# A Dynamic Malware Detection Mechanism Based on Deep Learning

**Wei YIN, Hongjian ZHOU, Mingyang WANG, Zhiwen JIN, Jun XU**

North China Institute of Computing Technology, China

**Summary**
Static malware analysis cannot identify malware that uses encryption or shell technology. Traditional dynamic malware analysis has fingerprints, such as using hooks to monitor function calls, which can be recognised and tampered by malware. To address this issue, this paper proposes a dynamic malware detection mechanism based on the cloud environment. Malware is running at the guest level while malware monitoring is conducted at the hypervisor level, therefore malware execution and monitoring environments are isolated. The breakpoint injection technology is utilised to capture the kernel function calls so that malware behaviours, such as processes, file access, registries and system services, can be monitored and the log is generated. The log is processed to extract four dimensions of information which is utilised as the input for the deep learning network. The deep learning network, trained by a large number of samples, can recognise and output the malware types at an accuracy as high as 97.3%.

***Key words:***
*dynamic malware detection, deep learning, guest monitoring*

## 1. Introduction

Static and dynamic analysis are two most common methods for malware detection. Static analysis allows analyzing the code without actually executing it, which involves code auditing, reverse engineering, unpacking, HEX checking, etc. Static malware analysis is difficult. To begin with, it is not convenient to obtain the malware source code. Furthermore, binary programs lost information about data structure and variable size. In addition, polymorphic malware, e.g. packer [1], that uses packing and encryption technology to alter the way it looks makes static analysis even harder.

Dynamic analysis methods analyze the code by executing it without requiring the source code. In the meantime, the analyzer monitors its behaviours, such as API calls, file accessed, network connection launched, etc. The benefit is that even polymorphic malware that changes the binary fingerprint cannot escape from detection, because it cannot hide the program behaviours when executing.

In dynamic malware analysis, a single function call reflects an operation. Multiple consecutive and relevant function calls represent the achieved functionality. All function calls can reveal the program behaviour. The Windows operating system provides multiple levels of function calls including Windows APIs, Windows Native APIs, System Calls and Kernel Function Calls, from top to bottom. Windows APIs consist of a set of functions that can achieve a particular functionality including security, management, etc.

Windows APIs are open and stable as Windows evolves so that analyzers can define user-level function hooks to monitor Windows API calls once for all Windows versions. However, malware can call Windows Native APIs, System Calls or Kernel Function Calls instead to avoid user-level Windows API call monitoring. Kernel Function Calls are the lowest level functions to complete an operation. If we capture Kernel Function Calls, we can solve the escape problem achieved by malware calling Windows Native APIs, System Calls or Kernel Function Calls.

However, defining function hooks, such as that in CWSandbox [2], to hook interesting function calls needs to modify the operating system. This could be a fingerprint that can be utilised by malware to identify the monitoring environment. Smart malware will not conduct any suspicious actions if the monitoring environment is detected. Even smarter malware can modify function hooks at the same time cheating analyzers that the monitoring environment is working fine so that malicious behaviours cannot be detected.

Therefore, the monitoring environment and malware execution environment should be isolated to avoid tampering, To build a malware detection mechanism based on the cloud environment could be a promising solution. The malware execution environment can be a guest system running above the hypervisor, e.g. Xen [3]. The malware monitoring environment can be realized at the hypervisor level to monitor and analyze the guest system behaviour. The hypervisor is invisible to the guest system and they are isolated so that malware executing at the guest level cannot tamper the hypervisor. In this paper, we design, implement and evaluate such a malware detection mechanism. We capture the Kernel Function Calls of the guest system at the hypervisor level and generate a figure, which is utilised as the input for the deep learning neural network. Finally, the figure is processed and the malware is classified.

The paper achieves the following contributions.

Manuscript received July 5, 2018
Manuscript revised July 20, 2018

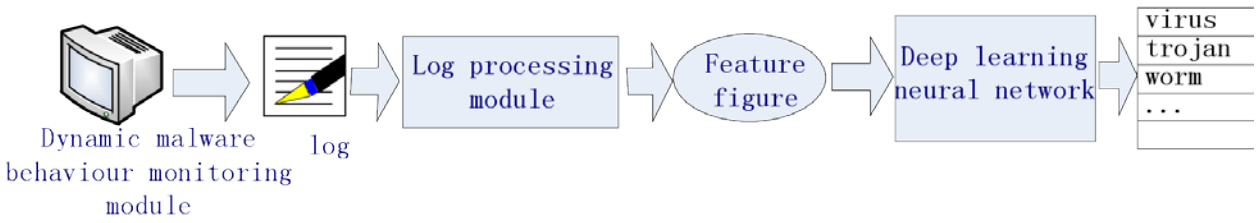


Fig.1 Mechanism modules

First, We design a transparent, secure dynamic malware analysis mechanism, which can secretively monitor function call execution of the guest system. Important information is extracted including the Kernel Function Call trace, function parameters, return values and function correlations, which forms the elements for the feature picture. It is proved that the information can well describe malware behaviours including processes, file access, network connections, registries, system services, etc.

Second, we execute 9000 pieces of malware samples on the mechanism and gather corresponding feature figures, which forms the training set. A neural network is constructed and trained with these samples. Other 3000 pieces of malware are utilised as the testing set. Experiments show that our mechanism can achieve a classification accuracy as high as 97.3\%.

The rest of the paper is organised as follows. Section 2 discusses the proposed mechanism. The experimental setup and results with the mechanism are discussed in section 3. The related work is presented in section 4. The paper concludes in section 5.

## 2. Proposed Mechanism

In this section, we discuss the proposed mechanism for dynamic malware analysis based on a deep learning neural network.

The mechanism, as shown in Fig. 1, consists of a dynamic malware behaviour monitoring module (DMBM), a log processing module (LPM) and a deep learning neural network (DLNN). The DMBM module follows Kernel Function Calls to capture malware behaviours. The output of the DMBM module is a log file which is processed by the LPM module to obtain four dimensions of information including Kernel Function Call sequence, function parameters, function return value and function correlations. Using this information, a figure is generated, which serves as the input of the neural network. Below we illustrate these three modules.

### 2.1 The DMBM Module

The DMBM module is shown in Fig. 2. It is designed and implemented on the Xen cloud platform [3]. It runs on dom0 that is a privileged domain. Dom0 boots first and manages the DomU unprivileged domains. Guest operating systems, e.g. Guest1, are running on DomUs. The DMBM module directly accesses the guest system memory through the open source LibVMI library [4]. Three pieces of information are monitored including Kernel Function Calls, function parameters and return values.

In order to get access to what is happening on the guest system, DMBM inserts a #BP(breakpoint) instruction (INT3, instruction code 0xCC) at the code where DMBM is interested. When the #BP instruction is executed, a VMEXIT signal is passed to the dom0 at the hypervisor level. Therefore, the hypervisor can capture any instruction executed at the guest system. So far the #BP instruction has been used in the debugging technique so that the code debugged cannot feel the existence of the debugger. In our mechanism, we use the breakpoint injection technique to trace the execution of the guest operating system to achieve transparency, efficiency and stealth. In this paper, we focus on Windows 7 SP1 guest system to discuss the monitoring principle.

First, we need to determine the kernel location in the memory. We find that Windows 7 stores the kernel virtual address (KVA) in the FS and GS registers. KVA points to the _KPCR data structure which is usually loaded into a relative virtual address (RVA) in the kernel. _KPCR can be identified by the KiInitialPCR symbol. We can search the symbol in the memory and determine the location of _KPCR, named addr1. Then substracting addr1 by RVA, we can obtain the kernel base address.

Second, we need to determine the memory location of various kernel functions. This is done by extracting relevant information from the kernel debugging information. Kernel debugging information is usually used for forensic analysis. In our mechanism, we use the Rekall [5] forensic analysis tool to process Windows debugging information and make a kernel function map.

Combining the kernel function map and the kernel base address, we can use the breakpoint injection technique to capture the function execution of the guest system after the malware is executed. When a function is called, the corresponding parameters and return values will be obtained from the registers and stacks. This information is

recorded in a log file which will be processed by the LPM module.

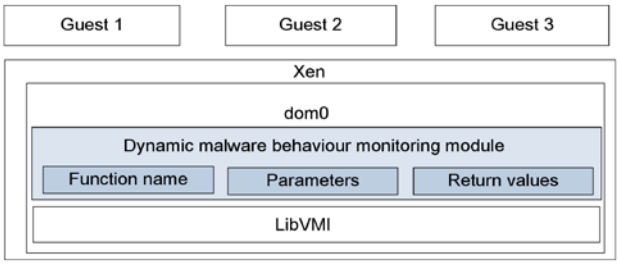


Fig. 2 Dynamic malware behavior monitoring module.

## 2.2 The LPM Module

The LPM module processes the log file that is generated by the DMBM module. Four pieces of information are extracted. The first one is the Function Call Sequence which reflects the malware's behaviours. An m *n matrix is generated, named A1, which represents what functions are executed for consecutive m kernel function calls. n is the total number of kernel functions in the operating system. n is obtained by processing the operating system's debugging information using the Rekall forensic analysis tool. Kernel functions can be represented by f1(),f2(),...fn(). m is the monitoring window, which is the number of consecutive kernel functions monitored after the malware is executed. If element aij in A1 is 1, this means that the ith function call is fj(). In the A1 matrix, there is only one element that equals to 1 and other elements are 0s for each row because, at each monitoring point, there is only one kernel function called. The LPM module extracts kernel functions from the log file in order and generates the A1 matrix.

The second and third piece of information is function parameters and return values. Malware may modify registry keys to make it starts as the system boots, or open a certain port to steal private data. Function calls can only coarsely reflects operations. The detail of an operation is not shown, for example, which registry key is modified to what value, or which network port is open and how many bytes are transmitted and what is in it. Therefore, it is necessary to record and analyze the function parameters. The return value can tell whether the function call is successful, the allocated memory address, etc. It is as important as the function parameters. We append function parameters and return values to the right of the A1 matrix to make an A2 matrix. When the VMEXIT signal is captured by the hypervisor, we obtain the parameters and return values using the VMI (virtual machine introspection) technology through the LibVMI library.

The fourth piece of information is function correlations. We generate another m*n matrix, called A3. This matrix represents which functions are correlated. The LPM module follows function parameters and return values. Two functions that are working on the same objects are correlated. For example, if the ith function fj() return a file handler and the kth function fp() writes the file handler, the LPM module correlates these two functions. The element aij and akp in A3 will be set to the same value z. z starts from 1 and increases as the number of groups of correlated functions increases. In this way, all captured kernel functions are classified into various groups. Correlated functions belong to the same group. Groups are differentiated by z.

Then we put the A3 matrix below the A2 matrix and add 0s at the right of A3 to make an A4 matrix. We take each element in the A4 matrix as a pixel value and generate the figure that is used as the input of the neural network.

## 2.3 The DLNN Module

The deep neural network is a machine learning model. In this model, neurons at different layers are connected by weights and activation functions. The key point is to learn the weights between neurons and figure out the hidden relationship between the input and output. The input is passed to the first layer of the neural network to generate a number of values. These values will be processed by the activation function and then the results are regarded as the input of the second layer. Subsequent layers repeat this action and finally, the neural network outputs the ultimate results.

The convolutional neural network is a type of deep learning networks. It is widely used in the domain of face or voice recognition. Typical applications of convolutional neural networks include GoogleNet [6], Microsoft ResNet [7] and AlphaGo [8]. In this paper, we choose the convolutional neural network to detect and classify malware. Our convolutional neural network consists of 13 convolutional layers, 13 max-pooling layers and 5 fully connected networks. The relu and sigmoid function are utilised as the activation function for the convolutional l

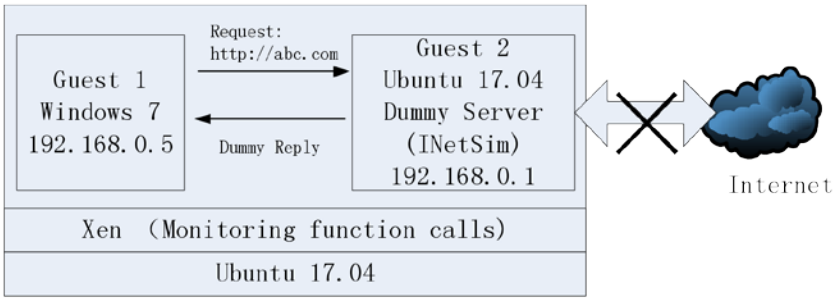


Fig.3 Experimental setupayer and fully connected

network layer respectively. However, we use the SoftMax function as the activation function at the last fully connected network layer to realise malware classification.

The A4 matrix is regarded as the input feature figure that is conducted a convolution operation with multiple filters. The filters are matrixes with a number of weights. The results are multiple figures that are processed by the max pooling layer. After this, multiple small figures are generated. These small figures become the input of the next convolutional layer.

The weights are calculated according to the following four steps. (1) Forward calculate the output of each neuron. (2) Backward calculate error terms. (3) Calculate weight gradient. (4) Update the weights using the gradient descent algorithm.

## 3. Experiments

In last section, we present the proposed mechanism and describe various modules. In this section, we discuss the experimental setup and results.

### 3.1 Experimental Setup

The experimental environment is shown in Fig. 3. The Xen hypervisor is running on a host running the Ubuntu 17.04 operating system. On Xen dom0, we implement the DMBM Module. Guest 1 installs Windows 7 SP1. On guest 1 malware is executed. Its behaviours are monitored and recorded at the Xen hypervisor level. Ubuntu 17.04 is installed on guest 2. INetSim is running on guest 2 to simulate common Internet services including HTTP, SMTP, DNS, FTP, etc. The experimental environment is isolated from the Internet, to prevent the Internet from malware executed. All traffic from guest 1 will be forwarded to guest 2 which responses to requests. After running a piece of malware, guest 1 will be installed a new clean Windows 7 virtual machine to minimize the interference from the last malware.

We gather 10000 malware samples from the Das Malwerk server [9]. These samples include virus, trojans, logic bombs, worms and spyware. Each has 2000 samples. We upload them to VirusTotal [10] and each sample is checked by multiple AntiVirus software. Finally, we assign a tag to each sample according to the results from VirusTotal. In addition, we obtain 2000 non-malware from the Internet and most of them are office, audio, and video software.

We classify each type of samples into 4 groups. We choose 3 groups as a subset of the training set and the fourth group as a subset of the testing set. We select four combinations of a training set and a testing set to conduct evaluations. The deep learning network is implemented by the Tensorflow [11] software. We use the training set to train the neural network and the testing set to test the classification accuracy.

### 3.2 Experiment Results

We present the results using a thermal map, as shown in Fig. 4. The x-axis represents the de-facto type of the malware. The y-axis represents the output of the neural network. For example, The element in the first row, the first column in Fig. 4(a) describes that the number of de-facto viruses which is recognised by the neural network as viruses is 481. From Fig. 4, we observe that our malware detection mechanism can classify most of the malware. After calculation, the classification accuracy is around 97.3%.

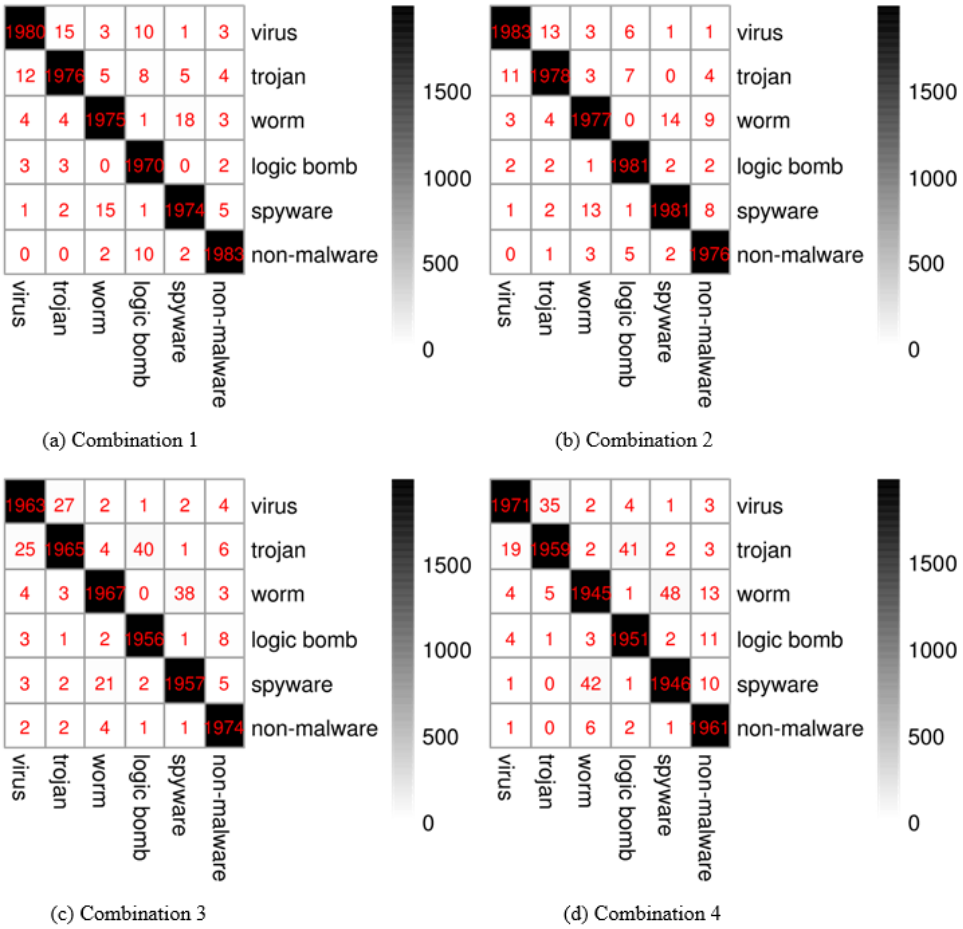


Fig. 4 Thermal map of malware detection and classification

However, the mechanism has some misjudgments. We select 20 misjudged samples and analyze the kernel function call sequence manually. We find that misjudgments may due to the following reasons. Firstly, for non-malware that is mistaken as malware, it has at least 3 of the following behaviours: (1) frequently create, delete or move files; (2) create or delete registry keys; (3) frequent network access; (4) open sensitive system services; and (5) create and hide processes.

Secondly, some logic bombs are misjudged as non-malware. This is because logic bombs behave normally before the trigger event, e.g. web access, occurs. Thirdly, some trojan horses are mistaken as viruses. This is because these trojan horses read and write files on the disk. Most viruses modify files to hide its existence. These operations are similar. Fourthly, some worms are misjudged as spyware, This is because worms propagate through networks. They sniff the network just as spyware does.

In addition, we study the training accuracy. Fig. 5 shows the results for group 1. At 100 iterations, the accuracy can reach 90%. After that, it increases slowly and finally reaches 97.3% at 2000 iterations.

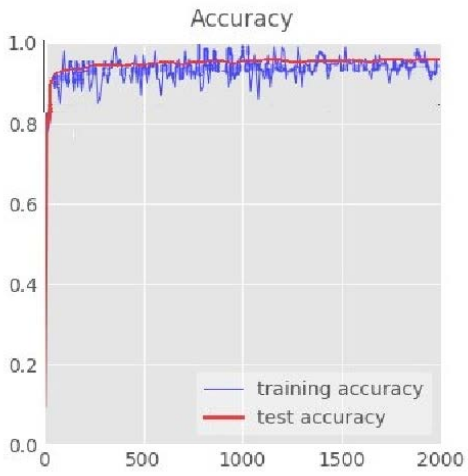


Fig. 5 Accuracy of malware detection

## 4. Related Work

Malware detection methods can be categorized into static and dynamic methods. Our work belongs to the latter. There are a number of dynamic solutions in the literature. For example, TTAnalye [12] monitors Windows APIs and Windows Native APIs and therefore can only detect user-level malware. Apart from user-level malware, our mechanism can detect rootkits because our mechanism monitors Kernel Function Calls. TTAnalye uses the PC emulator which simulates computer hardware including processors, keyboards, etc. It compares the program counter of the virtual processor with the head address of the monitored function so that it can record the function executed. Our mechanism executes the malware on the guest system and uses the breakpoint injection technique to monitor guest system (virtual machine) behaviours at the hypervisor level. In a computer simulator, all instructions are executed in software, but on virtual machines, a part of instructions are executed on real CPUs. So our mechanism is more efficient. In addition, TTAnalye requires logs to be processed manually by professionals. In our mechanism, this task is processed by the LPM module automatically.

CWSandbox [2] and Cuckoo [13] are dynamic analysis methods. They install kernel drivers, define function hooks to monitor Windows APIs and system calls. When the interested function is loaded into the memory, the function is rewritten and the code implementing the monitoring functionality is appended before or after the function. When it is executed, the function can be recorded. Malware can detect the monitoring method by integrity check of the function. In our method, the malware execution and monitoring environments are isolated. In addition, Different from our automatical log processing module, CWSandbox and Cuckoo require security professionals to process the logs manually.

DeepSign [14] is another dynamic analysis methods, which uses Sandbox to record API calls. The return value of some APIs in the Sandbox environment is different from that in the real environment [15]. This is utilised by malware to detect Sandbox. DeepSign uses unsupervised methods to train the neural network. Our mechanism assigns tags to the training samples, which uses the supervised method. The benefit is that it can better classify the malware with a relatively high accuracy.

In [16], Tobiyama et al. employ a convolutional neural network to analyze API calls. This is also a dynamic analysis method. The captured API calls are at the user level and it cannot detect kernel rootkits. In addition, the last layer of the neural network uses the sigmoid function. Compared to the SoftMax function in our mechanism, it can only detect whether a software is a malware or not. It cannot tell malware types. Saxe et al. [17] processes the binary file of malware. It is a static analysis method. It uses a deep-feed-forward neural network to detect malware, of which the last layer utilizes a sigmoid function.

Recently, the neural network is widely used to detect malware on the Android platform. However, most of them are based on static analysis. For example, R2-D2 [18] unzips an android APP to obtain the classes.dex file of which each byte is mapped to an RGB value. Then a figure is obtained. The figure is used as the input for the convolutional neural network. McLuaghlin et al. [19] disassemble the binary program and analyze the raw opcode sequence. DroidMiner [19] is another static analysis method. It extracts threat action mode sequence from the binary file. A machine learning method is utilised for malware detection. Yuan et al. [20] extract features based on a static and dynamic analysis. A deep belief network based on an unsupervised learning algorithm is used to train the neural network. Compared to the dynamic method used in our paper, these static analysis methods do not execute malware or study its behaviours, therefore, they cannot detect polymorphic malware that uses packing and encryption technology.

## 5. Conclusion

Static malware analysis cannot detect malware that uses packing and encryption technology while traditional dynamic malware analysis has fingerprints. To overcome this issue, we proposed a dynamic malware detection mechanism based on cloud computing to isolate the monitoring environment and the malware execution environment. The breakpoint injection technology is utilised to capture the malware (running at the guest level) behaviours at the hypervisor level. Malware detection is based on a deep learning neural network and can classify various malware types at an accuracy as high as 97.3%.

**Acknowledgments**

The work is supported by the NSFC project 61702542 and the China Postdoctoral Science Foundation project 2016M603017.

## References

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools.ACM computing surveys, 44(2):1-42, 2012.

[2] C. Willems, T. Hotz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. In Proceedings of IEEE Symposium on security and privacy, pages 1-9,2007.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Proceedings of SOSP, pages 164-177, 2003.

[4] LibVMI. https://github.com/libvmi/libvmi, 2018.
[5] Rekall Forensics. http://www.rekall-forensic.com/, 2018.
[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2015.
[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.
[8] D. Silve, A. Huang, C. J. Maddison, A. Guez, L. Sifre, and G. v. d. Driessche. Mastering the game go with deep neural netwoks and tree searchs. Nature, 529:484-489, 2016.
[9] Das Malwerk. http://dasmalwerk.eu/, 2018.
[10] VirusTotal. https://www.virustotal.com/, 2018.
[11] Mart__n Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensor ow.org.
[12] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. J. Comput Virol, 2:67-77, 2006.
[13] J. Bremer. Blackhat 2013 workshop: Cuckoo sandbox open source automated malware analysis. http://cuckoosandbox.org/2013-07-27-blackhat-las-vegas 2013.html, 2013.
[14] O. E. David and N. S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classication. In Proceedings of International Joint Conference on Neural Networks (IJCNN), 2015.
[15] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In Proceedings of the 9th ACM symposium on information, computer and communication security (ASIA CCS14), pages 1-9, 2014.
[16] S. Tobiyama, Y. Yamagachi, H. Shimada, T. Ikuse, and T. Yagi. Malware detection with deep neural network using process behavior. In Proceedings of the 40th IEEE annual computer software and applications conference, 2016.
[17] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In Proceedings of 10th International Conference on Malicious and Unwanted Software (MALWARE), 2015.
[18] T. H.-D. Huang and H.-Y. Kao. R2-d2: Color-inspired covolutional neural network (cnn)-based android malware detections. Cryptography and Security, 2017.
[19] N. McLuaghlin, J. M. d. Rlincon, B. Kang, and S. Yerima. Deep android malware detection. In Proceedings of the 7th ACM on conference on data and application security and privacy, 2017.
[20] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In Proceedings of the 2014 ACM conference on SIGCOMM, pages 371-372, 2014.

**Wei YIN** received his Ph.D degrees from the University of Queensland, Australia in 2012. He is a research engineer in North China Institute of Computing Technology. His research interest include rate adaptation in wireless networks, honeypots and honey encryption.