o-glassesX: Compiler Provenance Recovery with Attention Mechanism from a Short Code Fragment

Yuhei Otsubo*†, Akira Otsuka†, Mamoru Mimura‡†, Takeshi Sakaki§, and Hiroshi Ukegawa*
*National Police Agency, Tokyo, Japan
†Institute of information Security, Kanagawa, Japan
‡National Defense Academy, Kanagawa, Japan
§The University of Tokyo, Tokyo, Japan

***Abstract*—Program binaries are often the focus of forensic investigations. Extracting compiler provenance is important in digital forensics, as it provides crucial information about the process by which a malware binary is produced. Object files and libraries, which are components of an executable file, are often compiled in various environments. Thus the executable file bears important information about the development environment of the authors. It is expected to reveal valuable information of the authors if we could extract those compiler provenances with high accuracy enough to resolve each component. In this paper, we investigate the feasibility of compiler identification from a very short code fragment. Surprisingly, we found that a randomly sampled code fragment consisting of only 16 sequential instructions is sufficient to identify the program provenance, such as the source compiler family and the optimization level of machine code efficiently. Additionally, our method can visualize how much each instruction contributed to the classification result. The model uses the attention mechanism, widely used in natural language processing, and CNN with a local receptive field specialized for recognition of one instruction. Moreover, our method takes both opcodes and operands as input, while many of the existing methods had omitted operands. We used a dataset that consists of 28,074 binaries developed in 19 different classes. Each class corresponds to a combination of five different compilers, two different optimization levels (none/max), and two different CPU architectures (x86/x86-64). With this dataset, our method identifies one of the 19 classes with 0.956 accuracy, which is a competitive performance with the best-known results, from only 16 instructions or an equivalent binary sequence of length 30-60 bytes. This result is satisfactory in resolving the component-wise compiler provenance of an executable binary. Additionally, the attention mechanism allows us to extract typical instructions generated by each compiler automatically. The attention mechanism is useful not only for improving the classification performance but also for understanding the characteristics of each class.**

I. Introduction

Program binaries are often the focus of forensic investigations [25], [29], [26], covering numerous issues from copyright infringement [31] to malware analysis [3], [2]. Program binary analysis is a challenging task due to the absence of high-level information, which is found in source code, and the myriad variations in compiler provenance. The more general term, program provenance refers to information on how the target binary is developed, such as tools, libraries used, and their versions. Compiler provenance is a part of program provenance, consists of compiler family, compiler version, optimization level, and other compiler-related information. Compiler provenance gives us many clues on the program provenance. Extracting compiler provenance is important in program provenance digital forensics, as it provides crucial information about the process by which a program (*e.g.*, malware binary is produced. When presented with only a program binary or a snippet of binary code, the details of program provenance are not readily apparent. The black box between the program author and the program binary affords little foothold for simple tools or analyses.

According to existing research [28], [21], [4], they can identify hundreds of authors with high accuracy from program binaries. Although those results are scientifically impressive, this approach faces a big problem if they are applied in some real forensics scenario such as malware analysis. First, they require to prepare many binary code samples authored by all possible malware developers with true identity as they are based on supervised learning. Second, single-author identification is not enough as malware is usually composed of many binary snippets authored by different developers. Thus, author identification must be improved with a combination of extra information such as binary chunk identification with authorship difference. Compiler provenance is one of the crucial clues to identify such binary chunks.

In this paper, we propose a novel method, named o-glassesX, that incorporates the attention mechanism to achieve high performance for identifying the source compiler of program provenance. Our method can calculate how much each instruction contributed to the identification result and requires only a sequence of code for compiler provenance. Hence, our method does not need meta-data or other details of program headers. It is applicable even when such information has been stripped or is otherwise unavailable. Additionally, it can be used even when codes produced by multiple compilers coexist within a program binary, such as statically linked library code.

In summary, the main contributions of our approach are as follows:

- **Fine-grained compiler provenance from stripped binary.** Our classifier does not use any meta-data,

Workshop on Binary Analysis Research (BAR) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-62-2
https://dx.doi.org/10.14722/bar.2020.23001
www.ndss-symposium.org

but it extracts the compiler provenance from a short sequence of code in a stripped binary. Thus, it can extracts compiler provenance with high accuracy even in such complex programs. Our result shows that it bears enough information to identify compiler provenance that only 30-60 bytes of consecutive binary sequence sampled from any part of the executable code.

- **Proposal of a model that can calculate how much input data contributes to output in units of instructions.** In this paper, we apply an attention mechanism, widely used in natural language processing in recent years, to code fragment recognition. This model can calculate the degree of the contribution to the identification result on each instruction.
- **Knowledge gained from model judgment grounds.** By counting the appearance frequency of instructions that contributed to the compiler identification results, we can extract the typical instructions in each class automatically. These instructions help us to understand the characteristics of each class.

The remainder of this paper is structured as follows. In Sec. II, we first briefly describe a review of related work. In Sec. III, we describe some essential techniques for our method. In Sec. IV, we present our approach. In Sec. V, we evaluate the proposed approach and provide comparisons against existing work. In Sec. VI, we discuss about our approach. In Sec. VII, we finally present some concluding remarks on this work with a discussion of future research.

II. Related Work

Binary Analysis allows software engineers to analyze binary executables directly without access to source code. Tesauro *et al.* [33] first introduced a neural network-based method for recognizing virus in application binaries. Since then, machine learning-based binary analysis has become an important research topic in vulnerability detection [24], [38], [7], [11], [36], [5], [8], function recognition [27], [30], [2], [32], [35], and other areas.

In this section, we describe some methods about compiler provenance and a binary analysis method using attention mechanisms.

A. Compiler provenance

Several compiler identification tools are already available (*e.g.*, `IDA Pro`[1], `PEiD`[2], and `RD`[3]). These tools are roughly signature-based and typically relies on meta-data or other details in the program header. Their exact matching algorithm may fail if even a slight difference between signatures is present or if the header information has been stripped or is otherwise unavailable.

Rosenblum *et al.* used a conditional random field (CRF [20]) and set up a classifier which takes binary data as input to identify one of the three compiler families with 0.925 accuracy rate [29]. They disassemble the binary with IA-32 architecture and find typical matching instruction pattern called "idioms" to predict the compiler families. Their early result is only for the relatively easy compiler family identification with 3 classes. Rosenblum *et al.* have improved the above method and proposed a tool named `ORIGIN` [26]. `ORIGIN`'s SVM (linear support vector machines [6]) takes the feature of an independent "function" as input to predict the optimization level and the version in addition to the compiler family but faces difficulty in identifying the compiler versions. `ORIGIN`'s CRF takes the same features of multiple adjacent functions as input for the prediction and performs with better accuracy of 0.9 and above despite that the number of classes has increased from 3 to 18. However, the size of the input data required for compiler provenance is larger.

Rahimian *et al.* developed `BinComp` [25], an approach in which they analyze the syntax, structure, and semantics of disassembled functions to extract the compiler provenance. `BinComp` has an identification accuracy of 0.801 in 8 classes classification in their experiments.

The common point in these existing studies is that it is necessary to pay attention to the function structure rather than mere instruction sequence when trying to identify the optimization level and version in addition to the compiler family. We show that we can identify not only the compiler family but also the optimization level and version with high accuracy by merely analyzing instruction sequences without paying attention to functions.

B. Binary Analysis using attention

Yakura *et al.* proposed a method for detecting a region specific to some malware family by applying a CNN incorporating an attention mechanism to the imaged binary [37]. The input data is an image scaled down to a resolution of 64×64, and it outputs an attention level map indicating the region essential for classification in the image. As a result, the attention level map shows the region specific to the malware family, and it is expected to help analysis. However, the purpose of this method is malware family identification, and the input of the method requires the whole of an executable file. Thus, the method cannot achieve our goal of performing compiler provenance recovery from binary fragments.

III. Preliminaries

`o-glassesX` is an extension of `o-glasses` [23], which is a program snippet detection tool based on a convolutional neural network (CNN [18]). This method classifies the input block as either program-code or non-code. `o-glasses` can classify whether the input binary sequence is a program with a very high accuracy rate. `o-glasses` is designed to capture features of a single instruction by a corresponding multiple local receptive fields in the first CNN layer. `o-glassesX` follows this design strategy as well but with attention mechanisms so that it can tackle the more difficult problem of program provenance recovery.

For the sake of this, we used a Natural Language Processing (NLP) techniques called the attention mechanism in `o-glassesX`. In this section, we describe some preliminaries; First, we describe the essence of x86/x86-64 architecture, which is our target architecture. Next, we explain CNN with

[1]https://www.hex-rays.com/products/ida/
[2]https://www.aldeid.com/wiki/PEiD
[3]http://www.rdgsoft.net/

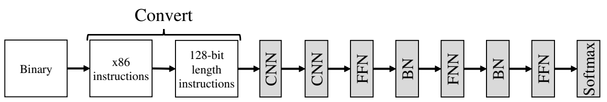


Fig. 1. Outline of `o-glasses`

a particular focus on the local receptive field. Then, we describe `o-glasses` briefly. Finally, we describe the attention mechanism.

A. x86/x86-64 Architecture

In our experiment discussed in later sections, we mainly targeted x86 and x86-64 architectures [13]. Note that, however, our method is not restricted to x86/x86-64 architectures but can be applied to other architectures such as arm64.

x86/x86-64 instruction sets are rich and complex, and most importantly, they support instructions of varying length. According to the specification of the x86/x86-64 architecture [13], instruction lengths range from just one byte (*i.e.*, instructions comprising just a one-byte opcode) to 15 bytes. Although 15 bytes is the basic maximum length of instruction, longer instructions could appear in theory (particularly when the file being interpreted as x86/x86-64 machine code is non-code) [4]

B. CNN

Tools based on CNN have now led to great results in a wide range of vision tasks [17]. Each unit in CNN has specially local connections to the input units, called a *kernel* or a *local reception field*. Every kernel shares the weight parameters with the others in the same layer so that one can greatly reduce the number of parameters in the network.

Several hyperparameters control the size of the output volume of the convolutional layer: the kernel size, depth, and stride. The depth (D) of the output volume controls the number of neurons in a layer that connect to the same region of the input volume. The stride (S) controls how depth columns around the spatial dimensions (width and height) are allocated.

The spatial size of the output volume can be computed as a function of the input volume W, the kernel size K, and the stride S. The output volume is given by $(W - K)/S + 1$.

C. `o-glasses`

`o-glasses` is written in Python, and uses the Chainer[5] 4.0.0 framework, which is a flexible framework for neural networks.

The whole network is shown in Fig. 1. The 1-bit change often causes the behavior of the machine code to be another one. Thus, we should design the neural network carefully, in the case of treating discrete data such as machine code; *e.g.*, pre-processing, parameter. The details of `o-glasses`' implementation is as follows.

1) Pre-processing: `o-glasses` disassembles the input binary assuming they consist of x86/x86-64 instructions. Each instruction is converted into N-bit fixed-length instructions padded with constant bits. They serialized a set of 16 fixed-length instructions into an array of 2048 bit values as input data.

2) CNN for recognizing instruction: Generally, the local receptive field successfully captures the local features in image recognition, where the values of adjacent pixels are strongly correlated. In the case of `o-glasses`, Otsubo *et al.* designed the first convolutional layer to capture the feature of the whole bits in every single instruction and the second layer to capture the local distribution of instruction features. This design principle worked surprisingly well against program provenance recovery as empirically shown in the later sections.

The first convolutional layer (Bit-CNN) takes 2048 bits as input; each unit has 1-dimensional 128-unit kernel with the stride of 128 and the depth of 96. The kernel field size and the stride of the first layer are set to N so that each kernel covers a single instruction. The second layer is also a convolutional layer (Instruction-CNN). They applied 256 2-filters to a 16×96 input volume with a stride of 1. They expect that the second layer will obtain the features of the relationship between two adjacent instructions.

3) Other network configurations: The 3rd to the 5th layers are fully connected MLP (multi-layer perceptron). Their output volumes are 400, 400, and K, respectively. K is the number of classes. Batch normalization [14] layers are inserted after the 1st and 2nd fully connected layers to speed up and stabilize the learning process. The activation function in each intermediate layer is ReLU [9], and that of the final layer of MLP is softmax.

They used the well-known stochastic gradient descent (SGD) method to minimize the error function in the back-propagation algorithm.

D. Existing NLP techniques

1) Attention: Bahdanau *et al.* first proposed the attention mechanism [1].The attention mechanism uses a vector (attention weight, AttW) that focuses only on a portion of the input information, not the entire input information. In 2017, Vaswani *et al.* proposed `Transformer` [34], which uses only the attention mechanism without using Recurrent Neural Network (RNN [12]), and showed surprisingly good performance in translation with a little training. Since this proposal, research using the attention mechanism has become very active in the field of NLP. Additionally, the model applying the attention mechanism is not only high in performance, but also is possible to infer essential input data that is the basis of the output result by visualizing AttW.

There are two types of attention mechanisms, depending on how to obtain AttW. One is Additive Attention [1], and another is Dot-Product Attention (or Multiplicative Attention) [19]. Additive Attention calculates the AttW in the hidden layer Feed-Forward Network (FFN). Dot-Product Attention calculates the AttW by an inner product. Dot-Product Attention is generally faster because it does not require parameters. Thus, we use Dot-Product Attention.

[4]The following sentence appears in the specification.

Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).

[5]https://chainer.org/

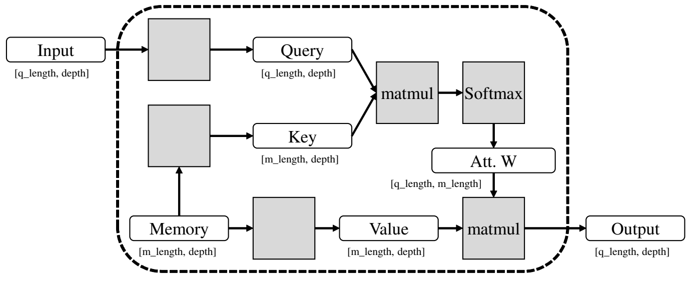


Fig. 2. Outline of Dot-Product Attention.

Fig. 2 shows the basic structure of Dot-Product Attention. The squares in the figure indicate processing such as FFN described later, and the rounded squares indicate tensors. This processing, such as FFN, transform the attention input into $\mathbf{Query}$ ($\boldsymbol{Q}$, search query). There is $\mathbf{Memory}$ in the hidden layer in the attention mechanism. The processing, such as FFN, transforms the $\mathbf{Memory}$ into $\mathbf{Key}$ ($\boldsymbol{K}$) and $\mathbf{Value}$ ($\boldsymbol{V}$). The following equation defines the attention mechanism.

$$\mathrm{Att}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \mathrm{softmax}\left(\boldsymbol{Q}\cdot\boldsymbol{K}^T\right)\cdot\boldsymbol{V}, \tag{1}$$

where $\boldsymbol{Q}$ is a matrix (generally a tensor) composed of d_k-dimensional vectors $q_i \quad (i = 1,\dots)$. Similarly, $\boldsymbol{K}$ and $\boldsymbol{V}$ are tensors composed of vectors of d_k and d_v dimensions, respectively. In particular, the output of $\mathrm{softmax}(\cdot)$ when the expression (1) is applied to one search query vector $\boldsymbol{q} \in \mathbf{Query}$ is called Attention Weight (AttW) . The following equation defines AttW.

$$\mathrm{AttW}(\boldsymbol{q},\boldsymbol{K}) = \mathrm{softmax}\left(\boldsymbol{q}\cdot\boldsymbol{K}^T\right). \tag{2}$$

At this time, $\mathbf{Memory}$ functions as an array of pairs in which each $\boldsymbol{k}$ and each $\boldsymbol{v}$ have a one-to-one correspondence, that is, a dictionary object. The dot product of $\boldsymbol{q}$ and $\boldsymbol{K}$ measures the similarity between $\boldsymbol{q}$ and each $\boldsymbol{k} \in \boldsymbol{K}$. The Attention Weight ($\mathrm{AttW}$) normalized by softmax represents the position of k that is most similar to q. The inner product of AttW and $\boldsymbol{V}$ extracts the value corresponding to the position of $\boldsymbol{k}$ as a weighted sum. In other words, the attention mechanism searches for the key ($\boldsymbol{k}$) that matches the search query ($\boldsymbol{q}$) and retrieves the corresponding value ($\boldsymbol{v}$). This behavior is the same as the function of a dictionary object.

Additionally, there are two types of Dot-Product Attention, depending on where $\mathbf{Memory}$ comes from. When the Attention input ($\mathbf{Input}$) and $\mathbf{Memory}$ match, it is called Self-Attention, and when it does not match, it is called Source Target Attention. Encoder (Classifier) often uses Self-Attention, and Decoder often uses Source Target Attention. Thus, we use Self-Attention to classify binary sequences.

2) Position-wise FFN (CNN): Position-wise Feed-Forward Network (PFFN) is a network used in `Transformer`. PFFN performs FFN processing independently for each position of a word sequence. PFFN can be implemented even by CNN with parameter adjustment, and it is expected to form a local receptive field specialized for recognition of one word. In this paper, it is expected to form a local receptive field specialized in recognition of one instruction by applying it to machine language instructions. This idea is the same as what was called bit-CNN by the authors of `o-glasses`[23].

The parameters for implementing PFFN in CNN is, for example, if the N bit fixed-length instructions are arranged in one dimension, the size of the kernel and stride are N, which is the same as the length of one instruction. In the case of PFFN's input is a two-dimensional vector in which multiple instruction vectors, we can implement PFFN by setting the size of the kernel and stride to 1. In our proposed model, CNN is implemented by adjusting parameters and implementing PFFN.

3) Positional Encoding: The above mechanism alone can use combinations of instructions in the code for learning, but cannot use the order of instructions for learning. We introduce Positional Encoding (PE) to add information on the order of instructions (relative or absolute position of instructions). There are various implementation methods for PE. In this paper, we adopt a method built into `Transformer`, and we expect the method to suitable with FFN. PE function adds the constant matrix PE to PE's input. The following equation defines the processing of PE.

$$\mathrm{PE}(\boldsymbol{X}) = \boldsymbol{X} + \alpha PE, \tag{3}$$

where α is the addition rate of PE. The followings are each component of PE.

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}) \tag{4}$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}), \tag{5}$$

where pos is the position of the instruction, d_{model} is the channel depth of the PE's input $\boldsymbol{X}$, $2i$ and $2i+1$ are the component dimensions (0 to $d_{model}-1$). This expression can represent PE_{pos+k} with the linear function of PE_{pos}. In FFN, the linear function of the previous layer can express the next layer. Therefore, we expect the PE method is well suitable with FFN that expresses.

IV. Proposed method

In this paper, we propose the following three.

- A deep learning network that can classify machine language instruction sequences with high accuracy and calculate the instruction-wise contribution to the classification results (IV-A)
- Carefully designed the attention mechanism that calculates the contribution of each instruction in a sequence to the classification result (IV-B)
- Automatic extraction of representative instructions which are typically found in each class based on the above the attention mechanism (IV-C)

A. Network model

We tried to design our attention model as simple as possible so that it can increase the effect of Attention Weight (AttW) on the Attention Output (Att. Output). Fig. 3 gives an overview of our attention model. The input data is as same as `o-glasses`' one, a $128 \times L$ bit value array which contains L instructions converted to 128-bit fixed-length instructions.

The first layer is CNN, which parameters are the same as `o-glasses`' parameters. We expect this layer to learn the

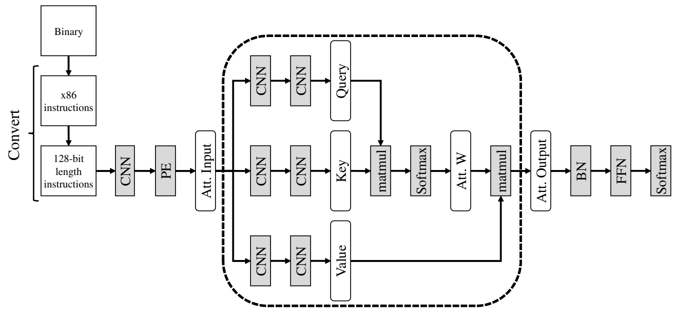


Fig. 3. Outline of our model.

feature of instruction structure by setting the kernel field size and the stride size to 128, which is the same as the instruction length. The output channel depth is 96.

The second layer is Positional Encoding (PE), and we described it in detail at III-D3. PE adds information on the order of instructions.

The third layer is the attention block, which is the block surrounded by the dotted line in Fig. 3. CNN transform the attention input to **Query**, **Key**, and **Value**. We set these CNN's parameters in line with the idea of Position-wise Feed-Forward Network (PFFN). We described PFFN in detail at III-D2. Each CNN has an independent weight. We set the depth of the first and second channel to $d_{model} \times 4$ and d_{model}, where d_{model} is the depth of the attention input, 96 in this case.

Immediately after the output of the attention block, we place the Batch Normalization (BN [14]) layer to stabilize and speed up the learning process. The last layer is fully-connected, and the number of nodes is K, where K is the number of classes to classify. The activation function of the intermediate network layer is ReLU [9], and the activation function of the output layer is the softmax function.

B. Calculating 'Why' with the attention mechanism

What is important to know in compiler provenance recovery is instruction-wise occurrences and their order which are typically found in each compiler's output codes. For calculating the contribution to the identification result one instruction by one instruction, we designed our network carefully. As described in III-D1, the attention weight **AttW** is computed by $\mathrm{softmax}$ of the product of a search query matrix $\boldsymbol{Q}$ and a memory matrix $\boldsymbol{M}$, where $\boldsymbol{Q}$ is a tensor composed of d_k dimensional vectors $\boldsymbol{q_i}$. The PFFN (Position-wise Feed Forward Network) transforms the input vector of machine instructions to a search query matrix $\boldsymbol{Q}$ such that its each search query vector ($\boldsymbol{q_i}$) only depends on one instruction additionally with its position information. Hence, each element in the attention weight matrix **AttW** corresponds to only one pair of machine instructions in its search query vector and in its memory vector in a different position. Therefore, the $\mathrm{softmax}$ operation takes the most significant machine instruction in the input vector contributed to the identification result.

An example visualization of **AttW** is shown in Fig. 4. **AttW** is a two-dimensional map corresponding to the combination of instructions of **Input** and **Memory**. **Memory** is the same as **Input** because we use the Self-Attention. For simplicity, in this paper, we use L_2-norm for calculating each instruction's output contribution. Denoted by $\mathrm{L2}_i$ we visualize the magnitude of the contribution of the i-th instruction to the classification result, and $\mathrm{L2}_i$ is given by the following equation.

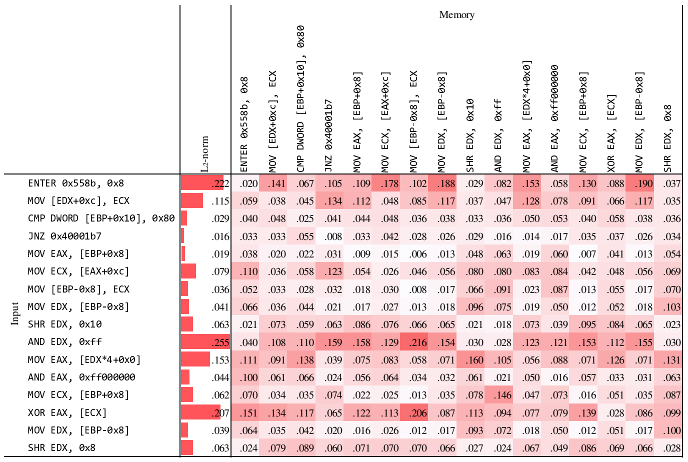
$$\mathrm{L2}_i = \|\, \boldsymbol{AttW}(\boldsymbol{q_i}, \boldsymbol{K}) \,\|_2^2 \quad (6)$$
$$= \|\, \mathrm{softmax}(\boldsymbol{q_i} \cdot \boldsymbol{K}^T) \,\|_2^2. \quad (7)$$

In the case of Fig. 4, it can be seen that the value of the squared L_2-norm of the 10th instruction "`AND EDX, 0xff`" is the largest at 0.255 and contributes the most to the output.

C. Automatic feature extraction for each class

It is possible to automatically extract the typical instructions of each class by calculating the output contribute, as described above. Algorithm 1 shows the automatically extracting procedure for compiler-typical instructions.

The outline of Algorithm 1 is as follows. First, we prepare a binary made by the compiler we want to know its typical instructions. Next, we pull out the first L instruction from the binary. We identify the compiler by inputting this code fragment into our model. If the classified result is correct and has high confidence (0.99 or higher), then we determine the instruction that contributes to the identification using the method described in Sec. IV-B. The offset address for compiler



Input	L_2-norm	Memory: `ENTER 0x558b, 0x8`	`MOV [EDX+0xc], ECX`	`CMP DWORD [EBP+0x10], 0x80`	`JNZ 0x40001b7`	`MOV EAX, [EBP+0x8]`	`MOV ECX, [EAX+0xc]`	`MOV [EBP-0x8], ECX`	`MOV EDX, [EBP-0x8]`	`SHR EDX, 0x10`	`AND EDX, 0xff`	`MOV EAX, [EDX*4+0x0]`	`AND EAX, 0xff000000`	`MOV ECX, [EBP+0x8]`	`XOR EAX, [ECX]`	`MOV EDX, [EBP-0x8]`	`SHR EDX, 0x8`
`ENTER 0x558b, 0x8`	.222	.020	.141	.067	.105	.109	.178	.102	.188	.029	.082	.153	.058	.130	.088	.190	.037
`MOV [EDX+0xc], ECX`	.115	.059	.038	.045	.134	.112	.048	.085	.117	.037	.047	.128	.078	.091	.066	.117	.035
`CMP DWORD [EBP+0x10], 0x80`	.029	.040	.048	.025	.041	.044	.048	.036	.038	.033	.036	.050	.053	.040	.058	.038	.036
`JNZ 0x40001b7`	.016	.033	.033	.055	.008	.033	.042	.028	.026	.029	.016	.014	.017	.035	.037	.026	.034
`MOV EAX, [EBP+0x8]`	.019	.038	.020	.022	.031	.009	.015	.006	.013	.048	.063	.019	.060	.007	.041	.013	.054
`MOV ECX, [EAX+0xc]`	.079	.110	.036	.058	.123	.054	.026	.046	.056	.080	.080	.083	.084	.042	.048	.056	.069
`MOV [EBP-0x8], ECX`	.036	.052	.033	.028	.032	.018	.030	.008	.017	.066	.091	.023	.087	.013	.055	.017	.070
`MOV EDX, [EBP-0x8]`	.041	.066	.036	.044	.021	.017	.027	.013	.018	.096	.075	.019	.050	.012	.052	.018	.103
`SHR EDX, 0x10`	.063	.021	.073	.059	.063	.086	.076	.066	.065	.021	.018	.073	.039	.095	.084	.065	.023
`AND EDX, 0xff`	.255	.040	.108	.110	.159	.158	.129	.216	.154	.030	.028	.123	.121	.153	.112	.155	.030
`MOV EAX, [EDX*4+0x0]`	.153	.111	.091	.138	.039	.075	.083	.058	.071	.160	.105	.056	.088	.071	.126	.071	.131
`AND EAX, 0xff000000`	.044	.100	.061	.066	.024	.056	.064	.034	.032	.061	.021	.050	.016	.057	.033	.031	.063
`MOV ECX, [EBP+0x8]`	.062	.070	.034	.035	.074	.022	.025	.013	.035	.078	.146	.047	.073	.016	.051	.035	.087
`XOR EAX, [ECX]`	.207	.151	.134	.117	.065	.122	.113	.206	.087	.113	.094	.077	.079	.139	.028	.086	.099
`MOV EDX, [EBP-0x8]`	.039	.064	.035	.042	.020	.016	.026	.012	.017	.093	.072	.018	.050	.012	.051	.017	.100
`SHR EDX, 0x8`	.063	.024	.079	.089	.060	.071	.070	.070	.066	.027	.024	.067	.049	.086	.069	.066	.028

Fig. 4. Example of visualizing a **AttW**.

Algorithm 1 Automatic feature extraction procedure

```
 1: InputFile ← Compile(SourceFile,Optimization).pullCode()
 2: Offset ← 0
 3: while Offset < InputFile.EndOffset do
 4:     Input ← InputFile.pull(Offset,L)
 5:     Result ← model.predictor(Input)
 6:     if Result.Label = RealLabel then
 7:         if Result.Confidence > 0.99 then
 8:             i ← argsmax(L2(Result.AttentionWeight))
 9:             Count[input[i]] ++
10:         end if
11:     end if
12:     Offset ++
13: end while
14: Print(Ranking(Count))
```

identification is shifted one byte at a time, and the appearance frequency of instructions that contributed to the identification is counted in the same procedure as described above. Finally, we display typical instructions in descending order of appearance frequency.

D. `o-glassesX`*'s implementation*

`o-glassesX` implemented our model is written in Python and uses the Chainer 4.0.0 framework.

Our model contains the Positional Encoding (PE) we described in Sec. III-D3. When additional rate (α) was 1.0, the influence of PE in the initial stage of the learning often have let the learning result a local solution. Then, the generalization performance did not improve. Note that the value of the PE input ($\boldsymbol{X}$) is optimized as learning progresses, so it seems that the value of α does not change the result of the optimal solution. Therefore we set the α 0.01 temporally for improving the generalization performance.

V. Evaluation

In this section, we conduct several experiments to address the following questions:

1 What is the size of the learning sample required for our method? (Sec. V-B)

2 How the performance change with the number of instructions input to our method? (Sec. V-B)

3 Does our method perform compiler identification better than existing methods? (Sec. V-B)

4 Does our method explain characteristic of each class? (Sec. V-C)

5 Can our method analyze an executable file in spite of learning from object files? (Sec. V-D)

6 Can our method explain characteristic of an APT group? (Sec. V-E)

In the following, we first describe our dataset, and then we describe the details of our experiments. The research artifacts are available at the following URL:

https://github.com/yotsubo/o-glassesX

A. Dataset

We chose C/C++ for the experiment from the various program languages. According to the TIOBE Index [6] for August 2019, among the languages that output native code, these languages are at the top. Among all languages, these have gained more than 20 % share. Additionally, we treat binaries as x86/x86-64 architecture code for evaluation experiment because the architecture occupies the PC market.

Our model input data needs the L fix-length instructions. The scale of the dataset changes with the value of L. TABLE I shows an outline of our dataset in the case of $L = 16$. We prepared two categories of dataset for our examination, both of which can be gathered easily. One category is labeled "Program" and comprises various sets of x86/x86-64 machine code generated from randomly sampled source code from Github[7]. The other category is called "Others" and consists of various document files and portions of data extracted from them. Document files contain data of various kinds: metadata, text, and packed data. All of these file areas differ not only in size but also in the level of information entropy. On the other hand, benign document files rarely contain machine code. For each category and source or file type, we constructed two types of the dataset: the whole files, and $(L \times 128)$-bit segments of code extracted from these files.

TABLE I. Overview of our dataset. (Code): the number of the short code fragments.

			Label	#Binaries	#Code
Program	VC2017	x86	VC17,32,none(Od)	1,170	369,605
			VC17,32,max(Ox)	1,147	255,143
		x86-64	VC17,64,none(Od)	1,456	540,568
			VC17,64,max(Ox)	1,242	542,020
	VC2003	x86	VC03,32,none(Od)	1,350	292,277
			VC03,32,max(Ox)	1,306	270,743
		x86-64	-	-	-
			-	-	-
	GCC	x86	GCC,32,none(O0)	2,111	227,004
			GCC,32,max(O3)	1,844	239,821
		x86-64	GCC,64,none(O0)	1,582	283,276
			GCC,64,max(O3)	1,580	287,775
	Clang	x86	Clang,32,none(O0)	1,205	101,024
			Clang,32,max(O3)	1,196	86,521
		x86-64	Clang,64,none(O0)	1,892	332,278
			Clang,64,max(O3)	1,883	246,500
	ICC	x86	ICC,32,none(Od)	1,761	1,494,677
			ICC,32,max(Ox)	1,724	1,161,499
		x86-64	ICC,64,none(Od)	1,796	1,419,705
			ICC,64,max(Ox)	1,728	1,046,958
			Others	101	912,855
			Total	28,074	10,110,249

The methods for making each of our types of dataset are as follows.

1) Binary: The following procedure is conducted for making the "Binary" dataset in the "Program" category.

- Gather various C/C++ source code files from GitHub
- Compile these files into x86/x86-64 object files by using various compilers (TABLE II)
- Extract only the machine code from these object files.

We did not compile the source code files into the executable files but into the object files. The object file is an intermediate data representation file including a machine code which is generated as a result of the compiler processing the source code. On the other hand, the machine code in the executable files contains various library code; *e.g.*, C run-time libraries. It is hard to know the compiler condition of these libraries.

[6]https://www.tiobe.com/tiobe-index/

[7]https://github.com/

TABLE II. DATASET COMPILATION SETTINGS.

Compiler family	Version	Architecture x86	Architecture x86-64	Optimization Level Low	Optimization Level High
VC	2003	✓		-Od	-Ox
	2017	✓	✓	-Od	-Ox
GCC	6.3.0	✓	✓	-O0	-O3
Clang	5.0.2	✓	✓	-O0	-O3
ICC	19.0.0.117	✓	✓	-O0	-O3

TABLE III. THE KEYWORD LIST FOR EACH LABEL

		keyword list
Others	CFB	"test",".doc"
	OOXML	"test",".docx"
	PDF	"test",".pdf"

In addition, we should know function boundaries to divide the library code and the code generated from the well-known source code. All the machine code in the object file is the code derived from the source code. We can obtain ground truth from the object files without these difficulties.

Next, to make the "Binary" datasets in the "Others" category, we used a search engine to gather various open-source document files. TABLE III shows the keywords used for this search. The "Others" category contains "CFB,""OOXML," and "PDF" files. CFB stands for compound file binary [22], and it is used as a container like the FAT16 file system. CFB is used in files with the extensions ".doc," ".xls," ".ppt," and so on. OOXML stands for Office Open XML [16], which is a zip container in reality. OOXML is used in ".docx," ".xlsx," and ".pptx" files. PDF stands for portable document format [15], which has the extension ".pdf." We downloaded document files from the beginning of the list of search results. We then checked these downloaded files using VirusTotal[8], and we removed suspicious files that were detected as malware. We adjusted the number of the document files so that the number of "Code" (described later) does not differ greatly from them of other labels.

2) Code: The following procedure is used to make "Code" datasets. First, we treat the files of the "Binary" dataset as x86/x86-64 machine code files, whether they come from the "Program" category or the "Others" category. Second, we separate these files into "instructions" (*i.e.*, disassembled the real or pretended x86/x86-64 machine code). Third, we convert each instruction into a 128-bit fixed-length instruction by padding it with "0." Finally, packing randomly selected L fixed-length instructions into one set, we make a $(L \times 128)$-bit sequence.

The reason we padded instructions to 128 bits (16 bytes) is the following. Fifteen bytes is basically the maximum length of one instruction as we described in Sec. III-A. The average of the lengths of each "instruction" is 3.69 bytes for the" Program" category and 2.38 bytes for the" Others" category. We did not find any instruction longer than 15 bytes in our experiment. However, theoretically, the length of one instruction maybe 16 bytes or more. Therefore we set the size of fixed-length instructions to 16 bytes (one byte larger than the maximum instruction length).

[8]https://www.virustotal.com/

B. Recognition Performance

We confirmed classification accuracy of compilers by our proposed method using the dataset described in Sec. V-A.

Evaluation experiments were carried out using the dataset described in the previous section. No compiler has been able to compile all source code. As a result, our dataset has a poor balance between sample numbers of each label. Therefore, we set (S) as the upper limit sample number for each label used in the evaluation experiment. Accuracy was obtained by 4-fold cross validation, and here is our parameter configuration:

- input length$(L) = 16$
- learning rate $(\eta) = 0.01$
- mini-batch size $= 1000$
- epochs $= 50$

TABLE IV shows a comparison between the existing methods and our proposed method. In TABLE IV, these accuracy stated the following values. Rosenblum's accuracy is the average of the accuracy described in Table 2 of the paper [29]. `ORIGIN`'s SVM's accuracy is the value of "All component" in Table 2 of the paper [26]. `ORIGIN`'s CRF's accuracy is the value of "Joint" in Table 3 of the paper [26]. Accuracy of `BinComp` is the average of accuracy described in Table 6 of the paper [25].

The number of labels included in our dataset is 19, which is the largest number compared with existing researches. However, in the paper [26], Rosenbulm *et al.* are experimenting with datasets consisting of four versions of GCC and three versions of VC. Regarding the compiler version differences, our dataset includes two versions (VC2017/VC2003). On the version of the compilers, their experiments are more challenging than our experiments.

The increase of input instructions improves the prediction accuracy. The effect is massive impact on our proposal model than `o-glasses`. The existing method `o-glasses`' prediction error $(1 - \text{Accuracy})$ changes 0.0677 to 0.0579 when the number of input instructions increases from 16 to 64. The error has decreased by 15%. On the other hand, our model's prediction error changes 0.0445 to 0.0116 when the number of input instructions increases from 16 instructions to 64 instructions. The error is reduced by about 74%. One of the biggest reason for this difference between our model's error and the `o-glasses`' error is that the model contains PE or not. `o-glasses` only considered a combination of instructions. On the other hand, our model containing PE increased the amount of input information, such as instruction context and distance. As a result, the effect of the increasing input instructions led to the prediction accuracy increase.

C. Automatic compiler feature extraction

In this section, we list the typical instructions of each compiler using Algorithm 1 and consider the knowledge obtained from them. We selected `aes.c` collected from GitHub for creating InputFile because all compilers can compile this source code successfully with all optimization levels.

Fig. 5 shows a list of the top five typical instructions in each compiler automatically extracted by the procedure described

TABLE IV. COMPARISON OF PERFORMANCE OF RELATED WORK. THE NUMBERS IN PARENTHESES DENOTE THE NUMBER OF LABELS.

			o-glassesX		o-glasses[23]		Rosenblum's[29]	ORIGIN[26]		BinComp[25]
			(L=64)	(L=16)	(L=64)	(L=16)		(SVM)	(CRF)	
Accuracy		All components	.9884 (19)	.9555 (19)	.9421 (19)	.9323 (19)	.924 (3)	.604 (18)	.918 (18)	.801 (6)
		Compiler family	.9886 (4)	.9670 (4)	.9140 (4)	.9271 (4)	.924 (3)	.983 (3)	.999 (3)	–
		Optimization	.9989 (2)	.9943 (2)	.9830 (2)	.9864 (2)	–	.971 (2)	.999 (2)	.917 (2)
		Architecture	.9997 (2)	.9985 (2)	.9959 (2)	.9940 (2)	–	–	–	–
		Code/Non-code	.9999 (2)	.9995 (2)	.9991 (2)	.9987 (2)	–	–	–	–
ML model			Attention	Attention	CNN	CNN	CRF	SVM	CRF	(k-means)
Dataset		Features	64 Instructions	16 Instructions	64 Instructions	16 Instructions	Byte Seq.	1 Function	Function Seq.	1 File
		#Samples	1,793,478	1,900,000	471,124	1,886,521	81,886,169	955,000	955,000	1,177
		#Binaries	28,074	28,074	28,074	28,074	1,119	2,686	2,686	1,177
	#Variations	VC	6	6	6	6	1	6	6	2
		GCC	4	4	4	4	1	8	8	2
		Clang	4	4	4	4	-	-	-	-
		ICC	4	4	4	4	1	4	4	2
		Non-code	1	1	1	1	-	-	-	-
K-fold cross-validation			4	4	4	4	(Anomalous)	(Anomalous)	(Anomalous)	10

in Sec. IV-C. Considering to Fig. 5, it is possible to catch the characteristics of each class. Followings are examples of knowledge obtained in the case of `aes.c`.

In the case of low optimization level, many compilers use `IMUL`. On the other hand, in the case of high optimization level, rarely compilers use this instruction. `IMUL` is a multiplication instruction. This instruction takes more than ten times longer to execute than shift instructions and add instructions. In the case of a constant multiple such as [$\texttt{eax} \times 3$], a replaced code which is a combination of shift instruction and addition instruction such as [$(\texttt{eax} << 1) + \texttt{eax}$] will increase the processing speed several times. Therefore, in the case of maximum optimization, many compilers may not use much `IMUL`.

On the other hand, focusing on VC2003 and VC2017 without optimization, it can be seen that VC2017 prefers `IMUL`, and prioritizes the readability of machine language instructions for debugging. When VC2003 was released, the processing time of `IMUL` was too long to prefer the readability. When VC2017 was released, the operating frequency of the CPU was high enough to be able to prefer the readability. Thus, we consider that Microsoft preferred the readability than the running speed at the time debugging.

In the case of ICC, `o-glassesX` focused on SSE2 instructions; i.e., `PXOR`. The SSE2 was first available in Intel Pentium 4 (2001) and became popular within a few years. However, at that time, it had taken time for the code generated by the compiler to refer to the newly released instruction set because of prioritizing compatibility. At present, there is almost no problem even if we do not consider the processor that cannot use the SSE2 instruction. Intel, designed x86/x86-64 architectures, developed ICC in order to be more prevalent the latest instruction set. Even if we are not familiar with this background, the proposed method extracts SSE2 instructions as a feature that distinguishes compilers.

D. Case Study 1: Various optimization levels in a `.text` *segment*

Fig. 6 shows a result obtained by sliding the position of a code fragment to be extracted from a certain executable file one byte from the beginning and visualizing the result of classification by the learner used in the experiment.

The leftmost `Bit-Image` in the figure is a visualization of a executable file in the same way as the binary editor `Stirling` [10]. One pixel corresponds to one byte, `Bit-Image` corresponds to one byte, NULL (0x00) corresponds to white, control character string (0x01 to 0x1F) to light blue, readable character string (0x20 to 0x7F) to red, and others to black. The remaining three parts of the figure are colored results of 19 classes classified by our method. On the left is coloring focusing on the type of compiler, coloring focusing on Visual C++ in the middle, coloring not focusing on version differences focusing on Visual C++ on the right.

As shown in the figure, the trend of the estimation result of the optimization option greatly differs in the first half and the second half of the execution code part. The first half is the execution code generated without optimization and the second half is the execution code generated with the maximum optimization. As a result of static analysis, the first half was the executable code which the author of the executable file thought to have hand-crafted, the second half was the static link library. It was confirmed that the execution code compiled with multiple optimization options was mixed in the same section due to the influence of the linked static link library. In addition, our method was able to visualize the situation well.

Therefore, when estimating the compiler and optimization option, it can be inferred that there is a limit to the accuracy of the method which does not assume that a plurality of compilers and optimization options are mixed in the same section. In the proposed method, input data required for estimation is as small as 16 instructions, so there is a high possibility that the input data contains only the execution code generated with a single compiler and optimization option.

E. Case Study 2: Tracking the change of Emdivi RATs in development environment

In this section, we describe the change of Emdivi creation environment by arranging the analysis results of these malware in order of compilation time. In order to accurately infer the creation environment of malware by our method, instructions created by the same compiler used to create the malware needs to be included in the learning dataset. It is difficult to exactly identify, because the compiler types and options are diverse. However, by analyzing malware in the same family and arranging the analysis results in chronological order, we can find changes in the environment used for malware creation.

	VC03,32,none(Od)	VC17,32,none(Od)	GCC,32,none(O0)	Clang,32,none(O0)	ICC,32,none(Od)
1	`XOR ECX, [EAX*4+0x0]`	`IMUL ECX, EAX, 0x0`	`AND EAX, 0xff000000`	`AND EAX, 0xff000000`	`IMUL EAX, EAX, 0x4`
2	`AND EAX, 0xff000000`	`IMUL ECX, EAX, 0x9`	`MOVZX EAX, AL`	`NOP WORD [EAX+EAX+0x0]`	`AND EAX, 0xff000000`
3	`XOR EDX, [EAX*4+0x0]`	`IMUL EDX, ECX, 0xc`	`LEA EDX, [EAX*4+0x0]`	`AND ECX, 0xff0000`	`IMUL EDX, EDX, 0x4`
4	`AND ECX, 0xff0000`	`XOR ECX, EBP`	`ADD DWORD [EBP-0x8], 0x1`	`XOR EAX, [ECX*4+0x1028]`	`CALL 0x4003c76`
5	`XOR EAX, [EDX*4+0x0]`	`IMUL ECX, EAX, 0x3`	`AND EAX, 0xff0000`	`NOP DWORD [EAX+0x0]`	`MOV EAX, 0xffffffff`

	VC03,32,max(Ox)	VC17,32,max(Ox)	GCC,32,max(O3)	Clang,32,max(O3)	ICC,32,max(Ox)
1	`AND ESI, 0xff0000`	`NOP`	`LEA ESI, [ESI+0x0]`	`NOP WORD [CS:EAX+EAX+0x0]`	`NOP DWORD [EAX+0x0]`
2	`AND ECX, 0xff0000`	`MOVZX ECX, BYTE [EAX+EBP]`	`LEA EDI, [EDI+0x0]`	`MOV EBP, 0xff000000`	`LEA EBX, [EDX+EDX]`
3	`AND EDI, 0xff`	`PUSH DWORD [ESP+0x24]`	`AND EAX, 0xff000000`	`MOV EDX, 0xff00`	`LEA ECX, [ESI+ESI]`
4	`XOR ECX, EBP`	`SUB DWORD [ESP+0x2c], 0x1`	`AND EAX, 0xff0000`	`MOV DL, [ESP+0x15]`	`LEA EDX, [ESI+ESI]`
5	`AND ECX, 0xff`	`XOR EAX, ESP`	`AND EBX, 0xff0000`	`MOV ESI, 0xff00`	`PXOR XMM0, [ESP+0x20]`

Fig. 5. Typical instructions for each compiler against `aes.c`.

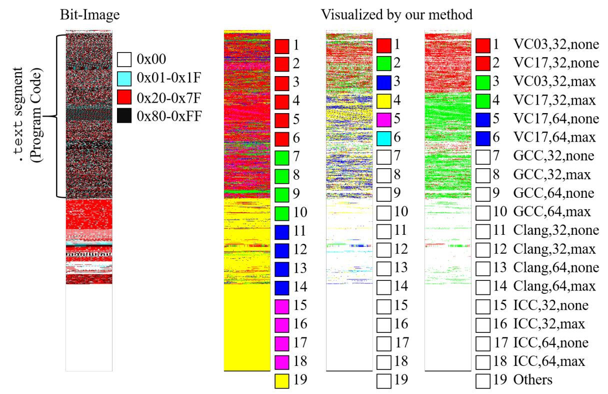


Fig. 6. An executable file visualized by our method

Emdivi is a malware family of RAT (Remote Access Trojan), which were observed many times in Japan from 2014 to 2015. We obtained 162 hashes of Emdivi RATs from the report[9] created by MACNICA NETWORKS. We got 148 Emdivi RATs from Virus Total based on this hash list. We visualized these files in the same way as in Sec. V-D. As a result, they are all executable files of the 32 Bit architecture created in Visual C++, and the optimization option was found to be the maximum optimization. Analysing of compiler versions, malware development environments were largely classified into three (Type A, B and C.)

Figure 7 shows the result of visualizing three representative files by our method. Type A is created by a relatively new compiler, and linked libraries are also created by relatively new compilers. Type B is created by a relatively old compiler, but the linked libraries are created by a relatively new compiler. Type C is created by a relatively new compiler, but some of the linked libraries are created by a relatively old compiler.

Table V shows the results of classifying each file into the three types mentioned above. "No." is a number assigned sequentially from the top of the already-mentioned hash list. "Version" is a version of Emdivi, inferred from communication contents. "Compile Times" is a compile date and time, which is based on header information. For types whose "No." are

[9]https://www.macnica.net/security/report_01.html/

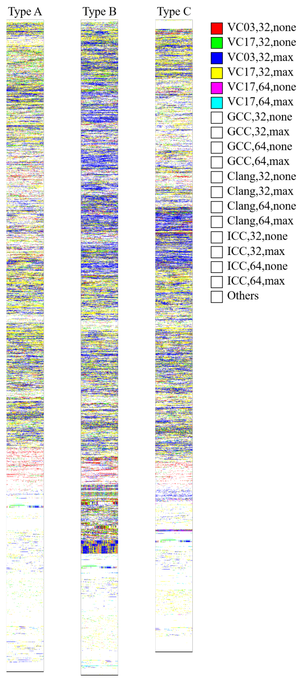


Fig. 7. The result of visualizing Emdivi RATs. Type A is created by a relatively new compiler, and linked libraries are also created by relatively new compilers. Type B is created by a relatively old compiler, but the linked libraries are created by a relatively new compiler. Type C is created by a relatively new compiler, but some of the linked libraries are created by a relatively old compiler.

smaller than 109, they are not described in the table because Type was all A.

Based on the above results, the change of the structure of the attack group was inferred by the following procedure.

1) Premise: Characteristics of attack method of attack group using Emdivi include the following.

- Attach to the e-mail just before sending the targeted mail t17 series malware is compiled.
- After successful intrusion, another malware (t20 series) is sent to another terminal in the attack target system network in order to expand the intrusion.

TABLE V. TRANSITION OF DEVELOPMENT ENVIRONMENT OF EMDIVI

No.	Version	Compile Times	Type
109	t17.08.27	2015/03/19 15:03:19	A
110	t17.08.27	2015/03/20 12:04:19	A
111	t17.08.27	2015/03/20 10:44:49	A
113	t17.08.27	2015/03/24 12:07:23	A
115	t17.08.29	2015/04/22 11:29:48	B
117	t17.08.29	2015/05/08 10:20:53	A
118	t20.19	2015/05/20 17:25:17	A
119	t20.19	2015/05/20 17:38:52	A
120	t17.08.30	2015/05/20 15:42:48	A
121	t20.19	2015/05/20 17:00:39	A
122	t17.08.30	2015/05/20 11:52:28	A
123	t20.19	2015/05/21 15:10:03	A
124	t20.19	2015/05/21 14:08:10	A
125	t17.08.30	2015/05/21 15:38:39	A
126	t17.08.30	2015/05/21 15:38:39	A
127	t17.08.30	2015/05/22 11:51:18	A
128	t17.08.30	2015/05/22 11:51:18	A
129	t17.08.30	2015/05/22 11:51:18	A
130	t20.20	2015/05/27 11:07:55	A
131	t17.08.30	2015/05/28 12:48:14	A
132	t20.20	2015/05/29 11:19:00	A
133	t17.08.30	2015/06/02 11:15:26	A
134	t20.20	2015/06/03 11:50:00	A
135	t20.20	2015/06/04 15:12:36	A
136	t17.08.31	2015/06/18 10:15:02	B
136	t17.08.31	2015/06/18 10:15:02	B
137	t20.22	2015/07/06 15:32:47	B
138	t17.08.31	2015/07/06 10:34:56	B
139	t17.08.31	2015/07/10 09:58:16	B
140	t20.22.1	2015/07/10 08:49:45	A
141	t17.08.31	2015/07/10 08:40:15	B
142	t20.22.1	2015/07/10 09:10:41	A
143	t17.08.31	2015/07/13 00:23:13	B
144	t17.08.31	2015/07/13 10:46:27	B
145	t17.08.31	2015/07/14 09:57:44	B
146	t17.08.31	2015/07/14 10:16:54	B
147	t17.08.31	2015/07/14 17:44:14	B
148	t17.08.31	2015/07/16 09:10:07	B
149	t17.08.31	2015/07/28 12:56:35	B
150	t20.23.1	2015/07/31 17:35:52	A
151	t20.23.1	2015/07/31 17:03:49	A
152	t17.08.31	2015/08/05 08:51:31	B
153	t17.08.34	2015/08/07 09:23:11	C
154	t20.25.1	2015/08/07 13:11:08	A
155	t17.08.34	2015/08/10 14:47:52	C
156	t17.08.34	2015/08/13 08:48:01	C
157	t17.08.34	2015/08/13 09:35:15	C
158	t20.26	2015/08/13 13:21:57	A
159	t20.26	2015/08/13 13:29:34	A
160	t17.08.34	2015/08/19 09:16:01	C
161	t17.08.34	2015/08/19 09:16:01	C
162	t17.08.34	2015/10/13 09:52:52	C

- t20 series of malware is compiled just before sending.

2) Assumption: The compilation time and the time to transmit malware are very close, so we made the following assumptions. The person who compiles, the person who sends malware, and the person who uses malware are highly likely to be the same person.

3) Activities of A, B and C:

A	Both initial compromise and compromise expansion staff. After June 2015, mainly compromise expansion staff.
B	Initial compromise staff (activity period: mid-June 2015 to early August)
C	Initial compromise staff (activity period: after early August 2015)

4) Guess: Although the organization before June 2015 is unclear, there is a possibility that from 2016 June onwards, responsible for intrusion (targeted mail sending) and explicit role sharing of intrusion expansion after successful entry is there.

VI. DISCUSSION

A. Benefits of our method

1) High Recognition Rate for Stripped Machine Code: It is applicable even when symbol information has been stripped or is otherwise unavailable because our approach relies only on characteristics of the binary code and not on meta-data or other details of program headers. Furthermore, it can be used even when codes produced by multiple compilers coexist within a program binary, such as statically linked library code because our method classifies sequences of code instead of whole binaries. Our approach extracts compiler provenance with high accuracy even in such complex programs.

2) Proposal of a model that can calculate how much input data contributes to output in units of instructions: The attention mechanism is rapidly spreading in natural language processing in recent years. In the field of machine translation, the attention model without RNN has obtained a translation score that exceeds the existing research using RNN [34]. Additionally, the model with the attention mechanism can calculate the degree of contribution to the decision.

In this paper, we applied the attention mechanism to code fragment recognition. At the best of our knowledge, `o-glassesX` is the first example of applying the attention mechanism to machine code classification. For calculating the contribution to the identification result one instruction by one instruction, we designed our network carefully. Our model can calculate the degree of the contribution to the identification result on each instruction.

3) Knowledge discovery through model's decision: Our model can identify the most important instruction of the input. Therefore, by counting the appearance frequency of instructions that contributed to the compiler identification results, we can extract the typical instructions in each class automatically.

We listed the compiler-typical instructions generated when compiling `aes.c` with various compilers. As a result, we caught some of the features of each compiler. Our method helps us to understand the characteristics of each class.

B. The limitation of our method

`o-glassesX` still suffers from some limitations. For instance, like most existing methods, `o-glassesX` works under the assumption that the machine code is already de-obfuscated. In practice, de-obfuscation of malware are very demanded. How to extract compiler provenance directly over obfuscated code is an important issue but very challenging future avenue of research. On the other hand, once we success de-obfuscate code snippet, we can identify source compiler since `o-glassesX` does not require meta-data.

Additionally, our method cannot support two or more CPU architectures at the same time. When dealing with two or more CPU architecture inputs at the same time, first we need identifying the CPU architecture from the input binary, and then it is necessary to input the binary to `o-glassesX` specialized for each architecture.

VII. CONCLUSION

In this paper, we proposed a novel method for binary analysis. To the best of our knowledge, this is the first application of the attention mechanism to machine code recognition. Our method can identify the source compiler family and optimization levels of machine code, an important element of program provenance. Additionally, our method can demonstrate how much one instruction by one in 16 or 64 instruction windows contributed to the decision of the compiler family and the optimization level.

We evaluated a large set of test binaries. Our result showed that our method could identify the source compiler family and optimization levels of machine code with about 0.99 accuracy rate almost perfect, even though the size of the input binary fragment is only 64 instructions. Additionally, our case study showed that our method can provide new knowledge; *e.g.*, the characteristic of instruction choices by each compiler.

It is left as an open problem to find a solution to more general program provenance and author identification from binary codes (malware.)

References

[1] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *ICLR2015*, 2015.

[2] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 845–860.

[3] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of n Proceedings of the 17th Network and Distributed System Security Symposium*. The Internet Society, 2010.

[4] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, "When coding style survives compilation: De-anonymizing programmers from executable binaries," in *the 2018 Network and Distributed System Security Symposium (NDSS)*, 02 2018.

[5] L. Cheng, Y. Zhang, Y. Zhang, C. Wu, Z. Li, Y. Fu, and H. Li, "Optimizing seed inputs in fuzzing with machine learning," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 2019, pp. 244–245.

[6] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

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

[8] Q. Feng, R. Zhou, Y. Zhao, J. Ma, Y. Wang, N. Yu, X. Jin, J. Wang, A. Azab, and P. Ning, "Learning binary representation for automatic patch detection," in *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2019, pp. 1–6.

[9] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323.

[10] K. Goto, "Stirling," https://www.vector.co.jp/soft/win95/util/se079072.html, 1998.

[11] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 85–96.

[12] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[13] Intel, "Intel 64 and ia-32 architectures software developer manuals," 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[14] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[15] ISO, "ISO32000–1:2008 Document management – Portable document format – Part 1 : PDF1.7." [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502

[16] ——, "ISO/IEC 29500:2012:Information technology – Document description and processing languages – Office Open XML File Formats," 2012.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[18] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[19] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[20] A. McCallum, "Efficiently inducing features of conditional random fields," in *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 2002, pp. 403–410.

[21] X. Meng, B. P. Miller, and K.-S. Jun, "Identifying multiple authors in a binary program," in *ESORICS*, 2017.

[22] Microsoft, "[ms-cfb]: Compound file binary file format." [Online]. Available: https://msdn.microsoft.com/ja-jp/library/dd942138.aspx

[23] Y. Otsubo, A. Otsuka, M. Mimura, and T. Sakaki, "o-glasses: Visualizing x86 code from binary using a 1d-cnn," *IEEE Access*, vol. 8, pp. 31 753–31 763, 2020.

[24] B. M. Padmanabhuni and H. B. K. Tan, "Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2015, pp. 450–459.

[25] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, "Bincomp: A stratified approach to compiler provenance attribution," *Digital Investigation*, vol. 14, pp. S146–S155, 2015.

[26] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 100–110.

[27] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Machine learning-assisted binary code analysis," in *NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security, Whistler, British Columbia, Canada, December*, 2007.

[28] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? identifying the authors of program binaries," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 172–189.

[29] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2010, pp. 21–28.

[30] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code." in *AAAI*, 2008, pp. 798–804.

[31] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. ACM, 2009, pp. 117–128.

[32] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 611–626.

[33] G. J. Tesauro, J. O. Kephart, and G. B. Sorkin, "Neural networks for computer virus recognition," *IEEE expert*, vol. 11, no. 4, pp. 5–6, 1996.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[35] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 388–398.

[36] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "Neufuzz: Efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36 340–36 352, 2019.

[37] H. Yakura, S. Shinozaki, R. Nishimura, Y. Oyama, and J. Sakuma, "Malware analysis of imaged binary samples by convolutional neural network with attention mechanism," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. ACM, 2018, pp. 127–134.

[38] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 797–812.

APPENDIX

A. *Details of Recognition Performance*

In this appendix, we provide the details of the recognition performance (Sec. V-B.)

1) The effect of the size of the dataset: Evaluation experiments were carried out using the dataset described in Sec. V-A. No compiler has been able to compile all source code. As a result, our dataset has a poor balance between sample numbers of each label. Therefore, we set (S) as the upper limit sample number for each label used in the evaluation experiment. The relationship between S and accuracy was obtained (Fig. 8). Accuracy was obtained by 4-fold cross validation, and here is our parameter configuration:

- input length(L) $= 16$
- learning rate (η) $= 0.01$
- mini-batch size $= 1000$
- epochs $= 50$

2) Confusion matrix: TABLE VI shows an overview of the results (L=64, S=100,000.)

In the comparison of the different algorithms, we use the F-measure defined by

$$F_{measure} = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}. \tag{8}$$

In this calculation, precision is given by

$$Precision = \frac{TP}{TP + FP}, \tag{9}$$

and recall is given by

$$Recall = \frac{TP}{TP + FN}, \tag{10}$$

where TP is the true positive rate, FP is the false positive rate, FN is the false negative rate, and TN is the true negative rate.

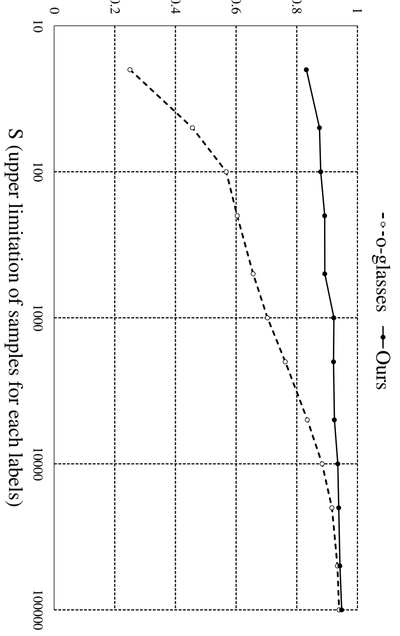


Fig. 8. Accuracy of 19 classification against S. L=16, S=20, 50, 100, 200, 500, 1,000, 2,000, 5,000, 10,000, 20,000, 50,000, and 100,000

TABLE VI. PERFORMANCE OF OUR METHOD TO RECOGNIZE X86/X86-64 CODE. L=64, S=100,000. (R): RECALL, (P): PRECISION, (F1): F-MEASURE.

Train \ Predict		VC						GCC				Clang				ICC				Others			
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	R	P	F1
VC03,32,none(Od)	1	.9604	.0355	.0026	.0002	.0002	.0002	.0001	.0000	.0000	.0000	.0001	.0000	.0000	.0000	.0001	.0002	.0000	.0000	.0004	.9500	.9604	.9552
VC17,32,none(Od)	2	.0463	.9517	.0002	.0007	.0001	.0001	.0000	.0000	.0000	.0000	.0001	.0000	.0000	.0000	.0000	.0002	.0000	.0000	.0006	.9625	.9517	.9570
VC03,32,max(Ox)	3	.0026	.0001	.9875	.0061	.0001	.0003	.0000	.0002	.0000	.0000	.0000	.0002	.0000	.0000	.0000	.0022	.0000	.0000	.0006	.9786	.9875	.9830
VC17,32,max(Ox)	4	.0004	.0010	.0144	.9774	.0001	.0005	.0000	.0001	.0000	.0001	.0001	.0008	.0000	.0001	.0000	.0044	.0000	.0000	.0007	.9887	.9774	.9830
VC17,64,none(Od)	5	.0002	.0001	.0002	.0001	.9978	.0008	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0001	.0000	.0001	.0001	.0001	.0004	.9977	.9978	.9977
VC17,64,max(Ox)	6	.0002	.0001	.0004	.0004	.0013	.9931	.0000	.0000	.0000	.0005	.0000	.0000	.0001	.0006	.0000	.0001	.0000	.0023	.0008	.9955	.9931	.9943
GCC,32,none(O0)	7	.0002	.0000	.0001	.0000	.0000	.0000	.9973	.0009	.0004	.0001	.0003	.0004	.0000	.0000	.0002	.0000	.0000	.0000	.0000	.9972	.9973	.9973
GCC,32,max(O3)	8	.0001	.0000	.0005	.0002	.0000	.0000	.0011	.9921	.0000	.0003	.0002	.0051	.0000	.0000	.0000	.0004	.0000	.0000	.0000	.9946	.9921	.9933
GCC,64,none(O0)	9	.0001	.0001	.0000	.0000	.0000	.0000	.0008	.0000	.9970	.0006	.0000	.0000	.0010	.0002	.0000	.0000	.0001	.0000	.0000	.9979	.9970	.9975
GCC,64,max(O3)	10	.0000	.0000	.0000	.0000	.0001	.0002	.0001	.0006	.0003	.9800	.0000	.0003	.0003	.0168	.0000	.0001	.0001	.0012	.0000	.9819	.9800	.9809
Clang,32,none(O0)	11	.0003	.0002	.0000	.0000	.0000	.0000	.0004	.0004	.0000	.0001	.9972	.0006	.0007	.0000	.0001	.0000	.0000	.0000	.0000	.9959	.9972	.9965
Clang,32,max(O3)	12	.0001	.0000	.0006	.0011	.0000	.0001	.0004	.0068	.0000	.0003	.0006	.9869	.0000	.0019	.0000	.0012	.0000	.0000	.0000	.9790	.9869	.9829
Clang,64,none(O0)	13	.0000	.0000	.0000	.0000	.0000	.0001	.0000	.0000	.0009	.0003	.0010	.0000	.9973	.0003	.0000	.0000	.0000	.0001	.0000	.9979	.9973	.9976
Clang,64,max(O3)	14	.0000	.0000	.0001	.0001	.0001	.0006	.0000	.0001	.0002	.0147	.0000	.0015	.0002	.9810	.0000	.0001	.0001	.0013	.0001	.9796	.9810	.9803
ICC,32,none(Od)	15	.0001	.0000	.0001	.0000	.0000	.0000	.0002	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.9994	.0002	.0000	.0000	.0000	.9995	.9994	.9994
ICC,32,max(Ox)	16	.0002	.0001	.0029	.0028	.0000	.0001	.0000	.0003	.0000	.0001	.0000	.0005	.0000	.0000	.0001	.9930	.0000	.0001	.0000	.9916	.9930	.9923
ICC,64,none(Od)	17	.0000	.0000	.0000	.0000	.0001	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.9996	.0001	.0000	.9995	.9996	.9995
ICC,64,max(Ox)	18	.0000	.0000	.0001	.0000	.0001	.0016	.0000	.0001	.0000	.0012	.0000	.0000	.0000	.0013	.0000	.0001	.0002	.9952	.0000	.9948	.9952	.9950
Others	19	.0000	.0000	.0000	.0001	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.0000	.9997	.9964	.9997	.9980