Pre-obfuscation model based on COIL intermediate language

Yubo Yang[1]*, Wei Huang[2], Wenqin Fan[2], Zhengming Hu[1]

[1]*Information Security Center, Beijing University of Posts and Telecommunications, 100876, Beijing, China*

[2]*School of Computer Science, Communication University of China, 102600, Beijing, China*

Received 1 March 2014, www.cmnt.lv

Abstract

Code obfuscation techniques changes and complicates the logic and structure within the program on the basis of assuring the procedures being implemented correctly, which needs to transform the original program to an intermediate form which can be analyzed. Most of the pre-obfuscation models at present are lack of unified and formalized intermediate language, and the results of the pre-processing cannot accurately extract the information needed in obfuscation. A pre-obfuscation model COTOOL which is based on COIL is put forward in this paper to accurately locate the distribution and weight of the obfuscation node in the program taking advantage of the pre-processing algorithm of COIL intermediate language. The experimental data shows that this model improves the obfuscation efficiency and effectiveness greatly.

Keywords: Pre-obfuscation model, intermediate language, CTL instruction, instruction weight

1 Introduction

The modern society starts with the rapid development of computer technology and industry, and all walks of life improve their work efficiency and development potential by applying computer technology or software. Due to such kind of demands, computer programs and software systems have greatly enriched and brought convenience to people's production and life, becoming an essential tool. And at the same time, they have brought a good deal of wealth and economic benefits to the individuals and businesses who engage in the IT industries.

Because of the particularity of the software programs themselves, they are easily to be copied and imitated. Once the attacker steals and utilizes the core algorithm and technology in the programs, and makes benefits with them, the sound development of the whole IT industries will be severely damaged. Therefore, for the computer technology and industry, the security technology which protects the intellectual property rights of the software products appears to be particularly important.

As a result, many software security technologies emerge at the right time. The technology of encrypting the software’s key information is used to prevent the attacker from gaining the key information inside the program. One fatal flaw of this technology lies in the preservation of the secret key. Once the key is gained by the attacker, the encryption algorithm will be useless no matter how complicated it is.

Also, a specific hardware can be used as software protector, in which physical protection is applied to realize the security of software. However, this kind of method not only increases the cost of developing software, but also enormously decreases the scalability and practicality of the software.

In the premise of ensuring that the execution of the original program logic remains the same, and through the conversion of semantic equivalence, code obfuscation technique complicates the program control flow and data flow, etc. making the attackers unable to restore right internal logic and data even if they have the executable file of the software. What’s more, even if they are able to accurately restore the internal logic and data of the original software, they have to pay great amount of cost. So they finally have to give up.

In 1997, Christian Collberg summarized and classified code obfuscation techniques in his thesis [1], in which he also firstly put forward the four forms of obfuscation classification. In the thesis, the author collected all the obfuscation algorithms published by others or him and included them in these four forms. Also, he put forward three criterions which judge the good and bad of obfuscation algorithm: *potency*, *resilience* and *cost*. Potency, the depth or complexity of its algorithm, means that to what extent can the after-obfuscation *P’* be understood. Resilience refers to the resistance strength in the process of reverse, namely the time and system space cost needed to in reversion and cost in developing this reversion algorithm. Cost means the additional system memory and performance period added when *P’* runs.

Code obfuscation techniques conducts program pre-processing analysis to the source code or binary file of the program to be obfuscated, thus generates an intermediate result (control flow diagram, intermediate language, etc.) which can be further analyzed, thus realizing the obfuscation with specific obfuscation algorithm on this basis. Compared with source code, the binary file can

*** *Corresponding author’s*** e-mail: yangyubobupt@163.com ignore the dependency of the programming language and make the analysis more universal and accurate, so the current mainstream of code obfuscation techniques all take binary file as the pre-processing object.

2 Related works

When conducting pre-processing before obfuscation to the binary file of the program, the current mainstream code obfuscation model mainly depend on the static program analysis platforms such as CodeSurfer [2], McVeTo [3], phoenix [4] and Jakstab [5] etc. The CodeSurfer program analysis platform mainly conducts static analysis to the binary file of the x86 platform with static analysis algorithm VSA which has quite accurate analysis to the internal storage operation of the program. McVeTo locates and analyzes the status of the execution path in the program with the help of DPG (Directed Proof Generation). Phoenix is a whole set of program analysis platform developed by the Microsoft Cooperation. It can convert other languages into SSCLI (Shared Source Common Language Infrastructure) framework code, which is convenient for later research and analysis. Jakstab tool is an analysis platform to Java code. It defines the disassembling algorithm of iterative disassembler, and generates the program through iterative analysis of the data flow. These platforms mainly convert the binary code into assembly code and conduct analysis on this basis. There is no optimization or disposition conducted to the intermediate result of the program in allusion to the later obfuscation needs, so the control flow graph and related information gained in pre-processing are not accurate enough, leading to an unsatisfactory result of obfuscation algorithm.

In order to make up the insufficient of static program analysis platform to the dynamic information acquisition of the program, the dynamic instruction level binary analysis platforms DynamoRIO [6], PIN [7] and Valgrind [8] adopt code cache technology. In the process of conducting simulation execution to binary program, the program’s execution code will be copied to the code cache, and a specific analysis will be conducted by the modification of the program’s code. The operation of code takes the basic block of the program as the operation unit. The executing state will be saved when each basic unit completes the execution, and it will then be switched to the next basic block to continue the execution. At the same time, these platforms have good expandability. The API provided by the platform can be used to write related program analysis plug-ins to provide information which is more rich and accurate to the analysis result of the pre-obfuscation processing. However, the features of the dynamic analysis platforms make it unable to be the main body of the pre-obfuscation processing, and only some partial codes can be conducted dynamic analysis on the basis of static analysis. The tradeoffs between these two kinds of analysis techniques then become the difficult point in pre-obfuscation model design.

In the current code obfuscation research field, LOCO [9] and PLTO [10] are two relatively mainstream code obfuscation systems. The LOCO includes the modules of LANCET and Diablo, in which LANCET is a set of GUI interface with which the control flow graph of the program will be presented in a visual way and which can conduct search and edit to the basic block in the program according to the association attributes. Diablo can conduct rewriting operation to the binary programs of the platforms when linking, and in this way it conducts fine granularity obfuscation to the binary program, providing three ways to operate it: full-automatic, semi-automatic and manual. The code obfuscation way of control flow flattening and opaque predicates is provided in this obfuscation system. While PLTO generates ICFG (Interprocedural Control Flow Graph) through analysis to the original program and conducts obfuscation to the program code with three steps. First, pruning and simplifying to the branching path branches building ICFG; second, operate the internal storage and register in the program, and deal with the functions; third, redistribute the basic block of the program. These two kinds of tools are also based on the static disassembly result of the program, and the obfuscation algorithms are relatively basic and simple when conducting obfuscation disposition, so they cannot improve the obfuscation effect effectively.

It can be found that in the pre-obfuscation stage, there is a lack of formalized and unified language to describe the intermediate result abstractly. This kind of situation will not only greatly decrease the universality of designing the code obfuscation tools, but also have quite a different obfuscation processing result with the expectations. The current mainstream intermediate languages are *SSA*, *CIL*, *GIMPLE* and *LLVM*, in which *GIMPLE* and *LLVM* are of *SSA* form. The procedure of pre-obfuscation is the decompilation of binary program to intermediate language, and it can accurately describe and locate the obfuscation nodes. Therefore, these languages cannot realize these functions well.

Based on the analysis of the current research status of the code pre-obfuscation model, a pre-obfuscation model COTOOL based on COIL intermediate language is proposed in this paper. This model will conduct initialized analysis to the binary file in the pre-processing stage, and convert it into COIL. By using of related pre-processing algorithm, a control flow graph will be accurately generated and the obfuscation nodes will be located to tail after the access situation of the related internal storage and register, making it more effective when conducting obfuscation algorithm later. Thus, the status that many code obfuscation systems have inaccurately analysis result in the pre-processing stage is well solved. On the other hand, the COIL itself provides many formalized description methods which are needed when conducting obfuscation processing, and conducts symbolic abstract formula to the performance of the program. By this way, the control flow and data flow information details of the program can be accurately described. As the intermediate layer of the operation, the COIL guarantees the integrity and unity in the model, thus ensuring the accuracy of the final obfuscation result.

3 Model construction and Algorithm analysis

3.1 MODEL STRUCTURE

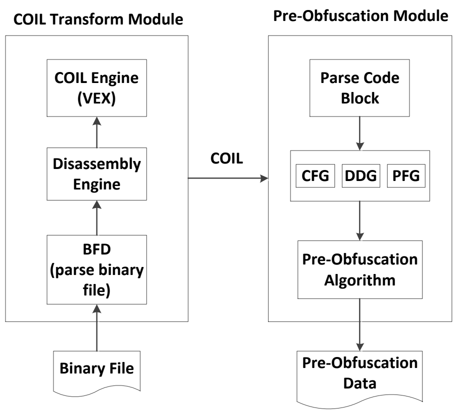

FIGURE 1 Framework of code pre-obfuscation model

The pre-obfuscation model COTOOL includes the COIL transform module and pre-obfuscation module. As the module which transforms the binary file to COIL, it is the basic module to conduct operation. And only on this basis, can the intermediate form of the program be pre-processed. The pre-obfuscation module conducts the COIL intermediate form of the input program and stores the related data information into the database. This is the core part of the model.

As is shown in Figure 1, the designing of COIL transform module mainly includes three functions. Firstly, the BFD (Binary format descriptor) is used to analyze binary program files. This tool can provide API to distinguish the object file formats and analyze related section contents in the binary files. Then, it will select the code section and submit to the disassembly engine to analyze. The file formats the tool supports are PE, ELF, etc.

Secondly, the gained binary code section will be analyzed through the disassembly engine and be transformed to assembly code. The disassembly tools IDA Pro and Kruegel Disassembler are used in the disassembly engine. The advantage of IDA Pro is that it can decrease the assembly code parsing error rate and high efficiency. Some obfuscation operation has been done to part of the programs before analysis, and it may lead to IDA Pro analysis error. The Kruegel Disassembler will work at such occasions to improve the accuracy of disassembly.

Last, the COIL engine will transform the assembly code to the COIL form. In this process, the VEX module [1], which is the module of Valgrind dynamic debugging tools, is firstly used to transform the assembly language to an initial intermediate language. The intermediate language type transformed by this module is relatively easy, and is not able to deal with the needs of the obfuscation operation. What's more, the initial intermediate expression has semantic fuzziness. Therefore, a later optimizing process is needed to optimize the initial intermediate language to the COIL form which could be applied in this model.

After the optimizing process, the form of COIL is finally output.

As is shown in Figure 1, the designing of pre-obfuscation module mainly includes two functions. The first is the generating of the control flow graph. The program is divided into basic blocks through the analysis of the COIL. Except the control flow graph, DDG (Data Dependence Graph) and PFG (Program Dependence Graph) can also be generated by using of the data flow analysis result.

When the program is divided into basic blocks and generates the control flow graph, it will be saved in the database. Therefore, when conducting pre-processing algorithm to this part, related information should be extracted from the database firstly. When conducting analysis with pre-processing algorithm, first locate the instructions, then distribute the nodes and at last handle the interface. As for the information which has been dealt with, save them in the database for to read and analyze directly and conveniently in future when conducting obfuscation process of the program with obfuscation algorithm.

3.2 COIL TRANSFORM

The grammar of COIL is made up of parameters which show the program state. It includes the current instruction state ($\prod$), the current variable state (Δ), the current label state ($\wedge$), the next instruction state of the program (θ) and the current register state of the program (φ). Their description is as shown in Table 1.

TABLE 1 Basic description of COIL

Argument	Description
$\prod$	instruction→memory address
Δ	var→special value
$\wedge$	label→label instruction
θ	next instruction status
φ	current register status

These parameters which show the basic state of the program can express the execution status in the program accurately. These formalized expressions have the following fundamental forms:

$$\frac{transformation\ rule}{stmt \Rightarrow stmt'}, \qquad (1)$$

where $stmt \Rightarrow stmt'$ represents the change of the program state and *transformation rule* represents the calculation rules of program status changes. Based on such formalized expressions, the intermediate language can be used to abstract various states in the process of program execution to corresponding mathematical forms and then analyze and deal with them.

jmp(e) in the program state can be describes with the following expression:

$$\frac{\wedge \| \Pi\Delta\forall e \Downarrow v \| \varphi, \theta = \sum[v]}{stmt(\wedge,\varphi,\Pi, jmp(e)) \Rightarrow stmt'(\wedge,\varphi,v,\theta)}. \tag{2}$$

cjmp(e$_1$, e$_2$, e$_3$) in the program state includes two kinds of jump conditions, *cjmp (0, e$_2$, e$_3$)* when the condition is false and *cjmp (1, e$_2$, e$_3$)* when the condition is true.

The *halt(e)* in the program state can be describes with the following expression.

$$\frac{\Pi\Delta\forall e \Downarrow v \| \varphi}{stmt(\wedge,\varphi,\Pi,halt(e)) \Rightarrow term}. \tag{3}$$

assert(e) in the program state includes exception handling of two situations, *assert(1)* when the condition is true and *assert(0)* when the condition is false. These two kinds of states can be described with the following expressions.

The expression when the exception handling condition is true:

$$\frac{\wedge \| \Pi\Delta\forall e \Downarrow 1 \| \varphi, \theta = \sum[v_1]}{stmt(\wedge,\varphi,\Pi,assert(e)) \Rightarrow stmt'(\wedge,\varphi,v_1,\theta)}. \tag{4}$$

The expression when the exception handling condition is false:

$$\frac{\wedge \| \Pi\Delta\forall e \Downarrow 0 \| \varphi, \theta = \sum[v_2]}{stmt(\wedge,\varphi,\Pi,assert(e)) \Rightarrow stmt'(\wedge,\varphi,v_2,\theta)}. \tag{5}$$

The expression description of *var* in the program is as following:

$$\frac{\Delta[x] = v}{stmt(\wedge,\varphi,\Pi,\Delta\forall x \Downarrow v)}. \tag{6}$$

The expression description of binary arithmetic operation $\Diamond_b$ in the program is as following:

$$\frac{\wedge \| \Pi\Delta\forall e_1 \Downarrow v_1, \Delta\forall e_2 \Downarrow v_2, v = v_1 \Diamond b v_2 \| \varphi}{stmt(\wedge,\varphi,\Delta\forall e_1 \Diamond b e_2 \Downarrow v)}. \tag{7}$$

The expression description of unary arithmetic operation $\Diamond_u$ in the program is as following:

$$\frac{\wedge \| \Pi\Delta\forall e_1 \Downarrow v_1, v = \Diamond b v_1 \| \varphi}{stmt(\wedge,\varphi,\Delta\forall \Diamond b e_1 \Downarrow v)}. \tag{8}$$

The program state of COIL and formalized expression of operation provides a mathematical form intermediate results which can be analyzed and calculated for the code obfuscation model operation to original program and later obfuscation processing, thus guaranteeing the accuracy and uniqueness of operation and analysis the changes of internal storage and register clearly in this process. This solves the problem of difference of intermediate expression form in the obfuscation processing, and at the same time solves the universality problem of obfuscation algorithm in code obfuscation model.

3.3 PRE-OBFUSCATION ALGORITHM

The basic principle of code obfuscation is that through changing and making complicated the execution paths of control flow or data flow in the program to guarantee the software safety. In order to effectively conduct obfuscation to the logic in the program, the CTI instruction in the program should be the primarily considered target in obfuscation. The CTI includes direct jump instruction, condition jump instruction, return instruction and function reference instruction. The specific COIL form of expression is the program state statements such as *jmp*, *cjmp*, *assert* and *halt*, etc. Therefore, in order to effectively conduct obfuscation process to the program, we should firstly locate these instructions in the program when pre-processing binary programs. The algorithm of locating CTI instruction is based on the *label* of the COIL.

The definition of *label* is given:

Definition 1 Assume that the CTI instruction in program *P* is *e*, then the *label* can be expressed as *label (e)*, and the formula expression is as following:

$$label(e) = \frac{\varphi \| \Pi[e] \Rightarrow \theta[e']}{\wedge[\vartriangle] \Rightarrow \wedge'[\vartriangle']}, \tag{9}$$

where *e* is the CTI instruction which is labeled, and the program's instruction jump change state and the next *label* location is recorded.

```
Input: Binary Program P
Output: Program P' based on COIL
language

Function: set_label_information
  while(instruction ins != NULL)
  {
    coil_ins= Convert_Binary_to_COIL(ins);
    if(coil_ins == CTI instruction)
    {
      set_label_tag(coil_ins);
      info = label(coil_ins);
    }
    else
    {
      info = NULL;
    }
      ins = next instruction;
  }
```

FIGURE 2 Pseudo code label to CTI instruction

After defining the statement of *label*, a series of operations should then be done to realize the adding and processing of *label* statements. It is mainly divided into the following three stages:

Firstly, in the process of transforming binary program to COIL, label all the CTI instructions, and fill up the related information of the *label* statements in allusion to the logic of this CTI instruction.

Secondly, when generating the control flow graph by COIL, analyze and summarize the *label* information in the basic blocks divided and generates the *label* information of the basic blocks which can mainly be used to label the branch jump information of the entry and exit points of the basic blocks.

At last, establish relationship between the label information of the basic blocks through logic of jumping. By this way, the CTI instructions gathering of the paths will be got based on the specified route of the control flow in the later operation. And this can be seen as a basis to conduct obfuscation process.

In the first stage, *label* to the CTI instruction of intermediate language should be done, and related information should be generated according to the formula definition of *label*. The pseudo code of the algorithm is as shown in Figure 2.

In the second stage, control flow graph of the generated COIL should be generated. And in the process of dividing the basic blocks of the control flow graph, the *label* statement information in the basic blocks should be merged and the jump information of the entry and exit points as well as the internal CTI instruction logic should be labeled. Figure 3 shows the information included in the *label* of the basic blocks.

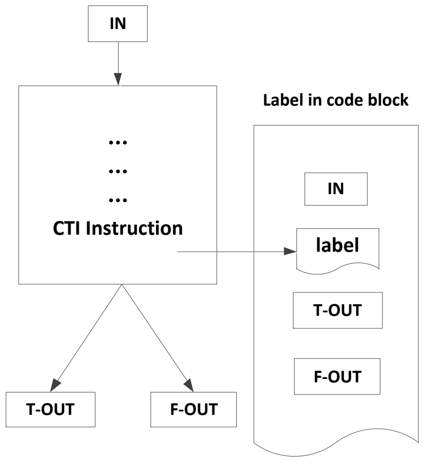

FIGURE 3 Label of the basic block

The last stage is to relate the information in the *label* of single basic blocks based on the control flow graph. In this way, there is no need to conduct secondary analysis to the information of the basic blocks, and the location of information of all the CTI instructions can be located through the information of *label* in the specified program execution path.

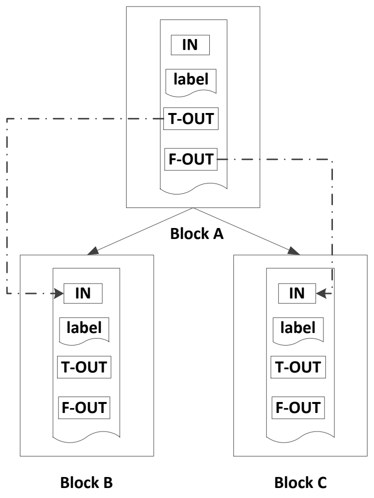

FIGURE 4 Interrelation of the label information in basic block

As is shown in Figure 4, when the basic blocks generate their own *label* information, a connection will be established through the logic relationship between input and output. The output in *label* has two kinds of situations: *T-OUT* when the condition is true and *F-OUT* when the condition is false. If there is no condition jump for the output of basic blocks, then set them unified as *T-OUT* and set *F-OUT* as *NULL*.

The process of obfuscation processing to binary program mainly includes two aspects. The first aspect is to locate the CTI instruction in the control flow graph of the original programs. And the second is to control the location, quantity and density, etc. when adding obfuscation nodes because the control of these attributes directly decide the improvement effect on security after the program obfuscation.

Therefore, to define the obfuscation nodes distribution based on control flow, firstly we need to analyze the weight of the CTI instructions of the basic blocks. Then, the Dijkstra algorithm will be used to conduct shortest path analysis to the control flow graph which has weight, and confirm the distribution of obfuscation nodes in the program according to the analysis result.

First, conduct weight classification to the CTI instruction in the basic blocks of the program control flow graph. The classification of the weight should be done based on the following three criteria.

The execution frequency of the CTI instruction: which is shown as CTI_{fre}, refers to the number of times the CTI instruction being executed in entire processing procedure in the program’s control flow graph. The CTI instruction which performs higher execution frequency always lies in the loop structure of program or the function entry point which is frequently called. This kind of CTI instruction is generally the entry point through which the key algorithm or specific algorithm is conducted.

The execution density of the CTI instruction: which is shown as CTI_{int}, refers to the continuously called CTI instruction quantity when it is called in an entire processing procedure in the program’s control flow graph.

This kind of situation always appears in the nested function, recursive function or the operations of accessing the address again and again. This kind of CTI instruction mainly focuses on the process of repeat call and data access of the program to some specific operations.

The execution priority of the CTI order: which is shown as *CTI$_{pri}$*, refers to the importance of CTI instruction when the program executes in the control flow graph. For example, the CTI instruction which calls some key algorithms or functions performs higher execution priority, while the ordinary jump statement, comparison and cyclic process perform lower execution priority.

Based on the above criteria to CTI instruction weight classification, the definition of CTI instruction weight is then put forward here.

The program control flow graph of *P* is *G*, and the number of basic blocks called in an execution path in *G* is *n*, then the quantity of CTI instruction called in this path will also be *n*, and the number of times CTI instruction being called will be *m*, and *m*>*n*. From this, the algorithm formula of *CTI$_{fre}$*, *CTI$_{int}$* and *CTI$_{pri}$* can be put forward.

Definition 2 Assume that the gathering of CTI instruction in the execution path can be shown as Ω, then $\forall CTI_i \in \Omega$, the standard deviation σ_{CTIF} of CTI instruction execution times in this path can be shown as:

$$\sigma_{CTIF} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(CTI_i - \frac{m}{n}\right)^2}\ . \tag{10}$$

Definition 3 Set the execution frequency of CTI instruction as *CTI$_{fre}$*, and the execution times of the current CTI order is *x*, then the *CTI$_{fre}$* can be expressed as:

$$CTI_{fre} = \begin{cases} 0 & , x-1 < \sigma_{CTIF} \\ \dfrac{\left(x-\dfrac{m}{n}\right)-\sigma_{CTIF}}{\sigma_{CTIF}} & , x-1 \geq \sigma_{CTIF} \end{cases}\ . \tag{11}$$

where the CTI instruction will be executed at least once, so $x \geq 1$.

Definition 4 Set the execution density of CTI instruction as *CTI$_{int}$*, and the continuous jump times of the current CTI instruction is *x*, then the standard deviation of CTI instruction execution density is σ_{CTIF}, so the *CTI$_{int}$* can be expressed as:

$$CTI_{\text{int}} = \begin{cases} 0 & , x < \sigma_{CTII} \\ \dfrac{\left(x-\dfrac{m}{n}\right)-\sigma_{CTII}}{\sigma_{CTII}} & , x \geq \sigma_{CTII} \end{cases}\ . \tag{12}$$

where $x \geq 0$.

Definition 5 Set the execution priority of the CTI instruction as *CTI$_{pri}$*. The priority cannot be quantized with formulas, so the way of specified rules will be used to define:

$$CTI_{pri} = \begin{cases} 1 & , basic\ priotity \\ 2 & , key\ branch \\ 3 & , entrance\ of\ \ function \end{cases}\ . \tag{13}$$

The equation of the CTI instruction weight will be received then.

Definition 6 The weight of CTI instruction here can be represented as *W$_{CTI}$*:

$$W_{CTI} = [CTI_{fre} + CTI_{\text{int}} + CTI_{pri}], \tag{14}$$

where *W$_{CTI}$* should be a positive integer.

When the defined equation of CTI instruction is gained, the control flow graph *G=(V,E)*, can be changed to *G'=(V,E,W)*, Then, apply the Dijkstra algorithm to calculate the shortest path between CTI instruction nodes in the control flow graph.

There are at most two exit edges in the structure of basic block of the process control flow graph, so an execution path which performs high weight will be got when the result of the Dijkstra algorithm is inverted. After that, confirm the number of obfuscation nodes on this path according to the expectations of obfuscation, and then locate them according to the high-low order of the weight.

```
Input: the CFG G of the program P
Output: new CO-node coverage

function CO_node_coverage
{
  while(CTI instruction != NULL)
  {
    generate_weigth_CTI(CTI instruction);
   next CTI instruction;
  }
  G'=Dijkstra(G, w, s);
  if($\vec{G}'!=0$)
  {
   insert_CO_node($\vec{G}', W_{CTI}$);
  }
}
```

FIGURE 5 Pseudo code of obfuscation node distribution algorithm

Therefore, the pseudo code of the obfuscation nodes distribution algorithm based on control flow is as shown in Figure 5.

The obfuscation nodes distribution algorithm based on control flow conducts the partition of the weight by the characteristic of CTI in execution. Then the CTI instruction in the control flow graph is located through the weight information generated, and an insert operation will be done to the obfuscation nodes according to related information. Thus, the distribution of obfuscation nodes can be conducted according to the importance of execution position in program control flow.

4 Experiment

In order to verify the effectiveness of code pre-obfuscation model on improving the program obfuscation efficiency and effect, the following experiment is performed.

In the experiment, the benchmark program *SPECint-2006 benchmark suite* is selected, with the latest version 1.2 [12]. There are 12 test benchmark programs in *SPECint 2006*. They are mainly compiled with C/C++ language. The program of test set will be provided by the form of source code, which is convenient for us to choose a proper compiler before compiling the program to a binary program to adapt to different operation system. In the experiment, the 6 programs shown in Table 2 are selected to perform comparison test.

TABLE 2 Benchmark of the experiment

Benchmark	Language	Compiler	Platform
400.perlbench	C	GCC	UBUNTU
401.bzip2	C	GCC	UBUNTU
429.mcf	C	GCC	UBUNTU
445.gobmk	C	GCC	UBUNTU
456.hmmer	C	GCC	UBUNTU
458.sjeng	C	GCC	UBUNTU

The experimental platform is 2.9Hz Intel Core2 Duo CPU, with an internal storage of 4GB. The operating system platform is Ubuntu 11. And the compiler is gcc version 3.4, with an optimization level of O3. The IDA Pro tool will accomplish the reverse work for the test program to assess the obfuscation result.

MPOP is used in the experiment as the obfuscation program. Through comparison between the obfuscation program and the program using the pre-obfuscation model, the experiment tests the effectiveness of this model from aspects of obfuscation efficiency, obfuscation effect and anti-reversion execution time.

When conducting comparison test to the obfuscation effect, the direct obfuscation which uses MPOP algorithm and obfuscation based on COTOOL model are respectively applied to the sample program, and the obfuscation efficiency comparison result in Figure 6 is showed.

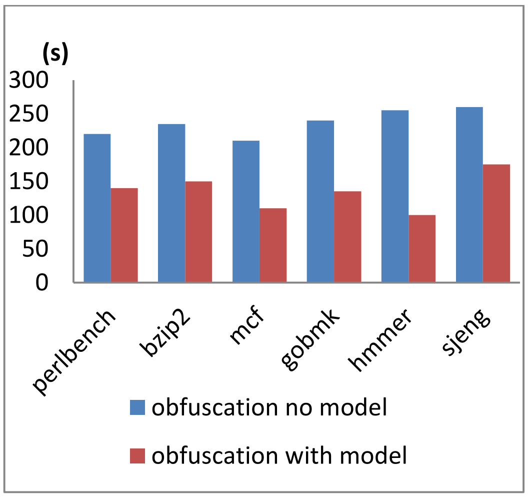

FIGURE 6 Time comparison of obfuscation efficiency

As is shown in Figure 6, compared to direct obfuscation, the obfuscation efficiency based on COTOOL model has significantly improved, with the average execution time decreasing about a half. The COTOOL model conducts extracts and analyzes the related information of the original program, so the time spent in traversing the entire program is decreased.

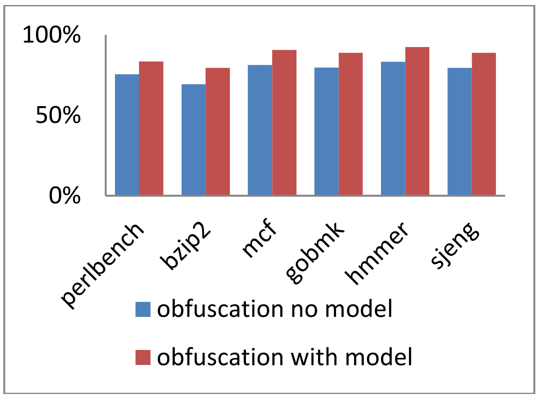

FIGURE 7 Effectiveness comparison of obfuscation

As is shown in Figure 7, the comparison result of anti-reversion ability after obfuscation of two ways applied in the sample program is revealed. The MPOP obfuscation algorithm itself has good obfuscation effect, being able to reach an average obfuscation degree 70%. With the COTOOL model, the obfuscation degree has an average increase of 15%, which is quite obvious.

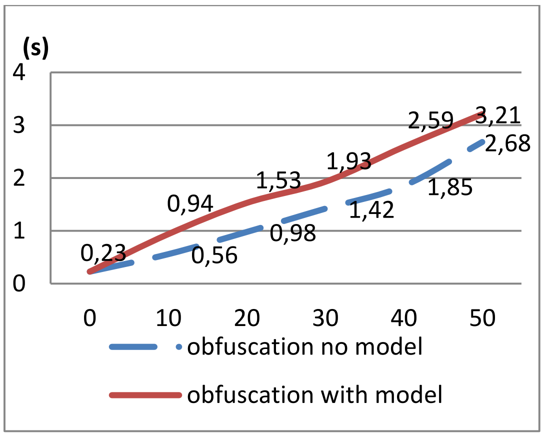

FIGURE 8 Performance time comparison of anti-obfuscation

As is shown in Figure 8, with the increasing of the obfuscation nodes added in the sample program, the average time the anti-obfuscation tool spent to traverse the execution program path also increases. Due to the fact that the obfuscation program based on COTOOL can locate the key position of the program more accurately and the effects of the obfuscation are more obvious, the execution time increases compared to the original obfuscation algorithm, making the anti-obfuscation program unable to accurately restore the original program because of the increasing of time cost.

5 Conclusion

Code pre-obfuscation model, COTOOL, which is based on COIL, is proposed in this paper. In allusion to the current situation that there is no formalized unified language to obfuscation pre-processing, this model proposes the COIL intermediate language to describe the key information and logic in the program. Also, it put forward the pre-processing algorithm which is based on COIL to extract the location distribution and weight information of obfuscation node in the original program, decreasing the time spent in the obfuscation algorithm traversing the entire program. In the experiment, it is indicated that through the tests of obfuscation efficiency, obfuscation effect and anti-reversion execution time, the COTOOL pre-obfuscation model can effectively improve the obfuscation efficiency and effect of obfuscation algorithm.

References

[1] Collberg C, Thomborson C, Low D 1997 A taxonomy of obfuscating transformations *Department of Computer Science University of Auckland* New Zealand

[2] Balakrishnan G, Gruian R, Reps T, Teitelbaum T 2005 CodeSurfer/x86 – a platform for analyzing x86 executables *Compiler Construction* 250-4

[3] Driscoll E, Thakur A, Reps T 2012 OpenNWA: a nested-word automaton library *Computer Aided Verification* 665-71

[4] "phoenix" 2013 [Online] Available: http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx

[5] Nguyen M H, Nguyen T B, Quan T T, Ogawa M 2013 A Hybrid Approach for Control Flow Graph Construction from Binary Code *Software Engineering Conference (APSEC 2013 20th Asia-Pacific)* 159-64

[6] Zhou N 2014 Dynamic Program Analysis and Optimization under DynamoRIO

[7] Luk C J, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi V J, Hazelwood K 2005 Pin: building customized program analysis tools with dynamic instrumentation *PLDI '05 Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* **40**(6) 190-200

[8] Carrez S 2013 Optimization with Valgrind Massif and Cachegrind

[9] Madou M, van Put L, de Bosschere K 2006 Loco: An interactive code (de) obfuscation tool *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* 140-4

[10] Hiltunen M A, Jim T, Rajagopalan M, Schlichting R D 2011 System and method for enforcing application security policies using authenticated system calls *Google Patents*

[11] Bandhakavi S, King S T, Madhusudan P, Winslett M 2010 VEX: Vetting Browser Extensions for Security Vulnerabilities *USENIX Security Symposium* **10** 339-54

[12] SPEC CPU2006 2013 [Online] Available: http://www.spec.org/cpu2006/

Authors

Yubo Yang, 1986, Shanxi, P.R. China.

Current position, grades: PhD candidate in Information Security Center, Beijing University of Posts and Telecommunications, China.
University studies: Beijing University of Posts and Telecommunications, China.
Scientific interests: information security, code obfuscation.

Wei Huang, 1983, Anhui, P.R. China.

Current position, grades: instructor at the School of Computer Science, Communication University of China.
University studies: Beijing University of Posts and Telecommunications, China.
Scientific interests: information security.

Wenqing Fan, 1983, Sichuan, P.R. China.

Current position, grades: Instructor in School of Computer Science, Communication University of China.
University studies: Beijing University of Posts and Telecommunications, China.
Scientific interests: information security.

Zhengming Hu, 1931, Hubei, P.R. China.

Current position, grades: Professor in Information Security Center, Beijing University of Posts and Telecommunications, China.
University studies: Beijing University of Posts and Telecommunications, China.
Scientific interests: information security.