RETURN ORIENTED OBFUSCATION

Vivek Balachandran[1], Sabu Emmanuel[2] and Ng Wee Keong[3]

[1]Singapore Institute of Technology, Singapore
vivek.b@singaporetech.edu.sg
[2]Kuwait University, Kuwait
sabu@cs.ku.edu.kw
[3]Nanyang Technological University, Singapore
wkn@pmail.ntu.edu.sg

ABSTRACT

Software reverse engineering is an active threat against software programs. One of the popular techniques used to make software reverse engineering harder is obfuscation. Among various control flow obfuscations methods proposed in the last decade there is a lack of inter-functional control flow obfuscation techniques. In this paper we propose an inter-functional control flow obfuscation by manipulating return instructions. In our proposed method each function is split into different units, with each unit ending with a return instruction. The linear order in which functions appear in the program is obscured by shuffling these units there by creating an inter-functional control flow obfuscation. Experimental results show that the algorithm performs well against automated reverse engineering attacks.

KEYWORDS

Software protection, Code obfuscation, Reverse engineering

1. INTRODUCTION

To develop high quality software, engineers use various software analysing tools to detect vulnerabilities and loopholes in the program thereby facilitating them with an environment to improve their software. However, software analysing tools are double-edged swords that can be used to reverse engineer the software for malicious intents like intellectual property theft or finding vulnerabilities to exploit. Tools and books on reverse engineering are readily available for download on various Internet websites [1, 2].

A major factor that makes it harder to prevent software reverse engineering is that the attacker is a user and has all the power of a user to control the software and its running environment. One of the ways to provide some security to the distributed program is to incorporate a security mechanism embedded within the program. Software obfuscation is one such effective mechanism that hinders the process of software reverse engineering. Obfuscation is the process of translating a software into a semantically equivalent obscure form, so that it is harder to understand the logic of the program. Obfuscation can be applied to an entire program or partly to a section of the program, like watermarked code [4]. Low performance overhead compared to other techniques like encryption, is one of the desirable properties of obfuscation [5]

Software obfuscation can be applied to a program at different stages of its compilation. Source code obfuscation refers to the application of obfuscation on the source code of the program [6-8]. Similarly binary level obfuscation refers to applying obfuscation algorithms on compiled binary

Natarajan Meghanathan et al. (Eds) : NETCOM, NCS, WiMoNe, GRAPH-HOC, SPM, CSEIT - 2016
pp. 25– 36, 2016. © CS & IT-CSCP 2016 DOI : 10.5121/csit.2016.61503

programs [10]. Obfuscation can be applied on various intermediate levels such as on bytecode representation in the case of Android applications [19] or Java programs.

Most of the modern reverse engineering tools, like IDAPro [1], are capable of constructing the control flow graph of a program by converting a binary program to its equivalent assembly representation.

A control flow graph shows the basic block [16] of instructions as vertices and the possible control flow directions as edges, which enables an attacker to follow the program logic and find possible points to attack. Thwarting the disassembly process, by not allowing the reverse engineering tools to determine the correct program representation will result in an erroneous assembly program generation, thereby making program analysis harder. This is the basic idea of most of the binary obfuscation algorithms. In the past years, many binary obfuscation algorithms have been designed to fool the reverse engineering tools. Signal based obfuscation [11], control flow flattening [18], self-modification based obfuscation [14], double process obfuscation [12], instruction embedding [13], are some of the binary level obfuscation algorithms.

One of the limitations of all these obfuscation techniques is that they are all trying to obfuscate the instructions within a function. So, even though the obfuscation does a good job in obscuring the program, the functions remain intact. A reverse engineering tool will still be able to find the number of functions in the program and will be able to differentiate the instructions of one function from the other.

In this paper we discuss an obfuscation technique where, we shuffle code fragments from different functions disturbing the linear order of functions in the program. The reverse engineering tool will identify more functions than the original program and each function will be a small code fragment of the original function.

The paper is organized as follows. The proposed algorithm is explained in section 2. Section 3 discusses about the implementation details of the obfuscation method. In section 4, we analyses the overhead created by our obfuscation on the program performance. Performance evaluation of our obfuscated algorithm is discussed in section 5. The paper concludes with section 6.

2. PROPOSED METHOD

In this section, our new obfuscation method against software reverse engineering is discussed. Our obfuscation algorithm takes an assembly program as input and split the functions in the program and shuffles them, while maintaining the semantics of the program. The assembly representation generated by any assembler maintains a functional structure of the program i.e., the functions in the program are spatially arranged one after another. Each function starts with the standard set of instructions, to set the stack, and ends with a return instruction(s). When a reverse engineering tool disassembles a binary program to an assembly representation, it is thus capable of identifying functions and could segregate them into different functional units. This can help the reverse engineer, better analyse the program or creating a function call graph.

The basic idea of our technique is to disturb this normal representation of the program. In our technique, a function is split into various code fragments by inserting return instruction at the end of each code fragment. Each of these code fragments are then shuffled between functions, giving a inter-functional mix as shown in Fig. 1. One of the advantages of this method is that the linear arrangement of functions (one after another) is obscured. One of the challenges in implementing such a technique is to maintain the semantics of the program. In a normal program, a return instruction is used to return the control flow from a *callee* function to a *caller* function. Adding

new return instructions could thus affect the behaviour of a function. In this section, we explain in detail about inserting the return instructions into functions while maintaining the semantics of the program.

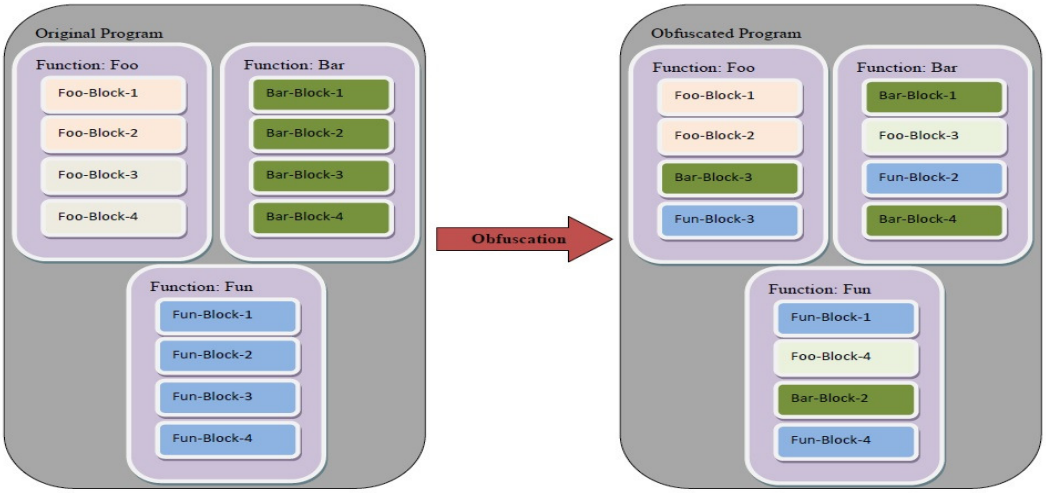


Figure 1 Overview of the algorithm

2.1. Splitting the function

The first step of our algorithm is splitting the function into different segments. The input assembly program is scanned for finding all the functions in the program. Once the functions are identified, each function is split into different code fragments. The obfuscator has the option to specify the number of splits in the function. In the default mode the obfuscator splits each function into four code fragments. While splitting the function into code fragments, our implementation put a constraint that the code fragment should contain at least five instructions.

For each function, the line numbers at which the function has to be split are identified and a randomly generated unique label is inserted. The unique label refers to the entry point of a code fragment. Fig. 2, shows the insertion of the labels to split the function into different segments.
In the example shown, two labels are inserted at the beginning of the code fragments.

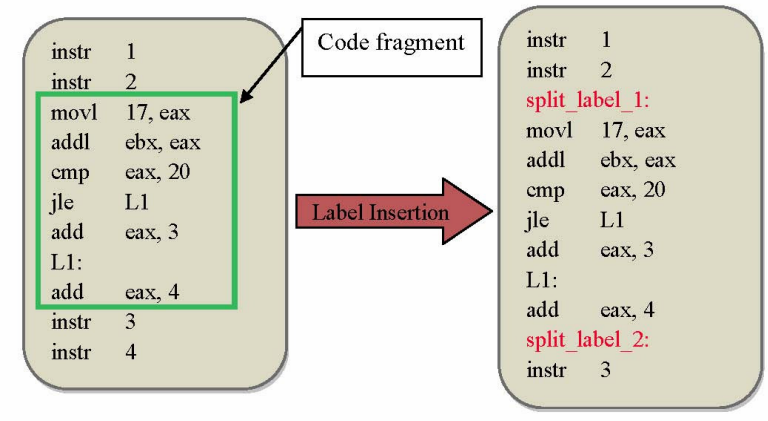


Figure 2 Inserting labels at the splits

2.2. Pushing the return address to the stack

We insert a *ret* (return) instruction at the end of each of the code fragment. This will make a disassembler think that each code fragment is a separate function. The *ret* instruction takes the return address from the stack and transfers the execution control to the particular address location. Thus, to maintain semantics, the current return address should be saved in the stack for later use and the address location of the next code fragment should be pushed into the stack as the new return address.

In our obfuscation algorithm, at the end of each code fragment two assembly instructions are added before inserting the *ret* instruction. One instruction stores the original return address in *ebp + 4* to a register that has not been used. It is followed by another instruction which stores the address of the next code fragment to the stack location *ebp + 4*.

In the example shown in Fig. 2, register edx is used to store the current return address in the stack. The address location *split_label_2* is then stored in the stack location *ebp + 4*. These two instructions can be stored anywhere between *split_label_1* and *split_label_2* and not necessarily at the end of the code fragment.

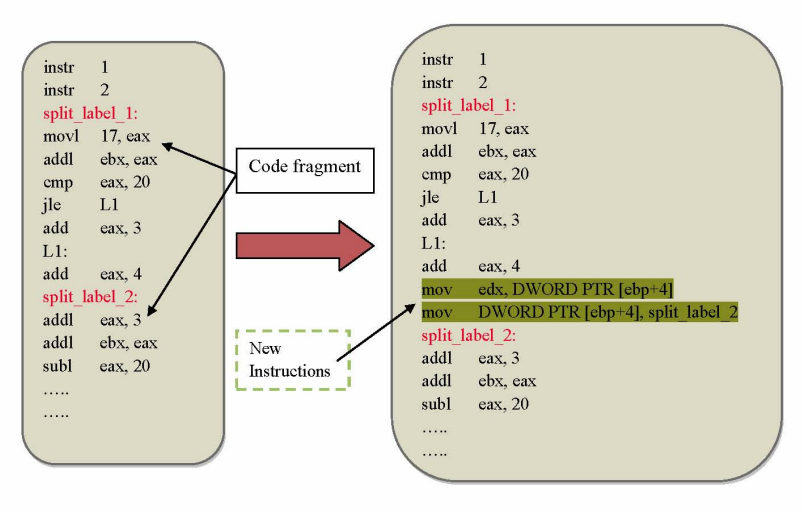


Figure 3 Inserting return address in the stack

2.3. Inserting return instruction

The next step in our technique is to insert the ret instructions in each of the code fragments. Like a standard return instruction the stack pointer and base pointer are reset using the two instructions, *move sp, ebp* and *pop ebp* which is then followed by the ret instruction. We add an extra instruction to store the stack pointer value to a free register. In the example shown in Fig. 4, the instruction is *mov ecx, esp*, is used to store the value of stack pointer to the register *ecx*.

We add an extra instruction to store the stack pointer value to a free register. In the example shown in Fig. 4, the instruction is *mov ecx, esp*, is used to store the value of stack pointer to the register *ecx*. The reason for this instruction is that we cannot reset the stack pointer value as the function is not completely returning to its caller function and it is needed in the following code fragments that will get executed.

With the address location *split_label_2* in stack and return address, the control, flows from the first code fragment to *split_label_2*, the beginning of the second code fragment when the *ret* instruction gets executed.

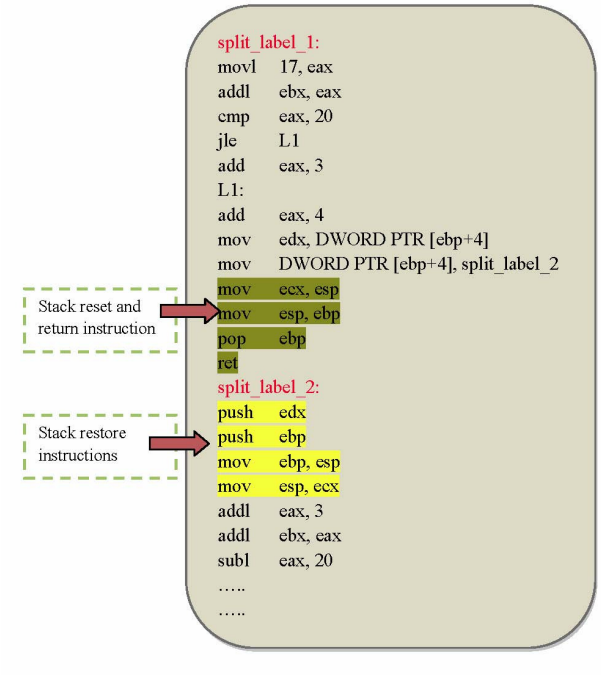


Figure 4 Inserting return instruction

2.4. Restoring the stack

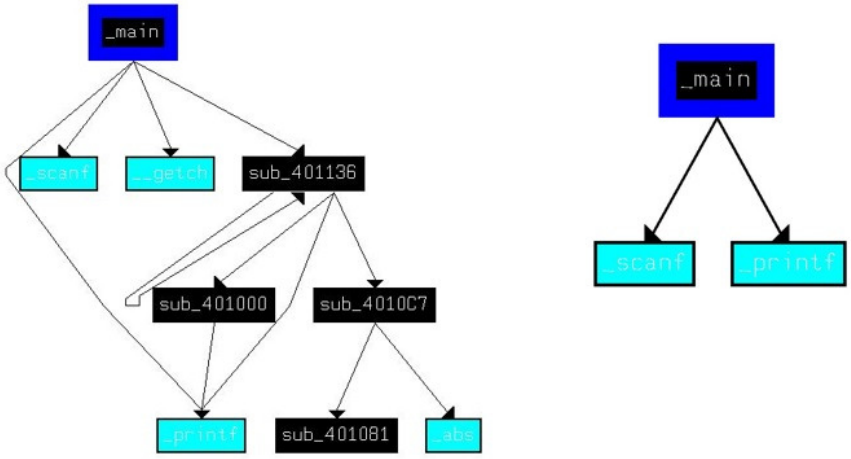
During the execution, after *ret* instruction from one code fragment is executed; the program execution control reaches the next code fragment. Since the stack pointer was reset during the return instruction, the stack has to be restored at the beginning of the new code fragment.

The first address that has to be restored in the stack is the original return address, which is stored in the register *edx*. By pushing the register *edx,* we can restore the original return address in the stack. Instructions *push ebp*, and *mov ebp, esp* restores the *ebp* register. The original stack pointer value is stored in ecx register as shown in the example in Fig. 4. Instruction *mov esp, ecx*, restores the original stack pointer value.

2.5. Shuffling the code fragments

The obfuscation algorithm treats each code fragment as a separate function unit and shuffles them randomly. The linear order of the function representation is disturbed and code fragments from different functions will be interleaved together. This helps in inter-functional control flow obfuscation.

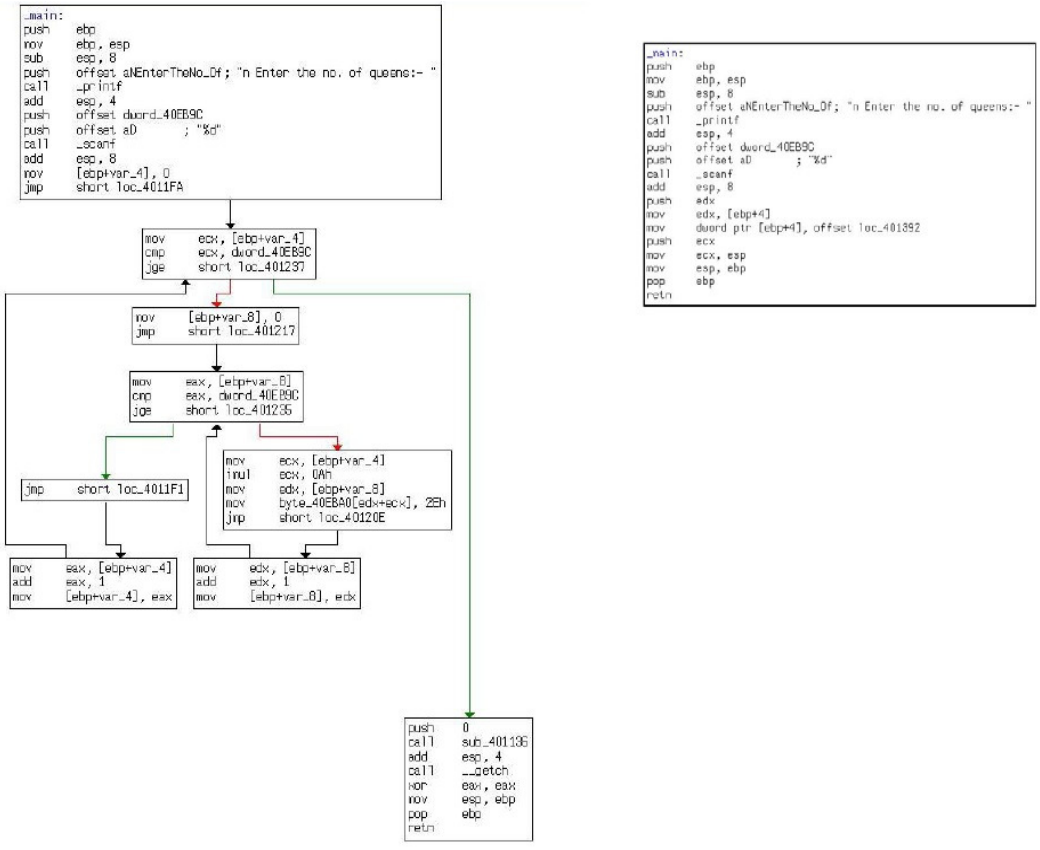
Fig. 5 shows the function call graph of *nqueens* program generated by IDAPro, before and after obfuscation. The obfuscation has clearly confused the IDAPro that it is unable to generate the function calls from *main* after obfuscation. Fig. 6 shows the disassembled *main* function of the program before and after obfuscation. It is clear from the figure that the control flow of the function is completely obscured by the obfuscation.



Before Obfuscation

After Obfuscation

Figure 5 Function call graph of nqueens program



Before Obfuscation

After Obfuscation

Figure 6 Main function of nqueens program

3. IMPLEMENTATION

The proposed obfuscation method is implemented in python programming language. Our implementation expects assembly level representation of the program to be obfuscated. We have implemented the obfuscation algorithm for Microsoft Windows XP and Ubuntu Linux 13.04 operating systems. Our implementation accepts Microsoft Visual Studio 10.0 generated assembly programs and assembly program generated by gcc 4.7.3 as input for obfuscation. Our algorithm generates the obfuscated assembly program which is assembled using the corresponding assembly program to generate obfuscated binary program.

The python code copies the input assembly program to a buffer. It analyses the buffer to find the functions and the start and end of the functions. The number of code fragments for each function is calculated according to the size of the function. The instructions to modify the stack for return address and restoring the stack pointer are added to the beginning and end of each code fragments. The line numbers of beginning and end of each code fragments are changed due to the insertion of instructions. The code fragments are given numbers in sequential order and are represented in a data structure with the number, starting line in the buffer and ending line in the buffer. A simple shuffling algorithm is used to shuffle the code fragments as shown in the following psuedocode. The code fragments are then stored into a new file in the shuffled order to generate the obfuscated assembly file.

```
Shuffle_code_fragments (Code_Fragment_list [])
L = Length (Code_Fragment_list)
        while  (L > 1)
                R = Random (1, L-1)
                Temp = Code_Fragment_List [R]
                Code_Fragment_List [R] = Code_Fragment_List [L]
                Code_Fragment_List [L] = Temp
                L = L -1
Return Code_Fragment_List
```

4. PERFORMANCE EVALUATION

In this section we perform experimental evaluation of our algorithm against reverse engineering. We measure the efficacy of our algorithm by measuring the potency against IDAPro [1]. Instruction disassembly error which calculates the number of instructions that the reverse engineering tool is unable to disassemble properly gives the potency of the obfuscation algorithm. In this section, we also analyse the space and time overhead caused by the obfuscation. The increase in space and time at different levels of obfuscation is analysed. We used the test programs from the lcc 4.2 [17] compiler source as input test programs for our obfuscation algorithm.

4.1. Instruction disassembly error

The potency of the obfuscation algorithm against reverse engineering tool, is measured by the error in the disassembly of assembly instructions. IDAPro [1] was used to disassemble the obfuscated test programs. We measured the total number of instructions in the original program and the instructions recognized by IDAPro [1] after reverse engineering the obfuscated program. Confusion factor is then calculated as the ratio of their differences, as defined by the following equation,

$$CF_{instr} = |T_{total} - T_{disasm}| / T_{total}$$

The total number of instruction addresses before obfuscation is represented by T_{total}. The number of instruction addresses recognized by IDAPro [1] after disassembling the obfuscated binary program is represented by T_{disasm}.

Table 1 shows the confusion factor while disassembling obfuscated test pro-grams by IDAPro [1]. The table shows varies levels of splitting the program. The first column represented by zero splits is the original program and all the instructions are reverse engineered successfully. Column 2 represents the obfuscated program, where every function is split into two and the mean disassembly error is 55.9% when functions are split into two. The instruction disassembly error increases as the splitting of the program increases. The splitting of a program saturates after a while. For instance, the program fields has the same instruction disassembly error for 16 splits and 32 splits. This is because the program is split to the maximum possible split by 16 splits and the program cannot be further split down.

Mean instruction disassembly error of 85.16% is obtained at level 8, where each function is split into 8 code fragments.

Table 1. Instruction Disassembly Error

Splits / Prog	0	2	4	8	16	32	64	128
8q	283	125	65	42	25	25	25	25
array	341	150	78	51	31	20	20	20
cf	184	81	42	28	17	11	11	11
cq	13786	6066	2068	2068	2068	2068	2068	2068
cvt	674	297	155	101	61	40	27	14
fields	339	149	78	51	20	20	20	20
incr	392	172	90	24	24	24	24	24
init	415	183	95	62	37	17	17	17
limits	162	71	37	24	12	12	12	12
sort	506	223	116	76	46	46	46	46
spill	433	191	100	65	39	26	26	26
struct	505	222	116	76	45	36	30	30
wf1	597	263	137	90	54	24	24	24
CF_{instr}		55.9%	82.9%	85.1%	86.6%	87.2%	87.3%	87.4%

4.2. Space overhead

The insertions of the new instructions have significant effect on the size of the program. If a function is split into 2 code fragments, then 10 new instructions are added into the program and 20 instructions are added if the function is split into 3 code fragments and so on. Let the number of instructions in the program be N_{before} and the original program is split into $n+1$ code fragments. The total number of instructions in the obfuscated program will be,

$$N_{after} = N_{before} + 10n$$

In the worst case, the entire program is split into code fragments with 5 instructions. Let the program is split into *n+1* code fragments, with each code fragment having four instructions, then the total number of instructions in the program before obfuscation is,

$$N_{before} = 5(n + 1)$$

After the obfuscation, 10 instructions are added per code fragment and the total number of instructions in the program after obfuscation is,

$$N_{after} = 5(n + 1) + 10n$$
$$N_{after} = 3N_{before} - 10$$

So, in the worst case there are three times more instructions in the obfuscated program than the original program. We can see in the experimental evaluation that this upper bound is held. $Space_{ovh}$ defines the increase in the size of the program.

$$Space_{ovh} = Space_{after} / Space_{before}$$

In Table 2, we show how the program size increases as the program is obfuscated. The size of the program increases as the number of splits increase. In the worst case, the size increases to 2.2 times the original size. But on an average, the program size increases by 1.57 times the original size with 128 splits, which is less than the theoretical upper bound.

Table 2 Space Overhead

Splits / **Prog**	0	2	4	8	16	32	64	128
8q	8743	8779	10256	13586	17843	17843	17843	17843
array	8678	8714	11324	14321	18545	18923	18923	18923
cf	8750	8786	9957	12985	13876	14178	14178	14178
cq	63599	75898	85699	85699	85699	85699	85699	85699
cvt	12983	13019	13091	13235	13523	14099	15251	21651
fields	8728	8764	12633	13589	13589	13589	13589	13589
incr	8594	8630	9987	12371	12371	12371	12371	12371
init	9134	9170	9242	9386	9674	9782	9782	9782
limits	8522	8558	8630	8774	12062	13458	13458	13458
sort	8856	8882	11954	12098	14350	14350	14350	14350
spill	8863	8899	10971	12115	16403	18799	18799	18799
struct	8843	8879	8951	9095	9383	14055	17127	17127
Wf1	13179	13215	13287	17521	18754	21875	21875	21875
Mean	177462	190193	215982	234775	256072	269021	273245	279645
$Space_{ovh}$	1	1.07	1.21	1.32	1.44	1.51	1.53	1.57

4.2. Time overhead

Obfuscation does have an effect on the execution time. The execution time of the program will increase, because of the execution of the additional instructions. We know that the size of the program increases by 3 times in the worst case. The input parameter of the time complexity thus increases by 3 times. If the original time complexity was *T(n),* then the new time complexity will be *3T(n)* in the worst case. The time complexity of the obfuscated program is thus between *T(n)* and *3T(n).*

$Time_{before}$ refers to the time taken by the program to execute without obfuscation and $Time_{after}$ is the time taken by the obfuscated program to execute. We evaluate the effect of obfuscation on execution speed with $Time_{ovh}$ defined as,

$$Time_{ovh} = Time_{after} / Time_{before}$$

Table 3, shows the time overhead caused by our obfuscation on various binary programs. In the worst case the time overhead is 2.36 times in the case of *cvt* with 128 levels of obfuscation. On an average the worst case time overhead is 1.86 which is lower than the upper bound of *3T(N).*

Table 3 Time Overhead

Splits / **Prog**	**0**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
8q	1300	1395	1534	1975	2589	2589	2589	2589
cf	2777	3276	3588	4215	5014	5581	5581	5581
cvt	1077	1121	1154	1178	1223	1498	1892	2548
fields	1089	1258	1685	1987	2406	2406	2406	2406
incr	1271	1301	1354	1537	1537	1537	1537	1537
spill	1174	1256	1325	1456	1658	2219	2219	2219
struct	1342	1398	1469	1566	1689	1726	1754	1754
wf1	1245	1322	1391	1499	1785	1997	2015	2015
array	1011	1250	1347	1678	2001	2022	2500	2694
cq	1025	1119	1243	1567	1874	2198	2198	2198
init	1198	1245	1376	1461	1653	1985	1985	1985
sort	1037	1134	1256	1370	1523	1523	1523	1523
Mean	15546	17075	18722	21489	24952	27281	28199	29049
$Space_{ovh}$	1	1.09	1.20	1.38	1.60	1.75	1.84	1.86

5. CONCLUSIONS

In this paper we proposed an obfuscation algorithm to perform inter functional obfuscation. Our method slices each function in the program into separate code fragments. Each fragment ends with a return instruction and starts with stack allocation instructions, thereby appearing itself like a function. The return instruction transfers the control flow to the next code fragment instead of returning to a caller function. The code fragments are shuffled disturbing the linear order of the

functions. Unlike other control flow obfuscation, our method adds more control flow instructions (return instruction) to increase the control flow obscurity instead of removing the control flow instructions. The experimental results show that obfuscating with 8 splits provides a good obfuscation without too much overhead on the space and time requirements of the program. Experimental analysis shows that that our method has an instruction disassembly error of 85.1 % with 8 levels of splitting. An average time overhead of 1.38 and space overhead of 1.32 are observed while obfuscating with 8 splits.

REFERENCES

[1] "Data Rescue," Available at: http://www.hex-rays.com/ [Last accessed: October 14, 2016]

[2] J. Miecznikowski and L. Hendren, "Decompiling java using staged encapsulation," in Reverse Engineering, 2001, pp. 368-374.

[3] W. Thompson, A. Yasinsac, and J. McDonald, "Semantic encryption transformation scheme," in International Workshop on Security in Parallel and Distributed Systems, San Francisco, CA, 2004.

[4] J. Hamilton and S. Danicic, A survey of static software watermarking, in World Congress on Internet Security, pp.100-107, 2011.Lecture Notes in Computer Science: Authors' Instructions 13

[5] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach, in ACM Workshop on Quality of Protection, ACM New York, NY, USA, 2007, pp. 15-20.

[6] W. F. Zhu, "Concepts and techniques in software watermarking and obfuscation," Ph.D. Thesis, University of Auckland, 2007.

[7] M. Sosonkin, G. Naumovich, and N. Memon, "Obfuscation of design intent in object oriented applications," in ACM workshop on Digital Rights Management, 2003, pp.142-153.

[8] C. Collberg and J. Nagra. "Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection (1st ed.)," Addison-Wesley Professional, 2009.

[9] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in ACM Symposium on Principles of Programming Languages, 1998, vol. 25, pp. 184-196.

[10] V. Balachandran and S. Emmanuel,Software Protection with Obfuscation and Encryption, Information Security Practices and Experiences Conference, ISPEC, volume 7863 of Lecture Notes in Computer Science, pp 309-320. Springer, 2013.

[11] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in USENIX Security Symposium, 2007, pp. 1-16.

[12] J. Ge and S. Chaudhari, "Control flow based obfuscation," in ACM Digital Rights Management, 2005.

[13] C. LeDoux, M. Sharkey, B. Primeaux, and C. Miles "Instruction Embedding for Improved Obfuscation," in Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12),pp.130-135, 2012.

[14] V. Balachandran and S. Emmanuel, "Software code obfuscation by hiding control flow information in stack," in IEEE Workshop on Information Forensics and Security, 2011.

[15] V. Balachandran and S. Emmanuel,"Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code," in IEEE Transactions on Information Forensics and Security, vol.8, no.4, pp.669-681, April 2013.

[16] "Basic Block," Available at: http://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html [Last accessed: October 14, 2016]

[17] C. W. Fraser, "lcc, A Retargetable Compiler for ANSI C," Available at: https://sites.google.com/site/lccretargetablecompiler/downloads [Last accessed: October 14, 2016]

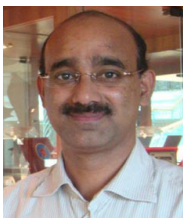
[18] L. Shan and S. Emmanuel, "Mobile agent protection with self-modifying code," in Journal of Signal Processing Systems, vol. 65, pp. 105-116, 2010.

[19] V. Balachandran, Sufatrio, D.J.J. Tan, and V.L.L. Thing. "Control Flow Obfuscation for Android Applications." in Computers and Security, vol.61,pp.72-93, Aug 2016.

AUTHORS

Dr. Vivek Balachandran holds a PhD in Computer Engineering from the Nanyang Technological University, Singapore, where he worked with the Center for Strategic Infocomm Technologies and Temasek Laboratories. After graduating he worked as a Research Scientist with Institute for Infocomm Research, A*STAR, Singapore, where he worked on mobile security and forensics. He has published numerous articles on software security. He is currently a Lecturer at the Singapore Institute of Technology, Singapore. His research interests includes software security, mobile forensics, program analysis, and compiler optimization.

Dr. Wee Keong Ng is Associate Chair (Research) of the School of Computer Science & Engineering, Nanyang Technological University, Singapore. He received his Ph.D. from the University of Michigan at Ann Arbor, USA. His research areas are secure data analytics, secure data storage, data monetization, and data security, and has published more than 200 technical papers in these areas. Dr. Ng has served in program/organizing committees of international conferences. In recent years, he is General Co-chair of the International Conference on Information and Communications Security 2016; Jury Member of the Second Dutch Cyber Security Research Award in March 2016; Senior PC Member of the 20th, 19th, 18th, 17th Pacific-Asia Conference on Knowledge Discovery and Data Mining. Dr. Ng has advised and graduated more than 20 Ph.D. students and 20 Master students



Dr. Sabu Emmanuel received the B.E. degree in electronics and communication engineering from Regional Engineering College, Durgapur, India, in 1988, the M.E. degree in electrical communication engineering from the Indian Institute of Science, Bangalore, in 1998, and the Ph.D. degree in computer science from the National University of Singapore in 2002. He is currently an associate professor with Kuwait University. Previously, he was an assistant professor in the School of Computer Engineering, Nanyang Technological University (NTU), Singapore. He began his career as a research and development engineer and then moved on to the academic profession. He has taught engineering students of Mangalore University, National University of Singapore, and NTU. His current research interests are in multimedia and software security and surveillance media processing. He has been a reviewer for ACM Multimedia Systems. He is a member of the Technical Program Committee of several conferences. Dr. Emmanuel has been a reviewer for IEEE Transactions on Circuits and Systems for Video Technology, IEEE Transactions on Multimedia, and IEEE Transactions on Information Forensics and Security.