ISSN 23499842(Online), Volume 1, Special Issue 2(ICITET 15), March 2015
International Journal of Innovative Trends and Emerging Technologies

CODE SECURITY BY CONFUSING LOGIC FLOW USING SELF MODIFYING CODE

[1]Mr.R.MANIKANDAN, [2]Mr. P.ASHOK KUMAR, [3]Mr.R.GOVINDARAJU,
[1] Research Scholar, [2] Lecturer,[3] Lecturer
[1] St.Peter's University, Department of CSE, Chennai.
[2,3]University College of Engineering Tindivanam, Department of CSE
Tamilnadu, India

Abstract **-**Reverse engineering process can regain the software program code which can be used either for good or bad. Various obfuscation techniques can be used by the developer to make reverse engineering process harder or worthless. Our algorithm which focuses on effectiveness along with secrecy of program can be used to obfuscate logical flow of software programs. It uses self-modifying code which modifies itself during execution and also it removes control flow information from the code area and hides them in the data area. In order to preserve the semantics of the program obfuscated instructions are reconstructed while executing. Intruder finds difficult to differentiate the obfuscated program from normal binary program which shows the stealthy of program.

Index Terms—Computer security, software engineering, soft- ware safety, software security.

I. INTRODUCTION

Software, over the years, has evolved from free code given along with the hardware to a valuable asset, automating almost all of the electronic equipment's and systems. The growth in the software analyzing tools has helped the software developers to analyze and better their software programs. Unfortunately, the same software analyzing technologies [1], [2] are used to reverse engineer software systems with malicious intent such as stealing the intellectual property of the program and for identifying the vulnerabilities in a program and exploiting them. Tools and documents on software reverse engineering are readily available in various websites [1],[2]. There have been several cases of software law suits involving intellectual property theft employing reverse engineering techniques. In 1992, Atari Games v.Nintendo [3]; in 2000, Sony v.Connectix [4] and in 2002, Blizzard v. bnetd are some law suits involving reverse engineering of software programs. Blizzard [5] entertainment's online multiplayer gaming service called Battle.net was reverse engineered into the software package bnetd. Blizzard won the United States lawsuit against bnetd's Original developers [6].

In [7], a self-modifying code based algorithm is proposed.In this method the control flow instructions like *jmp* are camouflaged with normal instructions like *mov* instruction. The opcode of the *jmp* instructions are changed and the target addresses are stored in the destination field of the move instruction. Modifying instructions to change the opcode back to the opcode of *jmp* instruction are added at the beginning of the program. A problem with this method is that, even though the control flow instructions are camouflaged with other instructions, the control flow information, that is the target address, is available in the pro- gram code sections, in the modifying instructions. So, this may be revealed during disassembly of the code section to an adversary. As explained earlier, our method handles this problem by moving the control flow information completely from code section to data area.

Control flow flattening is control obfuscation method

[8] to confuse the disassembler about the execution sequence of the procedure. The idea is that, all the basic blocks will be assigned with the same predecessor and successor block. Once a block is executed, the control flows to the successor block and then to the predecessor and eventually to the exact block from the predecessor block. One of the advantages of control flow flattening is that it provides very good control obfuscation. On the other hand, the performance overhead in terms of space and timeis high for this method. Instruction disassembly error is also less for control flow flattening, i.e., using an automated disassembly tool an adversary can disassemble a majority of instructions from the binary program.

In this paper we propose an algorithm to perform binary level obfuscation, which has good control flow and instruction obfuscation. In most methods performing binary level obfuscation, they introduce a new module to the program to support their obfuscations. In other methods like [7], [8], which do not use extra modules for obfuscation, the control flow information is available in the code area which can be seen during disassembly using tools like IDAPro[1]. So, this motivated us to develop an algorithm that blends the instructions to support obfuscation along with the original program, instead of having an extra module. Also, an algorithm that will not expose the control flow information when

disassembled using an auto- mated disassembly tool.

The basic idea used in our method is to camouflage control flow instructions, like jump instructions and storing their details, needed to reconstruct them in the data area. The target address information is thus in the data area and not in code area like in [7]. During runtime these instructions get re- constructed by the self-modifying code inserted during the obfuscation time. One advantage of this method is that the control flow information is stripped from the code section and an adversary will not be able to find the control flow information by just analyzing the code area. It is also not trivial to find the control flow information by analyzing the data area as they are defined and initialized similar to the ordinary variables.

Hence, the major contribution of our paper compared to other algorithm is that the target address location information is stripped from the code area and is stored in the data area. An adversary cannot reconstruct the control flow by analyzing just the code area. Another contribution of our paper is the introduction of junk bytes in the execution path. This facilitates the obfuscation of conditional jump instructions and adds more confusion to the adversary. Another advantage of our method is that extra modules are not added to the program so as to facilitate dynamic mutation. The self-modifying instructions are inserted within the program procedures. Thus our method does not have the overhead of protecting the additional modules.

The paper is organized as follows. Section II provides preliminaries necessary for understanding the proposed algorithm. Threat model assumption of the attack is discussed in Section III. Section IV covers the proposed algorithm in detail.

The implementation details are discussed in Section V. Section VI is conclusion.

II. PRELIMINARIES

A. Self Modifying Program

Self-modifying program is one which modifies itself while executing. This method is used in different binary obfuscation techniques in different form. The basic idea of this method is that, parts of the programs are removed or replaced by other instructions, thus statically the program looks different. During runtime the program is transformed back to its original form. Different methods are adopted to achieve this as presented in [7]. The basis of all the methods is to add extra code modules to the program which knows exactly which area of the program is to be modified and when to be modified.

The advantage of using self modifying programs is that it obscures the programs really well and makes it difficult for the static disassemblers to correctly disassemble the program. Statically, the program will look completely different and it gets fixed dynamically through self modifying code. Self modifying code can also be used for obfuscating program areas dynamically. So a dynamically restored code can again be obfuscated during runtime. So, the period in which the code is in its true form is during its execution. So, even if an adversary decides to run the program and break at some point and dynamically disassemble the program, his/her chance to get the program in its true form is low.

III. THREAT MODEL

For designing a protection mechanism for software, one should understand the threat faced by the software from the adversary. The assumption we make is that the adversary is trying to reverse engineer a binary program to assembly level representation. One of the factors to be considered in the threat model is the platform and the access level the adversary has. Our assumption is that the adversary owns the software program and pro- gram runs in the adversary's computer. We also assume that the adversary has complete control over the system, where the adversary can analyze the program, modify it and execute it.

Another assumption is that the adversary has access to reverse engineering tools that will help in disassembling the binary pro- gram to assembly representation. We assume that the adversary has access to disassembly tools like IDAPro[1] and ald. Our protection mechanism uses self modifying code, which mutates the program during runtime. We assume that the adversary has access to this information and uses dynamic analysis to disassemble.

IV. PROPOSED METHOD

A program consists of code area and data area. Different data areas are global, local and dynamic. Stack is an example for local data area and heap for dynamic. Our method is basically built on the fact that most reverse engineering tools and methods consider data area and code area separately. Reverse engineers and reverse engineering tools try to extract programming information from the code segments of the software and extracts data values and information about the data structures from the data segments and symbol tables.

The basic idea of our obfuscation is to hide the code information like jump instructions, in the data area, stack, with other data elements thus obscuring the program code. The process of hiding code information in data area is done at the obfuscation time. The information is stored in stack and hence it looks like ordinary variables defined in the function. It is harder for an adversary to distinguish this from ordinary variables by just analyzing the stack. Removing instructions from the code area or

camouflaging it with other instructions makes the program semantically different. The code information stored in the data area is used to reconstruct the original code at runtime and there by the execution of the program is semantically equivalent. This is achieved by inserting reconstruction instructions just above the original location. This will result in reconstructing the original instruction at runtime. We further explain our algorithm in detail.

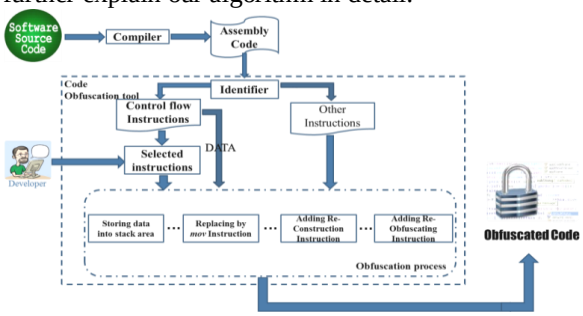


Fig.1 Proposed Architecture

A. Offline Obfuscation

This is the first phase of our obfuscation algorithm. The bi- nary program is converted to its equivalent assembly program using PLTO (Pentium Link Time Optimizer). It is then analyzed to find suitable instructions to be obfuscated. Once the obfuscation is done, the assembly program is assembled back to binary.

1) Selecting Instruction to be obfuscated: The first step of the algorithm is to identify which all instructions have to be camouflaged. The trivial method is randomly picking instructions from the code area. But, in our method jump instructions are chosen to be camouflaged for the following reasons.

Jump instructions decide the control flow of a procedure in the program. By obscuring the jump instructions in the procedure we are thus obfuscating the control flow of the program. Instructions which give information about the control flow of the program will help the adversary to easily understand the logic of the program. Another motivation for considering jump instructions, to be camouflaged, is the scope it provides for inserting junk bytes in the program. Camouflaging jump instructions obscures control flow of the program. This will lead in confusing the disassembly tool to assume wrong control flow to the program and makes it possible to add junk bytes between code blocks which are unreachable. This will increase the errors while an adversary tries to reverse engineer the binary program.

2) Storing Target Address in the Stack: With the instructions to be camouflaged known, the space required in the stack to store the target addresses of camouflaged instruction can also be calculated. In the method proposed, for each instruction in a procedure to be camouflaged, a variable space is allocated in the stack. The counts of instructions in the function which are going to becamouflaged are calculated and then the stack is expanded accordingly.

The expansion of the stack is possible with a small tweak in the assembly program. In the calling convention of the ELF (Extended Linker Format) programs in 86 platforms, the stack allocation for a function is done by the function itself. All the functions start with the following instructions:

pushebp
movebp, esp
sub esp,8

Once the function is called the base pointer of the caller function is pushed onto the stack. Then the current stack pointer is stored as the new base pointer (for the called function). The first two assembly instructions in the code segment are essentially doing that. The third instruction is where the allocation of the stack for the particular function happens .

Fig. 3.Storing code information in stack. The size of the stack needed by the function in this particular case is 8 bytes. By modifying the value in the third instruction, the size of the stack for that particular function can be changed. Once, the instructions that are going to be obfuscated and their count are known, the stack is expanded accordingly as mentioned in the previous paragraph. Since we know that we are moving *jmp* instructions, the target address to which jump happens constitutes the code information. This target address is what we store in the dataarea. Selecting stack area to store the code information has an ad- vantage over global data area. The code information in stack area is stored in a way similar to that of local variable definition. Self modifying instructions use these variables to reconstruct the control flow. The way the variables are used in the program are similar to manipulating ordinary variables loading the value from a variable to a register and analyzing the value. The variables of a function are used only by the instructions of that function.

On the contrary, if global data area was used to store the code information, then the code information will be stored in the global data area. Each local function will use only those variables which are used to store the control flow information of that particular function. A global variable used exclusively by a local function is suspicious and an adversary may easily notice it. Fig. 3 shows how the *jmp* instructions target address is stored in the stack area. The target address *xxxx* of the *jmp* instruction in the first block is stored in a stack variable.

3) Obfuscating the jmp Instructions:

The *jmp* instruction is ready to be obfuscated as the target address of the *jmp* has already been stored in

the stack. The *jmp* instructions are replaced with another instruction instead of removing. The *jmp* instructions are replaced by the following instruction, *moveax, 0*

The replacement of *jmp* instruction with *mov* results in the loss of control flow information. The new instruction, *mov*, is an ordinary instruction and does not have a say in the control flow of the program. When an automated disassembler tries to disassemble the program, it assumes the control flows just to the next address location after *mov*. We decided on the instruction *mov* to be used to replace *jmp* instructions owing to the fact that it is the most used instruction in a program. It is possible to use other instructions instead of *mov* to camouflage the *jmp* instructions. The logic remains thesame. Randomizing the selection of instruction to be used to replace *jmp* instruction will increase the challenge posed by the method to an adversary.

B. Runtime Deobfuscation

Camouflaging the instructions in the program as explained in the previous section changes the semantics of the program. Running this program just like that gives erroneous results and most probably crashes the program. And hence, the program has to be changed back to it soriginal form before it gets executed. In our method we do this dynamically at runtime with the help of self modifying code. Reconstruction instructions which reconstruct *jmp* instruction at runtime are inserted in a block that precedes the *jmp* instruction. The block should be a dominator block, which means it should precedes the *jmp* instruction in all execution paths.

The insertion of reconstruction instructions are shown in Fig. 3.The first step is to change the opcode of *mov* instruction to that of *jmp* instruction. The opcode of *jmp* instruction is 0xE9 and that of *mov* instruction is 0xB8. We insert an instruction to XOR the address location of *mov* instruction with 0x00000051. This changes the instruction to *jmp*offset0. Now the next step is to add the address offset stored in the data area to the instruction. We insert an instruction to addthe value in the local variable to the instruction address. Now the exact *jmp* instruction is created at the address location of *mov* instruction.

In Fig. 3, the camouflaged *jmp* instruction is at address location A1 in basic block B1. The *jmp* instruction is camouflaged into *mov* instruction and the reconstruction instructions are added before the camouflaged instruction.

B. Runtime Reobfuscation

With the reconstruction instructions in place, the program semantics are restored and program works perfectly well. Now, the instructions which are obfuscated are restored and is in its original form. An adversary, who tracks the image of the pro- gram at regular intervals, will be able to find the de-obfuscated instructions. A core dump of the image of the program will give the instructions in its true form if it is done after the reconstruction operations. A method to address this problem is by reobfuscating the instruction at runtime after its execution. This is achieved by adding extra reobfuscation instructions in the succeeding blocks to reobfuscate jmp instruction back to *mov*. Note that, the reobfuscation instruction should be inserted in all the successor blocks as the execution path is chosen dynamically at runtime. Reobfuscation is done by XOR-ing the jmp instruction with 0x00000051 to get the instruction:
Move ax, 0

According to the control flow of the example in Fig. 3, the basic block B3 follows after the execution of the jmp instruction. The reobfuscation instructions for the program are hence added in the beginning of the basic block B3.

Fig. 4, shows how the junk bytes are introduced in the program. The existence of junk bytes corrupts the original code in the program too, since partial junk bytes of an instruction are added.

In case the *jmp* instruction is a part of the loop, then a new basic block is added to the loop edge and the reconstruction instructions are added in that blocks shown in Fig. 4.

Fig.2 Runtime Reobfuscation

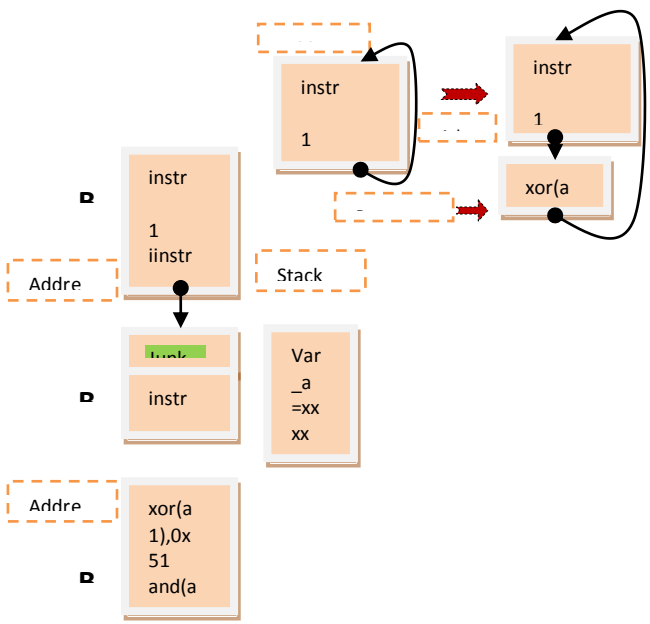


Fig.3. Junk byte

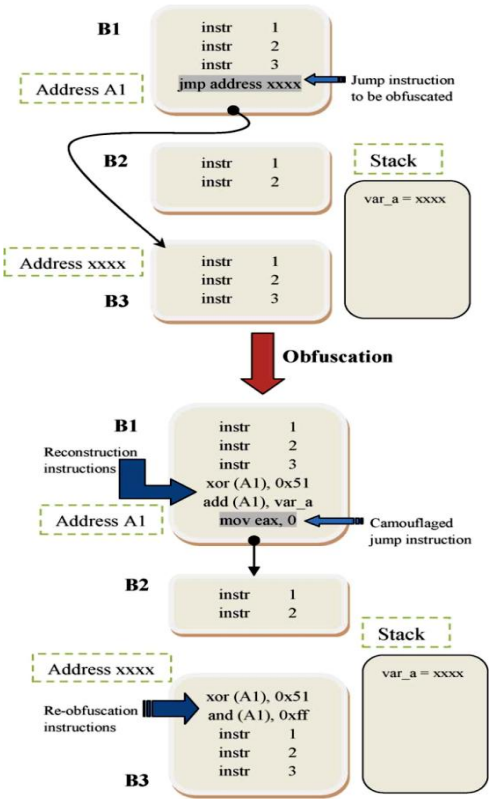


Fig.4. Reobfuscation in loops

E. Conditional Jump Instructions

Conditional jump instructions like, *jle*(jump if less than or equal), *jge* (jump if greater than or equal), *jz*(jump if zero), *jg*
(Jump if greater than) etc., also adds to the control flow of the procedures in a program. Obfuscation of the seinstructions can be done similar to unconditional jump instructions. Conditional jump instructions can be camouflaged using other ordinary instructions and the target address can be stored in the stack.

The problem is that the insertion of junk bytes, which is responsible for confusing the disassembler and increasing the instruction disassembly error can't be done with conditional instructions.

The basic reason for junk byte insertion is difficult with conditional instruction is that the instruction followed by the conditional jump instruction is a valid instruction point. Inserting junk bytes at that point will corrupt the program. To take care of this condition, our method deals with conditional jumps in a different manner, so as to get better obfuscation. In this method a junk byte is added just above the conditional jump instruction. This junk byte should be a partial byte of an instruction as explained in . This junk byte will club with the initial bytes of the conditional jump instruction, resulting in corrupting the jump instruction and few instructions after that.

In the example shown in Fig. 6, *10h* is the junk byteaddedabovethe jump instruction and the instruction *adc [esi], bh* will be seen when the program is disassembled.

The semantics of the program will be changed by this insertion of the junk byte and that is handled by self modifying code. Reconstruction instructions are added just like in the case of unconditional jump. But in this case, the reconstruction instructions are used to convert the junk byte into *nop* instruction no operation instruction. Thus the semantics of the pro- gram remains the same during runtime.

The *or*instruction in B1 of Fig. 6, converts the junk byte 10 to *0x90*, the opcode of *nop* instruction. Similar to the case of unconditional instructions, reobfuscation instructions are added in all the successor blocks. In this case, the reobfuscation instructions obfuscate the *nop* instruction back to the junk byte. The *and* instructions in B2 and B3 of Fig. 7, converts *0x90*, the opcode of *nop* instruction, to *0x10*.

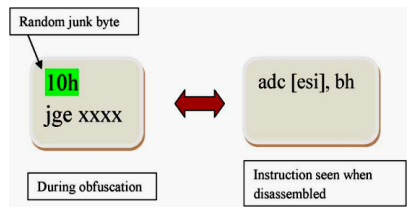


Fig. 5. Junk byte addition to obfuscate conditional jumps.

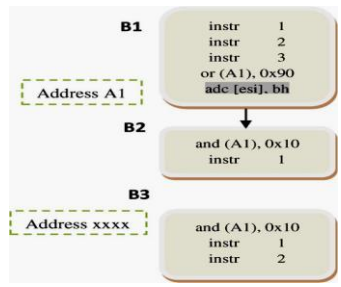


Fig. 6.Obfuscation of conditional jumpinstructions.

F. Indirect Jump Instructions

Indirect jump instructions also add to the control flow of a program. In an indirect jump instruction the address location to which the control flow transfer happens is stored in a register or a memory location. For example, *jmp eax* is an indirect jump instruction, where the control flow is transferred to the address stored in the register *eax*, as shown in Fig. 7.Obfuscation of the indirect jump instructions can be

done atCompile time by camouflaging the indirect jump instruction with normal instructions. The camouflaged instructions can be reconstructed, by adding reconstruction instructions above the camouflaged instruction. However, we have not considered indirect.

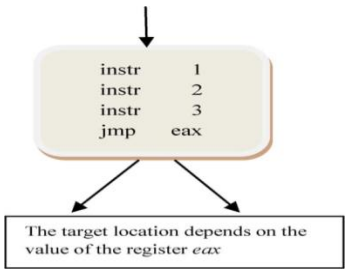
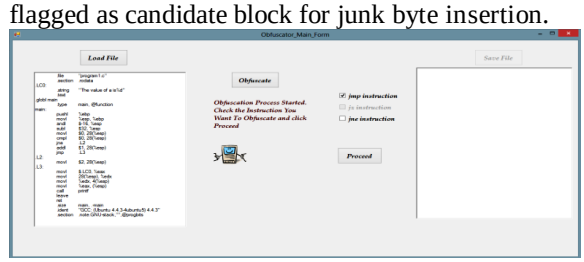


Fig. 7. Control flow of indirect jump.

Jump instruction for obfuscation as it is difficult to reobfuscate the indirect jump instructions during runtime. In the proposed algorithm, during runtime the reconstructed jump instructions are reobfuscated after the jump. This is done by adding reobfuscation instructions in the successor blocks. For indirect jumps the target locations of the jump depends on the value residing in the register or memory location used in the indirect jump instruction and hence can change dynamically. So, if we obfuscate the indirect jump instructions by camouflaging it at compile time and reconstructing it during runtime, it will be a onetime obfuscation, as the reconstructed instruction cannot be reobfuscated. Another problem is when indirect jump instruction jumps back creating a loop. In this case the reconstruction instructions used to convert the camouflaged instruction to jump instruction get executed again. So, the reconstruction instructions should be chosen in such a way that the indirect jump instruction is not affected when they get executed more than once. This will limit the instructions that can be used as reconstruction instructions.

V.IMPLEMENTATION

The proposed obfuscation is carried out at link time of the compilation process. The implementation expects a binary pro- gram as input, which is obfuscated and gives out an obfuscated binary program as output. The development platforms used is GNU Linux operating system and the input binary files are expected in the extended linker format (ELF). For the implementation of our algorithm at link time, PLTO, Pentium Link Time Optimizer was used. The input binary program is fed to PLTO which creates the control flow graph of the program. The control flowgraph thus generated is scanned to find possible candidate instructions to be obfuscated. Each function of the program is scanned block by block to find the unconditional jump instructions. Once the count of the jump instructions that are going to be obfuscated is finalized then the size of the stack is expanded. The local variables of each function are stored in the stack. The activation record for each function will be of constant size de- fined in the beginning of a function. It has the space required for storing local variables, parameters and return value. Every time a function is called, this constant space in stack is allotted for the function. Since our method stores the code information as variables in the stack, this stack size has to be expanded. The code in the function which defines the required stack size is modified according to the requirement. With this modification, when the function is called it pushes the stack pointer further and thus incorporating the space for the new local variables used to store the control flow information. For each function in the program, obfuscation is done in three rounds. In the first round all the unconditional jumps are handled. Junk byte insertions at locations after unconditional jumps are done in the second round. Conditional jumps are handled in the third round of the algorithm. The process repeats for all the functions. The exact sequence of implementation in the first round isas follows. The target address of each unconditional jump instruction is extracted from the instruction and is stored in the local variable. The jmp instruction is then replaced with *mov* instruction. The basic blocks in which the reconstruction instructions and reobfuscation instructions have to be inserted are calculated. Reconstruction instructions and reobfuscation, which use the variable where the address is stored, are inserted in the respective basic blocks. The successor block of the jmp instruction is flagged as candidate block for junk byte insertion.



In the second round, all the basic blocks which are flagged as candidate blocks for junk byte insertion are visited and from the set of junk bytes, which are partial instructions, randomly chosen junk byte is added to the beginning of the basic block.

The third round in the implementation is similar to the first round. Each basic block with unconditional jump instructions are visited. The junk byte to be inserted is randomly chosen and is stored in the variable in the stack to be used for reconstruction and reobfuscation instructions. The junk byte is then inserted just above the unconditional jump instruction. The basic blocks in which the reconstruction instructions and

reobfuscation instructions have to be inserted are calculated. Instructions which convert the junk bytes to *nop* instructions are inserted in the basic block for reconstruction instructions. The instructions for converting *nop* back to the junk byte are inserted in the basic blocks for reobfuscation instructions.

The obfuscated program should have right permissions for the reconstruction and reobfuscation instructions to modify the program code area. We introduce system calls in the program so that the write permissions can be given when it is necessary. The *sys_mprotect* system call is called at the beginning of a function, with flags to enable write permission to the necessary program code area. The write permissions are disabled by calling the *sys_mprotect* system call at the end of the procedure.

Enabling right permissions to the entire code area for self modification may lead to the risk of code injection attacks. Hence, we use *sys_mprotect* system call in the program to enable write permissions to address locations that are needed to be modified. But just enabling write permissions to the address locations to be modified will give away the information to the adversary about the areas ofself modifications. So, a tradeoff has to be made between giving write permissions to the entire code area and exact address locations, giving the adversary the information regarding the self-modifying addresses. Thus, in our current implementation, as a compromise, the *sys_mprotect* system calls are added at the beginning of a function and at the exit blocks of a function. When a function call is made, the *sys_mprotect* system call gets executed and enables write permission to the function code area, thereby enabling write permissions to reconstruction instructions. The write permissions are again disabled by *sys_mprotect* system call at the exit point of the function. This makes sure that the write permissions are activated only when a function is being executed. Just before the function returns, the write permissions of the function code area are disabled. The whole program, which is in the intermediate control flow Representation in the PLTO framework is then recompiled to binary executable.

VII.CONCLUSION

In this paper we proposed software obfuscation algorithm to increase the complexity while doing reverse engineer of software program. Main idea in this paper is to remove control flow instruction from code area and hide them in data area called stack and re-constructed dynamically on demand. Further the process of adding junk bytes is used to make the disassembly process little harder. The evaluation results show that the proposed system is effective in confusing dis-assemblers like IDAPro. In comparing to other obfuscations like signal based obfuscation the proposed system using control flow approach is better and cost effectiveness. Obfuscating all the instructions increases the complexity of the code which can be reduced by using any others means like using hash functions to select instructions for obfuscation. This eventually reduces the complexity of the program to greater extent.

REFERENCES

[1] Data Rescue [Online]. Available: http://www.datarescue.com/ [Last accessed: Feb. 14, 2012]

[2] J. Miecznikowski and L. Hendren, "Decompiling java using staged encapsulation," in Proc. Eighth Working Conf. Reverse Engineering,2001, pp. 368–374.

[3] Digital Law Online, Reverse Engineering [Online]. Available:http://digital-law-nline.info/lpdi1.0/treatise25.html [Last accessed: Feb. 14,2012]

[4]PRNewswire [Online]. Available: http://www.prnewswire.com/news- releases/siia-files-six-new-software-piracy-lawsuits-against-fraud- ulent-online-vendors-across-the-country-69854267.html [Last ac- cessed: Feb. 14, 2012]

[5] Blizzard [Online]. Available: www.blizzard.com [Last accessed: Feb.14, 2012]

[6] "Bnetd," Wikipedia [Online]. Available: http://en.wikipedia.org/wiki/ Bnetd [Last accessed: Feb. 14, 2012]

[7] L. Shan and S. Emmanuel, "Mobile agent protection with self-modifying code," J. Signal Process. Syst., vol. 65, pp. 105–116, 2010.

[8] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software- based survivability mechanisms," Depend. Syst. Netw., pp. 193–202,2001.

Mankandan.R received the B.E degree in computer science and Engineering from the Thirumalai Engineering College affiliated with Anna University and M.E degree in Computer Science and Engineering from St Peter's University and perusing Ph.D degree in computer science and Engineering in the area of Cryptography and Network Security at St Peter's University