http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 10 (12): 2292-2299, 2011
ISSN 1812-5638 / DOI: 10.3923/itj.2011.2292.2299
© 2011 Asian Network for Scientific Information

Simplification of Software Behavior Trace and its Application in Trustiness Evaluation

Tian Junfeng and Han Jine
College of Mathematics and Computer Sciences, Hebei University, China

Abstract: Software behavior, as the basis for evaluating software's dynamic trustiness, has become a hot issue in worldwide information security area. To describe the software behavior, a model named Software Behavior Trace (SBT) is proposed which regards not only the software execution process but also the execution background. We also give series of rules to simplify the SBT and explore the usage of SBT in software's trustiness evaluation. SBT consists of operation trace and function trace and it characterizes the execution process and background, respectively and it is able to describer software behavior completely. After simplified, the time and space overheads of trustiness evaluation reduce a lot. In order to improve the accuracy of trustiness evaluation, we use evaluating strategy of determining multiple attributes' weights by information entropy. Simulation experiments demonstrate that trustiness evaluation model based on simplified software behavior trace has improved evaluation effectively without performance degradation.

Key words: Software behavior, software behavior trace, simplify, trustiness evaluation

INTRODUCTION

With convenience, economy, high scalability and other advantages, cloud computing has attracted more attention of enterprises (Feng *et al*., 2011). But in current climate, many key issues obstruct the development of cloud computing and the first to be affected is the problem of information security. As a novel technology of information system security, trusted computing has become a new upsurge in international information security field (Changxiang *et al*., 2010) and provides a security guarantee for applications of cloud computing. Though technical route of TCG is able to insure the static trustiness of computer during the initialization phase, it may cause security risks without considering the dynamic trustiness problem in the course that software runs. To ensure the dynamic trustiness is to keep that the software always behaves in the expected way and realizes the expected purpose. Therefore, it is meaningful to research trustiness evaluation based on software behavior (Trusted Computing Group, 2007). It is a good idea to research credibility evaluation based on software behavior.

In the 1990s, the computer science theoretical subject-software behavior (Yanwen, 2004) was proposed by Professor Qu Yanwen and the subject describes the theories, models and methods of software behavior. Present study focuses on concept of software behavior trace and its usage in software trustiness evaluation. Obviously, the more concise expression of behavioral trajectory is, the more efficient its storage, access and other operations are. In order to improve the efficiency of trustiness evaluation, we present reduction rules of software behavior trace and give the trustiness evaluation framework based on simplified software behavior trace.

RELATED WORK

Currently, large quantities of research achievements have mounted in the field of the software behavior model.

Hofmeyr *et al*. (1998) proposed the first program behavior model based on system calls, called N-gram model. Wepsi *et al*. (2000) proposed variable-length sequence model, named Var-gram model and it is more suitable for the program's practical case. The model has higher detection capability and lower time and space overheads while it can not capture the relationships of the long span system call sequences. Sekar *et al*. (2001) gave a new behavior model, namely Finite State Automata (FSA), yet it has the impossible path problem. Wagner and Dean (2001) proposed a complex push-down automaton model, called abstract stack model. The model is able to consider not only the program counter but also state of the call stack. Feng *et al*. (2003) extended the work of Sekar *et al*. (2001) and generated abstract execution path through dynamic training method. Their Vt-path model avoids the impossible path problem and requires low system overhead. Wen *et al*. (2009) generated HFA model by static analysis and dynamic binding. Giffin *et al*. (2005) proposed context-based model Dyck which is generated by static analysis and dynamic analysis. This model analyses the data flow,

Corresponding Author: Tian Junfeng, College of Mathematics and Computer Science, Hebei University, China

there by increasing the ability of resisting impersonation attack and non-control flow attack.

The models based on system call sequences and the models based on context-free automata can't detect complex attacks, such as mimicry attack. The context-sensitive models which include parameters and environment variables organize the information not reasonability and it can't show the software behavior vividly. Most of the models don't consider branch sentences and are not simplified, so the efficiency is unsatisfactory.With taking full account of the operational process and surrounding, we propose Software Behavior Trace (SBT) to describe software behavior. We present reduction rules of SBT which improve the efficiency of trustiness evaluation. A trustiness evaluation framework based on simplification software behavior trace is proposed.

SBT

As the description of software behavior, SBT should not only extract the operation sequences which constitute of operation traces but also acquire the sequences of the corresponding scenes which compose the function trace.

The system call sequences reflect the software's control flow (Fen *et al*., 2010), so we extract the SBT at system call level. We set up check points at the important system call sites to evaluate whether the software is trustiness. Due to the problem of non-determinism caused by branches, it is necessary to set up check point at branch point. The branch point caused by system call is called system call branch point and the branch caused by branch sentence, such as if else, is named sentence branch point. Organizing the check points according to the executing order, we can get the expected operation trace. In addition, exacting the scenes at check point, we can get the expected function trace.

Figure 1 shows a simple SBT: The solid line graph describes operation trace, v_i is a check point; the dotted line graph shows function trace, s_i is the scene of v_i. The points and edges of operation trace and function trace are one to one correspondence and they make up SBT.

A SBT can be denoted by a directed acyclic credible view composed by a number of check point sequences and it is described by a 5-tuple SBT (V, S, E, vs_0, VS_q). The notions are described as follows:

V is a finite non-empty set of nodes, v_i0V records the identification of check point. The identifications of non-branch check points and system call branch points are system call numbers and others are -1.

S is a set of scenes. s_i0S is the scene gained at v_i. s_i is expressed as s_i = (Identification, Type, Scene). Identification is a numeric identification of the check point. Type describes the kind of check point: 0 denotes that the check point is the initial point; 1, final check

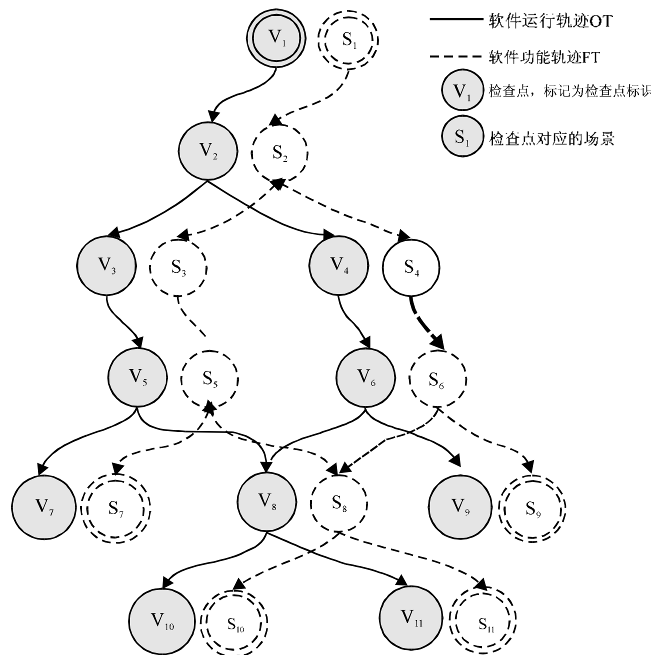

Fig. 1: A simple software behavior trace (SBT)

point; 2, non-branch check point; 3, system call branch point; 4, sentence branch point. Scene records the software surrounding collected at check point. Scene of non-branch check point contains the information as follows, function, parameters, result, CPU load, memory allocation and time offset, it is expressed as (Function, Param, Result, CPU, Memory, Time Offset). And the scene of branch point includes function, CPU load, memory allocation, time offset, branch condition, information of branch.

E is a set of edges connecting nodes associated with transitions. Its element e_i is a directed edge and is described as $e_i = \langle v_i, v_j \rangle$. As connections between scenes correspond to the connections between checkpoints, $e_i = \langle v_i, v_j \rangle$0E_$e_i = \langle s_i, s_j \rangle$0E:

- vs_0 is the initial node.
- Vs_q is the set of the final nodes

Operation Trace is described as OT (V, E, vs_0, VS_q); Function Trace is expressed as FT (S, E, vs_0, VS_q).

SIMPLIFICATION OF SBT

Because of the complexity of software structure and functions, SBT is quite large and contains large amounts of redundant information. So simplification of SBT is needed. The simplification methods of behavior models are mainly states merging methods based on finite state automaton. They are mainly RPNI method proposed by Oncina and Garci (1992), Blue-Fringe method given by Lang *et al*. (1998) and k-tail method proposed by Biermann and Feldman (1972). Dupont and his colleagues (Dupont *et al*., 2007) proposed the QSM method by extending the RPNI method and Blue-Fringe algorithms and Walkinshaw *et al*. (2008) described an extension of QSM with accounting for temporal logic formulae. Lorenzoli *et al*. (2008) gave the Gk-tail method based on k-tail and the method takes account of the relationship between data values and invocation sequences. These methods seldom take account of the surrounding information of the running software. Above methods don't take into account the emergences of new branches after simplified.

We give reduction rules based on branch point for SBT. In the simplification process, whether the nodes are equivalent depends on not only system calls but also other scene information and it makes the nodes merging much more precise. Whether merging two nodes or not doesn't rely on the length of their equivalent suffixes but on not creating new branch after merging.

To make the description of rules much easier, we give following definitions.

Definition 1: Pioneer set: For any $v \in V$, the pioneer set of v is:

$$\{u \mid u \in V \wedge \langle u, v \rangle \in E \wedge u \neq v\}$$

noted as $\Gamma^-(v)$. For any $s \in S$, the pioneer set of *s* is noted as $\Gamma^-(s)$.

Definition 2: Follow-up set: for any $v \in V$, the follow-up set of v is:

$$\{u \mid u \in V \wedge \langle v, u \rangle \in E \wedge u \neq v\}$$

noted as $\Gamma^+(v)$. The follow-up set of s is noted as $\Gamma^+(s)$.

Definition 3: Out-degree: for any $v \in V$, its out-degree is number of edges whose initial node is v, noted as d_v^+. For any $s \in S$, its out-degree is noted as d_s^+.

Definition 4: Trace segment is node sequence with v_i as its initial node and v_j as its finial node, noted as T_{ij}. It is denoted by (V_{ij}, S_{ij}, E_{ij}, v_i, VS_{qj}), where, $V_{ij} \in V^*$, $S_{ij} \in S^*$, $E_{ij} \in E^*$, $v_i \in V_{ij}$, $VS_{qj} \in V_{ij}^*$. L (T_{ij}) represents its length.

Definition 5: Delete operation is the operation of deleting node. Deleting v is expressed as del (v); del (T_{ij}) reflects the operation of deleting the trace segment T_{ij}. When deleting node, we delete the related edges.

For storing and handling conveniently, reduction rules must satisfy the following simplified condition: $d_v^+ \leq 2$ and $d_s^+ \leq 2$ holds for any $v \in V$ and $s \in S$ after simplification. The detailed reduction rules are introduced as follows.

Rule 1 (simplify traces with little probability): Neglect the scene of Trace segments handled exception because the opportunities of these traces run or be attacked are little. This rule doesn't affect the nodes' out-degree, so it satisfies the simplified condition.

Rule 2 (delete null call branches):

$$\begin{aligned} & e_{i1}, e_{i2} \in E \text{ and } e_{i1}, e_{i2} = \langle v_i, v_j \rangle \\ & \Rightarrow \Gamma^-(v_j) = \Gamma^-(v_j) \cup \Gamma^-(v_i) - v_i \text{ , del}(v_i), \Gamma^-(s_j) = \Gamma^-(s_j) \cup \Gamma^-(s_i) - s_i \text{ , del}(s_i) \end{aligned}$$

If the branches corresponding to the same branch point contain no check points, then delete the branch point and its two edges and the original edges pointing to the branch point should now point to the next node of the branch point. Figure 2 is an example.

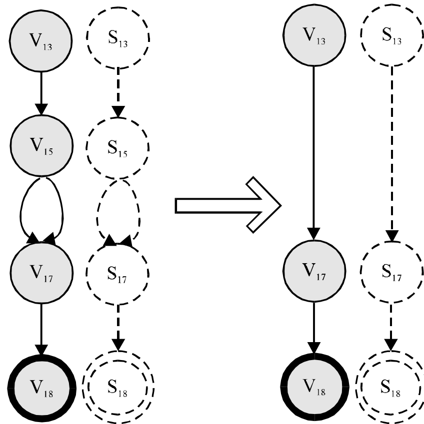

Fig. 2: Delete null call branches

Rule 3 (merge equivalent branches):

$$
\begin{aligned}
& SBT(V,S,E,vs_0,VS_q), T_{m_1n_1}(V_{m_1n_1},S_{m_1n_1},E_{m_1n_1},v_{m_1},VS_{qn_1}), T_{m_2n_2}(V_{m_2n_2},S_{m_2n_2},E_{m_2n_2},v_{m_2},VS_{qn_2}); \\
& v_{m_1},v_{m_2} \in \Gamma^+(v_i), \exists v_o \in VS_{qn_1}, \exists v_p \in VS_{qn_2} \text{ and } v_o,v_p \in \Gamma^-(v_1); \\
& \forall g = m_1 \ldots n_1, \exists k \in m_2 \ldots n_2, s_g(\text{Scene}) = s_k(\text{Scene}) \\
\Rightarrow\ & \Gamma^+(\Gamma^-(v_i)) = \Gamma^+(\Gamma^-(v_i)) - v_i + v_{m_1}, \Gamma^-(v_1) = \Gamma^-(v_1) - v_p, \\
& \Gamma^+(\Gamma^-(s_i)) = \Gamma^+(\Gamma^-(s_i)) - s_i + s_{m_1}, \Gamma^-(s_1) = \Gamma^-(s_1) - s_p, del(T_{m_2n_2}), del(s_i) \text{ and } del(v_i)
\end{aligned}
$$

If the two branches of branch point have the same information and can't be distinguished, then it is insignificance to inspect the branch point and we should delete the branch point and one of the branches. In Fig. 3, suppose that two branches of v_2 have equivalent FT, then delete v_2, s_2 and the branch with smaller identification. Figure 4 is a special case. v_2 is branch point, s_3(Scene) ≠ s_4(Scene), s_5(Scene) = s_6(Scene), then merge the

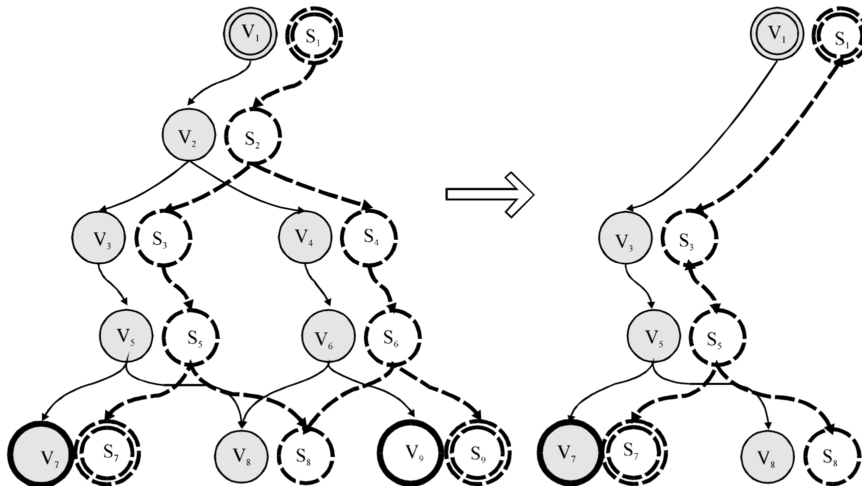

Fig. 3: Merge equivalent branches–whole branch

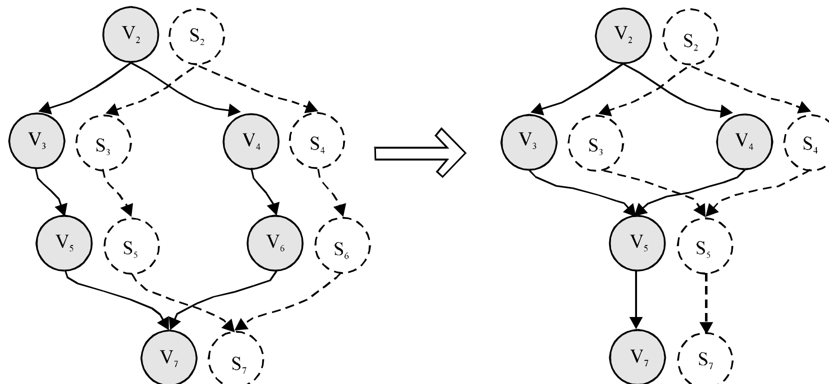

Fig. 4: Merge equivalent branches – branch suffix

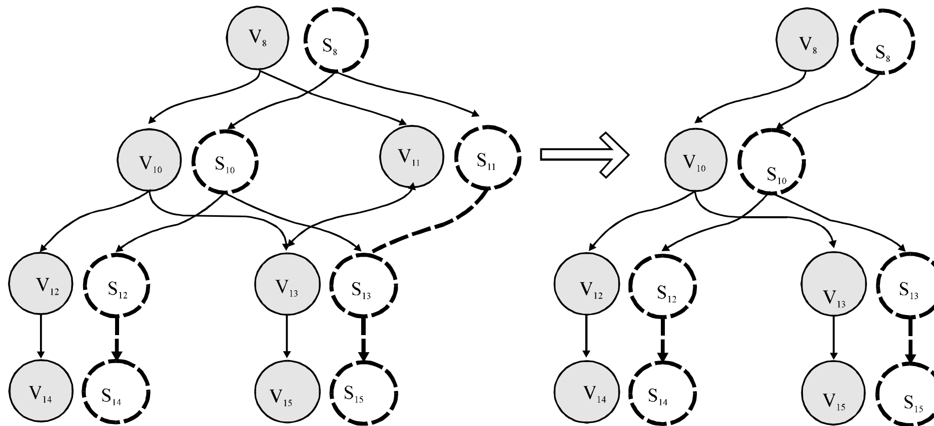

Fig. 5: Dissatisfy rule 3

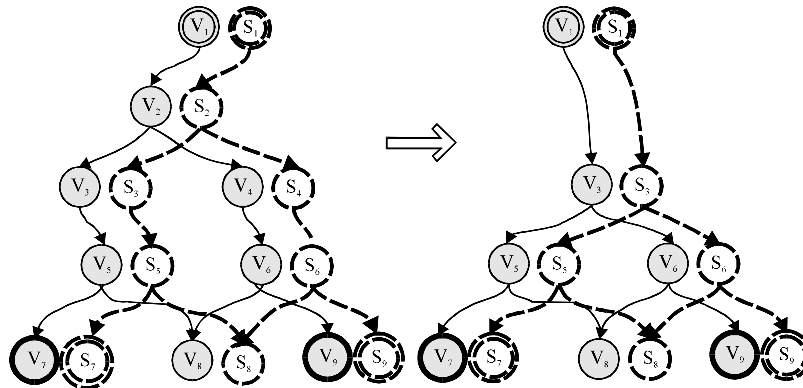

Fig. 6: Dissatisfy rule 3-merge branch infix

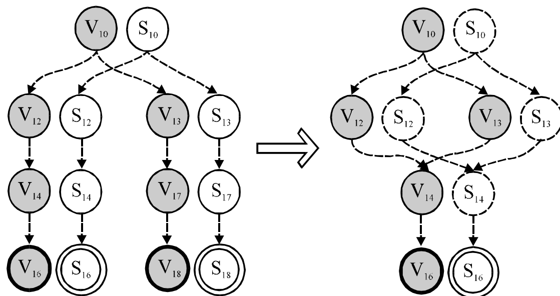

Fig. 7: Merge equivalent suffix

equivalent trace segments v_5, v_6 and their scenes. For Fig. 5, v_{10} is a branch point and v_{11} is a non-branch check point, so the merging in the Figure is against the rule. In Fig. 6, new branches is produced at v_3, the merging is also against the rule.

Rule 4 (merge equivalent suffix):

$$SBT(V,S,E,vs_0,VS_q), T_{m_1n_1}(V_{m_1n_1},S_{m_1n_1},E_{m_1n_1},v_{m_1},VS_{qn_1}), T_{m_2n_2}(V_{m_2n_2},S_{m_2n_2},E_{m_2n_2},v_{m_2},VS_{qn_2});$$
$$VS_{qn_1},VS_{qn_2} \subset VS_q; \forall g = m_1 \ldots n_1, \exists k \in m_2 \ldots n_2, s_g(Scene) = s_k(Scene)$$
$$\Rightarrow \Gamma^+\left(\Gamma^-\left(v_{m_2}\right)\right)=\Gamma^+\left(\Gamma^-\left(v_{m_2}\right)\right) - v_{m_2} + v_{m_1}, \Gamma^+\left(\Gamma^-\left(s_{m_2}\right)\right)=\Gamma^+\left(\Gamma^-\left(s_{m_2}\right)\right) - s_{m_2} + s_{m_1}, del\left(T_{m_2n_2}\right)$$

If two trace segments are equivalent and their final nodes belong to VS_q, then merge the two trace segments. In Fig. 7: Suppose s_{14}(Scene) = s_{17}(Scene) and s_{16}(Scene) = s_{18}(Scene), then the segments are equivalent and the segment whose identifications are smaller should be deleted.

THE USAGE OF SBT IN TRUSTINESS EVALUATION

Based on the research on SBT, we proposed a trustiness evaluation framework, as showed in Fig. 8.

Evaluation of OT mainly checks whether the practical identification is matched with the excepted one, If it is matched, the practical OT is credible, else not credible.

Evaluation of FT needs to consider multiple attributes of the scene and we adopt a strategy of determining multiple attributes' weights by information entropy for check point (Junfeng and Ye, 2011) to evaluate the FT. Weight of a attribute depends on dispersion degree of its values. In order to get weights of attributes, suppose that there is a set of n-

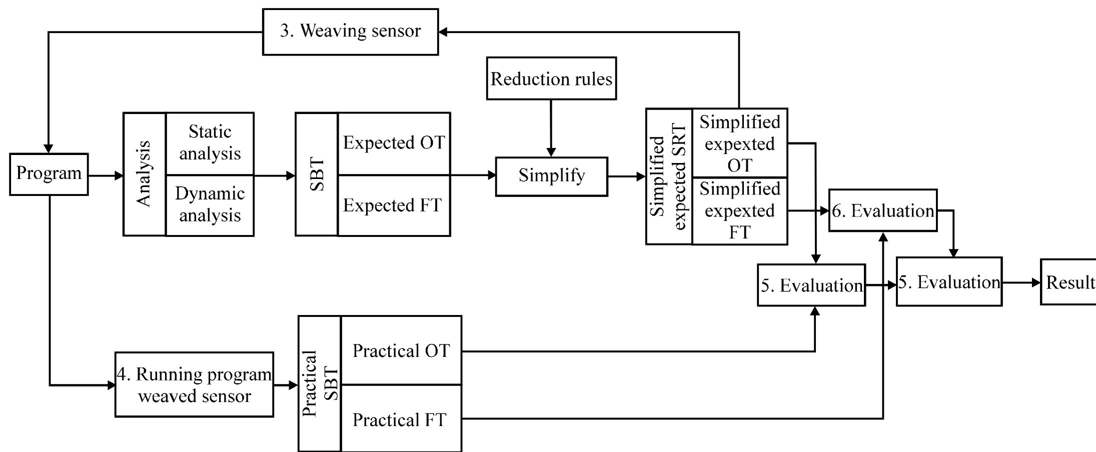

Fig. 8: Trustiness evaluation framework based on simplified SBT

samples,denoted by E = { $e_1, e_2, \ldots, e_n$} which is collected at certain ordinary check points of software execution.

Each sample e_j is described by the vector of attributes and it can be noted as e_j = <Function, Param, Result, CPU, Memory, Timeoffset>. So, the matrix E = {e (I, j)}, 1 = I = n, 1 = j = 6 is the source of determining multiple attributes' weights by information entropy. According to reference Junfeng and Ye (2011), we get the weight of j:

$$\omega_j = \frac{1 + k\sum_{i=1}^{n} p_{ij} \ln p_{ij}}{6 + \sum_{j=1}^{6} k \sum_{i=1}^{n} p_{ij} \ln p_{ij}} \quad k = \begin{cases} 1 & n_a = 1 \\ \frac{1}{\ln n_a} & n_a \geq 2 \end{cases} \quad , j = 1, 2, \cdots, 6$$

where,

$$p_{ij} = \frac{|e(i, j)|}{\sum_{i=1}^{n} |e(i, j)|} \quad i = 1, 2, \cdots, n$$

is the probability when the value of attribute j is e(i ,j); n_a is the count of different values of attribute j, ln n_a is the max value of information entropy, so $0 \leq e_j \leq 1$.

After get the weights of all attributes, we sum up all sample pattern with different weighting factor to evaluate the FT. The values of every attribute are described by abstract value range (Xiao *et al*., 2010). If the value of j belongs to the expected range, its evaluated value c_j = 1, else c_j = 0. The evaluated value C of check point is:

$$C = \sum_{j=1}^{6} \omega_j c_j$$

We define a credible threshold T. If C<T, it indicates the software has been attacked, else the software is trustiness. The value of T depends on special condition.

EXPERIMENT AND RESULTS

Environment and method: The simulation experiment is conducted in host with Intel Pentium 4 CPU3.06 GHz, the memory 2.49 GB, Linux kernel version 2.4.20. We use AspectJ as a development language, Eclipse and AJDT plug-in as development tools.

We use gzip, eject, ps as the test programs and we compare the unsimplified trustiness evaluation model based on Software Behavior Trace (SBTTM), simplified trustiness evaluation model based on software behavior trace (Simplified-SBTTM) against the Dyck model and GK-Tail model in three aspects.

Analysis of results

Precision: Precision is the matching degree between software model and practical software (Oncina and Garci, 1992). We adopt average branching factor metric which developed by (Giffin *et al*., 2005), to measure model's precision. Average branching factor is a dynamic measure of an attacker's chance to insert dangerous system calls into a running program's call flow. Figure 9 shows the precision of the three models on the three test programs. In Fig. 9, we can see that the average branching factors of SBTTM and Simplified-SBTTM are smaller. Dyck model just takes account of the branches relating to system calls and it ignores other branches, so the accuracy is low. Although GK-Tail has considered parameters and so on, it is uncertain and new branches may be generated during merging process. So its accuracy is lowest. SBTTM can determine which branch will be run according to the scene at branch point, so the average branching factor is low.

Table 1: Vulnerabilities of wu-ftpd-2.6.0 and evaluation results

Vulnerability	Description	Type of attack	Detect ability			
			FSA	Dyck	HFA	SBTTM
CVE-2004-0148	Restricted-gid option enabled	Data Flow	×	√	×	√
CVE 2004-0185	S/Key authentication stack overflow	Control Flow	√	√	√	√
CVE 2003-0466	Fb_realpath off-by-one bug	Control Flow, Data Flow	√	√	√	√
CVE-2001-0550	Heap corruption in file glob	Control Flow	√	√	√	√
CVE 2000-0573	SITE EXEC Buffer Overflow	Control Flow, Data Flow	√	√	√	√
CVE 2000-0574	Gain root access by a format attack	Control Flow, Data Flow	√	√	√	√
Background program	Memory footprint or destroying files	Trojan, virus	×	×	×	√

√ enable to detect attack, ×unable to detect attack

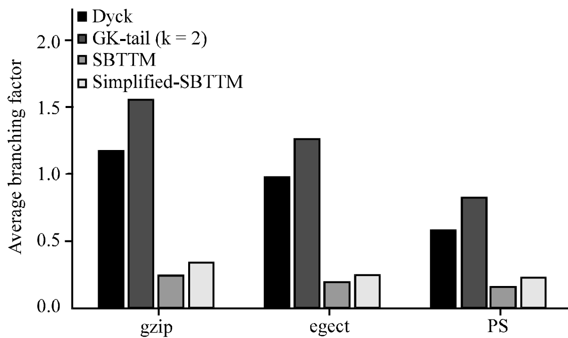

Fig. 9: Results of the average branching factor

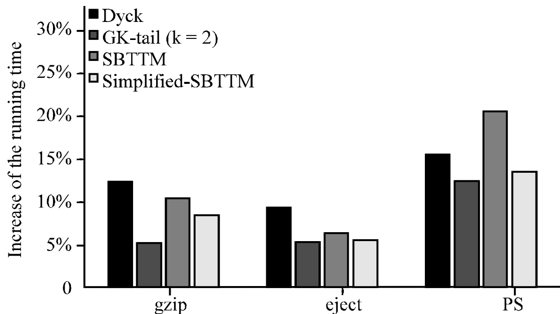

Fig. 10: Increase of running time

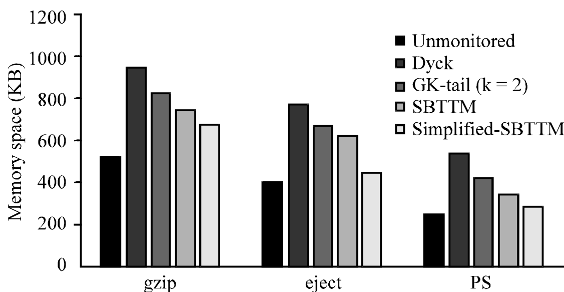

Fig. 11: Memory use (KB) of running programs

Efficiency: In order to test our model's efficiency, we compare the running time between models. Figure 10 shows the time overhead; the horizontal axis represents the increasing percentages of every monitored program's running time. Figure 11 is a comparison on space overhead where Unmonitored bar is the memory overhead without using monitor and Dyck, GK-Tail, SBTTM and Simplified-SBTTM correspond to their respective memory overheads.

GK-Tail model has higher efficiency with considering less scene information. After imperfect simplification, it also needs larger memory space. SBTTM evaluates each sense of important system call and makes judgments at each branch, so its efficiency is low. It doesn't recode the context of the program, so it has lower space overhead and higher retrieval efficiency. The program ps is an exception. Its large number of system calls makes the number of check points is very large and its operation trace takes much space. Simplified-SBTTM removes redundant information, so the efficiency is greater than other models.

Detect ability: In order to show the detect ability of model, this study selects wu-ftpd-2.6.0 as the object program which is a commonly used FTP server under Linux environment. For its typical bugs, we attack wu-ftpd-2.6.0 by attack program. By the way, we also make some tests on the monitoring capability of background program. In order to highlight the capacity of detection, three models were selected for comparative experiment, they are FSA, HFA and Dyck model.

In Table 1, the first column is Vulnerabilities and their descriptions are shown in column two, column three is the type of attack, mainly contains control flow and data flow, detect ability shows whether these models have the ability of checking out attacks.

Observe the results, FSA and HFA can only detect the attacks related to control flow. Without change control flow, CVE-2004-0148 will lead parameters exception and SBTTM can detect this abnormity during evaluating the practical FT. For other attacks, they all need to change software's control flow and the exception can be detected by the evaluation of OT. That is to say SBTTM can detect not only control flow attacks but also data flow attacks. Because of considering necessary running scene, such as CPU load and Memory allocation, SBTTM has obvious advantages in detecting background program.

CONCLUSION

In this study, we propose software behavior trace to describe software behavior and introduce some concepts, such as OT, FT, branch point, scene, etc. We simplify SBT by reduction rules under the premises of keeping the integrity of software behavior and without producing new branches and the simplification improves the efficiency of evaluation. Another contribution is that we give a trustiness evaluation framework based on simplified SBT. The simulation results indicate the precision, efficiency and detect capability of the evaluation framework.

REFERENCES

Biermann, A.W. and J.A. Feldman, 1972. On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput., C-21: 592-597.

Changxiang, S., Z. Huanguo, W. Huaimi, W. Ji and Z. Bo *et al*., 2010. Research and development of the trusted computing. Sci. China: Inform. Sci., 40: 139-166.

Dupont, P., B. Lambeau, C. Damas and A. van Lamsweerde, 2007. The QSM algorithm and its application to software behavior model induction. Applied Arti?cial Intell., 22: 77-115.

Fen, T., Y. Zhiyi and F. Jianming, 2010. Software behavior model based on system call. Comput. Sci., 37: 151-157.

Feng, H.H., O.M. Kolesnikov, P. Fogla, W. Lee and W. Gong, 2003. Anomaly detection using call stack information. Proceedings of the IEEE Symposium on Security and Privacy, May 11-14, Berkeley, CA., pp: 62-76.

Feng, D.G., M. Zhang, Y. Zhang and Z. Xu, 2011. Study on cloud computing security. J. Software, 22: 71-83.

Giffin, J.T., S. Jha and B.P. Miller, 2005. Efficient context-sensitive intrusion detection. Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection, September 2005, Seattle, Washington, pp: 1-15.

Hofmeyr, S.A., S. Forrest and A. Somayaji, 1998. Intrusion detection using sequences of system calls. J. Comput. Sec., 6: 151-180.

Junfeng, T. and Z. Ye, 2011. Trusted software constitution model based on trust shell. Adv. Mater. Res., 186: 251-255.

Lang, K.J., B.A. Pearlmutter and R.A. Price, 1998. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. Proceedings of the 4th International Colloquium on Grammatical Inference, July 12-14, Ames, Iowa, pp: 1-12.

Lorenzoli, D., L. Mariani and M. Pezze, 2008. Automatic generation of software behavioral models. Proceedings of the ACM/IEEE 30th International Conference on Software Engineering, May 10-18, Leipzig, pp: 501-510.

Oncina, J. and P. Garci, 1992. Inferring regular languages in polynomial updated time. Pattern Recognit. Image Anal., 1: 49-61.

Sekar, R., M. Bendre, P. Dhurjati and D. Bullineni, 2001. A fast automaton-based method for detecting anomalous program behaviors. Proceedings of the IEEE Symposium on Security and Privacy, May 14-16, Oakland, CA. USA., pp: 144-155.

Trusted Computing Group, 2007. TCG specification architecture overview. http://www. trustedcomputing group.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf.

Wagner, D. and D. Dean, 2001. Intrusion detection via static analysis. Proceedings of the IEEE Symposium on Security and Privacy, May 14-16, Oakland, CA., USA., pp: 156-168.

Walkinshaw, N. and K. Bogdanov, 2008. Inferring finite-state models with temporal constraints. Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, Sept. 15-19, L'Aquila, pp: 248-257.

Wen, L., D. Ying-Xia, L. Yi-Feng and F. Ping-Hui, 2009. Context sensitive host-based IDS using hybrid automaton. J. software, 20: 138-151.

Wepsi, A., M. Dacier and H. Debar, 2000. Intrusion detection using variable-length audit trail patterns. Proceedings of the 3rd International Workshop on Recent Advances in Intrusion Detection, Oct. 2-4, Toulouse, France, pp: 110-129.

Xiao, Q., Y.Z. Gong, Z.H. Yang, D.H. Jin and Y.W. Wang, 2010. Path sensitive static defect detecting method. J. Software, 21: 209-217.

Yanwen, Q., 2004. Software Behavior. Electronic Industry Press, China.