An Anomalous Behavior Detection Method Using System Call Sequences for Distributed Applications

Chuan Ma[1,3], Limin Shen[1,3] and Tao Wang[1,2,3]
[1] School of Information Science and Engineering, Yanshan University
Qinhuangdao, 066004 - China
[e-mail: tianyi_mc@126.com]
[2]Hebei Normal University of Science & Technology
Qinhuangdao, 066004 – China
[e-mail: yy_mma@126.com]
[2]The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province
Qinhuangdao, 066004 - China
*Corresponding author: Chuan Ma

Received July 10, 2014; revised November 20, 2014; accepted December 22, 2014; published February 28, 2015

Abstract

Distributed applications are composed of multiple nodes, which exchange information with individual nodes through message passing. Compared with traditional applications, distributed applications have more complex behavior patterns because a large number of interactions and concurrent behaviors exist among their distributed nodes. Thus, it is difficult to detect anomalous behaviors and determine the location and scope of abnormal nodes, and some attacks and misuse cannot be detected. To address this problem, we introduce a method for detecting anomalous behaviors based on process algebra. We specify the architecture of the behavior detection model and the detection algorithm. The anomalous behavior detection and analysis demonstrate that our method is a good discriminator between normal and anomalous behavior characteristics of distributed applications. Performance evaluation shows that the proposed method enhances efficiency without security degradation.

Keywords: Behavior detection, distributed applications, anomalous behavior, process algebra, system call

This research was supported by the National Natural Science Foundation (61272125) of China, the specialized research fund for the doctoral program of Higher Education (20121333110014), the Hebei Provincial Natural Science Foundation (F2011203234) and the science and technology plan project of Qinhuangdao（201401A053）.

1. Introduction

With the popularity and deepening of the network, distributed computing has to expand from its origins in shared-memory computing and local area networks to a wider context [1]. A distributed application is a program distributed on independent computers that exchange information with individual nodes by message passing to accomplish a common task. Compared to the traditional applications, distributed applications have many advantages such as providing increasingly complex functionality and acceptable performance that require parallelizing their operations on individual nodes. Unfortunately, a large complex behavior patterns appear in the distributed applications due to the nodes changing frequently [2]. Nowdays, distributed systems have become an increasingly important platform for providing multiple services, for example, distributed systems form the basis for critical infrastructure in different domains such as finance, medicine, aeronautics, telephony, and Internet services. The quality of such distributed systems is often crucial [3]. However, the development process of distributed applications is difficult and inefficiency. Its development cycle is long, and it is difficult to avoid and find implicit errors and defects. Meanwhile, hackers continually explore and develop new methods to perform attacks on distributed systems. Therefore, security for distributed applications is a concern, and providing safe, efficient parallel implementations of distributed applications remain a challenge.

There are many different security paradigms to protect distributed applications from some attacks. Irfan Gul and M. Hussain proposed an efficient multi-threaded distributed cloud IDS model to handle large scale network access traffic, avoiding sophisticated distributed intrusion attacks like Distributed Denial of Service (DDOS) [4]. Collberg et al. present a new general technique for protecting clients in distributed systems against Remote Man-at-the-end (R-MATE) attacks [5]. Idrees et al. proposed framework is an amalgamation of some of the existing state-of-the-art intrusion detection and prevention technologies for detection and prevention of known and unknown network and cloud computing vulnerabilities [6]. Many distributed real-time systems are often safety-critical and need to be certified, however their certification is hard due to their distributed nature. Meseguer et al. presented a formal model transformation that maps a synchronous design to reduce the design and verification complexities of achieving virtual synchrony [7]. Several assumptions have usually been overemphasized in the above security paradigms: That security policy can be distinctly and correctly specified, that distributed applications can be correctly implemented, and that systems can be correctly configured. In fact, distributed applications are not static: their distributed nodes being continually changed by connecting or disconnecting; Applications are added and removed, and configurations are changed. Anomalous behaviors in distributed applications are often hard to find. Many anomalous behaviors reflect discrepancies between a system's behavior and the programmer's assumptions about that behavior.

In this paper, we can improve security through detecting anomalous behaviors based on process algebra. This method is based on the assumption that distributed applications can be correctly designed, but that violations of security policy can be detected by monitoring and analyzing software behaviors. The basic steps are as follows: We obtain control flow graphs (CFG) of individual nodes by static binary code analysis. Then we transform the control flow graphs into the corresponding process expressions automatically by using the technique in our previous literature [8], and rewrite process expressions by eliminating the non determinism and adding concurrency operators. Finally, we construct a behavior detection model of distributed applications and give the detection algorithm.

The contributions can be summarized to the following four points.

1) The interactions and concurrent behaviors of distributed applications can be described accurately. We apply process algebra to distributed applications behavior modeling, which is a profitable security paradigms attempt.

2) We propose the concurrent mechanism in the distributed applications that is structured as synchronization process set and running state set. This reduces the complexity of behavior analysis and detection.

3) Our model only maintains the linear list, and does not use the backtracking algorithm because the non determinism of process expressions was eliminated. This reduces the runtime overheads.

4) We provide a formal analysis method for developers that are familiar or unfamiliar to distributed applications. And we also provide an anomalous behavior detection method for the system maintenance personnel.

The structure of this paper is as follows. In Section 2, we review the previous work in analysis and detection methods for distributed applications. Section 3 introduces process algebra. Section 4 specifies the construction process of our model. The experimental evaluation is discussed in Section 5, and we conclude the paper in Section 6.

2. Related Work

Applications use system calls to gain access to functions from an operating systems kernel. Therefore, it is theoretically possible to detect when a hacker may be exploiting a program by analyzing system call patterns of an application [9]. Since the original development of a model that takes advantage of the system call sequence for normal behavior of a program was presented by Forrest et al. [10-11], many scholars have researched behaviors using the system call. By dynamic training or static analysis, scholars use the system call analysis for behavior analysis and detection. These techniques can be divided into three categories: system call short sequences [11-13], automata [14-17], and the Virtual Path [18]. Of these techniques, modeling based on system call short sequences is efficient and can be implemented easily. However, this method is imprecise, and these intrusion detection models are much more prone to false positives. Compared with short sequences, branch and loop structures of programs can be expressed. Modeling based on automata improves the precision of behavior modeling and reduces the false positive rate. Unfortunately, these models still have some limitations. For example, impossible paths, prohibitively high space–time complexity, and they are unsuitability for analyzing concurrent behaviors.

There are many different analysis and detection methods for distributed systems. Moshirpour et al. proposed a method for detecting emergent behavior, which is an important issue in distributed systems' design [19]. Distributed and concurrent object-oriented applications are difficult to analyze due to the complexity of their concurrency, communication, and synchronization mechanisms. Din C. C. et al. established a proof system for partial correctness reasoning based on communication histories and class invariants, which allow components to be analyzed independently of their environment [3]. Yang Fan et al. designed an aspect-oriented programming language based on distributed tuple spaces to enforce security policies for distributed systems [20]. To address the problem of testing the large-scale network services for complex performance problems and configuration errors, Gupta et al. presented an approach to test distributed systems in which they multiplex all of the nodes [21]. As modern data centers run a variety of applications, detecting failures in distributed systems have limited scalability, or have results that are hard to interpret. Tan et al. presented a light-weight technique to quickly detect performance problems in distributed systems using only correlations of OS metrics [22]. Rohr et al. introduced a workload intensity sensitive timing behavior analysis method for distributed multi-user systems, which consider inter-dependencies between concurrent execution operations within a distributed system to reduce the standard deviation for succeeding analysis steps [23]. Moshirpour et al. proposed the utilization of an ontology-based approach to detect emergent behavior in distributed systems by a set of message sequence charts [24]. These methods solve some problems in distributed systems at a certain aspect.

In this paper, we address a distributed application that is composed of multiple computing subjects that accomplish computing tasks by cooperating with each other. The monitoring node in this paper refers to the deployment node of a distributed application—for instance, an application in a process space or an application deployed on a host. These nodes have a unified synchronous clock, so we can determine behavior traces by the occurrence order.

3. Process Algebra

Process algebra is a mathematical tool used for depicting concurrent systems [25-26], and is used for researching concurrent, distributed, interactive systems [27]. The "process" mentioned in process algebra refers to the behavior patterns that are shown by the distributed applications. That consists of a series of actions and the operators that are subordinate to the limited action set. This paper uses process algebra to describe the behaviors of distributed applications. We extract a common subset, the basic component of process algebra, containing the sequential composition operator (.), alternative composition operator (+), and parallel composition operator ($\|_A$). Let A be a finite set of synchronous actions (A). The syntax specifications are defined as follows:

$$P ::= \underline{0} \mid \surd \mid a.P \mid P_1 + P_2 \mid P_1 \|_A P_2$$

Their corresponding meanings are as follows:

1) $\underline{0}$ stands for the process down time, no action is performed. $\surd$ stands for process terminated successfully.

2) $a.P$ stands for prefix action a, then transformed into process P. Actions in this paper are the same as actions in CCS [28], divided into action (a) and co-action ($\bar{a}$), obviously $a = \bar{\bar{a}}$. The prefix action a can have parameters.

3) $P_1 + P_2$ stands for the choice of P_1 or P_2, according to the process subordinated by the following actions.

4) $P_1 \|_A P_2$ means that if action (a) in P_1 and co-action ($\bar{a}$) in P_2 are subordinated to set A, then P_1 and P_2 execute synchronously, while any other actions are executed asynchronously. After P_1 and P_2 executing, the actions are replaced by $\tau(a)$.

Definition 1 Guarded Expression. The process expression begins with the prefix action. e.g., $P = a.Q$.

Definition 2 Behavior Trace. Suppose the process P can be defined as a finite state transition of the form: $P \equiv P_0 \xrightarrow{a_1} P_1 \ldots \xrightarrow{a_{n-1}} P_{n-1} \xrightarrow{a_n} P_n$

$< a_1, a_2, \ldots a_n >$ is the behavior trace of process P. The set of all possible behavior traces is denoted by $traces(P)$.

Definition 3 Process Equivalence. If P, Q are two different processes, and ($traces(P) = traces(Q)$), i.e., $traces(P) \subseteq traces(Q)$ and $traces(Q) \subseteq traces(P)$, then these are denoted as process P, Q equivalent.

The process algebra is used for describing behaviors of distributed applications in this paper. We detect behaviors base on process expressions; thus, the process equivalence is based on the behavior trace. If two processes have the same behavior trace, they are considered equivalent. This also meets the requirements for behavior detection. However, it differs from equivalence based on mutual simulation of CCS [28]. It also differs from equivalence based on refusal sets in CSP [29].

4. Core Mechanisms

The architecture of our model consists of a modeling unit and a detection unit, shown in **Fig. 1**.

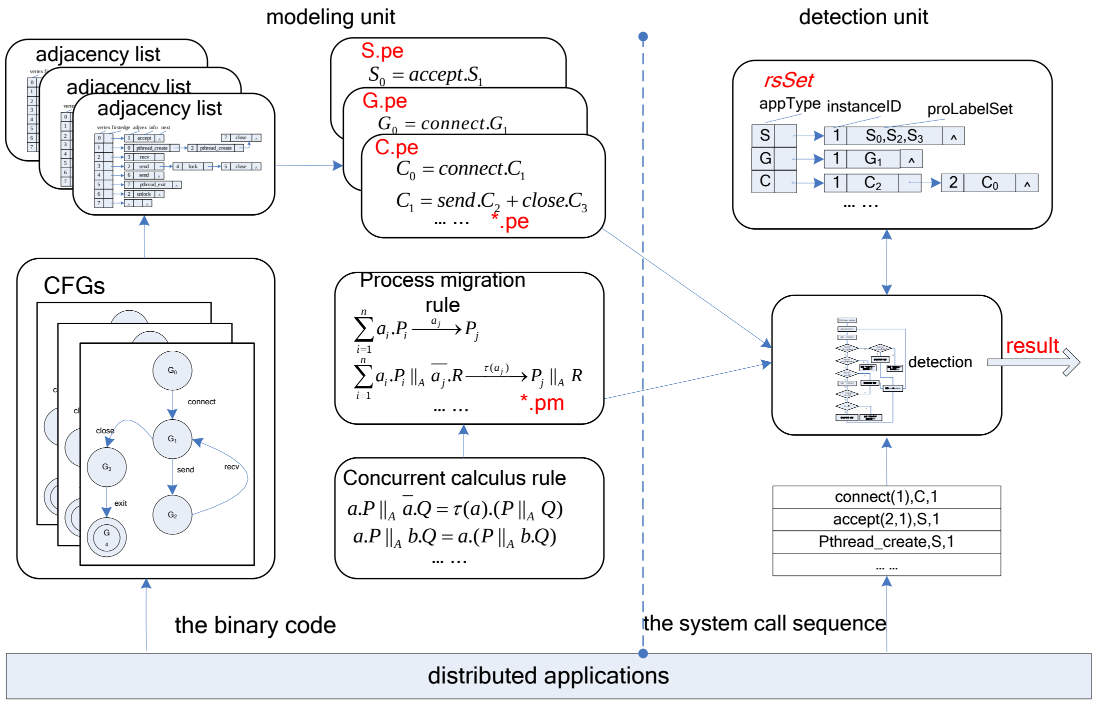

Fig. 1. Architecture of the model

The input to the modeling unit is the binary code of distributed applications, used to generate CFGs of the monitoring nodes. The modeling unit contains the process for building the normal behavior database. Process expressions are obtained from the corresponding CFGs. We rewrite the process expressions to accurately describe the interaction behaviors and concurrent behaviors of distributed applications. The rewritten process expressions are deemed as normal behaviors, saved in files *.pe. The process migration rules are deduced from concurrent calculus rules and are saved in files *.pm, used for detecting anomalous behaviors. The input to the detection unit is the system call sequences monitored at runtime, used as the detection source. The states of distributed applications constitute running state set *rsSet*. We use the files *.pe, files *.pm, and *rsSet* to detect the system call sequences extracted at runtime. Once an anomalous behavior is detected, the model will alert and determine the location and scope of abnormal nodes.

4.1 Generating Process Expressions

Based on IA32 platforms running the Linux OS, we use a bank queue management system as an example. The distributed systems consist of get-ticket client, call-ticket clients, and a server. The server maintains a user queuing list and listens to clients to create a thread for each connection. When the get-ticket client sends a request, the server adds the request information to the end of the queuing list. When the call-ticket clients send a request, the server takes out the head of the queuing list and deals with the new business. The processing procedure must be locked because there can be multiple call-ticket clients. We compile the example to binary code and use the EEL [15-16] method to generate the CFGs for each function of monitoring nodes. We eliminate the edges ε of CFG using the previously reported reduction algorithm [30], merging each function CFGs of monitoring nodes into a global CFG. We capture part of the work flow in **Fig. 2**. The two actions connected by the red dotted line are complementary actions.

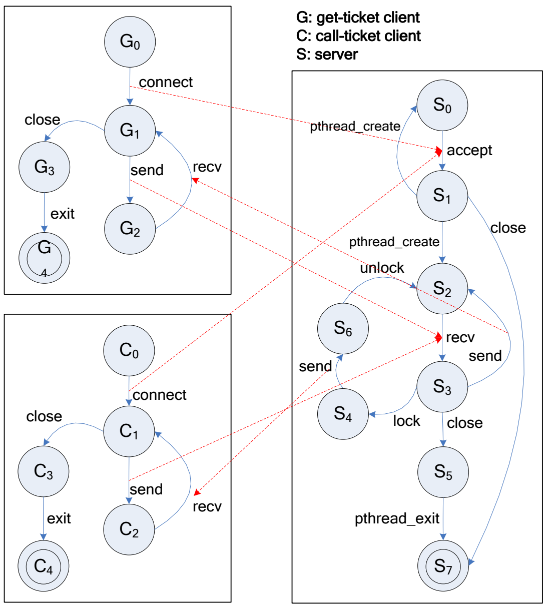

Fig. 2. A CFG for part of the work flow

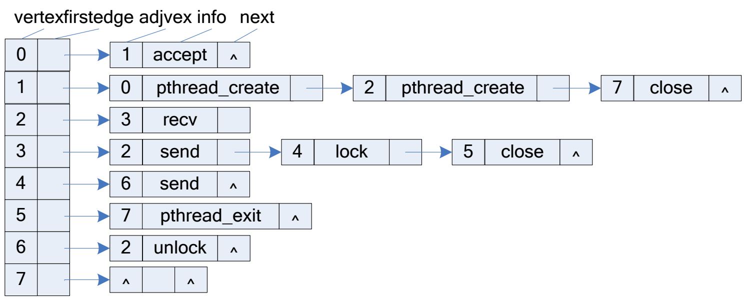

Fig. 3. The adjacency list of the CFG in Fig. 2

We denote the CFG as $G = \{V, E\}$, where V denotes vertices and E denotes the directed edge that is marked with system calls. We store the CFG with an adjacency list, which keeps the system calls on the edge. We obtain the corresponding adjacency list for the CFG of the server, as shown in **Fig. 3**.

The algorithm that transforms the CFGs into the corresponding process expression is given below.

1) All the vertices $v \in V$ are denoted by process label.

2) If $v_i, v_j \in V$ and v_i reaches v_j exist, there exists an output edge $e \in E$, which generates the process expression $v_i = e.v_j$.

3) If on the vertices $v_i \in V$ exist many output edges n(n ⩾ 2), then each output edge defined in (2) generates a process expression, connected using the alternative composition operator: $v_i = e_{j1}.v_{j1} + ... + e_{jn}.v_{jn}$.

4) If the vertices $v_i \in V$ do not contain an output edge, that information is denoted by the successful terminated process mark $\surd$.

We obtain the process expressions of the server based on the above algorithm.

S_0=accept.S_1
S_1= pthread_create.S_0+pthread_create.S_2+ close.S_7
S_2=recv.S_3
S_3=send.S_2+lock.S_4 +close.S_5
S_4=send.S_6
S_5=pthread_exit.S_7
S_6=unlock.S_2
S_7= $\surd$

4.2 Rewriting Process Expressions

4.2.1 Eliminating Nondeterminism

The concept "nondeterminism" describes a finite automaton that exists in several states at the same time. Similarly, if the process expression contains an item that has the form $a.P + a.Q$, it is nondeterminist. This kind of process expression can cause backtracking in the detection phase, reducing detection efficiency. Fortunately, we use an ingenious way to eliminate the nondeterminism.

According to definition 3, we know the left distributive law of alternative composition operators is established, i.e., $a.P + a.Q = a.(P + Q)$. Thus we use $P + Q$ as a new process to eliminate nondeterminism (examples below).

S_1 = pthread_create.S_0 + pthread_create.S_2 + close.S_7
 = pthread_create.(S_0+ S_2) + close.S_7
 = pthread_create.S_8 + close.S_7
S_8 = S_0 + S_2 = accept.S_1 + recv.S_3

We eliminate nondeterminism by adding a new expression, S_8. The effectiveness is similar to the automata theory; however, our method is more intuitive and easy to implement.

4.2.2 Adding Concurrency Operators

The function CFGs cannot express concurrency, so we must rewrite the process expressions to describe the concurrent characteristics of distributed applications.

4.2.2.1 Structure Synchronous Actions Set

According to the concurrent operator $P_1 \|_A P_2$ in Section 3, we must determine synchronous actions set A and its complementary actions. In order to accurately depict the behaviors of system calls, we need to capture and analyze arguments. For instance, we can determine whether the *send* and *recv* belong to the same socket channel. Then we can determine whether they are complementary actions. From 324 system calls (Linux-2.6.18 kernel), we extract the system calls that cause synchronous operation and analyze their arguments. Some of them are listed in **Table 1**, where the pairs *accept* and *connect*, *send* and *recv* are complementary actions.

Table 1. Arguments capture and analysis for system calls

System call	Modeling phase		Detection phase	
	Arguments	**Description**	**Arguments**	**Description**
accept	srcType desType	program type in local host program type in destination host (e.g., server, client)	newfd sockaddr	return value (the file descriptor, used for accessing destination address) local address pointer
connect	srcType desType	program type in local host program type in destination host (e.g., server, client)	sockaddr	destination address pointer
send	srcType desType	program type in local host program type in destination host (e.g., server, client)	sockfd	the file descriptor (for accessing destination address)
recv	srcType desType	program type in local host program type in destination host (e.g., server, client)	sockfd	the file descriptor (for accessing destination address)

We capture the arguments by analyzing the assembly code; for example.

```
0x080489c2:   mov        −0x18(%ebp),%eax
0x080489c5:   mov        %eax,(%esp)
0x080489c8:   call       0x8048608 <send@plt>
… …
0x08048a01:   mov        −0x18(%ebp),%eax
0x08048a04:   mov        %eax,(%esp)
0x08048a07:   call       0x8048548 <recv@plt>
… …
```

From the above assembly code, we know the parameters *sockfd* of *send* and *recv* are from *[EBP-18h]*. So *send* and *recv* pass messages to others by the same socket channel, and we mark *srcType* and *desType* as *c* and *s* respectively: *send(c,s)*, *recv(c,s)*. Because the distributed nodes are being continually changed by connecting or disconnecting, *c* and *s* are used to distinguish the different applications, such as a server application or a client application. Moreover, we use them during the modeling phase instead of specific parameter values such as *[EBP-18h]* for two reasons: (1) We only need to know the relation between parameters during the modeling phase rather than specific values. (2) The specific values can only be determined at runtime. Obviously, we cannot expect programmers to provide specific values because they are too much. However, the relation of parameters is not determined only by analyzing the assembly code. The best way to see the relation is to study the program's behaviors in correct execution. In other words, we can statistically "learn" the relation between parameters through training.

4.2.2.2 Adding Concurrency Operation

Concurrency operators are added to process expressions in the following situations:

1) Situation 1. The operations can create a new multi-process or multi-thread. When the multi-processes or multi-threads such as *fork* and *vfork* appear in the application, we should analyze jump sentences such as *JLE* and *JNE* and change their alternative composition operators to parallel composition operators (‘+’→’$\|_A$’).

For instance, in *pthread_create* of process expression S_1 in section 4.2.1, the alternative composition operators in S_8 are replaced with parallel composition operators: $S_8=S_0\|_A S_2=\text{accept}.S_1\|_A \text{recv}.S_3$.

2) Situation 2. The operations are used for the synchronization of multi-threads.

(a) The mutual exclusion operations appear in system calls, such as *lock*, *pthread_mutex_lock*, *unlock*, and *pthread_mutex_unlock*.

(b) The condition variables operations appear in system calls, such as *lock*, *pthread_mutex_lock*, *wait*, *signal*, *unlock*, and *pthread_mutex_unlock*.

(c) The read or write lock operations appear in system calls, such as *rlock* and *wlock*.

The method of rewriting a process expression is to create a process expression for a), b), c)—for instance, $S_{lock} = \overline{lock}.\overline{unlock}.S_{lock}$ —and let it run concurrently with an access process, such as $P \|_A S_{lock}$, as detailed in our previous publication [8].

(d) Parent and child process, the main thread or child thread synchronization operation:

In order to recycle the resources of a zombie process, after calling the child process *exit*, the parent process must call *wait*, *wait3*, *wait4*, *waitpid*, *pthread_join*, etc. Thus, we must add a synchronization mark: We insert an action sig_i after *wait* and insert a co-action $\overline{sig}_i$ after system call *exit* (*i* is a positive integer, the identification number). Meanwhile, we add the new inserted action into the synchronous action set *A*.

3) Situation 3. The operations are used for interaction in distributed monitoring nodes. For example, in the queue management system in section 4.1, after obtaining the process expressions of each monitoring node, we combine the process expressions into the whole process expression by parallel composition operators, i.e., $P_{QMS} = G_0 \|_A C_0 \|_A S_0$.

4.3 Process Migration Rules

According to section 3, the sequential composition operator and alternative composition operator can migrate as $a.P \xrightarrow{a} P$, $a.P + b.Q \xrightarrow{a} P$, $a.P + b.Q \xrightarrow{b} Q$. This gives us migration rule 1:

Migration rule 1: $\sum_{i=1}^{n} a_i.P_i \xrightarrow{a_j} P_j$,1≤j≤n, n≥1.

According to migration rule 1, the following migration can lead to system halt, causing an abnormality.

Path abnormality 1: If $a_i \neq b$, then $\sum_{i=1}^{n} a_i.P_i \xrightarrow{b} \underline{0}$,1≤j≤n， n≥1.

Based on definition 3, the concurrent relation laws are given as follows. We divide the concurrent relations into binary concurrent relations and multiplex concurrent relations.

1) Binary concurrent relations: The relation of two guarded expressions is connected with a parallel composition operator, such as $a.P \|_A b.Q$. According to the relation of the prefix action a, b, and synchronous set *A*, we present six laws for parallel compositions.

Law 1 (zero element) $\underline{0}$ is a zero element; i.e., $P \|_A \underline{0} = \underline{0}$.

Law 2 (identity) $\surd$ is an identity; i.e., $P \|_A \surd = P$.

According to Laws 1 and 2, we can obtain $\surd \|_A \underline{0} = \underline{0}$ and $\surd \|_A \surd = \surd$.

Law 3 If $a, \bar{a} \in A$, then $a.P \|_A \bar{a}.Q = \tau(a).(P \|_A Q)$.

Law 4 (**synchronous abnormality**) If $a, b \in A$ and $b \neq \bar{a}$, then $a.P \|_A b.Q = \underline{0}$.

According to laws 3 and 4, we know the actions in synchronous set *A* cannot be executed independently. Thus, they must be executed synchronously with corresponding co-actions.

Law 5 If $a \notin A$ and $b \in A$, then $a.P \|_A b.Q = a.(P \|_A b.Q)$.

Law 6 If $a, b \notin A$, then $a.P \|_A b.Q = a.(P \|_A b.Q) + b.(a.P \|_A Q)$.

Laws 5 and 6 indicate that actions outside the set *A* execute asynchronously, and that parallel composition operators ($\|_A$) can be ultimately transformed into alternative composition operators (+).

2) Multiplex concurrent relations: The relations of guarded expressions are connected with multiple composition operators, such as (.), (+), and ($\|_A$).

Law 7 If $a, \bar{a} \in A$, then $(a.P + b.Q) \|_A \bar{a}.R = \tau(a).(P \|_A R)$.

Law 7 indicates that synchronous operations will be performed preferentially when the synchronization condition is met. It is then easy to reach the following conclusion.

If $a, \bar{a}, b, \bar{b} \in A$, then $\begin{aligned}&(a.P + b.Q) \|_A (\bar{a}.R + \bar{b}.S) = \\ &\qquad \tau(a).(P \|_A R) + \tau(b).(Q \|_A S)\end{aligned}$.

If $a, b, c, d \notin A$, then $\begin{aligned}&(a.P + b.Q) \|_A (c.R + d.S) = \\ &a.(P \|_A (c.R + d.S)) + b.(Q \|_A (c.R + d.S)) \\ &+ c.((a.P + b.Q) \|_A R) + d.((a.P + b.Q) \|_A S)\end{aligned}$.

Law 8 If $a, \bar{a} \in A$, then $\begin{aligned}&a.P \|_A a.Q \|_A \bar{a}.R = \\ &\qquad \tau(a).(P \|_A a.Q \|_A R) + \tau(a).(a.P \|_A Q \|_A R)\end{aligned}$.

Law 8 indicates that if the process $a.P$ and $a.Q$ are competitive for $\bar{a}.R$, they must be concurrent with $\bar{a}.R$ separately.

The other forms of multiplex concurrent relations can be summed up by the above two relations. Law 7 and law 8 constitute the minimum complete law set of multiplex concurrent relations.

Next we give the other migration rules. According to law 3, law 7, and law 8, we know if $a, \bar{a} \in A$, then

$a.P \|_A \bar{a}.Q \xrightarrow{\tau(a)} P \|_A Q$

$(a.P + b.Q) \|_A \bar{a}.R \xrightarrow{\tau(a)} P \|_A R$

$a.P \|_A a.Q \|_A \bar{a}.R \xrightarrow{\tau(a)} a.P \|_A Q \|_A R$.

Thus we can obtain migration rule 2 by combining this with migration rule 1.

Migration rule 2: If $a, \bar{a} \in A$, then $\sum_{i=1}^{n} a_i.P_i \|_A \overline{a_j}.R \xrightarrow{\tau(a_j)} P_j \|_A R$, 1≤j≤n，n≥1.

According to law 5, we know that if $a \notin A$ and $b \in A$, then $a.P \|_A b.Q \xrightarrow{a} P \|_A b.Q$.

According to law 6, we know that if $a, b \notin A$, then $a.P \|_A b.Q \xrightarrow{a} P \|_A b.Q$ and $a.P \|_A b.Q \xrightarrow{b} a.P \|_A Q$.

According to migration rules 1 and 2, the following migration can lead to system halt, causing an abnormality.

Path abnormality 2: If $a_i \neq c$ and $b_i \neq c$, then $\sum_{i=1}^{n} a_i.P_i \,\|_A \sum_{k=1}^{m} b_k.R_k \xrightarrow{c} \underline{0}$.

4.4 Running State Set

We denote the part of the process expressions left of the equal sign as a *process label* to describe the process state. To record the running states of the application, we construct the running state set *rsSet*, consisting of distributed applications state *appState*, thusly: *rsSet={appState}*, *appState={appType,riLink}*. Here, *appType* is the application type identifier, and *riLink* is the pointers that point to running instance *ri*. *ri={instanceID,proLabelSet,next}*. Here, *instanceID* is the running instance identifier, and each running instance has a unique identifier that corresponds with the IP address and port number of the deployment node. *proLabelSet* is the process state set of the current running instance, consisting of *process label*. An application can have multiple running instance *ri*, and *next* represents the pointers pointing to the next *ri* that has the same *appType*. The construction algorithm is as follows.

(a) Initialize the *rsSet* so it is empty.

(b) If the process expression contains ($\|_A$), then we add the *process label* on both sides of the ($\|_A$) into *rsSet*.

The *rsSet* will be updated continually when detecting a system call *syscall* during the detection phase.The updating algorithm is as follows.

Input: the awaiting detection system call *syscall*; running state set *rsSet*.

Output: running state set *rsSet*.

Algorithm description:

```
Pe    Cp,Mp,P,Q；  /* declaration process expression */
appType atype;
instanceID iid;
processLabel pl;
atype=GetappType(syscall);   /*access the application type identifier of syscall */
iid=GetinstanceID(syscall);   /*access the running instance identifier of syscall */
if(pl=FindProcessID(atype,iid,rsSet))  /*if the rsSet have not the current running instance，then alert */
        alert();      /* abnormality, quit, alert */
Cp=GetPe(pl); /* access the process expression according to pl */
   if(Cp contain ||A){
      P=leftprocessLable(Cp);
      Q=rightprocessLable(Cp);
      rsSet.insert(P);
      rsSet.insert(Q);
   }
   else{
      Mp=migration(Cp);   /* process migration function */
      rsSet.delete(Cp);
      rsSet.insert(Mp)；
   }
   Return rsSet;
```

The migration method in the algorithm is given in Section 5.2, shown in **Fig. 4**.

5. Anomalous Behavior Detection and Analysis

In this section, we report the results of behavior detection. The performance of the proposed method is evaluated experimentally.

5.1 Extraction of System Call Sequences

There are two situations for extracting system call sequences.

1) When detecting the behaviors of a monitoring node, such as the get-ticket client, the system call sequences at runtime are marked as awaiting detection actions.

2) When detecting the interaction behaviors of multiple monitoring nodes, if the monitoring nodes have a global clock, then the system call sequences of monitoring nodes will combine into one sequence according to the global clock. If these monitoring nodes do not have a global clock, then the events of distributed applications that happen later may be marked with an earlier time tag. We use the time stamp ordering method proposed by Lamport [2] to combine the events into one sequence by logical time sequence.

5.2 Detecting Anomalous Behavior

5.2.1 Behavior Detection Algorithm

The primary steps of the detection algorithm are shown in **Fig. 4**.

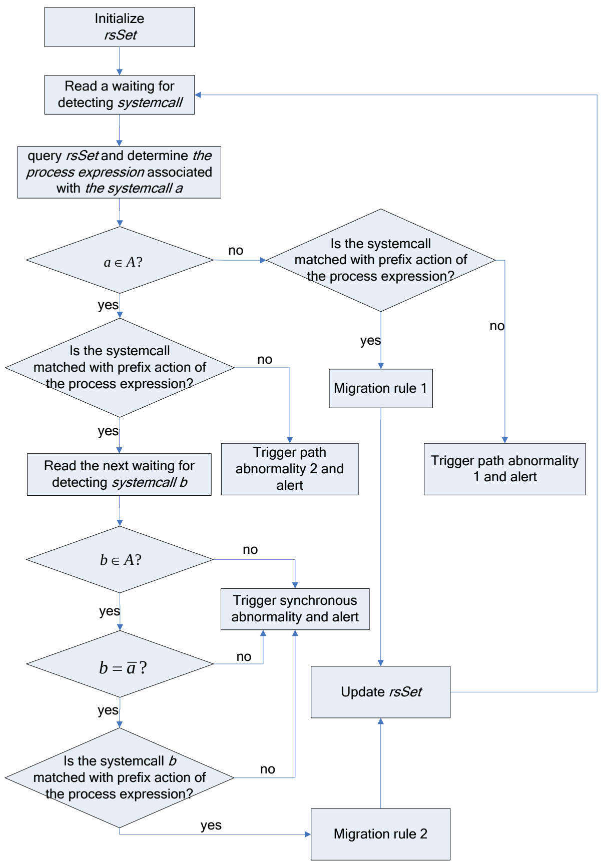
Initialize *rsSet*
Read a waiting for detecting *systemcall*
query *rsSet* and determine *the process expression* associated with *the systemcall a*
$a \in A$?
no
Is the systemcall matched with prefix action of the process expression?
yes
no
Is the systemcall matched with prefix action of the process expression?
no
yes
Migration rule 1
yes
Read the next waiting for detecting *systemcall b*
Trigger path abnormality 2 and alert
Trigger path abnormality 1 and alert
$b \in A$?
no
yes
Trigger synchronous abnormality and alert
$b = \bar{a}$?
no
Update *rsSet*
yes
Is the systemcall *b* matched with prefix action of the process expression?
no
yes
Migration rule 2

Fig. 4. The behavior detection flow chart

Its main idea is described as follows.

Input: the action sequences extracted according to the methods in Section 5.1;

Output: some alarm information, such as detection log, abnormal system call, and the current *rsSet*.

1) Initialize *rsSet* and read the action sequences in turn for detecting.

2) Query the running state set *rsSet* on the basis of system call information and obtain the running state of distributed applications. Then query the normal behavior database (files *.pe), finding the process expressions associated with the system call.

3) Match the system call with the prefix action of process expressions. If match is successful, then proceed to 4). Otherwise, alert and save the information.

4) Migrate the running state according to the migration rules (files *.pm) and update the migrated state into the running state set *rsSet*.

5) Read out the next system call and go to 2).

5.2.2 Behavior Detection Analysis

We analyze the queue management system of a bank in **Fig. 2** and capture the system call sequences for get-ticket client, call-ticket clients, and server at runtime, as shown in **Fig. 5**. We capture system calls and their parameters according to **Table 1**. Then, we analyze the IP address and port information of the local and destination hosts and make them correspond with the running instance identifier, replacing the original parameters of system calls with the running instance identifier, as shown in **Fig. 5**. We bind a system call to its *appType* and *instanceID*, for example, *{accept(2,1),S,1}*. That means the *appType* of the system call *accept* is *S*. Similarly, the *instanceID* of the system call *accept* is *1*. The *instanceID* of the running instance identifier that interacts with the system call *accept* is *2*.

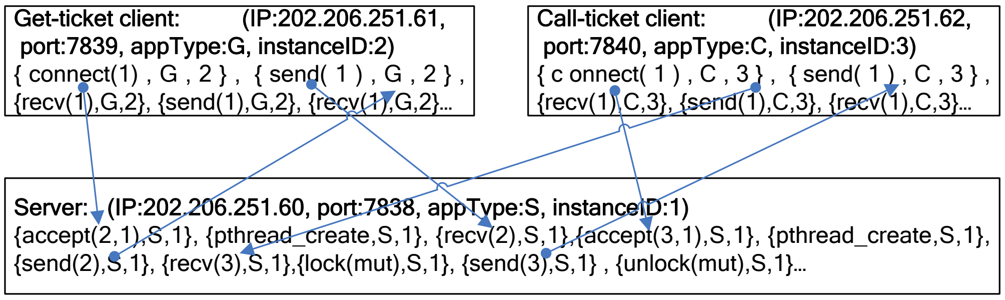
Get-ticket client: (IP:202.206.251.61, port:7839, appType:G, instanceID:2)
{connect(1),G,2}, {send(1),G,2}, {recv(1),G,2}, {send(1),G,2}, {recv(1),G,2}...
Call-ticket client: (IP:202.206.251.62, port:7840, appType:C, instanceID:3)
{connect(1),C,3}, {send(1),C,3}, {recv(1),C,3}, {send(1),C,3}, {recv(1),C,3}...
Server: (IP:202.206.251.60, port:7838, appType:S, instanceID:1)
{accept(2,1),S,1}, {pthread_create,S,1}, {recv(2),S,1},{accept(3,1),S,1}, {pthread_create,S,1},
{send(2),S,1}, {recv(3),S,1},{lock(mut),S,1}, {send(3),S,1} , {unlock(mut),S,1}...

Fig. 5. System call sequences at runtime

We use the method in Section 5.1 to obtain the system call sequences as follows.
{connect(1),G,2},{accept(2,1),S,1},{pthread_create,S,1},{send(1),G,2},{recv(2),S,1}, {connect(1), C,3}, {accept(3,1),S,1},{pthread_create,S,1},{send(2),S,1},{recv(1),G,2},{send(1),C,3},{recv(3),S, 1},{lock(mut),S,1}, {send(3),S,1}, {recv(1),C,3}, {unlock(mut),S,1} ...

The above system call sequences are read in turn, matching with the process expressions ($P_{QMS} = G_0 \|_A C_0 \|_A S_0$) of the queue management system. Meanwhile, process migrates according to migration rules. The behavior detection procedures are described in **Fig. 6**. We use the method in Section 4.2.2.1 to obtain the parameters' relation of process expressions in **Fig. 6**. The numbered arrows in the figure stand for detection steps. The detection steps are given for the first 10 times. For example, for the first detection read out *{connect(1),G,2}*, the type identifier of its application is *G* (get-ticket client), and the current running instance identifier is *2*. The initial state is *rsSet={{G，{2，{G_0}} },{C,{3,{ C_0}} }, {S,{1,{ S_0}} } }*, and thus the process expression G_0 is associated with the system call *connect(1)*. According to

Fig. 4, we know the awaiting detection action *connect(1)* belongs to the synchronous actions set A. Therefore, the next system call *accept(2,1)* needs be read out, so the process migrates according to migration rule 2; i.e., G_0 migrates to G_1 and S_0 migrates to S_1, updating *rsSet*. Then go to step 2. The red number in the box is the running instance identifier.

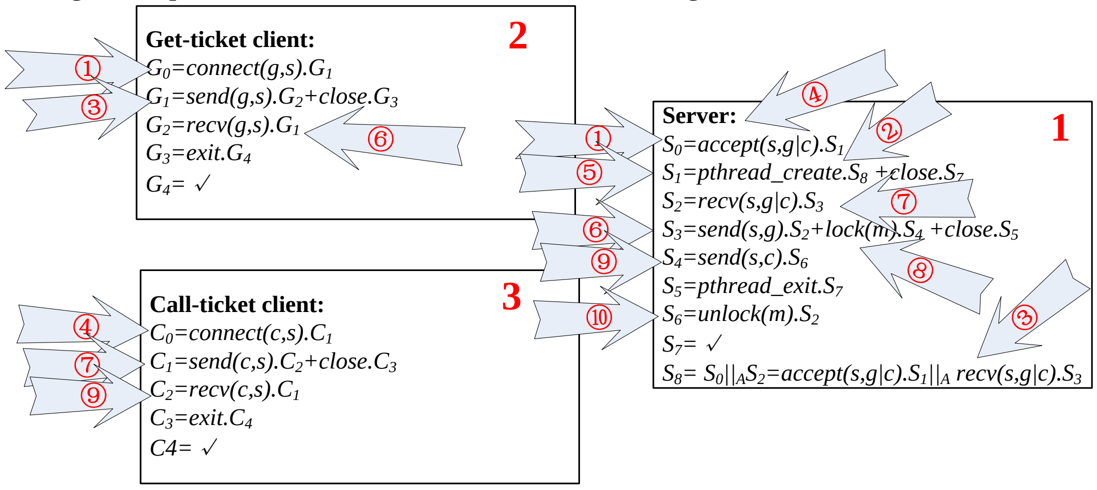

Fig. 6. Behavior detection procedures

Initial state: $rsSet=\{\{G, \{2, \{G_0\}\}\},\{C,\{3,\{C_0\}\}\}, \{S,\{1,\{S_0\}\}\}\}$;

*Step*①: *systemcall:* $\{connect(1),G,2\},\{accept(2,1),S,1\}$;
$rsSet=\{\{G, \{2, \{G_1\}\}\},\{C,\{3,\{C_0\}\}\}, \{S,\{1,\{S_1\}\}\}\}$;

*Step*②: *systemcall:* $\{pthread\_create,S,1\}$;
$rsSet=\{\{G, \{2, \{G_1\}\}\},\{C,\{3,\{C_0\}\}\}, \{S,\{1,\{S_0, S_2\}\}\}\}$;

*Step*③: *systemcall:* $\{send(1),G,2\},\{recv(2),S,1\}$;
$rsSet=\{\{G, \{2, \{G_2\}\}\},\{C,\{3,\{C_0\}\}\}, \{S,\{1,\{S_0, S_3\}\}\}\}$;

*Step*④: *systemcall:* $\{connect(1), C,3\}, \{accept(3,1),S,1\}$;
$rsSet=\{\{G, \{2, \{G_2\}\}\},\{C,\{3,\{C_1\}\}\}, \{S,\{1,\{S_1, S_3\}\}\}\}$;

*Step*⑤: *systemcall:* $\{pthread\_create,S,1\}$;
$rsSet=\{\{G, \{2, \{G_2\}\}\},\{C,\{3,\{C_1\}\}\}, \{S,\{1,\{S_0, S_2, S_3\}\}\}\}$;

*Step*⑥: *systemcall:* $\{send(2),S,1\},\{recv(1),G,2\}$;
$rsSet=\{\{G, \{2, \{G_1\}\}\},\{C,\{3,\{C_1\}\}\}, \{S,\{1,\{S_0, S_2\}\}\}\}$;

*Step*⑦: *systemcall:*,$\{send(1),C,3\},\{recv(3),S, 1\}$;
$rsSet=\{\{G, \{2, \{G_1\}\}\},\{C,\{3,\{C_2\}\}\}, \{S,\{1,\{S_0, S_3\}\}\}\}$;

*Step*⑧: *systemcall:* $\{lock(mut),S,1\}$;
$rsSet=\{\{G, \{2, \{G_1\}\}\},\{C,\{3,\{C_2\}\}\}, \{S,\{1,\{S_0, S_4\}\}\}\}$;

*Step*⑨: *systemcall:* $\{send(3),S,1\}, \{recv(1),C,3\}$;
$rsSet=\{\{G, \{2, \{G_1\}\}\},\{C,\{3,\{C_1\}\}\}, \{S,\{1,\{S_0, S_6\}\}\}\}$;

*Step*⑩: *systemcall:* $\{unlock(mut),S,1\}$;
$rsSet=\{\{G, \{2, \{G_1\}\}\},\{C,\{3,\{C_1\}\}\}, \{S,\{1,\{S_0, S_2\}\}\}\}$;

The matching procedure of system calls is described as follows.

1) Query *rsSet* and determine the process expression associated with the *systemcall*. Does the *systemcall* name match the prefix action name of the process expression? If the match is successful, then turn to (2).

2) Match the parameters of system calls with the parameters of process expressions. For instance, for *{connect(1),G,2}*, its *appType* is *G*, the *instanceID* of running instance identifier that interacts with it is *1*, and its *appType* is *S*. Therefore, it successfully matches with *connect(g,s)*. Note: While application types, such as G and S, are not case sensitive, we define the parameters of system calls with capital letters and define the action parameters of process expressions with lower-case letters.

If the system call sequences all match correctly, then the behavior is normal. If the matching procedure causes one of the abnormalities in Section 4.4, then an alert is generated. Moreover, according to the abnormal system call and the current *rsSet*, we can determine the associated process expression of the abnormal system call and the interactive processes, and then we can determine the location and scope of abnormal anomalous behavior nodes.

If a new node is connected during the detection phase, then we add its state into the running state set *rsSet*. For instance, when a call-ticket node number 4 connects with the server, *rsSet={{G，{2，{*G_1*}} },{C,{(3,{* C_2*}),(4,{*C_1*})} }, {S,{1,{* $S_{0,}$ S_4 *}} } }*. If a node is disconnected, then its state will be deleted from the running state set *rsSet*. In this way, the dynamic change of nodes will be dealt with well.

5.2.3 Detecting Intrusion

The sample code for the main function in the server is given in Section 5.2.2, whose function is used to invoke call-ticket or get-ticket. If the first character is a carriage return, then the call-ticket handling function is summoned. If the first character is A, then the get-ticket handling function is invoked. However, if the data packets from the get-ticket client were intercepted by anomalous behavior—such as if their character A is sullied by a carriage return character—the server remains rigid in determining their behavior according to the character. Although such an intrusion can happen, our model is effective in detecting it. Under these circumstances, our model can capture the system call sequences as follows.

*…{send(1),G,2},{recv(2),S,1},{connect(1),C,3},{accept(3,1),S,1},{pthread_create,S,1},**{lock(mut),S,1},{send(2),S,1}**,{recv(1),G,2},{unlock(mut),S,1} …*

When *{lock(mut),S,1}* is detected during the detection phase, it is easy to know the running state of the server is S_3 from **Fig. 6**. It can successfully match with the prefix action *lock(m)*. Thus, the process migrates to S_4 and then matches with the next system call *{send(2),S,1}* and prefix action *send(s,c)*. Here the running instance identifier that interacts with system call *send* is ***2***, and the corresponding *appType* is *G* rather than *C*. From **Fig. 6** we can see how noticeable intrusions are, and how easy anomalies are to detect.

5.3 Anomalous Behaviors Detection Analysis

A brief discussion on the relationship of several common attack types and exception types is as follows.

1) Code injection attacks.

Code injection attacks are the attackers from local or remote to insert dangerous executable shell code into an address space of the process, and then by some means to modify control flow of the process, make the process execute this shell code, eventually achieving the purpose of attacking behavior. Such as the buffer overflow attacks, the format string misuse and double-free. These kinds of attacks will trigger the path abnormality 1 and the path abnormality 2.

2) Impossible path attacks.

Impossible paths exist when multiple different call sites to the same target procedure exist. In order to achieve this kind of attack, the attackers need to construct the call stack to save function return address, and they may implement by injecting code. Once the impossible paths occurred, they will trigger path abnormality 1.Moreover, they may be trigger the other abnormities supposing that the attackers achieve their purposes indirectly by injecting code.

3) Mimicry attacks.

Mimicry attacks are that the attackers modify the parameters of system call using the rightful system call sequences to achieve the attack purpose. Our model only analyses the interactive action parameters of system call sequences, thus it can not detect the mimicry attacks. We will research on a wider range of data flow analysis to overcome this shortcoming in the future.

4) Denial of Service attacks / Distributed Denial of Service attacks.

According to the action mechanism, denial of service attacks / distributed denial of service attacks can be divided into resource depletion denial of service attacks and denial of service attacks based on the abnormality. The former refer to the attackers exhaust system resources by a large number of inputs, for example, setting up a large number of network connections, forcing process processing large files, etc. These kinds of attacks typically applied to distributed denial of service attacks, and they can not trigger the abnormality, thus our model can not detect these kinds of attacks. However, the latter depend on the specific defect of the process, for instance, infinite loops owing to the integer overflow. These kinds of attacks need to change the control flow of process to achieve the attack purpose, and they can trigger the path abnormality 1 and the path abnormality 2, thus they can be detected by our model.

5) The man-in-the-middle attacks.

The man-in-the-middle attack (MITM) requires an attacker to have the ability to both monitor and alter or inject messages into a communication channel. The attacker can intercept all relevant messages passing between the two victims and inject new ones. These kinds of attacks can cause the actions of the relevant nodes do not synchronize, and trigger the synchronous abnormality (Law 4).

6) Man-at-the-end attacks.

Man-at-the-end (MATE) attacks occur in settings where an adversary has physical access to a device and compromises it by tampering with its hardware or software. Remote man-at-the-end (R-MATE) attacks occur in distributed systems where un-trusted clients are in frequent communication with trusted servers over a network, and malicious users can get an advantage by compromising an un-trusted device. These kinds of attacks also need to change the control flow of process to achieve the attack purpose, and they can trigger the path abnormality 1 and the path abnormality 2, thus they can be detected by our model.

5.4 Comparative Study of Previous Approaches

The existing security paradigms proposed for distributed applications are aimed at some specific attacks, such as DOS/DDOS attacks and Remote Man-at-the-end (R-MATE) attacks. A comparative study on previously schemes is presented in **Table 2.**

Table 2. Comparative study of previous approaches

Ref	Detection Technique	Detection Time	Architecture	Coverage	Pros	Cons
This Work	Anomaly Based	Online	Host Network Distributed	Computers, Networks, Cloud computing, Distributed application.	Distributed operations, Multi-threaded processing, Real time detection.	Could not detect the mimicry attack and attacks based on data-flow.

[4]	Anomaly Based	Online	Host Network Distributed	Cloud computing	Multi-threaded cloud IDS.	Works only for Cloud computing
[5]	Anomaly Based	Online	Distributed	Distributed application	Distributed operations, Against Remote Man-at-the-end (R-MATE) attacks.	Could not detect : The programs in which the client does not need to frequently communicate with the server; The applications which need to completely prevent any tampering of client code.
[6]	Misuse & Anomaly Based	Online & Offline	Host Network Distributed	Computers, Networks, Cloud computing, Distributed application.	Hybrid detection, Distributed and non-distributed operations, Multi-threaded processing, Real time detection.	High computation cost

We also compare the precision of the above schemes and the result is presented in **Table 3.**

Table 3. Comparison of detection precision for schemes

Attacks	**Our scheme**	**Ref[4] scheme**	**Ref[5] scheme**	**Ref[6] scheme**
Code injection attacks	Yes	No	No	Yes
Impossible path attacks	Yes	No	No	No
Mimicry attacks	No	No	No	No
DOS/DDOS	Yes/No	Yes	No	Yes
The man-in-the-middle attacks	Yes	No	No	Yes
Remote Man-at-the-end (R-MATE) attacks	Yes	No	Yes	No

As can be seen from **Table 3**, the precision of our scheme is better than the other schemes. But our scheme is better suited for an application to distributed applications that pay close attention to perform orders and logical, such as mission critical system.

5.5 Performance Evaluation

5.5.1 Theory Analysis

We adopt two algorithm methods to search the adjacency list: depth-first and breadth-first. No matter which algorithm is used, the time complexity of the adjacency list is the same: $O(n+e)$, where n denotes the number of vertices and e denotes the number of adjacency edges. Obviously, the space complexity of the adjacency list is also $O(n+e)$. If we store a set of m concurrent processes, the space and time complexity is $O(m)$. Thus, the total space and time complexity is $O(n+e+m)$; that is to say, the proposed method has linear space–time complexity. Moreover, the running costs do not increase significantly with the increasing of application scale.

5.5.2 Experimental Analysis

We use Hadoop for the test objects and deploy a Hadoop computer cluster. Next we investigate a benchmark test case for *WordCount*, *Sort* and *Pi*, using 3, 6, 9, and 12 *Slave* nodes, as well as the *Master* node. We record the cost of time and space that the process of modeling and detection incurs. Our runtime environment is as follows: one server running Linux OS (rhel-server-5.4) on an Intel 2 GHZ six-core CPU with 8GB of RAM, as the *Master* node, and twelve hosts running Linux OS (rhel-server-5.4) on Intel 1.73 GHZ dual core CPUs with 2 GB of RAM, as the *Slave* nodes. We use the properties of three benchmark test cases listed in **Table 4**.

Table 4. The properties of benchmark test case

Program	Functional description	Workload
WordCount	A map/reduce program that counts the words in the input files	Counting 10,000,000 words
Sort	A map/reduce program that sorts data written by the random writer	Randomly generating and sorting 100MB per node
Pi	A map/reduce program that estimates pi using the Monte Carlo method	Calculating pi

We obtain the time and space overhead in the modeling phase, as shown in **Fig. 7**. Similarly, the time and space overhead in the detection phase are shown in **Fig. 8** and **Fig. 9** respectively. Program execution times are to record in seconds. The base execution time has no modeling and detection operation. Percentages compare against base execution.

As can be seen from experimental results, the time and space overheads of our model increase linearly in the detection phase, which basically was anticipated. The time and space overhead of the model are high in the modeling phase because of performing a large number of calculations. However, the time and space overhead of our detection model are satisfactory in the detection phase.

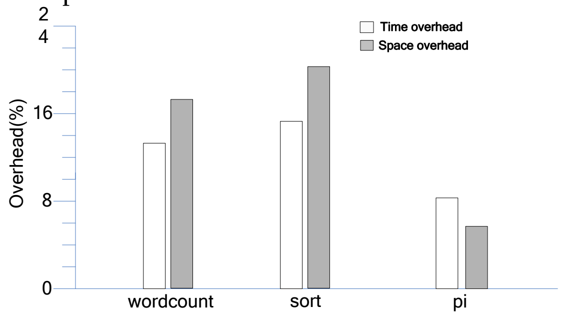

Fig. 7. The time and space overhead in the modeling phase

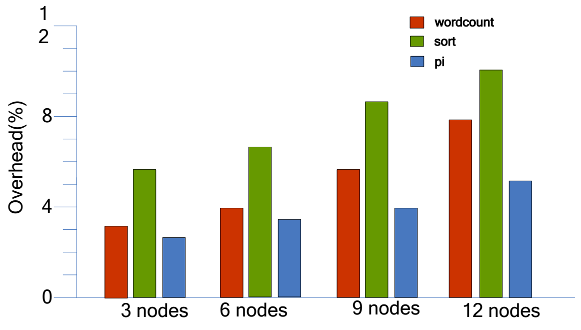

Fig. 8. The time overhead in the detection phase

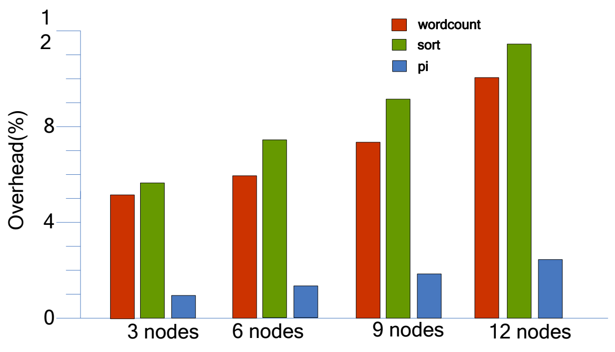

Fig. 9. The space overhead in the detection phase

6. Conclusions

In this paper, the problem of providing safe, efficient parallel implementations of distributed applications is investigated. We presented a method for anomalous behavior detection using system call sequences. Normal behaviors were defined in terms of system call sequences executed by running privileged processes. We use process algebra to describe behaviors of distributed applications and detect behaviors based on process expressions. Our profiles of normal behaviors, which were generated using process expressions rewritten by eliminating nondeterminism and adding concurrency operators, were precise and complete. The architecture of our model was specified, using a bank queue management system as an example to describe the procedure of the detection algorithm. Behavior detection and analysis results show that our method is a good discriminator between normal and anomalous behavior characteristics of distributed applications. Performance evaluation shows that the proposed method enhances efficiency without security degradation.

References

[1] D. Caromel and L.A. Henrio, "Theory of Distributed Objects," *Berlin:Springer-Verlag*, 2005. Article (CrossRef Link).

[2] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no.7, pp. 558-565, July, 1978. Article (CrossRef Link).

[3] C. C. Din, J. Dovland, E.B. Johnsen, and O. Olaf, "Observable behavior of distributed systems: Component reasoning for concurrent objects," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 3, pp. 227-256, April, 2012. Article (CrossRef Link).

[4] I Gul and M. Hussain, "Distributed cloud intrusion detection model," *International Journal of Advanced*, vol. 34, pp. 71-82, September, 2011. Article (CrossRef Link).

[5] C. Collberg, S. Martin, J. Myers and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proc. of the 28th Annual Computer Security Applications Conference. ACM*, pp. 319-328, December, 2012. Article (CrossRef Link).

[6] F. Idrees, M. Rajarajan and A. Y. Memon, "Framework for distributed and self-healing hybrid intrusion detection and prevention system," in *Proc. of ICT Convergence (ICTC), International Conference on. IEEE*, pp. 277-282, October, 2013. Article (CrossRef Link).

[7] J. Meseguer and P. C. Ö lveczky, "Formalization and correctness of the PALS architectural pattern for distributed real-time systems," *Formal Methods and Software Engineering. Springer Berlin Heidelberg*, vol. 6447, pp. 303-320, 2010. Article (CrossRef Link).

[8] W. Tao, S. Liming and M. Chuan, "A Process Algebra-Based Detection Model for Multithreaded Programs in Communication System," *KSII Transactions on Internet and*

Information Systems, vol. 8, no. 3, pp. 965-983, March, 2014. Article (CrossRef Link).

[9] J.Chu, "The Triple Pot and techniques in distributed system call intrusion detection," *University of Illinois at Urbana-Champaign*, 2014. Article (CrossRef Link).

[10] S. Forrest, S.A. Hofmeyr, A. Somayaji and T.A. Longstaff, "A sense of self for UNIX processes," *in Proc. of the IEEE Symp. on Security and Privacy. Oakland: IEEE Press*, pp. 120-128, May 6-8, 1996. Article (CrossRef Link).

[11] S.A. Hofmeyr, S. Forrest and A. Somayaji. "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151-180, January, 1998. Article (CrossRef Link).

[12] P. Helman and J.Bhangoo, "A statistically based system for prioritizing information exploration under uncertainty," *IEEE Trans.on Systems,Man and Cybernetics, Part A:Systems and Humans*, vol. 27, no. 4, pp. 449-466, July, 1997. Article (CrossRef Link).

[13] W. Lee and S.J. Stolfo, "Data mining approaches for intrusion detection," *in Proc. of the 7th USENIX Security Symp. San Antonio*, pp. 26-29, January, 1998. Article (CrossRef Link).

[14] D. Wagner and D. Dean, "Intrusion detection via static analysis," *in Proc. of the IEEE Symp.on Security and Privacy.Oakland:IEEE Press*, pp. 156-168, May 14-16, 2001. Article (CrossRef Link).

[15] J. Giffin, S. Jha and B. Miller, "Efficient context- sensitive intrusion detection," *in Proc. of the 11th Network and Distributed System Security Symp. San Diego*, 2004. Article (CrossRef Link).

[16] R. Gopalakrishna, E.H. Spafford and J. Vitek, "Efficient intrusion detection using automaton Inlining," *In Proc. of the IEEE Symp.on Security and Privacy. Oakland, CA, IEEE Press*, pp. 18-31, May 8-11, 2005. Article (CrossRef Link).

[17] F. Jianming, T. Fen, and W. Dan, "Software behavior model based on system objects," *Journal of Software*, vol. 22, no. 11, pp. 2716-2728, November, 2011. Article (CrossRef Link).

[18] H.H. Feng, J.T. Giffin, Y. Huang and S. Jha, "Formalizing sensitivity in static analysis for intrusion detection," *In Proc. of the IEEE Symp.on Security and Privacy. Oakland, CA, IEEE Press*, pp. 194-208. May 9-12, 2004. Article (CrossRef Link).

[19] M. Moshirpour, A. Mousavi and B. H. Far, "Detecting emergent behavior in distributed systems using scenario-based specifications," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, no. 06, pp. 729-746, September, 2012. Article (CrossRef Link).

[20] F. Yang, T. Aotain, H. Masuhare, et al., "Combining static analysis and runtime checking in security aspects for distributed tuple spaces," *Coordination Models and Languages. Springer Berlin Heidelberg*, vol. 6721, pp. 202-218, June 6-9, 2011. Article (CrossRef Link).

[21] D. Gupta, K. V. Vishwanath, M. McNett, et al., "DieCast: Testing distributed systems with an accurate scale model," *ACM Transactions on Computer Systems (TOCS)* , vol. 29, no. 2, Article No.4 , May, 2011. Article (CrossRef Link).

[22] J. Tan, S. Kavulya, R. Gandhi, et al., "Light-weight black-box failure detection for distributed systems," in *Proc. of the 2012 workshop on Management of big data systems. ACM*, pp.13-18, 2012. Article (CrossRef Link).

[23] M. Rohr, A. van Hoorn, W. Hasselbring, et al., "Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems," *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering. ACM*, pp. 87-92, 2010. Article (CrossRef Link).

[24] M. Moshirpour, R. Alhajj, M. Moussavi, and B. H. Far, "Detecting emergent behavior in distributed systems using an ontology based methodology," *In Systems, Man, and Cybernetics (SMC), IEEE International Conference on. IEEE*, pp. 2407-2412, October, 2011. Article (CrossRef Link).

[25] J.H. Morris, "Lambda-calculus Models of Programming Languages," *MIT, Cambridge, MAC,* U SA, 1968. Article (CrossRef Link).

[26] G.J. Milne and R. Milner, "Concurrent processes and their syntax," *Journal of the ACM*, vol. 26, no. 2, pp. 302-321, April, 1979. Article (CrossRef Link).

[27] J. C. M. Baeten, “A brief history of process algebra,” *Theoretical Computer Science*, vol.335, no. 2, pp. 131-146, May, 2005. Article (CrossRef Link).

[28] R. Milner, “A calculus of communicating systems,” *Lecture Notes in Computer Science*, Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1980. Article (CrossRef Link).

[29] C. Hoare, “Communicating sequential processes,” *Communications of the ACM* , vol. 21, no. 8, pp. 666-677, August, 1978. Article (CrossRef Link).

[30] J. Hopcroft, “An nlogn algorithm for minimizing states in a finite automaton,” *Theory of Machines and Computations*, New York: Academic Press, January, 1971. Article (CrossRef Link).

Chuan Ma is currently having his Ph.D study in Information Science and Engineering, Yanshan University. He received his B.S. and M.S. degrees in the School of Information Science and Engineering, Yanshan University, Qinhuangdao, China in 2003 and 2009, respectively. He is now working at the School of Information Science and Engineering Yanshan University as a lecturer. His current research interests include information security and software formal methods.

Limin Shen is currently a professor in the School of Information Science and Engineering, Yanshan University. He received his M.S. degree in computer application, Hefei University of Technology, China, in 1987. He received his Ph.D degree in electronic circuit and system, Yanshan University, China, in 2005. He worked in Department of Computer Science, Illinois Institute of Technology, USA from 2005 to 2007 as a visiting scholar. His main research interests are focusing on flexible software technology and information security, which has been funded partially by the National Natural Science Foundation of China and Chinese Government.

Tao Wang is currently having her Ph.D study in Information Science and Engineering, Yanshan University, Qinhuangdao, Hebei, China. She received her M.S. degree in the School of Information Science and Engineering, Yanshan University, Qinhuangdao, China in 2009. She is now working at Hebei Normal University of Science & Technology as a lecturer. Her current research interests are in the areas of intrusion detection and collaboration computing.