Software detection method based on running state

Xiaolin Zhao¹,a, Quanbao Chen¹,b, Chun Shan¹,c, Ting Wang¹,d, Yiman Zhang¹,e
¹School of Software, Beijing Institute of Technology, Beijing 100081, China
aEmail:zhaoxl@bit.edu.cn,bkenybens@sina.com,csherryshan@bit.edu.cn,dmobao10050@163.com,emobao10050@163.com

Abstract. It is extremely important for the study of software behavior modeling for software security research. This article determines whether the security of software operation by analyzing the credibility of software behavior, monitoring software running status. This paper focuses on a modeling algorithm, namely the GK-tail algorithm, which based on software behavior modeling method. At the same time, this paper improves the GK-tail algorithm, which focuses on data constraints and the interaction between software components. Restrictions on extending finite automaton can be obtained by using a combination of Daikon and ESC/JAVA tools. Restrictions can improve the accuracy of the generated model. So the generated behavior model can capture more accurate information. Finally, the paper designs and implements the software running state generator. It is feasible through the software state diagram to determine the feasibility of software security proved by the experiment.

1. Introduction
In recent years, with the rapid development of China's information industry, information technology has begun to be used in China's many other fields. It is an important issue how to protect the quality of the program.

It is an effective way to determine whether the software has been modified that analysis of software behaviour, monitoring software running status. The ultimate goal of behavioral analysis to obtain the semantic description and understanding of behavior by analyzing the behavioral feature data. Construction and detection is a key step to achieve the above goal. Dynamic analysis techniques are widely used to generate models that support test and verification software systems. The dynamic model of software behavior is used to dynamically detect anomalous behavior, static and dynamic checking of software components, validation protocols, generation of test cases, acquisition of exception event sequences, and validation of program attributes. The main technology of dynamic modeling of software behavior focuses on the constraints of data values or the interaction between software components. Some techniques that focus on data values are modeled as Boolean expressions, such as Daikon; some techniques that focus on component interaction are modeled as finite state machines [1].

This paper discovers it that software behavior modeling has an important application in software analysis, testing, and validation through the study of software security. The innovation of this research is to apply the software behavior modeling based on finite automata to the software security, and to improve the GK-tail algorithm so that the generated behavior model can capture more accurate information.

2. The Credibility of Software Behavior
At present, there are many views on the definition of credibility in academia and industry. The Trusted Computing Group (TCG) [2] defines credible with the expected nature of the entity's behaviour. If an entity is credible, then its behaviour is always in the expected way to achieve the desired goal. From this definition, trusted behaviour is a sufficient condition for a trusted entity. The key to a trusted behaviour
is that the act should occur as expected and complete the intended function. IEEE Pacific Rim International Symposium on Dependable Computing (PDRC) and our scientists [3] have developed a credible computing perspective developed by fault-tolerant computing. They believe that a trusted computing system is a computer system that provides reliability, availability, information, and behavioural security which view on research from a reliable and secure perspective. Combining the above two perspectives, trusted computing is a measure of the credibility of the software from the perspective of the expected and behavioural security of software behaviour. Although the industries trusted computing platform is not enough to ensure the credibility of the system's life cycle. It is correct to determine the credibility of the software that the basic point of view according to the expected behaviour of the software. There is a lot of research is from this point of view [4-5].

2.1. Definition of Trusted Computing
The expectation of software behaviour is embodied in the behaviour feature, behaviour input and output, behaviour process, behaviour attribute and so on. It is about the software behaviour characteristics of knowledge [6]. Whether the software behaviour in these areas conform to the compliance agreement, whether it is acceptable for the user.

First, we give a general definition of credible of software behaviour:

All acts or traces of conduct are expected behaviour for a given software. It is said that the behaviour of the software is credible, or that the software to meet the behaviour of the trusted requirements.

![Diagram of trusted computing](image)

**Figure 1.** The credible behaviour definition can be visually represented. The behaviour of the trusted software $v^t$ must conform to the trusted policy $P^t$; the verified untrusted software necessarily $v^u$ has a behaviour that does not conform to the trusted policy.

2.2. The Importance of Trusted Computing
At present, trusted computing has become one of the research hotspots in information security field. Trusted computing technology and products continue to emerge [5]. People are increasingly aware of the importance and necessity of trusted computing technology. However, the development of trusted computing still has some problems. First, theoretical research lags behind technology development. There is no universally accepted credible computing theory model so far. And there is no software dynamic credibility of the measurement methods and theory; second, the current credibility of the measurement can only ensure that the static integrity of the system resources in the system boot. It does not ensure that the dynamic credibility of runtime system [7].

Wagner and Dean build a software behaviour model to create a Finite State Automaton (FSA) by statically analyzing the source code. But the FSA does not extract the system context information, leading to an impossible path problem. GoPalakrishna et al. [8] proposed a Push-down Automaton (PDA) model that uses additional stack save function call information. The algorithm reduces the uncertainty of the automaton by inserting an empty call before and after the program call and rewriting the executable file to solve impossible path problem. But the operating cost is too high. Feng et al [9] dynamically monitors the program's audit data, captures the control flow of the program by monitoring the system call sequence and call stack information when the program is executed. They establish VtPath between the two system calls for dynamic analysis to fix an impossible path problem to a certain
extent. Liu et al. [10] extended the PDA model to combine the dynamic learning with the static analysis method. The constructed HPDA model has a stronger detection capability than the static method alone, which is lower than the dynamic method alone false alarm rate. Li Wen et al [11] proposed static analysis - dynamic binding of the hybrid model (HFA), which belongs to the combination of static analysis and dynamic learning method. But the model does not take into account the parameters of the system call. It can’t effectively detect those who simulate the normal program behaviour for evading monitoring and detection of mimic attack behaviour. Frossi et al. [12] used the enhanced system call model for anomaly detection by combining the Finite State Automaton (FSA) with the Markov stochastic model.

Giffin et al. [13] pointed out that there would be a problem that the parameter values were insensitive to the context of the system call. The problem existed when the same set of parameters were obtained along the different executable paths, regardless of the context. This problem leads to the program loses the relationship between the system call parameter value and the executable path which uses the parameter value. An attacker can use the parameter values obtained from executable path A to perform the same system call on path B. If an executable path can’t get the parameter value statically, all parameter values must be discarded from other executable paths for security. Therefore, it needs to be considered that the system called parameters in the context of the system call.

It is the easiest way to capture the current system call context that traversing the stack. But the cost of traversing the stack is too high. Spivey [14] proposed to dynamically create a call context tree (CCT), through the monitoring program in the CCT location to conduct a trusted evaluation. The shortcomings of the program runtime time are increased by 2 to 4 times the cost of most of the system It is unacceptable. Bond et al. [15] proposed a probabilistic call context model (PPC), which also reduces evaluation overhead. But its iterative method of calculating PCC values does not apply to multiple repetitions of the same function. Mutz et al. [16] proposed to create a set of normal parameter values for each system call. For each implementation path of the software, a set of parameters is specified for a system call. But the model is less accurate.

System invocation has its special status in software development. Most of the researches on software behaviour are focused on the system call sequence and its related information when the software is running. Forrest et al. proposed to use the system call short sequence to characterize the process. They monitor the privileged process of the system call sequence to establish the intrusion detection model. Yao Lihong et al [17] proposed CTBIDS detection model which using the system call feature tree description program behaviour characteristics. They discriminated program invasion through the abnormal finite accumulation or Hamming distance. Tan Xiaobin et al. [18] used the hidden Markov model to describe the regularity of local system calls during the normal operation of privileged processes.

Through the study of the trusted computing of software behaviour, it is found that there are many shortcomings in the feasible calculation of software behaviour, such as software behavior modeling ability, evaluation ability, evaluation accuracy, system practicality and so on. It still needs to be further improved.

The paper uses GK-tail algorithm to represent the software behaviour model. It is the key point to study of software behaviour. The paper will analysis and improvement of the GK-tail algorithm to get the better the software behaviour modeling.

3. Analysis and Improvement of GK - tail Algorithm

When describing the behavior of the software, it is found that it even if a very small software, it produces a huge amount of trace. It can be difficult to describe and analyze the problem with a large number of the trace. And trace description data will take up a lot of storage resources and be not conducive to long-term storage. In order to solve the above problems, the GK-tail algorithm uses the Extended Finite State Model (EFSM) to represent the software behavior model with data constraints.

3.1. Description of GK-Tail Algorithm

GK-tail is a dynamic model of software behavior based on extended finite state automata model (EFSM). It has four main steps:

1) Merge the software behavior trace which input equivalence;
2) Generate predicates;
3) Establish an initial extended finite state automaton;
4) Consolidate the equivalent state to obtain the final extended finite state automaton [19-20].

The paper calls the track to describe the sequence of the method call, the value of the label and the variable. The data is about the parameters, variables, or other information that invokes the method. They have their own values. Additional information is automatically added via the monitoring platform or manually by the tester. Figure 2 is an example of a call sequence. And Figure 3 is a corresponding set of interaction traces, as shown in the following figure.

In figure 2, the label under the label indicates the parameter values which online and the context variable which below the line. When a context variable exists, it is marked.

The GK-tail handles the interaction traces set, producing an EFA that handles all trajectories. The interaction traces represent the set of executions. Several interaction traces often trigger an identical behavior. This means that the different values of the parameters call the same method sequence. GK-tail uses this feature to simplify the input of the equivalent trace to the data set. Figure 4 shows the data set obtained from the interaction traces of Figure 2.

\[
\begin{align*}
\text{it1} &= (m1, 0, 1) (m1, 1, 2) (m2, (0, 0), -) (m3, \text{IT'}, -) (m1, 0, 3) (m2, (0, 0), -) \\
\text{it2} &= (m3, \text{UK'}, -) (m3, \text{UK'}, -) (m2, (0, 3), -) (m3, \text{UK'}, -) (m1, 0, 3) (m2, (0, 15), -) \\
\text{it3} &= (m1, 1, 5) (m1, 1, 2) (m2, (0, 0), -) (m3, \text{UK'}, -) (m3, \text{UK'}, -) (m2, (0, 15), -) \\
\text{it4} &= (m1, 0, 3) (m1, 1, 2) (m2, (0, 0), -) (m3, \text{IT'}, -) (m3, \text{IT'}, -) (m2, (0, 30), -)
\end{align*}
\]

**Figure 2.** Examples of four call sequences

**Figure 3.** Interaction traces

In the second step, GK-tail generates predicates from the data set. Figure 5 shows an example of using Daikon to automatically generate predicates.
Figure 5. Automatically generates predicates. A predicate summarizes the conditions from the accepted corresponding method calls. GK-tail uses Daikon to generate predicates.

In the third step, GK-tail generates an initial EFA by adding a common initial state from the trace of the tagged predicate. Figure 6 shows an example of an initialized EFA.

In the fourth step, GK-tail simplifies the initial EFA by combining the equivalent state to obtain a compact model of the observed behavior.

Figure 6. The initial EFA is obtained from the data set

Because the second step of the GK-tail algorithm uses the dynamic analysis tool Daikon, when using Daikon to obtain the predicate of the software behavior trace, the transition condition of the EFA state is obtained. This paper defines the improvement of the GK-tail algorithm by understanding the static and dynamic analysis tools. That means that the improvement of the analysis tools.

3.2. The Combination of ESC/Java and Daikon
In the second step of GK-tail algorithm, the resulting information is improved with higher accuracy because of the combination of the ESC/Java and Daikon tools.

The invariants of the program found by dynamic techniques are not necessarily correct, because of the limitation of times of running the tools. Static technology can verify that certain properties are always correct. But it is very difficult to choose a validation target for a static validator and add comments.

Integration of these two technologies, we can take their own excellent to make up for each other's shortcomings. Dynamic techniques can find invariants of programs which can be treated as annotations of programs or be used as static validation purposes. Static verifiers can verify invariants found by dynamic techniques [21].
Figure 7: Daikon and ESC/Java tools combined workflow. First, run Daikon to get the program invariants. The program invariants can be inserted into the JML language; then the JML annotation with the Java code input to the ESC/Java. Finally, verify the invariants generated by Daikon.

The collection dynamic invariant discovery tool Daikon and static validator ESC/Java can filter the false invariants generated by Daikon.

This part describes two analysis tools, static analysis tools ESC/Java and dynamic analysis tools Daikon. Based on the analysis of GK-tail algorithm and tools, it is determined how to improve the GK-tail and design and implement the software behavior modeling system to verify the feasibility of the algorithm.

3.3. Improved GK-Tail Algorithm

Now we define the interaction traces. Given a finite set of a method X, a set of possible input parameters R and the corresponding domain Dx, a possibly empty set of variables V and the corresponding domain DV, a parameter trajectory is a triple \((x, p, V)\), where \(x \in X\), \(p \in D_x\), \(V \in DV\).

\[
\tau = (x_1, p_{s1}, V_1) \ldots (x_n, p_{sn}, V_n),
\]

where \(x_i \in X\), \(p_i \in D_{x_i}\), \(V_i \in DV_i\).

Given a state of EFA and a parameter trajectory \(p_i = (x, p, V)\), \(p_i\) is able to complete a transition \(t = (s, x, P, s')\) when \(P(p, v) = True\). There is a transition sequence \(it = (x_1, p_{s1}, v_1) \ldots (x_n, p_{sn}, v_n)\), when an EFSM has the initial state \(s_0\), the final state \(s_f\) accepts an interaction trace, \((x_i, x, P_i, s_i)\) \((s_0, x, P_0, s_0)\) \((s_1, x, P_1, s_1)\) \((s, x, P, s_f)\) ... where \(P(p, v) = True\), \(\forall i = 1.. N\), \(s_i \in s_f\).

The improved algorithm is divided into four steps: (1) monitor software to obtain the monitoring trace log, merge the input-equivalent trace, and record it; (2) use Daikon and ESC/Java to generate the predicate of the relevant trace (3) produce an initial EFA; (4) merge the equivalent state, optimize the initial EFA, get the final EFA. These four steps will be described in below.

A data set is defined as a sequence \((x_i, dp_{x_i}, dv_{x_i})\) \((x_i, dp_{x_i}, dv_{x_i})\), where \(x_i \in X\) is a method, \(dp_{x_i} \in \mathcal{D}(D_{x_i})\) is an input parameter evaluation set, and \(dv_{x_i} \in \mathcal{D}(DV)\) is a variable evaluation set. Given an input equivalent trajectory set \(it_i = (x_i, p_{s1}^{i_1}, v_{s1}^{i_1}) \ldots (x_n^{i_n}, p_{s2}^{i_n}, v_{s2}^{i_n})\), the GKT-tail is merged into the data set \(d = (x, p_{s1}^{i_1} \cup \ldots \cup p_{s2}^{i_n}, v_{s1}^{i_1} \cup \ldots \cup v_{s2}^{i_n})\).

In the second step, after the merging of the interactive trajectory, the paper uses Daikon and ESC/Java to generate the relevant trajectory predicate. Formal definition of Daikon and ESC/Java as a function daikonEsc: \(\mathcal{D}(D_{x_i}) \cup \mathcal{D}(DV) \rightarrow P\), where \(D_{x_i}\) is an input parameter evaluation set for any \(x_i \in X\), \(D_{x_i}\) is an evaluation set of context variables, \(P: D_{x_i} \times D_{x_i} \rightarrow \{True, False\}\) is a predicate based on parameters and context variables. Daikon and ESC/Java guarantee that when daikonEsc \((Data, V)\) = \(P\), then \(\forall p_i \in Data, d_i \in V, P(p_i, d_i) = True\). This step is the place where the GK-tail algorithm is improved.

The GK-tail algorithm uses the Daikon tool to get the predicate of the software behavior trace in this step. Before the merge process, the set of software behavior trajectories may contain incorrect data, and
Daikon will not check again. The improved GK-tail algorithm uses the combination of ESC/Java and Daikon to acquire the software. The improved GK-tail algorithm uses the combination of ESC/Java and Daikon to acquire the software the behavioral trace predicate will filter out the erroneous data information contained therein, thus ensuring that the resulting extended finite automaton's state and transition condition information are more correct and improved in accuracy.

The combination of Daikon and ESC/Java aims to avoid incorrect data in the input parameter set R and the corresponding domain $D_{\xi}$. It leads to bring the wrong results in the follow work. It is also the innovation of this paper. Through the previous section of the two analysis tools of the theoretical description, the algorithm GK-tail is proved that it is feasibility theoretically.

In the third step, an initial EFA is generated by adding a common initial state from the trajectory of the tagged predicate. This step corresponds to the system data analysis module in the system design. for $\forall i = 1, ..., m$, the initial EFSM (S, T, $s_0$, $s_f$) is the set of trajectories for a given predicate $\{seq_1, ... seq_m\}$, where

$\text{seq}_i = (x'_1, P'_1) ... (x'_{n_i}, P'_{n_i})$, are defined as follows:

$S = \{s_0, s_1, ..., s_{n_1}, s_1, ..., s_{n_2}, ..., s_{m_1}, ..., s_{n_m}\}; s_{0} = \{s'_{i}\mid i = 1, ..., m\}; s_{0}$ is the initial state.

For each element in the input sequence set $(x'_1, P'_1), j > 1$, there is a transfer $t = (s'_{j-1}, x'_j, P'_j, s'_{j}) \in T$.

For each element in the input sequence collection $(x'_1, P'_1)$, there is a transfer $t = (s_0, x'_1, P'_1, s'_1) \in T$.

In the fourth step, the compact EFA is reduced by combining the equivalent state. It obtains a compact model of the observed behavior. The merging of EFA is an extension of the main rules of the kTail algorithm, which proposed by Bierman and Feldmann. They thought that these states are equivalent when the states pass through the same path. It is a unique logical state of multiple expression and can be safely merged when different EFSM states generate an equivalent set of behaviors.

In general, the state of the path set may be infinite, so the contrast state may be very expensive. We compare the length of the sequence to k, similar to that proposed by Bierman, Feldmann. Given a state $s \in S$, define the k-future (s) of the sequence set $\{seq_1, ... seq_m\}$, where

$\text{seq}_i = \{(x_1, P_1) ... (x_i, s_i)\}$, there exists a transfer sequence (S, $x_1$, $P_1$, $s_1$) ($x_1$, $x_2$, $P_2$, $s_2$) ... ($x_{i-1}$, $x_i$, $P_i$, $s_i$).

The above is an improvement to the GK-tail algorithm because Daikon's dynamically generated data information may be in error. ESC/Java verifies the data after using the Daikon to determine which data is incorrect, and then filter it out. The combination of Daikon and ESC/Java can the wrong information. It is shown that algorithm can improve the accuracy.

4. Implementation of Software State Monitoring System

Based on the above conclusions, the paper determines the Daikon as a monitoring program to run the tool. First, predicate is generated by Daikon from the data set, which is generated by the improved GK-tail algorithm. Finally, we can get the program running state diagram. This section will analyze the Daikon monitoring log with the related techniques such as program running trace, invariant detection method and program invariant extraction. In the GK-tail algorithm, the program invariant is found using dynamic detection technology, which is Daikon reasoning engine for invariant detection.

![Figure 8. Notepad program flow chart](image-url)
Daikon compiles, links to C language source, generates executable programs
Daikon makes the report log after running the program
Executable programs are generated under Daikon's monitoring

Figure 9. Flowchart of the Daikon monitor running

4.1. Daikon on Notepad Monitoring
Notepad is a simple program with create, loaded, and delete text functions. The Notepad can reload, modify, and delete the text file. The function flow chart shown in Figure 8.

Daikon monitors and generates a report file on the program. On the basis of the C source, Daikon integrates the compilation linker, which links the C code directly into an intermediate process file and generates an executable exe program. Daikon will execute this executable file with Purify's target code plug-in function and generate a log. The process shown in Figure 9.

Figure 10. The figure 10 shows the predicate generation module implementation flow chart. First, we use the Daikon reasoning engine to invariably analysis the input .dtrace file to get the JML description form of the likelihood invariant. And then it is entered into ESC/Java that the JML description of the likelihood variable and the system source. The ESC/Java detection will discard conflicting program invariants and output more accurate program invariants, the predicates needed in the second step of GK-tail.

4.2. Design of State Chart Generator
The paper introduced how to monitor the operation of the program by the Daikon tool and how to generate predicates of the process previously by GK-tail algorithm. At the next step, it needs to generate software running state diagram from these results.

This procedure is running at java environment. The program draws graphics by using jGraph plug-in. The input text file is from the report log which logged by Daikon when the program is running. The
output result is the program running state diagram corresponding to the log report. This procedure mainly draws the state graph by analysis the generated log report. First, the program read the target log file. And then the program records the function, call time and other information. Finally, the final state diagram is drawn. The process is shown in Figure 11.

![Diagram](image)

**Figure 11.** Generates the program flow diagram

5. System Running Instance
The program finally achieved the above design. The paper completed the expected function. The paper will show it how to generate a log text file from Daikon monitor notepad program and how to draw the state diagram by jGraph.

First, the Notepad program is executed. Daikon generates the report as shown in Figure 12.

At this time, we run the drawing state diagram program. The program running state diagram is drawn as shown in Figure 13.

When the program is running, the operating status shown in Table 1.
Table 1. Shows the running status of the program consistent with the order of the actual operation of the user. Through the operation of this example, it is proved that state diagram which drawn by Daikon and the jGraph is achieve the desired demand and design.

| Status unit contents | time      | description           |
|----------------------|-----------|-----------------------|
| START                | start     | new_file()            |
|                      | 19:46:38  | Create new note       |
| edit_file()          | 19:46:40  | Edit note             |
| load_file()          | 19:46:43  | Load note             |
| open_file()          | 19:46:45  | Open note             |
| del_file()           | 19:46:48  | Delete note           |
| END                  | 19:46:52  | end                   |

It will change the process of the program by the injection tool to attack the program. It will get a different state diagram. It is only an examples which loaded without creating note of the state diagram, as shown in Figure 14.

Figure 13. Normal calls to the software's state diagram

Figure 14. The software state diagram after being injected into the tool

6. Conclusion
Through the experiment, we get the following conclusions. First, the paper analyzes software behavior. The paper improves the software behavior dynamic modeling GK-tail algorithm based on extended finite state automata model. Second, the paper combines the GK-tail algorithm with Daikon, which is the invariant analysis software, to generate the invariant change. By summarizing their behavior, this paper gives a formal specification of the program. Third, on the basis of this result, this paper uses jGraph components to make software running state diagram. Through the analysis of behavioral feature, this paper obtains the semantic description of behavior. By comparing its normal running state, the software running behavior state is analyzed. Because the intrusion will change the normal
implementation of the software or call the sequence, it can be found opportu-
nely that the problem in runtime in the software. The GK-tail algorithm can judge whether software trusted behaviour is trusted.  
7. Acknowledgment  
This work was supported the National Key R&D Program of China under Grant No. 2016YFB0800700

8. References

[1] Davide Lorenzoli, Leonardo Mariani, Mauro Pezzé*.
Automatic Generation of Software Behavior Model [J]. ICSE’08, May 10–18, 2008.

[2] Trusted Computing Group. TCG Specification Architecture Overview: Revision 1.4 [EB/OL].

[3] Zhang Huanguo, Luo Jie, Jin Gang, et al. Development of Trusted Computing Research [J]. Journal of Wu2 han University (Natural Science Edition), 2006, 52(5):513-518.

[4] Li Xiaoyong, Gui Xiaolin, Mao Qian, et al.Adaptive dynamic trust measurement and prediction model based on behaviour monitoring [J]. Chinese Journal of Computers, 2009, 32(4):664-674.

[5] Yang Xiao-hui. Research on Dynamic Trusted Theory Model of Software Behavior [D].Heifei: University of Science and Technology of China, 2010.

[6] Qu Yan-Wen. Software Bahavior [M], Beiijng:Publishing House of Electronics Industry, 2004.

[7] YANG Xiao-hui, ZHOU Xue-hai, Tian Jun-feng, LI Zhen. Novel Dynamic Trusted Evaluation Model of Software Behavior [J]. Mini-Micro Systems, 2010, 31(11):2113-2120.

[8] Rajeev Gopalakrishna,Eugene H Spafford,Jan Vitk. Efficient intrusion detection using automata inlining[C]. Proceedings of the 2005 IEEE Symposium on Security and Privacy. Oakland, California, U SA, 2005, 18-31.

[9] Henry H Feng, Oleg M Kolesnikov, Prahlad Fogla, et al. Anomaly detection using call stack
information[C]. Proceedings of the 2003 IEEE Symposium on Security and Privacy, Oakland, California, USA, 2003, 62-75.

[10] Zhen Liu, Susan M Bridges, Rayford B Vaughan. Combining static analysis and dynamic learning to build accurate intrusion detection models[C]. Proceedings of the 3rd IEEE International Workshop on Information Assurance College Park, Maryland, USA, 2005, 164-177.

[11] LI Wen, DAI Ying-Xia, LIAN Yi-Feng, FENG Ping-Hui. Context Sensitive Host-Based IDS Using Hybrid Automaton [J]. Journal of Software, 2009, 20(1):138-151.

[12] Alessandro Frossi, Federico Maggi, Gian Luigi Rizzo, et al. Selecting and improving system call models for anomaly detection[C]. Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Como, Italy, 2009, 206-223.

[13] Jonathon T Gifhn, David Dagon, Somesh Jha, et al. Environment sensitive intrusion detection [C]. Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, Seattle, WA, ashington, USA, 2005, 185-206.

[14] Spivey JM. Fast accurate call graph profiling [J]. Software Practice & Experience, 2004, 34(3):249-264.

[15] Michael D Bond, Kathryn S Mckinley. Probabilistic call context[C]. Proceedings of the 22nd Annual ACM SIG PLAN Conference on Object-oriented Programming Systems and Applications, Montreal, Quebec, Canada, 2007, 97-112.

[16] Akahashi T, Hayahi N, Hayashi S. Structure and Properties of Shape-Memory Polyurethane Block Copolymers [J]. Appl Polym Sci, 1996, 60:1061-1069.

[17] YAO Li-hong, ZI Xiao-choa, HUANG Hao, MAO Bing, XIE Li. Research of System Call Based Intrusion Detection [ J]. Acta Electronica Sinica, 2003, 31(B):1134-1137, 2003.

[18] TAN Xiao-Bin, WANG Wei-Ping, XI Hong-Sheng, YIN Bao-Qun. A Hidden Markov Model Used in Intrusion Detection [J]. Journal of Computer Research and Development, 2003, 40(2):245-250, 2003.

[19] LI Xin-Ming, LI Yi, XU Xiao-Mei. Software Vulnerability Analysis [J]. Computer Science, 30(8):163-165, 2003.

[20] Sekar R, Bendre M, Bollineni P, et al. A fast Automata-Based method for detecting anomalous program behaviors[C]/IEEE Symp.on Security and Privacy. Oakland: IEEE Press, 2001:144-155

[21] Jeremy W. Nimmer, Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java [J]. First Workshop on Runtime Verification, 2001.