An Iteration Obfuscation Based on Instruction Fragment Diversification and Control Flow Randomization

Xin Xie, Fenlin Liu, Bin Lu, and Fei Xiang

***Abstract*—As the control flow graph can reflect the logic structure of programs, static and dynamic reverse methods are used to analyze the logic structure and instruction sequence, and the existing methods of control flow obfuscation have low potency to resist reverse attacks. To solve this problem, we propose an obfuscation method based on instruction fragment diversification and control flow randomization, diversified instruction fragments are generated by various equivalent transformation rules, and random functions are used to select one execution path from the multi-way branches of programs, then programs are iteratively obfuscated. Experiments and analysis show that diversified instruction fragments and multi-way branches can increase the difficulty of static reverse analysis, random selection for multi-way branches will increase the difficulty of dynamic instruction tracing, and iterative transformation for many times enhances the complexity of control flow graph.**

***Index Terms*—Code obfuscation, iterative transformation, instruction fragment diversification, control flow randomization.**

I. Introduction

Malicious attackers are always locating, analyzing and extracting important codes in programs by static and dynamic reverse analysis. Through malicious modification, core modules can be migrated to other programs which can be sold as new software products. This process seriously violates software intellectual property [1]. In order to protect software code, many technologies have been proposed, such as code obfuscation [2], software watermarking [3], software tamper-proofing [4] and software birthmarking [5]. They can be used to prevent software from reverse analysis, illegal distribution, copying and malicious tampering.

Code obfuscation is a typical software protection technology. On the basis of keeping program function, code are semantics-preserving transformed to be more difficult for attackers to reverse analyze and understand. The existing obfuscation techniques can be classified into four categories, including control obfuscation [6], data obfuscation [7], layout obfuscation, and preventive obfuscation [8]. Control obfuscation can make program logic complicated by modifying or hiding the real control flow information, the main methods are opaque predicates inserting, control structure flattening, and control flow hiding. Obfuscation algorithms based on opaque predicates add true, false and uncertain condition branch states into programs, thus it can make control flow more complicated. Obfuscation algorithms based on control flow flattening increase the number of basic blocks and control flow edges, and they can increase the complexity of control flow graph. Obfuscation algorithms based on control flow hiding transform control flow into data or exception information.

Existing methods of control flow obfuscation have increased complexity and stealthiness of control flow to a certain extent, and they can also resist static reverse analysis. However, both static and dynamic methods are used to analyze control flows. Control flow can also be analyzed by tracking instruction execution traces. Therefore the potency of existing control flow obfuscation should be improved for the goal of resisting static and dynamic reverse analysis. To solve this problem, iteration obfuscation based on instruction fragment diversification and control flow randomization is proposed. To split basic blocks, diversified instruction fragments are generated by equivalent transformation rules, and multi-way branches are constructed. These instruction fragments can be executed in some branches. Random functions are used to select different branches, and instruction fragments in basic blocks can be iteratively obfuscated.

II. Related Works

The first obfuscation was proposed by Diffie and Hellman [9] when they did not use the obfuscation word to describe the technique. Collberg *et al.* introduced the technique to protect Java code program. In recent years, many researchers have proposed various obfuscation methods that can mainly fall into five types, including data obfuscation, control flow obfuscation, layout obfuscation, preventive obfuscation and some other advanced obfuscation.

Data obfuscation is described as breaking abstractions and data structures. Collberg *et al.* suggested restructuring program arrays by splitting, merging, folding and flattening [8]. They described array obfuscation by simple examples, but did not show detailed algorithm. Zhu *et al.* applied homomorphic functions to improve the potency of program arrays obfuscation [10]. They made program arrays more obscure to be understood, but did not give the formal description of data obfuscation. Drape *et al.* regarded obfuscation as data refinement to prove the correctness of obfuscation and generalize array splitting obfuscation [11],

Manuscript received November 9, 2014; revised May 12, 2015. This work was supported by the grants of National Natural Science Foundation of China No. 61379151, 61274189, 61302159 and 61401512. Excellent Youth Foundation of Henan Province of China No. 144100510001.

Xin Xie is with Zhengzhou Information Science and Technology Institute, and the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, Henan 450002, China (e-mail: xiexin0011@gmail.com).

Fenlin Liu, Bin Lu, and Fei Xiang are with Zhengzhou Information Science and Technology Institute, China (e-mail: liufenlin@vip.sina.com, stoneclever@gmail.com, xiangfei2012@tsinghua.org.cn).

then they established a framework of the above theory [12]. Xin *et al*. designed an automatic data structure obfuscation technique as a GNU GCC compiler extension to implement field reordering [13], and more specific approaches, such as garbage field inserting and structure splitting, are under consideration.

Control flow obfuscation obstructs the control flow information of programs. Collberg *et al*. constructed opaque predicates based on numerical theory and pointer aliasing technology, and the branch states in program were modified by opaque predicates [14]. Yang *et al*. proposed the multi-point function opaque predicate obfuscation algorithm, which could make opaque predicates interdependent and avoid reverse slicing attacks [15]. Wang *et al*. transformed direct control jumps into indirect control jumps by pointer aliasing, and implemented structure flattening of control flow in procedure [16]. Considering the difficulty of intra-procedure calling analysis, Ogiso *et al*. transformed control structure of intra-procedure by function pointers [17]. Toyofuku *et al.* proposed to use random values to choose method calls in programs that could complicate method call graphs [18]. Popov *et al*. introduced exception handling mechanism to modify and replace control flow of programs that could hide the branch information in control flow [19]. Balachandran and Emmanuel proposed control obfuscation based on stack self-modified code, and the method removed control flow information from code section and hid them in data section [20].

Layout obfuscation obscures layouts of programs on the intermediate language code or source code. Chan *et al.* suggested overusing an identifier to improve Java layout obfuscation based on substituting the identifiers with meaningless names [21] Xu *et al*. applied whitespace randomization, comments randomization, and variable and function names randomization to JavaScript [22], these methods were all for source codetransformation.

Preventive obfuscation makes specific deobfuscation techniques more difficult to succeed. Linn *et al*. designed an obfuscation to thwart linear sweep and recursive traversal static disassembly algorithms [23], which introduced junk instructions, jump tables, and opaque predicates to disrupt the static disassembly process. Ledoux *et al*. proposed an instruction embedding method to improve instruction overlapping obfuscation [24]. Batchelder *et al.* presented layout obfuscation algorithm to make Java decompilers fail to produce legal source code or crash completely [25].

Advanced obfuscation transforms programs at run-time or deploys important code on the remote trusted entity. Anckaert *et al*. suggested a compromise between distributing identical copies and unique executions by diversifying at run-time [26], which made it harder to zoom in on a point of interest and may fool an attacker into believing that he had succeeded. Abadi *et al.* presented layout randomization for obfuscation and studied it in programming-language terms [27]. Roeder *et al*. designed proactive obfuscation that periodically restarting servers with these diverse versions [28], and the periodic restarts helped to bound the number of compromised replicas that a service ever concurrently ran to make an adversary's job harder. Wang *et al*. proposed a branch obfuscation approach that replaced explicit conditional jump instructions with implicit trap code and bogus code, and deployed jump conditions on the remote trusted entity [29].

III. Iteration Obfuscation Based on Instruction Fragment Diversification and Control Flow Randomization

A. Equivalent Transformation Rules

In order to implement instruction fragment diversification, some equivalent transformation rules are proposed such as instruction position transformation, register transformation, junk instruction insertion, equivalent instruction replacement, and instruction expansion transformation [30].

1) Instruction position transformation rule

When we exchange the positions of instructions which have no dependence, it will not influence the execution of programs, thus diversified instruction fragments can be generated. Let there be no dependence in n instructions (a_1, a_2,…, a_n), where instructions can be exchanged arbitrarily, therefore $n!$ different kinds of instruction sequences can be got. To analyze dependence between instructions in the sequence, position transformation rules are listed as follows:

1) If instructions define and use the same common register, their positions cannot be exchanged.
2) If instructions impact on the same flag register, their positions cannot be exchanged.
3) If instructions define and use the same memory, their positions cannot be exchanged.
4) If instructions push parameters and then call functions or APIs, their positions cannot be exchanged.

Example code fragment extracted from program is shown in Table I. By instruction position transformation rule, there is no control and data dependence among these instructions, and they can be arrayed as 6 different kinds of sequences.

TABLE I: Position Transformation of Example Code Fragment

(1) mov eax, 1	(1) xor ebx, ebx	(1) lodsd
(2) xor ebx,ebx	(2) mov eax,1	(2) mov eax, 1
(3) lodsd	(3) lodsd	(3) xor ebx,ebx
(1) lodsd	(1) xor ebx, ebx	(1) mov eax,1
(2) xor ebx, ebx	(2) lodsd	(2) lodsd
(3) mov eax,1	(3) mov eax,1	(3) xor ebx,ebx

2) Register transformation rule

On the basis of not affecting program normal execution, some registers in code fragments can be replaced by other registers, such as *mov reg, 0x1234, mov [0x5678], reg*, where *reg* can be the register that is randomly selected from *eax, ebx, ecx, edx* and so on. Register transformation rule can also be shown as follows: firstly values of needed registers are saved at the beginning of code fragments, and then these registers are used to replace other different registers, finally values of needed registers will be restored after the process of replacing, such as *and reg1, reg2, sub reg2, reg1* can be transformed into *push reg3, push reg4, and reg3, reg4, sub reg4, reg3, mov reg1, reg3, mov reg2, reg4, pop reg4, pop reg3*.

3) Junk instruction insertion rule

There are many forms of junk instructions that will not affect execution of instruction sequence in program. Some

junk instructions are as follows:

1) Nop instruction.
2) Add reg, value; sub reg, -value; push reg; pop reg.
3) Jmp offset address@, push eax, push ebx, call API (junk instructions), @mov eax, ebx, add eax, ebx (useful instructions).
4) mov eax, value1, add eax, value2, sub eax, value3 (junk instructions), mov eax ,reg1 (useful instructions).
5) Other junk instruction sequences.

4) Equivalent instruction replacement rule

Because an instruction can be replaced by another equivalent instruction, one code fragment is constructed by instruction sequence (I_1, I_2,…, I_n), where instruction I_i has the same function as I_j, thus I_i and I_j are equivalent. By replacing all instructions I_i with instructions I_j in instruction sequence, the semantics of instruction sequence will not be changed. Some replacement rules of equivalent instruction are shown in Table II.

TABLE II: SOME REPLACEMENT RULES OF EQUIVALENT INSTRUCTION

Original Instruction	Equivalent Instruction
Sub<register/memory>,<value>	Add<register>,-<value>
Not<register/[memory]>	Xor<register/[memory]>,-1
Nop	And<register/[memory]>,-1
Sub<register/memory>,<value>	Lea<register>, DWORD[<register>-<value>]
Mov<register a>,<register b>	Lea<register a>,DWORD[<register b>]
Mov<register>,<register>	Shl<register>,0
Add<register/memory>,<value>	Sub<register>,-<value>
Inc<register/[memory]>	Add<register/[memory]>,1
Lea<register>,[<register>]	Add<register/[memory]>,0

5) Instruction expansion transformation rule

TABLE III: SOME EXPANSION TRANSFORMATION RULES OF EQUIVALENT INSTRUCTIONS

Original Instruction	Equivalent Instruction
Push<value/register>	Mov DWORD PTR SS: [ESP-4], <Value/register> Sub esp,4
Mov<register a>,<register b>	Push<register a> Pop<register b>
Jmp<address>	Push<address> Retn
Pop<register/[memory]>	Mov <register>,DWORD [esp] Add Esp,4
Call<address>	Push EIP+bytes to next instruction Jmp<address>
Mov<register a>,<register b>	Mov<[memory]>,<register a> Mov<register b>,<[memory]>
Not<register/[memory]>	Neg<register/[memory]> Sub<register/[memory]>,1
Push<value/register>	Mov<register/[memory]>, <value/register> Push<register/[memory]>
Retn	Pop eax Jmp eax
Pop<register/[memory]>	Pop<[memory]> Mov<register>,<[memory]>
Call<address>	Mov<[memory]/register>,<address> Call<[memory]/register>
Neg<register/[memory]>	Not<register/[memory]> Add<register/[memory]>,1

Instruction expansion transformation refers to replace one instruction with instruction sequence of the same semantics. Code fragment is constructed by instruction sequence (I_1, I_2,…, I_n), where instruction I_i has the same function as (I_{p1},…, I_{pm}), thus instruction sequence (I_{p1}, I_{p2},…, I_{pm}) is a transformation rule of equivalent expansion for instruction I_i, instruction sequence semantics will not be changed by replacing instruction I_i with (I_{p1},…, I_{pm}). Some transformation rules of instruction expansion are shown in Table III. Another instruction expansion method is to make arithmetic and logical operations be more complicated, such as the formula of $A+B=A-(-B)$, $A\hat{}B=(A\&\sim B)|(\sim A\&B)$ and so on.

B. Instruction Sequence Diversification

Generally, single instruction sequence has a variety of semantics-preserving expression forms, such as instruction sequence $S=(o_1, o_2,\ldots, o_n)$, where o denotes an instruction, instruction sequence S can be transformed into many sequences S_1, S_2,…, S_n by equivalent transformation rules. If equivalent instruction sequences S_1, S_2,…, S_n are different from S, but they implement the same function as S, they are the diversified sequences of S.

Algorithm1 Diversified Instruction Fragments Generation Algorithm
Input: Instruction Fragment $S=(o_1,o_2,\ldots,o_n)$, Iteration Number m
Output: Set of Diversified Instruction Fragments VS

```
1: S=(o1,o2,…,on);
2: k=1, d=0;
3: while k<=n do
4:   INS={ok}
5:   while d<m do
6:     ins←random(INS);//Choose any ins in INS
7:     T=Transform(ins);//Expansion Transform of chosen instruction
8:     if T!=NULL then
9:       INStmp=T;
10:      INS= S + INStmp- ins;
11:      S=S + INStmp- ins;
12:      VS←S;
13:    end if
14:    d++;
15:  end while
16: k++;
17: end while
18: return VS
```

In order to increase the number of diversified sequences, instruction fragments in programs are iteratively obfuscated with instruction expansion transformation rules [31], as a result, the number of instruction sequences will be exponentially increased. Diversified instruction fragments generation algorithm is shown in Algorithm 1. Because there are many equivalent transformation rules, the composite transformation rules can be used to generate diversified instruction fragments.

C. Control Flow Randomization

Equivalent instruction fragments are the sequences in program execution paths, and multi-way branches are randomly constructed in programs. When a branch state appears in a program, a random function is used to select different branches. Due to the random selection of branches, the execution path is different each time.

Considering the generation speed and randomness, Microsoft security cookie mechanism is used for reference. The value of m_i can be obtained by calling system APIs during the execution of program, random value *Vr* calculated by formula (1) is used to be the seed of function *srand()*, where

m_1 (*GetSystemTimeAsFileTime*) is the current system time; m_2 (*GetCurrentProcessId*) is the current process ID; m_3 (*GetCurrentThreadId)* is the current execution thread ID*;* m_4 (*GetTickCount)* is the millisecond of the current operating system from boot to now*;* m_5 *(QueryPerformanceCounter)* is the CPU clock frequency.

$$Vr=Hash(m_1 \oplus m_2 \oplus m_3 \oplus \ldots \oplus m_5) \quad (1)$$

Code snippets of control flow randomization uses random value of *Vr* as a seed of random number generator, function *rand()* is used to generate pseudo random number, and the generated value is used to randomly select multi-way branches in the program, it is shown in Table IV, where H_0-H_3 have the same function.

TABLE IV: CODE SNIPPETS OF CONTROL FLOW RANDOMIZATION

```
1：Vr=Hash(m1 ⊕ m2 ⊕ m3 ⊕ … ⊕ m5)
2：srand(Vr);
3：N=rand()%4;
4：switch(N)
5：{
6：  case 0: H0, break;
7：  case 1: H1, break;
8：  case 2: H2, break;
9：  case 3: H3, break;
10：}
```

D. Obfuscation Step

Step 1: All head instructions are labeled including first instruction in function, target instruction of transfer instruction, instruction followed by transfer instruction. Basic blocks are divided according to head instructions, and then control flow graph $G=(V, E)$ is built.

Step 2: The basic blocks in control flow graph are chosen if the number of instructions are greater than 2, then they can be applied with diversified transformation rules.

Step 3: A single basic block will be divided into three parts: the first and last instruction in basic block, and the instruction sequence between them, denoted by $B=\{I_{start}, I_{end}, Seq\}$.

Step 4: The instruction sequence between the first and last instruction in basic block will be divided into two parts, denoted by $Seq=\{B_{seq1}, B_{seq2}\}$. The set of diversified instruction fragments are constructed with transformation rules, such as instruction position transformation, register transformation, junk instruction insertion, equivalent instruction replacement, and instruction expansion transformation, the instruction fragment set can be denoted by $\{\{B^1_{seq1}, B^2_{seq1}, \ldots B^n_{seq1}\}, \{B^1_{seq2}, B^2_{seq2}, B^3_{seq2} \ldots B^m_{seq2}\}\}$.

Step 5: Two layers of multi-way branches are built based on the set of diversified instruction fragments, such as $E=\{(B^1_{seq1}, B^1_{seq2}), (B^1_{seq1}, B^4_{seq2}), (B^1_{seq1}, B^j_{seq2}), (B^2_{seq1}, B^1_{seq2}), (B^2_{seq1}, B^2_{seq2}), (B^2_{seq1}, B^k_{seq2}) \cdots (B^n_{seq1}, B^n_{seq2}), (B^n_{seq1}, B^m_{seq2})\}$.

Step 6: Randomization of multi-way branches in a basic block is implemented by using random function. The basic block semantics will be not changed.

Step 7: After reiteratively executing step 3-6 for instruction fragments, the basic block will be more complicated.

Step 8: All basic blocks are iteratively obfuscated with diversified splitting rule.

Through the above steps, two basic blocks in Fig. 1(a) will be iteratively obfuscated. Control flow graph will be transformed into Fig. 1(b) with the iterative obfuscation rule first time. If the iterative obfuscation rule is used once again on the gray basic blocks in Fig. 1(b), control flow graph will be transformed into Fig. 1(c). There is no doubt that the number of basic blocks and control flow edges will increase greatly after many times of obfuscation, and the control flow graph will be more complicated, thus it can signally increase the difficulty of static and dynamic reverse analysis.

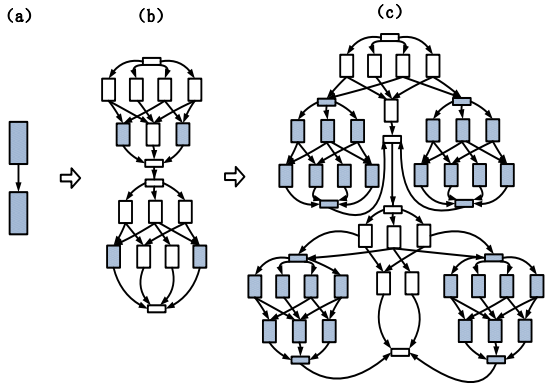

Fig. 1. Iterative obfuscation based on basic block diversity.

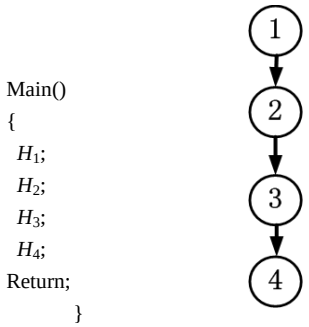

Fig. 2. Code snippets and control flow graph of original program.

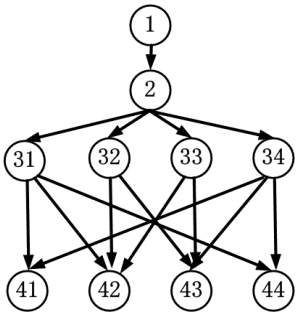

Fig. 3. Code snippets and control flow graph of obfuscated program.

E. A Case in Study

If an instruction fragment in a basic block can be divided into the execution modules such as H_1, H_2, H_3, and H_4, the code snippets and control flow graph are shown in Fig. 2. Obfuscated program is constructed by instruction fragment diversification and control flow randomization, as shown in Fig. 3, where instruction fragments (31, 32, 33, 34) and (41, 42, 43, 44) are generated from the instruction fragments 3 and 4 with diversified transformation rules. The original basic block only has one execute path, such as 1->2->3->4. Fig. 3 reflects that the number of execution paths in obfuscated program has increased to 10, including 1->2->31->41, 1->2->33->42 and so on. Obfuscation has greatly increased the complexity of control flow graph and the difficulty of static and dynamic reverse analysis.

IV. Metric Analysis

A. Potency Measurement

Potency includes the performance of obfuscation to resist static and dynamic reverse analysis. Static reverse analysis is used to analyze the nodes and edges of control flow graph nodes in control flow graph are instruction fragments generated with diversified transformation rules, and edges are execution paths that can be selected by random functions. Dynamic reverse analysis is used to analyze instruction traces.

Let the length of basic block $S=(o_1,o_2,\ldots,o_n)$ be n. The basic block is divided into two parts by reserving the first instruction o_1 and the last instruction o_n, and the basic block can denote $S=\{(o_1),(o_2,\ldots,o_l),(o_{l+1},\ldots o_{n-1}),(o_n)\}$. The instruction fragments $(o_2,\ldots,o_l)$ and $(o_{l+1},\ldots o_{n-1})$ can be transformed with equivalent transformation rules. Let the length of $(o_2,\ldots,o_l)$ be l-1. Each instruction in the sequence will be extended to k instructions, and $C_{l-1}^1+C_{l-1}^2+\ldots+C_{l-1}^{l-1}=2^{l-1}-1$ kinds of instruction fragments will be generated after first time obfuscation. And $C_{l-1}^1(2^k-1)+C_{l-1}^2(2^{2k}-1)+\ldots+C_{l-1}^{l-1}(2^{(l-1)k}-1)$ kinds will be generated after second time. On the basis of these fragments, the number of instruction fragments will increase after implementing composite transformation rules. Let instruction fragments $(o_2,\ldots,o_l)$ and $(o_{l+1},\ldots o_{n-1})$ respectively generate k and d kinds of instruction fragments with diversified transform rules, and they are used to be the instruction execution sequences in multi-way branches. If $k>d$, the minimum number of branch paths is k, and the maximum number of branch paths is kd.

Let the control flow graph of the original program be $G=(V,E)$, where the value of V is v, and value of E is e, so the complexity of graph is $e-v+2$ by cyclomatic complexity measurement [32]. Let the obfuscated instruction fragments be m kinds, and they are generated with equivalent transformation rules, so there are $m*m$ branch paths between instruction fragments. The Number of nodes will be $v(2m+2)$ after first time obfuscation, the number of control flow edges will be $e+v*(m*m+2m)$, and the cyclomatic complexity of control flow graph will be $e+v(m*m+2m)-v(2m+2)+2$. After k^{th} time iterative obfuscation, the nodes of control flow graph will be $v(2\sum_{n=0}^{k}(2m)^n+(2m)^{k+1})$, the edges of control flow graph will be $e+v((2m+m*m)\sum_{n=0}^{k}(2m)^n)$, and the cyclomatic complexity of control flow graph will be $e+v((2m+m*m-2)\sum_{n=0}^{k}(2m)^n-(2m)^{k+1})$.

B. Cost Measurement

Cost measurement mainly considers the execution time and file size of original and obfuscated programs. Let u instructions be executed k times in the condition of looping, if one instruction will cost a unit of storage space and execution time, the cost of space and time will be u and uk respectively. Instruction fragments are generated by transformation rules, and let m kinds of instruction fragments be generated after first time obfuscation, and obfuscation will be applied to the instruction fragments for n times. If the length of one instruction fragment is s_n, the time cost of obfuscated program is $k(2^n s_n+\sum_{n=1}^{k}2^n)$, and the space cost of obfuscated program is $2\sum_{n=0}^{k}(2m)^n+s_n(2m)^{k+1}$.

As the obfuscation time increased, the structure of control flow graph will be greatly complicated, and the file size of program will increase, and the number of instructions will be expanded. When we use the iterative obfuscation to transform programs, the time of iteration need to be controlled for the trade-off among space cost, time cost and potency.

C. Deobfuscation Measurement

The existing deobfuscation algorithms are based on the optimization theory [33], including peephole optimization, constant propagation, constant folding, operand folding, and stack optimization. These methods can remove junk instructions and invalid branches in functions that deobfuscate redundant codes to a certain extent.

Diversified instruction fragments are generated with junk instructions insertion, register replacement and some other transformations, therefore deobfuscation algorithms can optimize some instruction fragments. Equivalent instruction fragments in multi-way branches are generated by equivalent rules of instruction replacement and instruction expansion that present diversity. If these fragments will be normalized, the semantics of different instruction fragments will be analyzed, and it is very difficult to deobfuscate the obfuscation.

D. Stealth Measurement

Stealth measures the difficulty of recognizing whether obfuscation or not, the similarity of original and obfuscated programs can be the stealth metric of quantitative analysis. Similarity of programs can be calculated with sequence, set and graph formula [34].

To measure the stealth of obfuscation, similarities of original and obfuscated programs are calculated, similarity formula based on maximum common subgraph is used. Let G, G_1 and G_2 be graphs, if a subgraph in G_1 is isomorphism as G, and a subgraph in G_2 is isomorphism as G, so that G is the common graph between G_1 and G_2. If there be no common

subgraph *G'* that has more nodes than *G*, and *G* will be the maximum subgraph of G_1 and G_2 denoted by $G=mcs(G_1, G_2)$. The similarity of G_1 and G_2 is shown in formula (2).

$$sim(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{\max(|G_1|, |G_2|)}, \text{where } |G| = |v| + |e| \quad (2)$$

TABLE V: SIZE OF ORIGINAL AND OBFUSCATED PROGRAMS

Program Name	Original program size (kb)	Program size of one time iteration (kb)	Program size of two times iteration (kb)	Program size of three times iteration (kb)
ShellSort.exe	29	31.5	32.5	34
InsertionSort.exe	29	31.5	32.5	38.5
BubbleSort.exe	29	31.5	33	42.5
QuickSort.exe	29	32	33.5	42.5

TABLE VI: EXECUTION TIME OF ORIGINAL AND OBFUSCATED PROGRAMS

Program Name	Original program execution time (ms)	Program execution time of one time iteration (ms)	Program execution time of two times iteration (ms)	Program execution time of three times iteration (ms)
ShellSort.exe	30.7	320.6	495.9	633.6
InsertionSort.exe	5529.9	63763.1	170169.4	339284.4
BubbleSort.exe	14507.6	55820.5	58027.2	60454.5
QuickSort.exe	15.4	39.3	69.3	138.9

TABLE VII: BASIC INFORMATION OF DEOBFUSCATION PROGRAMS

Program Name	File Size(kb)	Execution Time(ms)	Number of Basic Block	Number of Edge	Number of Instruction
Shell1_deo	31	310.7	21	27	122
Shell2_deo	32	452.6	48	66	354
Shell3_deo	33	603.8	113	158	624
Insert1_deo	31	6074.5	17	22	95
Insert2_deo	32	146632.1	58	81	371
Insert3_deo	36	296981.6	363	376	1453
Bubble1_deo	31	48336.1	17	22	99
Bubble2_deo	32	53779.5	84	120	443
Bubble3_deo	40	578841.2	368	532	2333
Quick1_deo	30.5	32.6	32	47	187
Quick2_deo	31.5	55.8	113	170	610
Quick3_deo	40	113.3	514	751	2764

V. EXPERIMENT AND ANALYSIS

Based on the environment that is *Intel(R) Core(TM)2 CPU 1.86GHz, Microsoft Windows XP Professional 5.1.2600 Service Pack 3*, *Visual Studio 2008*, with the obfuscation steps, obfuscation based on instruction fragment diversification and control flow randomization is implemented, programs of Shell Sorting, Insertion Sorting, Bubble Sorting and Quick Sorting are obfuscated, the source code of four programs are shown in *Appendix A*, static reverse analysis tool of IDA[1] is used to analyze the obfuscated programs, and the performance of original and obfuscated programs are compared.

A. Cost Validation

Four debug version programs are built, and their control flow graphs are constructed by static disassembly. They are obfuscated by the followed method. Firstly, instruction fragment will be divided into two parts that will generate three kinds of instruction fragments respectively with the instruction equivalent transformation rules, and the forms of these fragments are different, but they are semantics-preserving. Random function is used to choose these instruction fragments. When the iteration times are 1, 2 and 3 respectively, a set of obfuscated programs are constructed, the size of original and obfuscated programs are shown in Table V. As the iteration times increased, the file size of programs will be increased. To reduce other factors that may affect program execution, the average execution time of 10 times are measured. Sorting programs are used to sort 50000 numbers generated by random function, the execution time of original and obfuscated programs are shown in Table VI. Due to the various sorting algorithms, execution time of different sorting programs is different. As the iteration time increased, and the number of random function and instruction is increased, it will enhance the program time overhead.

B. Potency Validation

Benchmark programs are obfuscated by obfuscation as iteration time being 1, 2 and 3. With the IDA plus, the number of basic blocks, edges and instructions can be gained that are shown in Fig. 4-Fig. 6. The numbers of basic blocks in four original programs are shown in Fig. 4, they are 14, 10, 11 and 11, and they will be exponentially increased with instruction fragments diversification, as iteration time be 3, the numbers of basic blocks are 113, 363, 368 and 514. The numbers of control flow edges in four original programs are shown in Fig. 5, they are 17, 12, 13 and 15. With multi-way branches and random functions, control flow paths are increased greatly, when iteration time is 3, they are 158, 376, 532 and 751. The numbers of instructions are shown in Fig. 6, the total numbers of instructions of programs are 74, 54, 56 and 71. With equivalent instruction expansion rules, diversified instruction fragments will be generated, and the numbers of instructions will be increased. With one time iteration, the numbers of instructions are 157, 119, 124 and 221. While the iteration time is three, the numbers of instructions in four programs are 860, 1949, 2869 and 3230. After many times iteration, it will be more difficult for attackers to reverse analyze obfuscated programs.

[1] IDA Multi-Processor Disassembler and Debugger. http://www.hex-rays.com

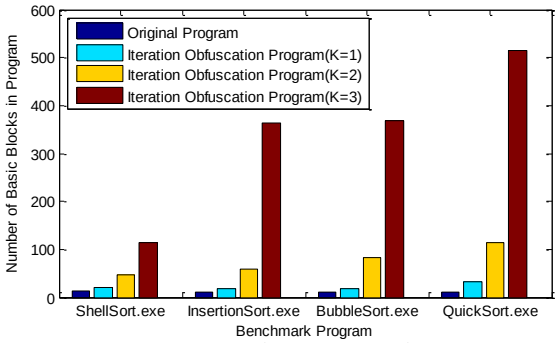

Fig. 4. Basic blocks number of original and obfuscated programs.

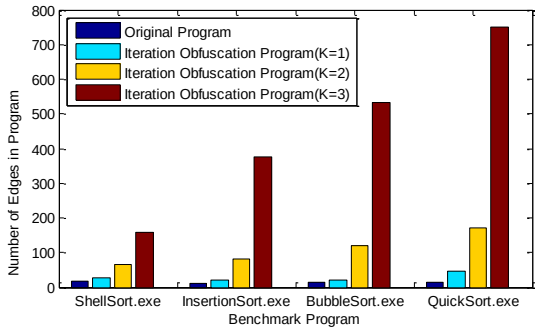

Fig. 5. Control edges number of original and obfuscated programs.

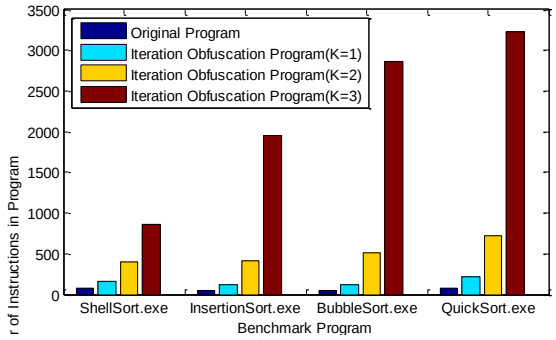

Fig. 6. Instructions number of original and obfuscated programs.

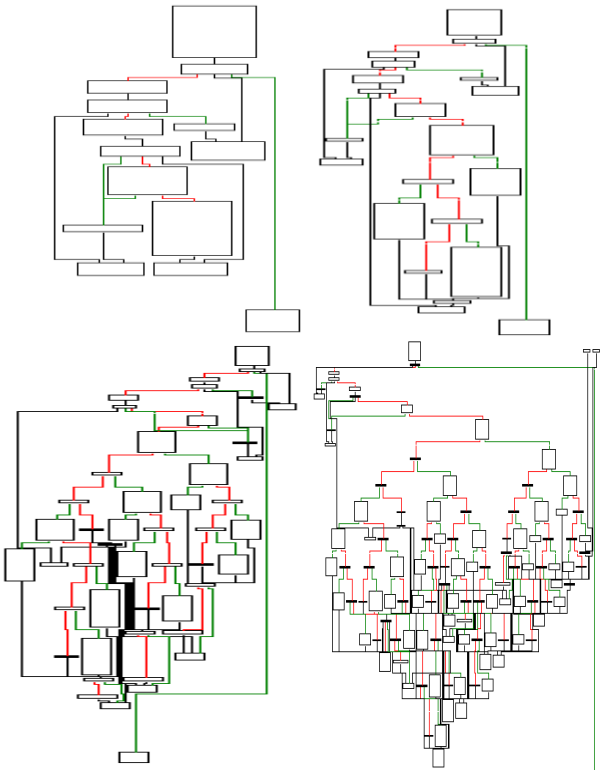

Fig. 7. Control flow graph of original and obfuscated (1-3 times iteration) shell sorting program.

Obfuscated programs of Shell Sorting, Insertion Sorting, Quick Sorting, and Bubble Sorting are constructed, and their control flow graphs are generated by IDA, all of them are shown in Fig. 7-Fig. 10. The original control flow graphs of four sort programs are simple, as the iteration time increasing, their control flow graphs will be complicated, and the difficulty of static and dynamic reverse analysis will be increased.

Execution traces of four sorting programs can be obtained by dynamic tracking. With the same input for many times, the execution traces of program are same. Through iteration obfuscation, control flow structure of sorting programs will be complicated, and the execution paths will be diversified. Each execution path will be different by affecting instruction fragment diversification and control flow randomization, and iteration obfuscation can resist dynamic reverse analysis to a certain extent.

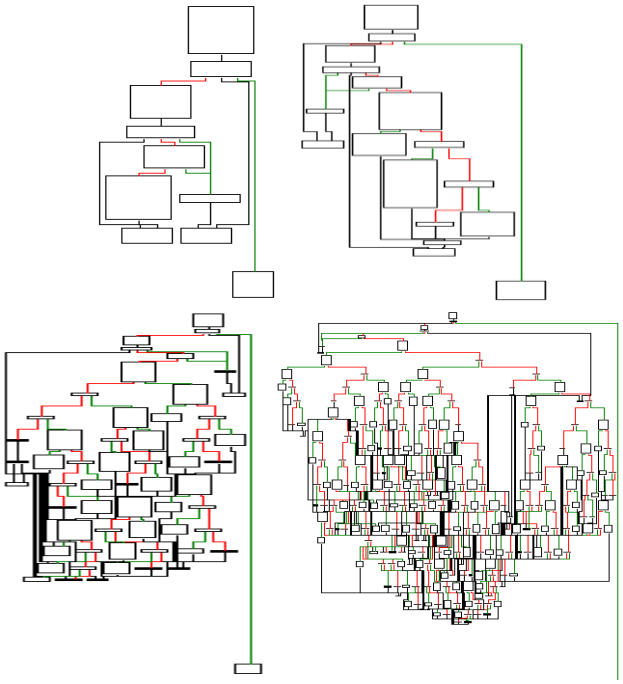

Fig. 8. Control flow graph of original and obfuscated (1-3 times iteration) insertion sorting program.

C. Deobfuscation Validation

Open source tool optimice[2] is used to attack obfuscated programs, and it can validate the deobfuscation performance of obfuscation. The tool implements control flow reduction, junk instruction removing, constant propagation and folding. When diversified instruction fragments are generated with junk instruction insertion, deobfuscation tool can effectively remove the invalid instructions in multi-way branches, but cannot normalize the instruction fragments in different branches. Thus the number of instructions in basic block will be reduced, but the control flow graph will not be changed. Four sort programs are deobfuscated by the tool, the detail information are shown in Table VII. The numbers of basic blocks and edges in obfuscated programs will not be changed by deobfuscation, their control flow graph will not be changed. Due to the deobfuscation algorithm, the size and instruction

[2]Code deobfuscation by optimization. http://code.google.com/p/optimice

number of programs are decreased to some extent, thus which decreases the program execution times, and it can be validated that iteration obfuscation have ability to resist the deobfuscation attacks.

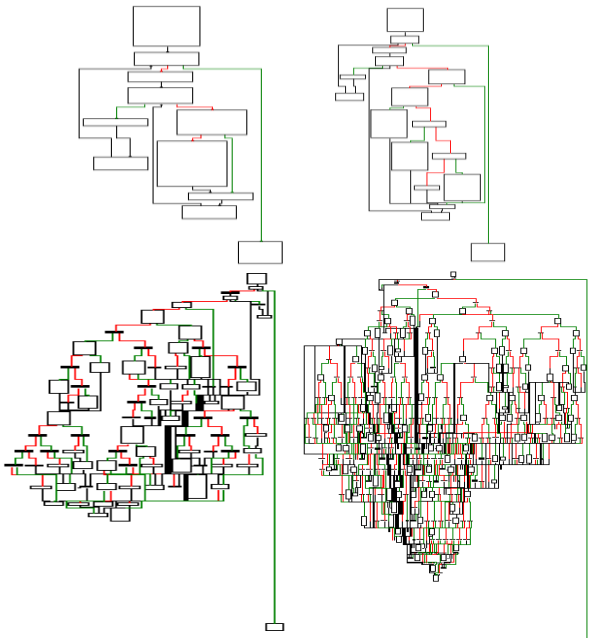

Fig. 9. Control flow graph of original and obfuscated (1-3 times iteration) quick sorting program.

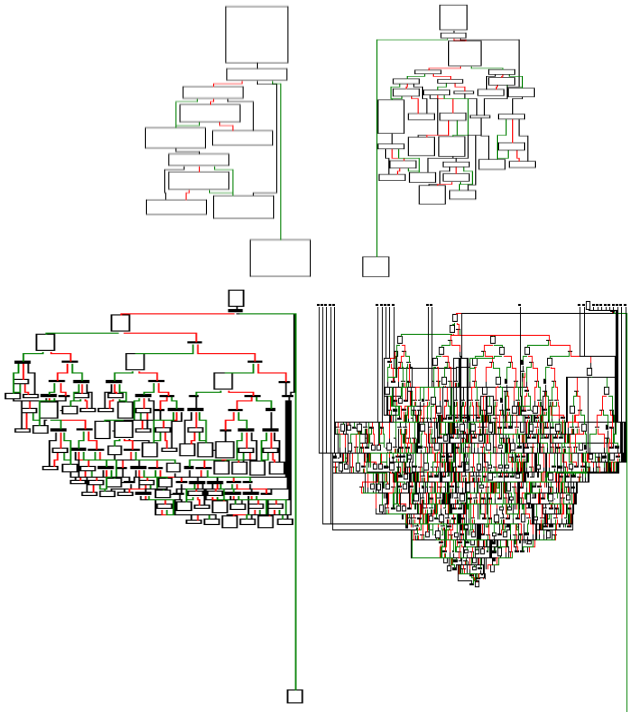

Fig. 10. Control flow graph of original and obfuscated (1-3 times iteration) bubble sorting program.

D. Stealth Validation

Stealth can be determined by the similarity of original and obfuscated programs. Similarity of original and 3rd time iteratively obfuscated programs are calculated, the values are 8.32%, 12.45%, 7.92%, 8.35% and 7.02%. It reflects that iteration obfuscation greatly increases the number of control flow nodes and edges. With iteration obfuscation, the structure of control flow graph is changed, and the stealth is low on the whole, it is easy for attackers to determine whether program is obfuscated or not.

VI. Conclusion and Future Works

Obfuscation based on instruction fragment diversification and control flow randomization is proposed, the rules of instruction expansion, equivalent instruction replacement, register transformation, junk instruction insertion and instruction position exchanging are used to construct diversified fragments, and multi-way branches are constructed. Instruction fragments are used to be the execution sequences, random functions are used to choose the multi-way branches. By iteration obfuscation, the structure of control flow graph will be complicated, nodes and edges of control flow graph will be greatly increased, and the difficulty of static reverse is increased. Multi-way branches are constructed, instruction trace will be different as the same input, and the difficulty of dynamic reverse is enhanced. In the future work, idea of diversification and randomization will apply to other obfuscation such data obfuscation and layout obfuscation to improve the potency and stealth of the obfuscation.

Appendix A

ShellSort.exe

```
void ShellSort(int v[],int n){
    int gap,i,j,temp;
    for(gap=n/2;gap>0;gap /= 2){
        for(i=gap;i<n;i++){
        for(j=i-gap;(j >= 0) && (v[j] > v[j+gap]);j -= gap ){
                temp=v[j];
                v[j]=v[j+gap];
                v[j+gap]=temp;
            }
        }
    }
}
```

InsertionSort.exe

```
void InsertionSort(int input[],int len){
        int i,j,temp;
        int ran;
        for (i = 1; i < len; i++)
        {
            temp = input[i];
            for (j = i - 1;j>-1&&input[j] > temp ; j--){
                input[j + 1] = input[j];
                input[j] = temp;
            }
        }
}
```

BubbleSort.exe

```
void BubleSort(int a[],int n){
    int i,j,k;
    int ran;
    for(j=0;j<n;j++){
        for(i=0;i<n-j;i++){
            if(a[i]>a[i+1]){
                k=a[i];
                a[i]=a[i+1];
                a[i+1]=k;
            }
        }
    }
```

```
}
```

QuickSort.exe

```
int Partition(int data[],int low,int high){
    int mid;
    int ran;
    data[0]=data[low];
    mid=data[low];
    while(low < high){
        while((low < high) && (data[high] >= mid)){
            --high;
        }
        data[low]=data[high];

        while((low < high) && (data[low] < mid)) {
            ++low;
        }
        data[high]=data[low];
    }
    data[low]=data[0];
    return low;
}

void QuickSort(int data[],int low,int high){
    int mid;
    if(low<high){
        mid=Partition(data,low,high);
        QuickSort(data,low,mid-1);
        QuickSort(data,mid+1,high);
    }
}
```

REFERENCES

[1] H. J. Wang, D. Y. Fang, N. Wang *et al*., "Method to evaluate software protection based on attacking modeling," in *Proc. the 10th International Conference on Embedded and Ubiquitous Computing*, 2013, pp. 837-844.

[2] A. Kulkarni and R. Metta, "A new code obfuscation scheme for software protection," in *Proc. the IEEE 8th International Symposium on Service Oriented System Engineering*, 2014, pp. 409-414.

[3] S. B. Che and Y. M. Wang, "A software watermarking based on PE file with tamper-proof function. Indonesian," *Journal of Electrical Engineering*, vol. 12, issue 2, pp. 1012-1021, 2014.

[4] Y. W. Yu and Y. X. Zhao, "Tamper proofing technique based on three-thread protection and software guard," *Journal of Computer Applications*, vol. 33, issue 1, pp. 1-3, 34, 2013.

[5] Y. Zeng, F. L. Liu, X. Y. Luo *et al.*, "Abstract interpretation-based semantic framework for software birthmark," *Computers & Security*, vol. 31, issue 4, pp. 377-390, 2012.

[6] H. Y. Tsai, Y. L. Huang, and D. A. Wagner, "Graph approach to quantitative analysis of control-flow obfuscating transformations," *IEEE Transactions on Information Forensics and Security*, vol. 4, issue 2, pp. 257-267, 2009.

[7] X. Xie, F. L. Liu, and B. Lu, "A data obfuscation based on state transition graph of mealy automata," in *Proc. the 10th International Conference on Intelligent Computing*, 2014, pp. 520-531.

[8] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Reports 148, Department of Computer Science, University of Auckland, 1997.

[9] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transaction on Information Theory*, vol. 22, issue 6, pp. 644-654, 1976.

[10] W. Zhu, C. D. Thomborson, and F. Y. Wang, "Obfuscate arrays by homomorphic functions," in *Proc. the IEEE Inernational Conference on Granular Computing*, 2006, pp. 770-773.

[11] S. Drape, "Generalising the array split obfuscation," *Information Sciences*, vol. 177, issue 1, pp. 202-219, 2007.

[12] S. Drape, C. Thomborson, and A. Majumdar, "Specifying imperative data obfuscations," in *Proc. the 10th International Conference on Information Security*, 2007, pp. 299-314.

[13] Z. Xin, H. Chen, H. Han *et al*., "Misleading malware similarities analysis by automatic data structure obfuscation," in *Proc. the 13th International Conference on Information Security*, 2010, pp. 181-195.

[14] C Collberg, C Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. the 25th the Annual ACM Symposium on Principles of Programming Languages*, 1998, pp. 184-196.

[15] Y. B. Yang, W. Q. Fan, W. Huang *et al*., "The research of multi-point function opaque predicates obfuscation algorithm," *Applied Mathematics & Information Sciences*, vol. 8, issue 6, pp. 3063-3070, 2014.

[16] C Wang, J Hill, J Knight *et al.*, "Software tamper resistance: obstructing static analysis of programs," Technical Report 12, Department of Computer Science, University of Virginia, 2000.

[17] T. Ogiso, Y. Sakabe, M. Soshi *et al*., "Software obfuscation on a theoretical basis and its implementation," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E86-A, issue 1, pp. 176-186, 2003.

[18] T. Tatsuya, T. Toshihiro and S. Kouichi, "Program obfuscation scheme using random numbers to complicate control flow," in *Proc. the Embedded and Ubiquitous Computing*, 2005, pp. 916-925.

[19] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proc. the 16th USENIX Security Symposium*, 2007, pp. 275-290.

[20] V. Balachandran and S. Emmanuel, "Potent and stealthy control flow obfuscation by stack based self-modifying code," *IEEE Transactions on Information Forensics and Security*, vol. 8, issue 4, pp. 669-681, 2013.

[21] J. Chan and W. Yang, "Advanced obfuscation techniques for Java bytecode," *Journal of System and Software*, vol. 71, issue 1/2, pp. 1-10, 2004.

[22] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious JavaScript code: A measurement study," in *Proc. the 7th IEEE International Conference on Malicious and Unwanted Software*, 2012, pp. 9-16.

[23] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. the 10th International Conference on Computer and communications security*, 2003, pp. 290-299.

[24] C. LeDoux, M. Sharkey, B. Primeaux *et al.*, "Instruction embedding for improved obfuscation," in *Proc. the 50th Annual Southeast Regional Conference*, ACM, 2012, pp. 130-135.

[25] M. Batchelder and L. Hendren, "Obfuscation Java: The most pain for the least gain," in *Proc. the International Conference on Compiler Construction*, 2007, pp. 96-110.

[26] B. Anckaert, M. Jakubowski, R. Venkatesan *et al.*, "Run-time randomization to mitigate tampering," *Advances in Information and Computer Security*, pp. 153-168, 2007.

[27] M. Abadi and G. D. Plotkin, "On protection by layout randomization," *ACM Transaction on Information and System Security*, vol. 8, issue 2, 2012.

[28] T. Roeder and F. B. Schneider, "Proactive obfuscation," *ACM Transaction on Computer and System*, vol. 4, issue 2, 2010.

[29] Z. Wang, C. Jia, M. Liu *et al*., "Branch obfuscation using code mobility and signal," in *Proc. the IEEE 36th International Conference on Computer Software and Applications Workshops*, 2012, pp. 553-558.

[30] Y. Ilsun and Y. Kangbin, "Malware obfuscation techniques: A brief survey," in *Proc. the International Conference on Broadband, Wireless Computing, Communication and Applications*, pp. 297-300, 2010.

[31] H. Y. Wang, D. Y. Fang, G. H. Li *et al*., "TDVMP: improved virtual machine-based software protection with time diversity," in *Proc. the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2014.

[32] M. Thomas, "A complexity measure," *IEEE Transaction on Software* Engineering, vol. SE-2, issue 4, pp. 308-320, 1976.

[33] Y. Guillot and A. Gazet, "Automatic binary deobfuscation," *Journal in Computer Virology*, vol. 6, pp. 261-276, 2010.

[34] C. Silvio and X. Yang, *Software Similarity and Classification*, London: Springer, 2012, pp. 63-70.

Xin Xie received the B.S. degree and the M.S. degree from Zhengzhou Information Science and Technology Institute, Zhengzhou, China, in 2008 and 2011, respectively. Currently, he is a Ph.D. candidate of Zhengzhou Information Science and Technology Institute. His research interest includes software security, software protection and code obfuscation.

Fenlin Liu is currently a professor of Zhengzhou Information Science and Technology Institute. His research interests include information security and software protection theory. He is the author or co-author of more than 100 refereed international journal and conference papers. He obtained the support of the National Natural Science Foundation of China and the Found of Innovation Scientists and Technicians Outstanding Talents of Henan Province.

Bin Lu received the B.S. degree and the M.S. degree from Zhengzhou Information Science and Technology Institute, Zhengzhou, China, in 2004 and 2007, respectively.

Currently, he is a Ph.D. candidate of Zhengzhou Information Science and Technology Institute. His research interest includes software security, software protection and cyberspace security.

Fei Xiang received the B.S. degree from the Department of Computer Science and Technology of Tsinghua University, Beijing, China, in 2012. Currently, he is a M.S. candidate of Zhengzhou Information Science and Technology Institute. His research interest includes network security, network entity location.