Journal of Computational Information Systems 7:5 (2011) 1436-1445
Available at http://www.Jofcis.com

Optimization of Program Behavior Model for Trusted Computing Dynamic Attestation

Fajiang YU[1,2,†], Yuewei XU[3], Yue YU[1], Yang LIN[1], Yaohui WANG[1]

[1] *School of Computer, Wuhan University, Wuhan 430072, China*
[2] *Key Laboratory of Aerospace Information Security and Trusted Computing, Wuhan 430072, China*
[3] *Zhejiang Provincial Testing Institute of Electronic Products, Hangzhou, Zhejiang 310012, China*

Abstract

Program's behavior verification is the core of trusted computing dynamic attestation. The first step of program's behavior verification is building program's trusted behavior model. Static analysis based behavior model building can cover all running paths, but there may be many ε run circuits, which can lead to the failure of dynamic attestation execution. This paper gives out an improved Floyd algorithm to find out and remove one ε run circuit with shortest length. The action is repeated until there is no ε run circuit, which can optimize the trusted behavior model. This paper also carries out one theoretical analysis of the time complexity and space complexity, the optimization method is better than the method based on the traditional algorithm of finding out all elementary circuits in directed graph. Finally, this paper does some optimization experiments about real Windows binary program's behavior model. The result shows there is a large reduction in the number of directed graph's nodes and edges in the behavior model after deleting all ε run circuits.

Keywords: Trusted Computing; Dynamic Attestation; Behavior Model; ε Run Circuit

1. Introduction

Trusted Computing is an information system security solution for the basic computing security problem. It has become one of hot subjects in the information security research field [1, 2]. From the angle of behavior, Trusted Computing Group (TCG) gives us a standard to verify the computing platform whether is trusted or not: "an entity can be trusted if it always behaves in the expected manner for the intended purpose" [3]. The technology which the trusted computing platforms currently adopted does not comply with the requirements that the behaviors are trusted. We need to verify the dynamic behavior of components as well.

Currently, there are few researches on the theory and method of trusted computing dynamic verification. The related researches are focus on extended trusted network connection and remote attestation, trustworthy software, host-based intrusion detection, malicious software detection based on behavior analysis, and etc.

In the aspect of extended trusted network connection and remote attestation, Vivek Haldar et al. have given us a basic framework on the semantic remote attestation for the trustworthiness of a Java program running behavior [4]. Xiaoyong Li et al. have proposed one model of system behavior based trustworthiness attestation for computing platform [5]. Ahmad-Reza Sadeghi et al. have presented a method of property-based attestation for computing platforms [6]. Dengguo Feng et al. have proposed a complete dynamic update attestation scheme for multiple remote attestation instance in trusted computing environment [7]. These researches have improved some disadvantages of TCG trusted network connection, but it does not verify the dynamic behavior of the components.

† Corresponding author.
Email addresses: qshxyu@126.com (Fajiang YU)

Current research on trustworthy software are mainly concentrated on the fields of model building and verification, software trustworthiness design, the analysis and test on software trustworthiness, the evolution of software trustworthiness [8], and etc. The research covers every software lifecycle stage. The trusted computing dynamic verification pays more attention to verify the dynamic behavior.

Since Stephanie Forrest et al. started to use the sequence of Linux system call to describe the program behavior and conduct host-based intrusion detection system [9, 10], many researchers have done a lot of work in this area. But the behavioral characteristic of trusted computing dynamic verification is not limited to the sequence of system call.

To improve the ability of detecting metamorphic viruses and unknown viruses, the detecting method based on behavior analysis has been proposed [11-13]. At present, the main behavioral characteristic for malicious code detection is API, but for trusted computing dynamic verification, it is not limited to API.

The first step of trusted computing dynamic attestation is building the behavior model of all platform components. Generally, there are two methods for model building: dynamic training and static analysis. Dynamic training always faces the classic problem of machine learning. It is hard to build a training set which can cover all possible program execution paths. In order to solve this problem of dynamic training, David Wagner et al. started using static analysis to build a Finite State Automata (FSA) model for the program behavior [14]. Jonathon T. Giffin et al. inserted one “null call” before and after calling one sub function for building one context-sensitive model to guarantee the high efficiency of FSA model and remove impossible paths [15, 16]. Henry Hanping Feng et al. built a stack-deterministic Push down Automata (sDPDA) model to improve the efficiency [17]. Rajeev Gopalakrishna et al. directly embedded the sub functions for building global FSA to improve the efficiency [18]. LI Wen et al. connected every sub function only when the application is running to reduce stack size [19].

These methods which use static analysis to build the FSA program model mainly focus on the characteristic of system call or API. It is regarded as the input alphabet symbol of FSA state transition. But there may be no code of the system call or the API call in many program blocks. This will cause that the generated FSA has a lot of ε transitions. It means that there is no need of inputting any alphabet symbol when the state of FSA shifts. These ε transitions may form ε run circuits in the FSA, which can cause the FSA falls into endless loop and can not stop when it is used to do trusted computing dynamic attestation. So when we build the FSA behavior model for the program, we need to find out and delete ε run circuits, and then remove other non-ε transitions in order to guarantee it executes correctly and efficiently by using this model to do attestation.

This paper mainly makes the following three contributions: (1) We give out an improved Floyd algorithm to find out and delete one shortest ε run circuit. (2) This paper also carries out one theoretical analysis of the time complexity and space complexity, the optimization method is better than the method based on the traditional algorithm of finding out all elementary circuits in directed graph. (3) Finally, this paper does some optimization experiments about real Windows binary program’s behavior model. The result shows there is a large reduction in the number of directed graph’s nodes and edges in the behavior model after deleting all ε run circuits. Either in time efficiency or space efficiency, the optimization method is better than the method based on the traditional algorithm of finding out all elementary circuits in directed graph.

2. Behavior Modeling Based on Static Analysis of Binary Program

Figure 1 is the basic framework of trusted computing dynamic attestation. First we build the model for the trusted behavior of each computing platform components. Then we use this model to verify platform’s real running behavior. The trusted behavior model is the base of trusted computing dynamic attestation. Because dynamic training cannot cover all running paths, the method considered by this paper is to apply static analysis of the binary PE file and build the program trusted behavior model. It is more consistent with the reality that many commercial softwares do not open the source code.

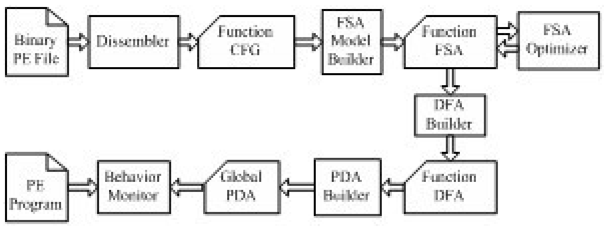


Fig.1 The Framework of Trusted Computing Dynamic Attestation

The process of constructing static analysis-based behavior model includes the following seven stages:

1) CFG building from binary code
2) FSA constructing from CFG
3) Deleting ε run circuit
4) Deleting non-circuit ε transition
5) Constructing Deterministic FSA
6) Deleting empty DFA
7) Building Global PDA

You can see the detail stages and algorithms about building trusted behavior model based on static analysis of PE binary file in our previous work [20].

3. Searching and Deleting ε Run Circuits

The existence of ε run circuits can lead the failure of trust attestation, so we need to find out and delete the ε run circuits of FSA during the process of building trusted behavior model, in order to finish the model optimization.

Definition 1. There is a input symbol sequence on one FSA $M = (Q, \Sigma, \delta, S, F)$, the sequence is $u = \alpha_0\alpha_1 \dots \alpha_{n-1}(\alpha \in \Sigma, i = 0, 1, \dots, n-1)$, whose corresponding state sequence is $q_0q_1 \dots q_{n-1}, q_i, q_{i+1} \in Q, \delta(q_i, \alpha_i) = q_{i+1}$, $i = 0, 1, \dots, n-1$. We call this state sequence is a **run** of FSA M on the input symbols μ, which is denoted as $\mathbb{R}$.

Definition 2. $\mathbb{R}$ is a ε **run**, if $\alpha_i = \varepsilon, i = 0, 1, \dots, n-1$.

Definition 3. $\mathbb{R}$ is a ε **run circuit**, if $\alpha_i = \varepsilon, i = 0, 1, \dots, n-1$ and $q_i \neq q_j$ for $i \neq j$ except that $q_0 = q_n$.

3.1. Judging Whether There are ε Run Circuits

In order to delete ε run circuits in the FSA, we need to judge whether there are ε run circuits or not firstly. The FSA whose all non-ε transitions have been deleted is denoted as $M_\varepsilon = (Q_\varepsilon, \Sigma_\varepsilon, \delta_\varepsilon, S_\varepsilon, F_\varepsilon)$, Where $Q_\varepsilon = Q$, $\Sigma_\varepsilon = \{\varepsilon\}$, $delta_\varepsilon = \{\delta_\varepsilon(u, \varepsilon) = v | \delta(u, \varepsilon) = v \in \delta\}$, $S_\varepsilon = S$, $F_\varepsilon = F$.

FSA M_ε also is a directed graph, which is denoted as $G_\varepsilon = \langle V_\varepsilon, E_\varepsilon \rangle$, where $V_\varepsilon = Q_\varepsilon$, $E_\varepsilon = \{(u, v) | \delta_\varepsilon(u, \varepsilon) = v \in \delta_\varepsilon\}$. We carry out a topological sorting for this graph:

1) Select one state node u_ε from G_ε, which has no precursor;

2) Delete this state node and all the edges whose head node is u_ε, it means $V_\varepsilon = V_\varepsilon - \{u_\varepsilon\}$, $E_\varepsilon = E_\varepsilon - \{(u_\varepsilon, v_\varepsilon) | (u_\varepsilon, v_\varepsilon) \in E_\varepsilon\}$.

We repeat the two steps until $V_\varepsilon = \Phi$ or there is no state node which has no precursor in V_ε.

For the FSA M that has been carried out a topological sorting, if $V_\varepsilon = \Phi$ then there must be no ε run circuit in M; if $V_\varepsilon \neq \Phi$ then there must be some ε run circuits in M, and the ε run circuits must be consisted with the state nodes left in V_ε.

3.2. ε Run Circuits Searching Based on Algorithm of Generating Elementary Circuits in Directed Graph

There are a lot of algorithms of generating elementary circuits in directed graph. This paper uses one algorithm proposed by Wang Yuying et al. [21]. Compared with other algorithms, this algorithm need not search the node which has been discovered, and it is more efficient.

The ε transition FSA directed graph that has been carried out a topological sorting is denoted as $G_T = \langle V_T, E_T \rangle$, where $V_T \neq \Phi$ and there are k state nodes in V_T. There must be ε run circuits in this FSA.

The algorithm of searching ε run circuits in FSA based on the method of generating elementary circuits in directed graph is denoted as **Algorithm 1** (You can its detail in [21].).

The version of notepad.exe in the Windows XP SP1 system for simplified Chinese is 5.1.2600.0.. We delete all non-ε transitions of the FSA (Fig.2), which corresponds to sub-function "sub_1005303" in this notepad.exe, and carry out a topological sorting on it, the directed graph simplified is denoted as G_T.

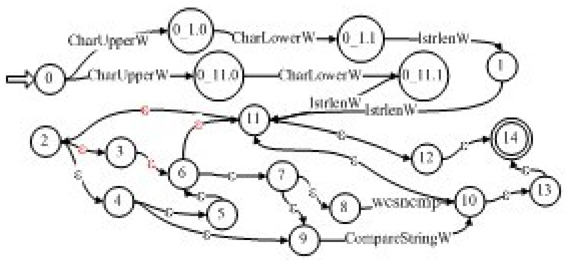


Fig.2 The FSA of One Sub-function "Sub_1005303" in Notepad.exe

We use the algorithm of generating elementary circuits to search ε run circuits in the FSA of sub_1005303. we find out two ε run circuits in sub_1005303 FSA: $11 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 11$ and $11 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 11$.

3.3. Searching ε Run Circuits Based on the Improved Floyd Algorithm

The searching method based on the algorithm of generating elementary circuits will find out all circuits one time. The multiple ε run circuits in FSA may be connected with each other. It is impossible to delete all ε run circuits one time, because after one ε run circuit has been deleted, other circuits may haven been broken and new circuits also may be formed.

In order to increase the time and space efficiency of trusted behavior model optimization, we make some improvement on the Floyd algorithm. We use the improved Floyd algorithm to get the shortest distance between any two state nodes in the ε transition FSA directed graph (G_T) which has been carried out a topological sorting, it means we get the shortest distance matrix of G_T,$(d_{ij})_{k\times k}$. If $d_{ii} = \infty$, for every, $i = 0, 1, \ldots, k-1$ this FSA does not have any ε run circuit which start and end with v_i, otherwise it has the circuit. In order to further increase the efficiency of trusted behavior model optimization, we only find out one shortest ε run circuit firstly and delete it, then repeat this procedure until there is no ε run circuit in the FSA. The length of the shortest ε run circuit is $min(d_{ii}), i = 0, 1, \ldots, k-1$.

The algorithm of searching the start state node (which also is the end state node) in the shortest ε run circuit of FSA based on the improved Floyd algorithm is described as Algorithm 2.

Algorithm 2: Searching the start state node (which also is the end state node) in one shortest ε run circuit of FSA

Require: One ε transition FSA directed graph that has been carried out a topological sorting and is denoted as $G_T = \langle V_T, E_T \rangle$, where $V_T \neq \Phi$ and there are k state nodes in V_T
Ensure: The starting state node of the shortest ε transition circuit in FSA, which is denoted as v_s; the shortest matrix path $(p_{ij})_{k \times k}$

```
1:  for $i = 0$ to $k - 1$ do
2:      for $j = 0$ to $k - 1$ do
3:          if $(v_i, v_j) \in E_T$
4:              $d_{ij} = 1$;
5:          else
6:              $d_{ij} = \infty$;
7:          end if
8:      end for
9: end for
10: for $i = 0$ to $k - 1$ do
11:     for $j = 0$ to $k - 1$ do
12:         $p_{ij} = v_{\text{no}}$;
13:     end for
14: end for
15: for $l = 0$ to $k - 1$ do
16:     for $i = 0$ to $k - 1$ do
17:         for $j = 0$ to $k - 1$ do
18:             if $d_{ij} > d_{il} + d_{lj}$ then
19:                 $d_{ij} = d_{il} + d_{lj}$;
20:                 $p_{ij} = v_l$;
21:             end if
22:         end for
23:     end for
24: end for
25: $d_{\min} = d_{00}$;
26: $v_s = v_{\text{no}}$;
27: for $i = 0$ to $k - 1$ do
28:     if $d_{\min > d_{ii}}$ then
29:         $d_{\min} = d_{ii}$;
30:         $v_s = v_i$;
31:     end if
32: end for
```

After we ascertain the start state node v_s on the shortest ε run circuit, and the shortest path matrix $(p_ij)_{k \times k}$ of FSA, we could find out all nodes on the circuit by using Algorithm 3.

Algorithm 3: $SALLNODESPATH(v_s, v_e)$

Require: start state v_s; final state node v_e; matrix with the shortest distance $(p_{ij})_{k \times k}$
Ensure: a sequence of state nodes on the shortest path, denoted as $s_{v_s v_e}$

```
1:  $s_{v_s v_e} = \phi$
2:  if $p_{se} = v_{\text{no}}$ then
3:  $s_{v_s v_e} = (v_s v_e)$;
4:  return $s_{v_s v_e}$
5:  else
6:      $s_{v_s p_{se}} = SALLNODESPATH(v_s, p_{se})$;
7:      $s_{p_{se} v_e} = SALLNODESPATH(p_{se}, v_e)$;
8:      $s_{v_s v_e} = s_{v_s p_{se}} * s_{p_{se} v_e}$;
9:      return $s_{v_s v_e}$
10: end if
```

We use the improved Floyd algorithm to search the shortest ε run circuit in sub_1005303 FSA. The corresponding shortest ε run circuit is $2 \to 3 \to 6 \to 11 \to 2$.

3.4. Deleting the ε Run Circuit in FSA

After finding out one ε run circuit $q_0 \to q_1 \to \cdots \to q_n \to q_0$ in FSA $M = (Q, \Sigma, \delta, S, F)$, we can delete the ε run circuit with the following steps:

1) Merging all state nodes on ε run circuit into one node, $Q = Q - \{q_0, q_1, \cdots, q_n\} + \{q_{new}\}$;

2) Deleting the ε run between state nodes on ε run circuit,

$$\delta = \delta - \{\delta(q_i, \varepsilon) = q_j | \delta(q_i, \varepsilon) = q_j \in \delta, i, j = 0, 1, \cdots, n\};$$

3) Changing all non-ε runs between state nodes on ε run circuit, all $\delta(q_i, \alpha) = q_j, i, j = 0, 1, \cdots, n$ in δ should be replaced by $\delta(q_{\text{new}}, \alpha) = q_{\text{new}}$;

4) Changing non-ε run from state node on ε run circuit to other state nodes, all $\delta(q_i, \alpha) = q_{\text{other}}$, $i = 0, 1, \cdots, n, q_{\text{other}} \in Q, q_{\text{other}} \notin \{q_0, q_1, \cdots, q_n\}$ in δ should be replaced by$\delta(q_{\text{new}}, \alpha) = q_{\text{other}}$;

5) Changing non-ε run from other state nodes to nodes on ε run circuit, it means replacing all $\delta(q_{\text{other}}, \alpha) = q_i, i = 0, 1, \cdots, n, q_{\text{other}} \in Q, q_{\text{other}} \notin \{q_0, q_1, \cdots, q_n\}$;

6) If $q_i \in S, i = 0, 1, \cdots, n$, then $S = S - \{q_i\} + \{q_{\text{new}}\}$;

7) If $q_i \in F, i = 0, 1, \cdots, n$, then $F = F - \{q_i\} + \{q_{\text{new}}\}$.

4. Performance Analysis

4.1. Time Complexity Analysis

There are mostly two methods to optimize the programs' trusted behavior model. Method 1 is following steps 3.1, 3.2 and 3.4, which is based on traditional algorithm of generating all elementary circuits in directed graph Method 2 is following steps 3.1, 3.3 and 3.4, which is based on the improved Floyd algorithm. (In the rest paper, Method 1 and Method 2 are used to represent the two model optimization methods).

Method 1 has the same first and third steps as Method 2, so the time difference spent is mainly between the algorithm described in 3.2 and 3.3.

According to the document [21], the time that algorithm in 3.2 takes is:

$$T_1(n, L, L_i) = \leqslant k(n \times e \times L)$$

k is a constant, n is the number of nodes in the directed graph after topological sorting, $L_i (i = 1, 2, \ldots, n-1)$ is the number of the circuits whose smallest vertex is i, L is the number of circuits in the graph, expression $Adj(\mu)$ is the adjacency list of μ (e is the number of edges).

The time that algorithm in 3.2 takes can be simplified as: $T_1(n) = O(n^3 2^{2n})$.

For Method 2, first we use Algorithm 2 in 3.3 to find out the start node of the shortest ε run circuit in the directed graph, the time it takes is: $T_{21}(n) = 4n^3 + n^2 + 4n$;

Then we recursively find out all nodes in the circuit by using the algorithm, the taken time is:

$$T_{22}(n) = \begin{cases} 3 & n = 2 \\ T(n-1) + 2 & (n > 2) \end{cases}$$

The calculation result of T_{22} is: $T_{22}(n) = 2n - 1$

The total time that the algorithm in 3.3 takes is:

$$T_2(n) = T_{21}(n) + T_{22}(n) = 4n^3 + n^2 + 6n - 1$$
$$T_2(n) = O(n^3)$$

It is obvious that Method 2 takes less time than Method 1 by comparing T_1 and T_2.

4.2. Space Complexity Analysis

Method 1 has the same first and third steps as Method 2, so the space difference the two methods need is mainly between the algorithm described in 3.2 and 3.3.

Algorithm 1 in 3.2 requires many rounds iterative calculations. And in every round it uses two matrixes,. The two matrixes' elements are some path information. There are two data structures that are used in programs to record the path information of matrix element:

```
typedef struct {                                   typedef struct {
    char IndexPath[STATE_MAX][INDEX_MAX];              stuStateIndexPath * ssipaIns[PATH_MAX];
    int iPathLen;                                      int iPathNum;
} stuStateIndexPath;                               } stuFLPEiWiElement;
```

STATE_MAX, INDEX_MAX, PATH_MAX are constants, which can be different value in different situation. We suppose that the storage space taken by one structure variation of stuStateIndexPath is C_{11}B, and the storage space of stuFLPEiWiElement is C_{12}B.

In every round iterative calculation of the algorithm in 3.2, the path information of all elements in edge matrix and path matrix should be saved. In the first round calculation, edge matrix and path matrix can both have $n \times n$ elements, n is the number of nodes in the directed graph that after topological sorting. The biggest number of path is m among all path information. The worst situation is that all elements have m paths in the first round calculation, and the space it takes is:$M_1(m,n) = 2n^2(mC_{11} + C_{12})$ (B). m, the biggest number of element path, is related to the number of ε run circuit.

There are also two matrixes used in the algorithm of 3.3: the shortest distance matrix and the shortest path matrix, n is the number of nodes in the directed graph after topological sorting. The element of distance matrix is the information of distance length, the space one element takes is C_{21}B. The element of path matrix is the information of nodes, the space one element takes is C_{22}B。The main space taken by the algorithm in 3.3 is: $M_2(n) = n^2(C_{21} + C_{22})$ (B).

Comparing M_1 and M_2, the space taken by structure variation of stuStateIndexPath (C_{11}B) is far greater than the space used to save distance length (C_{12}B), the space of stuFLPEiWiElement (C_{12}B) also is bigger than the space used to save the node information (C_{22}B). Besides, M_1 has relation with the number of ε run circuit (m). Considering that C_{11} is far greater than C_{21}, M_1 can be represented as: $M_1(m,n) \approx 2mn^2C_{11}$ (B).

We can see that, with the increasing of ε run circuit number, the space taken by the algorithm of 3.2 also is increased in multiples, so it is likely that space overflow would happen. But for the algorithm of 3.3, the space it takes has no relation with ε run circuit number.

From the above analysis we can see that the space taken by Method 2 is far smaller than the space of Method 1.

5. Experiment and Analysis

We do our experiment of building the trusted behavior model of program notepad.exe in Windows XP SP2. On our computer for experiment, the CPU is Intel(R) Core(TM)2 Duo CPU T8100 @ 2.10GHz, and the RAM is DDR2 1024MB. At present, behavior model pays most attention to the behavioral feature of API in kernel32.dll. notepad.exe not only directly call the API of kernel32.dll, but call some other dynamic link libraries (DLLs). These DLLs also may call the API of kernel32.dll, and these APIs also are the behavioral feature of notepad.exe.

Building behavior model for notepad.exe also needs building the behavior models of the related DLLs. There are many sub functions in notepad.exe and related DLLs. We choose 8 sub functions from 3 DLLs for building model and optimizing. From the table we can see that after the optimization, all ε run circuits have been deleted, and the number of nodes and edges in function directed graph is greatly reduced.

5.1. Time Efficiency Tests

In order to compare the model optimization time efficiency of traditional algorithm of generating all elementary circuits in directed graph and the improved Floyd algorithm, we optimize the behavior model of every sub function in Table 1 by following the steps 3.1, 3.2(or 3.3) and 3.4, the time they take are seen in Table 1.

The number of directed graph nodes after topological sorting is the result of the first time of doing topological sorting. The number of deleted ε run circuits is equal to the times

From Table 1 we can see when there are only a few ε run circuits, the time taken by Method 1 and Method 2 have little difference; when there are more ε run circuits, the time taken by Method 2 is less than Method 1. Because Method 1 is based on the algorithm of generating elementary circuits in directed graph, every time of following step 3.2, all ε run circuits could be found out, it would take more time if there are more circuits. The experiment result is consistent with our analysis in 4.2. In Table 2 he space overflows when sub function sub_77D3B1DA is being optimized by using Method 1.

Table 1 The Time Taken by Model Optimization of Every Sub-function

Name of Sub-function	Number of Nodes after Topological Sorting	Number of Deleted ε Run Circuit	Time (s)	
			Method 1	Method 2
tree_size_ndr	45	20	7.076274	6.733020
tree_into_ndr	48	22	8.502247	9.021329
data_into_ndr	51	24	10.337455	10.370068
sub_77D5FAD4	81	31	32.868784	33.119412
data_from_ndr	85	33	36.543700	36.862505
sub_77D3B1DA	105	4	overflow	17.867722
sub_77C481F5	101	41	64.731101	61.411450
sub_77D3B6AF	153	60	221.496020	221.496020

5.2. Space Consumption Test

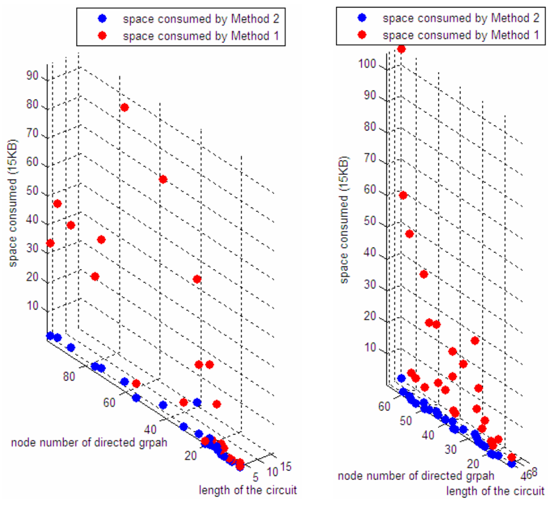
In order to compare the amount of space consumed by Method 1 and Method 2, we optimize the behavior model of every sub function in Table 1 by following the steps 3.1, 3.2 (or 3.3) and 3.4. The peak amount of the space consumed during the behavior model optimization of every sub function is as Table 2 shown.

Table 2 The Peak Amount of Space Consumed during the Behavior Model Optimization of Every Sub-function

Name of Sub-function	the Peak Amount of Space Consumed (KB)	
	Method 1	Method 2
tree_size_ndr	444	4.05
tree_into_ndr	503	4.609
data_into_ndr	601	5.202
sub_77D5FAD4	1577	13.122
data_from_ndr	1751	14.450
sub_77D3B1DA	overflow	22.050
sub_77C481F5	1688	20.402
sub_77D3B6AF	10141	46.818

From Table 2 we can see the more nodes of the directed graph after topological sorting and ε run circuits in the graph is, generally the bigger peak amount of space consumed by Method 1 is. Sub-function “data_from_ndr” has less nodes and ε run circuits than “sub_77C481F5”, but the space it takes up is more than “sub_77C481F5”. The reason is that the amount of space consumed also has relation with the length of ε run circuits. The longer the length is, the more information need be saved, so the more space it takes up. During the optimization of the behavior model for sub function “sub_77D3B1DA”, the temporary paths are so complex that they need occupy large storage and overflow the space when the second ε run circuit is in processing. The amount of space taken up by Method 2 only is related with the node number in the directed graph. And Method 1 takes up great less space than Method 1. The experiment result is consistent with our analysis in 4.3. In order to know more about the relationship between the amount of space consumed and the length of ε run circuit, we record the amount of space consumed by Method 1 and Method 2 when we deal with several ε run circuits of some sub functions, which is shown in Figure 3. Figure 3 (a) is the statistics of some sub functions in gdi32.dll when doing modeling optimization, (b) is the statistics of some sub functions in shlwapi.dll.

From Figure 3 we can see the longer the length of ε run circuit is, the more space Method 1 will take up. But the space consumed by Method 1 is no relation with the length of ε run circuit, only is related to the number of nodes in directed graph. From the experiment we know the amount of space consumed by Method 2 is much less than Method 1.



(a) Gdi32.dll (b) Shlwapi.dll

Fig. 3 Statistics of Space Taken and Length of ε Run Circuit

6. Conclusion and Future Work

Program's behavior verification is the core of trusted computing dynamic attestation. The first step of program's behavior verification is building program's trusted behavior model. Static analysis based behavior model building can cover all running paths, but there may be many ε run circuits, which can lead to the failure of trust attestation. This paper gives out an improved Floyd algorithm to find out and delete one shortest ε run circuit. The action is repeated until there is no ε run circuit, which can optimize the trusted behavior model. This is a new application and improvement of Floyd algorithm.

Compared with the model optimization method based on the traditional algorithm of generating the elementary circuit in directed graph, the optimization method proposed in this paper only need to find out one shortest ε run circuit every time, but the method based on traditional algorithm has to find out all circuits, which must waste more time and space. The optimization method proposed in this paper would need less time and space if there are more nodes and longer ε run circuits in the directed graph of behavior model, which is confirmed by our experiment.

In the future, we still have much work to do. 1) We will use the trusted behavior model to dynamically monitor the programs' real running behavior; 2) We will integrate static analysis and dynamic training to improve the efficiency and accuracy of trusted behavior model building, and consider how to add API arguments into the behavior model; 3) We will apply the method of program slicing, Fuzzing test, symbolic execution, and taint analysis into trusted computing dynamic attestation.

Acknowledgement

This work is supported by the National Natural Science Foundation of China (No. 60970115, 91018008), the Fundamental Research Funds for the Central Universities in China (No. 3101044), and open foundation of Zhejiang Key Laboratory of Information Security in China.

References

[1] Sheng Changxiang, Zhang Huanguo, Wang Huaimin, et al. Research and development of trusted computing. *Science China: Information Science*, 2010, 40(2): 139-166. (in Chinese)

[2] Sheng Changxiang, Zhang Huanguo, Feng Dengguo, et al. Survey of information security. *Science China: Information Science*, 2007, 37(2): 1-22. (in Chinese)

[3] Trusted Computing Group. *TCG specification architecture overview, specification revision 1.4*. http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf, 2010-06-12.

[4] Vivek Haldar, Deepak Chandra, Michael Franz. Semantic remote attestation — a virtual machine directed approach to trusted computing. *Proceedings of the 3rd conference on virtual machine research and technology symposium*, California, USA, USENIX, San Jose, 2004: 29-41.

[5] Li Xiaoyong, Zuo Xiaodong, Shen Changxiang. System behavior based trustworthiness attestation for computing platform. *Acta Electronica Sinica*, 2007, 35(7): 1234-1239. (in Chinese)

[6] Ahmad-Reza Sadeghi, Christian Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. *Proceedings of the 2004 workshop on new security paradigms*, Nova Scotia, Canada, ACM Press, 2004: 66-77.

[7] Feng Dengguo, Qin Yu. Research on attestation method for trusted computing environment. *Chinese Journal of Computers*, 2008, 31(9): 1640-1652. (in Chinese)

[8] Wang Huaimin. Building Integration environment of trustworthy software. *Communications of China Computer Federation*, 2009, 5(2): 56-61. (in Chinese)

[9] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, et al. A sense of self for unix processes. *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Los Alamitos, California, USA, IEEE Computer Society, 1996: 120-128.

[10] Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998, 6(3):151-180.

[11] Chen Jianmin, Shu Hui, Xiong Xiaobing. Fuzzing test approach based on symbolic execution. *Computer Engineering*, 2009, 35(21): 33-35. (in Chinese)

[12] Chen Kai, Feng Dengguo, Su Purui. Exploring multiple execution paths based on dynamic lazy analysis. *Chinese Journal of Computers*, 2010, 33(3): 493-503. (in Chinese)

[13] Li Jiajing, Wang Tielei, Wei Tao, et al. A polynomial time path-sensitive taint analysis method. *Chinese Journal of Computers*, 2009, 32(9): 1845-1855. (in Chinese)

[14] David Wagner, Drew Dean. Intrusion detection via static analysis. *Proceedings of 2001 IEEE Symposium on Security and Privacy*, Oakland, California, USA, IEEE Computer Society Press, 2001: 156-168.

[15] Jonathon T. Giffin, Somesh Jha, Barton P. Miller. Detecting manipulated remote call streams. *Proceedings of the 11th USENIX Security Symposium*, Berkeley, California, USA, USENIX Association, 2002: 61-79.

[16] Jonathon T. Giffin, David Dagon, Somesh Jha, et al. Environment-sensitive intrusion detection. *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, Seattle, Washington, USA, LNCS3858, Springer-Verlag, 2005: 185-206.

[17] Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, et al. Formalizing sensitivity in static analysis for intrusion detection. *Proceedings of 2004 IEEE Symposium on Security and Privacy*. Oakland, California, USA, IEEE Computer Society Press, 2004: 194-208.

[18] Rajeev Gopalakrishna, Eugene H. Spafford, Jan Vitek. Efficient intrusion detection using automaton inlining. *Proceedings of 2005 IEEE Symposium on Security and Privacy*, Oakland, California, USA, IEEE Computer Society Press, 2005: 18-31.

[19] Li Wen, Dai Yingxia, Lian Yifeng, et al. Context sensitive host-based IDS using hybrid automaton. *Journal of Software*, 2009, 20(1): 138-151. (in Chinese)

[20] Yu Fajiang, Yu Yue. Static analysis-based behavior model building for trusted computing dynamic verification. *Wuhan University Journal of Natural Sciences*, 2010, 15(3): 195-200.

[21] Wang Yuying, Chen Ping, Su Yang. An efficient algorithm of generating all elementary circuits in directed graph. *Computer Application and Software*, 2009, 26(12): 27-29. (in Chinese)