Comparison of Three Deep Learning-based Approaches for IoT Malware Detection

Khanh Duy Tung Nguyen[1], Tran Minh Tuan[2], Son Hai Le[1], Anh Phan Viet[1], Mizuhito Ogawa[3], and Nguyen Le Minh[3]

[1]Le Qui Don Technical University, Hanoi
Email: tungkhanhmta@gmail.com, lehaisonmath6@gmail.com, anhpv@mta.edu.vn

[2]University of Engineering and Technology, Vietnam National University, Hanoi
Email: tranminhtuan@vnu.edu.vn

[3]Japan Advanced Institute of Science and Technology
Email: {mizuhito, nguyenml}@jaist.ac.jp

***Abstract*—The development of IoT brings many opportunities but also many challenges. Recently, increasingly more malware has appeared to target IoT devices. Machine learning is one of the typical techniques used in the detection of malware. In this paper, we survey three approaches for IoT malware detection based on the application of convolutional neural networks on different data representations including sequences, images, and assembly code. The comparison was conducted on the task of distinguishing malware from nonmalware. We also analyze the results to assess the pros/cons of each method.**

I. Introduction

Malware threat becomes more serious every year. According to the McAfee report in the first quarter of 2018 [1], the averages are 45 million malicious files, 57 million malicious URLs, and 84 million malicious IP addresses per day. We focus on IoT malware, which is doubled each year since 2015.

For PC malware, commercial antivirus software investigates syntax patterns that are analyzed from known malicious samples, often with machine learning techniques, e.g., finding characteristic bytes n-gram [2]. However, recent PC malware evolved with advanced obfuscation techniques [3], [4], which make difficult to identify semantical similarity from syntax patterns [5]. Actually, Symantec confessed that antivirus software can detect only 45% of PC malware on May 2015. Dynamic analysis in the sandbox, e.g., CWSandbox [6], ANUBIS [1], is another typical approach, which observes the behaviors on registers [7], API calls [8], and the memory [9]. However, anti-debugging and anti-tampering techniques may recognize the sandbox, and the trigger-based behavior, e.g., malicious actions occur at the specific timing, will be rarely detected. One of the authors developed a binary code analyzer *BE-PUM* based on dynamic symbolic execution on x86 [10]. It overcomes obfuscation techniques and provides precise disassembly of malware. The drawback is the heavy load by the nature of dynamic symbolic execution.

Compared to PC malware, IoT malware often does not use obfuscation techniques. Thus, we can apply quite immediately statistical methods like machine learning and easily disassemble by using commercial disassemblers, such as `IDApro`.

This paper compares three different convolutional neural networks on 1,000 real IoT malware samples for x86, collected by `IoTPOT`[2] of Yokohama National University.

The first model adapts the features of fixed-sized byte sequences, which is basic and easy to implement. The second uses the features of fixed-sized color images on AlexNet CNN. From the entropy feature of a binary code, the image is generated by the Hilbert curver. The last model adapts the features of the assembly instruction sequences, which is generated by `objdump`. Different from the standard CNN, this model accepts variable-sized sequences. Thus, the training requires more time and effort. We compare the effectiveness among models by experimental and address the future directions.

The rest of the paper is as follows. Section 2 is for preliminaries. Three CNN modeling for detecting IoT malware are presented in Section 3. The experiments are presented in Section 4. Section 5 concludes with the discussion.

II. Preliminaries

A typical convolutional neural network (CNN) includes three types of layers including convolution, pooling, and fully-connected. Wherein, the success of the network mostly depends on convolutional layers

[1]http://anubis.seclab.tuwien.ac.at

[2]https://github.com/IoTPOT/IoTPOT

978-1-5386-6113-0/18/$31.00 ©2018 IEEE

that are responsible to automatically learn data features from the low level to high level of abstraction. Next, we will describe a simplest CNN from the input to the output layer.

A. Feature extraction and data structures

There are many kinds of features to detect whether the file is malicious. File features can be extracted from contents and execution traces/logs, and stored in the data structure, which is classified into either fixed-sized or variable-sized data structures.

Fixed-sized data structure means that different files are represented in the data structure of the same size, e.g., vectors with the same dimension and images. Variable -sized data structure means that different files are left in various sizes, e.g., sequences, trees, and graphs. With variable-sized data structure, classical machine learning models need to be adapted to fit them.

B. Convolutional layer

All three models use a layer called the convolutional layer, which is the core block of a Convolutional Neural Network (CNN) [11], [12]. CNN is known to be effective, especially on the image classification. The basic idea of the convolutional is to combine the neighborhoods to emphasize local characteristics. The idea is that localized concepts among points close to each other will share more correlations. For example, in the image, pixels next to each other will be likely similar unless there is an edge, and an edge is an important feature.

According to [13], many studies have tried to generalize CNN on other data structures, such as acoustic data [14], videos [15] and Go boards [16].

The convolution layers are composed either sequentially or parallelly. The sequential composition transfers the output of a convolution layer to the input of the next layer in the network but in [17]. The parallel composition combines the outputs of several convolution layers to the single output, which intends to avoid the vanishing gradient problem caused by sigmoid activation functions.

C. Pooling layer

A convolutional layer is often followed by a pooling layer, whose function summarizes the neighborhoods to reduce the size of the representation and the number of parameters in the network, and to control over-fitting. Examples of the neighborhoods are close pixels of an image, adjacent nodes of a graph, and close time regions of acoustic data.

There are two typical types of the pooling layers: a local pooling layer and a global pooling layer. The operation of the local pooling layer is depicted in Fig. 1, in which the output size depends on w, h, the size of sliding window, and the padding. Recently, the global pooling layer is considered to minimize overfitting by drastically reducing the number of the parameters in the model. For example, Fig. 2 shows the reduction of the dimensions $h * w * d$ to $1 * 1 * d$ in a global pooling layer. It reduces by mapping each $h*w$ features to their mean value. It is often used at the backend of a CNN with dense layers to get a shape of the data, instead of the flattening. Another example of the global pooling layer is to transform the variable-sized data into the fixed-sized data as in Fig. 2, in which the size of the output is always $1 * 1 * d$ independent from w and h.

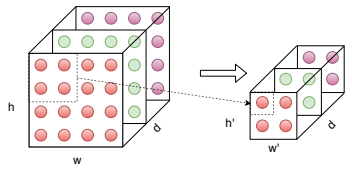

Fig. 1. The summerization of the local pooling layer

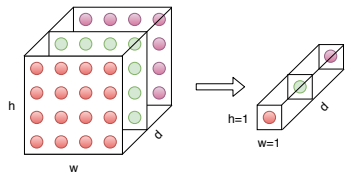

Fig. 2. The summerization of the global pooling layer

D. Fully-connected layer

The output from the convolutional layers represents high-level features in the data. While the output can be flattened and connected to the output layer, adding a fully-connected layer is a cheap way to learn these features. Neurons in the fully connected layer have the full connections to all activations in the previous layer, as regular Neural Networks. The fully-connected layer is often associated with a softmax layer, which outputs the probability of each class.

III. Three approaches

A. CNN on byte sequences (CNN_SEQ)

Fig. 3 shows CNN_SEQ, inspired by MalConv [18]. CNN_SEQ is simple and easy to implement and scale.

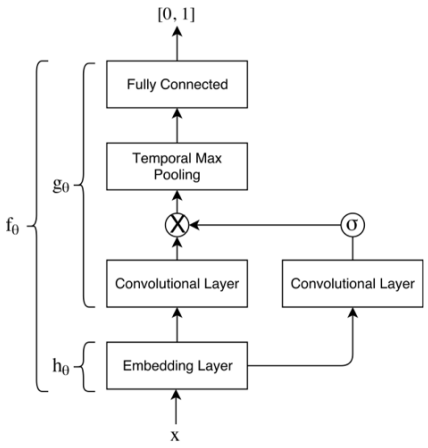

Fig. 3. Architecture of CNN_SEQ for IoT malware detection

1) Feature extraction and data representation: Let $X = 0, ..., 255$ be the integer representation of a byte. A binary code is composed of the k bytes data $(x_1, ..., x_k \in X)$,

- if the binary code is shorter than k bytes, zeros are padded as the suffix until k bytes.
- if the binary code is larger than k bytes, they are selected from the section with the executable and the writable permissions, e.g., (.init, .text, .data), and set lower priority for read-only data segments, e.g., (.rodata, .bss). They are extracted by the "readelf -section" tool (Fig. 4). Each byte x_j is weighted as $z_j = \Phi(x_j)$ (where the mapping Φ is learned by the network during training), and composes a matrix Z.

```
$ greadelf --sections IoTMalwares/00b8e12b9f5de07a0f23519de311d147
There are 21 section headers, starting at offset 0x2c478:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .reginfo          MIPS_REGINFO    004000b4 0000b4 000018 18   A  0   0  4
  [ 2] .init             PROGBITS        004000cc 0000cc 00008c 00  AX  0   0  4
  [ 3] .text             PROGBITS        00400160 000160 01b110 00  AX  0   0 16
  [ 4] .fini             PROGBITS        0041b270 01b270 00005c 00  AX  0   0  4
  [ 5] .rodata           PROGBITS        0041b2d0 01b2d0 009464 00   A  0   0 16
  [ 6] .eh_frame         PROGBITS        00424734 024734 000004 00   A  0   0  4
  [ 7] .ctors            PROGBITS        00464738 024738 000008 00  WA  0   0  4
  [ 8] .dtors            PROGBITS        00464740 024740 000008 00  WA  0   0  4
  [ 9] .jcr              PROGBITS        00464748 024748 000004 00  WA  0   0  4
  [10] .data.rel.ro      PROGBITS        0046474c 02474c 000a8c 00  WA  0   0  4
  [11] .data             PROGBITS        004651e0 0251e0 004010 00  WA  0   0 16
  [12] .got              PROGBITS        004691f0 0291f0 000574 04 WAp  0   0 16
  [13] .sbss             NOBITS          00469764 029764 000020 00 WAp  0   0  4
  [14] .bss              NOBITS          00469790 029764 0063b4 00  WA  0   0 16
  [15] .comment          PROGBITS        00000000 029764 000c18 00      0   0  1
  [16] .mdebug.abi32     PROGBITS        00000c18 02a37c 000000 00      0   0  1
  [17] .pdr              PROGBITS        00000000 02a37c 002060 00      0   0  4
  [18] .shstrtab         STRTAB          00000000 02c3dc 00009a 00      0   0  1
  [19] .symtab           SYMTAB          00000000 02c7c0 0030f0 10     20 319  4
  [20] .strtab           STRTAB          00000000 02f8b0 00244c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

Fig. 4. Example of the data segment extraction

2) Description model architecture: Two parallel convolutional layers are prepared for processing the matrix Z, in which the activation functions are

$$f(x) = \begin{cases} \text{ReLU (Rectified Linear Unit)} & max(0, x) \\ \text{Sigmoid} & \frac{1}{1+e^{-x}} \end{cases}$$

They are combined through the gating [17], which multiplies element-wise the matrices computed by the two layers. This avoids the vanishing gradient problem caused by sigmoid activation function. The result is forwarded to a temporal max pooling layer, which performs a 1-dimensional max pooling, followed by a fully-connected layer with the ReLU activation. This results a 2-dimensional vector (x_0, x_1). To avoid overfitting, we follow [18] that applies DeCov regularization [19] to minimize the cross-covariance. The last step, the softmax activation, evaluates the probability a_i (for $i = 0, 1$) as follows, where a_0 and a_1 are the probablity of being goodware and malware, respectively. If $a_1 \geq 0.5$, we conclude malware.

$$a_i = \frac{exp(x_i)}{exp(x_0) + exp(x_1)} \quad \text{for } i = 0, 1 \qquad (1)$$

We use tensorflow [20] and keras [21] to deploy the above network.

B. CNN on the color images (CNN_IMG)

For IoT malware detection, previously the conversion to a greyscale image is tried and the accuracy has reached 94% [22]. Instead, we convert a binary code into a fixed-sized color image and AlexNet is used for the data classification.

1) Feature extraction and data representation:

A. *Calculate the entropy of a binary file*: Similar to SEQ_SEQ, let $X = 0, ..., 255$ be the representation of a byte. First, we compute the sequence of the entropy of a byte sequence. The entropy shows how much the data is disordered, and we use Shannon entropy

$$H(x) = -\sum_{i=1}^{255} P(x_i) log_b\ P(x_i) \qquad (2)$$

where x is a sliding window, x_i is the number of occurrences of i in x, and P is the ratio (i.e., $\frac{|x_i|}{|x|}$). We set the size of the sliding window to be 32x32 and the base b to be 10.

B. *Convert entropy to RGB color*: The entropy is converted to a color by following to BINVIS[3],

$$r = 255 * (F(x - 0.5)), \quad g = 0, \quad b = 255 * x^2 \qquad (3)$$

where x is the entropy, $F(x) = (4x - 4x^2)^4$, r, g, b are the red, the green, and the blue values, respectively.

C. *Convert color sequence to image*: A space-filling curve fulfills the 2-dimensional unit square by a bent line, e.g., Zigzag, Z-order, Hilbert (Fig. 5).

We choose Hilbert curver for the *locality preservation*, i.e., keeping the close elements in 1-dimensional as nearer as possible in 2 dimensions. The function `drawmapsquare` in BINVIS is used as Hilbert curve by setting options: *parameter map* = *square*, *size* (of the image) =

[3]https://github.com/cortesi/scurve

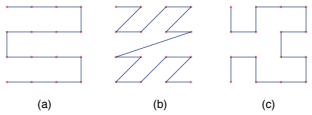

Fig. 5. Curvers (a) Zigzag, (b) Z-order, (c) Hilbert

224, *color* = *entropy*. Fig. 6 shows an example visualization of `busybox` in Ubuntu.

Fig. 6. Visualize the binary file `busybox` with Hilbert curve

2) Description model architecture: After converting a binary file to a square color image, AlexNet is applied [23]. The architecture and the details of each layer in the AlexNet is shown in Fig. 7 and 8, respectively. We use tensorflow [20] and keras [21] to deploy the above network.

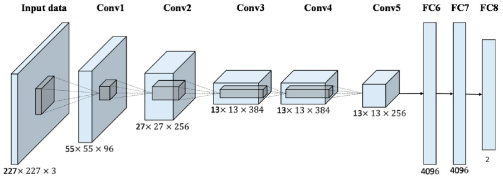

Fig. 7. Alexnet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	2	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

Fig. 8. Details of the layers in AlexNet

C. CNN on assembly sequences (CNN_ASM)

For IoT malware detection, previous studies use handcrafted features (e.g., n-grams and API calls) and different machine learning algorithms. Instead, we directly analyze the assembly code, obtained by a commercial disassembler. The disassembled code is abstracted on register names and memory addresses (which are often changed by the offset) and is tailored as a variable-sized vector. Fig. 9 shows the overview of the processes, which was inspired by [24].

1) Feature extraction and data representation:

- *Disassembling binary files*: The first step disassembles binary executable files to assembly codes. By reading the file header, all of our IoT malware samples are in the ELF file format on multiple CPU architectures (Table I). To disassemble them, we use the `objdump` command in Ubuntu, which is a multi-architectural disassembler. Among them, we target only on x86 in the experiments.

TABLE I
CPU ARCHITECTURE STATISTICS

Architecture	Number
MIPS	2814
ARM	2774
i386	2353
PowerPC	1247
sh-linux	1199
x86	1196
m68k	1153
SPARC	1140

- *Vector Representations*: An instruction may vary the name and operands, in which some may change by the offset and the use of different registers. To abstract such differences, the operands of block names, the register names and the literal values are simplified by the symbols "name", "reg", and "val", respectively. For instance, the instruction `addq $32, %rsp`, is converted to `addq, value, reg`. As in NLP techniques, we encode each word to a 30-dimensional real-valued vector, which is choosen randomly. Then, the i-th instruction is encoded to

$$\bar{x}_i = \frac{1}{C}\sum_{j=1}^{C} \bar{x}_{i,j}, \tag{4}$$

where C is the number of the words and $\bar{x}_{i,j}$ is the encoding of the j^{th} word in the i-instruction, and the sum is computed element-wise. Then, an assembly sequence with n instructions is the contatenation $\bar{x}_{1:n} = \bar{x}_1.\bar{x}_2.\cdots.\bar{x}_n$.

2) Description model architecture: **Convolutional layers** The convolutional layers automatically learn the defect features from instruction sequences. We design a set of the feature detectors (filters) to capture local dependencies in the original sequence. Each filter is a convolution with the sliding window to produce a feature map, i.e., at position i, the feature value c_i^l of the l^{th} filter is:

$$c_i^l = f(W_l \cdot \bar{x}_{i:i+h-1} + b_l) \tag{5}$$

where $W_l \in \mathbb{R}^{h\times k}$, $\bar{x}_{i:i+h-1} = \bar{x}_i.\bar{x}_{i+1}.\cdots.\bar{x}_{i+h-1}$, f is an activation function, and b_l is a bias.

In general, deeper neural networks potentially achieve better performance [12]. However, using many layers leads to more parameters, which require large

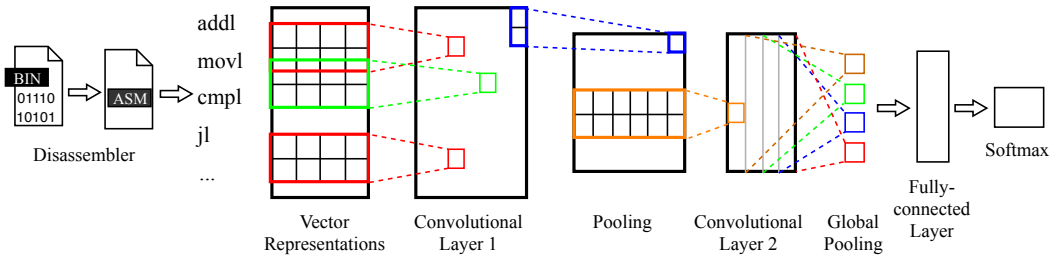

Fig. 9. CNN on assembly instructions for IoT malware detection

datasets for training networks. In this work, two layers of convolutions are prepared for our 1,000 IoT malware samples compiled for x86.

Pooling layers Often, a pooling layer is inserted between successive convolutional layers to reduce the dimensions of feature maps. In our case, the input sequence has up to thousands of instructions, and the feature map length is similar. We choose the max pooling, expecting works better [25].

In the model, the intermediate convolutions are followed by the local max-pooling with the filter size of 2. For the last convolution, the global max-pooling is applied to generate the vector representation for the corresponding view, in which each element is the result of pooling the feature map. We use tensorflow [20] and keras [21] to deploy the above network.

IV. Experiments

A. Dataset

We prepare the dataset for experiments, both IoT malware and goodware in the ELF format.

- 15,000 IoT malware samples are supplied by Prof.Katsunari Yoshioka (Yokohama National University). They run on various platforms, such as the ARM, MIPS, and x86. For experiments, we select 1,000 malware samples of x86 binaries.
- 1,000 goodware samples are taken from x86 binaries of Ubuntu 16.04.

We mix all of them in the single dataset. Then, we randomly select 5 parts and evaluate by the 5-fold cross-validation.

B. Comparision and discussion

Three approaches are compared by several aspects: the pre-processing, the training data and the execution time, the extensibility, and the accuracy.

- *Pre-processing*: CNN_SEG and CNN_IMG are quite simple as they only perform data extraction. However, in CNN_ASM, the disassemble process depends heavily on CPU architectures. Fortunately, IoT malware rarely uses obfuscation techniques compared to PC malware.
- *Training data generation and execution time*: The byte sequences and the color images are the fixed-sized data structures, and we can set the size for inputs, which is under the tradeoff between the accuracy and the execution time for the training. For instance, CNN_SEG has the balanced tradeoff at 2M bytes length.

Length of Bytes	Accuracy	Training Time
5M bytes	91.6%	$>$ 2 hours
2M bytes	90.58%	$\sim$ 1 hour
1M bytes	83.86%	$<$ 1 hour

In contrast, the assembly code sequence is a variable-sized data structure. Instead of setting the input size, we set the number of the convolution layers, which is under the tradeoff between the accuracy and the execution time for the training (equivalently the number of parameters to train). Since current data set for our preliminary experiments is 1,000 (thus 800 samples for each layer), we use fairly shallow models with two layers.

- *Extensibility*: All models can be easily adapted to other malware datasets. We also try the model of LSTM for byte sequences (in CNN_SEG), and the result is lower than CNN. We observe that LSTM seems working well for files $\leq$ 0.5MB, whereas the average size of IoT malware samples is 1.0MB.
- *Accuracy*: We estimate the accuracy by the average hit rate. The accuracy of each method is shown in Table II. CNN_IMG and CNN_ASM achieve higher accuracy than CNN_SEQ, Fig. 10 shows the convergence of each method, which is generally good.

  We observe two more points among the results.

  - As Fig. 11 shows, the color images of malware and non-malware are visually different, and non-malware mostly looks darker. This means that the entropy of malware is higher than that of non-malware.
  - We take goodware samples from Ubuntu, which are generally much smaller (the average is $0.07MB$) than malware samples (the average is $1.0MB$). The accuracy of SEQ_ASM may be biased by the size difference.

TABLE II
COMPARISON OF THE ACCURACY.

Fold	CNN_SEQ	CNN_IMG	CNN_ASM	
			NoOp	Ops
1	86.3	100	100	100
2	82.8	100	100	100
3	97.5	100	100	98.25
4	97.5	100	100	100
5	88.8	100	100	100
Avg.	90.58	100	100	99.65

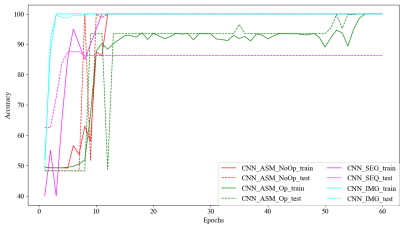

Fig. 10. Average accuracy transition during training process.

V. CONCLUSION

This paper compared three CNN-based approaches for IoT malware detection on 1,000 IoT malware samples for x86. The approaches vary with the input data structures, i.e., byte sequences, color images, and assembly instruction sequences. Among them the first two data structures are fixed-sized, and the last is variable-sized. Experimental results showed that either approach works quite well, probably partially because IoT malware does not use obfuscation techniques. We also observe and compare them from several criteria, e.g., the complexity of the pre-processing step, the training data and the execution time, the extensibility and the accuracy. Our experiments are preliminary and we would like to try on larger sets of malware (as well as other platforms different than x86) to confirm our current observation.

Fig. 11. Dataset (a) malware and (b) goodware

REFERENCES

[1] "Mcafee labs threat report," Tech. Rep., March 2018.

[2] J. O. Kephart *et al.*, "A biologically inspired immune system for computers," in *Artificial Life IV: proceedings of the fourth international workshop on the synthesis and simulation of living systems*, 1994, pp. 130–139.

[3] M. Sharif, A. Lanzi *et al.*, "Automatic reverse engineering of malware emulators," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 94–109.

[4] P. O'Kane, S. Sezer *et al.*, "Obfuscation: The hidden malware," *IEEE Security & Privacy*, vol. 9, no. 5, pp. 41–47, 2011.

[5] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 4, 2013.

[6] C. Willems, T. Holz *et al.*, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.

[7] M. Ghiasi, A. Sami *et al.*, "Dynamic malware detection using registers values set analysis," in *Information Security and Cryptology (ISCISC), 2012 9th International ISC Conference on*. IEEE, 2012, pp. 54–59.

[8] C. Ravi and R. Manoharan, "Malware detection using windows api sequence and machine learning," *International Journal of Computer Applications*, vol. 43, no. 17, pp. 12–16, 2012.

[9] X. Jiang, X. Wang *et al.*, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[10] N. M. Hai, M. Ogawa *et al.*, "Obfuscation code localization based on cfg generation of malware," in *International Symposium on Foundations and Practice of Security*. Springer, 2015, pp. 229–247.

[11] Y. LeCun, L. Bottou *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[12] Y. LeCun, Y. Bengio *et al.*, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[13] Y. Hechtlinger, P. Chakravarti *et al.*, "Convolutional neural networks generalization utilizing the data graph structure," 2016.

[14] G. Hinton, L. Deng *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[15] Q. V. Le, W. Y. Zou *et al.*, "Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE, 2011, pp. 3361–3368.

[16] D. Silver, A. Huang *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[17] Y. N. Dauphin, A. Fan *et al.*, "Language modeling with gated convolutional networks," *arXiv preprint arXiv:1612.08083*, 2016.

[18] E. Raff, J. Barker *et al.*, "Malware detection by eating a whole exe," *arXiv preprint arXiv:1710.09435*, 2017.

[19] M. Cogswell, F. Ahmed *et al.*, "Reducing overfitting in deep networks by decorrelating representations," *arXiv preprint arXiv:1511.06068*, 2015.

[20] M. Abadi, A. Agarwal *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[21] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[22] J. Su, D. V. Vargas *et al.*, "Lightweight classification of iot malware based on image recognition," *arXiv preprint arXiv:1802.03714*, 2018.

[23] A. Krizhevsky, I. Sutskever *et al.*, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[24] A. V. Phan and M. Le Nguyen, "Convolutional neural networks on assembly code for predicting software defects," in *Intelligent and Evolutionary Systems (IES), 2017 21st Asia Pacific Symposium on*. IEEE, 2017, pp. 37–42.

[25] A. Conneau, H. Schwenk *et al.*, "Very deep convolutional networks for text classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, vol. 1, 2017, pp. 1107–1116.