Available online at www.sciencedirect.com
SCIENCE DIRECT®
Microprocessors and Microsystems 28 (2004) 531–546
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

Defend mobile agent against malicious hosts in migration itineraries

Y.C. Jiang[a,*], Z.Y. Xia[b], Y.P. Zhong[a], S.Y. Zhang[a]

[a]*Center for Networking and Information Engineering, Department of Computing and Information Technology, Fudan University, Room 409, Yifu Building, No 220 Handan Road, Shanghai 200433, China*
[b]*Department of Computer Science and Engineering, Nanjing University of Aeronautics and Astronautic, Nanjing 210043, China*

Available online 11 September 2004

Abstract

Agent integrity veri cation and *fault-tolerance* are the two prevalent methods among the solutions to the Problem of Malicious Hosts in Mobile agent system. Agent integrity verification enables the owner of the agent to detect upon its return whether a visited host has maliciously altered the state of the agent based on agent integrity verification [6]. A known drawback of such method is that it cannot detect the tampering of agent immediately, and the tampering can be detected only when the agent returned. Agent fault-tolerance is one method that achieves agent fault-tolerance in migration itineraries by agent replication and majority voting [11]. The drawback of such method is that the agent replication and majority voting can produce many agent replicas in every agent migration step, which may cost significant resource and time. Aiming at those drawbacks, the paper incorporates the two methods, and presents a novel agent migration fault-tolerance model based on integrity verification, which can defend mobile agent against malicious hosts in migration itineraries effectively. The novel agent fault-tolerance model cannot only realize the fault-tolerant execution, but also reduce the complexity and resource cost of agent migration communication.

Keywords: Problem of malicious hosts; Agent fault-tolerance; Agent integrity verification; π-calculus; Spi-calculus

1. Introduction

Mobile agents are software programs that may move from host to host as necessary to carry out their functions. Such systems violate some of the assumptions that underlie most existing computer security implementations [1]. Problem of Malicious Host (POMH) (i.e. how to protect agents against the malicious hosts) is a serious security problem in mobile agent system. To solve the POHM, various methods have been developed, such as Time Limited Blackbox [2], Reference States [3], Cryptographic Traces [4], Authentication and State Appraisal [5], etc. Among these, the two most prevalent solutions of POMH are agent integrity verification [6] and agent fault-tolerance [11].

The set of hosts visited by the mobile agent is termed as *itinerary*. At any host, the execution environment of the agent is controlled by the host. Hence, mobile agents are vulnerable to attacks by malicious host in migration itinerary; one of the attacks to the mobile agent is the tampering of the data carried by the agent. To detect such attack, the method of agent integrity verification was proposed. In Ref. [6], Karjoth et al. proposed the notion of AppendonlyContainer for detecting the tampering of an agent's data by individual malicious hosts. However, the mechanism in Ref. [6] does not address the problem that two or more malicious hosts *collude* with each other to delete the data of other hosts. To solve such problem, Vijil and Sridhar Iyer [7] incorporated and extended the notion of the *AppendOnlyContainer* to include not only the detection of tampering but also the identification of the malicious host, so as to detect the colluding malicious hosts in mobile agent itineraries. Otherwise, Jeff et al. [8] proposed several defense against the truncation attack and the related growing-a-fake-stem attack for the protection of the partial computation results of free-roaming agents.

However, all of the above works about agent integrity verification only detect the attacks when the agent returns to its owner. Therefore, the attacks cannot be detected immediately as it takes place, which may influence the execution of agent.

* Corresponding author. Tel.: +86-21-6564-3235; fax: +86-21-6564-7894.
E-mail address: jiangyichuan@yahoo.com.cn (Y.C. Jiang).

0141-9331/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2004.08.008

Moreover, though the above approaches are effective for solving the POMH in some degree, they do not cope with how to keep the mobile agent system uninterrupted during operating when the POMH takes place. Aiming at such problem, the concept of *agent fault-tolerance* was suggested; among the related researches the measure of agent replication and voting was often adopted. Schneider [11] integrated the concept of fault-tolerance and the principle of cryptography to make the mobile agent system have fault-tolerant ability. The solution in Ref. [11] makes agent produce many replicas at every migration step so that the crash of one agent cannot influence the operation of the whole system. Unfortunately, such solution is not feasible in practice, since it assumes that replicated servers fail independently [16] and requests that all agent replicas be kept alive until the end of agent migration, and the large numbers of agent replicas cost a lot of network and host resource. In the meantime, the result voting among the replicas of agent also cost significant resource and time.

On the base of our original work in Ref. [25], we incorporate the above two methods and extend them, and present a novel agent migration fault-tolerance model based on integrity verification (AMFIV) and its improved version (P-AMFIV). This model can detect the tampering of agent immediately as it takes place and carry out the fault-tolerant execution. It also reduces the complexity and resource cost of agent migration communication. Lastly, this paper makes π/spi-calculus analysis and experiment for P-AMFIV, which prove our solution is feasible and efficient.

The rest of the paper is organized as follows. Section 2 introduces the related research work on agent integrity verification and fault-tolerance. Section 3 presents the novel agent migration fault-tolerance model AMFIV and P-AMFIV. Section 4 models P-AMFIV based on π-calculus and spi-calculus. Section 5 describes simulation experiment. Then the paper concludes in Section 6.

2. Overview of agent integrity verification and fault-tolerance

2.1. Agent integrity veri cation

The agents need to be protected such that they can acquire new data on each host they visit, but any tampering with pre-existing data must be detected by the agent's owner (and possibly by other hosts on the agent's itinerary) to keep the agent integrity [9]. The issue of agent integrity has heretofore always been ignored in the realm of agent literature. With the existence of malicious hosts and inaccurate information, along with many unsolved problems arising from agent interaction, the agent's integrity is in jeopardy. Given enough time, a malicious host can examine the agent's code or carried data in order to alter them to suit its own needs or desires. Strategies must be developed to protect the integrity of agents [10].

In Ref. [6] Karjoth et al. propose a notion of *AppendOnlyContainer*. The idea is to protect agent objects in a container such that new objects can be added to it but any subsequent modification of an existing object can be detected by the agent's owner with the use of *checksum*.

When an agent starts its itinerary, the agent's owner initializes the *checksum* based on a random nonce r. The nonce must be kept secret by the agent owner and is not carried by the agent. The initial checksum is as follows:

$$C_0 = E_{\text{owner}}\{r\} \tag{1}$$

When a host i wants to insert data D_i, then the data item D_i, $\text{Sig}_i(D_i)$ and the identity of the host i, are inserted into the appropriate arrays in the *AppendOnlyContainer*. The checksum is then updated as follows:

$$C_i = E_{\text{owner}}\{C_{i-1}||\text{Sig}_i(D_i)\} \tag{2}$$

Upon the agent's return, the agent owner successively decrypts the checksum, extracts the signature, and verifies the signature against the corresponding object in the container. If in the last iteration, the agent owner recovers the original random nonce r, it can be inferred that the *AppendOnlyContainer* has not been tampered with.

However, the mechanism in Ref. [6] does not indicate the identity of the malicious host, and cannot detect the modification of an agent by two colluded malicious hosts. In Ref. [7] Vijil et al. incorporate and extend the notion of the *AppendOnlyContainer* to include not only the detection of tampering but also the identification of the malicious host, and introduce the notion of *Expected Number of Deletions (END)* to detect deletion of data by colluding malicious hosts in static as well as dynamic settings. Otherwise, Cheng and Wei propose some protocols to prevent the two-colluder truncation attack and identify the exact pairs of colluders for prosecution [8].

From the description of above works on agent integrity verification, we can see that they often make verification while the agent returns; if an agent is tampered in the migration itinerary, it is not detected immediately. And if an agent is tampered, we cannot get the correct execution result of agent. Therefore, there is a significant demand for the real-time attacks detection and the fault-tolerance agent migration.

2.2. Agent fault-tolerance

We introduce the concept of *trace* to describe the nodes sequence of agent migration. The ith node in the trace can be called as the ith stage of agent migration.

The linear agent migration model can be viewed as a pipeline [12] shown as Fig. 1. Where nodes represent hosts, and edges represent migration of an agent from one host to another. Each node corresponds to a stage of the pipeline.

Obviously, this model is not fault-tolerant. If a host is malicious, then the agent cannot migrate to the destination

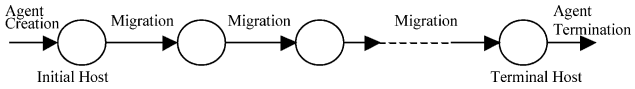

Fig. 1. The linear agent migration model.

successfully and correctly. Therefore, the linear model is not fitted for an unsafe network environment though it is simple.

To achieve the fault-tolerance of agent migration, a generally used approach is *agent replication and voting*. In such method, many agent replicas are produced, and the replicas run independently on different hosts. After the run of replicas, the ultimate result is obtained by collecting votes among the results of replicas.

The method suggested in Ref. [11] is representative of these relative works [11–15]. Fig. 2 is the model of replication agent migration computation with majority voting described in Ref. [11], which can be called by us as RAMMV.

In RAMMV, a node p in stage i takes as its input the majority of the inputs it receives from the nodes comprising stage $i-1$. And, p sends its output to all of the nodes that it determines comprise stage $i+1$ [11]. Fig. 2 shows such a fault-tolerance execution. The voting at each stage makes it possible for the computation to heal by limiting the impact of the faulty host in one stage on hosts in subsequent stages. More precisely, in the architecture of RAMMV, it is possible to tolerate faulty values from a minority of the replicas in each stage.

However, as discussed in Section 1, such model is not feasible in practice, mainly as this model requests that all agent replicas be kept alive until the end of agent migration, and the large numbers of agent replicas may cost significant network and host resource. At the same time, the result voting among the replicas of agent is also resource and time consuming.

To make up the deficiency of the previous works about agent integrity verification and fault-tolerance, and to solve the POMH effectively, we suggest a novel agent AMFIV and P-AMFIV.

3. A novel solution to the POMH in mobile agent migration

3.1. AMFIV model

To solve the POMH effectively, we incorporate the ideas of agent integrity verification and fault-tolerance, and present a novel agent migration fault-tolerance model based on integrity verification called AMFIV.

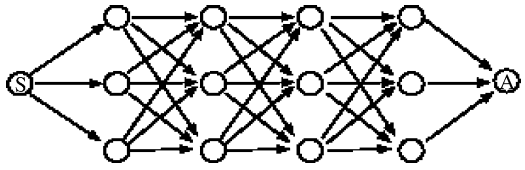

Fig. 2. Replicated agent migration computation with majority voting (RAMMV).

Now we take a trace example to illustrate the principle of AMFIV shown as Fig. 3.

Fig. 3 can be explained as follows:

- After the agent at stage i executes on $host_i$, it selects a node with the highest priority as the next host to migrate, i.e. $host_{i+1}(0)$;
- The agent spawns a replica, and the replica migrates to $host_{i+1}(0)$;
- After the agent replica executes on $host_{i+1}(0)$, $host_i$ makes integrity verification for it; if the integrity verification result is ok, the agent on $host_i$ is terminated, and the agent on $host_{i+1}(0)$ spawns a replica to migrate to $host_{i+2}(0)$; otherwise, it shows that $host_{i+1}(0)$ is a malicious one, then the agent on $host_i$ re-selects another host with the second priority as the next one to migrate, i.e. $host_{i+1}(1)$, and the model will execute the operations as the same as above steps.
- If $host_{i+1}(1)$ is also malicious, then the agent on $host_i$ will re-select another host $host_{i+1}(2)$ with the third priority as the next one to migrate…until there exists a normal host to migrate or there do not exist any other adjacent nodes to select. If $host_i$ has not any other adjacent nodes, then the agent on $host_i$ returns to $host_{i-1}$, and selects another node as $host_i(1)$.

Sometimes, $host_i$ cannot obtain answer from $host_{i+1}$ after sending verification request to it, then $host_i$ will send verification request again. If $host_i$ still cannot obtain any answer from $host_{i+1}$ after some times of request, then $host_i$will also consider $host_{i+1}$ as a malicious one.

From Fig. 3, we can see that agent does not need to produce replica at every migration step. In AMFIV, first agent migrates according to linear trace, only when the agent integrity is tampered by a malicious host then a new path is re-selected. But RAMMV model requires that the replicas be produced at every migration step. On the other hand, AMFIV limits the problem to be solved in single hop, which avoid the multiple steps accumulative problem.

3.2. Simple analysis of AMFIV

3.2.1. Complexity and real-time property of AMFIV

Let the number of agent migration steps is n, and the number of standby nodes in every step is m. Now we analyze the complexity of AMFIV.

Obviously, the complexity of agent migration communication degrees in RAMMV is $O(n\times m)^2$, and the one in AMFIV is $O(n\times m)$. So AMFIV reduces the complexity of agent migration communication degrees from cube level to square level. Therefore, the network load in AMFIV can be reduced accordingly.

On the amount of replicas produced, the average complexity in RAMMV is $O(n\times m)$; but in AMFIV, only

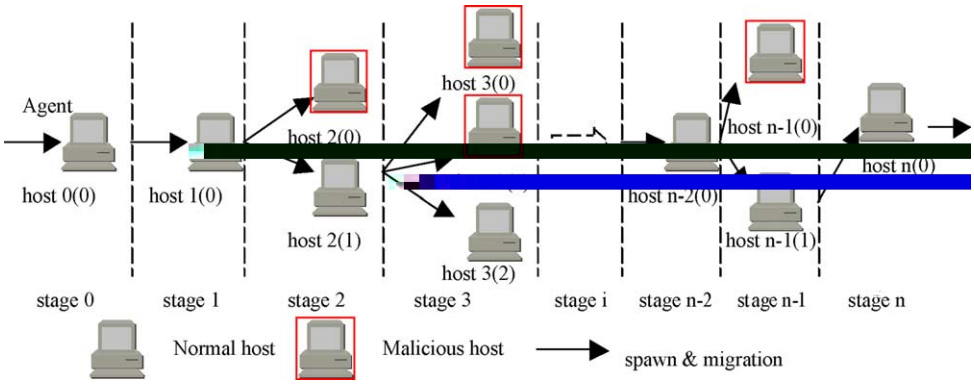

Fig. 3. The agent migration trace example of AMFIV.

under the worst situation, i.e. the first $m-1$ nodes are all malicious in every step, the complexity can reach $O(n \times m)$. Obviously, the worst situation seldom takes place in practice, so AMFIV can also reduce the amount of agent replicas.

In AMFIV, the host in stage i can make verification for the agent in stage $i+1$. Therefore, if the host in stage $i+1$ compromises the agent, the host in stage i can detect it immediately, then the system may reselect a new itinerary to migrate. So, AMFIV has real-time property, which outperforms the other related works on agent integrity verification.

3.2.2. Nonblocking and exactly-once property of AMFIV

Stefan et al. [15] proposed that fault-tolerant mobile agent execution should have two properties: *nonblocking* and *exactly-once*. Nonblocking ensures that agent execution can proceed despite a single failure of the agent or the machine; Exactly-once ensures that successful agent execution is made only once.

Blocking occurs if a single failure prevents the execution from proceeding. In contrast, an execution is nonblocking if it can proceed despite a single failure. In AMFIV, if a host in stage i is malicious, the host in stage $i-1$ can detect it. And the agent on the host of stage $i-1$ can re-select another host to migrate. Therefore, one malicious host cannot prevent the agent execution from proceeding, AMFIV has the nonblocking property.

Replication allows us to prevent blocking, but it can also lead to a violation of the exactly-once execution. However, such situation does not exist in AMFIV. In AMFIV, at first agent migrates according to linear trace; if a malicious host tampers the agent, the execution result on the malicious host is discarded fully, and the agent in the former stage re-selects a new host to execute. Therefore, at anytime only one execution result is achieved. Therefore, AMFIV has the exactly-once property.

3.3. Improved version of AMFIV: P-AMFIV

However, in AMFIV, the following attack may take place: a malicious host at stage $i+1$ pretends to be a normal one, and keeps the integrity of agent until the integrity verification process (by $host_i$) is finished. However, after the integrity verification is finished, the malicious host at stage $i+1$ compromises the agent integrity. Therefore, $host_i$ will regard $host_{i+1}$ as a normal one, so it will terminate agent on itself, and transfer the control power to $host_{i+1}$, and $host_{i+1}$ can make the compromised agent migrate to $host_{i+2}$. In this case, the ultimate result of agent migration is not correct.

To solve such a problem, the AMFIV model can be improved as follows: we can delay the time of agent integrity verification. In AMFIV, the agent integrity verification is executed immediately after the run of the agent on $host_{i+1}$, but now we can delay the agent integrity verification until the agent replica migrates to $host_{i+2}$ from $host_{i+1}$. We call such improved AMFIV as P-AMFIV. After agent runs on $host_{i+1}$, it can select a node as $host_{i+2}$, and spawns a replica to migrate to $host_{i+2}$. But after the agent replica migrates to $host_{i+2}$, it cannot execute immediately, at this time $host_i$ first makes integrity verification for the agent replica on $host_{i+2}$. From Fig. 4 we can see that the verification time point of AMFIV and P-AMFIV. If the integrity verification of the agent replica on $host_{i+2}$ is eligible, then the agent on $host_i$ is terminated and $host_i$ transfers the control power to $host_{i+1}$, and the agent replica on $host_{i+2}$ can run. In the other case, it shows that $host_{i+1}$ is malicious, so the agent on $host_i$ re-selects a new host to migrate, and the agent replica on $host_{i+2}$ is terminated, $host_{i+1}$ is also isolated. In P-AMFIV, we suppose that the situation in which two consecutive malicious hosts co-operate to make attack is not existent, or else P-AMFIV is invalid.[1]

From Fig. 4, we can see that the agent integrity verification of agent on $host_{i+1}$ does not take place between $host_i$ and $host_{i+1}$, but between $host_i$ and $host_{i+2}$. For simplicity, we do not consider the detail about when and how to send verification requirement message by $host_i$. And the detail of communication connection between $host_i$ and $host_{i+1}$ (or between $host_i$ and $host_{i+2}$) is not discussed here either. In our model and simulated environment, we suppose $host_i$ can communicate with $host_i$ directly.

[1] Annotation: In this paper, the 'migration' signifies the migration of replica, i.e. the agent on now host spawns a replica and the replica migrates to next host, but the agent on current host keeps alive.

3.4. The agent integrity veri cation in P-AMFIV

Agent integrity includes the integrity of agent code, data and state. The agent state integrity verification is very difficult since the state of agent is dynamic during execution and the owner of agent cannot make digital signature to it. So we mainly concern about the integrity of agent code and data, only discuss how to make agent code and data integrity verification in P-AMFIV model. In our verification protocol, we suppose the hosts have a shared key.

▶ Agent code integrity verification

In agent migration execution, only the agent owner can change its code. Therefore, the agent code cannot be modified in the migration itineraries.

After the agent executes on $host_{i+1}$, it spawns a replica and the replica migrates to $host_{i+2}$. Before the agent replica executes on $host_{i+2}$, we make code integrity verification to detect that if the agent code is tampered by $host_{i+1}$. The code integrity verification protocol is shown as Fig. 5.

The agent code integrity verification protocol is explained as follows: ($K_{i,i+2}(x)$ denotes that encrypting x with the key shared by $host_i$ and $host_{i+2}$).

(A) $host_i \rightarrow host_{i+2}$: i, R_i, $K_{i,i+2}(t_i)$;
(B) $host_{i+2} \rightarrow host_i$: R_{i+2}, $K_{i,i+2}(R_i, t_{i+2})$;/*R_i denotes the request message sent by $host_i$, and R_{i+2} denotes the request message sent by $host_{i+2}$*/
(C) $host_i \rightarrow host_{i+2}$: $K_{i,i+2}(R_{i+2})$;/*(A), (B), (C) denote the identification authentication between $host_i$ and $host_{i+2}$*/
(D) $host_{i+2} \rightarrow host_i$: $K_{i,i+2}(hash(Code_{i+1}||t_{i+2}))$;/*$host_{i+2}$ sends the hash value of the agent code on $host_{i+2}$ with time stamp to $host_i$*/[2]
(E) $host_i$: Check:
compute $hash(Code_i||t_{i+2})$;
if $hash(Code_i||t_{i+2}) == hash(Code_{i+1}||t_{i+2})$
then Agent code integrity is ok;
else Agent code integrity isn't ok.
/*$host_i$ computes the hash value of the agent code
tt

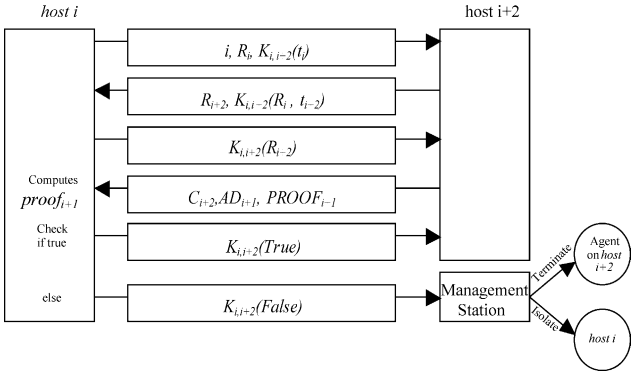

Fig. 6. The protocol for agent data integrity verification.

(C) $host_i \rightarrow host_{i+2}$: $K_{i,i+2}(R_{i+2})$;/*Similar to the protocol of agent code integrity verification, (A), (B), (C) are used for identification authentication between $host_i$ and $host_{i+2}$*/

(D) $host_{i+2} \rightarrow host_i$: C_{i+2}, AD_{i+1}, $PROOF_{i+1}$;/*$host_{i+2}$ passes the agent data information to $host_i$*/

(E) $host_i$: Computes $proof_{i+1} = hash(AD_{i+1} - AD_i$, $hash(C_i)$, $PROOF_i)$;/*Computes $proof_{i+1}$ on $host_i$*/

(F) $host_i$: if ($proof_{i+1} == PROOF_{i+1}$) and ($C_{i+2} == hash(hash(C_i))$)

then agent data integrity is ok;

else agent data integrity is not ok.

Analysis for the protocol: since $C_{i+1} = hash(C_i)$, $host_{i+1}$ cannot obtain C_i from C_{i+1}, $host_{i+1}$ does not know C_j ($j < i+1$), and cannot modify D_j ($j < i+1$), so it cannot forge *PROOF*. Therefore the protocol is secure. Obviously, if the original data of agent is tampered by $host_{i+1}$, then $proof_{i+1}$ is not equal to $PROOF_{i+1}$, so the tampering of data integrity can be detected, therefore the protocol is correct.

Obviously, we can see that the protocol can only detect any tampering of the data collected before $host_{i+1}$, and cannot detect whether $host_{i+1}$ collects dirty data. Therefore, the protocol only guarantees the integrity of validly collected data.

4. Modeling P-AMFIV based on π-calculus and spi-calculus

4.1. Introduction to π-calculus and spi-calculus

The π-calculus is a mathematical model of processes whose interconnections change as they interact. The basic computational step is the transfer of a communication link between two processes; the recipient can then use the link for further interaction with other parties. This makes the calculus suitable for modeling systems where the accessible resources vary over time. It also provides a significant expressive power since the notion of access and resource underlie much of the theory of concurrent computation, in the same way as the more abstract and mathematically tractable concept of a function underlies functional computation [18].

Table 1
Basic syntax of π-calculus

Syntax We take an infinite set $\mathbb{N}$ of *names* of channels, ranged over by *a*, *b*, etc. The *process terms* are then those defined by the grammar:	
$P,Q ::= O$	Nil
$P\|Q$	parallel composition of P and Q
$\bar{c}(v)$	output v on channel c
$c(v)$	input v from channel c
$c(w).p$	restriction: P is a process that makes a new, private name n and then behaves as P.
$(vn).P$	restriction: P is a process that makes a new, private name n and then behaves as P.
$[M=N].P$	match: behaves as described by P if M and N are the same, otherwise is stuck
$let\ (x,y) = M\ in\ P$	pair splitting: behaves as $P[N/x][L/y]$ if term MM is the pair (N,L). Otherwise the process is stuck
case M of O: $P\ suc(x):Q$	behaves as P if term M is O, as $Q[N/x]$ if M is $suc(N)$, and otherwise is stuck.

The basic concept behind the π-calculus is naming or reference. Names are the primary entities, and they may refer to links. Table 1 is the basic syntax of π-calculus [19–21].

The simplest form of semantics for this calculus consists of a reduction relation—a binary relation between process terms, written as $P \rightarrow Q$, indicating that P can perform a single step of computation to become Q.

The spi-calculus is an extension of the π-calculus with cryptographic primitives. It is designed for describing and analyzing security protocols, such as those for authentication and verification. These protocols rely on cryptography and on communication channels with properties like authenticity and privacy. Accordingly, cryptographic operations and communication through channels are the main ingredients of the spi-calculus [22].

Next, we will model P-AMFIV based on π-calculus and the agent data integrity verification based on spi-calculus. Then we use the π/spi calculus model reduction to simulate the execution of P-AMFIV and the integrity verification. If the π/spi calculus model reduction results are correct, it shows that P-AMFIV and the integrity verification are correct.

4.2. Modeling P-AMFIV based on π-calculus

Now we will model P-AMFIV based on π-calculus, the channels used are seen in Table 2.

The whole model of P-AMFIV is comprised of $host_i$, $host_{i+1}$ and $host_{i+2}$. We first model them respectively based on π-calculus.

We can define the π-calculus model of $host_i$ as follows:

$$\begin{aligned} host_i &\stackrel{def}{=} next_1(cha_1).\overline{cha}_1(RUN(agent_i)) \\ &|chr(x).(let\ \ (code_{i+1}, data_{i+1}) = x\ \text{in} \\ &(\overline{chv_1}(VALIDATE(code_{i+1}, data_{i+1})). \\ &\overline{chv_2}(VALIDATE(code_{i+1}, data_{i+1})). \end{aligned} \tag{3}$$

$\overline{chv_3}(VALIDATE(code_{i+1}, data_{i+1}))))$

$|chv_1(y).([y = true].TERMINATE(agent_i)$

$+ [y = false].(\overline{next}_1(SELECT(i+1)).host_i))$

Eq. (3) is explained as follows:

- $Host_i$ obtains the identity of $host_{i+1}$ (denoted as channel cha_1) from channel $next_1$; then spawns a replica of $agent_i$ after executing on $host_i$, and the replica migrates to $host_{i+1}$ through channel cha_1.
- $Host_i$obtains the executing result of $agent_{i+1}$ on $host_{i+1}$ from channel chr, and makes verification ($VALIDATE$) for its code and data integrity; then passes the verification result to channels chv_1, chv_2, chv_3.
- $Host_i$ obtains the verification result from channel chv_1, if it is true the agent on $host_i$ is terminated, or else $host_i$ should re-select a new node to migrate, and passes the new node to channel $next_1$, then repeats all the acts of the model.

We can define the π-calculus model of $host_{i+1}$ as follows:

$$\begin{aligned} host_{i+1} \stackrel{def}{=} & cha_1(agent_{i+1}).next_2(cha_2). \\ & \overline{cha_2}(RUN(agent_{i+1}))|\overline{next_2}(SELECT(i+2)) \\ & |chv_2(z).([z = true].GETMASTER \\ & + [z = false].ISOLATED) \end{aligned} \tag{4}$$

Eq. (4) is explained as follows:

- $Host_{i+1}$ receives the agent replica from channel cha_1 and runs it, $host_{i+1}$ selects the next node, then spawns the agent after running and the replica migrates to the next node.
- $Host_{i+1}$ receives the verification result from channel chv_2, if the verification result is true then $host_{i+1}$ gets the control power, or else $host_{i+1}$ is a malicious one and it should be isolated.

We can define the π-calculus model of $host_{i+2}$ as follows:

$$\begin{aligned} host_{i+2} \stackrel{def}{=} & cha_2(agent_{i+2}).\overline{chr}(agent_{i+2}) \\ & |chv_3(u).([u = true].RUN(agent_{i+2}) \\ & + [u = false].TERMINATE(agent_{i+2})) \end{aligned} \tag{5}$$

Eq. (5) is explained as follows:

- From channel cha_2, $host_{i+2}$ receives the agent replica after execution on $host_{i+1}$, and passes the code and data of the agent replica to $host_i$ through channel chr.
- $Host_{i+2}$ receives the verification result from channel chv_3, if the result is true then $host_{i+2}$ executes the agent replica, or else the agent replica is terminated.

Therefore, we can define the π-calculus model of P-AMFIV as follows:

$$P - AMFIV\_model \stackrel{def}{=} (vnext_1, next_2, cha_1, cha_2, chr, chv_1, chv_2, chv_3)(host_i|host_{i+1}|host_{i+2})$$

$$\equiv (vnext_1, next_2, cha_1, cha_2, chr, chv_1, chv_2, chv_3)(next_1(cha_1).\overline{cha}_1(RUN(agent_i))$$

$$|chr(x).(let(code_{i+1}, data_{i+1}) = x\ in$$

$$(\overline{chv_1}(VALIDATE(code_{i+1}, data_{i+1}))|chv_2(next).\overline{chv_2}(code_{i+1}, data_{i+1})|$$

$$\overline{chv_3}(VALIDATE(code_{i+1}, data_{i+1}))))$$

$$|chv_1(y).([y = true].TERMINATE(agent_i) + [y = false].(\overline{next}_1(SELECT(i+1)).host_i))$$

$$|cha_1(agent_{i+1}).next_2(cha_2).\overline{cha_2}(RUN(VALIDATE$$

/*$host_{i+1}$ selects the next node to migrate*/

$\xrightarrow{next_2(SELECT(i+2))}(vnext_1, cha_2, chr, chv_1, chv_2, chv_3)$

$|chr(x).(let(code_{i+1}, data_{i+1}) = x \text{ in}$

$(\overline{chv_1}(VALIDATE(code_{i+1}, data_{i+1})).\overline{chv_2}(VALIDATE(code_{i+1}, data_{i+1}))$

$\overline{chv_3}(VALIDATE(code_{i+1}, data_{i+1}))))$

$|chv_1(y).([y = true].TERMINATE(agent_i) + [y = false].(\overline{next}_1(SELECT(i+1)).host_i))$

$|\overline{SELECT(i+2)}(RUN(RUN(agent_i)))$

$|chv_2(z).([z = true].GETMASTER + [z = false].ISOLATED)$

$|SELECT(i+2)(agent_{i+2}).\overline{chr}(agent_{i+2})$

$|chv_3(u).([u = true].RUN(agent_{i+2}) + [u = false].TERMINATE(agent_{i+2})))$

/*$host_{i+1}$ spawns the agent after executing and the replica migrates to $host_{i+2}$*/

$\xrightarrow{\tau}(vnext_1, chr, chv_1, chv_2, chv_3)$

$(chr(x).(let(code_{i+1}, data_{i+1}) = x \text{ in}$

$(\overline{chv_1}(VALIDATE(code_{i+1}, data_{i+1})).\overline{chv_2}(VALIDATE(code_{i+1}, data_{i+1}))$

$\overline{chv_3}(VALIDATE(code_{i+1}, data_{i+1}))))$

$|chv_1(y).([y = true].TERMINATE(agent_i) + [y = false]$

$.(\overline{next}_1(SELECT(i+1)).host_i))$

$|chv_2(z).([z = true].GETMASTER + [z = false].ISOLATED)$

$|\overline{chr}(RUN(RUN(agent_i)))$

$|chv_3(u).([u = true].RUN(RUN(RUN(agent_i)))$

$+ [u = false].TERMINATE(RUN(RUN(agent_i)))))$

/*$host_{i+2}$ passes the received agent code and data back to $host_i$*/

$\xrightarrow{\tau}(vnext_1, chv_1, chv_2, chv_3)$

$(let(code_{i+1}, data_{i+1}) = RUN(RUN(agent_i)) \text{ in}$

$(\overline{chv_1}(VALIDATE(code_{i+1}, data_{i+1})).\overline{chv_2}(VALIDATE(code_{i+1}, data_{i+1}))$

$\overline{chv_3}(VALIDATE(code_{i+1}, data_{i+1})))$

$|chv_1(y).([y = true].TERMINATE(agent_i) + [y = false]$

$.(\overline{next}_1(SELECT(i+1)).host_i))$

$|chv_2(z).([z = true].GETMASTER + [z = false].ISOLATED)$

$|chv_3(u).([u = true].RUN(RUN(RUN(agent_i)))$

$+ [u = false].TERMINATE(RUN(RUN(agent_i)))))$

/*$Host_i$ makes integrity verification for the agent executed on $host_{i+1}$ (returned by $host_{i+2}$) and passes the verification result to the corresponding channels*/

$$\xrightarrow{\tau}(\nu next_1)(([VALIDATE(code_{i+1}, data_{i+1}) = true].TERMINATE(agent_i)$$

$$+ [VALIDATE(code_{i+1}, data_{i+1}) = false].(\overline{next}_1(SELECT(i+1)).host_i))$$

$$|([VALIDATE(code_{i+1}, data_{i+1}) = true].GETMASTER$$

$$+ [VALIDATE(code_{i+1}, data_{i+1}) = false].ISOLATED)$$

$$|([VALIDATE(code_{i+1}, data_{i+1}) = true].RUN(RUN(RUN(agent_i)))$$

$$+ [VALIDATE(code_{i+1}, data_{i+1}) = false] .TERMINATE(RUN(RUN(agent_i))))$$

$$\begin{cases} \xrightarrow{VALIDATE(code_{i+1}, data_{i+1})=true} TERMINATE\ (agent_i) \mid GETMASTER \\ \qquad\qquad\qquad\qquad \mid RUN(RUN(RUN(agent_i))) \\ \xrightarrow{VALIDATE(code_{i+1}, data_{i+1})=true} (\nu next_1)next_1(\mathrm{SELECT}(i{+}1)).\mathrm{host}_i \setminus ISOLATED \\ [redacted] \end{cases}$$

*/*Annotation: $RUN(RUN(agent_i))$ denotes that agent runs on $host_{i+1}$, which is equal to $RUN(agent_{i+1})$; $RUN(RUN(RUN(agent_i)))$ denotes that agent runs on $host_{i+2}$, which is equal to $RUN(agent_{i+2})$.*

Obviously, when agent migrates to $host_{i+2}$ and before it runs on $host_{i+2}$, we can denote the agent as $RUN(RUN(agent_i))$./*

From above π-calculus model reduction of P-AMFIV, we can see: if the integrity of agent code and data is ok, the ultimate result is $TERMINATE(agent_i)$ $|GETMASTER$ $|RUN(RUN(RUN(agent_i)))$, so the agent on $host_i$ is terminated, $host_{i+1}$ gets the control power, agent runs on $host_{i+2}$, and agent migrates according to a linear trace; if the integrity of agent is tampered by $host_{i+1}$, the result is $(vnext_i)\overline{next}_1(SELECT(i+1).host_i|ISOLATED$ $|TERMINATE$ $(RUN(agent_i)))$, so $host_i$ re-selects another node as the next one to migrate, and repeats the acts of the model, and $host_{i+1}$ is isolated, the agent on $host_{i+2}$ is terminated.

Therefore, from the above reduction of the π-calculus model of P-AMFIV, the reduction result is correct, so we can see that P-AMFIV is correct accordingly.

4.3. Modeling the agent data integrity veri cation based on spi-calculus

Spi-calculus is the extension of π-calculus, which is used for describing the cryptography protocol [22]. As the agent code integrity verification sub-module is relatively simple, so here we only make analysis to the agent data integrity verification sub-module. We can model the agent data integrity verification based on spi-calculus, and make reduction for the model to simulate the execution of integrity verification.

In our spi-calculus model, the channels used are shown in Table 3.

In our agent data integrity verification sub-module, we can define the spi-calculus model of $host_i$ as follows:

$$host_i \stackrel{def}{=} \overline{pass_1}([hash(C_i)]_k, AD_i, PROOF_i)|chan_{in}(x).$$

$$(let(w, AD_{i+1}, PROOF_{i+1}) = x \qquad (7)$$

$$in\ case\ w\ of\ \{[C_{i+2}]_k\}\ in$$

$$(\overline{chan_c}(C_{i+2}).\overline{chan_{out}}(hash(hash(C_i)), hash(AD_{i+1} - AD_i, hash(C_i), PROOF_i))))$$

$$|chan_c(C_{i+2}).chan_{out}(y).(let(hash^2(C_i), proof_{i+1}) = y$$

$$in\ ([(hash^2(C_i) = C_{i+2})\ and\ (proof_{i+1} = PROOF_{i+1})].\overline{result}(true)$$

$$+[(hash^2(C_i) \neq C_{i+2})\ or\ (proof_{i+1} \neq PROOF_{i+1})].\overline{result}(false))$$

Table 3
List of channel name in the spi-calculus

Channel	Sender	Receiver	Message
$Pass_1$	$host_i$	$host_{i+1}$	$[hash(C_i)]k$, AD_i, $PROOF_i$
$Pass_2$	$host_{i+1}$	$host_{i+2}$	$[hash(hash(C_i))]k$, AD_{i+1}, $hash(D_{i+1}, hash(C_i), PROOF_i)$
$chan_{in}$	$host_{i+2}$	$host_i$	C_{i+2}, AD_{i+1}, $PROOF_{i+1}$
$chan_{out}$	$host_i$	$host_i$	$proof_{i+1}$ computed on $host_i$
$chan_c$	$host_i$	$host_i$	C_{i+2} that extracted from the message received from $host_{i+2}$.
Result	$host_i$	The system (includes $host_i$, $host_{i+1}$, $host_{i+1}$)	The verification rust of agent data integrity

$[hash(C_i)]_k$ denotes that encrypts $hash(C_i)$ with the key k.

Eq. (7) is explained as follows:

- $Host_i$ passes $[hash(C_i)]_k$, AD_i and $PROOF_i$ to $host_{i+1}$ through channel $pass_1$.
- From channel $chan_{in}$, $host_i$ receives C_{i+2}, AD_{i+1} and $PROOF_{i+1}$ that passed by $host_{i+2}$; $host_i$ computes $hash(hash(C_i))$, $proof_{i+1} = hash(AD_{i+1} - AD_i, hash(C_i), PROOF_i)$, and passes the computing result to channels $chan_c$ and $chan_{out}$;
- From $chan_c$ and $chan_{out}$, $host_i$ receives $hash^2(C_i)$ and $proof_{i+1}$, and then compares $hash^2(C_i)$ and $proof_{i+1}$ with C_{i+2}, $PROOF_{i+1}$. If they are the same, then passes 'true' to channel *result*, or else passes 'false' to channel *result*.

We can define the spi-calculus model of $host_{i+1}$ as follows:

$$host_{i+1} \stackrel{def}{=} pass_1(z).(let(u, AD_i, PROOF_i) = z$$

$$in\ case\ u\ of\ \{[hash(C_i)]_k\}\quad in$$

$$\overline{pass_2}([hash(hash(C_i))]_k, AD_{i+1}, hash(D_{i+1}, hash(C_i), PROOF_i))) \tag{8}$$

Eq. (8) is explained as follows:

- From channel $pass_1$, $host_{i+1}$ receives $[hash(C_i)]_k$, AD_i and $PROOF_i$ that passed from $host_i$, and executes agent, then computes $PROOF_{i+1} = hash(AD_{i+1} - AD_i, C_{i+1}, PROOF_i)$, and passes $PROOF_{i+1}$ with AD_{i+1} to channel $pass_2$.

We can define the spi-calculus model of $host_{i+2}$ as follows:

$$host_{i+2} \stackrel{\text{def}}{=} pass_2(v).\overline{chan_{in}}(v) \tag{9}$$

Eq. (9) is explained as follows: From channel $pass_2$, $host_{i+2}$ receives $PROOF_{i+1}$ and AD_{i+1}, and passes them back to $host_i$ through channel $chan_{in}$.

Therefore, we can define the spi-calculus model of the agent data integrity verification in P-AMFIV as follows:

$$VALIDATE\_model \stackrel{def}{=} (vpass_1, pass_2, chan_{in}, chan_{out}, result)(host_i | host_{i+1} | host_{i+2})$$

$$\equiv (vpass_1, pass_2, chan_{in}, chan_{out}, result)(\overline{pass_1}([hash(C_i)]_k, AD_i, PROOF_i) | chan_{in}(x). \tag{10}$$

$$(let\ (w, AD_{i+1}, PROOF_{i+1}) = x$$

$$in\ case\ w\ of \{[C_{i+2}]_k\}\quad in$$

$$(\overline{chan}_c(C_{i+2}).\overline{chan_{out}}(hash(hash(C_i)), hash(AD_{i+1} - AD_i, hash(C_i), PROOF_i))))$$

$$| chan_c(C_{i+2}).chan_{out}(y).(let(hash^2(C_i), proof_{i+1}) = y$$

$$in([(hash^2(C_i) = C_{i+2})\ and\ (proof_{i+1} = PROOF_{i+1})].\overline{result}(true)$$

$$+ [(hash^2(C_i) \neq C_{i+2})\ or\ (proof_{i+1} \neq PROOF_{i+1})].\overline{result}(false))$$

$$| pass_1(z).(let(u, AD_i, PROOF_i) = z$$

$$in\ case\ u\ of \{[hash(C_i)]_k\}\quad in$$

$$\overline{pass_2}([hash(hash(C_i))]_k, AD_{i+1}, hash(D_{i+1}, hash(C_i), PROOF_i)))$$

$$| pass_2(v).\overline{chan_{in}}(v))$$

Now we can use the spi-calculus operational semantics and reduction rules to describe the execution of the agent data integrity verification in P-AMFIV.

/*__The initial spi-calculus model of agent data integrity verification__*/

$VALIDATE\_model$

$\equiv(\nu pass_1, pass_2, chan_{in}, chan_{out}, result)(\overline{pass_1}([hash(C_i)]_k, AD_i, PROOF_i)|chan_{in}(x).$

$(let(w, AD_{i+1}, PROOF_{i+1}) = x$

$in\ case\ w\ of\ \{[C_{i+2}]_k\}\ in$

$(\overline{chan_c}(C_{i+2}).\overline{chan_{out}}(hash(hash(C_i)), hash(AD_{i+1} - AD_i, hash(C_i), PROOF_i))))$

$|chan_c(C_{i+2}).chan_{out}(y).(let(hash^2(C_i), proof_{i+1}) = y$

$in([(hash^2(C_i) = C_{i+2})\ and\ (proof_{i+1} = PROOF_{i+1})].\overline{result}(true)$

$+ [(hash^2(C_i) \neq C_{i+2})\ or(proof_{i+1} \neq PROOF_{i+1})].\overline{result}(false))$

$|pass_1(z).(let(u, AD_i, PROOF_i) = z \quad in$

$case\ u\ of\ \{[hash(C_i)]_k\}in$

$\overline{pass_2}([hash(hash(C_i))]_k, AD_{i+1}, hash(D_{i+1}, hash(C_i), PROOF_i)))$

$|pass_2(v).\overline{chan_{in}}(v))$

/*__$host_i$ encrypts C_{i+1}, and passes it with AD_i and $PROOF_i$ to $host_{i+1}$__*/

$\xrightarrow{\tau}(\nu pass_2, chan_{in}, chan_{out}, result)(chan_{in}(x).$

$(let(C_{i+2}, AD_{i+1}, PROOF_{i+1}) = x$

$in\ case\ w\ of\ \{[C_{i+2}]_k\}in$

$(\overline{chan_c}(C_{i+2}).\overline{chan_{out}}(hash(hash(C_i)), hash(AD_{i+1} - AD_i, hash(C_i), PROOF_i))))$

$|chan_c(C_{i+2}).chan_{out}(y).(let(hash^2(C_i), proof_{i+1}) = y$

$in([(hash^2(C_i) = C_{i+2})\ and\ (proof_{i+1} = PROOF_{i+1})].\overline{result}(true)$

$+[(hash^2(C_i) \neq C_{i+2})\ or\ (proof_{i+1} \neq PROOF_{i+1})].\overline{result}(false))$

$|\overline{pass_2}([hash(hash(C_i))]_k, AD_{i+1}, hash(D_{i+1}, hash(C_i), PROOF_i)))$

$|pass_2(v).\overline{chan_{in}}(v))$

/*__$host_{i+1}$ encrypts C_{i+2}, and passes it with AD_{i+1} and $PROOF_{i+1}$ to $host_{i+2}$__*/

$\xrightarrow{\tau}(\nu chan_{in}, chan_{out}, result)(chan_{in}(x).$

$(let(C_{i+2}, AD_{i+1}, PROOF_{i+1}) = x$

$in\ case\ w\ of\{[C_{i+2}]_k\}in$

$(\overline{chan_c}(C_{i+2}).\overline{chan_{out}}(hash(hash(C_i)), hash(AD_{i+1} - AD_i, hash(C_i), PROOF_i))))$

$|chan_c(C_{i+2}).chan_{out}(y).(let(hash^2(C_i), proof_{i+1}) = y$

$$in\ ([(hash^2(C_i) = C_{i+2})\ and\ (proof_{i+1} = PROOF_{i+1})].\overline{result}(true)$$

$$+[(hash^2(C_i) \neq C_{i+2})\ or\ (proof_{i+1} \neq PROOF_{i+1})].\overline{result}(false))$$

$$|\overline{chan_{in}}([hash(hash(C_i))]_k, AD_{i+1}, hash(D_{i+1}, hash(C_i), PROOF_i))))$$

/*host$_{i+2}$ passes C_{i+2}, AD_{i+1}, $PROOF_{i+1}$ back to host$_i$*/

$$\xrightarrow{\tau}(\nu chan_{out}, result)$$

$$((let(C_{i+2}, AD_{i+1}, PROOF_{i+1}) = ([hash(hash(C_i))]_k, AD_{i+1}, hash(D_{i+1}, hash(C_i), PROOF_i)))$$

$$in\ case\ w\ of\ \{[C_{i+2}]_k\}in$$

$$(\overline{chan}_c(C_{i+2}).\overline{chan_{out}}(hash(hash(C_i)), hash(AD_{i+1} - AD_i, hash(C_i), PROOF_i))))$$

$$|chan_c(C_{i+2}).chan_{out}(y).(let(hash^2(C_i), proof_{i+1}) = y$$

$$in([(hash^2(C_i) = C_{i+2})\ and\ (proof_{i+1} = PROOF_{i+1})].\overline{result}(true)$$

$$+ [(hash^2(C_i) \neq C_{i+2})\ or\ (proof_{i+1} \neq PROOF_{i+1})].\overline{result}(false)))$$

/*host$_i$ computes C_{i+2} and proof$_{i+1}$*/

$$\xrightarrow{\tau}(\nu result)([(hash^2(C_i) = C_{i+2})\ \text{and}\ (proof_{i+1} = PROOF_{i+1})].\overline{result}(true)$$

$$+ [(hash^2(C_i) \neq C_{i+2})\text{or}(proof_{i+1} \neq PROOF_{i+1})].\overline{result}(false))$$

/*Making comparison and get the ultimate verification result*/

$$\begin{cases} \xrightarrow{(hash^2(C_i)=C_{i+2})\ and\ (proof_{i+1}=PROOF_{i+1})} (\nu result)\overline{result}(true) \\ \xrightarrow{(hash^2(C_i)\neq C_{i+2})\ or\ (proof_{i+1}\neq PROOF_{i+1})} (\nu result)\overline{result}(false) \end{cases}$$

From the reduction result above, we can see that our agent data integrity verification module can get correct result; if the agent data integrity is not tampered by $host_{i+1}$, then the ultimate result is 'true', or else the ultimate result is 'false'.

Since the reduction result of the spi-calculus model of agent data integrity verification is correct, the integrity verification scheme presented by us is also correct.

5. Simulation experiment

Based on Aglets Software Development Kit v2 (Open Source release) [23] and MAS Simulator [24], we construct the simulation experiment environment and develop a prototype system. We make some simulation experiments, the network topology used in our experiment is shown as Fig. 7. On every host of the migration path, agent collects some data from the host. In our simulation experiment, we mainly compare the P-AMFIV with other agent fault-tolerance model.

In the network topology, the migration priority comparison among different nodes at each stage is as follows: $B_1 > B_2 > B_3$, $C_1 > C_2 > C_3$, $D_1 > D_2 > D_3$.

▶ Test 1. Test the fault-tolerance ability of linear model, RAMMV and P-AMFIV

Test measure: By setting the hosts as normal or malicious ones, then make simulation agent migration.

In Table 4, *N* denotes that the host is normal, and *M* denotes that the host is malicious. If a host is malicious, then the agent will be compromised and never move further. As it is not convenient to figure out the exact path in RAMMV, so we only denote the migration result of RAMMV as 'success' or 'fail'.

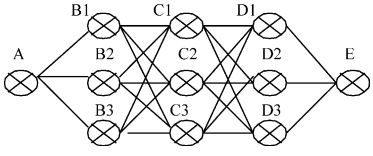

Fig. 7. Network topology used in the simulation experiment.

Analyses to the result of Test 1:

(1) If there are not malicious hosts, the three models all make agent migrate from A to E successfully.
(2) If there are any malicious hosts in the migration itinerary, the linear model fails, so it has no fault-tolerance ability. RAMMV and P-RAMFIV can make agent migrate from A to E successfully, so they have fault-tolerance ability.
(3) If the nodes in a stage are all malicious, such as No. 10, the three models all fail.

► **Test 2. Test the agent migration time of the three models when all nodes are normal hosts.**

Test measure: We set all nodes as normal hosts. By software modulation, we make the network transmission rate, network load and host CPU load change, then make several agent migration simulation experiments. In the simulation experiment, agent migrates from A to E, too.

The agent migration time comparison in Test 2 is shown in Fig. 8.

Analyses for the result of Test 2:

(1) The agent migration speed of linear model is the fastest;
(2) Though the agent migration trace of P-AMFIV is linear when all nodes are normal, the migration speed of P-AMFIV is slower than the linear model since the agent integrity verification in P-AMFIV takes time.
(3) Since many replicas are produced at every stage in RAMMV, at each stage every node waits all agent results of previous stage, and the majority voting takes time, therefore the agent migration speed in RAMMV is the slowest.

► **Test 3. Test the agent migration time of RAMMV and P-AMFIV when there are malicious hosts.**

Test measure: we set C_1 and D_1 as malicious hosts. By software modu0 0 0045ge ware n 1 kRA-10.1(ission)]TJmodug several agentmigrasimul,host59Te2. the simula7n,ta agentwaTJTt5s 2.5(A2(voti6A)-306.3(t)0(o)-300.2(E)0(,)-

Analyses for the result of Test 3:

(1) The agent migration speed of RAMMV is slower than that of P-AMFIV, since the majority voting and communication of many agent replicas in RAMMV take more time than the integrity verification in P-AMFIV.
(2) The agent migration speed gap between RAMMV and P-AMFIV in Fig. 9 is less than the one in Fig. 8, since in Test 2 all nodes are normal so the agent migration trace in P-AMFIV is linear.

▶ Test 4. Test the max memory cost of the three models when all nodes are normal hosts

Test measure: We set all nodes as normal hosts, and keep the states of the network and hosts stable, then test the max memory cost of the all nodes of the three models while executing agent simulation migration.

The max memory cost of hosts in the three models when all nodes are normal hosts can be seen in Fig. 10.

Analyses for the result of Test 4:

(1) In the linear model and P-AMFIV, since $AB_1C_1D_1E$ is the agent migration path, the memory costs of those nodes on the path are higher than that of other nodes. In RAMMV, since all nodes execute agent replicas and majority voting, the memory costs of all nodes are the highest.
(2) On the nodes A, B_1, C_1, D_1 and E, the memory cost in linear model is the lowest, the memory cost in P-AMFIV and RAMMV are almost the same.

▶ Test 5. Test the max memory cost of RAMMV and P-AMFIV when there are malicious hosts

Test measure: We set some nodes as malicious hosts (here we set B_1, C_1, C_2 and D_1 as malicious hosts), and keep the states of the network and hosts stable, then test the max memory cost of the all nodes of the two models while executing agent simulation migration.

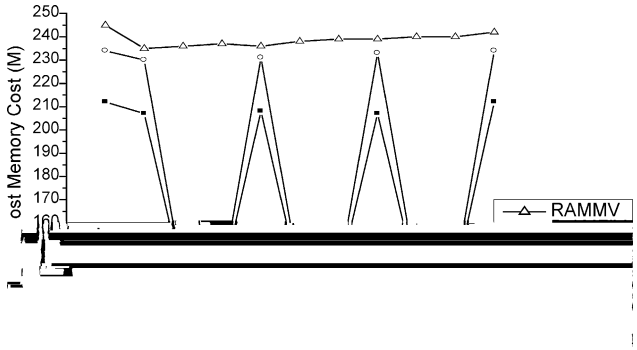

Fig. 10. The max host memory cost of each node in the three models in Test 4.

Fig. 11. The max host memory cost of each node in RAMMV and P-AMFIV in Test 5.

The max memory cost of the hosts in the two models when B_1, C_1, C_2 and D_1 are malicious hosts can be seen in Fig. 11.

Analyses for the result of Test 5:

(1) In P-AMFIV, the max memory cost of the hosts on the migration path ($AB_2C_3D_2E$) and the malicious hosts is higher than the one of other hosts.
(2) Generally, the host memory cost in RAMMV is higher than the one in P-AMFIV.
(3) Generally, the max memory cost of malicious hosts is a little more than the ones of the normal hosts, since the malicious hosts cost more memory resource when they undertake malicious acts.

Summarization: from the results of above 5 tests, we can see that P-AMFIV provided by us has the fault-tolerance ability similar to RAMMV, but the P-AMFIV can improve agent migration speed and save hardware resource cost compared to RAMMV. Moreover, the tampering of agent integrity can be detected in single hop, so the agent can re-select a new itinerary immediately.

6. Conclusion

In this paper, by incorporating the ideas of agent integrity and fault-tolerance, we suggest a novel agent migration fault-tolerance model based on integrity verification called AMFIV and its improved version P-AMFIV. Comparing to other related works of agent integrity verification, our model can detect the attack immediately and need not wait for the return of agent; comparing to other works of agent fault-tolerance, our model save network load and host resource, and also improve the agent migration speed.

To testify the correctness of P-AMFIV, this paper models P-AMFIV and the agent data integrity verification based on π/spi-calculus. The correctness of P-AMFIV is testified by making reduction based on π/spi-calculus.

To testify the efficiency of P-AMFIV, this paper makes some simulation experiments for comparing P-AMFIV with RAMMV. The result proves that P-AMFIV outperforms RAMMV.

We will continue improving our works in the future research in area, such as the security of the integrity verification protocol and the authentication between hosts, etc.

References

[1] D.M. Chess, Security issues in mobile code systems in: G. Vigan (Ed.), Mobile Agents and Security, LNCS1419, Springer, Berlin, 1998, pp. 1–14.

[2] F. Hohl, Time limited blackbox security: protecting mobile agents from malicious hosts in: G. Vigna (Ed.), Mobile Agents and Security, Springer, Berlin, 1998, pp. 92–113.

[3] F. Hohl, A protocol to detect malicious hosts attacks by using reference states 2000, available at: http://elib.uni-stuttgart.de/opus/volltexte/2000/583/.

[4] G. Vigna, Cryptographic traces for mobile agents Mobile Agents and Security, in: G. Vigna (Ed.),, Mobile Agents and Security, Springer, Berlin, 1998, pp. 137–153.

[5] W.M. Farmer, J.D. Guttma, V. Swarup, Security for mobile agents: authentication and state appraisal, Proceedings of the Fourth European Symposium on Research in Computer Security, Rome, Italy, Sept. 1996, pp. 118–130.

[6] G. Karjoth, N. Asokan, C. Gulcu, Protecting the computation results of free-roaming agents, Proceedings of the Second International Workshop on Mobile Agents (MA′98), LNCS 1477, Springer, Berlin, 1998, pp. 195–207.

[7] E.C. Vijil, S. Iyer, Identifying collusions: co-operating malicious hosts in mobile agent itineraries, In Proceedings of the 2nd International Workshop on Security in Mobile Multi-Agent Systems (SEMAS-2002), Bologna, Italy, 2002.

[8] J.S.L. Cheng, V.K. Wei, Defense against the truncation of computation results of free-roaming agents, LNCS2513, Springer, Berlin, 2002, pp. 1–12.

[9] V. Roth, On the robustness of some cryptographic protocols for mobile agent protection, LNCS 2240, Springer, Berlin, 2001. pp. 1–14.

[10] M.J. Grimley, B.D. Monroe, Protecting the integrity of agents: an exploration into letting agents loose in an unpredictable world, ACM Crossroads 1999, available at: http://www.acm.org/crossroads/xrds5-4/integrity.html.

[11] F.B. Schneider, Towards fault-tolerant and secure agentry, Invited paper, Proceedings of 11th International Workshop on Distributed Algorithms, Sarbucken, Germany, 1997.

[12] Y. Minsky, R. van Renesse, F.B. Schneider, Cryptographic support for fault-tolerant distributed computing, Proceeding of the Seventh ACM SIGOPS European Workshop, Ireland, 1996, pp. 109–114.

[13] B. Hardekopf, K. Kwiat, S. Upadhyaya, Secure and fault-tolerant voting in distributed systems 2001, available at: http://www.cs.buffalo.edu/(shambhu/resume/aero01.pdf.

[14] S. Pears, J. Xu, C. Boldyreff, Mobile agent fault tolerance for information retrieval applications: an exception handling approach, Proceeding of The Sixth International Symposium on Autonomous Decentralized Systems (ISADS′03), April 2003.

[15] S. Pleisch, A. Schiper, Fault-tolerant mobile agent execution, IEEE Transactions on Computers 52 (2) (2003) 209–222.

[16] B.S. Yee, A sanctuary for mobile agents, in: DARPA Workshop on Foundations for Secure Mobile Code 1997 available at: http://www.cs.ucsd.edu/~bsy/pub/sanctuary.ps.

[17] P. Maggi, R. Sisto, Experiments on formal verification of mobile agent data integrity properties 2002, available at: www.labic.disco.unimib.it/woa2002/papers/15.pdf.

[18] J. Parrow, An introduction to the π-calculus in: J.A. Bergstra, B.A. Ponse, S.A. Smolka (Eds.), Handbook of Process Algebra, Elsevier, New York, 2001, pp. 479–543.

[19] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I and II, Information and Computation 100 (1) (1992) 1–77.

[20] P. Sewell, Applied π-a Brief Tutorial, Computer Laboratory, University of Cambridge, July 28 2000, available at: www.cl.cam.ac.uk/users/pes20/apppi.pdf.

[21] R. Milner, The polyadic π-calculus: a tutorial in: F.L. Bauer, W. Braueer, H. Schwichtenberg (Eds.), Logic and Algebra for Specification, Springer, Berlin, 1993, pp. 203–246.

[22] M. Abadi, A.D. Gordan, A calculus for cryptographic protocols-the spi calculus 2000, available at: ftp://ftp.cl.cam.ac.uk/papers/adg/spi.ps.gz.

[23] Aglets Software Development Kit v2 (Open Source), 2002, available at: http://www.trl.ibm.com/aglets/.

[24] MAS Simulator 1.4.1, 2002. available at: http://dis.cs.umass.edu/download.html.

[25] Y. Jiang, Z. Xia, Y. Zhong, S. Zhang, The construction and analysis of an agent fault-tolerance model based on π-Calculus, Proceedings of the 2004 International Conference on Computational Science, LNCS 3038, Springer, Berlin, 2004, pp. 591–598.

Yichuan Jiang was born in 1975. He received his MS degree in computer science from Northern Jiaotong University, China in 2002. He is currently a PhD candidate in computer science of the Department of Computing and Information Technology, Fudan University, China. His research interests include mobile agent system, artificial intelligence and network security.

Zhengyou Xia was born in 1974. He received his MS degree in fuse technology from Nanjing University of Science and Technology in 1999, and received his PhD degree in computer science from Fudan University in 2004. He is currently a lecturer in the Department of Computer, Nanjing University of Aeronautics and Astronautics, China. His research interests include information security, mobile agent and active network.

Yiping Zhong was born in 1953. She is now an associate professor, and also the associate director of the Department of Computing and Information Technology of Fudan University, China. Her research interests include network system, information security and data communication.

Shiyong Zhang was born in 1950. He is now a professor and PhD supervisor, and also the director of the Center of Networking and Information Engineering of Fudan University, China. His research interests include network system, mobile agent system and network security.