Software Behavior Model Based on Functional Slicing

Binglin Zhao[1, a], Yinhao Wang[1], Zheng Shan[1], Chao Fan[1]

[1]State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China

[a]email: gzu_zhaobl@126.com

Keywords: Software Model; Function slicing; System Call Analysis

Abstract. According to the problem that the accurate and efficiency of the software model, this paper proposes a new method to describe the software behavior. The method using program slicing to dynamic analysis the system calls of the running software, help the system to capture the key system calls. And then build a software model combining import system calls and its arguments. Experiments shows that it can construct the software model with high accuracy and can efficient detect the abnormal behavior.

Introduction

With the development of Internet and computer, software trustworthiness plays an important role in the computer security. When attackers control the software to do some malicious acts, the behavior will different from the normal software profile. Therefore, anomaly measurement with normal behavior model becomes a popular research in software trustworthiness. Most measurement systems monitor the system calls of the running software, and then build the system call sequence of the software.

There has been a large amount of research on software trustworthiness using system calls. Forrest et al. [1] capture the behavior of privileged process by short sequences of system calls and build the normal behavior feature library for intrusion detection. Based on Forrest et al.'s method, many researchers study on software behavior to build the software model. Wagner et al. [2] proposed FSA model and PDA model with static analysis. The HPDA model proposed by Liu et al. [3] combined static analysis and dynamic learning supplement to build system sequences. Li et al. [4] proposed the HFA combined static analysis and dynamic learning. Li et al. [5] proposed a software behavior automaton model based on system call and context. Bimabaum et al. [6] proposed an Intrusion Detection Systems using n-grams of object access graph. However, all of these software models do not involve function path or function properties about software behavior.

According to these problems, we proposed a software model for dynamic trustworthiness with program slicing, extend the dynamic slicing by using the system call classification as a function slicing policies, and combine the function properties and function path to build software behavior model.

Software Behavior Automaton Model

The software behavior automaton model is a 5-tuple: (P, S, E_0, E_S, S_T). The tuple is explaining as follows:

$P = \{ p | p \in Property\}$, P is the finite set of function slicing.

$S = \{ s | s \in Software\text{-}State\}$, S is the running state of the software, and *Software-State* is the set of the state. In our model, the set *Software-State* is defined as: $Software\text{-}State = \{AS \times AF\}$, where AS is the key system calls of the software and AF is the key function of the software.

$E_0 \in S$, E_0 is the set of the start point.

$E_S \in S$, E_S is the set of the end point.

Software has many functions, so every function has its start point and end point.

$S_T = \{ t| t\in Software\text{-}Transfer\}$, S_T is the state transform of the software, *Software-Transfer* is the set of the state transform.

System Call Classification and Trace

System call is an event, which happens at the user-kernel interface. In this paper, the classification of system call improves Xu et al.'s method [7]. In LINUX 2.6 there are about 300 system calls. Every call has its own threat level in the operation system and low threat level system calls assigned a larger number of system calls. Our method makes a concentrate on high level system calls. The high level calls are classified about five categories by their functionality, i.e. process, file system, network, module and signal. Every function has its start and end point, and the point can describe by the system call. Each call category is classified into three groups, i.e. start calls, system calls and end calls. The system call classification is reported in Table 1.

Table 1 System call categories

Call group	Start calls	System calls	End calls
File system	ioctl, open, dup, dup2, creat,	chmod, chown, chown32, fchmod, fchown, fchown32, lchown, lchown32, link, mknod, mount, rename, symlink, unlink, flock	close
Process	clone, execve, ptrace, fork, vfork	setfsgid, setfsgid32, setfsuid, setsuid32, setgid, setgid32, setgroups, setgroups32, setregid, setregid32, setresgid, setresgid32, setresuid, setresuid32, setreuid, seteuid32, setuid	exit, kill
Module	init_module		delete_mouule
Network	socket, bind	connect, accept, listen, socketpair, recv, recvfrom, recvmsg	close
Signal	pipe, shmget		msgctl, shmctl

File system plays an important role in the system security because it provides a complex interface to operating system services and hardware devices. The calls in table 1 which can change the file content, file attribute, file system depend and can take the file names as its parameters. The read system call or write system call to operation file needs the open's return parameter. Because the open's return parameter is file descriptors, and at end the close system call can over the operation of the file using close to kill the descriptor. So the start call is output the file descriptors and the end calls is finish the file descriptors.

The execve system call in process management is the key system call. In operation system, the execve can start every process, so it is also the start call.

In module group, attackers may use init_module or create_module to loading a malicious module.

In network group, socket initializes the socket and connects to the network, it is the start call.

In Linux operation system, there are many ways to monitor the system calls and its running arguments like loadable kernel module (LKM) and strace. For convenient, we chose the strace mechanism to monitoring the program in our system. Strace, a debugging utility, is included in the Linux operating system and is capable of monitoring system calls from all non-system processes [8].

Functional Slicing

Program slicing is a viable method to restrict the focus of a task to specific sub-components of a program. The concept of program slice was proposed by Weiser et al. [9]. After that, researchers proposed many other notions of program slices and different applications have different program slices. This paper we focus of a police to specific the function of a program. Due to the function classification, the model can pay more attention to the high threat level behavior.

Slicing rule is the basic of the program slicing. In this paper we proposed some functional slicing rules. These rules allow us to focus any function about the program. The rules can be described as follows:

1) (Pro, LN_0, Fun_0). In this rule, Pro is the running program which we monitor in the system, LN_0 is the slicing point and Fun_0 is the function that we focus on. This rule is about the set of system calls at the point LN_0 that affect the function Fun_0 in the program.

2) (Pro, LN_0, Res_0). In this rule, Res_0 is the system resource. This rule is about the set of system calls at the point LN_0 that affect the system Res_0 in the program.

3) (Pro, Fun_0). This rule is about the set of system calls in the program that affect the function Fun_0.

4) (Pro, LN_0, $Sysc_0$). In this rule, $Sysc_0$ is the one or some system calls in the system. This rule is about the set of system call $Sysc_0$ at the point LN_0 in the program.

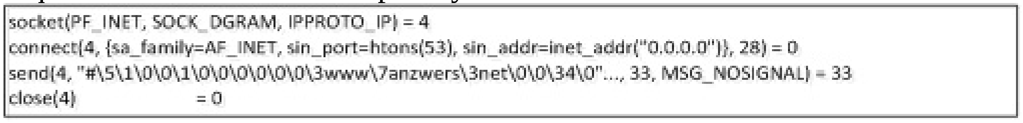
5) (Pro, $Sysc_0$). This rule is about the set of system call $Sysc_0$ in the program.

Slicing Algorithm

Algorithm. Functional slicing analysis
Input: Program running
Output: System call sequence of slicing
Step 1: Take the rule (Pro, LN_0, Fun_0), as the sample. Define the slicing rule, and send the rule to the slicing module.
Step 2: The slicing module read the rule. The Pro, LN_0 and Fun_0 are defined. LN_0 defined the end point of the sequence. Fun_0 defined the monitor function in the program.
Step 3: Running the program, and trace the system call of it.
Step 4: If the system call is in table 1 and the function of the system call is the subset of Fun_0, then the algorithm would carry on the next step. Otherwise, the algorithm returns to step 3.
Step 5: Record the system call to the slicing file, then the algorithm would carry to step 3.

Experimental Evaluation

To confirm our slicing is capable to trace the software behavior about any behavior, we ran several tests with the rule, and then capture the system calls. The rule is (Pro1, LN1, Network), Pro1 is a program that has many network operation and file operation, LN1 is the end trace point. Fig.1 shows the slicing result, and it shows that focus on a function with a program can reduce the size of the system call sequence and reduce the complexity of model.



```
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("0.0.0.0")}, 28) = 0
send(4, "#\5\1\0\0\1\0\0\0\0\0\0\3www\7anzwers\3net\0\0\34\0"..., 33, MSG_NOSIGNAL) = 33
close(4)                    = 0
```

Fig.1. Slicing Result

The accuracy is the different value between the forecast model and the truth model. To scale the accuracy of the model is good or not, it can be use the different value. The guide line of the model accuracy is as follows [10]:

$$MAD = \frac{\sum_{t=1}^{n} f_t}{c} \tag{1}$$

MAD (Mean Absolute Deviation), in the equation, f_n is the different value between the forecast model and the truth model at the state n. c is the total number of the experiment.

We use the “New Mexico University sendmail sample” as the experiment sample and drew a comparison between our model and HPDA.

There are 300 states in the model, and we set the number n as the state collection number, where n is the random state in software. Fig.2 shows the MAD comparison between two models, it shows that the MAD of our model is better than the HPDA.

The efficiency of the model can indicate by the runtime of the model. We compare the runtime between our model, HPDA and HFA model. We select 10 states as the compared point. As is shown in Fig.3, at every point our model has high efficiency than other models.

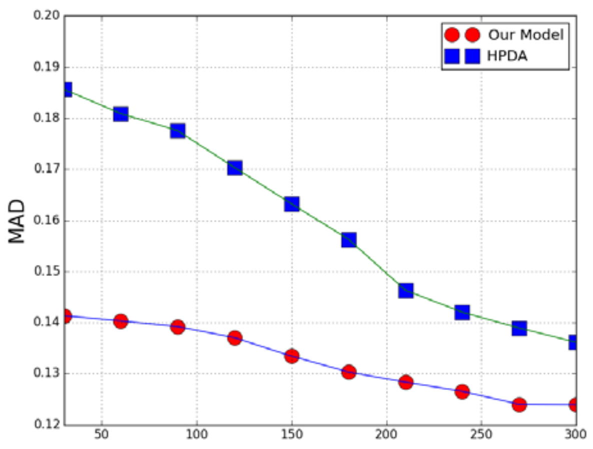


Fig.2. MAD

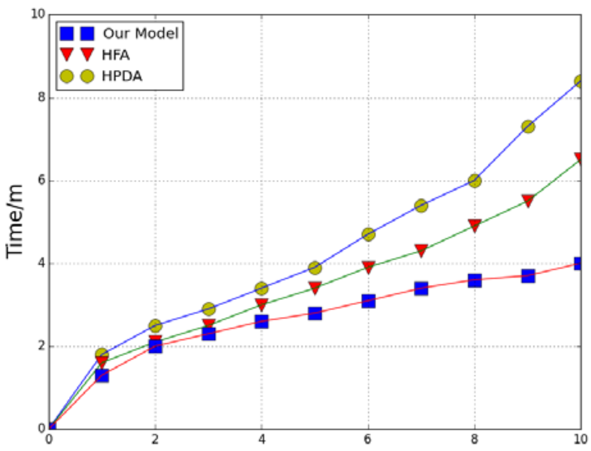


Fig.3. Runtime of model

Conclusions

Based on the static analysis, dynamic learning, system call trace and function slicing, the software behavior model has been built. The model is able to provide a new method to insure the software trusted. Expriments show that the model has high accurancy and efficiency.

Acknowledgements

The financial support of the State Key Laboratory of Mathematical Engineering and Advanced Computing are gratefully acknowledged.

References

[1] Forrest S, Hofmeyr S A, Somayaji A, et al. A Sense of Self for Unix Processes[J]. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, 1996.

[2] Wagner D, Dean D. Intrusion detection via static analysis[C]//Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on. IEEE, 2001: 156-168.

[3] Liu Z, Bridges S M, Vaughn R B. Combining static analysis and dynamic learning to build accurate intrusion detection models[C]//Information Assurance, 2005. Proceedings. Third IEEE International Workshop on. IEEE, 2005: 164-177.

[4] Wen L, Ying-Xia D, Yi-Feng L. Context Sensitive Host-Based IDS Using Hybrid Automaton[J]. Journal of Software, 2009.

[5] Li Z, Tian J. A software behavior automaton model based on system call and context[J]. Journal of Computers, 2011, 6(5): 889-896.

[6] B irnbaum Z, Dolgikh A, Skormin V. Intrusion Detection Using N-Grams of Object Access Graph Components[C]//ICDS 2014, The Eighth International Conference on Digital Society. 2014: 209-215.

[7] Xu M, Chen C, Ying J. Anomaly detection based on system call classification[J]. Journal of Software, 2004, 15(3): 391-403.

[8] Online, strace software, http://linux.die.net/man/1/strace

[9] Weiser M. Program slicing[C]//Proceedings of the 5th international conference on Software engineering. IEEE Press, 1981: 439-449.

[10] Gardner Everette S, Jr. Automatic monitoring of forecast errors. Journal of Forecasting, 2006, 2(1): 1-21.