Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 3, March 2019, pp. 1053-1060
DOI: 10.23940/ijpe.19.03.p35.10531060

Return Instruction Identification in Binary Code with Machine Learning

Jing Qiu and Guanglu Sun*

School of Computer Science and Technology, Harbin University of Science and Technology, Harbin, 150080, China

Abstract

Binary code analysis is the main method for malware analysis. In this paper, the analysis is started by identifying return instructions to disassemble binary code. The return instruction identification problem is converted into a binary classification problem: is a byte in binary code the first byte of a return instruction? The 32 bytes around a byte in binary code are considered the feature of the byte. A multilayer perceptron is employed to build the classification model. Then, the model is trained with 1,383 binaries from Windows XP SP3. The evaluation results on several open sources show that our approach is feasible and has high accuracy.

Keywords: return instruction; binary code analysis; machine learning; reverse engineering

(Submitted on November 12, 2018; Revised on December 15, 2018; Accepted on January 16, 2019)

1. Introduction

Most software is distributed in binary code form. This is good for intellectual property right protection because binary code is not easy to analyze. It also simplifies the process for users as it does not require compiling, which is not easy for the average user. Although analyzing binary code is hard compared with source code analysis, in some situations, it is important. For example, when analyzing malicious software whose source code is not available, analyzing its binary code is the only feasible approach.

The first step of binary code analysis is disassembly. However, code disassembly is challenging in some cases. For example, in embedded devices, the binary code of their firmware could be packed in an unknown format. Therefore, determining whether a byte is code or data is difficult in binary code when its format is unknown.

Code extraction is also challenging even if the file format is known. First, instructions have different lengths in the Intel instruction set. As a result, different disassembly results will be obtained if we disassemble at different start positions.

Second, data can be embedded in x86 code. For example, switch statements in C language code are often translated into jump or indirect tables. For a jump table, function pointers are stored. These functions in the jump table are invoked dynamically and their code cannot be determined to be code or data statically.

Third, the bytes of a return instruction could be part of another instruction. For example, the byte of "ret" is 0xC3 and 0xC3 is part of "0xF6 0xC3 0x 0x20", which is "test bl, 0x20". Therefore, it is unable to identify return instructions if it only searches the bytes of return instructions.

Finally, handcrafted assembly code can be another challenge. Compilers are programs that use fixed function epilogues and prologues to generate binary code. However, handcrafted assembly code could have totally different function epilogues and prologues.

There are a few works on binary code extraction using machine learning techniques [1-3]. However, they focus on the head bytes of function prologues, which could be easily changed (e.g., exchange the order of some instructions). In this

* Corresponding author.
E-mail address: sunguanglu@hrbust.edu.cn

paper, a new approach for function identification by recognizing return instructions is proposed. The principle behind it is straightforward. Every control flow of a function should end with a return instruction, no matter how the code is transformed.

The function return instructions are used as recognition feature. Specifically, the forward and backward 16 bytes of return instruction are used as a feature, training a machine learning model to identify return instructions. Return instructions are classified into two categories: non-return instructions (i.e. their bytes are part of another kind of instructions) and return instructions in a function. For any location that holds bytes of a return instruction, the trained machine learning model will determine which class the location is according to the forward and backward 16 bytes of the location. Based on the result, the range of each function in a binary code will be partitioned. Finally, for each identified function return instruction, a reverse extended control graph will be used to identify the function entry point [4] and finally finish the function identification work.

In summary, this paper makes the following main contributions:

1) We firstly proposed a machine learning based approach for identifying return instructions. Our approach is independent of compilers and platforms.

2) We evaluated our approach with open sources and binaries from Microsoft Windows. The results show that the proposed approach is effective and has high precision.

In the remainder of this paper, we will give a motivating example in Section 2, provide the details of our design in Section 3, and present the evaluation and discussion in Section 4. Finally, the related work is given in Section 5, and Section 6 concludes.

2. Motivating Example

Current approaches use the function head and tail bytes as recognition features. It is feasible to deal with binary code that is compiled by common compilers in most instances by using such features. However, for some code, such as code without head or tail features (Figure 1), these approaches will not work.

```
mov     edi, edi
push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
push    esi
push    edi
mov     edi, [ebp+arg_14]
...
```

(a) With common epilogue

```
mov     eax, [esp+arg_4]
mov     ecx, [esp+arg_C]
or      ecx, eax
mov     ecx, [esp+arg_8]
mov     eax, [esp+arg_0]
mul     ecx
retn    10h
```

(b) Without common epilogue

Figure 1. Functions in binary code where bold instructions are the common epilogue

In the example, the code is similar to handcrafted code. Such code will be identified by current approaches. Additionally, current approaches all use the cross-reference information to identify functions. In other words, during identification, if the target of a call instruction is legal, then the target is also a legal function entry point. However, this heuristic rule will not work for functions that do not have cross-references (for example, callback functions). Moreover, a function may have no head/tail features and no cross-reference at the same time. At this point, the current approaches are unable to correctly identify the function. Our method can recognize this kind of function by using the function return instruction as the recognition feature. Our method will be described in detail below.

3. Design

3.1. Return Instruction

As the name implies, this type of instruction is designed for returning the control flow from the called function to the next instruction at the call point. A function, or procedure, is designed to make the code reusable. A function will be called multiple times in a program execution. Each call requires a return instruction invoking to return the control flow. In other

words, the function return instruction is an indispensable part of a function and must be the last instruction of a function control flow. In the Intel x86 instruction set, there are two forms of return instructions. One has parameters and the other has no parameters. The parameter of a return instruction represents the values to be accumulated for the stack pointer register. Since each function must have one or more return instructions, functions can be identified by characterizing the return instructions.

3.2. Machine Learning Model

Return instruction identification can be regarded as a binary classification problem. That is, for each byte of a binary code, we will predict if it is the first byte of a return instruction. Suppose bytes of a n-bytes binary code are $\{x_1, \cdots, x_n\}$. Let y_i be the label of x_i. $y_i = 1$ if x_i is the first byte of a return instruction; otherwise, $y_i = 0$. Suppose x_m is the first byte of a return instruction. Let $X = \{x_m, x_{m-1}, \cdots, x_{m-15}, x_{m+1}, \cdots, x_{m+15}\}$. $Y = \{1, 0\}$ if $y_m = 0$, and $Y = \{0, 1\}$ if $y_m = 1$.

As the opcode (i.e., the first byte) of a return instruction is known and in a limited set, the identification can be started at the locations that hold the opcode of a return instruction. Other bytes are labeled as false opcode of a return instruction.

We choose a neural network to build our model. Our machine learning model is shown in Figure 2. It consists of an input layer, three hidden layers, and a sigmoid output layer. The input layer has 32 neurons that correspond to the 32 bytes around the location that holds the opcode of a return instruction. Each hidden layer has 64 neurons. The output layer has two neurons that output the classification result (i.e., whether x_m is true or false return instruction). Except for the output layer, all the activation functions in the network are ReLU.

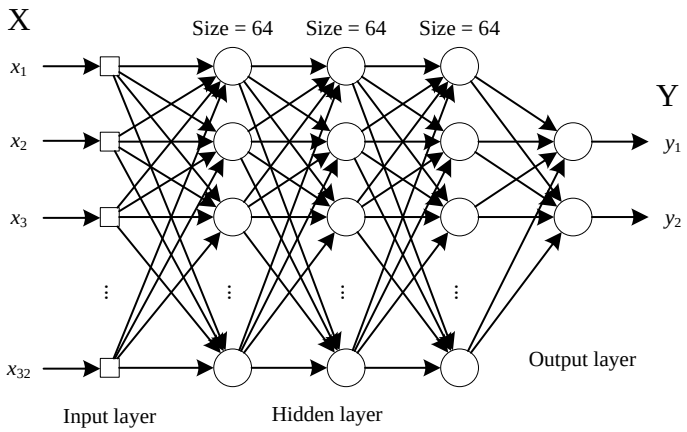


Figure 2. Our neural network skeleton

In the implementation, Keras [5], a high-level framework for TensorFlow [6], is chosen to build the machine learning model. In the training, the cost function is cross-entropy, the dropout is 0.1, the batch size is 1,000,000, and the number of epoch is 5,000.

The code for building the machine learning model is shown in Figure 3.

```
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=32))
model.add(Dropout(0.1))
model.add(Dense(64, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Figure 3. Main code for building machine learning model

4. Evaluation

4.1. Setup

The proposed prototype is evaluated on the following binary sets (Table 1), and the prototype is run on a 3.70 GHz Intel Core i7-8700K machine equipped with an NVIDIA TITAN Xp graphics card.

a) Test set:

(1) Four open source software: Notepad++ 7.6.1, Putty 0.70, WinMerge 2.16.0, and PHP 7.3.0. They are compiled by using Microsoft Visual C++ to generate program debug databases (PDBs) that hold debugging information for the ground truth.

(2) Four applications from the Windows 7 operating system: Windows Mail, Windows Media Player, Windows Photo Viewer, and Windows Defender. Their PDBs are provided by the Microsoft Symbol Server.

b) Training set: Binaries from the Windows XP SP3 under the directory \windows\system32. Their PDBs are provided by Microsoft. These binaries are compiled by VC7 (detected by the packer detection tool Exeinfo PE [7]).

Table 1. Programs for evaluation and training the model

Program	Version	Compiler	Files	False ret	True ret	Size/KB
Notepad++	7.6.1	VC8	1	9,472	7,647	1,672
Putty	0.70	VC6	6	13,216	11,162	3,009
WinMerge	2.16.0	VC9	5	31,649	29,582	11,128
PHP	7.3.0	VC14	60	94,879	99,459	61,885
Win mail	6.1.7601	VC9	7	5,190	7,610	5,074
Win media player	12.0.7601	VC9	17	13,545	13,588	6,000
Win photo viewer	6.1.7601	VC9	5	18,267	10,031	4,253
Win defender	6.1.7601	VC9	12	6,881	12,366	2,862
WinXP	SP3	VC7	1,383	803,605	965,373	295,418

Additionally, we choose two famous disassembly tools to compare with our approach. They are IDA Pro 6.8 and Jakstab [8]. Both of them use function prologues, epilogues, and other heuristics to identify functions that also include function return instructions. We choose code from MASM32 SDK 11 examples [9] as a test set. In order to validate the effectiveness of our approach, we slightly modified the code by 1) removing cross references (e.x. “call f” is rewrited to “mov eax,f-2/add eax,2/call eax”) and 2) adding “OPTION PROLOGUE:NONE” and “OPTION EPILOGUE:NONE”, which ensures there is no default function prologues and epilogues in the compiled binary code.

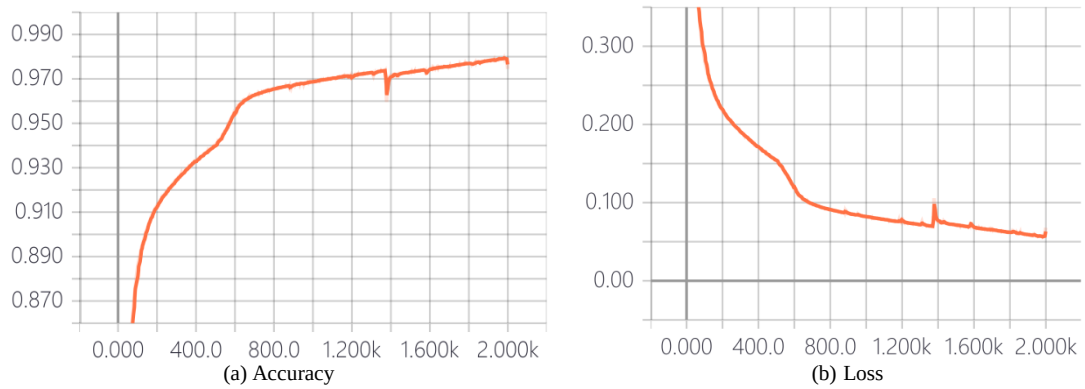
For Jakstab, we use the “-h” option in the command line to force it to use heuristics to discover more functions. A function is regarded as correctly identified by IDA or Jakstab if the function entry point is identified.

4.2. Result

The evaluation result of the test set is shown in Table 2. The variation of accuracy and loss during training is visualized in Figure 4. The training takes about one hour. The evaluation result of the MASM32 code example is given in Table 3. All identification for a program is done in less than one second.

Table 2. Identification results of the test set

Program	Precision	Recall	F1 score
Notepad++	0.98	0.99	0.99
Putty	0.95	0.98	0.97
WinMerge	0.97	0.98	0.97
PHP	0.97	0.98	0.97
Win mail	0.99	1.00	1.00
Win media player	1.00	0.99	1.00
Win photo viewer	0.97	0.94	0.96
Win defender	1.00	0.97	0.98
Average	0.98	0.98	0.98



(a) Accuracy (b) Loss

Figure 4. Variation of accuracy and loss during training; the horizontal axis is the epoch

Table 3. Identification result of MASM32 SDK 11 code examples

Program set	IDA Pro			Jakstab			Ours		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Example01	0.93	0.18	0.30	1.00	0.35	0.52	1.00	0.98	0.99
Example02	1.00	0.38	0.55	1.00	0.50	0.67	0.99	0.84	0.91
Example03	0.87	0.33	0.48	1.00	0.50	0.66	1.00	0.90	0.95
Example04	0.95	0.39	0.56	1.00	0.43	0.60	0.99	0.85	0.92
Example05	0.99	0.41	0.58	1.00	0.48	0.64	1.00	0.91	0.95
Example06	1.00	0.56	0.72	1.00	0.66	0.80	1.00	0.99	1.00
Example07	0.96	0.41	0.72	1.00	0.76	0.86	0.98	1.00	0.99
Average	0.96	0.38	0.57	1.00	0.53	0.68	0.99	0.92	0.96

4.3. Discussion

4.3.1. Recall and Precision

The evaluation result of the test set (Table 2) shows that our model has high precision and high recall. This result is as expected. Because the first byte of a return instruction is known, the identification is started from scanning all possible locations that hold the first byte of a return instruction. There is a sufficient number of samples (1,768,978) for training, and the numbers of true and false return instructions in the training set are very close. As a result, the precision is high. Meanwhile, all possible return instructions are classified; therefore, the recall is high.

Binaries in the training set and test set are compiled by different compilers, but the precision and recall are still high. This indicates that the accuracy of our approach is independent of the version of compilers.

In the evaluation of the examples of MASM32 (Table 3), Jakstab has the highest precision of 100%. However, its recall is low, which indicates that Jakstab uses a conservative strategy and cannot identify functions without prologue/epilogue and cross references. The recall of IDA Pro is the lowest. Although its precision is high, it is somehow not practical in real world analysis because too much code is misidentified as data. As expected, our approach has high precision and high recall. There are very few false positives in our result. This indicates that our approach achieves a good balance between precision and recall.

The precision could be improved by increasing the epochs of training. However, as shown in Figure 4, the improvement of accuracy will be limited and the training will take more time.

4.3.2. False Positives/False Negatives

The reason for the false positives is as follows. If code that follows a possible return instruction can be correctly disassembled without any unknown instructions, the possible return instruction is classified as a true return instruction by our model. However, the possible return instruction could be a false return instruction (Figure 5).

False negatives could be caused by two factors. First, as shown in Table 1, compilers for programs in the training set and test set are different. Rules for optimization code are different in different compilers. As a result, the same high-level

language code could generate different binary code with different compilers. The return statements and code around a return instruction could vary for different compilers. Therefore, there could be new samples in the test set.

```
0043D897 74 0A      jz      short loc_43D8A3
0043D899 66 85 D2   test    dx, dx
0043D89C 74 05      jz      short loc_43D8A3
0043D89E 66 3B D0   cmp     dx, ax
0043D8A1 74 C2      jz      short loc_43D865
0043D8A3 8B 4D F0   mov     ecx, [ebp+var_10]
0043D8A6 8B 7D F4   mov     edi, [ebp+var_C]
0043D8A9 8B 5D F8   mov     ebx, [ebp+var_8]
0043D8AC 0F B7 C0   movzx   eax, ax
0043D8AF 0F B7 D2   movzx   edx, dx
0043D8B2 2B D0      sub     edx, eax
```

(a) True disassembly code where the byte at 0x0043D8A2 is an operand

```
0044174A D9 ED                fldln2
0044174C D9 C9                fxch    st(1)
0044174E D9 E4                ftst
00441750 9B DD BD 60 FF FF FF fstsw   word ptr [ebp-0A0h]
00441757 9B                   wait
00441758 F6 85 61 FF FF FF 41 test    byte ptr [ebp-9Fh], 41h
0044175F 75 D2                jnz     short negYTOXerror
00441761 D9 F1                fyl2x
00441763 C3                   retn
```

(b) Disassembly result if there is a return instruction at 0x0043D8A2

Figure 5. An example of false positive from Putty pageant.exe

Second, the code of programs in the training set and test set are totally different. As a result, there exist new samples in the test set with high probability. For example, float point computations are rare in the training set, which are binaries for an operation system but are common in Putty (Figure 6).

```
0044174A D9 ED                fldln2
0044174C D9 C9                fxch    st(1)
0044174E D9 E4                ftst
00441750 9B DD BD 60 FF FF FF fstsw   word ptr [ebp-0A0h]
00441757 9B                   wait
00441758 F6 85 61 FF FF FF 41 test    byte ptr [ebp-9Fh], 41h
0044175F 75 D2                jnz     short negYTOXerror
00441761 D9 F1                fyl2x
00441763 C3                   retn
```

Figure 6. A false negative which contains float computations in Putty

Similar to other machine learning problems, the possible way to decrease the false negatives could be to collect more training data.

5. Related Work

Binary code disassembly has many applications. For example, in binary instrumentation [10-11] or binary rewriting [12-14], the first step is to extract code from binaries. Linear sweep and recursive traversal are two common strategies to disassemble code [15]. However, these algorithms need a correct start location, which is uncertain if the format of target binary is unknown. Therefore, function identification is important for code disassembly.

If the entry point is uncertain, then machine learning based approaches can be used. Wartell et al. presented a context-sensitive language model based approach to extract code from binaries [16]. It converts binary code into sub sequences of bytes, and then each subsequence is classified into code or data.

Kruegel et al. [17] used function prologues to locate function entry points and then divided the binary into functions. Harris and Miller detected functions in the binary with compiler idioms [18]. IDA Pro [19] uses many heuristics for disassembly including cross-reference information, function prologues, and epilogues [16]. It is still relatively conservative because although some data section is correctly disassembled, they are still marked as data in IDA.

Return instruction identification could be employed to identify functions in binary code. Machine learning technologies have already been applied to function identification in binary code. Rosenblum et al. identified function entry points in binary code [1]. They formulated the problem as structured classification using Conditional Random Fields. Bao et al. proposed a method to recognize functions by using weighted prefix trees, which are trained with *n*-bytes of each function head [2]. The main problem of these two works is that they cannot deal with obfuscated code.

Wang et al. [3] extracted features from basic blocks. They applied symbolic execution on a basic block and obtained the assignment formulas of each register. Then, they extracted the lexical and syntactic features from these formulas. The motivation was to fight against obfuscation. However, this work may not work for binaries with indirect memory addressing.

For functions without classic prologue/epilogue, the above techniques may fail to identify these functions. The only one that identifies functions using the return instructions is reference [4]. Qiu et al. identified functions by scanning all possible return instructions and then building reverse extended control flow graphs (RECFG) from return instructions. They used TOPSIS to select the best RECFG when there was a conflict among RECFGs. The selection considered the byte length, the cyclomatic complexity, and the proportion of full-sized instructions of a RECFG. The main drawback of the approach is that using the three attributes for selection is not reliable in some situations.

6. Conclusions

In this paper, a new approach is proposed for return instruction identification. The return instruction identification problem is converted into a binary classification problem. First, the 32 bytes around a location are considered as the feature of the location. Then, a neural network machine learning model is trained with real world binaries. The evaluation results show that the proposed approach has high accuracy and has nothing to do with the compilers.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (No. 61702140) and Natural Science Foundation of Heilongjiang Province of China (No. LC2018030).

References

1. N. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to Analyze Binary Computer Code," in *Proceedings of the Twenty-Third Conference on Artificial Intelligence* (*AAAI-08*), Chicago, IL, 2008
2. T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to Recognize Functions in Binary Code," in *Proceedings of the* 23rd *USENIX Conference on Security Symposium*, pp. 845-860, 2014
3. S. Wang, P. Wang, and D. Wu, "Semantics-Aware Machine Learning for Function Recognition in Binary Code," in *Proceedings of 2017 IEEE International Conference on Software Maintenance and Evolution* (*ICSME*), pp. 388-398, 2017
4. J. Qiu, X. Su, and P. Ma, "Identifying Functions in Binary Code with Reverse Extended Control Flow Graphs," *Journal of Software*: *Evolution and Process*, Vol. 27, No. 10, pp. 793-820, 2015
5. "Keras: The Python Deep Learning Library," (https://keras.io, last accessed on January 1, 2019)
6. "Tensorflow: An Open Source Machine Learning Framework for Everyone," (https://www.tensorflow.org, last accessed on January 1, 2019)
7. "Exeinfo PE: A Packer, Compressor Detector," (http://exeinfo.atwebpages.com, last accessed on January 1, 2019)
8. J. Kinder and H. Veith, "Jakstab: A Static Analysis Platform for Binaries," in *Proceedings of International Conference on Computer Aided Verification*, pp. 423-427, Springer, 2008
9. "The MASM32 SDK," (http://www.masm32.net/, last accessed on January 1, 2019)
10. M. A Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient Static Binary Instrumentation for Linux," in *Proceedings of 2010 IEEE International Symposium on Performance Analysis of Systems & Software* (*ISPASS*), pp. 175-183, 2010
11. S. Nanda, W. Li, L. Lam, and T. Chiueh, "Bird: Binary Interpretation using Runtime Disassembly," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 358-370, IEEE Computer Society, 2006
12. B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "PLTO: A Link-time Optimizer for the Intel IA-32 Architecture," in *Proceedings of 2001 Workshop on Binary Translation* (*WBT-2001*), 2001
13. M. Prasad and T. Chiueh, "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks," in *Proceedings of USENIX Annual Technical Conference, General Track*, pp. 211-224, 2003
14. M. Smithson, K. An, A. Kotha, K. Elwazeer, N. Giles, and R. Barua, "Binary Rewriting Without Relocation Information," University of Maryland, Tech. Rep, 2010
15. B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited," in *Proceedings of 2002 Ninth Working Conference on Reverse Engineering*, pp. 45-54, IEEE, 2002
16. R. Wartell, Y. Zhou, K. W Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating Code from Data in x86 Binaries," *Machine Learning and Knowledge Discovery in Databases*, pp. 522-536, Springer, 2011

17. C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *Proceedings of the 13th Conference on USENIX Security Symposium*, Vol. 13, pp. 18, 2004
18. L. C. Harris and B. P. Miller, "Practical Analysis of Stripped Binary Code," in *Proceedings of Workshop on Binary Instrumentation and Applications* (*WBIA2005*), 2005
19. "The IDA Pro Disassembler and Debugger," (https://www.hex-rays.com/products/ida/index.shtml, last accessed on January 1, 2019)

Jing Qiu received his B.S. degree from Harbin Institute of Technology in 2005 and his M.S. degree from Harbin University of Science and Technology in 2009. He received his Ph.D. from Harbin Institute of Technology in 2015. He currently works at Harbin University of Science and Technology. His main interests include binary code analysis and binary code deobfuscation.

Guanglu Sun received his bachelor's degree, master's degree, and Ph.D. from the School of Computer Science and Technology at Harbin Institute of Technology. He was an assistant researcher at the Post-doctoral Mobile Station in Tsinghua University's Computer Science Department from 2008 to 2011. He was a visiting scholar at Northwestern University from 2014 to 2015. He is currently a professor in the School of Computer Science and Technology and the director of the Center of Information Security and Intelligent Technology at Harbin University of Science and Technology. He is also a senior member of the China Computer Federation and a member of IEEE. His current research interests include computer networks and security, machine learning, and intelligent information processing.