inaccurate.

To address these problems, this paper proposes an encoding extended selection network, which tries to disassemble from each byte position for an arbitrary code sequence, and transforms the obtained disassembly results into a CFG linked to become an extended control flow graph (ECFG). Then the correct nodes in the ECFG are distinguished by a graph attention network [Velickovic et al., 2018] to obtain the CFG for each function.

The rest of the paper is organized as follows. Section 2 introduces the related work. Section 3 provides the details of the code expansion selection network. Section 4 shows the evaluation and discussion. Finally, Section 5 summarizes our work and discusses the future work.

2 Related work

The main process of disassembly is to map binary code to assembly instructions according to a certain strategy, and the common methods are mainly divided into static disassembly and dynamic disassembly [Harris and Miller, 2005]. Static disassembly can work directly on binary code. Dynamic disassembly, on the other hand, must be performed at program runtime and has a lower code coverage. The traditional static disassembly work is divided into linear sweep and recursive traversal. Linear sweep, i.e., disassembling code one instruction after another from the start byte, cannot distinguish between mixed code and data. The wrong disassembly of data into code will affect the subsequent disassembly results. Recursive traversal is control-flow oriented and continues disassembling along the control flow, but the code coverage is lower.

Andriesse et al. tested binary files generated by disassembly tools compiled in the real world [Andriesse et al., 2016]. They used 981 x86 and x64 binaries from C/C++ projects. These projects were compiled using different compilers and compilation options. They found that some high-level language constructs, such as function boundaries, were more difficult to recover than in the literature and gave a discussion and analysis of where the disassembler capabilities did not match the literature. Li et al. used a different approach to evaluate against some traditional disassembly tools [Li et al., 2020]. They used 879 binaries from unused projects that used multiple compilers and optimization settings.

Bauman et al. have implemented a new binary rewriting tool that can rewrite stripped binaries [Bauman et al., 2018]. They used two basic techniques, a superset disassembler and an instruction rewriter, to first construct a superset containing all the legal instructions of the binary code and redirect it to a new address using the instruction reassembly technique in the dynamic binary instrument via indirect control flow. Miller et al. used some heuristic rules based on the construction of the superset disassembler to calculate each assembly instruction's probability to confirm whether the instruction is a true assembly instruction [Miller et al., 2019].

A data log-based disassembly technique was proposed by Flores-Montoya et al [Flores-Montoya and Schulte, 2020]. For a stripped binary file with binary rewriting, they found that the data log inference process is particularly suitable for disassembly and corresponding analysis, and generates disassembly code with accurate symbolic information based on the data log. Ammar Ben Khadra et al. proposed a tool for rule-based disassembly methods Spedi [Ben Khadra et al., 2016]. First, all possible basic blocks are speculated and then conflicting basic blocks are refined by analysis to complete disassembly. Most of the call and jump table targets can be recovered at the same time and can be adapted to obfuscation without any symbolic information.

Pei et al. used transfer learning for the disassembly task [Pei et al., 2021]. The model is first preprocessed and trained, and then fine-tuned to perform the disassembly task, i.e., recovering function bounds and assembly instructions. It was evaluated on a set of x86/x64 binaries, and the experimental results showed that the method works well.

The input of the above approaches are executable files. They are not designed to disassemble an arbitrary byte sequence. In this paper, code extension selection network is firstly proposed to do the work. It can be used to replace or assist manual analysis for byte sequences. It is useful in binary analysis tasks.

3 Our work

An extended control flow graph (ECFG) G is a two-tuple $G = (V, E)$, where V is a set of instructions, and $E \subseteq V \times V$ is the set of possible control flow transfers between the instructions in V. Code extension selection network (CESN) is a graph neural network (GNN) [Velickovic et al., 2018] that is used to identify the correct instruction and control flow in an ECFG.

3.1 Disassembly

The first step in CESN is disassembling a byte sequence starting at each byte. Linear sweep is used to obtain all possible disassembly code. The disassembly result starting from the same position is unique. For a byte sequence $\{b_1, \cdots, b_n\}$, let DisasmFrom(i) be the disassembly result starting from byte i. Assuming that byte $b_1b_2b_3$ can form an assembly instruction, then the byte sequence $\{b_1, \cdots, b_n\}$ and $\{b_4, \cdots, b_n\}$ have the same assembly result after byte b_4. Thus, DisasmFrom$(b_4) \subset$ DisasmFrom(b_1). The whole algorithm is shown in Algorithm 1.

Algorithm 1: Disassemble a byte sequence

Input: L: a byte sequence
Output: R: disassembly result
$V \leftarrow \emptyset$, $R \leftarrow \emptyset$;
foreach $i \in L$ **do**
  if ADDRESS$(i) \notin V$ **then**
    $D \leftarrow$ DisasmFrom(i);
    $R \leftarrow R \cup D$;
    foreach $j \in D$ **do**
      $V \leftarrow V \cup \{$ADDRESS$(j)\}$
return R

3.2 ECFG construction

The extended control flow graph (ECFG) construction algorithm is shown in Algorithm 2. The overall process is similar to the construction of the CFG. If an instruction is a jump instruction and the jump target exists, an edge from the instruction to the jump target is added. If it is a conditional jump, or if it is a non-jump, an edge from the instruction to the next instruction is added [Federico and Agosta, 2016]. Figure 1 gives an example of ECFG construction.

Algorithm 2: Build an ECFG

Input: R: disassembly result
Output: G: ECFG
$G \leftarrow \emptyset$;
foreach $r \in R$ **do**
  foreach $i \in r$ **do**
    if i is a jump **then**
      if TARGET$(i) \in R$ **then**
        $G \leftarrow G \cup \{i \rightarrow$ TARGET$(i)\}$;
        if i is a conditional jump **then**
          $G \leftarrow G \cup \{i \rightarrow$ NEXT$(i)\}$
    else
      $G \leftarrow G \cup \{i \rightarrow$ NEXT$(i)\}$
return G

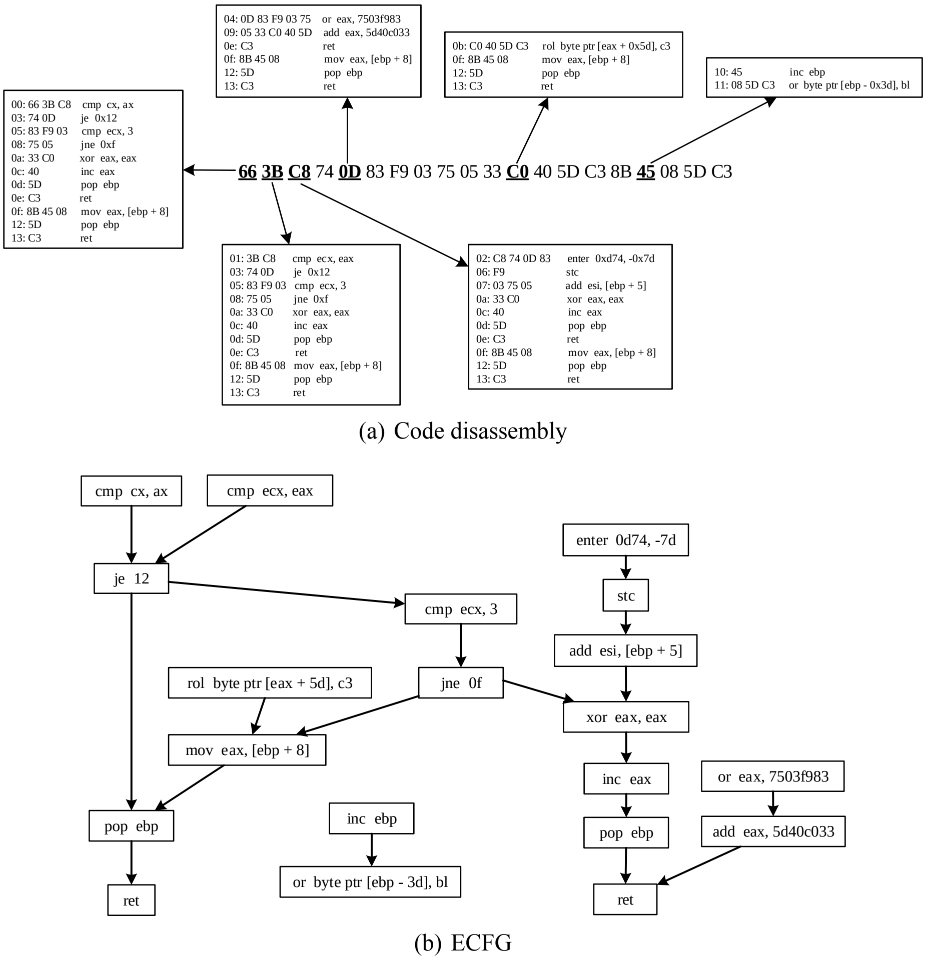


Figure 1: Example of building an ECFG

3.3 ECFG pruning

In an ECFG, there are some illegal instructions. These instructions and all their preceding instructions should be deleted. Illegal instructions are defined as follows.

1. For a conditional jump instruction, if the jump address is the instruction next to itself, it is illegal.
2. If the target of a conditional branch instruction is illegal, the instruction is illegal.
3. If no ancestor node of a conditional branch instruction modifies the corresponding register or flag bit, the branch instruction is illegal.
4. High-privilege instructions (such as I/O instructions) in normal user programs are illegal.

Removing illegal instructions in an ECFG greatly improves the running speed and reduce memory usage when analyzing large binary files.

3.4 Connected subgraph search

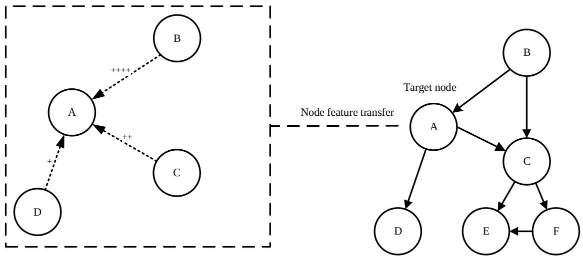
For a pruned ECFG, there may be multiple disconnected subgraphs. Each concatenated subgraph corresponds to one possible CFG of a function. Functions are usually terminated with a return/jump instruction. If a subgraph of an ECFG does not end with a return/jump instruction, the subgraph is considered illegal and can be deleted. The search process of the connected subgraphs is shown in Algorithm 3.

Algorithm 3: ECFG Subgraph Search

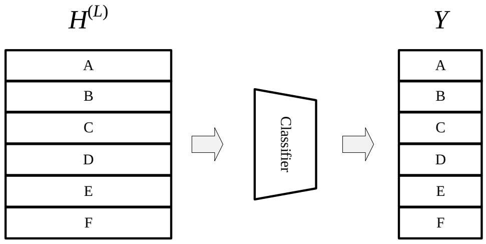
```
Input: G: an ECFG
Output: GS: connected subgraphs
GS ← ∅, workList ← ∅, visited ← ∅;
/* Search connected subgraphs                                  */
for v ∈ G do
    if v ∉ visited then
        workList.push(v);
        G′ ← ∅;
        while workList ≠ ∅ do
            v′ ← workList.pop();
            visited ← visited ∪ {v′};
            foreach p ∈ v′predecessor do
                G′ ← G′ ∪ {p → v′};
                workList.push(p);
            foreach p ∈ v′successor do
                G′ ← G′ ∪ {v′ → p};
                workList.push(p);
        GS ← GS ∪ G′;
/* Remove illegal subgraph                                     */
GS.removeIf(g → !g.exists(v → v is leaf node and v is RET or JMP));
return GS
```

3.5 CFG prediction

For a pruned ECFG, some of the nodes are true instructions, while others may be data or the result of disassembly from other addresses. The disassembly problem translates into finding the true disassembly nodes in an ECFG. Graph neural network (GNN) [Scarselli et al., 2009, Guo and Wang, 2021] is a deep learning method for feature extraction of graph data structures. After training the graph attention neural network, the logical



(a)



(b)

Figure 2: Graph attention neural network

relationship between instructions enables the model to find all correctly disassembled nodes in an ECFG.

An ECFG may contain one or more functions. The graph attention network assigns different weights to different neighboring nodes [Wang et al., 2019a]. Then, it obtains the output of each node by weighting and summing the features of the neighboring nodes with the attention mechanism [Wang et al., 2019b]. Finally, it aggregates the features of the neighboring nodes to the central node through node-by-node computation. Each operation is done by cyclically traversing all nodes on the graph [Rong et al., 2020]. The GNN is demonstrated in Fig. 2. Each central node containing the features of the neighboring nodes is classified using a fully connected network to determine whether the node is a disassembly instruction [Goodfellow et al., 2015].

The input of the graph attention layer is a node feature vector set $h = \{h_1, h_2, \dots, h_n\}$, $h_i \in R^F$, where n is the number of nodes, F is the number of node features, and R represents the features of a certain node. The output of each layer is $h' = h'_1, h'_2, \dots, h'_n, h' \in R^{F'}$. A weight matrix is trained for all nodes, $W \in R^{F'*F}$, and it is the relationship between the input F features and the output F' features. The attention coefficients are $e_{ij} = a(Wh_i, Wh_j)$, where a indicates a function that calculates the degree of correlation between two nodes.

The vector h is the feature vector of the nodes. The subscripts i, j denote the ith node and the jth node. To make the attention coefficients easier to compute and compare, `softmax` is introduced to regularize the jth and ith node where the former is the neighboring node of the latter. α is the attention coefficient, α_{ij} is the attention coefficient of nodes h_j to j_i.

$$\alpha_{ij} = \text{softmax}_j\,(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N_i} \exp(e_{ik})} \tag{1}$$

Combining the first two formulas, the complete regularization formula can be obtained as follows. The LeakyReLu is an activation function used to introduce non-linearity.

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLu}(a^T\,[Wh_i||Wh_k]))}{\sum_{k \in N_i} \exp(\text{LeakyReLu}(a^T\,[Wh_i||Wh_k]))} \tag{2}$$

The attention coefficients between different nodes after regularization are obtained through the above operations. It can be used to predict the output characteristics of each node.

$$h'_i = \sigma(\sum_{j \in N_i} \alpha_{ij} W h_j) \tag{3}$$

W is the weight matrix multiplied by the feature. α is the attention cross-correlation coefficient calculated previously. σ is the non-linear activation function. The jth node ($j \in N_i$) represents all nodes adjacent to the ith node.

In order to stabilize the learning process of the self-attention mechanism, the capabilities of the model is enhanced by using k independent attention mechanisms to execute the formula to obtain the final one:

$$h'_i = \sigma(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in N_i} \alpha_{ij}^k W^k h_j) \tag{4}$$

A total of K attention mechanisms need to be considered. k represents the kth in K, and the kth attention mechanism is a^k. The linear transformation weight matrix of the input feature under the kth attention mechanism is expressed as W^k.

Using an ECFG as input, the overall task is transformed into a node classification problem by learning all the nodes in the ECFG through graph attention networks. All nodes in the graph are classified into correct, and invalid instructions (including disassembly results from other bytes as well as data). Through training, the correct nodes in the graph can be identified. An example of classification is given in Fig. 3. After node classification, the correct disassembly result in the ECFG can be obtained. The correct nodes builds the correct CFG. The first byte of the first instruction of the CFG is the starting byte, and bytes that are not in the CFG are judged as data.

4 Experiment

4.1 Setup

4.1.1 Data set

The data set comes from three sources.

1. Win10 Professional 32-bit (version number 1507) 306 executable files under \Windows \System32. The mixing of code and data in the .text section of these files is more obvious.

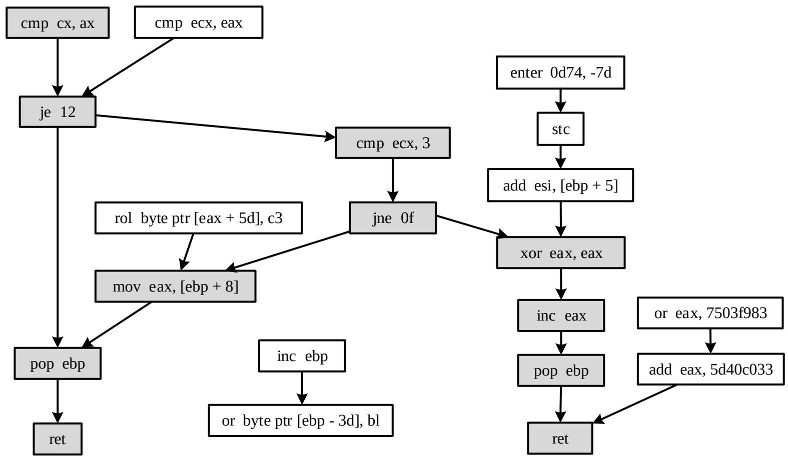


Figure 3: Example of node classification by graph attention network

2. MASM32 \examples hand-written assembly code programs. There is less data in the .text section of these files.

3. 128 pictures were randomly selected from internet for training and testing, including GIF, JFIF, JPG, PNG, WEBP picture files. They are used to test the performance of the model on data and code.

For executable files, the ground truth of disassembly code is obtained by the program database files (PDB, which contains debugging information of the executable file). For the disassembly result starting from a byte (single instruction), there are two cases: 1) the disassembly result at that byte position is correct (TrueDisasm); 2) the disassembly result at that byte position is incorrect (FalseDisasm).

The details of the data set are shown in Table 1. For compiler-generated, hand-written, and data files, these files are extracted as byte sequences and their ECFGs are constructed. In an ECFG, a node is a **TrueDisasm**, which is a true instruction corresponding to the source code; or a **FalseDisasm**, which is a false instruction disassembled from data or an incorrect address of code.

The code in a file has a high probability of having the same code style. So the training set and test set are divided in files to avoid similar style of code in them. The training set and test set files are divided by 5:1 to verify the robustness and effectiveness of a model.

4.1.2 Model settings

The disassembly engine used in this experiment is Capstone, and the deep learning framework is tf_geometric, a graph neural network library based on Tensorflow. Four layers of graph attention network are used to learn the parameters, with 8 attention heads of each layer and the activation function LeakyRelu. The final classification is performed using full connectivity with the activation function sigmoid. L2 regularization is performed with a learning rate of 5e-3 and iterations of 10000.

Data set		Files	TrueDisasm	FalseDisasm	Total
Compiler-generated	Train	284	880,992	1,552,041	2,433,033
	Test	22	69,311	119,834	189,145
Hand-written	Train	70	3,683	9,834	13,517
	Test	14	30,647	88,284	118,931
Data	Train	108	0	471,062	471,062
	Test	20	0	51,112	51,112
Total		518	984,633	2,292,167	3,276,800

Table 1: Data set in the evaluation

4.2 Result

4.2.1 Training result

The compiler-generated executables, the hand-written assembly-generated executables, and the data files are put together for training, and the results are shown in Fig. 4.

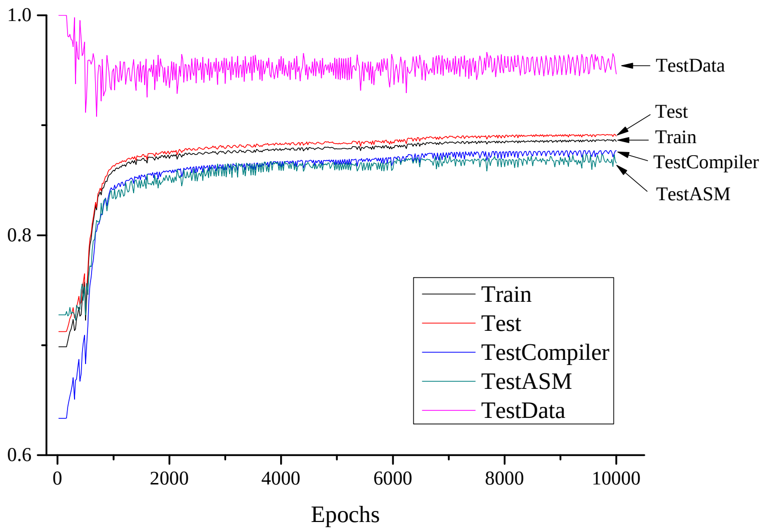


Figure 4: Accuracy of CESN on different data sets. TestCompiler represents the test set of high language programs. TestASM is test set of hand-crafted programs.

Fig. 4 shows that putting all training sets together for training can classify more accurately the disassembly results of data files, compiler-generated executables, and hand-written assembly-generated executables. The precision, recall, F1 score, and PTrueDisasm (the proportion of TrueDisasm predicted after classification among all generated nodes) are introduced to have a more accurate evaluation of the model. The models trained with all training sets, hand-crafted files, compiler-generated files are called CESN, MCG, and CGG, respectively.

$$Precision = TP/(TP + FP)$$
$$Recall = TP/(TP + FN)$$
$$F1\_score = 2 \times Precision \times Recall/(Precision + Recall)$$
$$PDisasm = TrueDisasm/(TrueDisasm + FalseDisasm)$$

Where TP refers to the predicted and actual results are TrueDisasm, TN refers to the predicted and actual results are FalseDisasm, FP refers to the predicted result is TrueDisasm while actual result is FalseDisasm, FN refers to the predicted result is FalseDisasm while actual result is TrueDisasm.

Table 2 indicates the detailed evaluation metrics for the compiler-generated and hand-written code segments of the test set, as well as the detailed evaluation metrics for the data file.

Data set	Model	Precision	Recall	F1 Score	PTrueDisasm	Accuracy
Compiler-generated	CGG	0.82990	0.83512	0.83250	**0.36875**	0.87686
	MCG	0.70093	0.61430	0.65476	0.32116	0.76262
	CESN	0.82990	0.83512	0.83250	**0.36875**	0.87686
Hand-written	CGG	0.80084	0.72387	0.76041	0.24628	0.87571
	MCG	**0.83014**	**0.90497**	**0.86594**	0.29703	0.92365
	CESN	0.77208	0.71925	0.74473	0.25383	0.86565
Data	CGG	–	–	–	0.27070	0.72930
	MCG	–	–	–	0.17606	0.82394
	CESN	–	–	–	0.05337	**0.94663**

Table 2: Result of test set

Table 2 shows that the proposed method can disassemble the complete code segment for identification and also analyze the data file. The lower accuracy is caused by the fact that the code and data are more similar, while the PTrueDisasm of the data segment is often lower than that of the code segment.

For the identification of binary sequences, 256-bytes sequences were intercepted from the compiler-generated test set and the hand-written data set to test the accuracy of the proposed method for byte sequence identification. For the compiler-generated executable files, the intercepted byte sequences were 0x500-0x600, 0x1000-0x1100, 0x1500-0x1600, and 0x2000-0x2100. Due to the small size of the manually written assembly files, only 0x100-0x200, 0x500-0x600 were intercepted. The detailed results are shown in Table 3.

Furthermore, 1,280-bytes sequences were intercepted from the compiler-generated test set and the hand-written data set to test the accuracy of the method for byte sequence identification. For the compiler-generated executable file the intercepted byte sequences were 0x500-0xa00, 0x1000-0x1500, 0x1500-0x1a00, and due to the hand-written assembly file is small, only 0x100-0x600, 0x500-0xa00 are intercepted, as shown in Table 4.

Data set	Address	Model	Precision	Recall	F1 Score	PTrueDisasm	Accuracy
Compiler-generated	500-600	CGG	0.41107	0.89474	0.56333	0.15777	0.89946
		MCG	0.44973	0.59649	0.51282	0.09613	**0.91785**
		CESN	**0.45289**	0.81330	**0.58180**	0.12267	**0.92013**
		Capstone	0.19455	**0.97569**	0.32442	0.22656	0.72570
	1000-1100	CGG	0.66262	0.93705	**0.77630**	0.17389	**0.93359**
		MCG	0.55895	0.57654	0.56761	0.12684	0.89199
		CESN	**0.67910**	0.85980	0.75884	0.15569	0.93280
		Capstone	0.33309	**0.98440**	0.49776	0.40234	0.76214
	1500- 1600	CGG	0.57692	**0.88920**	**0.69981**	0.20598	**0.89805**
		MCG	0.49645	0.59139	0.53978	0.15920	0.86522
		CESN	**0.60662**	0.78899	0.68589	0.17382	**0.90342**
		Capstone	0.32509	0.87273	0.47371	0.46094	0.74809
	2000-2100	CGG	0.59368	0.93168	**0.72524**	0.23583	0.89391
		MCG	0.58567	0.62961	0.60684	0.16155	0.87740
		CESN	**0.61961**	0.84662	0.71554	0.20534	**0.89884**
		Capstone	0.40768	**0.98411**	0.57653	0.23828	0.78682
Hand-written	100-200	CGG	0.76938	0.66724	0.71468	0.25481	0.84347
		MCG	0.85691	0.91897	0.88686	0.31510	0.93110
		CESN	0.74621	0.67931	0.71119	0.26748	0.83789
		Capstone	**0.95840**	**0.98630**	**0.97215**	0.27344	**0.98389**
	500-600	CGG	**0.77627**	0.63967	0.70138	0.149670	0.90107
		MCG	0.77488	0.91341	**0.83846**	0.21410	**0.93607**
		CESN	0.75503	0.62849	0.68598	0.15119	0.89549
		Capstone	0.31156	**0.97245**	0.47193	0.30469	0.61426

Table 3: Result of 256-bytes sequences

DataSet	Address	Model	Precision	Recall	F1 Score	PTrueDisasm	Accuracy
Compiler-generated	500-0a00	CGG	**0.59291**	0.90079	**0.71512**	0.146783	**0.93066**
		MCG	0.55409	0.60943	0.58044	0.10626	0.91488
		CESN	0.63157	0.80850	0.70916	0.12368	0.93593
		Capstone	0.30723	**0.91284**	0.45973	0.04531	0.79614
	1000-1500	CGG	**0.63912**	0.93114	**0.75798**	0.17813	**0.92730**
		MCG	0.55887	0.59933	0.57839	0.13112	0.89317
		CESN	0.66697	0.86683	0.75388	0.15891	0.93080
		Capstone	0.35306	**0.98702**	0.52008	0.44609	0.78074
	1500-1a00	CGG	**0.65806**	**0.91224**	**0.76458**	0.18246	**0.92606**
		MCG	0.58377	0.61304	0.59805	0.13822	0.89154
		CESN	0.68196	0.82661	0.74734	0.15954	0.92643
		Capstone	0.37557	0.88131	0.52670	0.34609	0.79706
Hand-written	100-600	CGG	0.76897	0.66059	0.71068	0.18121	0.88654
		MCG	**0.81799**	0.91563	**0.86406**	0.23612	**0.93923**
		CESN	0.74725	0.71572	0.73115	0.20204	0.88897
		Capstone	0.44129	**0.99570**	0.61154	0.28516	0.74170
	500-0a00	CGG	0.79273	0.71524	0.75200	0.14990	0.92162
		MCG	**0.82780**	0.87857	**0.85243**	0.17633	**0.94946**
		CESN	0.76902	0.69338	0.72925	0.14980	0.91446
		Capstone	0.28077	**0.99273**	0.43773	0.29219	0.58906

Table 4: Result of 1,280-bytes sequences

For the identification and analysis of binary byte sequences, it is clear from the experiments that the method of this paper is better than the direct disassembly of binary byte sequences using Capstone in general. Due to the mixture of code and data, the direct disassembly work leads to low accuracy, and a large amount of data is disassembled into assembly code incorrectly. The proposed method can effectively distinguish between code and data. It has a better accuracy in general.

Table 5 lists PTrueDisasm of CESN on test set. In real-world situations, since it is not known whether the input byte sequences belong to compiler-generated, hand-written-generated or data, CESN is used for testing and analysis.

	All bytes	256-bytes	1,280-bytes
Compiler-generated	**0.36875**	0.16438	0.14738
Hand-written	0.25383	**0.20933**	**0.17592**
Data	0.05337	0.05337	0.05337

Table 5: PTrueDisasm of CESN on test set

4.3 Discussion

4.3.1 Disassembly

Table 2 and Table 3 shows that the accuracy is about 87% and the F1 score is 79% in the identification of the instructions of the complete code segment; the accuracy is about 91% and the F1 score is 71% in the identification of the disassembly instructions of the code segment. The accuracy calculates the number of correctly identified nodes among all nodes. In the real scenario, the recognition as FalseDisasm does not have analytical significance, so in the F1 score calculation, only the result of TrueDisasm is calculated.

CESN is not ideal for the recognition of disassembly instructions for complete code segments and code fragments, which is caused by the more similar code and data. File where.exe was randomly selected for detailed analysis. The results were divided into the following two cases.

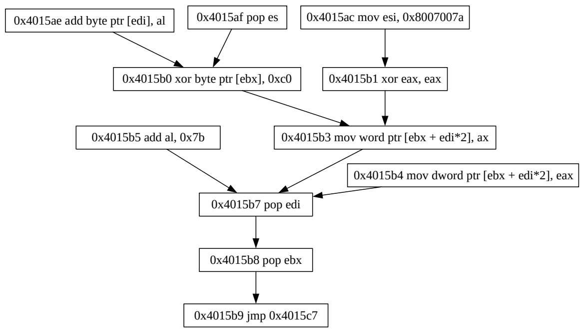
4.3.1.1 FalseDisasm is predicted as TrueDisasm

In Fig. 5, code between 0x4015AC and 0x4015B9 is the true disassembly. CESN determines that the corresponding instructions at all addresses in the figure are correct. However, there is a misjudgment at 0x4015B0 and 0x4015B4. The byte 80 33 C0 at 0x4015B0 is identified as instruction “xor byte ptr [ebx], 0xc0”, and the byte 89 04 7B at 0x4015B4 is identified as instruction “mov dword ptr [ebx + edi*2], eax”. This is probably due to the fact that the instruction at 0x4015B0 is similar to the instruction at 0x4015B1, and the instruction at 0x4015B4 is similar to the instruction at 0x4015B3. This can be corrected according to the control flow.

In Fig. 6, the true disassembly is between 0x4015E0 to 0x4015EA and it is judged as TrueDisasm. This may be due to the existence of similar instructions in the training set that are all TrueDisasm, leading to a similar situation in the test set where FalseDisasm is judged as TrueDisasm. Since the instruction `pop ebp` does not have a parent in the

```
0x4015AC BE 7A 00 07 80     mov     esi, 8007007Ah
0x4015B1  33 C0              xor     eax, eax
0x4015B3  66 89 04 7B        mov     [ebx+edi*2], ax
0x4015B7  5F                 pop     edi
0x4015B8  5B                 pop     ebx
0x4015B9  EB 0C              jmp     short loc_4015C7
```

(a) Real Disassembly

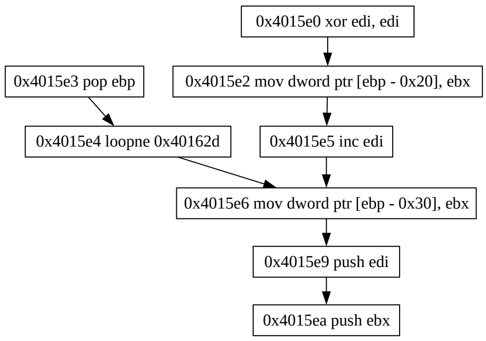


(b) ECFG

Figure 5: FalseDisasm is predicted as TrueDisasm in where.exe

```
0x4015E0 33 FF        xor     edi, edi
0x4015E2 89 5D E0     mov     [ebp+var_20], ebx
0x4015E5 47           inc     edi
0x4015E6 89 5D D0     mov     [ebp+var_30], ebx
0x4015E9 57           push    edi
0x4015EA 53           push    ebx
```

(a) Real Disassembly



(b) ECFG

Figure 6: FalseDisasm is predicted as TrueDisasm in where.exe

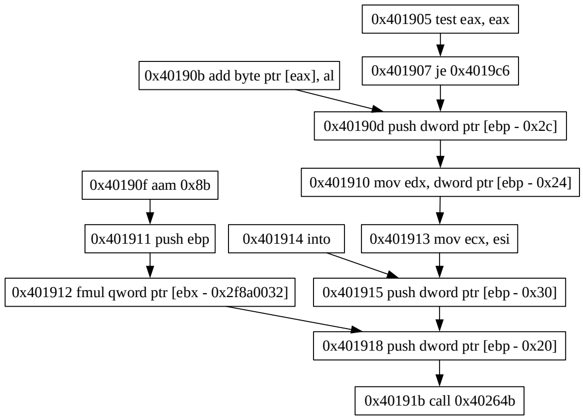
control flow, and the target of `loopne` (0x40162d) is FalseDisasm at both the real case and the predicted result, some heuristic rules could be applied to remove the false instruction at 0x4015E3 and 0x4015E4.

4.3.1.2 TrueDisasm is predicted as FalseDisasm

In Fig. 7, the true disassembly is between 0x401905 and 0x40191B. However, in the prediction result of CESN, the instruction at 0x401913 is discriminated as FalseDisasm. This may be due to the fact that there are fewer consecutive `mov` instructions in the training set and the model is not sensitive to the address articulation.

```
0x401905 85 C0                test    eax, eax
0x401907 0F 84 B9 00 00 00    jz      loc_4019C6
0x40190D FF 75 D4             push    [ebp+var_2C]
0x401910 8B 55 DC             mov     edx, [ebp+var_24]
0x401913 8B CE                mov     ecx, esi
0x401915 FF 75 D0             push    [ebp+var_30]
0x401918 FF 75 E0             push    [ebp+var_20]
0x40191B E8 2B 0D 00 00       call    _FindforFileRecursive@20
```

(a) Real Disassembly



(b) Extended Control Flow Graph

Figure 7: TrueDisasm is predicted as FalseDisasm in where.exe

In Fig. 8, the real disassembly is between 0x401B10 and 0x401B33. However, the model judges the instructions at 0x401B26, 0x401B2B, and 0x401B2E as data. It is probably due to the fact that the bytes corresponding to these instructions in the model training set are data. In a real scenario, it is not possible to have a broken control flow.

4.3.2 Compiler-generated and hand-written data set

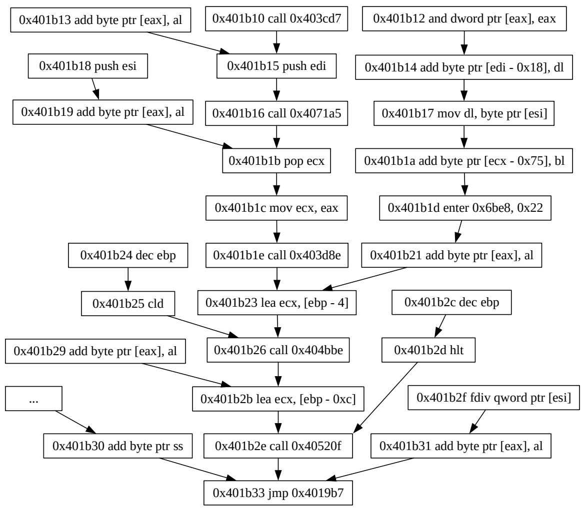
Executables, which are built with different compilers, optimization options, and architectures, contain the features belonging to the compilers. Common approaches are currently based on these features for disassembly, function identification or similarity matching. However, these features may not work well for assembly programs which may not have standard features. The proposed method is not based on the head and tail features of functions, and can effectively disassemble the hand-written executables. As shown in Table 2, Table 3, and Table 4, the accuracy of CESN in hand-written reaches 87% and the F1 score is 74%. The accuracy in the code fragment reaches 88% and the F1 score is 71%.

4.3.3 Code and data distinction

CESN can disassemble any byte sequence and identify code and data in it. For an ECFG, the instructions in the graph contain true and false instructions. Among them, in the

```
0x401B10 E8 C2 21 00 00    call   _SaveLastError@0
0x401B15 57                push   edi
0x401B16 E8 8A 56 00 00    call   ___acrt_iob_func
0x401B1B 59                pop    ecx
0x401B1C 8B C8             mov    ecx, eax
0x401B1E E8 6B 22 00 00    call   _ShowLastErrorEx@8
0x401B23 8D 4D FC          lea    ecx, [ebp+var_4]
0x401B26 E8 93 30 00 00    call   _FreeMemory@4
0x401B2B 8D 4D F4          lea    ecx, [ebp+var_C]
0x401B2E E8 DC 36 00 00    call   _DestroyDynamicArray@4
0x401B33 E9 7F FE FF FF    jmp    loc_4019B7
```

(a) Real Disassembly



(b) ECFG

Figure 8: TrueDisasm is predicted as FalseDisasm in where.exe

complete code segment, about 30% instructions are true. In the code segment, about 20% instructions are true. In the data, there is no disassembly code, i.e., the number of instructions should be 0.

Table 5 shows that CESN can distinguish more obviously whether a byte sequence belongs to code or data. In the real scenario, for a byte sequence, if there are more than 10% TrueDisasm, it can be considered that the byte sequence contains code.

5 Conclusions and future work

In this paper, a new disassembly method is proposed. First, by disassembling each byte of the byte sequence as the first address in turn, an ECFG is constructed for the disassembly results, and the graph nodes are classified using a graph attention network to obtain the correct CFG. The experimental results show that the proposed approach can effectively disassemble binary code sequences and provide a new way of thinking to distinguish code and data, which possesses good applicability even in the face of complex data sets with complex data and different code writing styles.

However, there are three shortcomings. First, the proposed method cannot identify indirect jumps. It can be considered to introduce dynamic analysis or use link prediction in graph neural network to find the addresses of indirect jumps. Second, disassembly rules could be used to optimize the graph neural network after it classifies nodes. The rules include data segments must be preceded by jump instructions, functions end with jump, return, call instructions, etc. These rules filter out the misclassified nodes. Finally, the bytes is used as node features, ignoring the relationship between bytes. In future, embedding could be used to obtain a larger feature space.

Acknowledgment

This work is supported by National Natural Science Foundation of China 61702140, Heilongjiang Provincial Natural Science Foundation of China F2018017, the Fundamental Research Foundation for Universities of Heilongjiang Province LGYC2018JC015, and Zhejiang A&F University Research Development Fund Talent Initiation Project 2021LFR048.

References

[Anand et al., 2013] Anand, K., Smithson, M., Elwazeer, K., Kotha, A., Gruen, J., Giles, N., and Barua, R. (2013). A compiler-level intermediate representation based binary analysis and rewriting system. In EuroSys ’13.

[Andriesse et al., 2016] Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., and Bos, H. (2016). An in-depth analysis of disassembly on full-scale x86/x64 binaries. In 25th USENIX Security Symposium (USENIX Security 16), pages 583–600.

[Bauman et al., 2018] Bauman, E., Lin, Z., and Hamlen, K. W. (2018). Superset disassembly: Statically rewriting x86 binaries without heuristics. In Proceedings 2018 Network and Distributed System Security Symposium. Internet Society.

[Ben Khadra et al., 2016] Ben Khadra, M. A., Stoffel, D., and Kunz, W. (2016). Speculative disassembly of binary code. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems. ACM.

[Besson et al., 2001] Besson, F., Jensen, T. P., and Métayer, D. L. (2001). Model checking security properties of control flow graphs. J. Comput. Secur., 9:217–250.

[Cui et al., 2019] Cui, Z., Du, L., Wang, P., Cai, X., and Zhang, W. (2019). Malicious code detection based on cnns and multi-objective algorithm. J. Parallel Distributed Comput., 129:50–58.

[Cui et al., 2018] Cui, Z., Xue, F., Cai, X., Cao, Y., ge Wang, G., and Chen, J. (2018). Detection of malicious code variants based on deep learning. IEEE Transactions on Industrial Informatics, 14:3187–3196.

[Djoudi and Bardin, 2015] Djoudi, A. and Bardin, S. (2015). Binsec: Binary code analysis with low-level regions. In TACAS.

[Federico and Agosta, 2016] Federico, A. D. and Agosta, G. (2016). A jump-target identification method for multi-architecture static binary translation. 2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES), pages 1–10.

[Flores-Montoya and Schulte, 2020] Flores-Montoya, A. and Schulte, E. (2020). Datalog disassembly. In 29th USENIX Security Symposium (USENIX Security 20), pages 1075–1092.

[Goodfellow et al., 2015] Goodfellow, I. J., Bengio, Y., and Courville, A. C. (2015). Deep learning. Nature, 521:436–444.

[Guo and Wang, 2021] Guo, Z. and Wang, H. (2021). A deep graph neural network-based mechanism for social recommendations. IEEE Transactions on Industrial Informatics, 17:2776–2783.

[Harris and Miller, 2005] Harris, L. C. and Miller, B. P. (2005). Practical analysis of stripped binary code. SIGARCH Comput. Archit. News, 33:63–68.

[Kruegel et al., 2004] Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. (2004). Static disassembly of obfuscated binaries. In USENIX security Symposium, volume 13, pages 18–18.

[Li et al., 2020] Li, K., Woo, M., and Jia, L. (2020). On the generation of disassembly ground truth and the evaluation of disassemblers. In Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation, FEAST'20, page 9–14, New York, NY, USA. Association for Computing Machinery.

[Liu et al., 2013] Liu, K., Tan, H. B. K., and Chen, X. (2013). Binary code analysis. Computer, 46:60–68.

[Ma et al., 2020] Ma, Z., Ge, H., Wang, Z., Liu, Y., and Liu, X. (2020). Droidetec: Android malware detection and malicious code localization through deep learning. ArXiv, abs/2002.03594.

[Miller et al., 2019] Miller, K., Kwon, Y., Sun, Y., Zhang, Z., Zhang, X., and Lin, Z. (2019). Probabilistic disassembly. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE.

[Pei et al., 2021] Pei, K., Guan, J., Williams-King, D., Yang, J., and Jana, S. (2021). XDA: Accurate, robust disassembly with transfer learning. In Proceedings 2021 Network and Distributed System Security Symposium. Internet Society.

[Rong et al., 2020] Rong, Y., Huang, W., Xu, T., and Huang, J. (2020). Dropedge: Towards deep graph convolutional networks on node classification. In ICLR.

[Rosenblum et al., 2008] Rosenblum, N. E., Zhu, X., Miller, B. P., and Hunt, K. (2008). Learning to analyze binary computer code. In AAAI.

[Scarselli et al., 2009] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. IEEE Transactions on Neural Networks, 20:61–80.

[Song et al., 2008] Song, D. X., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. (2008). Bitblaze: A new approach to computer security via binary analysis. In ICISS.

[Velickovic et al., 2018] Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio', P., and Bengio, Y. (2018). Graph attention networks. ArXiv, abs/1710.10903.

[Wang et al., 2019a] Wang, X., He, X., Cao, Y., Liu, M., and Chua, T.-S. (2019a). Kgat: Knowledge graph attention network for recommendation. Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.

[Wang et al., 2019b] Wang, X., Ji, H., Shi, C., Wang, B., Cui, P., Yu, P., and Ye, Y. (2019b). Heterogeneous graph attention network. The World Wide Web Conference.

[Wartell et al., 2011] Wartell, R., Zhou, Y., Hamlen, K. W., Kantarcioglu, M., and Thuraisingham, B. (2011). Differentiating code from data in x86 binaries. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pages 522–536. Springer.

[Zhang et al., 2007] Zhang, Q., Reeves, D. S., Ning, P., and Iyer, S. P. (2007). Analyzing network traffic to detect self-decrypting exploit code. In Proceedings of the 2nd ACM symposium on Information, computer and communications security, pages 4–12.