Detection and Classification of Non-self Based on System Call Related to Security

Jimin Li [1,2] and Zhen Li [2]
[1] College of Computer Science and Technology,Tianjin University, Tianjin, China
[2]College of Mathematics and Computer, Hebei University, Baoding, China
ljm@hbu.edu.cn, lizhen_hbu@126.com

Kunlun Li
College of Electronic and Information Engineering, Hebei University, Baoding, China
likunlun@hbu.edu.cn

***Abstract*—Based on the immune mechanism, we present a computer system security model used to detect and classify non-self, which overcomes some drawbacks of traditional computer immune system based on system call: the large number of system calls intercepted, the loss of useful information owing to paying no attention to the arguments of system calls, distinction between self and non-self just by rule matching, etc. We introduce the process of non-self detection and classification based on rule and Sandbox further distinguishing the process of unknown type, based on the definition of system call related to security and event related to security. We resolve the problem of traditional sandbox system: the unreliability and insecurity of process and the display of process behavior incompletely caused by denying the execution of a system call. Experimental results verify the effectiveness of distinguishing non-self and its class based on system call related to security, and show that our model can detect non-self in Sandbox which is unknown type by rule matching without imposing heavy performance impact upon operating system.**

***Index Terms*—system call, computer immune, detection of non-self, classification, sandbox**

I. INTRODUCTION

As an important method to insure computer security, intrusion detection technique has become the current focus of research in the field of information security. The theory that the immune system can protect body from invasion provides a new thinking to the design of intrusion detection system.

Based on the immune mechanism, there are many studies about intrusion detection using system call describing process behavior [1,2,3]. However, traditional methods have some drawbacks.

First, because there are a large number of system calls produced by the running program, intercepting and storing all system calls must cause low efficiency. However, not all system calls are related to system security, so it is unnecessary to record all system calls.

Second, if we just focus on system calls and don't record the arguments of them, some useful information will be lost. This method is not enough to discriminate self and non-self and classify non-self. However, if the arguments of system calls are recorded, they will occupy a huge storage space and the subsequent analysis of system call sequences will cause low efficiency.

Third, the distinction between self and non-self by rule matching is inadequate. The process, which is unknown type by rule matching, needs to be intensively distinguished.

In order to solve the above problems, we propose a computer system security model based on system call related to security, which compensates the above drawbacks of traditional computer immune system based on system call. This model is structured by agents imitating the immune cells. Through cooperation the agents discriminate self and non-self and classify non-self.

First, non-self is detected and classified based on system call related to security in Imitated MC Agent and Imitated TH Agent, which reduces the number of system call recorded and improves efficiency.

Second, Sandbox is used to further distinguish process which is unknown type by rule matching. In Sandbox, we define different event related to security according to different arguments of each system call related to security. Non-self detection is realized by access control based on event related to security.

Third, we introduce Virtualization Sandbox which resolves the problem of traditional sandbox system: the unreliability and insecurity of process and the display of process behavior incompletely caused by denying the execution of a system call.

II. RELATED WORK

A. Immune Intrusion Detection Based on System Call

The research group of Stephanie Forrest in the University of New Mexico plays a leading role in the computer immune model and computer immune system application. They introduce a method for intrusion detection by monitoring system calls of privileged processes and designing intrusion detection system based on short sequences of system calls. They focus on sequence of system calls and ignore the arguments of

them [1]. The group of Wenke Lee in Columbia University extends the work pioneered by Forrest and applies a machine learning approach to learn normal and abnormal patterns of program behavior from its execution trace [4]. Some researchers present a novel technique to build a table of variable-length patterns or a dynamic window size model which has better detection effect [2]. Some other researchers propose intrusion detection based on system calls and homogeneous Markov chains [3], hidden Markov model for system call anomaly detection [5], motif extraction for system call sequence classification [6], etc.

B. *Sandbox*

Sandboxing is a technique for creating confined execution environments to protect sensitive resources from illegal access. A sandbox, as a container, limits or reduces the level of access its applications have. The concept of sandboxing is first introduced by Wahbe et al. in the context of software fault isolation [7]. Janus [8], to our knowledge, is the first to propose using these techniques. Systrace [9] expands on Janus by proposing novel techniques that efficiently confine multiple applications and support multiple policies.

Sandboxing can be used to improve security of file access [10], analyze malicious codes [11], make sure the data written with no sensitive information [12], etc. Many studies have improved the traditional sandbox toward dissimilar emphases: sandbox executing speculative security checks [13], sandbox with a dynamic policy [14], dynamic sandbox quarantining untrusted entities [15], etc.

Many sandbox systems are based on system call interception. System call interception-based sandbox systems restrict a process behavior by preventing the execution of any system call that would violate a predetermined security policy. They usually use binary permission that may be assigned one of two possible values: *allow* or *deny*. Denying the execution of a system call can have a detrimental impact on the operation of the process, potentially undermining its reliability and even its security [16], and have inconvenient trace record of intrusion because the process behavior cannot be displayed completely. Our computer system security model can resolve these problems.

III. The Structure of Computer System Security Model

The structure of computer system security model based on system call related to security is shown in Fig. 1. Imitate MC Agent discriminates self and non-self and realizes the function of immune detection. Imitate TH Agent classifies non-self that has been detected by imitate MC Agent. The process, which is unknown type, is migrated to Sandbox by Imitate TC Agent. Sandbox is used to intensively distinguish the process, which is unknown type passed by Imitate MC Agent and Imitate TH Agent, and create and update Self Rule Base and all kinds of Non-self Rule Bases.

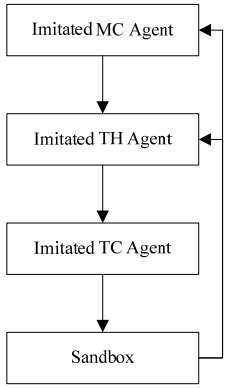

Figure 1. The structure of computer system security model

We mainly discuss Imitate MC Agent, Imitate TH Agent and Sandbox, which are closely related to non-self detection and classification. The structure of Imitate MC Agent is shown in Fig. 2. Collector records system calls related to security produced by process and forms the queue related to security. Detector detects short sequences of system calls by Self Rule Base, and sends non-self short sequences of system calls to Imitate TH Agent. Self Rule Base is created and updated by Sandbox.

The structure of Imitate TH Agent is shown in Fig. 3. Imitate TH Agent receives non-self short sequences of system calls related to security from Imitate MC Agent, classifies non-self, and sends non-self which is unknown type to Imitate TC Agent. All kinds of Non-self Rule Bases are created and updated by Sandbox.

IV. Related Definitions

There are a large number of system calls produced by the running program. Intercepting and storing all system calls must cause low efficiency. Considering that not all system calls are related to system security, we just focus on system calls related to security. This can not only

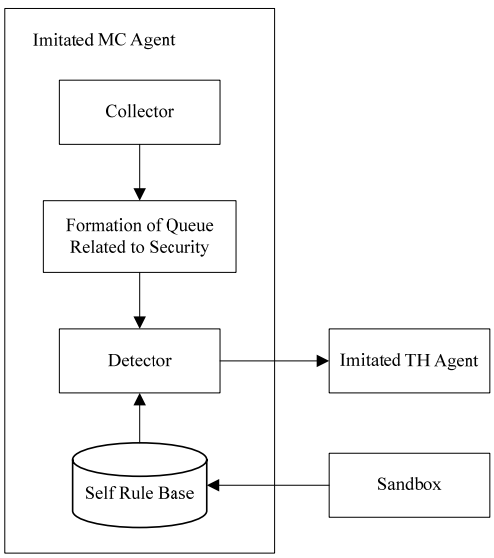

Figure 2. The structure of Imitated MC Agent

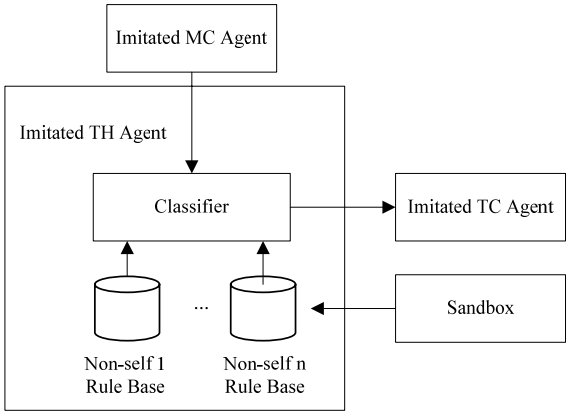

Figure 3. The Structure of Imitated TH Agent

detect non-self successfully, but also improve efficiency.

A. System Call Related to Security

System call related to security refers to the system call that is relevant to security in the system. Take *open* system call for example, its prototype is as follows:

int *open*(const char **pathname*, int *flag*, mode_t *mode*);

The *pathname* argument points to the path of the file to open. The *flag* argument must have a set bit corresponding to exactly one of O_RDONLY (open file read only), O_WRONLY (open file write only) and O_RDWR (open file for read/write), and zero or more of O_APPEND (set the file offset to EOF for appending), O_CREAT (create the file if necessary, use the *mode* argument), etc. The *mode* argument is used to set the file's access mode at creation time.

Another example is *chmod* system call. The prototype is:

int *chmod*(const char **pathname*, mode_t *mode*);

It changes the mode of the file specified by *pathname* to *mode*. The arguments are the same meaning as they are used in the *open* system call.

From the arguments and realized functions of *open* and *chmod* system call, we can see that their access modes and access permissions are closely related to system security. Therefore, these system calls are defined as system calls related to security.

B. Event Related to Security

If we just focus on system calls and don't record the arguments of them, some useful information will be lost, which causes inaccurate detection of non-self. In view of this, we introduce the definition of event related to security.

We define different events related to security according to different arguments of each system call related to security. For example, for *open* system call related to security, we define some events related to security: OPEN_RD, OPEN_WR, OPEN_RW, OPEN_CREATE, and so on.

In Linux, the introduction of system call related to security and event related to security not only reduces the number of system calls recorded, but also takes arguments of system calls into account. Therefore, non-self can be detected with high efficiency and accuracy.

C. Queue Related to Security

Queue related to security is a queue which element is a system call related to security.

We slide a window of size W across the sequence of system calls related to security. In the real-time detection, we use a circle queue with size W to realize the queue related to security in order to make full use of space.

Queue related to security is an empty queue initially. In the process of interception, system calls related to security enter the queue related to security. When the number of system calls related to security in the queue reaches the specified window size W, that is, when the queue is full, a short sequence of system calls related to security is formed. At this time, the sleeping process is awakened and detected. Then, the front of queue related to security is out of the queue and the next system call related to security enters the queue. Thus, a new short sequence of system calls related to security is formed, which awakens the sleeping process again. The cycle continues until all the system calls related to security have been intercepted.

D. Correlation Degree and Correlation Coefficient

The correlation degree $R(i, \text{Nonself}_j)$ of short sequence of system calls i and the jth non-self class Nonself_j is defined as follows:

$$R(i, \text{Nonself}_j) = \begin{cases} 1, & \text{the short sequence of system call } i \text{ matches a rule in Non-self } j \text{ Rule Base} \\ 0, & \text{else} \end{cases} \quad (1)$$

N denotes the number of short sequences of system calls of non-self program Nonself_test deleting the short sequences of system calls that Self Sequence Base has.

The correlation coefficient r (Nonself_test, Nonself_j) of non-self program Nonself_test and the jth non-self class Nonself_j is defined as follows ($0 \le r \le 1$):

$$r(\text{Nonself\_test}, \text{Nonself}_j) = \frac{\sum_{i=1}^{N} R(i, \text{Nonself}_j)}{N} \quad (2)$$

Ⅴ. Detection and Classification of Non-self Based on Rule

In Imitate MC Agent, Detector decides whether the short sequence of system calls related to security is self or not by Self Rule Base. If the short sequence of system calls related to security is non-self, it is sent to Imitate TH Agent.

The intrusion behavior happens suddenly, so the non-self short sequence of system calls related to security shows aggregation. In real-time detection, every N short sequences of system calls related to security as an area are a group according to the sequence of interception. If the proportion of non-self short sequences of system calls related to security in an area is more than threshold r, the process is reported as non-self process. If the proportion

of non-self short sequences of system calls related to security in any area isn't more than threshold *r*, the process is reported as self process.

In Imitate TH Agent, for an area of N short sequences of system calls related to security, if the correlation coefficient of non-self process and a certain non-self class is more than the threshold *s*, the process is reported as this non-self class. Otherwise, the process is reported as unknown non-self class and is sent to Sandbox by Imitate TC Agent.

VI. SANDBOX

After the process passes by Imitate MC Agent and Imitate TH Agent, the process reported as unknown non-self class mainly includes two classes: self process that is undetected by Imitate MC Agent and non-self process of unknown class that isn't distinguished by Imitate TH Agent. Therefore, it is necessary to distinguish intensively non-self from self.

Sandbox distinguishes process, which is unknown type after passing by Imitate MC Agent and Imitate TH Agent, creates and updates Self Rule Base and all kinds of Non-self Rule Bases in Imitate MC Agent and Imitate TH Agent. The architecture of Sandbox is shown in Fig. 4. The procedure of non-self detection is as follows:

(1) The process is migrated to sandbox by Imitate TC Agent. The Collector intercepts system calls related to security.

(2) In System Call Processing Module, System Call Related to Security Gateway requests a policy decision from Security Policy Cache in kernel for each system call related to security.

(3) If there is no permission information of the object in Security Policy Cache, Security Policy Cache consults Security Policy Database in userland.

(4) System Call Processing Module receives the result from Security Policy Cache. If the result is *allow*, resume the execution of system call. Otherwise, enter Virtualization Sandbox to access the copy of object without sensitive information.

(5) Finally, the system call result returns to Collector and the process resumes.

A. Security Policy Database

Security Policy Database provides security policy for Sandbox. Security Policy Database defined by BNF is as follows:

<*request*>::=<*access_modes*><*objects*><*bina_perm*>
<*access_modes*>::= {<*access_mode*>","}<*access_mode*>
<*objects*>::=<*file_object_list*>|<*dir_object_list*>|
<*device_object_list*>|<*IPC_object_list*>|<*SCD_object_list*>|<*process_object_list*>
<*bina_perm*>::= *allow* | *deny*
<*access_mode*>::= OPEN_RD | OPEN_WR | CHANGE_OWNER| EXECUTE | …

Among them, *access_modes* denotes the access request set, that is, the set of event related to security. *object* denotes a system entity on which an operation can be performed. It includes file, directory, device, inter-process communication (IPC), system control data (SCD) and process. *bina_perm* denotes a binary permission that may be assigned one of two possible values: *allow* or *deny*.

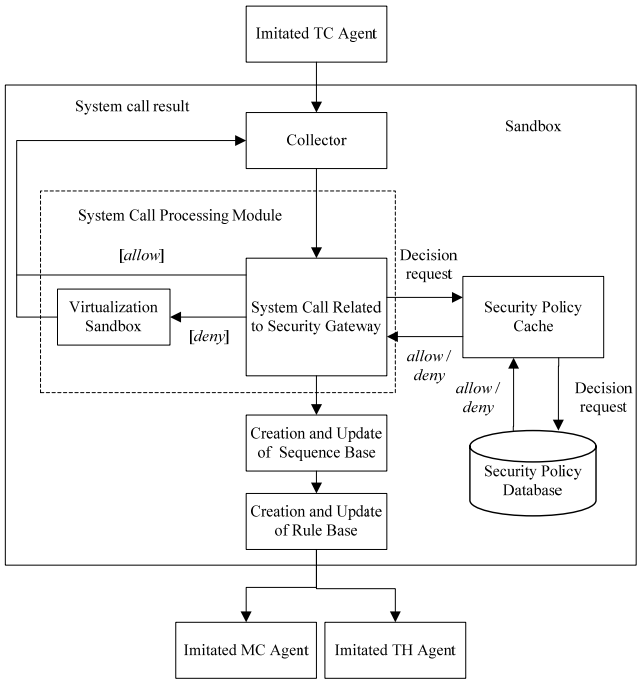

Figure 4. The structure of Sandbox

B. Security Policy Cache

In order to improve the decision efficiency, we put access permission of objects visited recently into Security Policy Cache in kernel and make sure that frequently accessed objects can be accessed quickly and efficiently. We adopt a technique of splay tree which can achieve our goal by being self-adjusting. Nodes that are frequently accessed will frequently be lifted up to become the root, and they will never drift too far from the top position. Inactive nodes, on the other hand, will slowly be pushed farther and farther from the root.

Let's take file system as an example. The location information of an object is uniquely determined by <*I*, *D*>. *I* denotes inode number and *D* denotes the device number in the file system the inode belongs to. In order to distinguish objects in different file system, each file system has an independent cache which creates when file system loads and revokes when file system unloads. For each cache, access permission is queried by splay tree based on inode number as a key.

C. Interception of System Calls Related to Security

In Collector, we intercept system calls using loadable kernel module (LKM) in Linux. The implementation of each system call related to security is modified to make policy decision. Take *open* system call for example.

int *new_open*(char **filename*, int *flags*, int *mode*)
// the modified *open* system call
{
get request access permission *RequestPerm* by parameter values;

send decision request（*filename, RequestPerm*）to Security Policy Cache;
receive the processing result *bina_perm* from Security Policy Cache;
if the processing result is deny
enter Virtualization Sandbox;
return (**old_open*)(*filename*,*flags*,*mode*);
// the original *open* system call
}

D. Virtualization Sandbox

In the process of system call interception, denying the execution of system call can have a detrimental impact on the operation of the application, potentially undermining its reliability and even its security, and have inconvenient trace record of intrusion because the behavior of application cannot be displayed completely. Virtualization Sandbox is used to resolve the problems caused by denying system calls.

Virtualization Sandbox protects sensitive resources by executing system calls no matter whether the decision result is *allow* or not. If the decision result is *allow*, system call executes normally. If the decision result is *deny*, the application enters Virtualization Sandbox after an intrusion alarm. System call executes in Virtualization Sandbox and access the copy of sensitive resources without sensitive information. We take file system as an example to explain the design ideas of Virtualization Sandbox.

Virtualization Sandbox can be implemented by virtual machine such as User Mode Linux. However, the performance cost is considerable. For the above reason, our design ideas in Linux are as follows: We construct Virtualization Sandbox using *chroot* to change the root file system to a target directory and using NFS to resolve synchronization problems between directories/files in Virtualization Sandbox and non-sensitive directories/files in the original system.

E. Creation and Update of Sequence Base and Rule Base

Each record of Sequence Base includes the short sequence of system calls related to security, class and emergence times. The emergence times represent the confidence of belonging to this Sequence Base.

If the system call of self process intercepted is system call related to security, it enters the queue related to security. When the queue related to security is full, a short sequence of system calls related to security is formed. If the Self Sequence Base doesn't have this short sequence of system calls related to security, put it in Self Sequence Base. Its class is "self" and the emergence times are 1. Otherwise, the emergence times of the record of this short sequence of system calls related to security in Self Sequence Base add 1.

The creation of Non-self *i* Sequence Base is similar to that of Self Sequence Base. It must be point out that the short sequence of system calls which is put in Non-self *i* Sequence Base should not exist in Self Sequence Base. For all kinds of Non-self Sequence Base, the repeated records in different Non-self Sequence Base, that is records of shared non-self class, are deleted. These records just show non-self, but they cannot show the non-self class.

We create Rule Base using C4.5 algorithm for 80% records that the confidence is relatively high in each Sequence Base. With the update of Sequence Bases, the pruning of low confidence records can delete some records emerged once in a while or out of date, so the records of Sequence Bases can reflect the latest process behavior.

Ⅶ. EXPERIMENTS

A. Short Sequence of System Calls Related to Security Used to Distinguish Different Programs

The following experiments are used to compare the capacity of distinguishing different programs based on short sequence of system calls and short sequence of system calls related to security. If they cannot distinguish different programs, it is impossible for them to be data source for non-self classification.

We choose three common commands ls, ps and vi in Linux for experiments. We train these three commands with different arguments to create three Sequence Bases respectively based on short sequence of system calls and short sequence of system calls related to security, create corresponding Rule Bases using C4.5 algorithm, and then test these three commands with other arguments.

The experimental results of Wenke Lee show that the accuracy of classification model with the size of sliding window from 6 to 14 doesn't have obvious differences [4]. Considering computing cost, we choose 7 as the size of sliding window. We use C4.5 algorithm to form 251 rules which error rate is 1.8%, and create Rule Base with rules which confidence is more than 0.85. The results are shown in Table Ⅰ.

In the experiments based on system call related to security, we choose 7 as the size of sliding window, use C4.5 algorithm to form 218 rules which error rate is 2.2%, and create rule base with rules which confidence is more than 0.85. The results are shown in Table Ⅱ.

From the above experimental results, we can learn that both short sequence of system calls and short sequence of system calls related to security are able to distinguish different programs. However, the number of programs in training set is less, so there are relatively more short sequences of unknown class detected. Compared with recording all system calls of process, recording system calls related to security loses some details about program running to some extent and makes error rate increase lightly, but the detection based on short sequence of system calls related to security still can distinguish different programs. Meanwhile, because the number of system calls related to security intercepted is less than that of system calls, so the number of rules is less, then the space used to store rules and the matching time are less. Thus there is higher running efficiency in the real-time detection.

B. Distribution of Non-self

We perform experiments over the synthethic sendmail data set collected by the University of New Mexico [17]. In experiments, we realize non-self detection and classification based on system call related to security. Synthethic sendmail data set has 48 system calls related to security. We extract system calls related to security from the sequence of system calls of each program in the data set and slide a window of size 7 across the sequence of system calls related to security. We train self programs and sscp, decode, following loops three non-self programs, create Sequence Bases, and then create Rule Bases using C4.5 algorithm. The total situation of non-self short sequences of system calls related to security in test programs is shown in Table Ⅲ. *r* denotes the correlation coefficient of non-self program and the non-self class it belongs to.

The intrusion behavior happens suddenly, so the non-self short sequences of system calls related to security show aggregation. It isn't enough to distinguish non-self from self or different classes of non-self by the proportion of non-self short sequences of system calls related to security in the total number of short sequences or the proportion of a certain non-self short sequences of system calls related to security in the total number of non-self short sequences. We use local statistical method to detect and classify non-self.

We test all kinds of non-training programs and the results of test programs are similar. Let's take the test program of sscp non-self class for example.

There are 350 short sequences of system calls related to security made by test program. Compared with 428 short sequences of system calls made by test program, the number of short sequences of system calls related to security decreases significantly, which improves time and space efficiency. Every 35 short sequences of system calls related to security as an area are a group and the program is divided into ten areas to analyze. In the first few areas, some unknown non-self class short sequences of system calls related to security are intercepted and most of them are shared non-self class. When the program runs more than half, there are lots of short sequences of system calls related to security of sscp non-self class emerged. In the experiment, the threshold r=70%, s=80%. When the program runs to the 7th area, the proportion of non-self short sequences of system calls related to security in the total number of short sequences is 77.1%, and the proportion of sscp non-self class short sequences of system calls related to security in the total

TABLE Ⅰ.
THE DETECTION RESULTS BASED ON SYSTEM CALL

Test program	The number of short sequences belonging to the class(%)				Total number	The class belonged to
	ls class	ps class	vi class	unknown class		
ls -n	106(60.9%)	0	0	68(39.1%)	174	ls class
ls -color	133(64.3%)	0	0	74(35.7%)	207	ls class
ps -x	19(1.5%)	1010 (81.0%)	0	218 (17.5%)	1247	ps class
ps -l	37(3.1%)	869 (73.2%)	0	281 (23.7%)	1187	ps class
vi /pattern filename	29(6.4%)	1(0.2%)	337(74.1%)	88(19.3%)	455	vi class

TABLE Ⅱ.
THE DETECTION RESULTS BASED ON SYSTEM CALL RELATED TO SECURITY

Test program	The number of short sequences belonging to the class(%)				Total number	The class belonged to
	ls class	ps class	vi class	unknown class		
ls -n	59(57.8%)	5(4.9%)	0	38(37.3%)	102	ls class
ls -color	74(60.2%)	7(5.7%)	0	42(34.1%)	123	ls class
ps -x	7(0.7%)	807(75.2%)	0	259(24.1%)	1073	ps class
ps -l	31(3.0%)	736(72.2%)	0	253(24.8%)	1020	ps class
vi /pattern filename	4(1.2%)	5 (1.5%)	232(72.4%)	80(24.9%)	321	vi class

TABLE Ⅲ.
THE EXPERIMENTAL RESULTS OF TEST PROGRAM

Test program	Different classes of non-self				Total number of non-self	*r*
	sscp class	decode class	following loops class	unknown class		
nonself_sscp	61	1	2	34	98	0.62
nonself_decode	0	8	1	2	11	0.73
nonself_following loops	2	0	52	24	78	0.67

number of non-self short sequences is 96.3%. Therefore, when the program runs to the 245th short sequence of system calls related to security, sscp non-self class can be detected accurately.

The distribution of non-self in the experiment is shown in Fig. 5. When the abscissa is n, the ordinate denotes the number of a certain non-self class short sequences of system calls related to security in the area of (n-35, n] ($35 \le n \le 350$).

C. Test of Sandbox

Here we just use a simple test case to explain the process of intrusion detection when the process is migrated to Sandbox. The test case is that a normal user tries to open /etc/shadow file. The system gives an alarm when the program runs to the operation of opening /etc/shadow file. The program doesn't terminate but continues executing. The actual file opened is /etc/shadow file in Virtualization Sandbox without sensitive information. Sandbox detects intrusion successfully and records the relevant system calls traces for the creation and update of Sequence Bases.

In order to test the performance of sandbox impacting upon the original operating system, we make following experiments.

We have implemented the sandbox and experimented on a PC with AMD Phenom(tm) 8400 Triple-Core Processor 2.10GHz and 2GB of main memory running Linux kernel 2.4.20. We test the execution time of file operation *open*() and *close*() respectively in three modes: the original system, sandbox making a policy decision based on Security Policy Cache and sandbox making a policy decision based on Security Policy Database. In the third mode, Security Policy Cache isn't involved temporarily. We take the depth of file into account because the execution time of file operation depends on the depth of file in the directory tree. The same operation is repeated 10,000 times and we calculate the time for executing the operation once by obtaining the average value after many tests. The operation consists of opening a file and immediately closing it. The experimental results are shown in Fig. 6.

For the original system, the execution time of *open*() and *close*() increases with the increase of the depth of file.

For sandbox making a policy decision based on Security Policy Cache, the file node opened is surely located in the root through the reconstruction of splay tree because the operation is repeated 10,000 times. Therefore, it is the best case to searching this file node that can be searched successfully with only once. The depth of file basically has no effect on the time of policy decision.

For sandbox making a policy decision based on Security Policy Database, as the third mode we test, policy decision is based on Security Policy Database every time because Security Policy Cache is not involved temporarily. The execution time of *open*() and *close*() is longer than the first two modes. Moreover, the execution time increases faster with the increase of the depth of file. When the depth of file is 1, the execution time is 10.4 times that of the original system. Despite this, the performance is substantially better than that of Systrace [9] testing on *open*(). When Systrace makes policy decision in userland and the depth of file is 1, the execution time is 25 times that of the original system.

Furthermore, it can be shown that the performance loss after sandbox being incorporated to the original system can be obviously reduced through improving the hit ratio in Security Policy Cache.

Ⅷ. CONCLUSION

We present a computer system security model based on system call related to security, which overcomes some drawbacks of traditional computer immune system. This model is structured by agents imitating the immune cells. Through cooperation the agents discriminate self and non-self and classify non-self. Firstly, we give relative definitions and introduce the process of non-self detection and classification based on rule and Sandbox distinguishing intensively the process of unknown type.

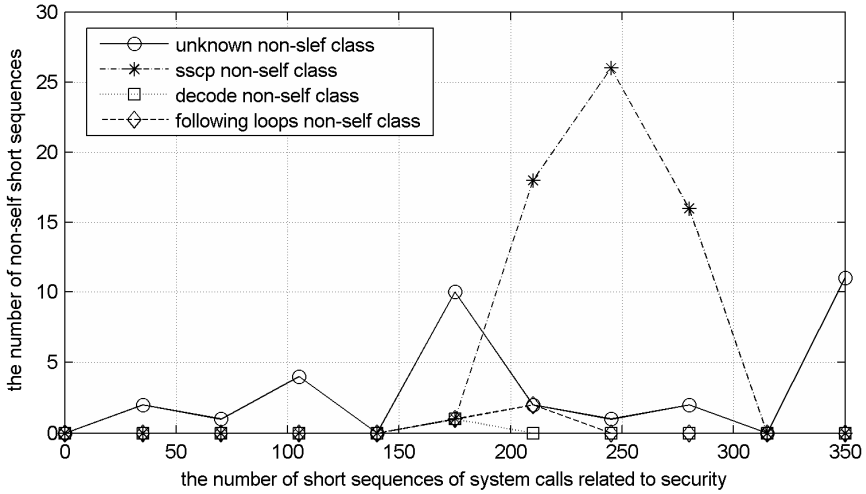

Figure 5. The distribution of non-self of sscp non-self class test program

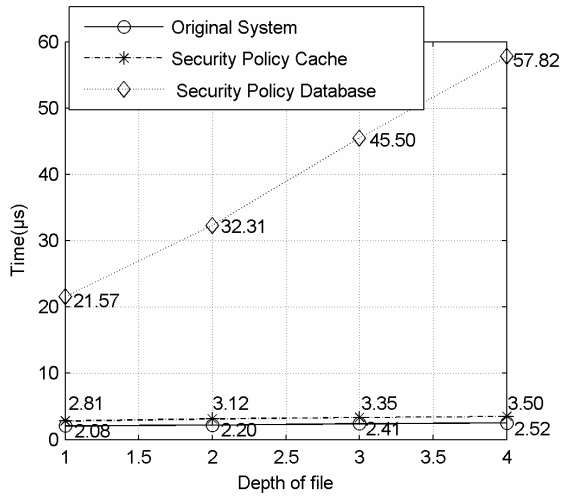

Figure 6. Comparison of the execution time in three modes

Secondly, we describe Virtualization Sandbox which ensures the reliability and security of process and makes process behavior display completely. Finally, experimental results show that the model based on system call related to security can distinguish non-self class accurately, and can detect non-self which is unknown type by rule matching without imposing heavy performance impact upon operating system.

REFERENCES

[1] S. Forrest, S.A. Hofemeyr, A. Somayaji, and T.A. Longstaff, "A sense of self for UNIX processes," in *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, IEEE Computer Society Press, Los Alamitos, CA, pp. 120-128, 1996.

[2] E. Eskin, W. Lee, and S. J. Stolfo, "Modeling system calls for intrusion detection with dynamic window sizes," in *Proceedings of DISCEX II, Anaheim, CA*, IEEE Computer Society Press, pp. 165-175, 2001.

[3] X. G. Tian, M. Duan, C. L. Sun, and W.F. Li, "Intrusion detection based on system calls and homogeneous Markov chains," *Journal of Systems Engineering and Electronics*, vol. 19, pp. 598-605, 2008.

[4] W. Lee, S. J. Stolfo, and P. K. Chan, "Learning patterns from unix process execution traces for intrusion detection," in *Proceedings of AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, pp. 191-197,1997.

[5] Q. Qian and M. J. Xin, "Research on hidden Markov model for system call anomaly detection," *PAISI 2007*, *LNCS*, pp. 153-159, 2007.

[6] J. W. Li, X. H. Zhang, C. Yuan, Z. H. Jiang, and H. Q. Feng, "Motif extraction with indicative events for system call sequence classification[," *Fuzzy Systems and Knowledge Discovery*, pp. 611-616, August 2007.

[7] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation", in *Proceedings of the Symposium on Operating System Principles*, 1993.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, " A secure environment for untrusted helper applications: confining the wily hacker," in *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[9] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th USENIX Security Symposium*, pp. 257-273, August 2003.

[10] L. Peng, "The Sandbox: Improving File Access Security in the Internet Age," May 2006.

[11] S. Miwa, T. Miyachi, and M. Eto, "Design and implementation of an isolated sandbox with mimetic internet used to analyze malwares," in *Proceedings of the DETER Community Workshop on Cyber-Security and Test*, 2007.

[12] T. Khatiwala, R. Swaminathan, and V.N. Venkatakrishnan, "Data sandboxing: a technique for enforcing confidentiality policies," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, pp. 223-234, 2006.

[13] Y. Oyama, K. Onoue, and A. Yonezawa, "Speculative security checks in sandboxing systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[14] T. Shioya, Y. Oyama, and H. Iwasaki, "A sandbox with a dynamic policy based on execution contexts of applications," *ASIAN' 2007*, pp. 297-311, 2007.

[15] M. Radhakrishnan and J. A. Solworth, "Quarantining untrusted entities: dynamic sandboxing using LEAP," in *Proceedings of 23rd Annual Computer Security Applications Conference,* December 2007.

[16] T. Garfinkel, "Traps and pitfalls: practical problems in system call interposition based security tools," in *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 2003.

[17] http://www.cs.unm.edu/~immsec/data-sets.htm.

Jimin Li, born in 1969, received M.S. degree in Computer Application from Hebei University of China in 2001. He is studying for PhD degree in College of Computer Science and Technology of Tianjin University, China. He is an Associate Professor at College of Mathematics and Computer of Hebei University. His main research interests include network security, machine learning and data mining. In these areas, he has published over 10 technical papers in refereed international journals or conference proceedings.

Zhen Li, born in 1981, received B.S. degree in Computer Science and Technology and M.S. degree in Computer Application from Hebei University, Baoding, China, in 2003 and 2006. She joins the College of Mathematics and Computer of Hebei University at present. Her current research interests include computer security, distributed computing and machine learning. She has published over 5 technical papers in refereed journals and conference proceedings.

Kunlun Li, born in 1962, received the PhD degrees in Signal & Information Processing from Beijing Jiaotong University, China, in 2004 and join College of Electronic and Information Engineering of Hebei University as the associate professor at present. His main research interests include machine learning, data mining, intelligent network security and biology information technology. In these areas, he has published over 20 technical papers in refereed international journals or conference proceedings.