IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received January 21, 2020, accepted February 2, 2020, date of publication February 7, 2020, date of current version February 20, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2972358

o-glasses: Visualizing X86 Code From Binary Using a 1D-CNN

YUHEI OTSUBO[1,2], AKIRA OTSUKA[2], (Member, IEEE), MAMORU MIMURA[3,2], AND TAKESHI SAKAKI[4]

[1]National Police Agency, Tokyo 100-8974, Japan
[2]Institute of information Security, Kanagawa 221-0835, Japan
[3]National Defense Academy, Kanagawa 239-8686, Japan
[4]Institute for Future Initiatives (IFI), The University of Tokyo, Tokyo 113-0033, Japan

Corresponding author: Yuhei Otsubo (dgs157101@iisec.ac.jp)

ABSTRACT Malicious document files used in targeted attacks often contain a small program called shellcode. It is often hard to prepare a runnable environment for dynamic analysis of these document files because they exploit specific vulnerabilities. In these cases, it is necessary to identify the position of the shellcode in each document file to analyze it. If the exploit code uses executable scripts such as JavaScript and Flash, it is not so hard to locate the shellcode. On the other hand, it is sometimes almost impossible to locate the shellcode when it does not contain any JavaScript or Flash but consists of native x86 code only. Binary fragment classification is often applied to visualize the location of regions of interest, and shellcode must contain at least a small fragment of x86 native code even if most of it is obfuscated, such as a decoder for the obfuscated body of the shellcode. In this paper, we propose a novel method, o-glasses, to visualize the shellcode by recognizing the x86 native code using a specially designed one-dimensional convolutional neural network (1d-CNN). The fragment size needs to be as small as the minimum size of the x86 native code in the whole shellcode. Our results show that a 16-instruction-sequence (approximately 48 bytes on average) is sufficient for the code fragment visualization. Our method, o-glasses (1d-CNN), outperforms other methods in that it recognizes x86 native code with a surprisingly high F-measure rate (about 99.95%).

INDEX TERMS Binary analysis, CNN, machine learning, MLP, shellcode, visualization.

I. INTRODUCTION

In recent years, targeted attacks have become a major threat. In a targeted email attack, an email contains a request to open an attached file or click on a hyperlink in the email body. If the recipient does so, then some malware is launched. Most such malware is newly crafted, unknown malware, and is thus often hard for antivirus scanners to detect. In particular, malicious document files used in targeted email attacks often contain an executable file embedded within a decoy document file: over 60% of the attached files in targeted email attacks occurring in 2014 were reported to be document files [27].

The left-hand side of Figure 1 shows a typical structure for a malicious document file. The malicious document file consists of four parts: exploit code, shellcode, an executable file, and a decoy document file. Exploit code is a piece of software, a chunk of data, or a sequence of commands that takes advantage of a bug or vulnerability of computer software, hardware, or something electronic to cause unintended or unanticipated behavior. Exploit code in the malicious document file is a program designed to exploit a word processing software vulnerability. Exploit code is executed when the malicious document file is opened, and typically leading to

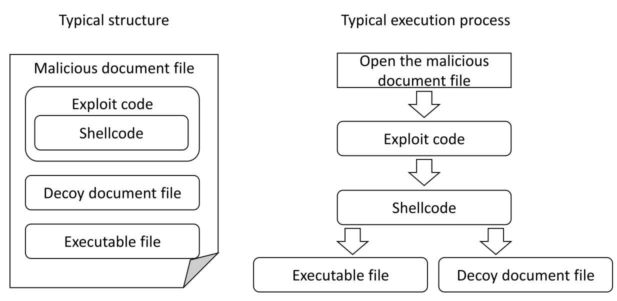

FIGURE 1. Typical structure and execution process of a malicious document.

The associate editor coordinating the review of this manuscript and approving it for publication was Donato Impedovo.

the execution of the shellcode. A shellcode is a piece of code that is executed after exploiting a software vulnerability. Modern operating systems implement ASLR (Address Space Layout Randomization[1]) and DEP (Data Execution Prevention[2]) to protect from execution of such malicious codes. However, there are already several known exploit codes (ex. CVE-2014-6362, CVE 2017-11882 and others) that can avoid such mitigations. The typical shellcode in the malicious document file is designed to create an executable file and a decoy document from the remainder of the file and to launch the executable file. Then, the victim that opened the malicious document file becomes controllable by attackers. The right-hand side of Figure 1 shows a typical execution process of a malicious document file.

To reach attackers' information, we should not only detect the malware but also figure out the features of the malware in detail. Here, we face several problems. First, we should prepare a runnable condition for the malware in order to conduct dynamic analysis. When the target file is a malicious document exploiting specific vulnerabilities, it is often difficult to prepare the activatable environment (OS versions, browsing software, language, patches, and so on) because the conditions are complicatedly intertwined. Therefore, we are often forced to conduct static analysis. When the target file is an executable file, it is easy to find the entry point for analysis. However, when the target file is a document file, it is not so easy to find the entry point. In this case, we focus on the shellcode executed after exploit code. When the malware uses JavaScript or Flash, we can figure out the location of the shellcode quickly. However, exploit code uses not only JavaScript and Flash but also font and image files, for example, a TIFF image (CVE-2017-5133 [19], [25], CVE-2004-1308 [16]), a jpeg2000 image (CVE-2016-8332 [18], [24]), and a TrueType font (CVE-2011-3402 [17]). When searching for shellcode, it is important to consider various types of exploit code. Thus, our target is a class of malicious document files that contain x86 native code hidden somewhere in them.

Although attackers tend to use obfuscation to protect their code, shellcode must contain at least a small fragment of x86 native code, such as a decoder. Figure 2 shows an example of a small decoder containing 17 opcodes in only a 29-byte sequence. This code was obtained from a malicious document file with a size of more than 100kB used in a real attack.

Our challenge, therefore, is finding a small amount of code like that shown in Figure 2 in often large document files. To do this, we introduce a novel method, called o-glasses, to visualize the shellcode by recognizing the x86 native code using a specially designed one-dimensional convolutional neural network (1d-CNN).

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

```
0000:60             PUSHA
0001:B967010000     MOV ECX,0x167
0006:EB0F           JMP loc_17
0008:90             NOP
0009:5E             POP ESI
000A:90             NOP
000B:56             PUSH ESI
000C:46             INC ESI
000D:8BFE           MOV EDI,ESI
000F:AC             LODSB
0010:34CC           XOR AL,0xCC
0012:AA             STOSB
0013:49             DEC ECX
0014:75F9           JNZ loc_F
0016:C3             RET
0017:E8ECFFFFFF     CALL loc_8
001C:61             POPA
```

FIGURE 2. An example of a small decoder with a 29-byte sequence. It contains only 17 opcodes, and it decodes the body of the shellcode with 1-byte-key xor encoding. ("0xCC" in this example).

A. EASILY COLLECTIBLE TRAINING DATASETS

One of the most significant problems in using machine learning is how to prepare the training dataset. Even an excellent model cannot demonstrate its performance without large samples. However, studies of malware using machine learning sometimes struggle to collect samples because they need examples of malware for training. In contrast, our approach does not need malware for the training dataset. Thus, samples for learning are easily available to anyone.

B. HIGH RECOGNITION RATE FOR X86 CODE

Conventional signature-based malware detectors do not work when an unknown code is embedded. On the other hand, program code is not supposed to be in document files. So, extracting shellcode from malicious document files becomes a reality if we can separate program code precisely from normal byte sequences in document files. The solution provided in this paper is based on the assumption that shellcode and general program code have similar distributions of code.

C. VISUAL ANALYSIS FOR SUPPORTING ANALYSTS

Visualizing a binary as an image helps to get an overview of the file quickly. While some experienced analysts can deduce the location of the embedded program code from a grayscale image converted from the binary file, even inexperienced analysts can achieve similar results using our proposed methods.

II. PRELIMINARIES

The proposed solution lies in the static analysis of files. We do not take into account the file structure. The only thing

[1] ASLR is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

[2] DEP is a system-level memory protection feature that is built into the operating system starting with Windows XP and Windows Server 2003 (cited from https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention).

of importance for us is whether a file fragment is a piece of x86 code.

A. X86 ARCHITECTURE

The x86 and x86-64 architectures are probably the most widely used CISC (Complex Instruction-Set Computing) architectures [8]. Their instruction sets are rich and complex, and most importantly, they support instructions of varying length. Instruction lengths range from just one byte (i.e., instructions comprising just a one-byte opcode) to 15 bytes.

B. ASSUMPTIONS

We made the following assumptions.

Assumption 1: The distribution of the byte sequence from x86 code is dissimilar to that from document files.

Assumption 2: The distribution of shellcode is the same as that of common x86 code.

In other words, we expect to find shellcode by detecting any x86 code.

We next describe Shannon entropy, conventional visualization methods, and the deep learning models (multi-layer perceptron (MLP) and CNN) used in the study.

C. SHANNON ENTROPY

We calculate the information entropy of each file fragment using the Shannon entropy rate given by

$$H(X) = -\frac{1}{8}\sum_{i=0}^{255} P(X = i)\log_2 P(X = i), \quad (1)$$

where X is a random variable over [0, 255]. The entropy rates are real numbers between zero and one, where one means the file fragment is uniformly random.

D. CONVENTIONAL VISUALIZATION METHODS

Visualizing a binary as an image is very helpful for getting a quick overview of the file. In this section, we describe the three conventional visualization methods.

1) GRAYSCALE

A technique for representing different files with grayscale images was introduced by Conti *et al.* [2] and was applied to automatic malware classification by Nataraj *et al.* [20].

2) BIT-IMAGE REPRESENTATION OF A BINARY FILE

Goto [5] implemented the visualization of a binary file as a ''bit-image'' in a hex editor named ''Stirling'' in 1998. In Stirling, a given binary is read as a vector of 8-bit unsigned integers and then organized into a two-dimensional array. This can be visualized as a bit-image in four colors: 0x00 (null) in white, 0x01-0x1F (control characters) in light blue, 0x20-0x7F (ASCII) in red, and 0x80-0xFF in black.

3) STRUCTURAL ENTROPY

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. When a document file may be considered as a system of such elements, then we can use the term structural entropy for its characterization. Sorokin [23] built entropy diagrams by using the sliding window method. He selected 256 bytes for the window (block) size and 128 bytes for the window (block) shift. In our experiment, we used the same block size but changed the block shift to 1 byte to provide more detail. We calculate the entropy level at each offset and visualize the structural entropy as a grayscale image.

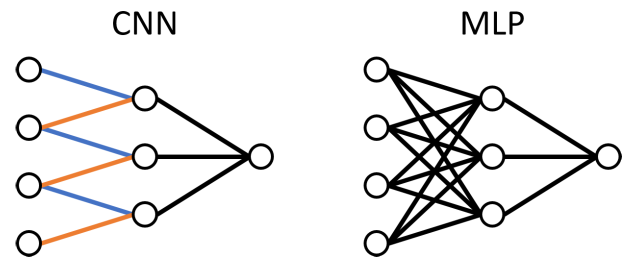

FIGURE 3. Schematic diagrams of a CNN and an MLP.

E. MLP

A standard MLP neural network has a three-tier structure: the input layer, the hidden layers, and the output layer. Every layer in an MLP consists of nodes fully connected with the nodes in the adjacent layer.

F. CNN

In our method for recognizing x86 native code, we use a 1d–CNN [14]. In contrast to an MLP, a CNN has limited connections between each layer (see Figure 3) and nodes in an intermediate layer receive only input from a localized part of the previous layer, which is called the receptive field.

Tools based on CNN have now led to great results in a wide range of vision tasks [13]. Generally, image data are continuous data. So, when image data are input to a CNN, high object recognition performance can be obtained by adjusting CNN's local receptive field. On the other hand, program code is classified as discrete data when viewed one byte at a time. Therefore, when binary data directly converted into an image are input to a CNN, there is the possibility that the benefit of the local receptive field cannot be obtained. On the other hand, program code is a sequence of instructions, which may reduce the variation, so the possibility of receiving the benefit of CNN's local receptive field is not entirely ruled out.

Weight sharing is a mechanism in which all links to nodes of a local receptive field have the same weight. In the case of Figure 3, the three blue links have the same weight. Similarly, the three red links have the same weight. By using the local receptive field in this way, the result of some input data is the same as a result of shifted input data. This allows us to reflect all the data in intermediate layers despite the limited connectivity to the input.

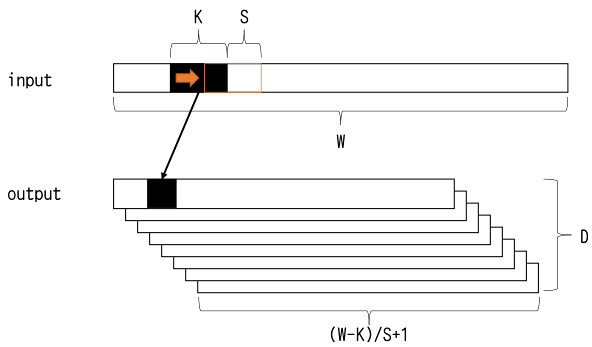

FIGURE 4. Illustration of the one-dimensional convolutional architecture.

Several hyperparameters control the size of the output volume of the convolutional layer (Figure 4): the kernel field size, depth, stride, and zero-padding. We will ignore zero-padding because we do not use it. The depth (D) of the output volume controls the number of neurons in a layer that connects 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 field size of the convolutional layer neurons K, and the stride with which they are applied S. The formula for calculating how many neurons "fit" in a given volume is given by $(W - K)/S + 1$.

III. RELATED WORK

Methods of analyzing malware can be divided into two types: static analysis and dynamic analysis. We focus on static analysis, as explained previously.

OfficeMalScanner [1] (OMS) is an analysis tool for document files. OMS scans entire files for generic shellcode patterns, an embedded signature of document files, or an embedded executable file. Although this method incorporates a fuzzy search, it is easy to avoid detection because the number of search patterns is small.

MDScan [28] is a standalone malicious document scanner. The tool analyzes PDF document files individually and detects malicious code. The tool combines static analysis of the document format representation and dynamic analysis of the embedded script code. The method focuses only on JavaScript in PDF files. Hence, the method does not work well when the exploit code is not written in JavaScript.

There are several approaches to malware detection that use binary or grayscale images (binary texture analysis [21], malware images [20], support vector machines [12] and visualization of binary files [7]). These approaches are aimed toward the detection and the classification of malicious software based on image processing techniques. Hence, they do not focus on finding a small amounts x86 code, such as shellcode, as we are doing here.

Binary fragment classification can visualize the location of regions of interest. The fragment size needs as small as the size of shellcode to find it. Xu *et al.* [30] treated a 1024-byte file fragment as a grayscale image and used an image classification method to classify file fragments. They focused on file type classification for digital forensics. It is difficult to make the fragment smaller because the texture of its grayscale image becomes harder to analyze. Hence, it is difficult to find shellcode using this method.

TABLE 1. Number of elements in each of our datasets.

Category & Source		File Num.	File Size	Block Num.	Code Num.
Program	GitHub	1,147	15,323 KB	49,577	258,793
Program	Ubuntu	295	19,083 KB	72,103	404,427
Program	Total	1,442	34,406 KB	121,680	663,220
Others	CFB	27	8,196 KB	31,779	223,904
Others	PDF	18	9,097 KB	33,495	220,669
Others	OOXML	27	8,638 KB	35,357	232,573
Others	Total	72	25,932 KB	100,631	677,146
Total		1,514	60,338 KB	222,311	1,340,366

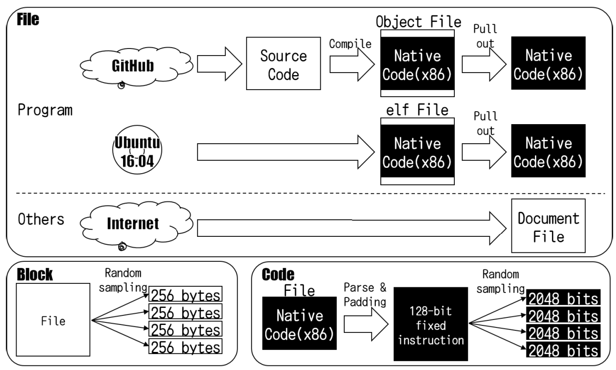

FIGURE 5. Method for reproducing our datasets.

IV. TRAINING DATA

We prepared two categories of a dataset for training, both of which can be gathered easily. One category is labeled "Program" and comprises various sets of x86 code taken from two sources: Github and Ubuntu 16.04. The other category is called "Others" and consists of various document files and portions of data extracted from them. The "Others" category contains "CFB," "OOXML," and "PDF" files. CFB stands for compound file binary [15], and it is used as a container like the FAT16 file system. CFB is used in files with the extensions ".doc," ".xls," ".ppt," ".jtd" (used by the "Ichitaro" Japanese word processor), ".hwp" (used by the "Araea Han-geul" Korean word processor), and so on. OOXML stands for Office Open XML [11], which is a zip container in reality. OOXML is used in ".docx," ".xlsx," and ".pptx" files. PDF stands for portable document format [10], which has the extension ".pdf." For each category and source or file type, we constructed three types of datasets: the whole files, 256-byte blocks extracted from these files, and 2048-bit segments of code extracted from the files. Table 1 shows an outline of our datasets. How to make our datasets is shown in Figure 5.

TABLE 2. The keyword list for each label.

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

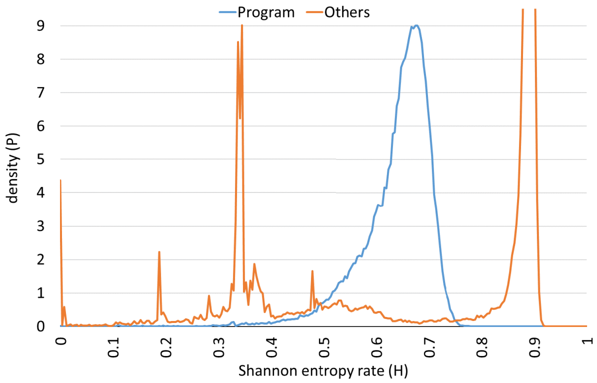

FIGURE 6. Distributions of the Shannon entropy rate for blocks in the two categories of dataset.

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

A. FILE

The following procedure is conducted for making the "File" dataset in the "Program:GitHub" category.

- Gather various C/C++ source code files from GitHub [3]
- Compile these files into x86 object files by using gcc [26]
- Extract only the native code from these object files.

To make the file dataset in the "Program:Ubuntu" category, we extracted program code from the elf files in the "/bin" and "/sbin" directories of Ubuntu 16.04 using the header information.

Finally, to make the file datasets in the "Others" category, we used a search engine to gather various open-source document files. Table 2 shows the keywords used for this search. We downloaded document files from the beginning of the list of search results. We then checked these download files using VirusTotal [29], and we removed suspicious files that were detected as malware.

B. BLOCK

"Block" datasets are made by extracting 256-byte blocks by random sampling from every file in a "File" dataset. We calculated Shannon entropy rates (Equation (1)) for each block in the "Block" datasets. The distributions of the entropy rates for blocks in the "Program" and "Others" categories are shown in Figure 6.

In Figure 6, "Program" has a peak near 0.7 on the x-axis. The program binary looks like a uniformly random byte sequence, but in fact, the machine language instruction sequence is slightly biased such that it contains many null bytes for various reasons such as immediate operands. Therefore, the expected entropy rate of the machine codes is lower than that of uniformly distributed data such as encryption/compression. On the other hand, "Others" has two peaks near 0.3 and 0.9 on the x-axis. The first peak is mainly caused by plain text strings. The document file contains various plain text strings. For example, the body text stored in a doc file, additional information stored before various object bodies in a PDF file. The second peak is caused by mainly encryption/compression typically found in images in document files or zip compressed data in docx containers.

Program

```
0000:83EC14          SUB ESP, 0x14
0003:53              PUSH EBX
0004:55              PUSH EBP
0005:8B6C2428        MOV EBP, [ESP+0x28]
0009:8BC5            MOV EAX, EBP
000B:2D80000000      SUB EAX, 0x80
0010:742D            JZ 0x3F
0012:83E840          SUB EAX, 0x40
0015:741C            JZ 0x33
0017:83E840          SUB EAX, 0x40
001A:740B            JZ 0x27
001C:5D              POP EBP
001D:B8E0FFFFFF      MOV EAX, 0xFFFFFFE0
0022:5B              POP EBX
0023:83C414          ADD ESP, 0x14
0026:C3              RET
```

CFB

```
0000:D0CF            ROR BH, 0X1
0002:11E0            ADC EAX, ESP
0004:A1B11AE100      MOV EAX, [0XE11AB1]
0009:0000            ADD [EAX], AL
000B:0000            ADD [EAX], AL
000D:0000            ADD [EAX], AL
000F:0000            ADD [EAX], AL
0011:0000            ADD [EAX], AL
0013:0000            ADD [EAX], AL
0015:0000            ADD [EAX], AL
0017:003E            ADD [ESI], BH
0019:0003            ADD [EBX], AL
001B:00FE            ADD DH, BH
001D:FF09            DEC DWORD [ECX]
001F:0006            ADD [ESI], AL
0021:0000            ADD [EAX], AL
```

FIGURE 7. A sample of disassembling a native-code or CFB file.

C. CODE

The following procedure is used to make "Code" datasets. First, we treat the files of a "File" dataset as x86 code files, whether they come from the "Program" category or the "Others " category. Second, we separate these files into "instructions" (i.e., disassemble the real or pretended x86 code). Third, we convert each instruction into a 128-bit fixed-length instruction by padding it with "0x00." Finally, packing 16 randomly selected fixed-length instructions into one set, we make a 2048-bit sequence.

The reason we padded instructions to 128 bits (16 bytes) is the following. According to the specification of the x86 architecture [8], 15 bytes is basically the maximum length of one instruction. Thus we padded each instruction with null bytes to convert into a fixed length of 16 bytes (one byte larger than the maximum instruction length) and combined 16 of these padded instructions to form a code segment that has a convenient length for our analysis. Although 15 bytes is the basic maximum length of instruction, longer instructions could appear in theory (particularly when the file being interpreted as x86 code is actually a document file).[3] However, we did not find any instruction longer than 15 bytes in our experiment. A sample of disassembling x86 native code and a CFB file is shown in Figure 7. The average of the lengths of each "instruction" is 2.95 bytes for the" Program" category

[3]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).

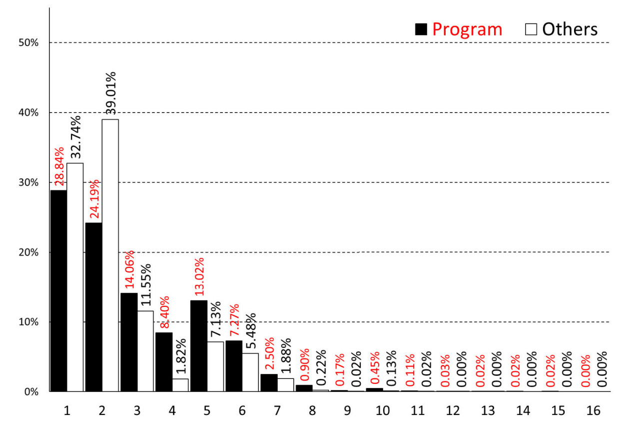

FIGURE 8. Distributions of instruction lengths in files from the "Program" and "Others" categories. The total number of instructions in each category is over 600,000.

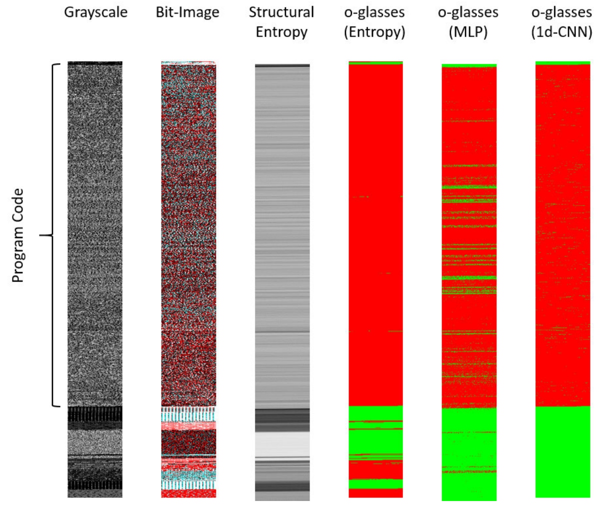

FIGURE 9. The result of visualizing "notepad.exe." In the case of the grayscale image, we adopted the conventional conversion techniques [2], [20] except for fixing a 128-pixel (byte) image width. In the case of the structural entropy image, we selected 256 bytes for the block size. Our methods classify the input block as either "Program" (red) or "Others" (green). The block sizes are 256 in `o-glasses` (Entropy) and `o-glasses` (MLP), and 16 instructions in `o-glasses`(1d-CNN). The block shifts are 1 byte in all the methods.

and 2.38 bytes for the" Others" category. As shown in the figure, there are various lengths of instruction in x86 CPU architecture, which appear to have no regular pattern. The frequencies of each instruction length in files from each category are shown in Figure 8.

In the case of variable-length instruction set architecture, we can make a sample not only at actual instruction offsets but at any point, including operand. This operation derives from the *self-repairing* [22] property of the x86 code instruction set. We can obtain correct disassembled instruction sequence after very few incorrect disassembled instructions, even if we disassemble binary sequences at wrong instruction offsets.

V. PROPOSED VISUALIZATION METHODS

In this section, we propose three visualization methods: `o-glasses` (1d-CNN), `o-glasses` (MLP), and `o-glasses` (Entropy), which are based on a 1d-CNN, an MLP, and entropy, respectively. These methods classify the input block as either "Program" or "Others" and visualize the input block as an image in two colors ("Program" is shown in red, "Others" is shown in green). Figure 9 shows the result of visualizing "notepad.exe" using our methods and three conventional methods. The details of each method are as follows.

A. `o-glasses` (1D-CNN)

First, we consider the `o-glasses` based on a 1d-CNN.

1) LOCAL RECEPTIVE FIELD FOR THE X86 INSTRUCTION SET

We aim to make our model specialize in recognition of native program code. If you directly input binary, such as an x86 instruction set, into a convolutional layer, you cannot identify single instructions as expected. The input data consist of instructions serialized as one-dimensional data. We convert the input data into N-bit fixed-length instructions to obtain features of the instructions. Additionally, the kernel field size and the stride should be adjusted to N. We selected 128 (16 bytes) as the value of N, because this is a convenient size that is just larger than the maximum size (15 bytes) of an x86 instruction. In the 1d-CNN, the first layer consists of local receptive fields against each instruction. Therefore, it is expected that the next layer obtains the relationships among instructions.

2) OUR 1D-CNN

In order to compare our 1d-CNN with other methods, we designed our network as simple as possible. Hyper parameters are chosen to produce the best results through several experiments with different parameters. The whole of our 1d-CNN is shown in Figure 10. We serialize a set of 16 fixed-length instructions into an array of 2048 bit values as input data. The first layer is a convolutional layer (Bit-CNN). We apply 96 layers of 128 bit-filters to a 2048 input volume. Choosing a stride of 128, the output volume is 16×96. The second layer is also a convolutional layer (Instruction-CNN). We apply 256 2-filters to a 16×96 input volume with a stride of 1. We expect that the second layer will obtain the features of the relationship between two adjacent instructions. Our 1d-CNN does not contain any Pooling layer. The 3rd to the 5th layers are fully connected. Their output volumes are 400, 400, and 2, respectively. We add two batch normalization [9] layers before the 1st and 2nd fully connected layers to speed up and stabilize the learning process. After each layer except the last one, we apply a ReLU [4] layer. The ReLU layer applies the function $f(u) = \max(u, 0)$ to all of the values

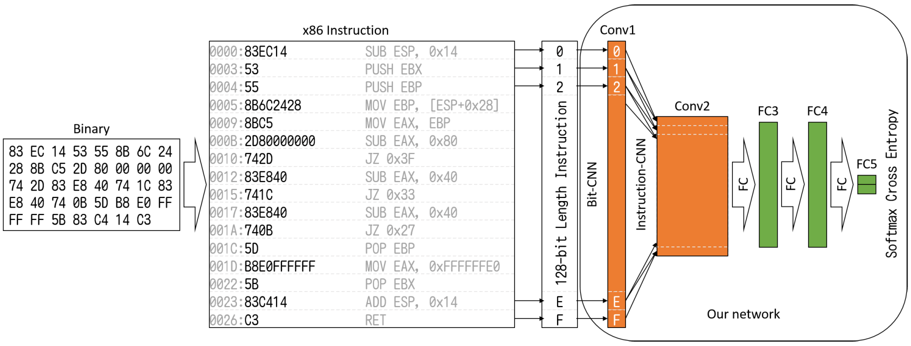

FIGURE 10. Outline of our 1d-CNN.

in the input volume. The softmax function is used in the final layer of our network.

$$y_k \equiv z_k^{(L)} = \frac{\exp(u_k^{(L)})}{\sum_{j=1}^{K} \exp(u_j^{(L)})}, \tag{2}$$

where $K = 2^4$ and $k \in \{1, 2\}$. Our network is trained under a cross-entropy regime. The cross-entropy function for one training sample (x_n, t_n) for $n \in [1, N]$ is

$$E_n(\mathbf{w}) = -\sum_{k=1}^{K} t_{nk} \log y_{nk}(x_n, \mathbf{W}), \tag{3}$$

where the input data is $x_n \in \{0, 1\}^{2048}$, the true label is $t_n \in \{0, 1\}^K$, and the number of output units is K. The sum of the errors E_n calculated from each training sample is the total error function E:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}), \tag{4}$$

where the number of samples is N.

3) STOCHASTIC GRADIENT DESCENT

We use the stochastic gradient descent (SGD) method to minimize the error function in the backpropagation algorithm. To economize on the computational cost of each iteration, SGD samples a subset of summand functions at every step. This is very effective in the case of large-scale machine learning problems.

The current weight $\mathbf{w}^t$ is updated to $\mathbf{w}^{t+1}$ using the following equation.

$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}=\mathbf{w}^t}, \tag{5}$$

where η is the learning rate.

A compromise between computing the true gradient and the gradient of a single example is to compute the gradient against more than one training example (called a "mini-batch") at each step.

$$E_m(\mathbf{w}) = \frac{1}{|N^m|} \sum_{n \in N^m} E_n(\mathbf{w}), \tag{6}$$

where N^m is a subset of the index set $\{1, \ldots, N\}$ such that $\bigsqcup_m N^m = \{1, \ldots, N\}$ and $N^{m_i} \cap N^{m_j} = \emptyset$ for $i \neq j$.

[4] $K = 2$ is a special case where a simple sigmoid function can be applied. Here, we chose Softmax function for ease of extending to multi-class classification ($K > 3$). Both are equivalent when $K = 2$

TABLE 3. Performance of our methods to detect x86 code.

Our methods	F1	Precision	Recall
Based on 1d-CNN	0.9995	0.9999	0.9992
Based on MLP	0.9840	0.9830	0.9851
Based on Entropy (Range:0.408–0.753)	0.9266	0.8725	0.9879

B. o-glasses *(MLP)*

Like the previous method, this method focuses on each block of the target file. The input data size is one block (a 256-byte sequence), and the block shift is 1 byte. The network containing hidden layers and the output layer is the same as fully connected layers 3–5, shown in Figure 10, for the 1d-CNN (see Section V-A).

C. o-glasses *(Entropy)*

The method detects program code based on whether the entropy of the block lies within a given range. When appropriate range criteria are selected, this method achieves reasonable accuracy in the detection of program code.

VI. EVALUATION

A. RECOGNITION PERFORMANCE

We investigated the detection rates of program code by our methods using the training datasets described in Section IV. Table 3 shows an overview of the results.

In the comparison of the different algorithms, we use the F-measure (F1) defined by

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}. \tag{7}$$

In this calculation, precision is given by

$$Precision = \frac{TP}{TP + FP}, \tag{8}$$

TABLE 4. Confusion matrix.

Predicated Condition	True Condition Program	True Condition Others
Program	TP	FP
Others	FN	TN

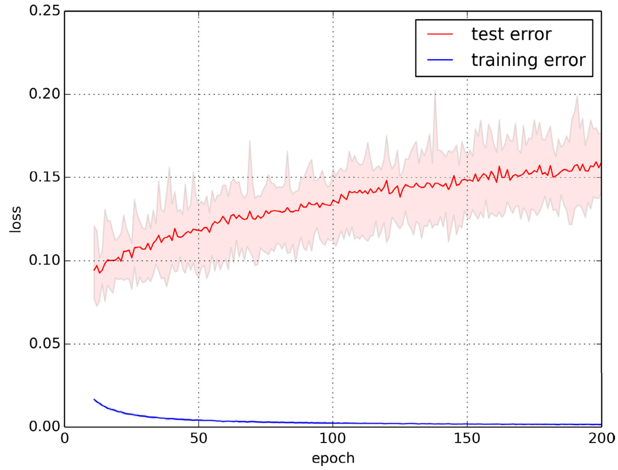

FIGURE 11. Learning curve of o-glasses (MLP).

and recall is given by

$$Recall = \frac{TP}{TP + FN}, \tag{9}$$

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 (see Table 4).

1) o-glasses (Entropy)

We examined many ranges for the entropy rate-based binary classifier and chose the range that gives the maximum F-measure for the training dataset. The F-measure for entropy in Table 3 was calculated using this "range" parameter (also shown in the table) against the test dataset.

2) o-glasses (MLP) AND o-glasses (1D-CNN)

To train and test our network, 10-fold cross-validation was used. After 200 epochs, we calculated the F-measure, the precision, and the recall of the test data.

Here is our parameter configuration:

- learning rate(η) $= 0.001$
- mini-batch size $= 100$

The learning curves of the error are shown in Figs. 11 and 12. The blue areas indicate the range of possible values of the training errors in the 10-fold cross-validation process. The solid blue lines indicate the average of the training errors in the 10-fold cross-validation process. The red areas indicate the range of possible values of the test errors in the 10-fold cross-validation process. The solid red lines indicate the average of the test errors in the 10-fold cross-validation process. From this figure, it can be seen that our 1d-CNN method does not cause over-fitting.

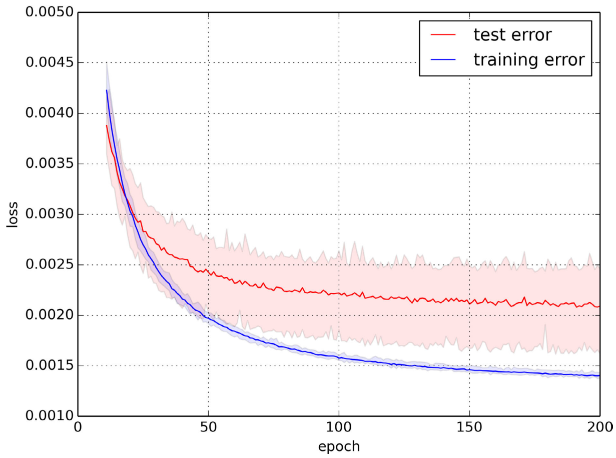

FIGURE 12. Learning curve of o-glasses (1d-CNN).

B. EXPERIMENTS WITH MALICIOUS DOCUMENTS

In this section, we visualize three malicious document files to discuss the effectiveness of our methods. Table 5 shows the overview of the malicious document files. The first malicious document file contains 127 bytes of x86 code. The second malicious document file contains 29 bytes of x86 code. The third malicious document file does not use vulnerabilities, and does not have any x86 code. These files are referred to in the following discussion as File 1, File 2, and File 3, respectively.

The parameters used in these experiments are the same as those described in the previous section. After 200 epochs of training using all our datasets, we visualized the three files.

1) FILE 1: CVE-2014-7247

File 1 contains a compressed executable. If we can analyze the file dynamically, it is easy to output the executable. However, this file is a.jtd document file for Ichitaro, which is a Japanese word processing software similar to Microsoft Word. The old version of Ichitaro had a vulnerability called CVE-2014-7247, which this document targets. So, we need the old version for dynamic analysis. When we do not have the old version, we must find the decoder for the executable file to output it. Therefore, we need to find the shellcode.

This document file contains 127 bytes of x86 code. The code is split two sequences; the size of the first sequence is 77bytes, and the size of the second sequence is 50 bytes. The first sequence is code for jumping to the second sequence. OMS could detect the entry point of the first sequence code.

Figure 13 shows the result of visualizing File 1. The o-glasses (1d-CNN) method shows an x86 code sequence at almost the same location as the first sequence. However, the method could not locate the second sequence.

2) FILE 2: CVE-2012-0158

File 2 contains an executable file encoded with a 2-byte-key xor. This document file is a Word (.doc) document file and attacks a vulnerability called CVE-2012-0158.

TABLE 5. The overview of the malicious document files and the detection results of shellcode by OfficeMalScanner (OMS.) The rightmost check shows that OMS could detect the shellcode in File 1. While o-glasses could indicate the locations of the suspected shellcodes in File 1-2 (as demonstrated in Figures 13-14).

#	Hash (MD5)	File type	Exploit	File size	x86 code size	OMS
1	d191fb6dd45eb6fbfff4e7b106478de4	.jtd	CVE-2014-7247	282,679	127	✓
2	ff3a9950147507743c312eec7e2a29ae	.doc	CVE-2012-0158	209,456	29	
3	7a7a281d66cce2a3ab2b92cdbf7b7e33	.doc	None (VBA)	37,888	0	–

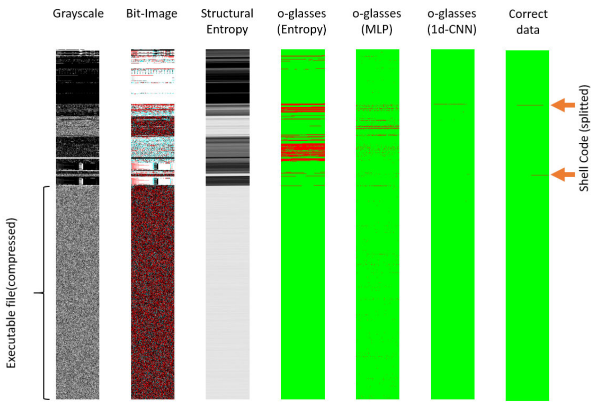

FIGURE 13. Results of visualizing File 1 by various methods.

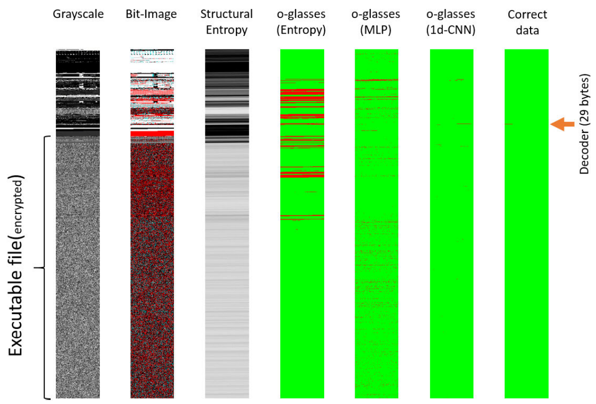

FIGURE 14. Results of visualizing File 2 using various methods.

This document file contains only 29 bytes of x86 code. This is the code which we mentioned in the Introduction. OMS could not detect the shellcode but could detect a decoy document embed the document file. Figure 14 shows the results of visualizing File 2. The o-glasses (1d-CNN) method could not locate the decoder. However, it found a sequence of "nop" instructions located just before the decoder.

3) FILE 3: VBA SCRIPT DONWLOADER

Unlike the other two files, File 3 does not contain any executable file. Additionally, this document file does not attack any vulnerabilities. Instead, a VBA script in this document file downloads an executable file from the Internet and runs it. Therefore, this document file does not contain any x86 code. Hence, OMS could not detect the shellcode but could detect a API string which is often used in exploits. As shown in

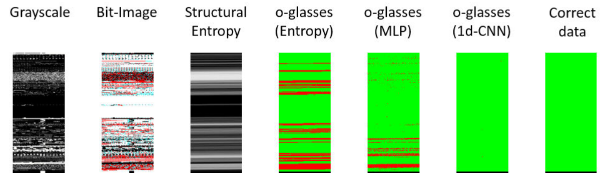

FIGURE 15. Results of visualizing File 3 by various methods.

Figure 15, o-glasses (1d-CNN) correctly reports no x86 code in this document, while the other methods report many false-positive blocks. Thus, human examiners can confidently focus on the positive blocks reported by o-glasses (1d-CNN) to search for real shellcode in malicious documents.

VII. DISCUSSION

In this section, we discuss the usage and limitations of our methods, and areas for future work.

A. EASILY COLLECTIBLE TRAINING DATASETS

One of the most significant problems using machine learning is how to prepare the training dataset. Even an excellent model cannot demonstrate its performance without large samples. Many studies of malware using machine learning have sometimes struggled to collect samples because they need hard-to-collect malware. In contrast, our approach does not need malware for the training dataset. Since all we need to collect is x86 code and normal document files, it is possible for anyone to create training datasets from easily accessible sources. Surprisingly, in spite of this fact, our method, o-glasses(1d-CNN), can find the locations of shellcode almost exactly. Therefore, our proposed methods suggest a possible beneficial effect for professional malware analysis. On the other hand, some shellcode is known to contain garbage code. We did not consider such cases, and therefore our dataset needs to be improved.

B. HIGH RECOGNITION RATE FOR X86 CODE

In this paper, we have presented a method of recognizing program code in document files using a 1d-CNN. Using a local receptive field and weight sharing, our 1d-CNN can capture important features of instructions. Thus, even if the input instruction sequence is shifted, our network can recognize program code with a high degree of success, as measured by the F-measure rate.

The result of our experiments, inputting 16 opcodes into our network, is that the F-measure rate reaches about 99.95%.

While this value seems to be very high at first glance, it means that, when the target file size is 100 KB, about 50 bytes of noise is generated in the visualization result. When looking for a small program like shellcode, this noise becomes an obstacle to analysis. Although our method has already achieved a real-use level of performance for human analysts, it still needs further improvement for automatic shellcode detection.

C. VISUAL ANALYSIS TO SUPPORT ANALYSTS

In this paper, we visualized several malicious document files and showed that we could find some small programs like shellcode. Furthermore, in the case of a document file which does not contain any x86 native code, other methods do not provide convincing evidence that x86 native code was not present. But, by using our method, we can be fairly confident that a file does not contain x86 native code.

However, some malicious document files do not contain x86 native code, but contain interpreted code such as JavaScript. Our methods do not cover such files. For these files, it is necessary to analyze the malware by another method, which may be combined with our `o-glasses` (1d-CNN) method.

D. BLACK BOX PROBLEMS

Deep learning has achieved success in various fields, including binary analysis. However, many methods containing ours cannot form the basis of the decision, and they are often called "black boxes". This problem may become a barrier to the use of these methods in fields requiring trust. In such a background, XAI (Explainable Artificial Intelligence) researches are famous recently. For example, Guo *et al.* proposed LEMNA [6], which can determine the relevance of features contributing to a prediction by approximating the decision function of a neural network. Therefore, a combination of our method and XAI approaches might defeat the Black Box problemes. The study of the combination is part of our future work.

VIII. CONCLUSION

In this paper, we proposed a 1d-CNN for detecting program code in document files. We observed that a local receptive field for a 128-bit fixed-length instruction is effectively formed in the first layer of our network. We can balance both high precision rate and high recall rate for detecting program code by using our network. Our network can narrow down a target for human static analysis of unknown malware. Future work includes increasing the number of malicious document files used to check the validity of our proposed method. Another task is to combine our network with various analysis methods for unknown malware.

REFERENCES

[1] F. Boldewin. *Analyzing Msoffice Malware With Officemalscanner*. [Online]. Available: http://www.reconstructer.org/papers/Analyzing%20MSOffice%20malware%20with%20OfficeMalScanner.zip

[2] G. Conti *et al.*, "A visual study of primitive binary fragment types," Black Hat, Las Vegas, NV, USA, Tech. Rep., 2010, pp. 1–17.

[3] GitHub. *The World's Leading Software Development Platform*. Accessed: Jan. 21, 2020. [Online]. Available: https://github.com/

[4] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, 2011, pp. 315–323.

[5] K. Goto. (1998). *Stirling*. [Online]. Available: https://www.vector.co.jp/soft/win95/util/se079072.html

[6] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "LEMNA: Explaining deep learning based security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: ACM, Oct. 2018, pp. 364–379.

[7] K. Han, J. H. Lim, and E. G. Im, "Malware analysis method using visualization of binary files," in *Proc. Res. Adapt. Convergent Syst. (RACS)*. New York, NY, USA: ACM, 2013, pp. 317–321.

[8] Intel. (2016). *Intel 64 and IA-32 Architectures Software Developer Manuals*. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," Feb. 2015, *arXiv:1502.03167*. [Online]. Available: https://arxiv.org/abs/1502.03167

[10] *Document Management—Portable Document Format—Part 1: PDF 1.7*, document ISO 32000-1:2008, 2008. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502

[11] *Information Technology—Document Description and Processing Languages—Office Open XML File Formats—Part 1: Fundamentals and Markup Language Reference*, document ISO/IEC 29500-1:2012, 2012.

[12] K. Kancherla and S. Mukkamala, "Image visualization based malware detection," in *Proc. IEEE Symp. Comput. Intell. Cyber Secur. (CICS)*, Apr. 2013, pp. 40–44.

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

[14] 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 Comput.*, vol. 1, no. 4, pp. 541–551, Dec. 1989.

[15] Microsoft. *[MS-CFB]: Compound File Binary File Format*. Accessed: Jan. 21, 2020. [Online]. Available: https://msdn.microsoft.com/ja-jp/library/dd942138.aspx

[16] MITRE. *CVE-2004-1308*. Accessed: Jan. 21, 2020. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1308

[17] MITRE. *CVE-2011-3402*. Accessed: Jan. 21, 2020. [Online]. Available: https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2011-3402

[18] MITRE. *CVE-2016-8332*. Accessed: Jan. 21, 2020. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8332

[19] MITRE. *CVE-2017-5133*. Accessed: Jan. 21, 2020. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5133

[20] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proc. 8th Int. Symp. Vis. Cyber Secur. (VizSec)*, New York, NY, USA: ACM, 2011, p. 4.

[21] L. Nataraj, V. Yegneswaran, P. Porras, and J. Zhang, "A comparative assessment of malware classification using binary texture analysis and dynamic analysis," in *Proc. 4th ACM Workshop Secur. Artif. Intell. (AISec)*. New York, NY, USA: ACM, 2011, pp. 21–30.

[22] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*. New York, NY, USA: ACM, 2010, pp. 21–28.

[23] I. Sorokin, "Comparing files using structural entropy," *J. Comput. Virol.*, vol. 7, no. 4, pp. 259–265, Nov. 2011, doi: 10.1007/s11416-011-0153-9.

[24] Talos. *OpenJPEG JPEG2000 MCC Record Code Execution Vulnerability*. Accessed: Jan. 21, 2020. [Online]. Available: https://www.talosintelligence.com/reports/TALOS-2016-0193/

[25] Talos. *Vulnerability Spotlight: Google PDFium Tiff Code Execution*. Accessed: Jan. 21, 2020. [Online]. Available: http://blog.talosintelligence.com/2017/10/GooglePDFium-Vulnerabilitiy.html

[26] The GCC Team. *Gcc, the Gnu Compiler Collection*. Accessed: Jan. 21, 2020. [Online]. Available: https://gcc.gnu.org/

[27] Trend Micro. (2015). *Targeted Attack Campaigns and Trends: 2014 Annual Report*. [Online]. Available: http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-targeted-attack-trends-annual-2014-report.pdf
[28] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, "Combining static and dynamic analysis for the detection of malicious documents," in *Proc. 4th Eur. Workshop Syst. Secur. (EUROSEC)*, Salzburg, Austria, Apr. 2011, p. 4.
[29] VirusTotal. *VirusTotal*. Accessed: Jan. 21, 2020. [Online]. Available: https://www.virustotal.com/
[30] T. Xu, M. Xu, Y. Ren, J. Xu, H. Zhang, and N. Zheng, "A file fragment classification method based on grayscale image," *JCP*, vol. 9, no. 8, pp. 1863–1870, 2014.

YUHEI OTSUBO was born in Fukuoka, Japan, in 1981. He received the B.S. degree from The University of Tokyo, Japan, in 2005, the M.S. degree from the National Graduate Institute for Policy Studies, Japan, in 2012, and the Ph.D. degree in informatics from the Institute of Information Security, Kanagawa, Japan, in 2016.

Since 2005, he has been a Technical Official with the National Police Agency (NPA), Japan. From 2012 to 2014, he was with the National Information Security Center as a seconded Staff. He is currently a Technical Official with NPA. His research interest includes information security. He was a Speaker of Black Hat USA 2016.

AKIRA OTSUKA (Member, IEEE) was born in Osaka, in 1966. He received the B.E. and M.E. degrees from Osaka University, in 1989 and 1991, respectively, and the Ph.D. degree from The University of Tokyo, in 2002.

Since 2002, he has been a Postdoctoral Fellow and a Cooperative Researcher with The University of Tokyo. From 2003 to 2005, he was a member of Cryptographic Technique Monitoring Subcommittee at CRYPTREC. Since 2005, he has also been with the National Institute of Advanced Industrial Science and Technology (AIST), serves as a Leader of Research Security Fundamentals, from 2006 to 2010. From 2007 to 2014, he was a Visiting Professor at Research and Development Initiative, Chuo University. Since 2017, he has also been a Professor with the Graduate School of Information Security, Institute of Information Security.

MAMORU MIMURA received the B.E. and M.E. degrees in engineering from the National Defense Academy of Japan, in 2001 and 2008, respectively, the Ph.D. degree in informatics from the Institute of Information Security, in 2011, and the M.B.A. degree from Hosei University, in 2014.

From 2001 to 2017, he was a member of the Japan Maritime Self-Defense Force. From 2011 to 2013, he was with the National Information Security Center. Since 2014, he has been a Researcher with the Institute of Information Security. Since 2015, he has been with the National Center of Incident readiness and Strategy for Cybersecurity. He is currently an Associate Professor with the Department of Computer Science, National Defense Academy of Japan.

TAKESHI SAKAKI received the B.S., M.S., and Ph.D. degrees from The University of Tokyo, Japan, in 2004, 2006, and 2013, respectively. He is currently the Director of the Research and Development Department, Hottolink, Inc., and also a Visiting Researcher with The University of Tokyo. His research interests include natural language processing, web mining, artificial intelligence, and computational social science.