Dynamic Obfuscation Algorithm based on Demand-Driven Symbolic Execution

Yang Yubo and Yang Yixian
Information Security Center, Beijing University of Posts and Telecommunications, 100876, Beijing, China
Email: satiago.yang@gmail.com, yxyang@bupt.edu.cn

Fan Wenqing and Huang Wei
School of Computer Science, Communication University of China, 100024, Beijing, China
Email: {fan.winking, huangwei.bupt}@gmail.com

Li Zhongxian
National Cyberspace Security Ltd, 100088, Beijing, China
Email: lzx@ncs-cyber.com.cn

Abstract—Dynamic code obfuscation technique increases the difficulty of dynamically reverse by the runtime confusion. Path explosion directly affects the efficiency and accuracy of dynamic symbolic analysis. Because of the defect, this paper presents a novel algorithm DDD (Demand-Driven Dynamic Obfuscation Algorithm) by using the demand-driven theory of symbolic analysis. First, create a large number of invalid paths to mislead the result of symbolic analysis. Second, according to the demand-driven theory, create a specific execution path to protect the security of software. The design and implementation of the algorithm is based on the current popular and mature SMT (satisfiability model theory), and the experimental effects are tested by Z3 - the SMT solver and Pex - the symbolic execution test tools. The experimental results prove that the algorithm enhance the security of the program.

Index Terms—Symbolic Execution; Dynamic Obfuscation; Path Explosion; Information Hiding

I. INTRODUCTION

Software intellectual property protection requires a lot of security technology such as software watermarking [1] and malware detecting [2]. Among these technologies, the code obfuscation has been proposed [3] as an important protect technology in the 1990s. The code obfuscation technology has developed gradually and become mature in the following 20 years. This kind of technology protect the key information by the direct confusion of internal logic and structure. Because it can guarantee the equivalent of logical semantics within the program, so the program can still operate normally.

Code obfuscation technique is divided into static obfuscation and dynamic obfuscation. In general, the purpose of the static obfuscation is to achieve the static program analysis confusion by changing the internal structure of the program, data and control flow. Because the static program analysis only analyze possible execution paths through the binary code of program rather than really execute the program. So using the opaque predicate (such as address, functions, pointers, etc.) instead of the judgment of the branch condition [4] can effectively enhance the difficulty of the branch condition and protect the key information. However, the static obfuscation become very vulnerable if the adversary uses dynamic program analysis method.

The basic idea of dynamic obfuscation is changing runtime code or execution path dynamically in the program during the operation of the program. This method can effectively resist the static program analysis, and because of the dynamic characteristics of the runtime code and execution path, it also has a good effect against the dynamic analysis of the program.

A basic structure of the dynamic model of confusion shows in Fig. 1. Assume that there is a program $P$ in the software domain, the function key is the core code block that need to be protected. The dynamic obfuscators $obf1$ and $obf2$ are also waiting to be called in the program domain during the operation process.

![Figure 1. A structure model of dynamic obfuscation](image-url)

There are two ways of dynamic confusion of original program: self-modifying code and dynamic execution paths. Self-modifying code is the most widely used method at present in dynamic obfuscation algorithm. As for example, in Fig. 1, it is divided into two stages: first, using the obfuscator $obf1$ to transform the protected code block key in the program compile stage, afterwards, using the obfuscator $obf2$ to restore the source code to execute.
the program in the runtime. So that is how the self-modifying code realized.

The method of self-modifying code can ensure that there is only a small program fragment restored at the same time in the window during the program is running. The adversary can not reverse the entire program in this way.

The new obfuscation algorithm DDD proposed in this paper based on the dynamic execution path. Take Fig. 2 as an example, assume that the key is the protected code block in the program P, the program execute key code block only through path 1 and 2. In its confusion, the program add obfuscator \( \text{obj1} \) and \( \text{obj2} \) (as one possibility, obfuscator can be set according to demand), controlling the execution path at runtime, according to the different conditions to generate different execution paths, such as the path 5 (invalid path) and path 6 (delayed execution), in order to mislead the reverse analysis results.

![Program P and Obfuscated program P](image)

Figure 2. The dynamic change of execution path

The main contribution of the DDD algorithm is providing strong protection against dynamic reverse engineering. In one hand, use the information of function summary in symbol analysis to generate invalid or irrelevant paths to enhance the complexity and difficulty of dynamic analysis. In the other hand, use the information of demand-driven to bypass or deferred the critical code block automatically in the runtime of program. The design of algorithm is based on the algorithm model SMT [5].

II. RELATED WORK

Code obfuscation technology was first proposed in 1993. Fred Cohen proposed a new theory of semantic transformation [3], and designed a converter system for system variety, which is transform the system into a new version with the same semantics, but syntactically different. This paper is the first proposed two algorithms of dynamic obfuscation, one is called \textit{build-and-execute}, which dynamically generated the execute code needed at the runtime, then jumps to perform. Another one is called \textit{self-modification}, it is the protected program that is being converted in the initialization step, dynamically restore code in the runtime. After this, the research of code obfuscation has begun to develop.

The basic definition of code obfuscation presented by Christian Collberg in his paper [6]. In simple terms, code obfuscation is confusing the source program \( P \) to \( P' \) using the algorithm \( T \), and \( P \) and \( P' \) are equivalent in semantics and logic. This conversion to comply with two principles: (1) If the source program \( P \) terminates failed or abnormally terminated, the program \( P' \) is not necessarily terminated; (2) On the contrary, if the source program \( P \) terminates normally, the program \( P' \) must terminates and has the same output as program \( P \).

It also summarizes and classifies code obfuscation techniques, the four forms of code obfuscation are first proposed in this paper. The author presents a summary of himself and other published algorithms. Then he includes these algorithms into four forms of confusion. Besides, he proposed three criteria to assess obfuscation algorithm, they are: potency, resilience and cost.

David Aucsmith proposed an important dynamic confusion algorithm in the paper [7], which is called \textit{self-modifying state machine}, the algorithm is divided into three parts: the first part is the deformation process, the program is divided into several fragments, fragments will continue to move and operate XOR with each other, and at the same time only some fragments in the program in the plaintext state. The second part is to add encryption technology in order to make less program fragment in the plaintext state. The third part is to construct a series of functions to protect each other in order to ensure the program’s functions are not modified maliciously.

After that, the code obfuscation algorithm is increasing, such as Yuichiro Kanzaki published paper [8], the main idea is to replace the instruction, namely replaces the original instruction to other instruction in the initialization step and restore the instruction when the program is running into here, this can ensure the program run correctly only in a very small time window.

Matias Madou proposed a dynamic code merge algorithm in the paper [10], which allows functions share the same address space of a memory. As the program execute, different functions are used through dynamic changes of the shared memory data.

The above discussion all belong to the category of self-modifying code algorithms. Researches in this area are already matured. Nikos in the paper [11] summarized the code obfuscation algorithm of self-modifying code, the current algorithm were systematically analyzed through three aspects: the way of hiding, encoding of confusion and the exposure of source code, and the existing advantages and disadvantages of self-modifying confusion algorithm were compared.

The research of dynamic execution path is not enough at present. Through the dynamic execution path to resist dynamic program analysis thesis was clearly proposed [12] by Sebastian. In the paper, he divide the program and the divided code blocks are called gadget, then borrow branch function algorithm principle [13] to achieve the relation and jump of gadget, and the program’s input as pre-conditions of the change of execution path, different input data will lead to the execution path showing dynamic changes.
The basic idea of this method is designed by the program at run time to generate a large number of invalid paths to increase the difficulty and cost of dynamic analysis of adversary in order to protect the critical information. The biggest flaw of the algorithm is a lot of gadget will always jump through the same function address (branch function), so the adversary can use this feature correctly restore the original program logic.

III. BACKGROUND

Here we assume that the program P will be confused, the program’s input parameters I, the vector set contains subroutine input data of various stages of the program. The tree logic of source code control flow graph (CFG) is T, expressed in various input I of all possible execution paths. The execution path of control flow is graph ρ which represents a collection of a series of the execution statements. Path constraint condition is \( \varphi_p \), indicate the mathematical characteristic of the specific execution path ρ when input is specified.

The path constraint condition \( \varphi_p \) based on the series set of program input conditions and output conditions, for the convenience of mathematical formula expression here, our program logic processing and sample is limited to T theory (including integer linear operation, pointer operations, arrays, and character as well as bit operating) [14]. Library function of operation system will not appear in this example.

Another reason for using T theory mathematical formula for constraint condition expression is that it can be used to validate the proposed code obfuscation algorithm results with the SMT solving model. The mainstream Lazy SMT model method [15] is based on the T theory first-order logic formulas for solving by putting theory into mathematical model algorithm can be more convenient with the SMT solver to ensure the reliability of the algorithm.

The design of DDD algorithm in this paper is for the current symbol dynamic execution analysis. From the perspective of mathematical logic, the purpose of obfuscation is to confuse the symbolic solvable execution path into unsolvable or difficult to solve—that is turn a solvable problem P into an unsolvable problem NP.

Based on the concepts of the design of algorithm, the relation of mathematical model, the research of mainstream dynamic execution theory [16] and mathematical logic extension, the DDD algorithm proposed four important obfuscation concepts: jump node\( (jn) \), node summary, target driven and program components. The symbol dynamic analysis can be more effectively confused with the novel algorithm.

A. Jump Node

Jump node is the main components of the DDD algorithm. By inserting \( jn \) in the program of T, the program execution path can be changed and controlled by the logic of \( jn \).

![Original program execution tree](image)

Figure 3. The effect of the execution path with \( jn \)

The \( jn \) is added when the program has been confused, and existed as a tree node in the execution tree of the program. Shown in Fig. 3, the program adds a \( jn \) and generates a new execution path, which is illustrated by the dotted line (jump between the \( jn \)). The tree node 1 to tree node 4 must go through tree node 2, while after the confusion directly from tree node 1 to node 4. The existence of \( jn \) summary information can guarantee the correct program logic of the new execution path.

Each \( jn \) corresponds to a unique ID, there are two function of the ID, the first one is to locate the node summary information in the lookup table through the hash function. The second one is to determine the relative position of the \( jn \) in the execution tree.

The position and number which \( jn \) added need to consider efficiency and safety performance after confusion. In general, the number of \( jn \) will be more near the protected code block, and to ensure that near the execution path there will be a certain number of \( jn \) to cooperate with it.

Since \( jn \) will calculate the next jump address through the current constraint conditions when executed, so when the number of \( jn \) is large, it may reduce the program execution performance, so that the number and density of \( jn \) need to be controlled.

B. Node Summary

Node summary holds the constraint conditions set of the precursor and the successor of \( jn \), such information is stored in the form of logical expression. The node summary information can be extracted and analyzed when through the \( jn \) to determine the location of the current node in the execution path and the address of the next jump.

Assume that \( w \) is the execution path through \( jn \), the constraint condition of execution path is \( \varphi_w \), and then the formula is as follows:

\[
\varphi_w = \text{pre}_w \land \text{post}_w \quad (1)
\]

In the formula (1), \( \text{pre}_w \) indicates the execution path constraint conditions set of input, and \( \text{post}_w \) indicates that the execution path constraint conditions set of output.

The formula of \( \text{pre}_w \) constraint conditions need to extract all the input of the path first. The input is \( x \), the internal function of \( jn \) is \( f_{jn} \), then the formula of \( \text{pre}_w \) is as follows:

\[
\text{pre}_w = \bigcap_{x \in \text{symbolic}(x, f_{jn})} \text{pre}_{xw} \quad (2)
\]
In the formula (2), symbolic indicate to symbolic the input conditions \( x \) of function \( f_{\mu} \) based on the SMT model.

The internal function \( f_{\mu} \) of \( post_{\mu} \) constraint condition returns the values \( ret \), and then \( post_{\mu} \) formula is as follows:

\[
post_{\mu} = \bigcap_{(ret, f_{\mu})} \text{symbolic}(ret, f_{\mu}) \tag{3}
\]

In the formula (3), the input conditions \( ret \) of function \( f_{\mu} \) based on the SMT model and the symbolic set as the output constraint condition of the path \( w \).

The set of the path \( w \) through the \( jn \) can be represented as node summary, \( \phi_{jn} \) represent as the node summary, the formula of node summary is as follows:

\[
\phi_{jn} = \bigcup_{\phi_{x}} \tag{4}
\]

Node summary information will be generated when the program is compiled and stored in the program's data section as the form of lookup table, each node summary of the \( jn \) as a unit in the table. The lookup table is loaded into memory when the program runs, and continuously revised and updated according to the information of input and output paths.

Every time the new path information through the \( jn \) compared with the node information stored in the current node to determine the execute direction of the node. The program will be semantic equal after the jump through the \( jn \).

If the new path information through the \( jn \) does not match the current node information, then the path for the first execution, the current path constraint conditions to do join operation with the current node information, the updated information is stored to the lookup table.

C. Target Driven

The basic idea of target-driven is set the node will execute the critical function and algorithm in the program execution tree as the target node \( (tn) \). The \( tn \) is the driven-node determines the location of the next jump of \( jn \).

To locate the \( tn \), the node summary information of \( jn \) near the \( tn \) will be needed. The basic information of the target-driven is the set of input and output constraint conditions through the specified \( tn \). The information of the \( tn \) was stored in the node summary information in the form of NOT logic.

Define \( \phi_{tn} \) as the target-driven information, \( \varphi_{jn} \) as the path constraint condition, the execution path through the target node as \( w_{jn} \), the formula expressed as follows:

\[
\phi_{tn} = \bigcup_{(\mu, jn \rightarrow \mu_{jn})} \varphi_{jn} \tag{5}
\]

In the formula (5), the target-driven information of the \( tn \) is the set of constraint conditions from the adjacent \( jn \) to the specific \( tn \). So, assume that \( \phi_{jn} \) is the information of the node summary of \( jn \) adjacent to the target node \( tn \), the formula is expressed as follows:

\[
\phi_{jn} = (\bigcup_{\phi_{x}}) \cup ((\bigcup_{(\mu, jn \rightarrow \mu_{jn})} \varphi_{jn})) \tag{6}
\]

\[
\Leftrightarrow \phi_{jn} = (\bigcup_{(\mu_{tn})} \varphi_{jn}) \tag{7}
\]

In the formula (6) (7), \( \phi_{jn} \) added the information of \( tn \) based on the information of the \( jn \) node summary, when the program executes these nodes, compared to the current path information and the node information \( \phi_{jn} \), if the NOT logic of the \( tn \) information \( \phi_{tn} \) doesn’t meet \( \phi_{jn} \), then the program will execute to the \( tn \), and the obfuscation algorithm will be used.

D. Program Component

Program component is a kind of the program to be fragmented. When slice the program, bind the \( jn \) and the adjacent node together as a component, and the program components in the algorithm is considered as a whole.

![Figure 4](image)

**Figure 4.** The function of the target node

Shown in Fig. 4, node 6 is a \( tn \), when the node 3 execute to \( jn \), because of the driven action of the \( tn \) 6, the execution path may go along the dashed line to the node 5 or skip node 6 continue to execute.

![Figure 5](image)

**Figure 5.** The program component

\( jn \) added in the confusion program execution paths bind some nodes based on the distance of the execution path as the components, each component may contain a certain number of \( jn \), and the node of components may overlap with each other. Overlapping nodes are called share node (sn). Shown in Fig. 5, the node 6 is \( sn \) of the two components, the \( jn \) of component 1 and component 2 analysis the mutual position and distance through the information of the node summary, and to jump as the unit of component, this is improved the efficiency of algorithm by increasing the relative jump distance.
IV. The Algorithm of DDD

DDD algorithm is divided into three parts, namely the main algorithm of the dynamically confused, and two sub-function algorithms: generate new execution path and bypasses the target code block. Through the combination of these three parts of the DDD algorithm, making obfuscated program can effectively resist the symbolic dynamic analysis.

The three parts of the DDD algorithm discussed below are the main logical body of the \( jn \), the jump between the \( jn \) through the logical processing of the three parts of the algorithm to achieve the purpose of the dynamic obfuscation.

A. The Main Algorithm of DDD

The main algorithm shows in algorithm 1, \( I \) passed by the previous node as the input of the currently executed \( jn \), the constraint expressions \( \varphi_{input} \) was converted by the function \( exInput \). After is to initialize some variables in the algorithm, first is to get the address \( Addr_{lookup} \) of the \( lookup \ table \) of the node summary address, and calls the function to locate the node summary information \( \varphi_{jn} \) and the target-driven summary information \( \varphi_{mn} \) of \( jn \) through the hash.

\[
\text{Algorithm 1. The main algorithm of jump node} \\
\text{Input: } I \text{ from the last node} \\
\text{begin:} \\
\text{Addr}_{lookup} \leftarrow \text{Lookup table} \\
\varphi_{input} \leftarrow exInput(); \\
\varphi_{jn} \leftarrow \text{ReadjnFuncSum( Addr}_{lookup} ); \\
\varphi_{mn} \leftarrow \text{ReadtnFuncSum( Addr}_{lookup} ); \\
\text{Result } \leftarrow \text{CompareFuncSum( } \varphi_{input}, \varphi_{jn}, \varphi_{mn} \text{ );} \\
\text{case: } \text{Result } < 0 \\
\text{Addr}_{jn} \leftarrow \text{AvoidExPath( } \varphi_{input}, \text{Result);} \\
\text{if } \text{Addr}_{jn} \neq \text{null} \\
\text{UpdateLookup( Addr}_{lookup} ); \\
\text{goto:} \\
\text{case: } \text{Result } = 0 \\
\text{break and go on execute} \\
\text{case: } \text{Result } > 0 \\
\text{← CreateNewPath( } \varphi_{input}, \text{Result);} \\
\text{if } \text{UpdateLookup( Addr}_{lookup} ); \\
\text{goto:} \\
\text{Output: Obfuscated P} \\
\]

Function \( \text{CompareFuncSum} \) compares the input \( \varphi_{input} \) and the node summary information \( \varphi_{jn} \) of the current \( jn \) to determine the next execution path, while the target-driven summary information \( \varphi_{mn} \) used to determine if it is allowed to execute the paths. Three types of result value will be returned based on the comparison information of the function in order to determine the two sub-function algorithms which to call.

When the return value is zero, indicating the execution path of the \( jn \) when the input is \( I \) does not drive to the \( tn \), the program execute normally, without any confusion, it can reduce unnecessary operations, improve the algorithm performance.

When the return value exceed zero, indicating that the \( jn \) is not in the same component with the \( tn \), and the distance between component is at least 1, in this case, it executes the algorithm of \( generate \ new \ execution \ path: \ CreateNewPath \).

When the return value is less than zero, indicating that the \( jn \) in the same component with the \( tn \), and the distance between the node is at least 1, in this case it executes the algorithm of \( bypass \ the \ target \ code \ block: \ AvoidExPath \).

When the execution of algorithm is completed, the node summary information of the \( lookup \ table \) needs to be updated instantly to ensure the accuracy of execution paths.

B. The Execution Path Generation

When the return value of the function \( \text{CompareFuncSum} \) exceeds zero, the main algorithm use Algorithm 2 to generate the new execution path. In Algorithm 2, there are two possible situation according to the different return value.

\[
\text{Algorithm 2. The execution path generation} \\
\text{Input: } \varphi_{input} \text{ from input } I \\
\text{Result from CompareFuncSum()} \\
\text{begin} \\
\text{case: } \text{Result } = 1 \\
\text{break and return null} \\
\text{case: } \text{Result } = 2 \\
\varphi_{sn} \leftarrow \text{FindShareNode();} \\
\text{Addr}_{jn} \leftarrow \text{FindNewjn( } \varphi_{input}, \varphi_{sn} \text{ );} \\
\text{goto:} \\
\text{Output: the address of the new jump node} \\
\]

When the return value is 1, indicating there is no \( sn \) between the component of the \( jn \) and the component of the \( tn \), the \( tn \) can not be executed, while the program execute normally.

When the return value is 2, indicating there is \( sn \) between the component of the \( jn \) and the component of the \( tn \). In this condition, it is necessary to determine the constraint condition \( \varphi_{sn} \) which execute to the \( sn \), and determine the address of the \( jn \) jumped according to the constraint condition \( \varphi_{input} \) and \( \varphi_{sn} \). Make sure the node does not exist in the current component and no \( sn \) with the component. Then executing normally from the new \( jn \) and forming a new execution path.

C. The Target Node Bypass

When the return value of function \( \text{CompareFuncSum} \) is less than zero, the main algorithm will use the algorithm 3 to bypass the target code block. In Algorithm 3, according to the different return values, there are two possible execution situations.

\[
\text{Algorithm 3. The target node bypass} \\
\text{Input: } \varphi_{input} \text{ from input } I \\
\]
When the return value is -1, indicating that there is the branch node between the path of current \( jn \) and the \( tn \), and the execution path to other branch nodes without passing through the \( tn \) based on the target-driven information, the program executed normally.

When the return value is -2, indicating the execution path between the current \( jn \) and the \( tn \) is no branch node or has the branch node, but the execution path will pass through the \( tn \) the next node constraint condition \( \varphi_{\text{forward}} \) is calculated by the function \( \text{FindForwardNode} \) use the target-driven information, then determine the jump address of the \( jn \) based on the constraint conditions of \( \varphi_{\text{input}} \) and \( \varphi_{\text{forward}} \), the function of bypass the \( tn \) is achieved by changing path through the \( jn \).

V. EXPERIMENT

A. Experimental Platform

The experimental platform is 2.9GHz Intel Core2 Duo CPU, RAM 4GB, operating system platform is Windows XP SP3, the test program based on Microsoft .NET platform and be written by Visual Studio 2010.

The experimental test for the DDD algorithm needs two tools, Z3 [17] and Pex [18].

Pex is the symbolic execution testing tools developed by Microsoft Corporation which based on the .NET platform, and use the SMT solver Z3 to analysis the program. The current version number is 0.94.51023.0. The tool can automatically generate and create conditions to generate the experimental data for dynamic symbolic execution, and analysis the execution paths as many as possible in the execution process.

Z3 is a solver tool based on the SMT model, developed by Microsoft, the latest version is 4.3.0. With Z3 tool can test the characteristic of the target-driven in the DDD algorithm, and get satisfiability results of the specified nodes, the number of execution paths and the time of execution. And study the algorithm performance through the compare experiment of the original program and obfuscated program.

B. Experimental Program Sets

In order to test the algorithm based on the SMT model, three different test programs were wrote in C#, two of them are operated on strings, one is for integer operations. The program logic can be represented by the symbolic first order logic formula that meets the test model requirement for Z3 and Pex.

The first program is called \( \text{FindString} \), the arbitrary string as input in the program, and compare its contents, if the input string contains both four sub-strings: “code”, "obfuscation", "symbol" and "execution", then the program jumps to a target function to print out "done", otherwise, print out "undone".

The second program is called \( \text{LocateString} \), the program locate the position of the four sub-strings based on the function of \( \text{FindString} \), if the relative four sub-strings "code", "obfuscation", "symbol", "execution" in this position, the program jumps to a target function, and print out "done", otherwise, print out "undone".

The third program is called \( \text{ChangeNumber} \), the program input a random integer as the variable and does linear operation several times, when the result meets the built-in conditions, the program jumps to the target function, and print out "done", otherwise, print out "undone".

C. Experimental Performance

When test the obfuscation performance of the DDD algorithm. The Program Sets will be tested by the symbolic execution test tool Pex. The experimental result is the number of possible execution paths in the original program, the number of possible execution paths in the obfuscated program and the running time of both situations.

The difference of the obfuscated program sets is the number of \( jn \) added, in the experimental test the test program sets are divided by the number of \( jn \): 0, 5, 10, 15, 20, 30, 40, 50 (0 indicates that the test program is not confused and the test result is from the original program).

When the test execution path generated more than 20,000 seconds indicating the number of execution paths dynamically generated after confusion has reached the upper limit of the test, labeled as Max. When the execution time of the program over 30,000 seconds, labeled as Max, means that in order to restore the obfuscated program, the execution price is unacceptable, so it can not correctly restore the original program.

| TABLE I. | THE EXECUTION PATH GENERATION |
| --- | --- | --- | --- |
| No. of jump node | \( \text{FindString} \) | \( \text{LocateString} \) | \( \text{ChangeNumber} \) |
| 0 | 39 | 98 | 46 |
| 5 | 75 | 126 | 98 |
| 10 | 143 | 774 | 684 |
| 20 | 2473 | 3452 | 2831 |
| 30 | 8673 | 9341 | 7754 |
| 40 | 16435 | 17721 | 13864 |
| 50 | Max | Max | Max |

| TABLE II. | THE EXECUTION TIME(S) |
| --- | --- | --- |
| No. of jump node | \( \text{FindString} \) | \( \text{LocateString} \) | \( \text{ChangeNumber} \) |
| 0 | 7 | 12 | 6 |
| 5 | 12 | 20 | 16 |
| 10 | 191 | 433 | 335 |
| 20 | 1879 | 3279 | 2462 |
| 30 | 8933 | 10461 | 8064 |
| 40 | 23830 | 29239 | 18577 |
| 50 | Max | Max | Max |

Fig. 6 and Fig. 7 is generated based on the experimental data of Table I and Table II, separately shown the trend of change about the number of the execution paths and the execution time.
Fig. 6 and Fig. 7 show that with the increase of the number of the \( jn \), the execution paths and the execution time has significantly increased compared to the original program, in particular, when the number of the \( jn \) is over 20, the range of increase is very large.

![Figure 6. The execution path generation](image1)

![Figure 7. The execution time](image2)

With the increase of the number of the \( jn \), it will spend more time when Z3 solving the constraint conditions, so as shown in Fig. 8, the average execution time for each program is increased significantly.

![Figure 8. The average time of execution](image3)

When the number of the \( jn \) reached to 40, the execution results of the confusion program has been unable to reach the \( tn \), the solution result is "undone", indicated the DDD algorithm can protect specific function or information effectively.

VI. CONCLUSION

This paper proposed a novel dynamic obfuscation algorithm DDD-achieve the protection of the information of the \( tn \) by making the path explosions and bypassing the target code block. Locate the position of the current node and determine the jumping way and the address of the new \( jn \) by the summary information of the \( jn \) and symbolic constraint conditions. In the experiment, we use the SMT solver Z3 and symbolic analysis tool Pex to compare the test results between the original program and the obfuscated program. Our test results shows the effectiveness of defending the dynamic symbolic analysis and protecting the specified target information. The experimental data also shows this algorithm still has enough space to improve.

ACKNOWLEDGMENT

This work is supported by National Key Technology R&D Program (2012BAH37B05), National Natural Science Foundation of China (61121061). The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

REFERENCES


Yang Yubo, born in 1986, is currently a PhD candidate in Information Security Center, Beijing University of Posts and Telecommunications, China. He received his bachelor degree from North China University of Technology, China, in 2011. His research interests include information security, code obfuscation.

Fan Wenqing, born in 1983, is currently an instructor in School of Computer Science, Communication University of China. He received his PhD degree from Beijing University of Posts and Telecommunications, China, in 2010. His research interests include information security, program analysis.

Huang Wei, born in 1983, is currently an instructor in School of Computer Science, Communication University of China. He received his PhD degree from Beijing University of Posts and Telecommunications, China, in 2010. His research interests include information security.