Wuhan University
Journal of Natural Sciences

2010, Vol.15 No.3, 195-200

Article ID 1007-1202(2010)03-0195-06
DOI 10.1007/s11859-010-0303-9

Static Analysis-Based Behavior Model Building for Trusted Computing Dynamic Verification

□ YU Fajiang[1,2], YU Yue[1]

1. School of Computer, Wuhan University, Wuhan 430072, Hubei, China;

2. Key Laboratory of Aerospace Information Security and Trusted Computing of Ministry of Education, Wuhan University, Wuhan 430072, Hubei, China

© Wuhan University and Springer-Verlag Berlin Heidelberg 2010

Abstract: Current trusted computing platform only verifies application's static Hash value, it could not prevent application from being dynamic attacked. This paper gives one static analysis-based behavior model building method for trusted computing dynamic verification, including control flow graph (CFG) building, finite state automata (FSA) constructing, ε run cycle removing, ε transition removing, deterministic finite state (DFA) constructing, trivial FSA removing, and global push down automata (PDA) constructing. According to experiment, this model built is a reduced model for dynamic verification and covers all possible paths, because it is based on binary file static analysis.

Key words: trusted computing; dynamic verification; behavior model; finite-state automata (FSA); push down automata (PDA)
CLC number: TP 391

Received date: 2009-12-10
Foundation item: Supported by the National High Technology Research and Development Program of China (863 Program) (2006AA01Z442, 2007AA01Z411) the National Natural Science Foundation of China (60673071, 60970115), and Open Foundation of State Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education in China (AISTC2008Q03)
Biography: YU Fajiang, male, Ph.D., research direction: information security, trusted computing. E-mail: qshxyu@126.com

0 Introduction

In recent years, great progress has been made in research on trusted platform module (TPM), trusted computing platform, trusted computing platform evaluation, trusted software, trusted network connect, remote attestation and trusted computing application[1-3]. Current technologies can ensure that the characteristic codes and configure data integrity of trusted computing platform's component are same as the expected integrity[4] but cannot ensure the behavior is trusted[5]. Presently, there has been little research on theory and technology on trusted computing dynamic verification, and the related work is mainly in host-based intrusion detection about system call.

Wagner *et al*[6] were the first to propose the use of static analysis for intrusion detection. However, they analyzed the source code for construct application mode, and this cannot be assumed that the availability of source code for analysis on commercial trusted computing platform. Giffin *et al*[7,8] inserted one "null_call" before and after calling one subfunction and gave one unique call name for building one context-sensitive model to remove impossible path. Feng *et al*[9] built stack-deterministic push down automata (PDA) to improve performance. Gopalakrishna *et al*[10] directly embedded subfunction for building global finite-state automata (FSA) to improve performance. LI Wen *et al*[11] connected every subfunction only when the application is running to reduce stack size. All forementioned research is about server- based intrusion detection, which generally aimed at one given network service application at Linux or Unix system. Our

method mainly is for building Win32 API behavior model for trusted computing dynamic verification.

This paper makes the following main contributions: ① proposing one method of trusted computing dynamic behaviors verification, ② implementation at Windows terminal, and ③ static analysis-based behavior model optimization in detail.

The rest of this paper is organized as follows: Section 1 gives a detail description of static analysis-based trusted computing dynamic behavior verification. Section 2 evaluates our method by experiment, and we conclude in Section 3 with description of future work.

1 Behavior Model Building

Construction of static analysis-based behavior model progresses in follow several stages: ① building up control flow graph (CFG) from program binary code, ② constructing FSA from CFG, ③ finding and removing ε run cycles in FSA, ④ removing ε transition in FSA, ⑤ constructing deterministic FSA from nondeterministic FSA, ⑥ removing trivial FSA, and ⑦ constructing global PDA.

1.1 CFG Building from Binary Code

Because we target commercial trusted computing platform, only program with binary code is available for us. We use an IDA (interactive disassembler) plug-in, named "wingraph32", to generate CFG for every subfunction of PE file. Node of CFG contains linear sequences of instructions; edge between nodes represents control flow. Figure 1 is the CFG of one subfunction "sub_10048C8" from dissembling notepad.exe.

CFG also can be represented by $G=\langle V,E\rangle$, where V is a finite set, and element of V is vertex $v\in V$, which is a linear sequences of instructions. E is a subset of $E(V)$, $E(V)=\{(u,v)\,|\,u,v\in V\}$, $E\subseteq E(V)$, element of E is an edge, each edge is a pair of vertices. CFG is a directed graph, each edge is an ordered pair, and it has a head vertex and a tail vertex. In CFG, $(u,v)\neq(v,u)$, which is drawn as an arrow in the diagraph.

1.2 FSA Constructing from CFG

Original vertex in CFG is a linear sequence of instructions. Our trusted computing dynamic verification is implemented at Windows terminal, and the behavior model is composed of Kernel32.dll's Win32 API sequences. Here, we only care instructions like Win32 API or other subfunction "call", because it is possible that other subfunction also has Win32 API call instruction. Therefore, after CFG building up, we should filter some

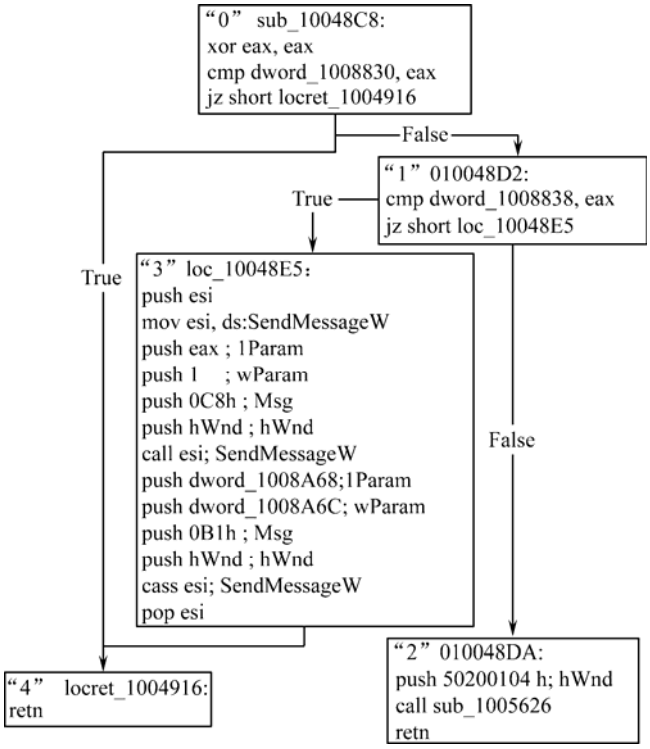

Fig. 1 CFG of sub_10048C8 in notepad.exe

useless instructions in each vertex. Figure 2 shows the CFG of sub_10048C8 after being filtered.

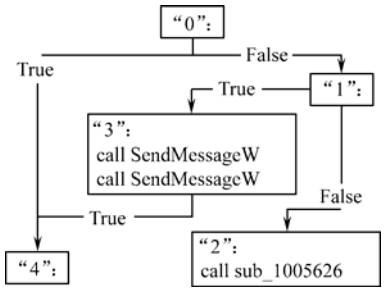

Fig. 2 CFG sub_10048C8 after filtered, exit node "2" has a call instruction

However, there is another special case. Some CFG exit nodes have some instructions like Win32 API or other subfunction call after being filtered; such has vertex "2" in Fig. 2. In this case, for translating CFG into an FSA, one new node must be added as an exit.

After filtering (some CFG maybe need a new exit node added), original CFG $G=\langle V,E\rangle$ is changed as $G'=\langle V',E'\rangle$, $E'=E$, and every $v'\in V'$ may be one type of the following vertices:

① Vertex v' does not have any instruction.

② Vertex v' has only one instruction like Win32 API or other subfunction call.

③ Vertex v' has multiinstructions like Win32 API or other subfunction call.

If we want to construct FSA from CFG, the vertex of type ③ requires some pretreatment, because in every FSA-state transition, there could not be multiinput symbols. Figure 3 shows the result after a new exit node is added, and vertex "3" is pretreated.

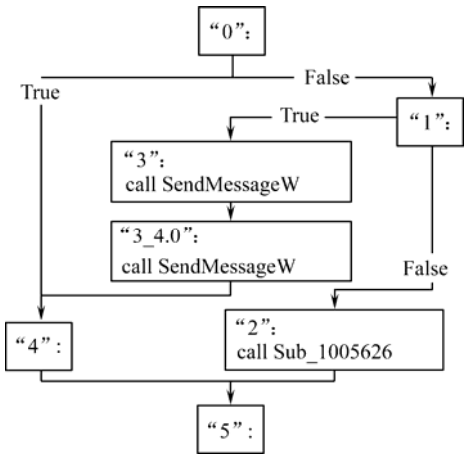

Fig. 3 CFG of sub_10048C8 after pretreatment

A new exit node has been added, original vertex "3" has been divided into two vertices, new "3" and vertex "3_4.0", each vertex has only one instruction like call

After the pretreatment, we could translate the CFG G'' into an FSA $M=(Q,\Sigma,\delta,S,F)$ by using Algorithm 1.

Algorithm 1 FSA constructing algorithm from CFG

a) Finite set of states $Q=V''$

b) Finite input alphabet

$\Sigma=\{$sub Fun or APIName$|\,v''$ has one instruction like "call", $v''\in V''\}+\{\varepsilon\}$

c) Transition Function $\delta:=Q\times\Sigma\to Q$

$\delta=\{\,\delta(u''$, sub fun name or APIname$\}=v''\,|\,(u'',v'')\in E'',u''$ is vertex of type 2$\}+\{\delta(u'',\varepsilon)=v''\,|\,(u'',v'')\in E'',u''$ is vertex of type 1$\}$

d) Initial States $S=\{v''\,|\,v''$ is CFG entry$\}$

e) Final States $F=\{v''\,|\,v''$ is CFG exit$\}$

By using this algorithm, FSA constructed from CFG is shown in Fig. 4.

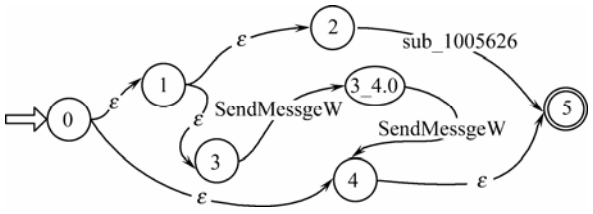

Fig. 4 FSA constructed from CFG of "sub_10048C8"

1.3 ε Run Cycle Finding and Removing

Definition 1 There is an FSA $M=(Q,\Sigma,\delta,S,F)$. There is also a run of FSA M on

$$u=a_0a_1\cdots a_{n-1}(a_i\in\Sigma,i=0,1,\cdots,n-1),$$

which is of a sequence state

$$q_0q_1\cdots q_n(q_i,q_{i+1}\in Q,\delta(q_i,a_i)=q_{i+1},i=0,1,\cdots,n-1),$$

noted as R.

R is a ε run, if $a_i=\varepsilon,i=0,1,\cdots,n-1$;

R is a ε run cycle, if $a_i=\varepsilon,i=0,1,\cdots,n-1$ and $q_i\neq q_j$ for $i\neq j$ except that $q_0=q_n$.

Definition 2 If there is one state $q\in Q,\delta(q,\varepsilon)=q\in\delta$, then q is a self ε run cycle state.

When using the FSA model to do dynamic verification for a given Win32 API sequence, if there is a ε run cycle, the FSA may never stop. Therefore, ε run cycle must be removed; first, we should judge whether there is a ε run cycle in a FSA and then find the ε run cycle.

After making sure that there are ε run cycles in one FSA ε run, we only need to find one ε run cycle first and need not to find out all ε run cycles. Because if one ε run cycle was removed, many other related cycles would disappear. We use the improved Floyd algorithm (Algorithm 2) to compute the minimal ε run cycle state number.

Algorithm 2 Improved Floyd algorithm for computing minimal ε run cycle state number.

```
For i from 0 to ε run state number
    For j from 0 to ε run state number
        If there has δ(q_i, ε) = q_j ∈ δ then Dist(i, j) = 1
        Else Dist(i, j) = ε run state number
MinCycleLen = ε run state number + 1
For k from 0 to ε run state number {
    For i from 0 to k
        If MinCycleLen > Dist(i,k) + Dist(k,i) {
            MinCycleLen = Dist(i,k) + Dist(k,i);
            Start = i ;
            End = k ;
        }
    For i from 0 to ε run state Number
        For j from 0 to ε run state Number
            If Dist(i, j) > Dist(i,k) + Dist(k, j) then
                Dist(i, j) = Dist(i,k) + Dist(k, j)
}
```

After Algorithm 2 has been used, the minimal ε run cycle is composed of ε run $q_{\text{Start}}\cdots q_{\text{End}}$ and $q_{\text{End}}\cdots q_{\text{Start}}$. Then, we use width-first searching method to find out the two ε runs. In notepad.exe's "sub_1003C5B", we find one ε run cycle, "45674", which is illustrated in Fig. 5. After removing ε run cycle "45674" in Fig. 5, the result is shown in Fig. 6.

1.4 ε Transition Removing

After ε run cycle is removed removing, there are still some ε transitions in one subfunction FSA. Then,

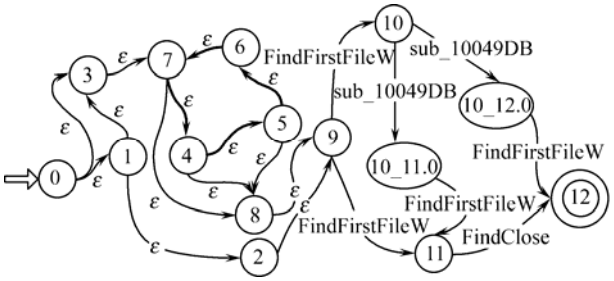

Fig. 5 FSA of "sub_1003C5B" in notepad.exe

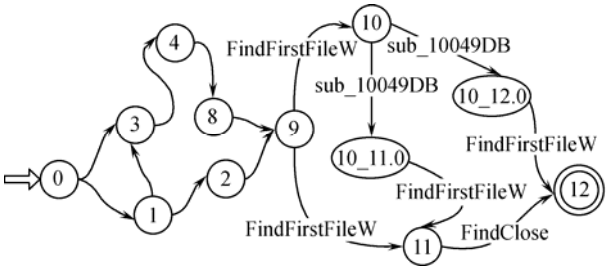

Fig. 6 FSA of "sub_1003C5B" in notepad.exe after ε run cycle being removed

when using the FSA model to do dynamic verification, there would be some ε runs, which could waste verification time. Therefore, we should remove the ε transitions. The result of ε striations in Fig. 6 is shown in Fig. 7.

1.5 Constructing Deterministic FSA

If one FSA $M=(Q,\Sigma,\delta,S,F)$ has multiinitial states or there is more than one possible next state for one state and input symbol, then this FSA is a nondeterministic FSA (NFA). Otherwise, it is a deterministic FSA (DFA). In NFA, there may be multitransitions:

$$\delta(q_i,a)=q_j\in\delta,\delta(q_i,a)=q_k\in\delta,q_j\neq q_k$$

FSA is one NFA, states "0", "1", "2", "3", "4", "8", and "9", all are initial states, and they also have two next states "10", and "11" for the same input symbol "FindFirstFileW" (Fig. 7).

If we directly use NFA to do dynamic verification, for recognizing the same Win32 API sequence, NFA

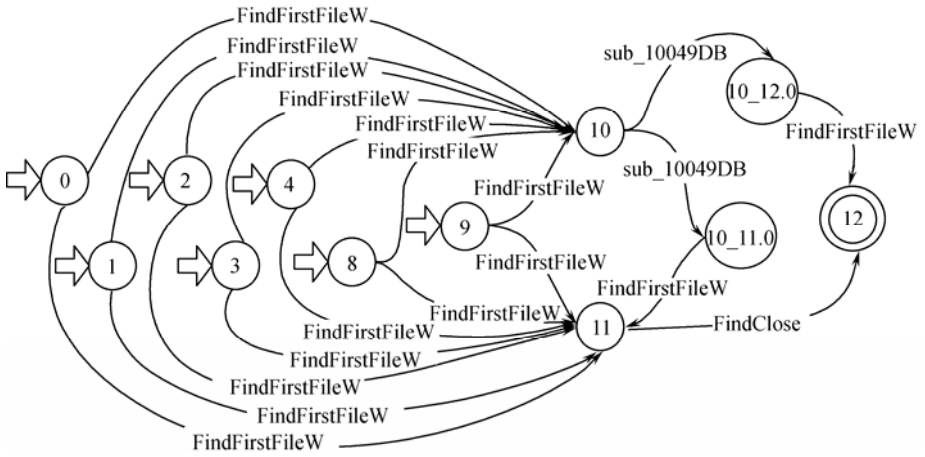

Fig. 7 FSA of "sub_1003C5B" in notepad.exe after ε transitions being removed

needs to make more run attempt. For one NFA, there must be one equivalent DFA. $M'=(Q',\Sigma',\delta',q_0',F')$. By using a corresponding algorithm to construct DFA from NFA in Fig. 7, the result is shown in Fig. 8.

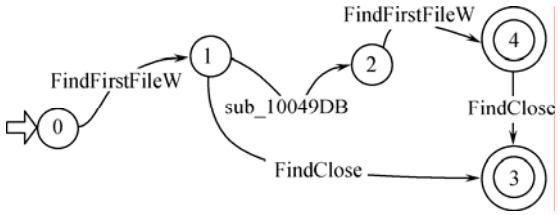

Fig. 8 DFA of "sub_1003C5B" in notepad.exe constructed from NFA

1.6 Trivial FSA Removing

Definition 1. There is an FSA $M=(Q,\Sigma,\delta,S,F)$. If ① it has only one vertex $Q=\{q\}$, which is an initial state, also is a final state, ② for the transition function set $\delta=\varnothing$; then M is a trivial FSA.

In notepad.exe's all original subfunctions FSA, there are 18 trivial FSAs. Because in one trivial FSA, there is not any transition, and dynamic verification is useless in recognizing Win32 API sequences.

1.7 Constructing Global PDA

Our dynamic verification is for one application, we must construct one global DFA from all local subfunction DFAs. Because one subfunction may be called in multipositions, for the subfunction to return correctly, the global DFA must be a global PDA; otherwise, global DFA exists in an "impossible path"[6,7]. We use Algorithm 3 to construct global PDA.

Algorithm 3 Algorithm for constructing global PDA

Global PDA $M=\{Q,\Sigma,\Gamma,\delta,q_0,F,Z_0\}$

$Q=\{q_{\text{start0}}\}, \Sigma=\Phi, \Gamma=\Phi, \delta=\varphi, q_0=q_{\text{start0}}, F=F_{\text{start}}, Z_0=\Phi$;
For every local DFA $M_L=\{Q_L, \Sigma_L, \delta_L, q_{L0}, F_L\}$
For every $\delta_L(q_L, a_L)=q'_L \in \delta_L$
If a_L is a subfunction name, then {
The Corresponding DFA is $M_s=\{Q_s, \Sigma_s, \delta_s, q_{s0}, F\}$;
$\delta=\delta+\{\delta(q_L, \varepsilon, S)=(q_{s0}, a_L S)\}$;
$Q=Q+\{q_L, q_{s0}\}$; $\Gamma=\Gamma+\{a_L\}$
For every $q_{sf} \in F_s$ {
$\delta=\delta+\{\delta(q_{sf}, \varepsilon, a_L S)=(q'_L{}', S)\}$;
$Q=Q+\{q_{sf}, q'_L\}$
}
}
else {
$\delta=\delta+\{\delta(q_L, a_L, S)=(q'_L, S)\}$;
$Q=Q+\{q_L, q'_L\}$; $\Sigma=\Sigma+\{a_L\}$;
}

Figure 9 shows us one "global" PDA constructed from two local DFAs by using this algorithm.

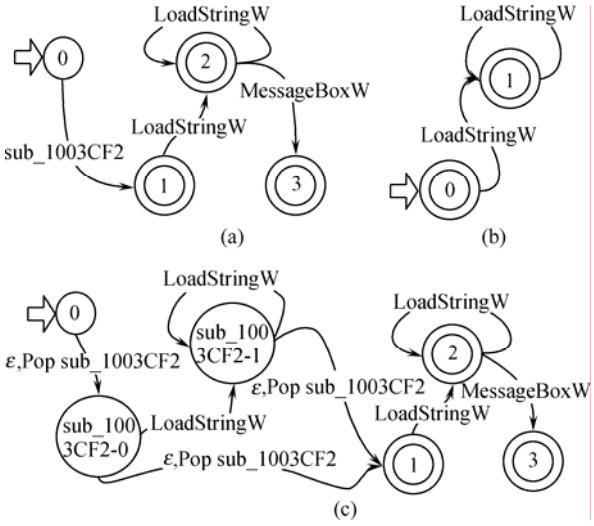

Fig. 9 One global PDA constructed from two local DFAs

(a) Local DFA of "sub_1003D57" in notepad.exe; (b) Local DFA of "sub_1003CF2" in notepad.exe; (c) "Global" PDA constructed from "sub_1003D57" and "1003CF2"

The global constructed from static analysis-based PE binary file is our behavior model for trusted computing dynamic verification.

2 Experiment

We evaluate our method for notepad.exe with Windows XP SP 1 and build its behavior model by binary file static analysis. At our Windows XP SP 1 terminal, notepad.exe version is 5.1.2600.0 and for zh-CN. We use Microsoft Visual Studio Tool "DEPENDS.EXE" to show the minimum set of files, which are required to run notepad.exe, such as DLL file. Figure 10 shows the DLL files and Driver files needed by notepad.exe and their hierarchy. In our current behavior model, we only care Win32 API exported from Kernel32.dll, so we do not analysis Kernel32.dll self.

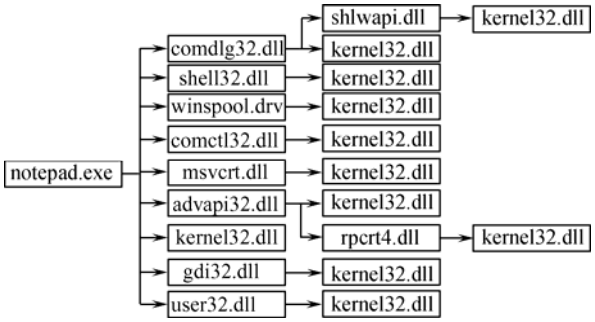

Fig. 10 Notepad.exe dependent DLL and driver files

First, we use IDA to generate a subfunction CFG file (*.gdl) for notepad.exe and its all dependent files, every PE file's size, and its subfunction number is shown in Table 1. Second, we construct one FSA from every subfunction CFG, remove ε run cycles, and remove ε transitions for every FSA. Third, we construct one DFA from every NFA and remove all trivial FSAs for every PE file, and the trivial FSA number is shown in Table 1. Finally, we construct one global PDA from all local FSAs.

Table 1 Behavior model information of notepad.exe and its dependent files

PE file name	Size / byte	Subfunction number	Trivial FSA number
notepad.exe	66 048	83	18
comdlge32.dll	250 880	707	173
shlwapi.dll	400 896	1 931	944
shell32.dll	8 194 048	13 681	5 298
winspool.drv	131 584	758	351
comctl32.dll	557 056	1 909	457
msvcrt.dll	323 072	1 568	649
advapi32.dll	615 424	1 897	949
rpcrt4.dll	530 432	3 466	1 777
gdi32.dll	250 368	1 514	798
user32.dll	556 544	2 061	1 117

From the experiment, we can see the trivial FSA number is about 50% of original sub function number. In fact, the state and transition number of every remainder "untrivial" FSA also are much less than their original

CFG nodes and edge numbers. Therefore, the static analysis-based behavior model is a reduced model for trusted dynamic verification, and this can help us judge easily whether the real application behavior is right.

3 Conclusion and Future Work

This paper gives the method of building behavior model for trusted computing dynamic verification based on binary file static analysis. We describe the building stages in detail, including CFG building, FSA constructing, ε run cycle removing, ε transition removing, DFA constructing, trivial FSA removing, and global PDA constructing. We give an improved Floyd algorithm to find and remove ε run cycles in one subfunction local FSA, this can stop FSA ε run in dynamic verification for recognizing Win32 API sequences. The original FSA model from binary source file is too large, according the experiment; our static analysis-based behavior model is a reduced model.

In the future, we still have much work to do. ① We will build static-analysis based behavior model for more Windows applications, such as internet explorer, office tools, etc. ② We will use the built behavior model to recognize real application's Win32 API sequence behavior, of which the result is used to judge whether the application has been hacked. Because Windows is a commercial operating system, its kernel schemes are not open for us, it will be very hard to integrate building static behavior model and really monitoring dynamic behavior trace. ③ Current behavior model only considered Win32 API name sequence but did not consider API parameters. However, one destroyed application may have the same API sequence, and only some arguments are different. We will consider API arguments and add them into current model. ④ Web security is one research hotspot; we will use this paper's method to do some research about web security. ⑤ Current static analysis-based behavior model is still too large; we will build one more Hash result. This is a great challenge for us, reduced model, which may be similar to that of one message.

References

[1] Shen Changxiang, Zhang Huanguo, Wang Huaimin, *et al*. Trusted computing research and development [J]. *Science China*: *Information Sciences*, 2010, **40**(2): 139-166 (Ch).

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

[3] Zhang Huanguo, Luo Jie, Jin Gang, *et al*. Development of trusted computing research [J]. *Wuhan University Journal of Natural Sciences*, 2006, **11**(6): 1407-1413.

[4] Trusted Computing Group. *TCG Specification Architecture Overview* Specification Revision 1.4 [EB/OL]. [2010-03-10]. *http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf*

[5] Trusted Computing Group. TCG Design, Implementation, and Usage Principles Version 2.0 [EB/OL]. [2009-12-16]. *http://www. trustedcomputinggroup.org/files/resource_files/59C26ECB-1D09-3519-AD469EA7AFBD2E91/Best_Practices_Principles_Document_V2_0.pdf*

[6] Wagner D, Dean D. Intrusion detection via static analysis[C]//*Proceedings of* 2001 *IEEE Symposium on Security and Privacy*, Oakland: IEEE Computer Society, 2001: 156-168.

[7] Giffin J T, Jha S, Miller B P. Detecting manipulated remote call streams [C]//*Proceedings of the* 11*th USENIX Security Symposium*. San Francisco: USENIX Association, 2002: 61-79.

[8] Giffin J T, Dagon D, Jha S. Environment- sensitive intrusion detection [C]//*Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection* (*RAID* 2005) *LNCS*3858. Seattle: Springer-Verlag, 2005: 185-206.

[9] Feng H H, Giffin J, Huang Y, Jha S, *et al.* Formalizing sensitivity in static analysis for intrusion detection [C]//*Proceedings of* 2004 *IEEE Symposium on Security and Privacy*. Oakland: IEEE Computer Society, 2004: 194-208.

[10] Gopalakrishna R, Spafford E, Vitek J. Efficient intrusion detection using automaton inlining [C]//*Proceedings of* 2005 *IEEE Symposium on Security and Privacy*. Oakland: IEEE Computer Society, 2005: 18-31.

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

□