A Proposed Security Protocol for Data Gathering Mobile Agents

Raja Al-Jaljouli
Software Engineering Department
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052
Australia
rjaljoli@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0510

December 2004

THE UNIVERSITY OF
NEW SOUTH WALES

Abstract

This paper addresses the security issue of the data which mobile agents gather as they are traversing the Internet. Several cryptographic protocols were presented in the literature asserting the security of gathered data. The security is based on the implementation of one or more of the following security technique: public key encryption, digital signature, and message authentication code, backward chaining, one-step forward chaining, and code-result binding. Formal verification of the protocols reveals unforeseen security flaws, such as truncation or alteration of the collected data, breaching the privacy of the gathered data, sending others data under the private key of a malicious host, and replacing the collected data with data of similar agents. So the existing protocols are not truly secure. In this paper, we present an accurate security protocol [21] which aims to assert strong integrity, authenticity, and confidentiality of the gathered data. The proposed protocol is derived from the *Multi-hops protocol* [14], where the security relies on a message authentication code, a chain of encapsulated offers, and a chained hash of a random nonce. The *Multi-hops protocol* suffers from security flaws, e.g. an adversary might truncate/ replace collected data, or sign others data with his own private key without being detected. The proposed protocol [21] refines the Multi-hops protocol by implementing the following security techniques: utilization of co-operating agents, scrambling the gathered offers, requesting a visited host to clear its memory from any data acquired as a result of executing the agent before the host dispatches the agent to the succeeding host, carrying out verifications during the agent's lifecycle in addition to the verifications upon agent's return to the initiator. The verifications are on the identity of the genuine initiator at the early execution of the agent at a visited host. The proposed protocol also implements the common security techniques such as public key encryption, digital signature, etc. The security techniques implemented in the proposed protocol would rectify the security flaws revealed in the existing protocols. We prove its correctness by analyzing the security properties using STA [44, 45], a finite-state verification tool.

Keywords: Security protocols, data security, mobile agents, formal verification methods, encryption.

1 Introduction

Mobile agents are autonomous programs that have one or more goals. They control where they execute and can run in heterogeneous environments. They act on behalf of users, and have some level of intelligence. They can collaborate and communicate with other programs and agents to accomplish their tasks. They traverse the Internet from one host to another to come closer to their data and hence, overcome limitations of latency, connectivity, and bandwidth. Also, they allow a large degree of flexibility in creating computations and organize the use of distributed resources on the Internet.

Mobile agents have been proposed for e-commerce applications, such as shopping applications [53]. They can be employed to search the network's and fellow processes' data to search for offers, negotiate the terms of agreements, or even purchase goods or services.

Mobile agents are expected to run in partially unknown and untrustworthy environments. They transport from one host to another host through insecure channels and may execute on non-trusted hosts. Thus, they are vulnerable to direct security attacks of intruders and non-trusted hosts, where intruders and non-trusted hosts can perform any of the following malicious acts:

1. Truncation of the gathered data.

2. Alteration of the collected data. It takes place if hosts collude with each other or the agent visits a host twice. The malicious host may send back the agent to an earlier host in the agent's itinerary. It could then truncate the data intermediary hosts provided and may alter the data it formerly provided without being detected if it replaces the current agent's state with the state that was present when the agent firstly visited it.

3. Impersonating the genuine initiator and hence breaching the privacy of the gathered data. An adversary may possibly intercept a message signed by the initiator of the agent. It could decrypt the signed message and then signs the decrypted message with its private key impersonating the genuine initiator. Executing hosts in the agent's itinerary would encrypt the data they provide to the agent with the public key of the adversary assuming it is the genuine initiator of the message. Hence, the adversary would be able to breach the privacy of the collected data.

4. Transmitting others data signed by the private key of a malicious host. Executing hosts in the agent's itinerary may send the data they provide to the agent signed with the corresponding private keys. An adversary might

intercept the signed data. It could decrypt the signed data and then signs the data of a particular host with its private key impersonating the genuine provider of the data.

5. Replacing the gathered data with data of similar agents. An adversary might intercept the agent. It could then replace the current agent's state with the state of a similar agent.

The security of mobile agents relies on two components: (i) *Program code and static data.* (ii) *Dynamic data* [27]. The dynamic data comprises three types of data [17] as follows:

- Fixed size changeable, e.g. global variables.
- Dynamically allocated static. Commonly are referred to as execution results
- Dynamically allocated changeable, e.g. register content and stack.

The focus of this paper is on the security of the *execution results* of mobile agents. The protocols presented in the literature [14, 20, 22, 23, 28, 29] build a proof on the security of the execution results, particularly the integrity of results based on the implementation of one or more of the following security technique: public key encryption, digital signature, message authentication code, backward chaining, one-step forward chaining, and code-result binding. The *public key encryption* uses the public key of the initiator to cipher the execution results at visited hosts so as to achieve secrecy of results. Also, it uses the public key of the succeeding host to encrypt particular verification terms, such as a hashed nonce. The *digital signature* is used when a host signs the results it provides with its private signing key so as to be authenticated as the initiator of the execution result. The rest of the methods are intended to ensure the integrity of results. The *message authentication code* incorporates the identity of the succeeding host and the characteristics of the previous execution results into the execution result at a host. The *backward chaining* incorporates characteristics of the previous execution results into the execution result at a host. The *one-step forward chaining* incorporates the identity of the succeeding host into the execution result at a host. The *code-result binding* binds the signed code to the execution results so as to ensure that the returned results belong to the code of concern. The methods are described in details in [4].

The security techniques implemented in the existing protocols are not satisfactory for the following reasons:

1. Initiators carry out verifications upon the agent's return based on both static and dynamic data that are stored within the migrating agent, which is susceptible to tampering. It is essential to ensure that the initial verification data are in a secure store independent of the store of the migrating agent. We propose to store the initial

verification data, e.g. a nonce that identifies the protocol run and the identity of the first host in the agent's itinerary, securely within a *stationary agent* that resides at the initiating host and co-operates with a major agent that traverses the Internet.

2. Initiators validate the data returned with the agent if the verification terms, e.g. message authentication codes that they compute are consistent with the returned verification terms that the agent stores. The verifications might not be accurate, especially in the case of colluding attacks. An attack can take place when two malicious hosts co-operate with each other to truncate the data acquired at intermediary hosts or substitute new data for the data they had previously provided to the agent. If a host conspires with a preceding host in the agent's itinerary and sends the agent back to it, then the preceding host would be able to truncate the data acquired at the intermediary hosts without being detected by replacing the agent's dynamic data that is current with the data that the agent had when it firstly visited its host, if the host had already stored the dynamic data including the register content and stack. Thus to ensure that none of the acquired data has been truncated or replaced and that the verification is fully accurate, it is essential to restrain colluding attacks by attempting to clear host's memory from any data acquired as a result of executing the agent. Hence, a malicious host would not be able to replace the agent's dynamic data that is current with the data that the agent had when it firstly visited its host. We propose to program the agent so that it requests an *executing host to clear its memory* from any data acquired as a result of executing the agent before it dispatches the agent to the succeeding host.

3. Executing agents encrypt the data they provide using the public key of the signer of the agent, although the signer may not be the genuine initiator. This would result in a breach of privacy of the collected data. It is essential to carry out verifications at the early execution of the agent at visited hosts to verify the identity of the genuine initiator and so ensure the encryption is done for the genuine initiator. We propose to enclose a cipher within the major agent that has the identity of the genuine initiator and carry out *verifications, at the early execution of the agent at visited hosts, on the identity of the genuine initiator* based on the securely stored cipher within the migrating agent. The verifications would detect if an adversary is impersonating itself as the genuine initiator for the purpose of breach of privacy of the gathered data.

4. Gathered results are arranged in the order of visit to executing hosts and transmitted as a chain of data. Hence, a malicious host is able to infer the data that belongs to the preceding host and may tamper with it. We propose to *jumble the gathered data within the chain* to mislead an adversary trying to truncate the offer of the preceding host. We propose to arrange the offers in a reverse order so having the dummy offer, which the initiator generates, as the last offer within the chain of offers. Assume that a non-trusted host has deleted the last offer/s in the chain. The

initiator would detect the malicious act upon the agent's return as it checks the availability of the dummy offer within the chain of offers.

The proposed protocol aims to accomplish data-confidentiality, data-authenticity, and strong data-integrity properties. It encompasses a comprehensive set of security countermeasures, which would prohibit or at least detect the malicious acts of intruders and non-trusted hosts, with emphasis on the attacks revealed in the existing protocols [14, 20, 22, 23, 28, 29]. They include the special techniques presented in the previous paragraph and the following techniques:

- Verifications upon the agent's return to the initiator.

- Transmission of cryptographic proofs of the data that agents have already gathered, along with the agent, which includes: (a) a counter that indicates the number of the actually visited hosts, (b) a chain of the encrypted offers, where each offer incorporates the following data: the data acquired at a host, an identifier of the protocol run, the order of the host among visited hosts stored as a chained hash of a random nonce, the identity of the genuine initiator, and the identity of the succeeding host, and (c) a data integrity code that encapsulates the execution results at the visited hosts.

The proposed protocol is derived from the *Multi-hops protocol* [14] where the security relies on a chain of encapsulated offers, a chain of plain data, a message authentication code, and a chained hash of a random nonce. The chain of encapsulated offers is replaced by a chain of publicly encrypted offers. The offer does not incorporate the execution results at the visited hosts. The message authentication code that incorporates the execution results at the visited hosts is replaced by a data integrity code. The Multi-hops protocol is found undetectable to the attacks, where an adversary might truncate data, replace data, or sign others data with its own private key. The proposed protocol refines the Multi-hops protocol by implementing the previously discussed set of security countermeasures and that can be summarized as follows:

1. Utilization of co-operating agents where the initial verification data is securely stored within a secondary agent that resides at the initiator and co-operates with a major agent that traverses the Internet. The intention to store the initial verification data within the secondary agent and not within the initiator's memory is to enable the initiator to trace any tampering with the initial verification data through the execution trace it creates and stores of the secondary agent, which Vigna recommends in [51].

2. Verifications at the early execution of the agent at visited hosts on the identity of the genuine initiator of the agent.

3. Scrambling the gathered offers so having a dummy offer, which the initiator generates, as the last offer within the chain of offers. Hence the malicious act of a non-trusted host, which has tried to delete the offer of its predecessor by deleting the last offer in the chain of offers, would be detected upon the agent's return. The initiator would check the availability of the dummy offer within the chain.

4. Clearing the memory of an executing host from any data acquired as a result of executing the agent before the host sends out the agent to the succeeding host in the agent's itinerary. An executing host may not respond to the request. The denial of clearing request can be traced by implementing the execution traces technique recommended by Vigna in [51].

5. Verifications upon the agent's return to the initiator.

6. Transmission of cryptographic proofs of the data that agents have already gathered, along with the agent.

Research is still ongoing for advances in securing the data that mobile agents gather. The security refers to certain security properties, such as authenticity, confidentiality, integrity, etc. The security techniques can be divided into two categories: (a) preventive techniques and (b) detective techniques. The preventive techniques hinder malicious acts to take place, whereas the detective techniques reveal the malicious acts that took place through verification processes.

Cryptographic protocols are used to secure the data acquired by mobile agents. The literature presents several protocols that assert the security of data acquired by mobile agents [12, 13, 20, 23, 28, 29, 42, 47, 50, 51, 54, 55, 56]. However, some of the existing protocols lack rigorous proofs of their correctness, such as the family of protocols in [23]: (i) Publicly verifiable chained digital signature. (ii) Chained digital signature protocol with forward privacy. (iii) Chained Mac protocol. (iv) Publicly verifiable chained signature.

From the early 90's and onwards the formal methods have been commonly used in the design and in the reasoning about the correctness of security protocols [4, 26, 30]. They provide rigorous analysis for the system design, and for establishing its correctness and reliability. Thus, they help in developing error-free security protocols. On the other hand, the testing of protocols is not enough to ensure the liability and correctness of their implementation, because of the unpredictable behavior and unbounded capabilities of adversaries, and the dynamic behavior of mobile agents. The analysis would be infinite since there are infinite set of traces and it is impossible to capture all configurations of the environment and processes and that may fail the system. The implementation and verification of the existing protocols using formal methods have later revealed subtle

flaws in security protocols and showed their failure to accomplish some or all of the claimed properties [4, 28, 30]. Intruders and malicious hosts were able to spy out data, alter data stored within agents, force unjust authenticity, etc. The CSP-based tools Casper [25] and FDR [18] are used to verify the data integrity properties of mobile agents in [20]. Also a model checker, which is based on symbolic data representation and uses Spi-calculus, is used to verify data integrity properties of mobile agents in [28].

The soundness of a protocol can be checked using formal methods of verification. We apply formal methods to model and verify proposed protocol. The protocol specifications and security properties are modeled formally using the formal verification method STA (Symbolic Trace Analyzer). STA is an infinite-state exploration that is based on symbolic techniques. It models a protocol as a system of concurrent processes, using syntax similar to the syntax of Spi-calculus [3]. A particular configuration of the system is expressed as: (a) a trace of input and output actions that results from interaction between a process and its environment, and (b) the environment's initial knowledge. Commonly, systems are analyzed by searching for an insecure state starting from an initial state, which might result in search of infinite transitions. The problem can be tackled by analyzing a finite state system by imposing restrictions e.g. finite number of messages an adversary can generate, though finding no attacks on the compact system does not guarantee that there would not be attacks on the large scale systems. STA analyzes transitions between configurations using a symbolic transition relation. The symbolic transition relation reduces infinite transitions to a single symbolic relation, where each input action should be preceded by a corresponding output action. Thus, it performs a complete exploration of the infinite state space [6, 7] without the need to impose restrictions to the model, e.g. finite set of messages an adversary can synthesize. According to the authors in [8, 9, 16] the symbolic analysis is sound and complete. Detecting an attack on the symbolic model would imply that an attack exists in the infinite standard model and vice versa. The Security properties are expressed as traces the protocol generates, and are verified by implementing the symbolic transition relation. We verified the proposed protocol for data confidentiality, data authenticity, and strong data integrity and the verifications detected no security attacks on the proposed protocol. Hence, the proposed protocol can truly accomplish the intended security properties: data confidentiality, data authenticity, and strong data integrity.

The rest of the paper is organized as follows. Section 2 presents the common notations used in describing protocols. In section 3 defines the security properties of the execution results of mobile agents which the proposed protocol aims to accomplish. Section 4 discusses the security protocols that have been presented in the literature, which address the security of the execution results of mobile agents, and the respective security flaws revealed in the protocols. Section 5 describes our security protocol and states initial assumptions. Section 6 recalls formal methods of protocol verification. Section 7 describes on the STA verification method. Section 8 models the proposed protocol and specifies its properties formally using the STA method. Section 9 presents

the results of analyzing the protocol using the STA method for different runs, such as single protocol run and two parallel runs of the protocol with the presence of an adversary. Section 10 summarizes our contribution in a conclusion. Section 11 discusses future directions of this work.

2 Protocols' Common Notations

In describing the security protocols of mobile agents, common notations are used [17, 29, 41]. The host that initiates a mobile agent is denoted as i_0 and is called the initiator. A host that the agent visits and where it gets executed is denoted as i_j, where *j* ranges from 1 to *n* respective to the order of visit. A terminating agent is an agent that returns back to the initiator i_0 following to its visit to the *n* executing hosts. Thus, the itinerary of a terminating agent is denoted as $i_0, i_1, i_2, \dots, i_n, i_0$. The execution result of the agent at host i_j is denoted as m_j. Through the agent's migration, the transfer of data *m* from host i_j to host i_k is denoted as $i_j \rightarrow i_k$: *m*, where i_j and i_k are executing hosts in the agent's itinerary. The program code and the static data are both denoted as $\prod$. The encryption of plaintext *m* into a ciphertext is written as $\{m\}_{K_{i_n}}$, where K_{i_n} is the public key of host i_n which encrypted the data. A digital signature is written as an encryption with a private signing key $S_{i_n}^{-1}$. The bare signature is the union of the digital signature and signed data and is written as $S_{i_n}^{-1}(m)$. It is assumed that it is possible to deduce the identity of the signer from a signature. The concatenation of two data m_1 and m_2 is denoted as $m_1||m_2$, where concatenation refers to appending data m_2 to another data m_1. The hash function is denoted as *h*, and $h(m)$ stands for the hashing of data *m*. Figure 1 shows the common notations used in describing protocols.

i_0	Initiating host
$i_0, i_1, i_2, \dots, i_n, i_0$	Agent's itinerary
$i_j \rightarrow i_k$: *m*	Transfer of data *m* from host i_j to host i_k
$\{m\}_{S_{i_n}^{-1}}$	Signing data *m* with the private key of host i_n
$\{m\}_{K_{i_n}}$	Encrypting data *m* with the public key of host i_n
$h(m)$	Hashing data *m*
$m_1 \| m_2$	Concatenation of data m_2 to data m_1

Fig. 1 Common notations in protocols' description

3 Security Properties

In this section specific security properties will be defined with respect to agent's data. The agent encompasses the two types of data: (i) *Static data.* (ii) *Dynamic data* [27]. The dynamic data comprises the following data types [17]:

- Fixed size changeable
- Dynamically allocated static
- Dynamically allocated changeable.

- The *fixed size changeable data* are the data that are set at the initiation of the agent but to which changes are authorized, such as global variables.

- The *dynamically allocated static data* are the data that are acquired during the life cycle of the agent but to which no later changes are authorized. Commonly, they are referred to as *execution results*.

- The *dynamically allocated changeable data* are the data that are acquired during the life cycle of the agent and to which changes might be authorized, such as register content and stack.

This paper focuses on the security of the *execution results* of terminating mobile agents. The principal scheme of data gathering mobile agents is that the agent is initiated by host i_0 and is sent out to a set of hosts $i_1, i_2, \dots, i_n$. The agent gets executed at each of the visited hosts in the agent's itinerary. The agent stores the result of execution at host i_j as m_j for $(1 \le j \le n)$. The execution results are modeled as a chain of $m_1, m_2, \dots, m_n$, which is stored within the agent. The agent, following to its visit to n executing hosts, returns the chain of results to the initiator i_0. The returned chain is expressed as $m'_1, m'_2, \dots, m'_n$. The returned execution result might differ from the genuine execution result m_j for $(1 \le j \le n)$ due to tampering acts of adversaries. Hence, it is denoted as m'_j for $(1 \le j \le n)$.

The security properties of the *execution results* of mobile agents are defined below [17, 29].

1. *Data integrity*: During the migration of the agent or its execution at visited hosts, tampering with the already stored execution results m_j for $(1 \le j \le n)$ is prevented, or at least any tampering will always be detected by the initiator upon agent's return. The data integrity requires that the chain of execution results $m'_1, m'_2, \dots, m'_n$ which is returned to the initiator i_0 matches the genuine chain of execution results $m_1, m_2, \dots, m_n$. Otherwise, the initiator i_0 has a proof that the genuine execution results had been tampered with. The data integrity property refers to the following classes of protection:

   - *Insertion resilience*: Data can only be appended to the chain $m_1, \dots, m_j$ for $(j < n)$.
   - *Deletion resilience:* Deletion of an execution result m_j for $(1 \le j \le n)$ from the chain of results $m_1, \dots, m_n$ is prevented, or at least is detected upon the agent's return to the initiator. If an execution result m_j is deleted and the chain of execution results reduces to $m_1, \dots, m_{j-1}, m_{j+1}, \dots, m_n$, then the

initiator i_0 has a proof that an execution result is deleted from the chain of results.

- *Truncation resilience:* Truncation of the chain $m_1, \ldots, m_j, \ldots, m_n$ at host i_j and reducing it to the chain $m_1, \ldots, m_j$ for $(j < n)$ is prevented, or at least is detected upon agent's return to the initiator i_0. The computed verification terms, e.g. message authentication code would indicate inconsistency with the returned chain of execution results.
- *Strong forward integrity*: None of the execution results in a chain can be modified. The property necessitates that the returned execution result m'_j matches m_j for $(1 \leq j \leq n)$.
- *Strong data integrity* requires the four classes of protection: insertion resilience, deletion resilience, truncation resilience, and strong forward integrity.

2. *Data non-repudiability*: The initiator i_0 can build a proof about the identity of host i_j that added the execution result m'_j $(1 \leq j \leq n)$ to the chain of results $m'_1, m'_2, \ldots, m'_n$.

3. *Data confidentiality*: The chain of execution results $m_1, \ldots, m_j, \ldots, m_n$ stored within the agent can only be read by the initiator i_0. No one else is permitted to learn the plain text of the ciphered execution results that are stored within the agent. Therefore, the unauthorized retrieval of information is prevented and the chain of the ciphered execution results should not reveal information about its contents to unauthorized entities. Note that an adversary may of course see the chain of ciphered execution results, shown below.

$$\{m_1\}_{K_{i_1}}, \ldots, \{m_j\}_{K_{i_j}}, \ldots, \{m_n\}_{K_{i_n}}$$

Nevertheless, as long as he is not able to get hold of the decryption keys, he is still unable to deduce the plain text m_j for $(1 \leq j \leq n)$ and the system is still deemed secure. Thus, it is fundamental to keep the encryption keys confidential during the run of a protocol session.

4. *Data authenticity*: Upon agent's return, the initiator i_0 can determine for sure the identity of host i_j that appended m'_j to the chain of execution results $m'_1, m'_2, \ldots, m'_n$. The initiator i_0 can be sure that the results that purport to be from a certain host were indeed provided by that host. Thus, an adversary should not be able to impersonate a host.

5. *Origin-confidentiality*: The identity of host i_j for $(1 \leq j \leq n)$ that generated and added the execution result m_j to the chain $m_1, \ldots, m_n$ can only be known at host i_0. An executing host i_k should not be able to deduce the identity of the previously visited hosts i_j for $(1 \leq j < k)$ from the agent. Though, it is possible for

a malicious host i_k to get the identity of host i_{k-1} where the execution result m_j was generated by analyzing the agent's dynamic data just before and after the agent visited it. Also, the identity of host i_{j-1} will possibly be revealed on the network layer. This can be prevented by using anonymous connections [49] which hide the identity of the previously visited host.

The objective of the proposed protocol is to accomplish authenticity, confidentiality, and strong integrity of the execution results of mobile agents.

4 Related work and Security flaws

Several cryptographic protocols have been presented in the literature [14, 20, 22, 23, 28, 29] that aim to secure the execution results of mobile agents. In this section we briefly discuss the protocols that aim to preserve the confidentiality, authenticity, or integrity of data gathered by mobile agents. The protocols are as follows:

- Targeted state protocol [22]
- Append only container protocol [22]
- Multi-hops protocol [14]
- Publicly verifiable chained digital signature protocol [23]
- Chained Digital Signature Protocol with Forward Privacy [23]
- Chained MAC Protocol [23]
- Publicly Verifiable Chained Signature Protocol [23]
- Configurable mobile agent data protection protocol [29]
- Mobile agent integrity protocol [20, 28]

4.1 Existing mobile agent security protocols

Reasoning about the correctness of the protocols reveals several security flaws [17, 28, 41]. The following discussion of the security protocols summarizes existing security techniques.

- The *Targeted State Protocol* [22] is proposed to ensure the confidentiality of data carried by a mobile agent. It is based on encrypting the data that should only be available to a trusted host with the public key of the host. The initiator may intend to transmit confidential data to a number of trusted hosts, so that a trusted host would only be able to learn the confidential data that is intended for it. The initiator encrypts each confidential data with the public key of the host for which the data should only be revealed, and then signs it with its private key. The targeted state would be as follows:

$$i_n \rightarrow i_{n+1}: \quad \{ \{m_1\}_{K_{i_1}}, \ldots, \{m_n\}_{K_{i_n}} \}_{S_{i_0}^{-1}}$$

The security flaw is that an adversary can strip off the initiator's signature from the targeted state, and then copy the targeted state into an agent of its own. Next, the adversary signs the targeted state with its private key impersonating itself as the genuine initiator. Next, the adversary sends its own agent to executing hosts i_1, …, i_n. Each host inspects the targeted state, decrypts the cipher text it can decrypt using its private decryption key and makes the plain text m_1, … , m_n available to the agent. The agent migrates back to the adversary carrying the plain text. Subsequently, the adversary possesses the text that is supposed to be confidential and the initiator would never detect such breach of privacy. The attack is illustrated by considering the agent's targeted state to contain a single plain text m_1 encrypted with the public key of host i_1. The initiator i_0 sends out the agent code $\prod_0$ and its targeted state to i_1, as follows:

$$i_0 \rightarrow i_1 : \quad \prod_0 , \{ \{m_1\}_{K_{i_1}} \}_{S_{i_0}^{-1}}$$

An adversary i_a intercepts the communication, and then strips off the initiator's signature. Next, it copies the targeted state $\{m_1\}_{K_{i_1}}$ into an agent $\prod_a$ of its own, and then signs the targeted state with its own signature. Next, it sends out the agent to host i_1 as follows:

$$i_a \rightarrow i_1 : \quad \prod_a , \{ \{m_1\}_{K_{i_1}} \}_{S_{i_a}^{-1}}$$

Host i_1 innocently decrypts the cipher text using its private key hence it makes the plain text m_1 available to the adversary. The agent migrates back to the adversary as follows:

$$i_1 \rightarrow i_a : \quad \prod_a , \{ m_1 \}_{S_{i_a}^{-1}}$$

- The *Append Only Container Protocol* [22] is proposed so that new objects can be appended to a container of objects in an agent but any subsequent modification or deletion of an object contained therein can be detected by the initiator i_0 upon agent's return. Also, the insertion of a new object can be detected. The protocol relies on an encrypted checksum C_n. The initial value of the checksum C_0 is a nonce r that is chosen randomly by the initiator and is encrypted with its public key so having $\{r\}_{K_{i_0}}$. The nonce must be kept secret by the initiator, and is used in the verification of the protocol upon agent's return.

The agent migrates to n hosts. Each host executes the agent, signs the execution results m_n with its digital signing key $S_{i_n}^{-1}$, computes a new checksum C_n from the previous checksum C_{n-1} and the signed results m_n , appends the execution results to the chain of objects, and then sends out the agent with the new chain to the succeeding host in the agent's itinerary. The *Append Only Container Protocol* is defined as follows:

$$i_n \rightarrow i_{n+1} : \{ \{m_1\}_{S_{i_1}^{-1}} , \ldots , \{m_n\}_{S_{i_n}^{-1}} , C_n \}$$

The checksum C_n is updated as follows:

$$C_n = \{C_{n-1} \parallel S_{i_n}^{-1}(m_n)\}_{K_{i_0}}$$

Upon agent's return, the initiator successively decrypts the checksums, and then extracts the signatures of executing hosts. Next, it verifies the extracted signatures with the corresponding objects in the container. The last verified checksum must be equal to the initial nonce *r*.

The security flaw is that an adversary may collude with a host i_j, which the agent had previously visited, and then learn the checksum C_j at i_j. Next, the adversary can truncate the container up to the *j*th object without being detected if it replaces the most recent checksum C_n with the checksum C_j that was present when the agent firstly visited host i_j for ($1 \leq j < n$). Moreover, the adversary can replace the initially provided execution result m_j with a new execution result m'_j if it computes a new valid checksum based on the learnt C_{j-1}. The same type of attack can take place if the agent visits a malicious host more than once. The attack violates the data integrity property. Suppose, the agent visits a malicious host i_j twice during its life cycle, then host i_j is able to perform any of the two attacks without being detected: (i) Truncate the chain at m_j so it is reduced to $m_1, \ldots, m_j$. Moreover, it can send out the agent with the reduced chain to a new set of executing hosts $i'_{j+1}, \ldots, i'_n$ and replace the chain of results $m_1, \ldots, m_j, \ldots, m_n$ with $m_1, \ldots, m_j, m'_{j+1}, \ldots, m'_n$. (ii) Replace the initially provided execution result m_j with a new execution result m'_j and then dispatch the agent again to hosts i_k for ($j < k \leq n$). Another security flaw is that an adversary can append arbitrary objects to the container without being detected if it updates the checksum accordingly.

The *Multi-hops Protocol* [14] has the same purpose as the Append Only Container Protocol. The protocol uses: (a) a hash chain γ_n, (b) a message authentication code μ_n, (c) a static part $\prod$ that includes: program code, and static (initialization) data, (d) a chain of execution results in plain M_n, and (e) A chain of encapsulated execution results P_n. The chain of execution results M_n at host i_n is the concatenation of the execution results m_j at visited hosts for ($1 \leq j \leq n$), and the corresponding hosts' identities i_n. The protocol binds the static part $\prod$ to the terms: M_n,P_n, γ_n, and μ_n. At instantiation, γ_n is set to $\gamma_0 = h(r)$ where *r* is chosen randomly by the initiator i_0, and the terms: μ_n, M_n,P_n are left empty. At each executing host, the agent updates the terms M_n,P_n, γ_n, and μ_n to incorporate the execution result at the host. The protocol is described as follows:

$$\gamma_n = h(\gamma_{n-1})$$
$$\mu_n = h(m_n, \gamma_{n-1}, \mu_{n-1}, i_{n+1})$$
$$P_n = P_{n-1} \parallel S_{i_n}^{-1}(\mu_n)$$

$$M_n = M_{n-1} \,||\, m_n \,||\, i_n$$
$$i_n \rightarrow i_{n+1}: \ (\prod, M_n, P_n), \{\gamma_n\}_{K_{i_{n+1}}}, \mu_n$$

The message authentication code μ_n that is computed at host i_n acts as a *chaining relation* that incorporates: (a) the nonce computed at the preceding host γ_{n-1}, (b) the message authentication code computed at the preceding host μ_{n-1}, which summarizes all execution results previously obtained by the agent, (c) the execution result at the current host m_n, and (d) the identity of next host i_{n+1} in the agent's itinerary. The protocol data P_n is a chain of the signed message authentication codes μ_n for $(1 \leq j \leq n)$.

The security flaw is that an adversary can collude with host i_j, which the agent had previously visited, and then learn γ_{j-1} and μ_{j-1}. Next, it can truncate the gathered data just after data of host i_j and then send the agent with the learnt values of γ_{j-1} and μ_{j-1} to host/s of its selection. The visited hosts would append data to the protocol and compute valid values for γ_j and μ_j using the learnt values of γ_{j-1} and μ_{j-1}. Moreover, the adversary can replace the initially provided execution result m_j with a new execution result m'_j if it computes valid values for γ_j and μ_j using the learnt values of γ_{j-1} and μ_{j-1}. Hence, the initiator would never detect the data truncation or replacement.

- The *Publicly Verifiable Chained Digital Signature Protocol* [23] aims to preserve the confidentiality and integrity of data acquired by mobile agents. The protocol uses a hash chain C_n and a chain of encapsulated execution results $(M_1, \ldots, M_n)$. The hash chain C_n binds the encapsulated execution result at the preceding host M_{n-1} to the identity of the next host in the agent's itinerary i_{n+1}. The encapsulated execution result M_n incorporates the execution result at the current host m_n, the randomly selected nonce r_n, and the hash chain C_n. The nonce r_n prevents an adversary from attacking the encryption. The m_0 is a dummy data provided by the initiator i_0.

The protocol is defined as follows:

$$M_n = \{ \{m_n, r_n\}_{K_{i_0}}, C_n \}_{S_{i_n}^{-1}}$$
$$C_n = h(M_{n-1}, i_{n+1})$$
$$M_0 = \{ \{m_0, r_0\}_{K_{i_0}}, C_0 \}_{S_{i_0}^{-1}}$$
$$C_0 = h(r_0, i_1)$$
$$i_n \rightarrow i_{n+1}: \{M_0, \ldots, M_n\}$$

The security of the protocol is based on the assumption that an attacker does not change the last element M_n in the chain.

The security flaw is that an adversary can truncate chain elements and can grow a fake stem, since the input to all previous chaining relations is known. Elements can be

appended to the chain at the discretion of the adversary, though the validity of the chaining relation is maintained. The adversary sends the agent with a chain of execution results, e.g. $M_0, \ldots, M_{j-1}$ to host i_j of its own choice and repeats the process until it is satisfied with the collected elements. Then, the adversary chooses an element and pastes it into agent and sends the agent to i_{j+1}. Another security flaw is that an adversary can append arbitrary objects, generated for the terms of the adversary rather than the initiator, to the container without being detected.

- The *Chained Digital Signature Protocol with Forward Privacy* [23] has the same purpose as the Publicly Verifiable Chained Digital Signature Protocol as well as forward privacy/ origin confidentiality. It proposes a change in the order of encrypting and signing the execution results so as to accomplish forward privacy. The execution results at a visited host are firstly signed by the host and then are encrypted with the public key of the initiator. Hence, no one other than the initiator can decrypt the ciphered execution results. The protocol uses a hash chain C_n and a chain of encapsulated execution results ($M_1, \ldots, M_n$). The hash chain C_n binds the encapsulated execution result at the preceding host M_{n-1} to the identity of the next host in the agent's itinerary i_{n+1} and a random nonce the host selects r_n. The encapsulated execution result M_n incorporates the execution result at the current host m_n, the randomly selected nonce r_n, and the hash chain C_n. The nonce r_n prevents an adversary from attacking the encryption. The m_0 is a dummy data provided by the initiator i_0.

The protocol is defined as follows:

$$M_n = \{\{m_n\}_{S_{i_n}^{-1}}, r_n\}_{K_{i_0}}, C_n$$
$$C_n = h(M_{n-1}, r_n, i_{n+1})$$
$$M_0 = \{\{m_0\}_{S_{i_0}^{-1}}, r_0\}_{K_{i_0}}, C_0$$
$$C_0 = h(r_0, i_1)$$
$$i_n \rightarrow i_{n+1}: \{M_0, \ldots, M_n\}$$

The problem with the protocol is that executing hosts would not be able to know the identity of the initiator of agent, since the signature of the initiator is encrypted within M_0.

The security flaw of the protocol is that an adversary can truncate chain elements and can grow a fake stem, since the input to all previous chaining relations is known. Elements can be appended to the chain at the discretion of the adversary, though the validity of the chaining relation is maintained. The adversary sends the agent with a chain of execution results, e.g. $M_0, \ldots, M_{j-1}$ to host i_j of its own choice and repeats the process until it is satisfied with the collected elements. Then, the adversary chooses an element and pastes it into agent and sends the agent to i_{j+1}. Another security flaw is that an adversary can append arbitrary objects, generated for the terms of the adversary rather than the initiator, to the agent without being detected.

▪ The *Chained MAC Protocol* [23] aims to preserve confidentiality, integrity, and forward privacy of data acquired by mobile agents. It does not provide authenticity. The protocol uses a hash chain C_n and a chain of encapsulated execution results (M_0, … , M_n). The executing host i_n computes the hash chain of the succeeding host C_{n+1}. The hash chain C_{n+1} binds the identity of the succeeding host i_{n+1} to the hash chain C_n, execution results m_n, and a random nonce r_n generated at the current host i_n. The encapsulated execution result at a host M_n binds the random nonce r_n and the execution results m_n generated at the host to the identity of the succeeding host i_{n+1}. -1

$$M_n = \{ r_n , m_n , i_{n+1} \}_{K_{i_0}} \qquad \text{for } n \geq 0$$

$$C_{n+1} = h (C_n , r_n , m_n , i_{n+1}) \qquad \text{for } n \geq 1$$

$$C_0 = \{ r_0 , m_0 , i_1 \}_{K_{i_0}}$$

$$i_n \rightarrow i_{n+1} : \{ M_0 , \dots , M_n \}, C_{n+1} \qquad \text{for } n \geq 0$$

The problem with the protocol is that executing hosts would not be able to know the identity of the initiator of agent.

The security flaw of the protocol is that an adversary can collude with host i_j, which the agent had previously visited and had stored the hash chain C_j , and sends the agent back to it. The host can then truncate the gathered data just after data of host i_j and replace its initial encapsulated execution result M_j with a new encapsulated execution result M'_j . Next, it sends the agent with the updated hash chain C'_{j+1} to host/s of its selection. Hence, the initiator would never detect the data truncation or replacement. Another security flaw is that an adversary can truncate chain elements and can grow a fake stem, since the input to all previous chaining relations is known. Elements can be appended to the chain at the discretion of the adversary, though the validity of the chaining relation is maintained. The adversary sends the agent with a chain of execution results, e.g. M_0, … , M_{j-1} to host i_j of its own choice and repeats the process until it is satisfied with the collected elements. Then, the adversary chooses an element and pastes it into agent and sends the agent to i_{j+1}. Another security flaw is that an adversary can append arbitrary objects, generated for the terms of the adversary rather than the initiator, to the container without being detected.

▪ The *Publicly Verifiable Chained Signature Protocol* [23] aims to preserve confidentiality and integrity of data acquired by mobile agents. The protocol does not provide authenticity. The protocol uses temporary key pairs (private signing key, and the corresponding verification key) and a chain of encapsulated execution results. Each host generates a pair of keys (private and public). The host encloses the public key y_{n+1} it generates within the encapsulated execution result computed at its host M_n and then signs the encapsulated execution result with the private key y_n it received from its predecessor. It provides its successor with the private key it generates y_{n+1}. At initiation, the initiator provides the agent with a dummy data m_0 and an initial key pair. It encloses the public

key y_1 it generates within the encapsulated execution result M_0 and then signs the encapsulated execution result with its private key. It provides host i_1 with the private key y_1. The hash chain C_n at a host bins the encapsulated execution results at the preceding host M_{n-1} to the identity of the succeeding host i_{n+1}. The encapsulated execution result M_n incorporates the following data terms that are generated at the host: data m_n , a random nonce r_n , a hash chain C_n , and a public verification key corresponding to the succeeding host y_{n+1}.

$$M_n = \{\{ m_n , r_n \}_{K_{i_0}} , C_n , y_{n+1} \}_{S_{y_n}^{-1}} \quad \text{for } n \geq 1$$

$$M_0 = \{\{ m_0 , r_0 \}_{K_{i_0}} , C_0 , y_1 \}_{S_{i_0}^{-1}}$$

$$C_n = h(M_{n-1} , i_{n+1}) \quad \text{for } n \geq 1$$

$$C_0 = h\{ r_0 , i_1 \}$$

$$i_n \rightarrow i_{n+1} : \{ M_0 , \ldots , M_n \}, \overline{y_{n+1}} \quad \text{for } n \geq 0$$

The security flaws of the protocol are: (i) An adversary can intercept the agent and then decrypts the encapsulated execution result M_0 with the signature verification key of the signer. Next, it signs the decrypted term of M_0 with its private key impersonating the genuine initiator. Executing hosts would encrypt the data they provide to the agent with the public key of the adversary believing that it is the genuine initiator. Hence, the adversary can breach the privacy of the gathered data. (ii) An adversary can collude with host i_j, which the agent had previously visited and had stored the private key it had received from the preceding host $\overline{y_n}$, and sends the agent back to it. The host can then truncate the gathered data just after data of host i_j and replace its initial encapsulated execution result M_j with a new encapsulated execution result M'_j . Next, it sends the agent to host/s of its selection. Hence, the initiator would never detect the data truncation or replacement. (iii) An adversary can truncate chain elements and can grow a fake stem, since the input to all previous chaining relations is known. Elements can be appended to the chain at the discretion of the adversary, though the validity of the chaining relation is maintained. The adversary sends the agent with a chain of execution results, e.g. $M_0, \ldots , M_{j-1}$ to host i_j of its own choice and repeats the process until it is satisfied with the collected elements. Then, the adversary chooses an element and pastes it into agent and sends the agent to $i_{j+1.}$ (iv) An adversary can append arbitrary objects, generated for the terms of the adversary rather than the initiator, to the container without being detected.

▪ The *Configurable Mobile Agent Data Protection Protocol* [29] is intended to accomplish a combination of the security properties: authenticity, confidentiality, integrity, origin confidentiality/ forward privacy, and non-repudiation. The protocol can be configured for the properties of concern. The security is based on: (a) securely storing the addresses of next hosts to be visited, and (b) binding the static part (program code and static data) $\prod$ to a chain of encapsulated execution results at hosts in the agent's

itinerary ($\mathcal{M}_0$, …, $\mathcal{M}_n$) [40, 41]. The static part $\prod$ is paired with a timestamp *t* and signed by the initiator so having $\prod_0 = \{\prod, t\}_{S_{i_0}^{-1}}$. The P_0 is the agent's initial itinerary. The P_n is the set of new hosts added to agent's initial itinerary. The $\mathcal{M}_n$ is a chain of encapsulated execution results. The $\mathcal{M}_n$ is composed of two parts: D_n and C_n. The D_n binds the execution results at a host d_n to P_n. The C_n binds the static part $\prod_0$ to: (a) the C_{n-1} that is computed at the preceding host, (b) the identity of the succeeding host i_{n+1}, (c) the execution result d_n at host i_n , and (d) the addresses of new hosts P_n added to agent's initial itinerary P_0.

The protocol configuration for data-authenticity, data-confidentiality, and data-integrity properties is as follows:

$$i_n \rightarrow i_{n+1}: \quad \prod_0, \{\mathcal{M}_0, \ldots, \mathcal{M}_n\}$$

where, $\prod_0 = \{\prod, t\}_{S_{i_0}^{-1}}$, and $\mathcal{M}_n = D_n \,||\, C_n$

$$D_n = \begin{cases} P_0 & \text{if } i_n = i_0 \\ \{d_n\}_{K_{i_0}^{+}}, P_n & \text{otherwise} \end{cases}$$

$$C_n = \begin{cases} S_{i_0}^{-1}(P_0, \prod_0, i_1) & \text{if } i_n = i_0 \\ \{S_{i_n}^{-1}(d_n, P_n, \prod_0, C_{n-1}, i_{n+1})\}_{K_{i_0}^{+}} & \text{otherwise} \end{cases}$$

The security flaw is that an adversary i_a may intercept the communication between i_1 and i_0, and then can strip off the initiator's signature from $\prod_0$ and learn the pair ($\prod$, *t*) in plain text. Next, it extracts the tuple (P_0, $\prod_0$, i_1), and then appends its signature to the tuple so having $S_{i_a}^{-1}$ (P_0, $\prod_0$, i_1). Next, it signs the pair ($\prod$, *t*) with its private key so having $\prod_0' = \{\prod, t\}_{S_{i_a}^{-1}}$, and computes $C_0' = S_{i_a}^{-1}(P_0, \prod_0, i_1)$. Next, it sends the agent with the fake identifiers $\prod_0'$, and C_0' rather than the original identities $\prod_0$, and C_0 to host i_1 impersonating the genuine initiator. Host i_1 would receive the agent and incorrectly authenticate i_a as the genuine initiator of the agent. Thus, it would encrypt its own data with the public key of the adversary i_a rather than that of the genuine initiator i_0. The agent continues migrating till all hosts defined in the set {P_0, … , P_n} are visited. Next, the adversary intercepts the agent and spies out the gathered data. Next, the intruder i_a sends the agent with the initial identifiers $\prod_0$, D_0, C_0 as a fresh protocol instance. The attack would result in erroneous authenticity to i_a and breach of privacy of the gathered data. Actually, host i_a sends two instances of the protocol. The first instance with D_0 and the fake identifiers: $\prod_0'$, and C_0', and the second instance with original identifiers: $\prod_0$, D_0, and C_0. As a result, the adversary would possess the data that should only be revealed to the genuine initiator. The initiator would never detect such breach of privacy. Another attack is that an adversary can truncate the data acquired at hosts visited between the first and the second visits of the agent to its host hence can alter the data it has provided to the agent in the first visit maintaining the consistency of checksums of

the gathered data.

▪ The *Mobile Agent Integrity Protocol* [20, 28] aims to preserve the integrity of data gathered by mobile agents. The protocol is based on the chain of execution results AD_n, a message integrity code MIC_n, and hash chain of a nonce C_n. The AD_n is a chain of execution results at hosts in the agent's itinerary as $\{D_1, \ldots, D_n\}$. The MIC_n binds the execution result D_n and hash chain of the nonce C_n computed at the current host to the MIC_{n-1} computed at the preceding host. Upon agent's return, the code is verified to detect any tampering with the already gathered data. At instantiation, C_n is set to $C_0 = r$ where r is chosen randomly by the initiator i_0. The MIC_0, and AD_0 are left empty. The protocol is defined as follows:

$$AD_n = \{D_1, \ldots, D_n\} = AD_{n-1} \cup \{D_i\}$$
$$MIC_n = h\,(D_n, C_n, MIC_{n-1})$$
$$C_n = h\,(C_{n-1})$$
$$i_n \rightarrow i_{n+1} : MIC_n, \{C_n\}K^{+}_{i_{n+1}}, AD_n$$

Upon agent's return, the initiator computes MIC'_n from the values of $\{C_0, \ldots, C_n\}$ and $\{D_1, \ldots, D_n\}$ and then verifies that the computed message integrity code MIC'_n matches the message integrity code that has just been received from the agent.

The security flaw is that an adversary can read and append arbitrary data to the already gathered data, since the terms MIC_{n-1} and C_{n-1} that are needed to compute the integrity code MIC_n at host i_n are known. Another security flaw is that a non-trusted host can append arbitrary objects, generated for the terms of the adversary rather than the initiator, to the container without being detected.

4.2 Causes of security flaws and remedies

The discussion in section 4.1 is advantageous in identifying the protocol specifications that would result in security flaws and guiding us in proposing a security protocol that: (a) implements the reliable existing security techniques, (b) avoids protocols' specifications that may result in security flaws, and (c) is capable of preventing or detecting the security threats that existing protocols fail to detect. Table 1 summarizes the flaws revealed in existing security protocols.

Table 1. Flaws revealed in the mobile agents security protocols

Security protocol	**Aimed properties**	**Failed property**	**Type of flaw/s ***
Targeted state protocol	Confidentiality	Confidentiality	Adversary can *breach privacy* of collected data
Append only container protocol	Authenticity Integrity	Authenticity Integrity	Returned data are *erroneously authenticated* Co-operating hosts can *truncate collected data* Adversary can *append fake data.* Hence, returned data may not belong to agent of concern
Multi-hops protocol	Integrity Authenticity	Integrity	Co-operating hosts can *truncate collected data*. Adversary can *append fake data.* Hence, returned data may not belong to agent of concern
Publicly verifiable chained digital signature protocol	Integrity Confidentiality	Integrity	Co-operating hosts can *truncate collected data* Adversary can *append fake data.* Hence, returned data may not belong to agent of concern
Chained Digital Signature Protocol with Forward Privacy Protocol	Integrity Confidentiality Forward privacy	Integrity	Co-operating hosts can *truncate collected data* Adversary can *append fake data.* Hence, returned data may not belong to agent of concern
Chained MAC Protocol	Integrity Confidentiality Forward privacy	Integrity	Co-operating hosts can *truncate collected data* Adversary can *append fake data.* Hence, returned data may not belong to agent of concern
Publicly Verifiable Chained Signature Protocol	Integrity Confidentiality	Integrity	Co-operating hosts can *truncate collected data* Adversary can *append fake data.* Hence, *data may not belong* to agent of concern
Configurable mobile agent data protection protocol	Confidentiality Authenticity Integrity Non-repudiation Forward privacy	Confidentiality Integrity	Adversary can *breach privacy* of collected data Co-operating hosts can *truncate collected data*
Mobile agent integrity protocol	Integrity	Integrity	Co-operating hosts can *truncate collected data* Adversary can *append fake data.* Hence, returned *data may not belong* to agent of concern

* Flaws would not be detected by the initiator i_0.

The security flaws are attributed to the followings with respect to the security attack:

A. Breach of privacy

Problem: In the *Targeted State Protocol* and static part in the *Configurable Mobile Agent Data Protection Protocol* are signed with the initiator's private key. An adversary might intercept the agent and decrypts the signed data/ part. It would then send the decrypted data/ part signed with its private key. Recipients would assume that the signer is the genuine initiator of the agent, and would then encrypt the data they provide to the agent with the public key of the adversary. Hence, the adversary would be able to learn the encrypted data.

Remedy: the initiator should follow the signing of a term by an encryption with the public key of the recipient. The term would be received signed with the signature of the genuine initiator. An adversary would not be able to decrypt the signed term and signs it with its private key.

B. Erroneous authentication

Problem: In the *Append Only Container Protocol*, the execution result at a host is just encrypted with the private key of the host. An adversary may intercept the agent and decrypts the execution results. It would then sign the execution results with private keys of co-operating hosts, and update the checksum accordingly. Consequently, the initiator would assume that the returned data were provided by genuine executing hosts. The same flaw exists in the *Chained MAC Protocol* and the *Publicly Verifiable Chained Signature* Protocol. The two protocols do not aim to preserve authenticity, but it is just a remark.

Remedy: an executing host should firstly sign the data it provides to the agent with its private key and then encrypts it with the public key of the initiator. The data would be received at the initiator signed with the respective private keys of the genuine executing hosts.

C. Appending a fake stem to the agent

The inputs that are needed to compute an encapsulated execution results at a hosts are available. Hence, an adversary can append fake execution results to the agent without being detected as follows:

1. An intruder may intercept the agent and append an offer to the results of the agent's execution. The intruder is a non-scheduled host in the agent's itinerary. The initiator would not detect the malicious act upon the agent's return. The attack is possible in the *Append Only Container Protocol*.

Remedy: The terms that are necessary to compute an encapsulated execution result at a host and were computed at the predecessor host should be transmitted from the predecessor host to the host encrypted with the public key of the host. The terms are depicted in Table 2

In the following protocols, a non-trusted host that participates in the protocol may send the agent to a succeeding host of its selection. The succeeding host would append its offer to the execution results of the agent, though the initiator would detect the malicious act upon the agent's return. Each partial execution result incorporates the identity of the respective succeeding host.

- Multi-Hops Protocol
- Publicly Verifiable Chained Digital Signature Protocol
- Mobile Agent Integrity Protocol
- Chained MAC Protocol
- Publicly Verifiable Chained Signature Protocol
- Configurable Mobile Agent Data Protection Protocol
- Chained Digital Signature Protocol with Forward Privacy Protocol

2. The execution results that are returned to the agent might be generated for a different protocol run or for a different initiator.

    - The execution results of the *Append Only Container* protocol neither incorporate an identifier of the protocol run of concern nor the identity of the initiator within the execution results.

    - The execution results of the following protocols incorporate a random nonce generated by the initiator within the execution results:

        - Multi-Hops Protocol
        - Mobile Agent Integrity Protocol

    - The execution results of the Configurable Mobile Agent Data Protection Protocol incorporate the followings within the execution results.

        - Timestamp generated by the initiator and uniquely identifies the protocol run of concern
        - Identity of the first host in the agent's itinerary
        - Identity of the initiator within the execution results

    - The execution results of the following protocols:

- Publicly verifiable chained digital signature protocol
- Chained Digital Signature Protocol with Forward Privacy Protocol
- Chained MAC Protocol
- Publicly Verifiable Chained Signature Protocol

incorporate the followings within the execution results:

- Random nonce generated by the initiator that uniquely identifies the protocol run of concern
- Dummy data generated by the initiator
- Identity of the first host in the agent's itinerary
- Identity of the initiator within the execution results.

Remedy: incorporate the following terms within each encapsulated execution result.

- Random nonce generated by the initiator or a timestamp generated by the initiator and uniquely identifies the protocol run of concern
- Dummy data generated by the initiator
- Identity of the first host in the agent's itinerary
- Identity of the initiator within the execution results.

Moreover, store the terms securely with an agent that is stationary at the initiator. It might be assumed that it is enough to store the terms securely in the memory of the initiator, though an adversary might tamper with the memory of initiator. Hence, the verifications of the two terms would not be accurate. The use of a secondary agent to store the verification terms (identity of the initiator and the identifier of the protocol run) would enable the initiator to trace any manipulation with the terms by the use of execution traces. Vigna in [51] recommends the agent executor to create a trace of the agent's execution. The trace contains the lines of the agent's code that were executed as well as any new values assigned to initial verification terms that were stored within the stationary agent. The trace of the agent's execution is to be stored at the executing host for a limited time. Upon the initiator's request, each executing the host signs the execution trace and forwards it to the succeeding host in the agent's itinerary. The accumulative execution traces are forwarded to the initiator. We propose to implement the technique to store the trace of execution of the secondary at the initiator, thus the initiator would be able to verify the terms upon the agent's return through the stored execution trace. Usually execution traces require large amounts of resources to the storage of validating information. Conversely the execution trace of the secondary agent would be short as compared to the execution traces of the migrating agent.

D. Truncation and/ or substitution of execution results

Problem: An adversary can truncate the data acquired at hosts visited between the first and the second visits of the agent to its host hence can alter the data it has provided to the agent in the first visit maintaining the consistency of the chaining relation of the gathered data. The attack requires that the non-trusted host has stored the chaining relation that was present when the agent has firstly visited it. Table 2 identifies the terms computed at the predecessor host that are necessary to enable an adversary to truncate and/ or alter gathered data and to maintain the consistency of chaining relation/s. Hence, the initiator would not be able to detect the malicious act of the adversary.

Remedy: Ensure that an executing host clears its memory from any terms acquired as a result of executing the agent before it dispatches the agent to the next host in the agent's itinerary. We propose to design the migrating agent in such a way that it requests an executing host to clear its memory from any terms acquired as a result of executing the agent before it dispatches the agent to the next host in the agent's itinerary. However, an executing host may not respond to the request. The denial of clearing request can be traced by implementing the execution traces technique recommended by Vigna in [51]. The technique requests an executing host to create and sign the execution trace, and to store it so as to be forwarded to the initiator upon request. We recommend the execution trace to be limited to the line of code that requests the clearing of the memory of the executing hosts; otherwise the trace of all executable lines of the agent would be extremely long and require large amounts of resources of storage at the executing hosts. Moreover, it would lead to overburden the communication channels as traces are transmitted to the initiator upon request.

Table 2 Terms computed at the predecessor host and are necessary for an adversary to perform a non-detectable data truncation/ alteration

Security protocol	**Necessary terms**
Targeted state protocol	None
Append only container protocol	Checksum C_{n-1}
Multi-hops protocol	Message authentication code μ_{n-1} , hash chain γ_{n-1}
Publicly verifiable chained digital signature protocol	Encapsulated execution results M_{n-1}
Chained Digital Signature Protocol with Forward Privacy Protocol	Encapsulated execution results M_{n-1}
Chained MAC Protocol	Hash chain C_n computed at its predecessor
Publicly Verifiable Chained Signature Protocol	Encapsulated execution results M_{n-1}, and the private signature it received from its predecessor y_{n-1}
Configurable mobile agent data protection protocol	Static part and a timestamp signed by initiator $\prod_0$, chaining relation C_{n-1}
Mobile agent integrity protocol	Hash chain of the nonce C_{n-1} , and a Message Integrity Code MIC_{n-1}

5 The Proposed Protocol

In the previous section, we can notice that the existing protocols failed to hinder or detect at least one of the following security threats:

- Breach of privacy
- Erroneous authentication
- Truncation of data
- Irrelevant data gathering

The *breach of privacy* can be prohibited by encrypting the data with the public key of the recipient host, whereas, the *erroneous authentication* cannot be prohibited just by transmitting a digitally signed data. An adversary can decrypt the signed data with the respective public verification key and then signs the data with its private key. However, a host can prohibit the erroneous authentication flaw by transmitting the signed data encrypted with the public key of the recipient host. The *irrelevant data gathering* can be detected by incorporating initial verification terms within the gathered data. The verification terms identify the agent of concern. The initiator could then check the availability of the terms within the data returned to the agent. If the check fails, then the gathered data are irrelevant to the agent of concern. The *truncation* of data is the common flaw in the existing security protocols and is the most difficult to deter. In this paper, we propose a security protocol that aims to preserve authenticity, confidentiality, and strong integrity of the execution results of data gathering mobile agents. The main focus is to ensure the strong integrity of the results with the emphasis on robustness to data-truncation.

In setting up the protocol, we assume free-roaming mobile agents, which are free to autonomously choose the next host in the agent's itinerary. The choice would be based on the data acquired through their execution. Agents are assumed to migrate through public channels. Also, we assume that hosts execute the right code of an agent. Thus, an agent migrating from one host to another is simply represented by a message that only contains the execution results of the agent. The public encryption keys of the initiator and a preceding host in the agent's itinerary, which a participating host would need for encrypting the data to transmit, can be found in the server's known-hosts list or are distributed to the host upon the request of the key from the relevant host. We consider the Dolev-Yao model of intruder [55] and apply it to mobile agents. A mobile agent migrates to every host in the agent's itinerary where it gets executed and gathers the execution results. Finally the agent returns to the initiator where the execution results are decrypted, verified and sorted out for a decision making. The intruder may impose an attack on the data provided by one or more of the executing hosts. It would be able to intercept, read, delete, fake, append, insert, or replace any of the data gathered by mobile agents. The initial knowledge of an adversary includes channels names, identities of

participating hosts and their respective public keys and signature verification keys, a message it can intercept, and an old nonce.

5.1 Notations specific to the protocol

Figure 2 summarizes the notations that are used to describe the proposed protocol.

$\mathcal{A}$	Major agent
$\mathcal{A}$s	Secondary agent
n	Number of visited hosts
j	Order of the current host among the visited hosts
i_0	Identity of the initiator
i_j	Identity of an executing host, where $1 \leq j \leq n$
r	Nonce freshly generated by the initiator that identifies a protocol run
m_j	Data requested by the major agent $\mathcal{A}$ and generated by host i_j
m_0	Dummy data generated by host i_0
δ_j	Data integrity code at i_j
λ_j	Offer provided at host i_j
λ	Chain of jumbled offers $(\lambda_j , \lambda_{j-1} , \dots , \lambda_1 , \lambda_0)$
γ_j	Hash chain of a nonce that indicates the number of the actually visited hosts

Fig.2 Notations used in describing the protocol

5.2 Formal description of the proposed protocol

The proposed protocol requires the initiator to create two co-operating agents $\mathcal{A}$ and $\mathcal{A}s$. The agent $\mathcal{A}$ is a *major agent* that traverses the Internet and gathers particular data. At the first instance of the protocol run, the initiator i_0 generates a fresh nonce r that identifies the run. Next, the initiator dispatches the agent $\mathcal{A}$ to host i_1, and then the agent is free to autonomously choose the next host to visit at each migration step during its life cycle. Each visited host provides the agent $\mathcal{A}$ with the requested data. When the agent $\mathcal{A}$ completes its execution at last host in the agent's itinerary, it returns to the initiator with the results of execution at the visited hosts. Upon the agent's return, the agent $\mathcal{A}$ co-operates with the secondary agent $\mathcal{A}s$, which resides at the initiating host and securely stores the initial verification data, and carries out a set of verifications on the execution results. The collective data from the two agents would be utilized in the detection of any malicious act performed on the execution results, and would be sufficient to identify any tampering with the results. The agent $\mathcal{A}$ can be represented as a sequence of messages communicated between the executing hosts, with the initiating host starting the sequence of messages by sending a preliminary message and finally receiving the summary message that summarizes the execution results at different executing servers. The

migration of the agent $\mathcal{A}$ from one host to anther is simply modeled by sending a data message. The protocol can be described as follows:

$$\gamma_j = h(\gamma_{j-1}), \qquad \text{where } \gamma_0 = r$$

$$\delta_j = h(m_j , \delta_{j-1}) , \quad \text{where } \delta_0 = h(i_0)$$

$$\lambda_j = \{ \{m_j , \delta_0 , i_{j+1} , \gamma_j \} S_{i_j}^{-1} \} K_{i_0}^{+} , \text{ where } \lambda_0 = \{m_0\} K_{i_0}^{+}$$

$$i_j \rightarrow i_{j+1} : \{ \lambda_j , \ldots , \lambda_1 , \ldots , \lambda_0, \delta_j , \{ \delta_0\} S_{i_0}^{-1} , \gamma_j \} K_{i_{j+1}}^{+}$$

5.3 Sequence of messages in the proposed protocol

The protocol can be expressed as a sequence of messages communicated between the hosts participating in the protocol run. For simplicity, we consider a small instance of the protocol of an initiator and three executing hosts. The agent starts its itinerary from the initiator i_0. Next, it migrates to hosts i_1, i_2, and i_3 successively. Finally, it returns to the initiator i_0 with the execution results. Figure 3 shows the protocol expressed as a sequence of messages.

Message 1 $i_0 \rightarrow i_1$:
$$\{ \{m_0\}_{K_{i_0}^{+}} , h(i_0), \{h(i_0)\} S_{i_0}^{-1} , r \} K_{i_1}^{+}$$

Message 2 $i_1 \rightarrow i_2$:
$$\{ \{ \{m_1, h(i_0), i_2, h(r)\} S_{i_1}^{-1} \} K_{i_0}^{+}, \{m_0\} K_{i_0}^{+},$$
$$h (m_1, h(i_0)), \{h(i_0)\} S_{i_0}^{-1} , h(r) \} K_{i_2}^{+}$$

Message 3 $i_2 \rightarrow i_3$:
$$\{ \{ \{m_2, h(i_0), i_3, h^2(r)\} S_{i_2}^{-1} \} K_{i_0}^{+},$$
$$\{ \{m_1, h(i_0) , i_2, h(r)\} S_{i_1}^{-1} \} K_{i_0}^{+}, \{m_0\} K_{i_0}^{+},$$
$$h(m_2, h(m_1, h(i_0))), \{h(i_0)\} S_{i_0}^{-1} , h^2(r) \} K_{i_3}^{+}$$

Message 4 $i_3 \rightarrow i_0$:
$$\{ \{ \{m_3, h(i_0), i_0, h^3(r)\} S_{i_3}^{-1} \} K_{i_0}^{+} ,$$
$$\{ \{m_2, h(i_0), i_3, h^2(r)\} S_{i_2}^{-1} \} K_{i_0}^{+} ,$$
$$\{ \{m_1, h(i_0) , i_2, h(r)\} S_{i_1}^{-1} \} K_{i_0}^{+}, \{m_0\} K_{i_0}^{+},$$
$$h\big(m_3, h\big(m_2, h(m_1, h(i_0)) \big) \big), \{h(i_0)\} S_{i_0}^{-1} , h^3(r) \} K_{i_0}^{+}$$

Fig. 3 Sequence of messages in the proposed protocol

5.4 Origin of the proposed protocol

The proposed protocol is derived from the *Multi-hops protocol* [14], which was described in section 4. The security of the Multi-hops protocol relies on : (a) a hash chain γ_n, (b) a message authentication code μ_n, (c) a static part $\prod$ that includes: program code, and static (initialization) data, (d) a chain of the execution results M_n in plain, and (e) a chain of encapsulated offers P_n. In the *Multi-hops protocol*, the message authentication code μ_n at host i_n incorporates: (a) the data m_n that the host provides, (b) the identity of the succeeding host i_{n+1}, (c) the message authentication code computed at the preceding host μ_{n-1}, and (d) a chained hash of a nonce γ_n. A message authentication code μ_n is secured by applying a hash function to the relation. Subsequently, the hash of the message authentication code is digitally signed by host i_n so that the initiator would authenticate host i_n as the provider of the data. At initialization the γ_n is set to $\gamma_0 = h(r)$, where r is a random nonce chosen by the initiator i_0. The chain of the execution results M_n at host i_n is the concatenation of the execution results m_j at the visited hosts for $(1 \leq j \leq n)$ and the corresponding hosts' identities i_n.. The agent's transmission between host i_n and host i_{n+1} is represented by the tuple $(\prod, M_n, P_n, \{\gamma_n\}_{K^{+}_{i_{n+1}}}, \mu_n)$.

In the *Multi-hops protocol*, an adversary can truncate data and/ or replace data without being detected. The proposed protocol rectifies the Multi-hops protocol by introducing the followings security techniques:

1. *Creating two co-operating agents*. A migrating agent and a secondary agent, which is stationary at the initiator. The *secondary agent* securely stores the initialization data necessary for accurate verifications of the protocol upon the agent's return.
2. Carrying out *verifications on the identity of the genuine initiator*, at the early execution of the agent at visited hosts, using a cipher text signed by the initiator and contained within the migrating agent. The cipher securely stores the identity of the genuine initiator.
3. *Jumbling the gathered offers* to mislead an adversary trying to truncate the offer of the preceding host.
4. Requesting a visited host to clear its own memory from any data acquired as a result of executing the agent, before the host sends out the agent to the succeeding host in the agent's itinerary.

In addition, the proposed protocol modifies the cryptographic proofs of the *Multi-hops protocol* as described below.

1. Remove the static part $\prod$ and the chain of execution results M_n from the protocol description

2. The chain of encapsulated offers P_n is replaced with a *chain of publicly encrypted offers*.
3. In the *Multi-hops protocol*, the encrypted offer incorporates the execution results at the previously visited hosts. The proposed protocol instead incorporates the cipher that securely stores the identity of the genuine initiator. The offer a host provides incorporates: (a) the data a host provides, (b) the cipher that securely stores the identity of the genuine initiator, (c) the identity of the succeeding host, and (d) a hash chain.
4. In the *Multi-hops protocol*, each executing host computes a hash of the offer it provides and then signs the computed hash with its private key, whereas each executing host in the proposed protocol signs the offer with its private key and then encrypts it with the public key of the initiator.
5. The message authentication code μ_n, which is computed at the most recently visited host and incorporates the execution results at visited hosts, is replaced with a *data integrity code*. The data integrity code is computed as a hash of the data a host provides, and the data integrity code computed at the preceding host.
6. In the *Multi-hops protocol* the most recently computed hash chain, which stands alone and is not within an offer, is encrypted with the public key of the succeeding host in the agent's itinerary. In the proposed protocol, the accumulative execution results are encrypted with the public key of the succeeding host in the agent's itinerary.
7. In the *Multi-hops protocol* the agent's transmission is represented by a message that incorporates: the static part; a chain of execution results; a chain of encapsulated offers; the most recent hash chain and encrypted with the public key of the succeeding host; the most recent hash chain. In the proposed protocol, the agent's transmission is represented by an encrypted message that incorporates: the chain of the publicly encrypted offers; the most recent data integrity code; a cipher that securely stores the identity of the genuine initiator; the most recent hash chain. The message is encrypted with the public key of the succeeding host in the agent's itinerary.

5.5 Informal description of the proposed protocol

- *Initialization*

The initiator i_0 completes the followings in the sequence given below:

1. Creates *two co-operating agents $\mathcal{A}$ and $\mathcal{A}s$.* The agent $\mathcal{A}$ is a migrating agent, while the agent $\mathcal{A}s$ is stationary at host i_0.
2. Picks a nonce *r* randomly and assigns it to γ_0,
3. Chooses the first host in the agent's itinerary i_1.

4. Generates a dummy data m_0 and computes λ_0 by encrypting m_0 with its public key, so having $\lambda_0 = \{m_0\}_{K_{i_0}^+}$.
5. Computes δ_0 as a hash of i_0, and then signs it with its own private key. The term δ_0 *securely stores the identity of the genuine initiator*, which would be used for the verifications at the early execution of the agent at visited hosts.
6. Encrypts the tuple $(\lambda_0, \delta_0, \{\delta_0\}_{S_{i_0}^{-1}}, \gamma_0)$ with the public key of the succeeding host in the agent's itinerary $_{K_{i_1}^+}$.
7. Securely stores the tuple (m_0, i_1, r) within the secondary agent $\mathcal{A}$s, which is needed for an accurate verifications upon the agent's return. Commonly, initiators carry out verifications upon agent's return based on the initial verification data, which are stored within the migrating agent. Nevertheless, an adversary might tamper with the initial verification data as the agent transfers through public channels, and thus the verifications carried out upon the agent's return would not be truly accurate. It is essential to ensure that the initial verification data are in a secure store independent of the store of the migrating agent that is susceptible to tampering. We propose to store the data securely with a stationary agent that co-operates with the migrating agent. The storing of the initial verification data within the secondary agent ensures that the data remain intact. It might be assumed that it is enough to store the terms securely within the initiator's memory, though an adversary might tamper with the memory of initiator. Hence, the verifications would not be accurate. The use of a secondary agent to store the verification terms would enable the initiator to trace any manipulation with the terms by the use of execution traces recommended by Vigna in [51].
8. Dispatches the agent $\mathcal{A}$ with the encrypted tuple to the first host in the agent's itinerary.

- *At an executing hosts*

When the *agent $\mathcal{A}$ arrives at host* i_j, the followings are completed in the sequence given below:

1. The host decrypts the ciphered message using its own decryption key.
2. The host deduces the identity of the signer of the term $\{\delta_0\}_{S_{i_0}^{-1}}$, and then decrypts the signed term. Next, the agent computes a hash of the deduced identity, and then verifies that the computed hash matches the decrypted term. If the verification fails the execution of the agent terminates, otherwise the execution continues. The verification is intended to *detect if an adversary is attempting to impersonate the genuine initiator*. If the verification passes, then the deduced identity is truly the identity of the genuine initiator and it would be used to encrypt the data host i_j provides to the agent.
3. The host provides the agent with data m_j.

4. The agent chooses the next host to visit.
5. The agent computes the hash chain γ_j as a hash of the received hash chain γ_{j-1} that was computed at the proceeding host i_{j-1}. The hash chain γ_j indicates the number of the actually visited hosts.
6. The agent generates an offer that incorporates:

   - Data host i_j provided m_j to the agent
   - Initial data integrity code δ_0
   - Identity of the succeeding host in the agent's itinerary i_{j+1}
   - Hash chain γ_j

7. The offer is the tuple (m_j, δ_0 , i_{j+1}, γ_j) signed by host i_j, and then encrypted with the public key of the host that signed the initial data integrity code δ_0 and its identity is deduced, so having the offer λ_j .
8. The agent jumbles the collected offers $\{ \lambda_0, \ldots, \lambda_j \}$ in such a way that the most recent offer is the first offer in the chain of offers, and the dummy offer is the last offer in the chain of offers, i.e. offers are arranged in a reverse order within the chain of offers. Hence, the chain would be stored as $\{ \lambda_j, \lambda_{j-1}, \ldots, \lambda_1, \lambda_0 \}$. The *jumbling is intended to mislead an adversary trying to truncate the data collected at a preceding host*.
9. The agent $\mathcal{A}$ computes the data integrity code δ_j as a hash of the data m_j host i_j provided to the agent, and the received data integrity code δ_{j-1} that was computed at the preceding host i_{j-1}.
10. The agent encloses the cumulative results in a tuple ($\lambda_j, \ldots, \lambda_0, \delta_j, \{ \delta_0 \} S_{i_0}^{-1}, \gamma_j$) and encrypts it with the public key of the succeeding host in the agent's itinerary.
11. The agent *requests the current host* i_j *to clear its memory from any data acquired as a result of executing the agent*.
12. The host dispatches the encrypted tuple to the succeeding host i_{j+1} in the agent's itinerary.

- *Termination*

The agent returns to the initiator with the following message:

$$\{ \lambda'_n, \ldots, \lambda'_j, \ldots, \lambda'_0, \delta'_n, \{ \delta_0 \} S_{i_0'}^{-1}, \gamma'_n \} K_{i_0}^{+}$$

Upon the agent's return, the initiator completes the followings in the sequence given below.

1. Decrypts the message using its own decryption key
2. Decrypts the offer λ'_j ($0 \le j \le n$) using its own decryption key.

3. Deduces the identity of the signer of the offer λ'_j, and then decrypts the offer using the signature verification key of the signer. The decrypted offer would be a tuple of the form of (m'_j , δ'_0 , i'_{j+1} , γ'_j). If the decryption succeeds, it then authenticates that the data m_j is provided by host i_j. Next, it constructs the set of data provided by the visited hosts $\{ m'_n , \ldots , m'_0 \}$.
4. Checks that m'_0 matches m_0, which is securely stored with the secondary agent.
5. Counts the number of elements in the set $\{ m'_n , \ldots , m'_0 \}$ excluding m'_0, which is supposed to be the dummy offer generated by the initiator, e.g. x. The count x is supposed to be equal to the *actual number of visited hosts*, though the count x might differ from the actual number of visited hosts due to the data-truncation attack by adversaries. Next, it computes the hash chain γ'_n based on the following equation:

$$\gamma'_n = h^x(r)$$

, where x is the *count of the returned offers*

r is the nonce stored within the secondary agent

The γ'_n is computed by hashing r as many times as x. Next, it checks that the computed hash chain γ'_n matches the returned γ_n.

6. Computes a data integrity δ'_n based on the following equations:

$$\delta'_0 = h(i_0)$$
$$\delta'_n = h(m'_n , \delta'_{n-1})$$

and using the set $\{ m'_n , \ldots , m'_0 \}$. Next, it checks that δ'_n matches the returned data integrity code δ_n. If they do not match, then it implies that the data acquired at the visited host were truncated or was illegitimately inserted into the execution results of the agent.

7. Assembles the actual agent's itinerary from the received chain of offers $\{\lambda'_n , \ldots , \lambda'_j , \ldots , \lambda'_1\}$, where each offer λ'_j for $(1 \leq j \leq n)$ indicates the identity of the executing host i_j, and incorporates the identity of the succeeding host i_{j+1}. Next, it verifies that the first host in the assembled agent's itinerary matches the identity of the first host i_1, which is stored within the secondary agent. Next, it checks that the partial agent's itineraries are consistent.
8. Verifies that the received offer λ'_j for $(1 \leq j \leq n)$ was generated for the genuine initiator i_0 of the agent, by comparing the term δ'_0, which is enclosed within the received offer with a hash of the identity of the initiator $h(i_0)$.
9. Verifies that the received offer λ'_j for $(1 \leq j \leq n)$ was generated for the protocol run of concern, which is identified by the random nonce r. It computes the hash chain γ'_j for each received offer λ_j for $(1 \leq j \leq n)$, by hashing the nonce r that is stored within the secondary agent as many times as the order of the visited host within the

assembled agent's itinerary, e.g. if the host is the second visited host then $\gamma'_j = h^2(r)$. Next, it verifies that the computed chained hash γ'_j matches the returned hash chain γ_j, which is enclosed within the respective offer λ'_j for $(1 \leq j \leq n)$.

If the whole set of verifications passes, then the data returned within the agent $\mathcal{A}$ are truly authenticated, intact, confidential and belong to the protocol run of concern. Adversaries were not able to truncate, delete, or alter any of the gathered offers. Also, adversaries were not able to illegitimately insert data into the gathered offers. The privacy of the gathered data could not be breached and the data could be accurately authenticated, since each visited host uses the public key of the verified genuine initiator to encrypt the offer it signs and provides to the agent. Hence, we can be certain that the execution results have not been tampered with and would be accepted with confidence.

5.6 Sequence of processes of the proposed protocol

The sequence of processes at the initiator is depicted in figure 4, whereas the sequence of processes at an executing agent in the agent's itinerary is depicted in Figure 5.

5.7 The proposed protocol vs. the Multi-hops protocol

The protocol is *derived from the Multi-hops protocol* [14] with some enhancements and amendments which are intended to hinder the type of flaws revealed in the existing protocols [14, 20, 22, 23, 28, 29] as follows:

1. The offers $\lambda_0 , \ldots , \lambda_n$, which are provided by the visited hosts, are jumbled within the chain of offers λ to mislead an adversary trying to delete the recently gathered offer/s. Hence, deleting the last offer from the chain of the gathered offers does not mean that it deleted the most recently gathered offer (the offer acquired at the preceding host). Also, the chain of offers is initiated with a dummy offer λ_0 that has to be positioned as the last offer in the chain of the gathered offers each time offers are jumbled, so having the chain of offers arranged in a reverse order as $\{\lambda_n , \lambda_{n-1} , \ldots , \lambda_1 , \lambda_0\}$. Hence, it would appear as if it is the offer provided by the most recently visited host.

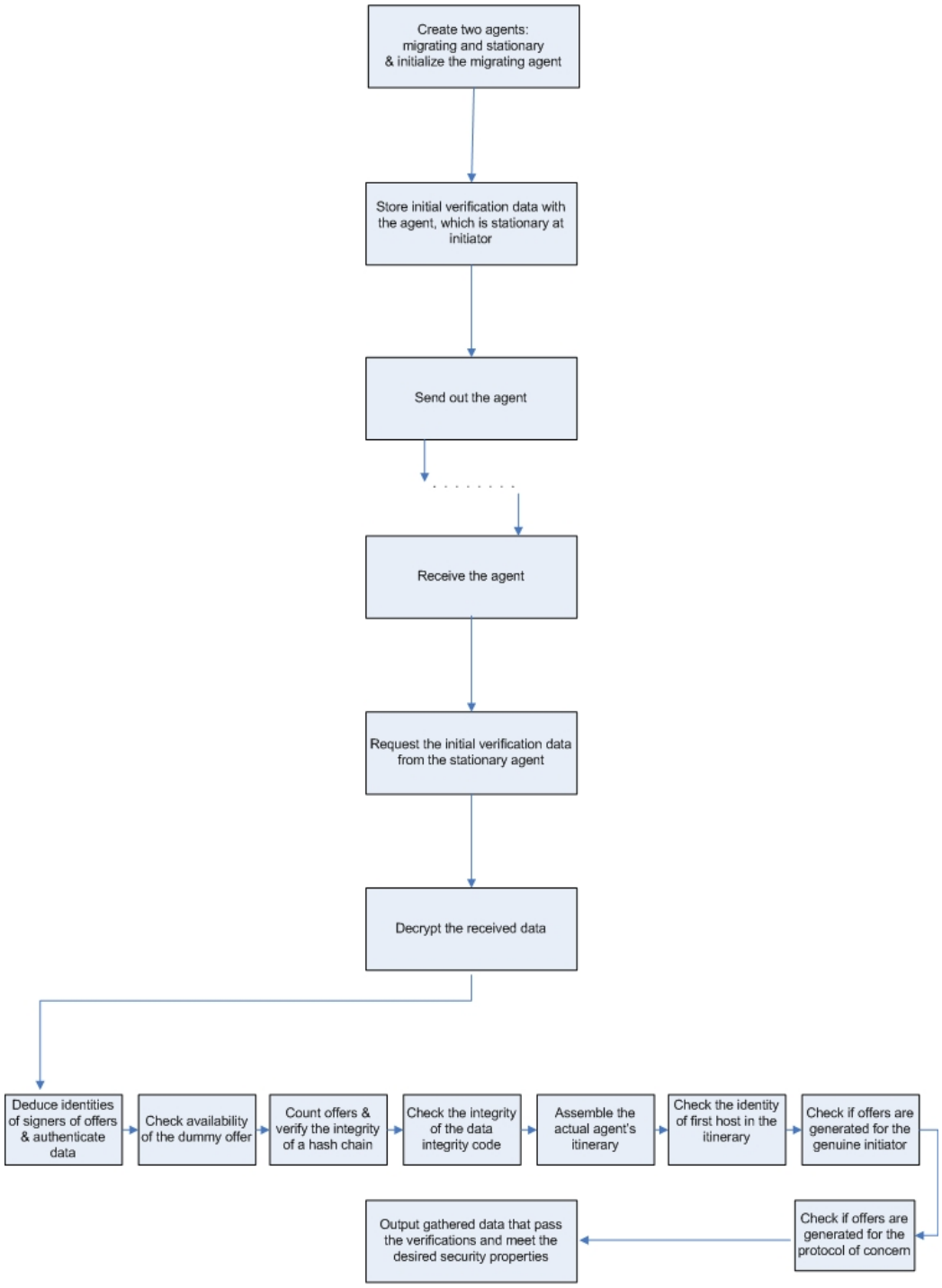

Fig. 4 The sequence of processes at the initiator

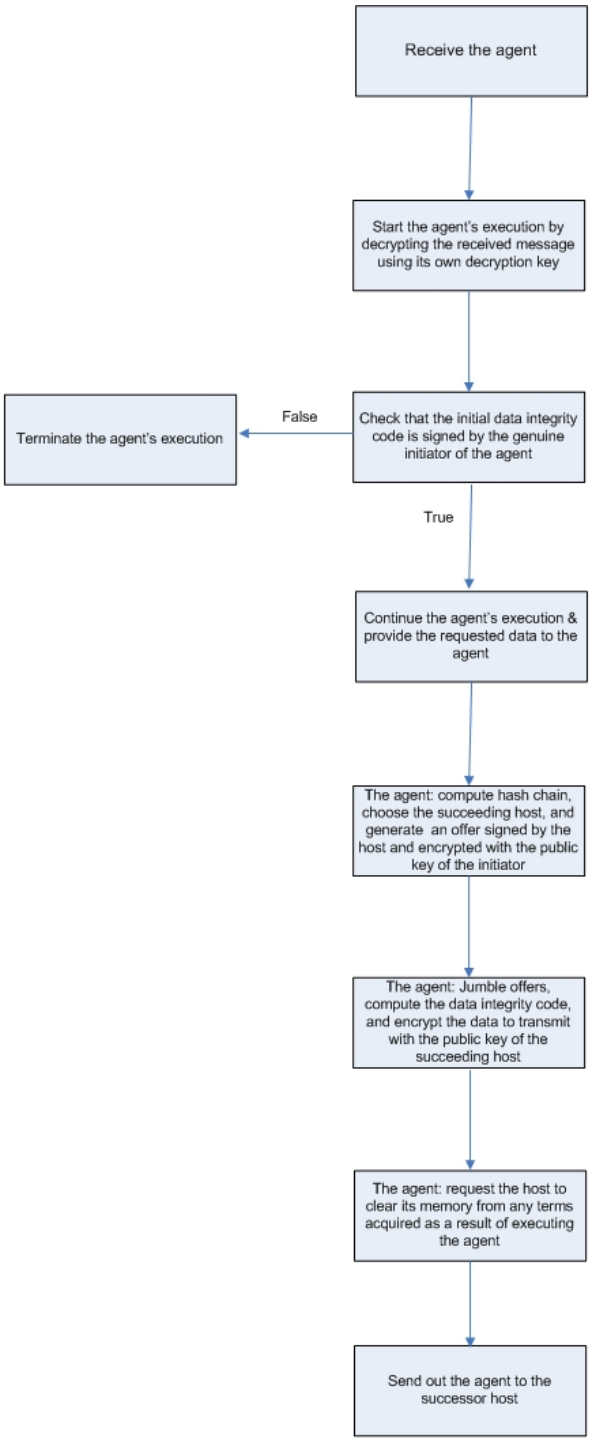

Fig. 5 The sequence of processes at an executing host

2. The term $\{h(i_0)\}_{S_{i_0}^{-1}}$ is appended to the protocol to enable the verification on the identity of the genuine initiator of the agent during the life cycle of the agent. At the early execution of the agent at a visited host the agent deduces the identity of the signer of the transmitted term, and then decrypts the signed term with the signature verification key of the signer. Next, it hashes the deduced identity and verifies that the result is the same as the decrypted term. If the verification passes, then it uses the public encryption key of the signer to encrypt the offer the host provides. Otherwise, the execution of the agent terminates. The verification can detect if a malicious host is impersonating the genuine initiator, with the intension to breach the privacy of the data to be gathered from subsequent hosts at some time later. If the malicious act is detected, the execution of the agent terminates. Hence, the protocol can detect the malicious act of impersonating the genuine initiator and the breach of privacy of the gathered data is improbable. The *Multi-hops protocol*, does not aim to preserve confidentiality. To preserve the confidentiality of gathered offers, it is necessary to encrypt offers with the public key of the genuine initiator. The protocol does not indicate the identity of the initiator for which offers should be encrypted. An adversary might mask it-self as the genuine initiator of the agent. Hence, executing hosts would encrypt the offers they provide with the public key of the adversary. The adversary would be able to breach the privacy of the encrypted offers and then encrypts the offers with the public key of the genuine initiator. The malicious act would not be detected by the initiator.

3. The offer which a visited host provides is digitally signed with its private key, and then it is encrypted with the public key of the initiator. Therefore, the offer would be confidential, and correctly authenticated. Also, the identity of the providing host could not be deduced. In the *Multi-hops protocol*, the offer is sent in two forms: as a plain text and as a hash of the provided offer signed by the private of the executing host. The encryption scheme enables an intruder to infer the identity of the signer of an offer, thus the deletion of offer/s of competitive host/s would be straight forward and accurate.

4. The execution results of the agent should be encrypted with the public key of the succeeding host in the agent's itinerary, before the results are transmitted, while in the *Multi-hops protocol* it is just the hash chain γ_n that should be encrypted. The proposed encryption would prevent an intruder from learning or tampering with any of the execution results it may intercept.

5. The data integrity code δ_j replaces the message authentication code μ_n. The data integrity code δ_j is a hash chain of the data acquired at the visited hosts and is based on the identity of the genuine initiator of the agent, whereas the message authentication code μ_n binds the previously gathered data and the identity of the succeeding host to the data the current host provides. An adversary might replace the agent's data with data that were generated for another initiator. The proposed protocol would detect the attack, whereas the *Multi-hops protocol* would not detect the attack.

The proposed protocol can detect the attack as follows:

Upon the agent's return, the initiator decrypts each of the gathered offers in the chain $\{\lambda'_n , \lambda'_{n-1} , \dots , \lambda'_1 , \lambda'_0\}$ using its private key. Next, it deduces the identity of the signer of the offer λ'_j for $(1 \leq j \leq n)$, and then decrypts the offer with the signature verification key of the respective signer. Next, it computes the data integrity code δ'_n based on the identity of the genuine initiator of the agent i_0, and then checks if the computed data integrity code δ'_n matches the data integrity code returned with the agent δ_n. If the verification passes, it implies that the returned data were generated for the genuine initiator i_0 and the data are intact. But, if the verification fails it implies that tampering with the gathered data took place.

The *Multi-hops protocol* would not detect the attack, since the message authentication code is not based on the identity of the genuine initiator.

6. In the *multi-hops protocol* the static part $\prod$ is coupled with the dynamic part of the agent to be able to verify that the returned data belong to the agent of concern. Though, the verification may not be accurate, e.g. the returned data might be generated for a different initiator. In our protocol we omitted the coupling of the static code $\prod$ with the dynamic code and made use of the co-operating agent $\mathcal{As}$, which securely stores: (a) the *random nonce r* that uniquely identifies the agent and the protocol run, (b) the *first scheduled host* i_1 in the agent's itinerary, and (c) the dummy data generated by the initiator m_0. Upon the agent's return, the terms would be called from the secondary agent to verify accurately that the returned data belong to the agent of concern and to the particular protocol run.

7. The chain M_n is omitted so as to ensure the confidentiality of the transmitted offers. In the *Multi-hops protocol*, the chain consists of the data that each visited host provides and the identity of the respective host in plain text. The transmission of plain text violates the confidentiality of data. The proposed protocol incorporates the data that the host provides m_n in the encrypted offer λ_n.

8. The data integrity code δ_0 is enclosed within the offer λ_j for $(1 \leq j \leq n)$ to indicate the identity of the initiator for which the offer is generated. If the data integrity code δ'_0 that is enclosed within the received offer λ'_j for $(1 \leq j \leq n)$ does not match $h(i_0)$, then it implies that an adversary has impersonated the genuine initiator.

9. In order to prevent the data-truncation and data-alteration attack, we propose the agent to be programmed in such a way that it instructs each visited host i_k to clear its memory from any data acquired as a result of executing the agent before the host dispatches the agent to the next host in the agent's itinerary. In the data truncation and data alteration attacks, host i_k would co-operate with host i_j to delete the offers m_x for $(k < x < j)$ and

replace its previously provided offer λ_k with a new offer λ'_k. When host i_j receives the agent, it sends the agent back to host i_k, which has already stored the agent's state that was present when the agent firstly visited it. Host i_k would then replace the current agent's state with the state that was present when the agent firstly visited it and append an offer λ'_k to the agent's state.

5.8 Security scheme of the proposed protocol

Our attempt to accomplish the security properties is described below:

• *Data-authenticity*: the data that each host provides is signed digitally with the private key of the host. Hence, a host cannot forge the data it signed with its private key. However, a signed message does not truly indicate the genuine signer. An adversary might intercept a signed message, and can then strip off the signature from the signed message. Next, the adversary signs the message with its private key. Hence, the recipient of the signed message is deceived on the identity of the genuine signer. In order to ensure accurate authenticity in mobile agents, the digital signing of the data that a host provides should be followed by the encryption of the signed message with the public key of the initiator i_0. An adversary might be able to intercept the transmitted data, however it would not able to decrypt the data, or impersonate the genuine sender, since the data is encrypted and the needed decryption key is private to the initiator i_0. Consequently, the proposed encryption scheme ensures accurate authentication.

• *Data-confidentiality*: the data that hosts provide are encrypted with the initiator's public key $\kappa^{+}_{i_0}$. Hence, an adversary would not be able to decrypt the data since it needs the decryption key $\kappa^{-}_{i_0}$ which is private to the initiator i_0.

• *Data-integrity*: the property is the commonly breached property and the most difficult to accomplish. It requires a very careful and thorough analysis, especially the capabilities of the adversary who would intrude the data that mobile agents acquire in different ways, as follows:

1. Delete the data acquired at the visited hosts in the agent's itinerary.
2. Illegitimately insert or append data to the agent's execution results.
3. Truncate the acquired data.
4. Replace the agent's dynamic data with data of a similar agent or a different protocol run.

We set the protocol to include specific terms and to use certain encryption keys, which would ensure the robustness of the protocol to the four kinds of malicious acts of adversaries. The terms and encryption keys that the protocol implements are described below:

- *Data integrity code* δ_j: It is used to verify if any of the data acquired at visited hosts was deleted. The data integrity code δ_j at a host i_j for $(1 \leq j \leq n)$ is a hash of the data the host provides m_j and the data integrity code that was computed at the preceding host δ_{j-1}, as described below:

$$\delta_j = h(m_j , \delta_{j-1})$$

  The initial parameter δ_0 is set to $h(i_0)$.

- *Encryption key of the succeeding host in the agent's itinerary* $K^{+}_{i_{j+1}}$: Following to the execution of the agent at host i_j, the agent encrypts the execution results with the public key of the succeeding host i_{j+1} in the agent's itinerary. Next, the host dispatches the agent to host i_{j+1}. An intruder might intercept the communication between host i_j and host i_{j+1}, and then would acquire the encrypted execution results of the agent. However, it would not be able to learn the plain execution results. It has to have the decryption keys of the initiator $K^{+}_{i_0}$ and of scheduled succeeding host $K^{+}_{i_{j+1}}$, which are private to the hosts i_0 and i_{j+1} respectively. Moreover, the intruder would not be able to append any valid offer to the chain of offers. It has to have the terms: δ_0, δ_j, and γ_j that are necessary to compute a valid offer, though the terms are encrypted with the public key of host i_{j+1}. Thus, the encryption scheme reduces the chances of an intruder to tamper with the execution results and prevents the illegitimate insertion/ appending of offers to the chain of offers.

- *Offer* λ_j: The data that host i_j provides is firstly signed with its private key $S^{-1}_{i_j}$ and then it is encrypted with the public key of the initiator $K^{+}_{i_0}$. Thus, the data contained within the offer is confidential and would be correctly authenticated.

- *Terms* δ_j, $\{\delta_0\} S^{-1}_{i_0}$, γ_j: When host i_{j+1} receives the agent from host i_j it deduces the identity of the signer from the term $\{\delta_0\}S^{-1}_{i_0}$, and then decrypts the term with the signature verification key of the signer. Next, it computes a hash of the deduced identity, and then checks that the computed hash matches the decrypted term. If the check passes, then it uses the deduced identity to encrypt the offer to the host i_{j+1} provides to the agent. The terms δ_j and γ_j are necessary to compute the data integrity code δ_{j+1}, and the hash chain hash γ_{j+1} respective to the current host.

- *Chain of offers* $\{\lambda_n , \lambda_{n-1} , \ldots , \lambda_1 , \lambda_0\}$: The offers that are provided by the visited hosts i_j for $(1 \leq j \leq n)$ are jumbled to deceive any adversary trying to truncate the data acquired at the preceding host. They are arranged in a reverse order. Assume the offers are jumbled such as λ'_4, λ'_3, λ'_2, λ'_1, λ'_0 and the adversary deleted the last two offers, so the chain of offers reduces to λ_4, λ_3, λ_2. Upon the agent's return, the initiator decrypts each encrypted offer, and then learns the data m'_2, m'_3, and m'_4, which were provided by the hosts i_2, i_3, and i_4. Next, it

computes the data integrity code δ'_n using m'_2, m'_3, m'_4 and using the following equations

$$\delta'_0 = h(i_0)$$
$$\delta'_n = h(m'_n , \delta'_{n-1}) \quad , \text{where } (2 \leq n \leq 4)$$

It then checks that the computed data integrity code δ'_n matches the data integrity code enclosed within the agent δ_n. The verification would show that the two data integrity codes do not match. Hence, it implies that an adversary has truncated offer/s from the chain of offers.

- *Offer* λ_j : the offer of host i_j incorporates the tuple (m_j, δ_0, i_{j+1} , γ_j), where m_j is the data provided by host i_j. The data integrity code δ_0 is enclosed within the offer λ_j for ($1 \leq j \leq n$) to indicate the identity of the initiator for which the offer is generated. If the data integrity code δ_0 that is enclosed within the received offer λ_j for ($1 \leq j \leq n$) does not match $h(i_0)$, then it implies that the returned data were not generated for the genuine initiator. The identity of the successor host i_{j+1} is included to record the agent's partial itinerary. The chained hash γ_j is included to indicate the actual number of visited hosts. The parameter γ_j is initially set to a random nonce *r*. The random nonce *r* is expected to be hashed as many times as the number of visited hosts.

- *The data integrity code* δ_j is only encrypted with the public key of the succeeding host i_{j+1}. So adversaries would not be able to learn the terms: δ_0, δ_j, and γ_j that are necessary to compute a valid offer. Only the intended succeeding host i_{j+1} would be able to compute a valid offer.

Table 3 summarizes the security scheme of the proposed protocol with respect to the aimed for security properties, and in the view of the threats of adversaries and the flaws revealed in the existing security protocols.

Table 3 The security scheme of the proposed protocol

Security properties	**Adversaries' attempts to violate security**	**Security scheme**	**Threat status**
Confidentiality	Breach the privacy of the collected offers	A host signs its offer digitally, and then encrypts it with the encryption key of the initiator. Next, it encrypts the agent's execution results with the public encryption key of the succeeding host in the agent's itinerary.	prevented

Table 3 The security scheme of the proposed protocol

Security properties	**Adversaries' attempts to violate security**	**Security scheme**	**Threat status**
Confidentiality	Impersonate the genuine initiator [1]	At the early execution of the agent at a visited host, the host deduces the identity of the signer of $(\delta_0)S_{i_0}^{-1}$ and then it decrypts the term using the decryption verification key of the signer. Next, it computes a hash of the deduced identity and then it verifies that the computed hash is the same as decrypted term, otherwise the agent execution terminates.	Detected
Authenticity	Impersonate the genuine provider of an offer [1]	The offer λ_j is digitally signed with the private key of host i_j, and then encrypted with the public key of the initiator. Hence, an adversary would not be able to read the offer or send the offer under its private key.	Prevented
	Append arbitrary or fake offer [1]	A host should sign the offer it provides with its private key. Hence, an adversary would not append an arbitrary offer for which it is held responsible and can not repudiate it.	Prevented
Integrity	Replace the collected data with data of a similar protocol run	Upon the agent's return, the initiator should do the followings in sequence: – Computes a count (λ) and assigns it to a variable x. – Computes a hash of the random nonce r stored within the secondary agent, i.e. $h^w(r)$, where $w = x - 1$. – Checks that the computed hash chain matches the hash chain γ'_n returned with the agent. The check would fail if the attack took place. Hence, the protocol detects the attack.	Detected

Table 3 The security scheme of the proposed protocol

Security properties	**Adversaries' attempts to violate security**	**Security scheme**	**Threat status**
Integrity	Replace the collected data with data of a similar agent	The terms r, m_0, and i_1 returned within the agent $\mathcal{A}$ are verified with the corresponding terms stored within the secondary agent $\mathcal{A}s$. The verification would fail if the attack took place.	Detected
	Truncate data:		
	- Try to delete the offer of the preceding host	Collected offers are jumbled so the dummy offer is arranged at the end of the collected offers to conceive an adversary trying to delete the offer of the preceding host	Not possibly accurate
	- Delete the dummy offer λ_0	Upon the agent's return, the initiator calls the dummy data m_0 which is securely stored within the secondary agent. Next, it checks the availability of the same data within the data returned with the agent. The unavailability of the data implies that the attack took place.	Detected
	- Delete the offer λ_1	Upon the agent's return, the initiator checks that the first host in the assembled agent's itinerary is host i_1. If the check fails, it implies that the attack took place.	Detected
	Delete the offers λ_1 and λ_0	Upon the agent's return, the initiator does the followings: – Calls the dummy data m_0 which is securely stored within the secondary agent. Next, it checks the availability of the same data within the data returned with the agent. The unavailability of the data implies that the attack took place. – Checks that the first host in the assembled agent's itinerary is host i_1. If the checks fail, it implies that the attack took place	Detected

Table 3 The security scheme of the proposed protocol

Security properties	**Adversaries' attempts to violate security**	**Security scheme**	**Threat status**
Integrity	Delete the offer λ_2	Upon the agent's return, the initiator decrypts the offer λ_1 and learns the identity of the succeeding host to host i_1 as given in the offer. Next, it checks that the succeeding host exists in the assembled agent's itinerary. If the check fails, it implies that the attack took place.	Detected
	Delete the offer λ_K	Upon the agent's return, the initiator should do the followings in sequence: - Computes a hash of the random nonce r stored within the secondary agent, i.e. $h^w(r)$, where $w = x - 1$. Next, it checks that the computed hash matches the hash chain γ'_n returned with the agent. The check would fail and it implies the attack took place. - Deduces the partial agent's itineraries, where each signed offer λ_j includes the identity of the succeeding host. Next, it Checks that the partial agent's itineraries are consistent. The assembled agent's itinerary would indicate a missing connection and it implies that the attack took place. - Checks that the computed data integrity code matches the returned data integrity code. The data integrity code is a hash chain of the data acquired at the visited host. The check would fail and it implies that the attack took place.	Detected

Table 3 The security scheme of the proposed protocol

Security properties	**Adversaries' attempts to violate security**	**Security scheme**	**Threat status**
Integrity	Upon the second visit of the agent to the host i_k, it replaces its previous offer λ_k with a new offer λ'_k so as to substitute m'_k for m_k	Terms that are necessary to replace a previous offer with a new valid offer, are assumed to be cleared from the memory of the host during the first visit of the agent to the host and just before the host dispatches the agent to the succeeding host. Hence, the attack is not possible. However, a malicious host may not clear its memory from the terms then the next check would detect the attack if it took place. Checks that the computed data integrity code matches the returned data integrity code. The data integrity code is a hash chain of the data acquired at the visited host. The check would fail and it implies that the attack took place.	Prevented/ Detected
	Illegitimately insert or append offer/s [2]	The intruder needs to learn the terms that are necessary to compute a new offer. But, the terms are encrypted with the public key of scheduled succeeding host in the agent's itinerary. Hence, the intruder would not be able to insert or append offer/s.	Prevented

[1] The act of a malicious participating host
[2] The act of an intruder

6 Related Formal Verification Methods

Formal methods have played an important role in specifying, modeling, verifying and revealing unforeseen flaws in the existing security protocols. They have the following capabilities [26]:

- Characterize the system specifications precisely.
- Define accurately the desired security properties.
- Set clearly the interaction between the system and its environment.
- Identify security flaw/s in protocols if any exists.
- Provide systematic and exhaustive analysis of protocols.
- Provide a proof of security, if a system meets the desired properties.
- Provide verification tools at design stages as well as analysis stages. Applying formal methods at the design stage would save the expense of redesign of an existing flawed protocol.

The existing formal methods can be classified into five categories [26] as follows:

- *Methods based on modal logic*
- *Methods based on finite-state exploration*
- *Methods based on theorem proving*
- *Methods based on modal algebra*
- *Methods based on infinite-state exploration*

- *Methods based on modal logic* require translating a protocol into a set of logic statements about the initial beliefs or knowledge in a distributed system. The verification of a protocol is a deductive reasoning process, where inference rules are used to derive new beliefs from the initial beliefs and/ or new knowledge from the initial knowledge. If the derived beliefs are equivalent to the required beliefs, then the protocol is considered correct by proofs. The disadvantages of modal logic are: (a) the verification is usually done manually, and (b) the verification is error-prone and non-systematic. The best known and most influential logic is BAN logic [11] (logic of authentication). BAN logic does not attempt to model either trust or knowledge. Therefore, BAN logic can not be used to prove results about secrecy. It can only be used to reason about authentication. BAN logic found flaws in the Needham-Schroeder public key and the Kerberos protocols.

- *Methods based on finite-state exploration* model the honest hosts that participate in a protocol and an intruder as communicating processes and analyze the system under the Dolev-Yao intruder model [55]. The intruder may store, hide, replace, or replay messages transmitted over the Internet. Moreover, the intruder can generate new messages by decrypting, encrypting, faking intercepted messages. They model a certain protocol as a finite-state system and verify by exhaustive search that all reachable states satisfy some properties. Properties are stated in some logic (usually temporal logic). The advantages of the finite-state exploration methods are: (i) Properties can be verified automatically. (ii) Little user intervention is required. (iii) If the protocol fails, it is able to generate the sequence of events that invalidates the protocol. On the other hand, the disadvantages are: (i) It is usually applied to systems of finite number of states, hence, if no attack is found

there still may be an attack on the real system with large number of states. (ii) The method becomes intractable for large state systems due to state explosion problem. In order to keep the model finite, it is necessary to place a bound on the number of protocol runs and a bound on the number of messages the intruder can generate. The Methods include: Interrogator [32], NRL protocol Analyzer [31], CSP model checker FDR [24, 25], SPIN model checker [27], Murφ [35]. The NRL has been used to find several previously undiscovered security flaws in cryptographic protocols.

- *Methods based on theorem proving* use logical notations to model a protocol and specify its properties, and use logic theories to verify if the protocol satisfies the properties. The advantage of the method is that it can be used to verify real protocols with large number of states. Whereas, the disadvantage is that it requires expert guidance. The Induction method [38] uses the Isabelle Theorem Prover to verify the TLS Internet protocol and the Kerberos protocol.

- *Methods based on modal algebra* use algebra to express a protocol as a set of concurrent communicating processes. The protocol is modeled as a set of hosts which send messages to each other and an environment modeling malicious hosts or intruders who can perform any sort of attack. Security properties can be expressed via the notion of equivalence (e.g. may-testing) between two parallel processes. The disadvantage of modal algebra is that equivalences suffer from universal quantification over attackers, which makes equivalence checking between processes very hard as having infinitely many such processes. Modal algebra methods include: π-Calculus [1], applied-π Calculus [2], Spi-Calculus [3], Distributed-π calculus [44], seal calculus [52], and Crypto-loc Calculus [5].

- *Methods based on infinite-state exploration* are based on a variety of symbolic techniques [8]. The intruder is not modeled explicitly and it makes no assumptions on the type and number of messages it can generate. These methods are promising in two aspects: (i) They can accomplish a complete exploration of the state space based on symbolic runs of processes. (ii) They do not suffer from any state explosion problem that is usually induced by the exchange of messages, since every input action gives rise exactly to one symbolic transition and properties are formalized as correspondence assertions, where each execution of an action must be preceded by the execution of a corresponding action [6, 7], thus improving the run time and the accuracy of verification. The STA [46] is an infinite-state exploration method based on symbolic execution.

The formal methods have been successfully employed in the verification of the security properties of the classical message-based protocols, such as authentication protocols, though the specification and verification of the security of mobile agent paradigms deal with new aspects: locations and mobility on the top of cryptography. There have been good advances in expressing mobility using process algebra [3, 6, 19, 33, 43, 52]. Some experiments used existing formal methods to verify data protection protocols of mobile agents. The data integrity properties of mobile agents are verified

using CSP-based tools Casper [25] and FDR [18] in [20], and a model checker that is based on symbolic data representation and uses Spi-calculus in [28]. In this paper, we utilize the STA formal method to analyze the security properties of the proposed protocol. The STA describes a protocol using process algebra, and specifies and verifies the system using symbolic techniques, which explore the whole state space.

7 STA Automatic Verifier

The STA (Symbolic Trace Analyzer) [7, 45, 46] is a tool for the analysis of security protocols. It is a recent approach that takes advantage of concepts derived from process calculi. It is based on symbolic trace analysis that performs a complete exploration of the whole infinite-state model. It detects the flaws in Needham-Schroeder, Yahalom, Otway-Rees, and Kerberos protocols [6, 46].

We use the STA formal method in the analysis of the proposed protocol as it is characterized by the followings:

- *It does not suffer from state explosion problem* [9, 16]. A protocol in STA is modeled as a set of processes and properties are verified by considering the computation traces of processes. Generally, the set of computation traces are infinite. STA implements symbolic techniques that reduce infinite transition to a single symbolic relation, where every input action should be preceded by a corresponding output action. STA can analyze the whole infinite state space generated by a finite set of participants. According to the authors in [8, 9, 16] the symbolic analysis is sound and complete. Detecting an attack on the symbolic model would imply that an attack exists in the infinite standard model and detecting no attack on the symbolic model would imply that the security property is satisfied. The verification on a finite set can be using symbolic techniques in two stages. The first is the symbolic reduction of processes where inputs are evaluated formally. The symbolic runs of a process are finite. The second is the symbolic procedure that uses the knowledge of the environment to construct symbolic models of processes comprising the symbolic runs. The symbolic models do not yield the state-explosion problem induced by message exchange, since every input action gives rise exactly to one symbolic transition and properties are formalized as correspondence assertions, where every execution of action α must be preceded by some execution of action β for a given α and β [6, 7]. Conversely, the finite state exploration methods analyze execution traces of systems by searching for an insecure state starting from an initial state and assume that the Internet is under the control of adversaries. The search might be infinite due to the unpredictable behavior of adversaries. An adversary might generate arbitrary number of new messages, and replay, replace, or delete an intercepted message. According to the authors in [26] the state space

of a system might increase exponentially as the size of the system grows linearly. Therefore, methods require that systems have a finite number of states by imposing the following restrictions on the model to analyze [7, 9, 16, 26]:

- Finite set of participants
- Finite set of messages a participant in a protocol would receive
- Finite set of states a participant may enter
- Finite set of steps a participant may perform in the protocol
- Finite number of messages the attacker can generate and send to participants in the protocol

If any of the restrictions is not imposed, infinite transitions would be generated. The problems with the finite model are as follows [7, 16]:

- The restrictions are not always precise.
- If no attack is detected on a small system there might be an attack on a larger system.
- Establishing proper restrictions would require familiarity of how the protocol works.
- Attackers may send messages of any size by manipulating messages available in the environment. Hence, analyzing systems of finite set of participants might sometimes result in a search of infinite traces.

- *It does not require expert guidance:* Modeling and verifying the system is simple and straight forward, whereas the theorem proving methods are time consuming and require a lot of expertise [4, 15]. In STA, modeling the system requires familiarity with process algebras. The system configuration is the parallel composition of the roles of participating hosts and the initial knowledge of an intruder, such as an intercepted message, public keys of participating hosts, etc. The security properties are expressed in terms of correspondence assertions. In theorem proving, modeling the system and specifying properties require acquaintance with logic theories. Protocols are modeled as a set of all possible traces encoded in logic and properties are verified by induction on the traces. The verification can be long and require human guidance to develop lemmas and theorems as needed [4].

- *It does not need to model the intruder explicitly*: The intruder is represented by the environment's initial knowledge, e.g. an intercepted message, and the public keys, the signature verification keys, and the identities of the participating hosts, whereas the model checking methods assume that the intruder is a participant in the system, and capable of initiating communication with a participating host. It requires modeling the intruder explicitly [6, 7, 35]. Modeling the intruder is

often relatively complicated and time consuming, due to the unlimited capabilities and unpredictable behavior of the intruder.

- *The verification is automatic*: Automatic verification saves time as compared to methods which require hand-written proofs, such as the methods based on modal logic or modal algebra. Modal algebra methods require checking equivalence of processes. The checking of two processes that they are indistinguishable for any tester process is difficult, especially processes are infinite and proofs are hand-written [15].

STA is a simple and an efficient tool for the analysis of security protocols. The analysis of Needham-shroeder and Kerberos protocols with STA in [46] and with Murφ in [35] shows the advantages of using symbolic methods over finite-state exploration methods.

The STA is an ML [34] based tool, where, ML is basically a functional polymorphic programming language that allows parallel processing and is employed to develop verification tools [36, 39, 48]. The theory underlying the STA is explained in full details in [9]. The STA tool requires the Moscow ML [37], a compiler for the standard ML.

According to Boreale and Buscemi in [7], a protocol in STA is modeled as a system of concurrent processes and a state of the system is modeled as a pair ⟨s, *P*⟩ called configuration. The *s* is a trace of past I/O actions that results from interaction between a process and its environment, and represents the current intruder's knowledge. The *P* is a Spi-calculus term that describes the intended behavior of honest participants. The syntax of the STA is analogous to the syntax of Spi-calculus [3] with slight differences and is shown in Table 4. The syntax of STA is as follows: *a!M* is an output action. *a?x* is an input action, where *a* is a reference to I/O action. *M* is a message. *x* is a variable. *stop* is a terminated process. >> is a sequence of actions. *P*1 || *P*2 is a parallel composition. *K new_in* is a fresh name *K*. *(M)^+K is a* asymmetric encryption of *M* with the public key *K* (*-K* is private). *(M)^+sigK is a* digital signing of *M* with the private key *+sigK* (*-sigK* is public). *(M1, M2) is a* pairing. *(M is N)* is an equality test. In the STA, cryptographic functions are modeled as process operators.

Transitions between the configurations represent interactions between s and *P*, and take the form of <s, *P*> → <s′, *P*′>. The STA analyzes the execution traces of the system to detect possible faults of security properties of the protocol. However, searching all execution traces for an insecure state starting from an initial configuration may result in search of infinite transitions. The STA introduces a transition relation $\rightarrow_s$ that condenses infinite transitions to a single symbolic transition, where each input action should be preceded by a corresponding action.

Table 4. Syntax of the STA

A! M	Output action
A? x	Input action
Stop	Terminated process
>>	Sequence of actions
*P*1 || *P*2	Parallel composition of the processes *P1* and *P2*
+K	Public encryption key
+SigK	Digital signing key
(M is N)	Equality test of the terms *M* and *N*
(M1, M2)	Pairing of the terms *M1* and *M2*
hsh(M)	Hash of the term *M*
M pkdecr(-K, x)	Decryption of the variable *x* with the public key *–K* to get the plain message *M* (asymmetric decryption)
M pkdecr(-SigK, x)	Decryption of the variable *x* with the signature verification key *–SigK* to get the plain message *M* (asymmetric decryption)

Security properties are expressed in terms of the traces the protocol generates. In particular, properties are formularized as correspondence assertions, that is given a configuration ⟨s, *P*⟩ and a trace s′, then ($\alpha \leftarrow\!\!\!\!\sim \beta$) means that every instance of β must be preceded by the corresponding instance of α. Authentication property is expressed in terms of a trace of the kind that any message that is accepted by B at the final step should actually originate from A. Secrecy is expressed by using the "absurd" action ⊥ in the formula: $\perp \leftarrow\!\!\!\!\sim \alpha$ which means an action α should never take place. The secrecy of the protocol *P* is such that it does not reveal the sensitive data *d* in the presence of a guardian *g* that can at any time pick up one message *x* from the network. The relation: *P* || *g*(*x*).0 and the deduction relation |= can be used to express how the environment can generate new messages starting from an initial set of messages. Thus, the secrecy property can be expressed as : $\langle \varepsilon, P \parallel g(x).0 \rangle \models \perp \leftarrow\!\!\!\!\sim g\langle d \rangle$, where ε is the empty trace. Upon the verification, if no attack against the security property exists then the tool reports "No attack was found". Otherwise, it reports the attack in the form of an execution trace that violates the specified property.

In the STA, the protocol can be specified using four kinds of declarations [45]: identifiers declarations, processes declarations, configurations declarations, and properties declarations. Table 5 summarizes the STA declarations. The declarations of the four kinds are described below.

- *Identifiers* are names, variables, and labels. A name can be an encryption key, a host identity, or the data generated at a host. A variable is a received term, which might differ from the original transmitted term. Suppose a host A transmits a name, then host B might receive it altered by the malicious acts of adversaries. Thus, it is considered received as a

variable. A label is a reference for I/O action. The declaration of identifiers should conform to the following rules: (i) Names must begin with a capital letter. (ii) Variables must begin with one of the letters u, x, y, w or z. (iii) Labels take any of the remaining letters. The declarations of identifiers can be as follows [45]:

DecName $ K1, K2, … , Km $;
DecVar $ x1, x2, … , xr $;
DecLabel $ a1, a2, … , an $;

- *Process* is the sequence of I/O actions at a host. The declaration of a participating host named *Pr* can be as follows:

val *Pr* = *P*;

The *P* is the sequence of I/O actions at the host *Pr*.

- *Configuration* is the pair ⟨s, *P*⟩, where s is the initial environment's knowledge and *P* is a description of processes at honest participants. The declaration is as follows:

val *Conf* = (*L* @ *Pr*);

The *L* is the initial environment's knowledge representing the adversary's knowledge, and the *Pr* is a process or the parallel composition of processes that are previously declared.

- *Property* is a trace that the protocol generates. In particular, any action should be preceded by a single action. The declaration is as follows:

val *Prop* = (*A* ← *B*);

The *Prop* is a property name such as *Auth, and A* and *B* are I/O actions.

Suppose host A sends a name *X* through an output action *a2*!*X1* and host B receives it as a variable *wX1* through an input action *b1*?*wX1*, then host B would not authenticate host A as the provider of the variable *wX1* unless the input action was preceded by the name *X1* sent out through the output action *a2*!. The declaration would be as follows:

val *Auth* = (*a2*!*X1* ← *b1*?*wX1*);

The protocol to analyze should be modeled and specified using the four kinds of STA declarations, then the STA script should be saved as *.sml file. Next, the file should be compiled with STA, and then the properties should be verified. The command used to analyze a security property is as follows:

CHECK *Conf Prop*

The *Conf* is the configuration of the system, and *Prop* is the property to check. The configuration and security properties should be declared in the STA script. The STA carries out trace analysis on the symbolic traces and verifies the correspondence assertions.

Table 5. STA declarations

Identifiers	
Named	DecName $ K1, K2, … , Km $;
Variables	DecVar $ x1, x2, … , xr $;
Labels	DecLabel $ a1, a2, … , an $;
Processes	val *Pr* = *P*;
Configuration	val Conf = (*L* @ *Pr*);
Properties	val *Prop* = (*A* ← *B*);

8 Modeling the Protocol and Specifying Properties Formally

8.1 Modeling the system

Modeling a security protocol [7, 9, 16] in a finite-state model checker requires approximation of the actual model to make the analysis finite. It is necessary to reduce the model by the establishment of two bounds: (a) a bound on the number of protocol runs, and (b) a bound on the number of possible messages the attacker can generate and send to trusted hosts participating in the protocol [9, 16]. Conversely, symbolic methods can analyze the whole infinite state space generated by a limited number of participants based on symbolic techniques. It discards the bound (b), since modeling the adversary is not required and it is sufficient to specify the environment's initial knowledge [9, 16].

The verification is carried out for an instance of the protocol. There are factors that have to be considered when choosing a size of the instance to be verified. They are: (i) The execution slows down as the number of participants and consequently the possible number of data values increases [7]. Therefore, the instance to analyze has to have a limited number of participants. (ii) The instance should be susceptible to the various malicious acts of intruders, especially the colluding attacks. For example, considering a model of an initiator and two executing hosts would not analyze the system for the colluding attacks. The smallest reasonable instance size would be four hosts including the initiator, which would allow the analysis of the colluding attacks. The second and the fourth hosts can be malicious hosts trying to truncate the data gathered at the intermediary host (the third host in the agent's itinerary). Also, any of the visited hosts might be malicious and try to attack the data that the agent has already gathered. The instance we considered is of three executing hosts *A*, *B*, and *C* and the initiator *I*.

Hosts *A* and *C* might be un-trusted hosts that co-operate with each other to amend the data they already provided, delete the data acquired at the intermediate hosts, spy out the gathered data, or insert arbitrary data to the gathered data. Also, hosts *A*, *B*, or *C* can be malicious hosts attempting individually to truncate the trailing data, append arbitrary data, impersonate the genuine initiator or spy out some confidential data. The selection of a model of four hosts would improve efficiency in terms of execution time, memory occupation, and execution traces. We would say that the *selection of a model of four nodes is reasonable. The system would be susceptible to colluding attacks as well as individual attacks of adversaries.*

The model we considered is depicted in Figure 6. The agent is initialized by the initiator *I*, then it migrates to hosts *A*, *B*, and *C* in order to gather some data, and finally it returns to the initiator *I*.

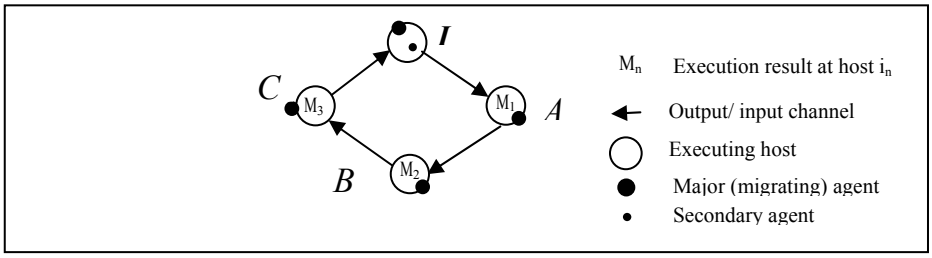

Fig. 6 Instance of proposed protocol with an initiator *I* and the visited hosts *A*, *B*, and *C*

The STA script of the proposed protocol consists of the declarations of: identifiers, processes, system configuration, and properties. The notations used in the declarations are shown in Figure 7.

For illustration, the I/O actions at host *I* are: *i1*! is an output action, *i2*? is an input action, and *i3*? is an input action by the secondary agent that communicates (outputs) the initial verification terms to the major (migrating) agent. An example of the *disclose* output action is *disclose*!(*Rold*), which denotes an output action that leaks the old nonce *Rold* to the environment. The action would represent the initial environment's knowledge.

In STA, an identifier that begins with *x*, *y*, *z*, or *w* is a variable representing a received term that might differ from originally sent term due to malicious acts of adversaries. For example, *M1* is a name that host *A* transmits to host *B,* then the name would be received at host *B* as a variable denoted as *yM1*.

I, *A*, *B*, *C*	Identities of the initiator, and the three honest hosts that participate in the protocol respectively
R	Fresh nonce that the initiator chooses randomly and would uniquely identify the protocol run
Rold	Old nonce
M0	Dummy data that the initiator generates
M1, *M2*, *M3*	Data that the initiator gathers from hosts *A*, *B*, and *C* respectively
+sigI, *+sigA*, *+sigB*, *+sigC*	Digital signature of hosts: *I*, *A*, *B*, and *C* respectively
+KI, +KA, +KB, +KC	Public key of hosts: *I, A, B, and C* respectively
-sigI, *-sigA*, *-sigB*, *-sigC*	Signature verification keys of hosts *I*, *A*, *B*, and *C* respectively
i1, *i2*, *i3*	(I/O) actions at hosts *I*
a1, *a2*	(I/O) actions at host *A*
b1, *b2*	(I/O) actions at host *B*
c1, *c2*	(I/O) actions at host *C*
Accept!	Output action that outputs the gathered data that pass all the necessary security verifications
guard?	Input action 'guardian' that can detect if the environment learns some piece of confidential data
disclose!	Output action that leaks some sensible data to the environment

Fig. 7 Notations used in the STA script of the proposed protocol

The declarations of labels, names, and variables used in the STA script of the proposed protocol are as follows:

DeclLabel $	a1, a2, b1, b2, c1, c2, i1, i2, i3, disclose, guard, Accept $;
DeclName $	Rold, R, SigI, SigA, SigB, SigC, I, A, B, C, M0, M1, M2, M3, KI, KA, KB, KC $;
DeclVar $	xM0, yM0, yM1, zM2, zM1, zM0, wM3, wM2, wM1, wM0, x, y, w, z, x1, x2, x3, x4, x4', x5, u, x1, x2, x3, x4, x4', x5, y1, y2, y3, y4, y5, y6, y5', w1, w2, w3, w4, w5, w6, w7, w1', w2', w3', w4', w6', w7', xIC, yIC, zIC, wIC, yDIC, zDIC, wDIC, xR, yR, zR, wR0, wR1, wR2, wR3, y, xIS, xID, yIS, yID, zIS, zID, wID, wIS $;

The declaration of the I/O actions at the initiating host *I* is as follows:

```
val iI = M0 new_in R new_in
        i1!((M0)^+KI, hsh (I), (hsh (I))^+SigI, R)^+KA >>
        i2?(((wM3, wID, I, wR3)^+SigC)^+KI,
            ((wM2, wID, C, wR2)^+SigB)^+KI,
            ((wM1, wID, B, wR1)^+SigA)^+KI,
            (wM0)^+KI, wDIC, wIS, wR3)^+KI >> (wID pkdecr (-SigI, wIS)) >>
            (wID is hsh(I)) >>
        i3?(M0, A, R) >>
            (wDIC is hsh(wM3, hsh(wM2, hsh(wM1, hsh(I))))) >>
            (wR3 is hsh(hsh(hsh(R)))) >>
            (wR2 is hsh(hsh(R))) >>
            (wR1 is hsh(R)) >>
            (wID is hsh(I)) >>
            (wM0 is M0) >>
            Accept!((A,M1), (B,M2), (C,M3)) >>   stop;
```

The declaration of the I/O actions at the participating host *A* is as follows:

```
val rA = M1 new_in
          a1?(xM0, xIC, xIS, xR)^+KA >> (xID pkdecr (-SigI, xIS)) >>
              (xID is hsh(I)) >>
          a2!(((M1, xID, B, hsh(xR))^+SigA)^+KI,
               xM0, hsh(M1, xIC), xIS, hsh(xR))^+KB >> stop;
```

The declaration of the I/O actions at the participating host *B* is as follows:

```
val rB = M2 new_in
          b1?(yM1, yM0, yDIC, yIS, yR)^+KB >> (yID pkdecr (-SigI, yIS)) >>
              (yID is hsh(I)) >>
          b2!(((M2, yID, C, hsh(yR))^+SigB)^+KI,
               yM1, yM0, hsh(M2, yDIC), yIS, hsh(yR))^+KC >> stop;
```

The declaration of the I/O actions at the participating host *C* is as follows:

```
val rC = M3 new_in
          c1?(zM2, zM1, zM0, zDIC, zIS, zR)^+KC >> (zID pkdecr (-SigI, zIS)) >>
              (zID is hsh(I)) >>
          c2!(((M3, zID, I, hsh(zR))^+SigC)^+KI,
               zM2, zM1, zM0, hsh(M3, zDIC), zIS, hsh(zR))^+KI >> stop;
```

The system declaration consists of: (a) the parallel composition of the role of the honest hosts that participate in the protocol and the respective public keys, and (b) a 'guardian' that can detect if the environment learns some sensible information, like *y*. The declaration of the system *Sys* is as follows:

```
val Sys = KI new_in KA new_in KB new_in KC new_in iI || rA || rB || rC ||
          guard?y >> stop;
```

The initial configuration of the system consists of: (a) the initial environment's knowledge where the *disclose!* output action leaks information to the environment, such as the public encryption keys and signature verification keys of hosts participating in the protocol, and (b) the role of the system *Sys*. The declaration of the configuration *Conf* is as follows:

```
val Conf = ( [disclose!(Rold, I, A, B, C, +KI, +KA, +KB, +KC, -SigI, -SigA,
                -SigB, -SigC) ]@Sys);
```

The proposed protocol carries out two types of verifications during the lifecycle of the agent. They are as follows:

1. Verifications on the identity of the genuine initiator at the early execution of the agent at the visited hosts. The verifications are necessary to detect if an adversary is impersonating the genuine initiator, and consequently to terminate the agent execution at the visited host if the malicious attack is detected.

2. Verifications upon the agent's return to the initiator. The verifications are necessary to analyze the authenticity, confidentiality, and strong integrity of the data that the mobile agent has gathered and returned to the initiator, and consequently to output the data if the verifications are passed.

The two types of verifications are explained in details and are expressed in STA as described below.

1. Initially, the agent is dispatched from host *I* to the first host in the agent's itinerary, which is host *A*. The term (hsh(I)^+SigI) that securely store the identity of the genuine originator would be received at host *A* as a variable denoted as xIS, as depicted in Figure 8. Upon the reception of the agent at host *A*, the host decrypts the agent's execution result with its private key -*KA*, and then it decrypts the term xIS, with the signature verification key of the initiator –SigI, so having the term xID. If the decryption is successful, then it compares the term xID with the hash of the identity of the genuine initiator hsh(I). If the verification fails it terminates the agent's execution, otherwise it continues the execution of the agent. The offer that

host *A* provides and signs is to be encrypted with the public key of the genuine initiator *I.* The verification is expressed in STA as follows:

$$(\text{xID pkdecr } (\text{-SigI, xIS})) \;>>\; (\text{xID is hsh(I)}) \qquad (1)$$

The verification is repeated upon the reception of the agent at every visited host in the agent's itinerary. The term (hsh(I)^+SigI) that securely store the identity of the genuine originator would be received at host *B* as a variable denoted as yIS, as depicted in Figure 8. The verification at host B would be as follows:

$$(\text{yID pkdecr } (\text{-SigI, yIS})) \;>>\; (\text{yID is hsh(I)}) \qquad (2)$$

The term (hsh(I)^+SigI) that securely store the identity of the genuine originator would be received at host *C* as a variable denoted as zIS, as depicted in Figure 8. The verification at host *C* would be as follows:

$$(\text{zID pkdecr } (\text{-SigI, zIS})) \;>>\; (\text{zID is hsh(I)}) \qquad (3)$$

In brief, the verification (1), (2), and (3) detect if an adversary is impersonating the genuine initiator so as to breach the privacy of the gathered data. And subsequently the executing host terminates the execution of the agent if the malicious act is detected.

2. Upon the reception of the agent at host *I*, the initiator performs the verifications depicted in Table 3 to detect any violation of the data integrity property. The verifications are expressed in STA as described below.

- The initial verification terms (r, m_0, i_1) that are stored within the secondary agent are verified with the terms that are returned with the major (migrating) agent. Figure 8 shows the flow of certain verification terms and the corresponding variables names as received at honest hosts participating in the protocol.

  For example, host *I* transmits the initial verification term *r* as a name R, then host *A* receives the term as a variable xR. Next, host *A* computes hsh(xR) and transmits hsh(xR) to host *B*, and then host *B* receives it as a variable yR and within the offer of host *A*. Next, host *B* computes hsh(yR) and transmits it to host *C*, and then host *C* receives it as a variable zR and within the offers of hosts *A*, and *B*. Next, host *C* computes hsh(zR) and transmits it to host *I*, and then host *I* receives it as a variable wR1 within the offer signed by host *A* , a variable wR2 within the offer signed by host *B*, and a variable wR3 within the offer signed by host *C*. Also, host *I* receives wR3 as the last term in the execution results of the migrating agent. The flow of the term is illustrated in Figure 8. The returned variables wR1, wR2,

and wR3 are expected to have the following values:

wR3 = hsh(hsh(hsh(R)))
wR2 = hsh(hsh(R))
wR1 = hsh(R)

Upon the agent's return, the major agent carries out the verifications in (4), (5), and (6) to check that the gathered and returned data belong to the protocol run of concern, which is identified by the term R, using the term R that is securely stored with the secondary agent and that is communicated to the agent through the input action *a3*!.

(wR3 is hsh(hsh(hsh(R)))) (4)
(wR2 is hsh(hsh(R))) (5)
(wR1 is hsh(R)) (6)

Initially host *I* generates a dummy offer M0 and transmits it within the migrating agent. Finally, it receives the offer as a variable wM0, as depicted in Figure 8. Upon the agent's return, the major agent carries out the verification in (7) to check that the returned dummy offer wM0 that the initiator generated at the initialization of the agent is returned intact using the term M0, which is securely stored within the secondary agent and that is communicated to the migrating agent through the input action *a3*!.

(wM0 is M0) (7)

♦ At the initiation, host *I* digitally signs the term hsh(I) and transmits it within the agent's execution results. Finally, it receives the signed term as a variable wIS, as depicted in Figure 8. Upon the agent's return, the initiator verifies that the returned data are generated for genuine initiator that signed the term hsh(I). The initiator decrypts the agent's execution results with its private key -*KI*, and then decrypts the varaible wIS using the signature verification key of the initiator -*SigI* so having the variable wID. If the decryption is successful, it compares the variable wID to hsh(I). If the verification passes, then it asserts that the gathered and returned data were generated for the genuine initiator host *I*. The decryption of the signed term wIS and the verification on the identity of the signer is expressed in STA and given in (8).

(wID pkdecr (-SigI, wIS)) >> (wID is hsh(I)) (8)

♦ The returned data integrity code wDIC is a chained hash of the data acquired at the visited hosts, as depicted in Figure 8. Upon the agent's return, the agent computes the data integrity code DIC using the terms wM1, wM2, and wM3 that are enclosed within the returned offers as in (9).

DIC = hsh(wM3, (hsh(wM2, (hsh(wM1, (hsh(I))))))) (9)

Next, it verifies that the computed data integrity DIC matches the variable wDIC. If the verification fails, then it deduces that data truncation took place and discards the gathered and returned data. The verification is expressed in STA as in (10).

(wDIC is hsh(wM3, hsh(wM2, hsh(wM1, hsh(I))))) (10)

The specifications and the security properties of the proposed protocol are expressed in STA as shown in Appendix A. The sequence of the verifications the proposed protocol carries is depicted in Figure 9. The major (migrating) agent starts its itinerary from host *I* and is dispatched by the initiator to host *A*. The process at host *I* starts with an output action *i1*, and then host *A* receives the agent through the input action *a1*?. Next, host *A* executes the agent and carries out the verification on the identity of the genuine initiator. If the verification passes, it provides its offer to the agent and dispatches the agent to host *B* through an output action a2!. Host *B* receives the agent through the input action *b1?*. Next, host *B* executes the agent and carries out the verification on the identity of the genuine initiator. If the verification passes, it provides its offer to the agent and dispatches the agent to host *C* through an output action *b2!*. Host *C* receives the agent through the input action *c1?*. Next, host C executes the agent and carries out the verification on the identity of the genuine initiator. If the verification passes, it provides its offer to the agent and dispatches the agent to host *I* through an output action *c2!*. Finally, host *I* receives the agent through the input action *i2?*, and then carries out the final verifications. If the verifications pass, then it sends out the gathered data through the output action i3!.

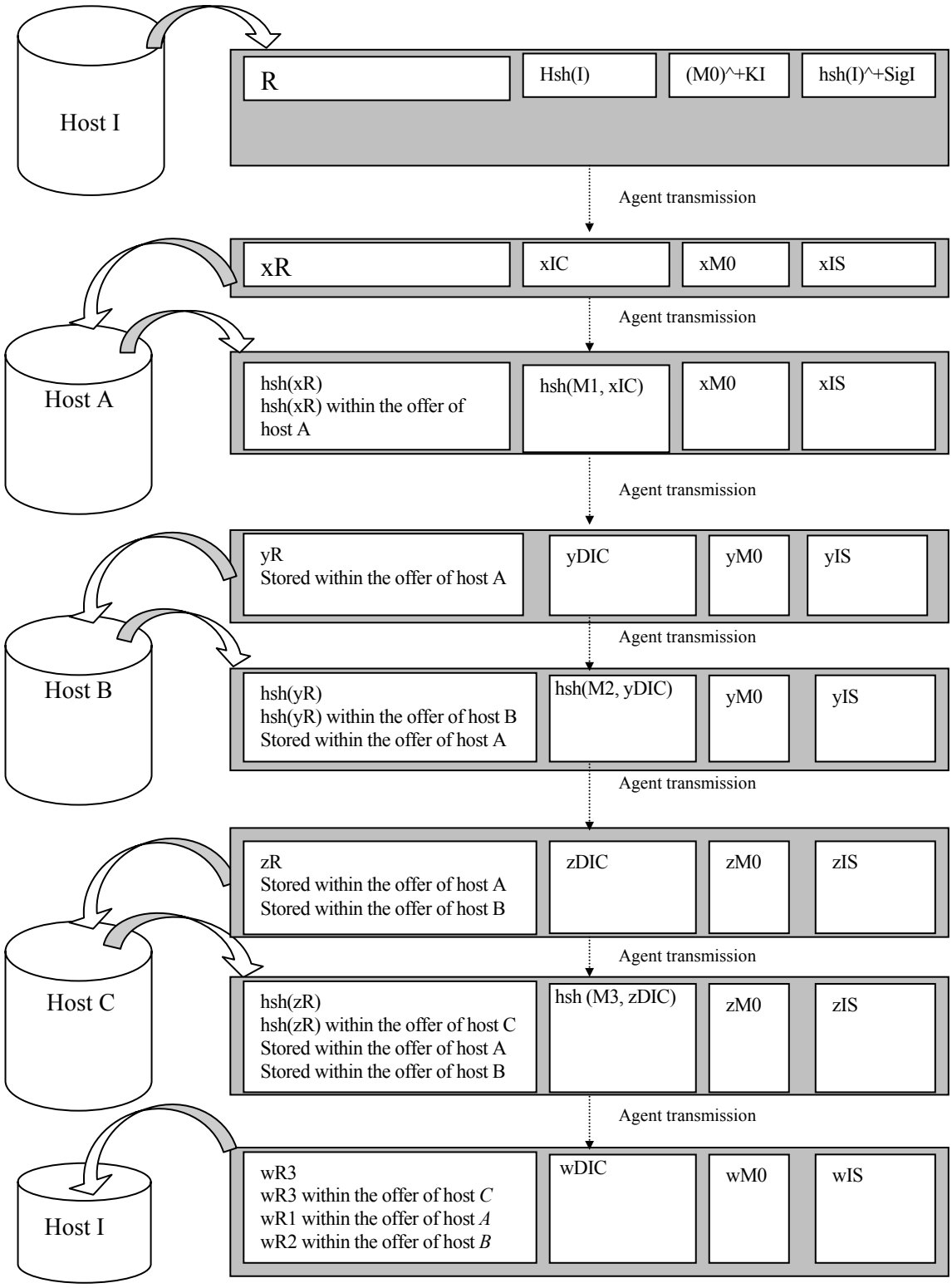

Fig. 8 The flow of the verification terms and the corresponding variables' names

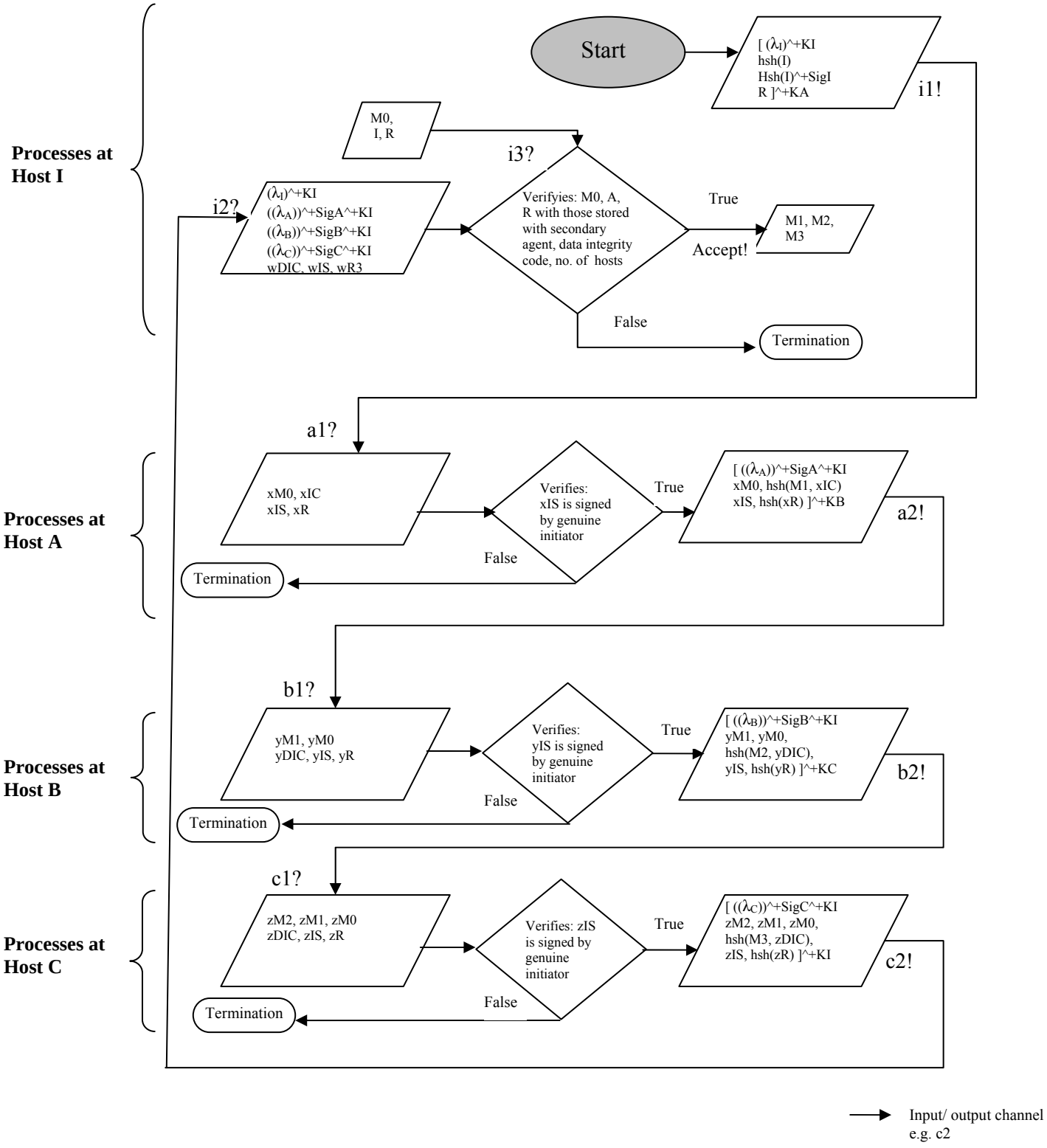

Fig. 9 A flow chart of the sequence of verifications in the proposed protocol

8.2 Modeling the environment

The environment represents the intruder's knowledge and capabilities. An intruder may intercept, fake, delete, insert, append, replace messages, or spy out confidential data. Also, two intruders may co-operate with each other to delete the data acquired at intermediate hosts or append the data of hosts of their own selection.

8.3 Formalizing the security properties

We are concerned with data authenticity, data confidentiality and strong data integrity of the data gathered by mobile agents. The properties are formalized as follows:

- The *Authentication* of *A* towards *B* requires that every trace generated by *B*'s input action is preceded by an *A*'s output of the same message. In our model, the hosts *A*, *B*, and *C* transmit *M1*, *M2*, and *M3* representing m_1, m_2, and m_3 respectively, through the output actions *a*2!, *b*2!, and *c*2!. Upon the agent's return, the initiator receives the three variables: *wM1*, *wM2*, *wM3* through the input actions *i*2?. We need to verify that the three variables actually originated from the respective hosts: *A*, *B*, and *C*. Each variable is contained within the offer that the respective host provided. Hence, verifying the authenticity of an offer concludes the authenticity of the respective variable.

The offer that host *A* provides is denoted as w3 and is sent through the output action *a2!*. The offer w3 encloses the data *M1* that host *A* provided to the agent. Host *I* receives the offer through the input action i2? as w3'. The authentication of the data *M1* is expressed in STA as in (11).

val *Auth8* = (a2!(w3, w4, w5, w6, w7) <-- i2?(w1, w2, w3', w4, w5, w6', w7)); (11)

The *Auth8* property means that the input action *i2*?*wM1* is really preceded by the corresponding output action of *a2*!*M1*. To verify *Auth8* property for the declared configuration *conf*, the following command is to be given at the ML interaction window:

> CHECK *Conf Auth8*; (12)

If the check (12) passes then it authenticates that *wM1* is the term *M1* which truly originated from host *A*.

The offer that host *B* provides is denoted as w2 and is sent through the output action *b2!*. The offer w2 encloses the data *M2* that host *B* provided to the agent. Host *I* receives the offer through the input action i2? as w2'. The authentication of the data *M2* is expressed in STA as in (13).

val *Auth9* = (b2!(w2, w3, w4, w5, w6, w7) <-- i2?(w1, w2', w3, w4, w5, w6', w7)); (13)

The *Auth9* property means that the input action *i2*?*wM2* is really preceded by the corresponding output action of *b2*!*M2*. To verify *Auth9* property for the declared configuration *conf*, the following command is to be given at the ML interaction window:

> CHECK *Conf Auth9*; (14)

If the check (14) passes then it authenticates that *wM2* is the term *M2* which truly originated from host *B*.

The offer that host *C* provides is denoted as w1 and is sent through the output action *c2!*. The offer w1 encloses the data *M3* that host *C* provided to the agent. Host *I* receives the offer through the input action i2? as w1'. The authentication of the data *M3* is expressed in STA as in (15).

val *Auth10* = (c2!(w1, w2, w3, w4, w5, w6, w7) <-- i2?(w1', w2, w3, w4, w5, w6', w7)); (15)

The *Auth10* property means that the input action *i2*?*wM3* is really preceded by the corresponding output action of *c2*!*M3*. To verify *Auth10* property for the declared configuration *conf*, the following command is to be given at the ML interaction window:

> CHECK *Conf Auth10*; (16)

If the check (16) passes then it authenticates that *wM3* is the term *M3* which truly originated from host *C*.

If any of the *authenticity verifications fails*: *(12), (14), (16)*, then the authenticity is unjust.

- The *Data secrecy* is such that an intruder can not reveal any communicated secret term. In our model the terms: *M0, M1*, *M2*, and *M3* are secret terms that should not be revealed except to the initiator *I*. Secrecy of the terms *M0, M1*, *M2*, and *M3* is expressed in STA as in (17), (18), (19), and (20) respectively.

val *Secrecy1* = (Absurd ← *guard*?*M0*); (17)
val *Secrecy2* = (Absurd ← *guard*?*M1*); (18)
val *Secrecy3* = (Absurd ← *guard*?*M2*); (19)
val *Secrecy4* = (Absurd ← *guard*?*M3*); (20)

For example, the *Secrecy1* property is verified by assuming a guardian that can at any time pick a message and tries to synthesize the secret *M1*. The input action guard?*M1* is such that the secret *M1* is learnt through the input action '*guard*', and the property: Absurd ← *guard*?*M1* is such that the input action *guard*?*M1* never takes place. To verify

the secrecy properties: (17) , … , (20) for the declared configuration *Conf*, the following commands should be given at the ML interaction window:

> CHECK *Conf Secrecy1*; (21)
> CHECK *Conf Secrecy2*; (22)
> CHECK *Conf Secrecy3*; (23)
> CHECK *Conf Secrecy4*; (24)

A check passes if the guardian never learns the secret *M1*. If any of the *secrecy verifications (21), ... , (24) fails*, then the secrecy property is breached.

- The *Strong Data integrity* property would be verified by carrying out the following four types verifications:

1. Verifying that the data term (*M1*, *M2*, *M3*) gathered at the respective hosts *A*, *B*, and *C* are received intact at host *I*. The data gathered are received at host I as variables (*wM1*, *wM2*, *wM3*). Each variable is contained within an offer that the host provided. Hence verifying that the offers that are received at host *I* correspond to those acquired at the hosts *A*, *B*, and *C*, implies that the data terms are received intact at host *I*. The verification would detect any of the following malicious acts of non-trusted hosts or intruders:

   a. Deletion of the data acquired at intermediate host/s, or alteration of the data the non-trusted host provided to the agent at an earlier time

   b. Truncation of the data acquired at the visited hosts

   Each offer is signed with the private key of the corresponding host. Then no one would be able to alter the data contained within the offer. Hence, verifying the authenticity of each offer ensures the integrity of the offers.

   The offer that host *A* transmitted through the output action *a2!* is denoted as w3, and is received through the input action *i2?* as w3′. The identity of the initiator for whom the offer was generated at host *A* is transmitted as w6 through the output action *a2!*, and is received as w6′ through the input action *i2?*. Also, the identity of the initiator of the agent is contained within w3 and w3′. The verification is expressed in STA as given in (12).

   The offer that host *B* transmitted through the output action *b2!* is denoted as w2, and the offer is received through the input action *i2?* as w2′. The identity of the initiator for whom an offer was generated at host *B* is transmitted as w6 through the output action *b2!*, and is received as w6′ through the input action *i2?*.

Also, the identity of the initiator of the agent is contained within w2 and w2′. The verification is expressed in STA as given in (14).

The offer which host *C* transmitted through the output action *c2!* is denoted as w1, and the offer is received through the input action *i2?* as w1′. The identity of the initiator for whom an offer was generated at host *C* is transmitted as w6 through the output action *c2!*, and is received as w6' through the input action *i2?* . Also, the identity of the initiator of the agent is contained within w1 and w1′. The verification is expressed in STA as given in (16).

2. Verifying that the term *R*, which uniquely identifies the protocol run of the agent that host *I* initiated is maintained intact during the agent's lifetime. The verification would detect if an adversary is able to replace the agent's dynamic data with the data of a similar protocol run without being detected. The verification requires two sorts of verifications, as follows:

   a. Verifying that the chained hash of the term *R* is maintained intact during its transmission from one host to the succeeding host in the agent's itinerary. The verifications are among the following input and output actions through which the chained hash of the term *R* is transmitted/ received:

      - *i1!* and *a1?*
      - *a2!* and *b1?*
      - *b2!* and *c1?*
      - *c2!* and *i2?*

   The chained hash of the term *R* is transmitted through the output actions i1!, a2!, b2!, and c2! as w7, and then it is received through the input actions a1?, b1?, c1?, and i2?, respectively as w7′. To express that every w7 is preceded by the corresponding w7′ during the lifecycle of the agent, the following properties are declared in STA:

```
val Auth4 = (i1!(w4, w5, w6, w7) <-- a1?(w4, w5, w6', w7'));                    (25)
val Auth5 = (a2!(w3, w4, w5, w6, w7) <-- b1?(w3, w4, w5, w6', w7'));            (26)
val Auth6 = (b2!(w2, w3, w4 ,w5, w6, w7) <-- c1?(w2, w3, w4, w5, w6', w7'));   (27)
val Auth7 = (c2!(w1, w2, w3, w4, w5 ,w6 ,w7) <-- i2?(w1, w2, w3, w4, w5, w6', w7')); (28)
```

   To verify the properties: (25), … , (28) for the declared configuration *Conf*, the following commands should be given at the ML interaction window:

```
> CHECK Conf  Auth4;   (29)
> CHECK Conf  Auth5;   (30)
> CHECK Conf  Auth6;   (31)
```

> CHECK *Conf Auth7*; (32)

b. Verifying that a participating host transmits a hash of the chained hash of *R* that the host received. For example, host *B* receives hsh(R) as yR, then it should transmit it as hsh(yR). The verifications are among the following input and output actions through which the chained hash of *R* is transmitted/ received:

- *a1?* and *a2!*
- *b1?* and *b2!*
- *c1?* and *c2!*

At initialization, host *I* transmits the term *R* to host *A*. The term *R* is received at host *A* through the input action *a1?* as xR. We need to verify that the host transmits a hash of the term xR through the output action *a2!*. The correspondence assertion is expressed in STA as given in the *Auth1* property (33). Next, the hashed term hsh(xR) is received at host *B* through the input action *b1?* as yR. We need to verify that the host transmits a hash of the term yR through the output action *b2!*. The correspondence assertion is expressed in STA as given in the *Auth2* property (34). Then, the hashed term hsh(yR) is received at host *C* through the input action *c1?* as zR. We need to verify that the host transmits a hash of the term zR through the output action *c2!*. The correspondence assertion is expressed in STA as given in (35).

val Auth1 = (a1?xR <-- a2!hsh(xR)); (33)
val Auth2 = (b1?yR <-- b2!hsh(yR)); (34)
val Auth3 = (c1?zR <-- c2!hsh(zR)); (35)

To verify the properties in (33), … , (35) for the declared configuration *Conf*, the following commands should be given at the ML interaction window:

> CHECK *Conf Auth1*; (36)
> CHECK *Conf Auth2*; (37)
> CHECK *Conf Auth3*; (38)

3. Verifying that the term ((H(I)^+SigI) that identifies the genuine initiator is maintained intact during the agent's lifetime. The verification would detect if an adversary was able to impersonate the genuine initiator without being detected during the agent's lifetime. The malicious act is usually intended for a breach of privacy of the collected data. The verification requires two sorts of verifications, as follows:

a. Verifying that the term (H(I)^+SigI)) is maintained intact during its transmission from one host to the succeeding host in the agent's itinerary. The verifications are among the following input and output actions:

- *i1!* and *a1?*
- *a2!* and *b1?*
- *b2!* and *c1?*
- *c2!* and *i2?*

The term (H(I)^+SigI) is transmitted through the output actions *i1!*, *a2!*, *b2!*, and *c2!* as w6, and then is received through the input actions *a1?*, *b1?*, *c1?*, and *i2?*, respectively as w6′. The verifications are expressed in STA as given in (29), (30), (31), and (32).

b. Verifying that an executing host transmits the term (H(I)^+SigI) the same as it is received without any tampering. The verifications are among the following input and output actions through which the signed term is transmitted/ received:

- *a1?* and *a2!*
- *b1?* and *b2!*
- *c1?* and *c2!*

The term (H(I)^+SigI) is received through the input actions *a1?*, *b1?*, and *c1?* as w6 and is transmitted through the output actions *a2!*, *b2!*, and *c2!*, respectively as w6′. The correspondence assertion is expressed in STA as given below.

val Auth12 = (a1?(w4, w5, w6, w7) <-- a2!(w3, w4, w5, w6', w7)); (39)
val Auth13 = (b1?(w3, w4, w5, w6, w7) <-- b2!(w2, w3, w4, w5, w6', w7)); (40)
val Auth14 = (c1?(w2, w3, w4, w5, w6, w7) <-- c2!(w1, w2, w3, w4, w5, w6', w7)); (41)

To verify the properties: (39), … , (41) for the declared configuration *Conf*, the following commands should be given at the ML interaction window:

> CHECK *Conf Auth12*; (42)
> CHECK *Conf Auth13*; (43)
> CHECK *Conf Auth14*; (44)

4. Verifying that the dummy offer *M0* which host I generated at the initiation of the agent is returned intact. The offer is encrypted with the public key of the host *I* and transmitted through the output action *i1!* as w4 and is returned to the host through the input action *i2?* as w4'. The correspondence assertion is expressed in STA as given in (45).

val Auth11 = (i1!(w4, w5, w6, w7) <-- i2?(w1, w2, w3, w4', w5, w6, w7)); (45)

To verify the property for the declared configuration *conf*, the following command should be given at the ML interaction window:

> CHECK *Conf Auth11*; (46)

If any of the *four integrity verifications given in (12), (14), (16),* (29) , … , (32), (36) , … , (38), (42) , … , (44), (46) *fails*, then the strong data integrity property is violated. Hence, the gathered data should be discarded.

9 Formal Verification of the Protocol

We analyzed the protocol for the following key configurations:

1. *Configuration 1*: A single run of the protocol with an initiator *I* and three executing hosts *A*, *B*, and *C*. The agent's itinerary is *I*, *A*, *B*, *C*, *I*. The protocol run is identified by a fresh nonce *R* that has been randomly chosen by the initiator, and a dummy data generated by the initiator: *M0*. The I/O actions of the protocol run are referenced as: *i1!*, *i2?*, *i3?*, *a1?*, *a2!*, *b1?*, *b2!*, *c1?*, and *c2!*. The configuration is depicted in Figure 10.

2. Two parallel runs of the protocol for three different configurations, as described below.

    a. *Configuration 2*: Two parallel runs. The first run with the initiator *I* and three executing hosts *A*, *B*, and *C*. The second run with initiator *I* and three executing hosts *E*, *B*, and *C*. The agent's itinerary in the first run is *I*, *A*, *B*, *C*, *I*, whereas the itinerary in the second run is *I*, *E*, *B*, *C*, *I*. The host *E* is a malicious host that has registered it-self as a participating host. The first and the second protocol runs are identified by a distinct nonce: *R*, and *R′*, respectively, and have been randomly chosen by the initiator *I*. The first protocol run is identified by a distinct nonce *R* that has been randomly chosen by the initiator, and a dummy data generated by the initiator: *M0*. The second protocol run is identified by a distinct nonce *R′* that has been randomly chosen by the initiator, and a dummy data generated by the initiator: *M'0*. The I/O actions of the first protocol run are referenced as: *i1!*, *i2?*, *i3?*, *a1?*, *a2!*, *b1?*, *b2!*, *c1?*, and *c2!* . The I/O actions for the second protocol run are referenced as: *i'1!*, *i'2?*, *b'1?*, *b'2!*, *c'1?*, and *c'2!*.The configuration is depicted in Figure 11.

    b. *Configuration 3*: Two parallel runs. The first run with the initiator *I* and three executing hosts *A*, *B*, and *C*. Hosts *A* and *C* are malicious hosts that have

registered themselves as participating hosts. The second run with initiator *I* and three executing hosts *E*, *B*, and *C*. The host *E* is a malicious host that has registered it-self as a participating host. The agent's itinerary in the first run is *I*, *A*, *B*, *C*, *I*, whereas the itinerary in the second run is *I*, *E*, *B*, *C*, *I*. The first protocol run is identified by a distinct nonce *R* that has been randomly chosen by the initiator, and a dummy data generated by the initiator: *M0*. The second protocol run is identified by a distinct nonce *R'* that has been randomly chosen by the initiator, and a dummy data generated by the initiator: *M'0*. The I/O actions of the first protocol run are referenced as: *i1!*, *i2?*, *i3?*, *b1?*, and *b2!* . The I/O actions for the second protocol run are referenced as: *i'1!*, *i'2?*, *b'1?*, and *b'2!*. The configuration is depicted in Figure 12.

c. *Configuration 4*: Two parallel runs. The first run with the initiator *I1* and three executing hosts *A*, *B*, and *C*. The second run with initiator *I2* and three executing hosts *E*, *B*, and *C*. The agent's itinerary in the first run is *I1*, *A*, *B*, *C*, *I1*, whereas the itinerary in the second run is *I2*, *E*, *B*, *C*, *I2*. The first protocol run is identified by a distinct nonce *R* that has been randomly chosen by the initiator *I1*, and a dummy data generated by the initiator: *M0*. The second protocol run is identified by a distinct nonce *R'* that has been randomly chosen by the initiator *I2*, and a dummy data generated by the initiator: *M'0*. The I/O actions of the first protocol run are referenced as: *i1!*, *i2?*, *i3?*, *a1?*, *a2!*, *b1?*, *b2!*, *c1?*, and *c2!* . The I/O actions for the second protocol run are referenced as: *i'1!*, *i'2?*, *a'1?*, *a'2!*, *b'1?*, *b'2!*, *c'1?*, and *c'2!*. The configuration is depicted in Figure 13.

The role of a malicious host is not explicitly modeled. It is implicitly modeled by the environment that is described in the configuration *Conf*.

The STA scripts of the four protocol runs are shown in Appendices A, B, C, and D respectively, including the declaration of identifiers, system processes, configuration, and properties due to limitation in space. In the configuration we set the environment initial knowledge to: (a) old nonce *Rold,* (b) participating hosts' identities and the associated public keys and signature verification keys, and (c) an intercepted message.

The verification of data authenticity, strong data integrity and data confidentiality properties using STA reported no attacks. The results of verifications of the proposed protocol with the reached symbolic configurations are shown in Figures 14, 15, 16 and 17 respectively. The results of the analysis of the small instance of the protocol in the four key configurations implies that the proposed protocol is free of security flaws and would provide a motivation for a proof of security of the protocol of an arbitrary size.

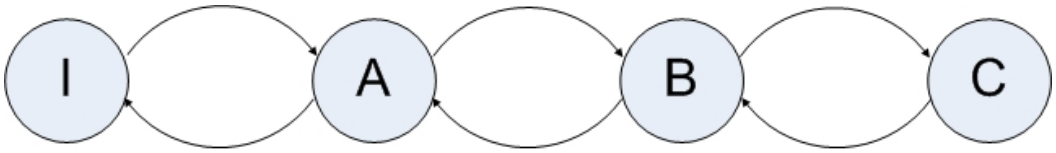

Fig. 10 Configuration 1: Initiator I, and three executing hosts A, B, and C

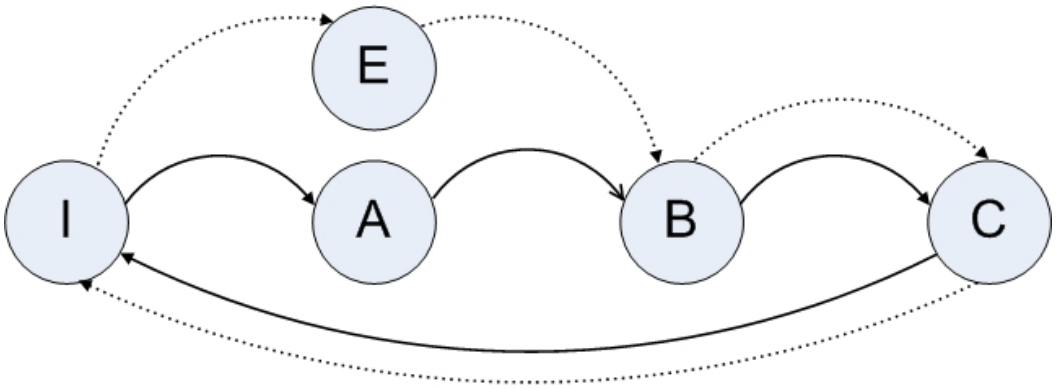

Fig. 11 Configuration 2: Two parallel runs
1st Run: Initiator I, and three executing hosts A, B, and C
2nd Run: Initiator I, and three executing hosts E (malicious host), B, and C

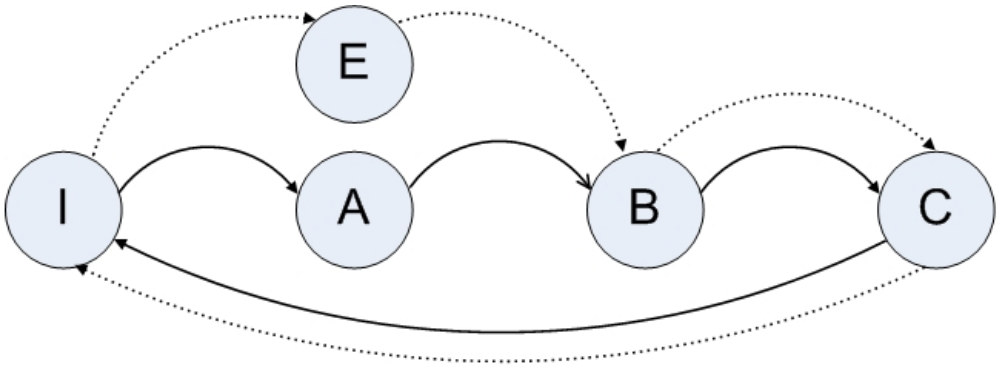

Fig. 12 Configuration 3: Two parallel runs
1st Run: Initiator I, and three executing hosts E (malicious host), B, and C
2nd Run: Initiator I, and three executing hosts A (malicious host), B, and C (malicious host)

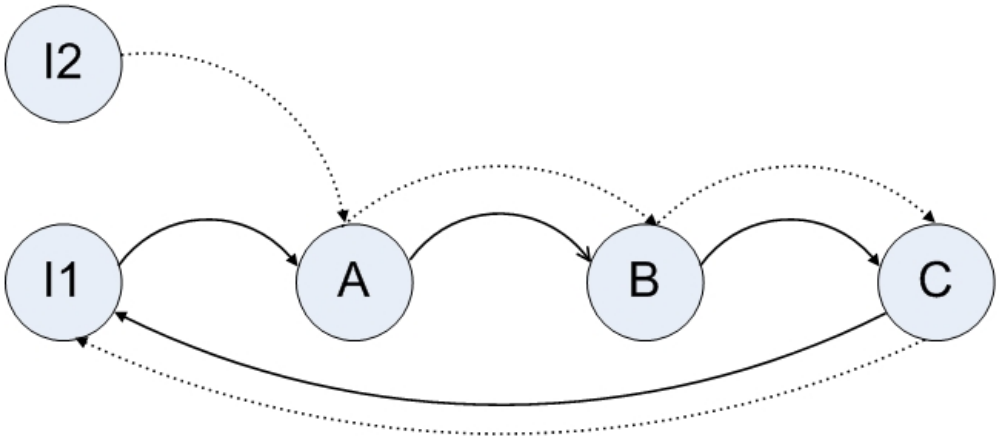

Fig. 13 Configuration 4: Two parallel runs
1st Run: Initiator I1, and three executing hosts A , B, and C
2nd Run: Initiator I2, and three executing hosts E, B, and C

```
> val it = "No attack was found  327 symbolic configurations reached." : string
```

Fig. 14 Result of analyzing the configuration 1 with a single run of the proposed protocol

```
> val it = "No attack was found  109603 symbolic configurations reached." : string
```

Fig. 15 Result of analyzing the configuration 2 of two parallel runs of the proposed protocol. The 2nd run with a malicious host

```
> val it = "No attack was found  328 symbolic configurations reached." : string
```

Fig. 16 Result of analyzing the configuration 3 of two parallel runs of the proposed protocol: (a) the 1st run with a malicious, and (b) the 2nd with two co-operating malicious hosts

```
> val it = "No attack was found  1959 symbolic configurations reached." : string
```

Fig. 17 Result of analyzing the configuration 4 of two parallel runs of the proposed protocol with different initiators: I1, and I2

The results of analyzing the proposed protocol with the reachable states are summarized in Table 6.

Table 6. The verification results of the proposed protocol with the reachable states

Configurations of the protocol runs	Verification results
a single run of the proposed protocol	No attack, 327 states
Two parallel runs of the proposed protocol. The 2nd run with a malicious host	No attack, 109603 states
Two parallel runs of the proposed protocol: *(a) the 1st run with a malicious* *(b) the 2nd with two co-operating malicious hosts*	No attack, 328 states
Two parallel runs of the proposed protocol with different initiators: I1, and I2	No attack, 1959 states

10 Conclusion and Motivation

Several protocols were presented in the literature aim to assert the security properties of mobile agent's execution results such as integrity, confidentiality, and authenticity in

the presence of malicious hosts and intruders. However, they were not able to completely achieve the aimed for security properties. They did not achieve particular security properties, such as strong data integrity [14, 20, 22, 23, 28, 29]. It is attributed to incomplete designs, where a proper design of a security protocol should consist of: (a) precise security requirements, (b) clear assumptions, (c) various capabilities of adversaries, especially the conspiracy of non-trusted hosts, (d) formal specifications and properties, (e) and formal verification of the security properties a protocol aims to accomplish.

In this paper we present a security protocol which is intended for the protection of the execution results of mobile agents and targets strong data integrity, authenticity, and confidentiality. The protocol is derived from the *Multi-hops protocol* [14], where the security relies on a chain of signed offers, a message authentication code and a chained hashing of a random nonce. The *Multi-hops protocol* does not accomplish strong data integrity. It is not able to detect the data truncation or replacement attacks, which can take place when a host conspires with a preceding host in agent's itinerary and sends the agent back to it, so the preceding host would be able to truncate the data acquired at the intermediary hosts and alter the data it formerly provided without being detected by replacing the recent agent's dynamic data with the former data that was current when the agent firstly visited it, as long as it is still storing the former data. Also, it can not detect the attack, where an adversary might sign others data with its own private key, since the collected data are transmitted in plain text in M_n. We enhanced the protocol so that it hinders or at least detects the attacks which the *Multi-hops protocol* and the protocols presented in the literature [20, 22, 23, 28, 29] were not able to detect. The proposed protocol refines the Multi-hops protocol by: (a) employing two co-operating agents, a migrating agent and a stationary agent, (b) requesting any executing host to clear its memory from the data acquired as a result of executing the agent before it dispatches the agent to the succeeding host, (c) jumbling of collected offers to mislead an adversary trying to truncate offers collected at preceding hosts, and (d) carrying out intermediate verifications at visited hosts on the identity of the genuine initiator, based on storing securely the identity of the genuine initiator within the migrating agent.

The *two co-operating agents* are a *major agent* and a *secondary agent*. The major agent traverses the Internet searching for particular data, and the secondary agent resides at the initiating host and securely stores the terms needed for accurate verifications on: (a) the nonce *r* which uniquely identifies a particular protocol run, (b) the dummy data generated by the initiator m_0 , and (c) the identity of the first host in agent's itinerary i_1. Upon agent's return, the secondary agent communicates the terms to the initiator to carry out the followings: (i) Verify that the computed γ'_n with *r* as an initial value matches the returned γ_n . (ii) Deduce the actual agent's itinerary from hosts' identities which are enclosed within the chain λ, and then verify that first host in the assembled agent's itinerary is i_1. (iii) Verify the γ_j enclosed in each encapsulated offer λ_j matches the computed γ'_j based on the order of the host in the assembled agent's itinerary for all

offers in the chain λ. (iv) Verify that the last decrypted term in the chain λ matches m_0. Commonly, verifications are based on initial data that are stored within the migrating agent. However, an adversary might tamper with the initial data as the agent transfers through public communication channels, and thus the verifications are not truly accurate. Hence, the storing of the initial verification data within the secondary agent ensures that the data are intact and verifications are truly accurate. The intention to store the initial verification data within the secondary agent and not within the initiator's memory is to enable the initiator to trace any tampering with the initial verification data. Adversaries might attempt to tamper with the initiator's memory. The execution traces, which Vigna recommends in [51], might be implemented. The technique requests an agent executor to create and store a trace of the execution of the secondary agent. Upon the agent's return, the initiator verifies the initial verification data stored within the secondary agent through the stored execution trace. If the verification passes, then the subsequent verifications would be accurate.

The protocol implements certain security techniques which would ensure the integrity of the acquired data such as *jumbling of the acquired offers, and computing data integrity code and a counter of the actually visited hosts*. The jumbling of offers is intended to deceive an adversary trying to delete the offers acquired at preceding host/s. The offers are jumbled so having the dummy offer, which is generated by the initiator, as the last offer within the chain of offers. Hence, the malicious act of deletion of the last offer in the chain of offers would be detected upon the agent's return to the initiator. The initiator checks if the dummy offer is within the chain of offers. If the verification fails, then it implies that tampering with the chain of offers took place. The data integrity code and the counter of visited hosts are used to detect any tampering with the acquired offers such as deletion, insertion or modification of data. Each acquired offer includes the data provided by the executing host, the data integrity code as computed at the host, the identity of the succeeding host, and the term which identifies the genuine initiator for whom the offer is generated. *Intermediate verifications are carried out during the execution of the agent at executing hosts to verify the identity of the genuine initiator*. Each executing host, at the early execution of the agent should verify that the signed term matches the hash of the identity of the genuine initiator, otherwise the execution terminates. Also, the *transmitted data is encrypted with the public key of the succeeding host in agent's itinerary* to reduce the chances of intruders to learn any of the terms enclosed within the transmitted message in case the intruder intercepts the message. The enclosed terms include the identity of the genuine initiator, the nonce which identifies the protocol run, the terms needed to append or insert a new offer e.g. data integrity code. In order to accomplish strong data integrity, and prevent or at least detect two malicious hosts co-operating with each other with the intension of truncating the data acquired at intermediate hosts, and replacing the data which they already provided with a new data trying to have agent's decision in their favors, the *protocol requests each executing host to clear its memory from the terms acquired during the execution of the agent* such as the data integrity code before the host transmits the agent to the

succeeding host. An executing host may not respond to the request. The denial of clearing request can be traced by implementing the execution traces technique recommended by Vigna in [51]. Also, the initiator can *assemble the actual agent's itinerary* from the chain λ which binds the identity of an executing host to the identity of the succeeding host as acknowledged in the acquired offers. Moreover, the initiator can *deduce the actual the number of visited hosts* by counting the number of times the nonce is hashed. Hence, the initiator can detect the malicious act of deletion, truncation or insertion of offers.

The proposed protocol accomplishes the proper design stages of security protocols. We used formal methods to model the system and to analyze its security properties. A small instance of the protocol is verified for the security properties: strong data integrity, data authenticity, and confidentiality. The results of the formal verification showed that he protocol is free of security flaws in a four key configurations. Moreover, we reasoned about the security of the protocol of a general model in section 5.8 and showed that the protocol is capable of preventing or at least detecting the attacks revealed in the existing protocols. Hence, we can say that the proposed protocol is free of the flaws revealed in the existing protocols and would overcome the attacks of adversaries which were revealed in [14, 20, 22, 23, 28, 29]: (a) truncation of collected data, (b) alteration of the data which a host formerly provided in case the host co-operates with a succeeding malicious host, (c) impersonating the genuine initiator and hence breach the privacy of collected data, (d) sending others data under the private key of a malicious host, and (e) replacing the collected data with data of similar agents.

This paper demonstrates the usefulness of employing formal methods in analyzing the proposed security protocol. We verified the security protocol using the STA tool for data authenticity, data confidentiality, and strong data integrity properties. The STA tool is as an automatic verification tool that is used to analyze the properties of security protocols. It performs a complete exploration of the state space and analyzes execution traces based on symbolic transitions, which would lead to a compact model, whereas the analysis of execution traces in model checking or the checking of the may-testing equivalence of Spi-calculus suffer from state-explosion problem. The STA tool is practically efficient compared to other formal methods, such as theorem proving. It takes less than half an hour to write the STA script of a protocol run. According to the authors in [8, 9, 16] the symbolic analysis is sound and complete. Detecting an attack on the symbolic model would imply that an attack exists in the infinite standard model and vice versa. The execution is automatic, though, it slows down as the number of data values increases.

In conclusion, the proposed mobile agent security protocol is analyzed and formally verified for various kinds of attacks and the verification of the different runs of the protocol showed no flaws in the protocol. We found that the protocol is secure for the modeled configuration and would satisfy the intended security properties. Hence, the

proposed protocol would inspire the deployment of mobile agents in e-commerce applications having assurance in the searched and gathered data.

11 Future Works

A protocol designer should specify and verify a new protocol formally before presenting it for implementation and claiming the assertion of particular security properties. Many of the existing protocols are not yet formally described and verified. It is very beneficial to apply formal methods to existing protocols, which would help in fixing such protocols or getting a formal proof of their correctness and safety, as applicable. This would be the issue of future work. Care should be taken in selecting a formal method for the verification of a security protocol. In section 6 various formal methods were briefly discussed. The modal logic methods lack automated tools, model particular security properties such as authentication, and are error-prone. The theorem proving methods require special skills. The modal algebra methods require checking equivalence of processes and the checking that two processes are indistinguishable for any tester process is difficult, especially the proofs are not automated. The model checking methods relies on finite approximation of the actual model, which requires a bound on the number of protocol runs and the number of possible messages the attacker can generate and send to honest participating hosts at any moment. Hence, a prior knowledge of the protocol to analyze is required. Also, the adversary has to be modeled explicitly. The symbolic methods make no assumption with respect to the infinite model. The adversary is represented by the environment's initial knowledge and no explicit modeling of the adversary is required. Thus, it does not require prior knowledge of the protocol to analyze. It requires familiarity with process algebra. The state space of a system might increase exponentially as the size of the system grows linearly, and accordingly the execution slows down. Thus, a variety of factors should be considered when selecting a formal method [7]: (i) Usability, such as a high level user interface [25]. The user interface of STA is rudimentary. However, specifying the protocol in STA requires about half an hour if being familiar with process algebras. (ii) Required knowledge and skills to model the system. (iii) System specifications of interest, e.g. the security properties of interest. (iv) Memory occupation. According to Boreale and Buscemi, STA implements a depth-first search approach, which controls the memory occupation appropriately [7]. (v) Availability of automatic proofs and the expected execution time. For example, the execution time is greatly shorter in STA as compared to Murφ [7, 35], which depends on the number of possible transitions branching from states. (vi) The model size, as the model is expected to increase dramatically as the number of participants and consequently the possible data values increase. (vii) Explicit modeling of the adversary. For example, in Murφ it takes more than half of the time required to model the protocol [35]. The appropriate selection of a method would result in an efficient verification and would certainly detect flaws in a protocol, if any exists.

It is worthy to verify the security properties of the proposed protocol using other formal methods, such as model checking and theorem proving, so as to compare the results of verification in terms of simplicity, usability, accuracy and efficiency.

Acknowledgement

I am grateful to Nandan Parameswaran, Kai Engelhardt, Amjad Hudaib, Ralf Huuck, and John Zic for their close guidance, and the frequent and fruitful discussions contributed to the writing of the paper in its final form. Also, I would like to express my appreciation to Cathy Meadows for the thorough reviewing of the paper and providing valuable comments, and to Marzia Buscemi for commenting on the paper, especially the fine comments on the sections of design and analysis of the proposed protocol.

References

1. Abadi, M., and Blanchet, B., and Fournet, C. Just fast keying in the Pi. Manuscript. Available at: http://www.di.ens.fr/~blanchet/crypto/jfk.html, Dec. 2003.
2. Abadi, M., and Fournet, C. Mobile values, new values, and secure communications. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pp. 104-115, Jan. 2001.
3. Abadi, M., and Gordon, A. D. A calculus for cryptographic protocols: The Spi-Calculus. In *Information and Computation*, 148(1): 1-70, 1999.
4. Aziz, B., Gray, D., Hamilton, G., Oehil, F., Power, J., and Sinclair, D. Implementing protocol verification for E-commerce. In *Proceedings of the 2001 International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2001)*, L'Aquila, Italy, 6-12 Aug. 2001.
5. Blanchet, B., and Aziz B. A Calculus for Secure Mobility. In *18th Asian Computing Science Conference (ASIAN'03)*, volume 2896 of LNCS, pp. 188-204, Springer-Verlag, 2003.
6. Boreale, M. and Gorla, D. Process calculi and the verification of security protocols. In *Journal of Telecommunications and Information Technology – Special Issue on Cryptographic Protocol Verification (JTIT 2002/4)*, Warsaw, Poland, 2002.
7. Boreale, M. G., and Buscemi, M. Experimenting with STA, a tool for automatic analysis of security protocols. *ACM Symposium on Applied Computing 2002*, ACM Press, 2002.
8. Boreale, M. Symbolic trace analysis of cryptographic protocols. In *Proceedings Of ICALP'01*, volune 2076 of LNCS, Springer, 2001.
9. Boreale, M., and Buscemi, M. G. A framework for the analysis of security protocols. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002)*. Lecture Notes in Computer Science, vol. 2076. Springer-Verlag, Heidelberg, 667-681.
10. Bradshaw, J. M. An Introduction to software agents, In *Software Agents*, J. M. Bradshaw, Ed., Chapter 1, pp. 3-46, AAAI Press, 1997.
11. Burrows, M., Abadi, M., and Needham, R. A logic of authentication, ACM *Transactions in Computer Systems*, 8(1): 18-36, Feb. 1990.
12. Cachin, C., Camenisch, J., and Kilian, J., and Müller, J. One round secure computation and secure autonomous mobile agents. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP)*, U. Montanari, J. P. Rolim, and E. Welzl, editors, volume 1853 of LNCS, pp. 512-523, Springer-Verlag, 2000.
13. Corradi, A., Montanari, R., and Stefanelli, C. Mobile agents integrity in E-commerce applications. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop (ICDCS'99)*, IEEE Computer Society Press, pp. 59-64, Austin, Texas, May 31- June 5, 1999.
14. Corradi, A., Montanari, R., and Stefanelli, C. Mobile agents protection in the Internet environment. In *the 23rd Annual International Computer Software and Applications Conference (COMPSAC '99)*, pp 80 – 85, 1999.
15. Durante, L., Sisto, R., and Valenzano, A. A state-exploration technique for spi-calculus testing equivalence verification. In *Proceedings of the IFIP International Joint Conference on Formal Description Techniques for Distributed Systems and*

Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX) 2000. Kluwer Academic Publishers, Dordrecht, 155-170.
16. Fiore, M., and Abadi, M. Computing Symbolic models for verifying cryptographic protocols. In *Proceedings of the 14th IEEE ComputerSecurity Foundations Workshop (CSFW 2001)*. IEEE Computer Society Press, Washington, 160-173.
17. Fischer, L. *Protecting integrity and secrecy of mobile agents on trusted and non-trusted agent places*. Thesis Dissertation, University of Bremen, Germany, April 2003. Available at: http://www.sec.informatik.tu-armstadt.de/lang_neutral/diplomarbeiten/docs/fischer_diplom.pdf.
18. Formal Systems (Europe) Ltd. Failures Divergence Refinement. FDR2 User Manual. Available at: http://www.formal.demon.co.uk/fdr2manual/index.html, 3May 2000.
19. Fournet, C., Gonthier, G., Lévy, J. J., Maranget, L., and Rémy, D. A calculus for mobile agents. In *7th International Conference on concurrency Theory (CONCUR'96)*, volume 1119 of LNCS, pp. 406-421, Springer, Aug. 1996.
20. Hannotin, X., Maggi, P., and Sisto, R. Formal specification and verification of mobile agent data integrity properties: A case study. Volume 2240 of LNCS, pp. 42-53, Springer-Verlag, 2001.
21. Jaljouli, R. Boosting m-Business using a truly secured protocol for information gathering mobile agents. To appear in *Proceedings of the 4th International Conference on Mobile Business*, IEEE Computer Society Press, 2005.
22. Kanik, N., and Tripathi, A. Security in the Ajanta mobile agent system. Technical Report TR-5-99, University of Minnesota, Minneapolis, MN 55455, U.S.A., May 1999.
23. Karjoth,G., Asokan, N. and Gülcü, C. Protecting the computation results of free-roaming agents. In *K. Rothermel and F. Hohl editors, Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477 LNCS, pp. 195-207, Springer-Verlag, Heidelberg, Germany, 1998.
24. Lowe, G. Breaking and fixing the Needham-Schroeder public key protocol using FDR. In *Proc. Tools and Algorithms for the construction and Analysis of systems (TACAS'06)*, volume 1055 of LNCS, pp. 147 – 166, Springer-Verlag, 1996.
25. Lowe, G. Casper, A compiler for the analysis of security protocols. In *Proceedings of the 10th Computer Security Foundation Workshop (PCSFW)*, IEEE Computer Society Press, 1997.
26. Ma, L., and Tsai, J. J. P. Formal Verification Techniques for Computer Communication Security Protocols. In Handbook of Software Engineering and Knowledge Engineering vol. 1. Available at: ftp://cs.pitt.edu/chang/handbook/12.pdf
27. Maggi, P., and Sisto, R. Using SPIN to Verify Security Properties of Cryptographic Protocols. In *Proceedings 9th International Spin Workshop on Model Checking of Software (SPIN 2002)*, volume 2318 of LNCS, pp. 187-204, Grenoble, France, April 2002.
28. Maggi, P., Sisto, R. Experiments on formal verification of mobile agent data integrity properties. Available at: http://www.labic.disco.unimib.it/woa2002/papers/15.pdf
29. Maggi., P and Sisto, R. A configurable mobile agent data protection protocol. In *Proceedings of AAMAS'03*, ACM Press, New York, NY, USA, 2003, pp. 851–858.
30. Meadows, C. Formal Verification of Cryptographic Protocols: A Survey. In *Advances in Cryptography – ASIACRYPT'94*, pp. 135-150.
31. Meadows, C. Language generation and verification in the NRL protocol analyzer. In *9th IEEE Computer Society Foundations Workshop,* Kenmare, Ireland, March 1996.

32. Millen, J. K., Clark, S. C., and Freedman, S. B. The interrogator: protocol security analysis, IEEE *Transactions on software engineering*, SE-13(2), 1987.
33. Milner, R., Parrow, J., Walker, D. A calculus for mobile processes (part I and II). In *Information and Computation*, 100:1-77, 1992.
34. Milner, R., Tofte, M., Harper, R., and MacQueen, D. The definition of standard ML-Revised, MIT Press, 1997.
35. Mitchell, J. C., Mitchell, M., and Stern, U. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of Symposiums on Security and Privacy*, IEEE Computer Society Press, 1997, pp. 141-153.
36. Mobility Workbench. Available at: http://www.it.uu.se/research/group/mobility/mwb
37. Moscow ML. Available at: http://www.dina.dk/~sestoft/mosml.html
38. Paulson, L. C. Proving properties of security protocols by induction. In *Proceedings of the 10th Computer Society Foundations Workshop,* June 1997.
39. Process Algebra Compiler. Available at: http://www.reactive-systems.com/pac
40. Roth, V. Empowering mobile software agents. In *Proceedings 6th IEEE Mobile Agents Conference*, Suri, N. editor, volume 2535 of LNCS, pp. 47-63, Springer-Verlag, Oct. 2002.
41. Roth, V. Programming Satan's agents. In *Proceedings of the 1st International Workshop on Secure Mobile Multi-Agent Systems*, Montreal, Canada, 2001.
42. Sander, T., and Tschudin, C. F. Protecting mobile agents against malicious hosts. In *Mobile agents and Security*, volume 1419 of LNCS, pp. 379-386, Springer-Verlag, 1998.
43. Sangiori, D. Expressing mobility in process algebra: first order and higher order paradigms, PHD thesis, university of Edinburgh, 1992.
44. Sewell, P. Global/ Local Subtyping and Capability Inference for a Distributed Calculus. In *Automata, Languages and Programming, 25th International Colloquium (ICALP'98)*, volume 1443 of LNCS, pp. 695-706, Springer-Verlag, July 1998.
45. STA Documentation. Available at: http://www.dsi.unifi.it/~boreale/documentation.html.
46. STA: a tool for trace analysis of cryptographic protocols. ML object code and examples, 2001. Available at: http://www.dsi.unifi.it/~boreale/tool.html.
47. Stephen, R., and KE Xu, T. Mobile agent security through multi-agent cryptographic protocols. In *Proceedings of the 4th International Conference on Internet Computing (IC 2003)*, volume 2003, pp. 462-468, 2003.
48. Symbolic Model Prover. Available at: http://www-2.cs.cmu.edu/~modelcheck/symp.html
49. Syverson, P.F., Goldschlag, M., and Reed, M. G. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy*, pp. 44-54, Oakland, California, 1997.
50. Tschudin, C. Mobile agent security. In *Intelligent Information Agents: Cooperative, Rational and Adaptive Information Gathering on the Internet*, Klusch, M. editor, LNCS, pp. 431-446, Springer-Verlag, Barlin, Germany, 1999.
51. Vigna, G. Cryptographic traces for mobile agents. In Mobile Agent Security, Vigna, G., editor, 1419 LNCS, pp. 137-153, Springer-Verlag, Heidelberg, Germany, 1998.
52. Vitek, J., and Gastagna, G. Seal: A Framework for Secure Mobile Computations. In *Internet Programming Language ICCL'98* Workshop, volume 1686 of LNCS, pp. 47-77, Springer-Verlag, May 1999.
53. Wang, T., Guan, S., and Chan, T. Integrity protection for Code-on-Demand mobile agents in e-commerce. In *Systems and software*, 60(3): 211-221, 15 February 2002.

54. Wang, X. F., Yi, X., Lam, K. Y., and Okamoto, E. Secure information gathering agent for internet trading. In *Proceedings of the 4th Australian Workshop on Distributed Artificial Intelligence on multi-Agent Systems: theories, Languages, and Applications (DAI-98)*, Zhang, C., and Lukose, D. editors, volume 1544 of LNCS, pp. 183-193, Springer-Verlag: Heidelberg, Germany, July 1998.
55. Yao, Y., and Dolev, D. On the security of public key protocols. IEEE *Transactions on Information Theory*, 29(2): 198-208, March 1983.
56. Yee, B. S. A sanctuary for mobile agents. Technical Report CS97-537, UC San Diego, Department of Computer Science and Engineering, April 1997.

Appendix A

STA script for a single run of the protocol with R as a fresh nonce generated by the initiator i_0.

```
(* A single run of the proposed protocol
I: initiator; A: 1st executing host, B: 2nd executing host,
C: 3rd executing host, Adv: adversary, R: fresh none, Rold: old nonce,
M0: dummy data generated by host I, M1: execution result at host A,
M2: execution result at host B, M3: execution result at host C
```

Host I sends the mobile agent to gather information from hosts A, B, and C. At the end of the protocol, host I needs to analyze the integrity, confidentiality, and authenticity of the returned execution results M1, M2, and M3 acquired at hosts A, B, and C respectively.

Notations used in declaring identifiers are as given below.
i1!: output action at host I, i2?: input action at host I,
i3?: input action at host I that follows the output action of the secondary agent $\mathcal{A}s$ which communicates the initial verification data to major agent $\mathcal{A}$,
a1?: input action at host A, a2!: output action at host A,
b1?: input action at host B, b2!: output action at host B,
c1?: input action at host C, c2!: output action at host C,
disclose!: an output action that leaks information to the environment,
guard?: an input action 'guard' such that a guardian learns some secret data,
accept!: an output action that outputs the execution results which satisfy the intended security properties to host I upon agent's return,
KA, KB, KC, KI: public keys of hosts A, B, C, and I respectively,
SigA, SigB, sigC, SigI: private signing keys of hosts A, B, C, I respectively)

```
DeclLabel $ a1, a2, b1, b2, c1, c2, i1, i2, i3, disclose, guard, Accept $;
DeclName $ Rold, R, SigI, SigA, SigB, SigC, I, A, B, C, M0, M1, M2, M3, KI, KA,
           KB, KC $ ;
DeclVar $  xM0, yM0, yM1, zM2, zM1, zM0, wM3, wM2, wM1, wM0,
           y, w1, w2, w3, w4, w5, w6, w7, w1', w2', w3', w4', w6', w7',
           xIC, yIC, zIC, wIC, yDIC, zDIC, wDIC,
           xR, yR, zR, wR0, wR1, wR2, wR3,
           xIS, xID, yIS, yID, zIS, zID, wID, wIS $;

(*The process at the initiator is declared as iI)
val iI = M0 new_in R new_in
        i1!((M0)^+KI, hsh (I), (hsh (I))^+SigI, R)^+KA >>
        i2?(((wM3, wID, I, wR3)^+SigC)^+KI,
```

```
        ((wM2, wID, C, wR2)^+SigB)^+KI,
        ((wM1, wID, B, wR1)^+SigA)^+KI,
        (wM0)^+KI, wDIC, wIS, wR3)^+KI >> (wID pkdecr (-SigI, wIS)) >>
        (wID is hsh(I)) >>
    i3?(M0, A, R) >>
        (wDIC is hsh(wM3, hsh(wM2, hsh(wM1, hsh(I))))) >>
        (wR3 is hsh(hsh(hsh(R)))) >>
        (wR2 is hsh(hsh(R))) >>
        (wR1 is hsh(R)) >>
        (wID is hsh(I)) >>
        (wM0 is M0) >>
        Accept!((A,M1), (B,M2), (C,M3)) >>   stop;

(*The process at the 1st executing host is declared as rA)
val rA = M1 new_in
        a1?(xM0, xIC, xIS, xR)^+KA >> (xID pkdecr (-SigI, xIS)) >>
            (xID is hsh(I)) >>
        a2!(((M1, xID, B, hsh(xR))^+SigA)^+KI,
              xM0, hsh(M1, xIC), xIS, hsh(xR))^+KB >> stop;


(*The process at the 2nd executing host is declared as rB)
val rB = M2 new_in
        b1?(yM1, yM0, yDIC, yIS, yR)^+KB >> (yID pkdecr (-SigI, yIS)) >>
            (yID is hsh(I)) >>
        b2!(((M2, yID, C, hsh(yR))^+SigB)^+KI,
              yM1, yM0, hsh(M2, yDIC), yIS, hsh(yR))^+KC >> stop;

(*The process at the 3rd executing host is declared as rC)
val rC = M3 new_in
        c1?(zM2, zM1, zM0, zDIC, zIS, zR)^+KC >> (zID pkdecr (-SigI, zIS))
            >>(zID is hsh(I)) >>
        c2!(((M3, zID, I, hsh(zR))^+SigC)^+KI,
              zM2, zM1, zM0, hsh(M3, zDIC), zIS, hsh(zR))^+KI >> stop;

(*The whole system is declared as: (1) the parallel composition of the role of hosts
participating in the protocol and their respective public keys; (2) a 'guardian' that can
detect if the environment learns some piece of sensible information, like y)
val Sys = KI new_in KA new_in KB new_in KC new_in iI || rA || rB || rC ||
          guard?y >> stop;

(*The initial configuration consists of: (1) the environment's initial knowledge where the
disclose channel leaks information to the environment, such as the public encryption
```

keys and signature verification keys of hosts participating in the protocol; (2) the role of the system *Sys*)
```
val Conf = ( [disclose!(Rold, I, A, B, C, +KI, +KA, +KB, +KC, -SigI, -SigA,
                -SigB, -SigC) ]@Sys);

(*Checks integrity of the nonce that uniquely identifies the protocol run)
val Auth1 = (a1?xR <-- a2!hsh(xR));
val Auth2 = (b1?yR <-- b2!hsh(yR));
val Auth3 = (c1?zR <-- c2!hsh(zR));

(*Checks the integrity of the two identifiers: (1) identifier of the genuine initiator; (2) identifier of the protocol run of concern, as the agent migrates through the output action of a preceding host and is received at the input action of a host, i.e. an intruder did not tamper with the two identifiers as the agent migrates through communication channels)
val Auth4 = (i1!(w4, w5, w6, w7) <-- a1?(w4, w5, w6', w7'));
val Auth5 = (a2!(w3, w4, w5, w6, w7) <-- b1?(w3, w4, w5, w6', w7'));
val Auth6 = (b2!(w2, w3, w4 ,w5, w6, w7) <-- c1?(w2, w3, w4, w5, w6', w7'));
val Auth7 = (c2!(w1, w2, w3, w4, w5 ,w6 ,w7)  <--  i2?(w1, w2, w3, w4, w5, w6', w7'));

(*Checks the integrity of the execution result M1 provided by host A)
val Auth8 = (a2!(w3, w4, w5, w6, w6, w7) <-- i2?(w1, w2, w3', w4, w5, w6, w7));

(*Checks the integrity of the execution result M2 provided by host B)
val Auth9 = (b2!(w2, w3, w4, w5, w6, w7) <-- i2?(w1, w2', w3, w4, w5, w6, w7));

(*Checks the integrity of the execution result M3 provided by host C)
val Auth10 = (c2!(w1, w2, w3, w4, w5, w6, w7) <-- i2?(w1', w2, w3, w4, w5, w6, w7));

(*Checks the integrity of the dummy data M0 generated at host I)
val Auth11 = (i1!(w4, w5, w6, w7) <-- i2?(w1, w2, w3, w4', w5, w6, w7));

(*Checks the integrity of the identifier of genuine host as the agent is executed at a host participating in the protocol, i.e a host does not tamper with the identifier)
val Auth12 = (a1?(w4, w5, w6, w7) <-- a2!(w3, w4, w5, w6', w7));
val Auth13 = (b1?(w3, w4, w5, w6, w7) <-- b2!(w2, w3, w4, w5, w6', w7));
val Auth14 = (c1?(w2, w3, w4, w5, w6, w7) <-- c2!(w1, w2, w3, w4, w5, w6', w7));

(*Checks the secrecy of the execution results M1, M2, M3, and the dummy data provided by hosts A, B, C, and I respectively)
val Secrecy1 = (Absurd <-- guard?M0);
val Secrecy2 = (Absurd <-- guard?M1);
val Secrecy3 = (Absurd <-- guard?M2);
val Secrecy4 = (Absurd <-- guard?M3);
```

Appendix B

The STA script of two parallel runs of the protocol with R, and R' as identifiers of the first and the second protocol runs respectively. The agent's itinerary in the first run is I, A, B, C, I, whereas the itinerary in the second run is I, E, B, C, I. Host E is a malicious host. The role of the host is not explicitly modeled.

(* A single run of the proposed protocol
I: initiator; A: 1st executing host, B: 2nd executing host
C: 3rd executing host, Adv: adversary, R: fresh none, Rold: old nonce
M0: dummy data generated by host I, M1: execution result at host A
M2: execution result at host B, M3: execution result at host C

Host I sends the mobile agent to gather information from hosts A, B, and C. At the end of the protocol, host I needs to analyze the integrity, confidentiality, and authenticity of the returned execution results M1, M2, and M3 acquired at hosts A, B, and C respectively.

Notations used in declaring identifiers are as given below.

i1!: output action at host I, i2?: input action at host I
i3?: input action at host I that follows the output action of the secondary agent $\mathcal{A}s$ which communicates the initial verification data to major agent $\mathcal{A}$
i'1!: output action at host I, i'2?: input action at host I
a1?: input action at host A, a2!: output action at host A
b1?: input action at host B, b2!: output action at host B
b'1?: input action at host B, b'2!: output action at host B
c1?: input action at host C, c2!: output action at host C
c'1?: input action at host C, c'2!: output action at host C
disclose!: an output action that leaks information to the environment
guard?: an input action 'guard' such that a guardian learns some secret data
accept1!: an output action that outputs the execution results which satisfy the intended security properties to host I upon agent's return
KA, KB, KC, KI: public keys of hosts A, B, C, and I respectively
SigA, SigB, sigC, SigI: private signing keys of hosts A, B, C, I respectively)

```
DeclLabel $ a1, a2, b1, b2, c1, c2, i1, i2, i3, i'3,
              i'1, i'2, b'1, b'2, c'1, c'2, disclose, guard, Accept1, Accept2 $;
DeclName $ Rold, R, SigI, SigA, SigB, SigC, SigE, I, A, B, C, E,
              M0, M1, M2, M3, KI, KA, KB, KC, KE, M'0, M'1, M'2, M'3, R' $ ;
DeclVar  $ xM0, yM0, yM1, zM2,zM1, zM0, wM3, wM2, wM1, wM0, wM'0, wM'1,
           wM'2, wM'3, y, w1, w2, w3, w4, w5, w6, w7, w7',
           w1', w2', w3', w4', w6', xIC, yIC, zIC, wIC, yDIC, zDIC, wDIC,
```

```
        xR, yR, zR, wR0, wR1, wR2, wR3, wM'3 , w'R3, wM'2, w'R2, wM'1,
        w'R1, w'DIC, yM'1, yM'0, y'DIC, y'R, zM'2, zM'1, zM'0, z'DIC, z'R, xIS,
        xID, yIS, yID, zIS, zID, wID, wIS $;

(*The process at the initiator is declared as iI)
val iI = M0 new_in R new_in
        i1!((M0)^+KI, hsh (I), (hsh (I))^+SigI, R)^+KA >>
        i2?(((wM3, wID, I, wR3)^+SigC)^+KI,
            ((wM2, wID, C, wR2)^+SigB)^+KI,
            ((wM1, wID, B, wR1)^+SigA)^+KI,
            (wM0)^+KI, wDIC, wIS, wR3)^+KI >> (wID pkdecr (-SigI, wIS))
            >>(wID is hsh(I)) >>
        i3?(M0, A, R) >>
            (wDIC is hsh(wM3, hsh(wM2, hsh(wM1, hsh(I))))) >>
            (wR3 is hsh(hsh(hsh(R)))) >>
            (wR2 is hsh(hsh(R))) >>
            (wR1 is hsh(R)) >>
            (wID is hsh(I)) >>
            (wM0 is M0) >>
        Accept1!((A,M1), (B,M2), (C,M3)) >> stop
        ||
        M'0 new_in R' new_in
        i'1!((M'0)^+KI, hsh(I), (hsh(I))^+SigI, R')^+KE >>
        i'2?(((wM'3, wID, I, w'R3)^+SigC)^+KI,
             ((wM'2, wID, C, w'R2)^+SigB)^+KI,
             ((wM'1, wID, B, w'R1)^+SigE)^+KI,
             (wM'0)^+KI, w'DIC, wIS, w'R3)^+KI >> stop ;

(*The process at the 1st executing host is declared as rA)
val rA = M1 new_in
          a1?(xM0, xIC, xIS, xR)^+KA >> (xID pkdecr (-SigI, xIS)) >>
              (xID is hsh(I)) >>
          a2!(((M1, xIC, B, hsh(xR))^+SigA)^+KI, xM0,
              hsh(M1, xIC), xIS, hsh(xR))^+KB >> stop;

(*The process at the 2nd executing host is declared as rB)
val rB = M2 new_in
          b1?(yM1, yM0, yDIC, yIS, yR)^+KB >> (yID pkdecr (-SigI, yIS)) >>
              (yID is hsh(I)) >>
          b2!(((M2, yID, C, hsh(yR))^+SigB)^+KI,
                yM1, yM0, hsh(M2, yDIC), yIS, hsh(yR))^+KC >> stop
          ||
          M'2 new_in
```

```
        b'1?(yM'1, yM'0, y'DIC, yIS, y'R)^+KB >> (yID pkdecr (-SigI, yIS)) >>
             (yID is hsh(I)) >>
        b'2!(((M'2, yID, C, hsh(y'R))^+SigB)^+KI,
                yM'1, yM'0, hsh(M'2, y'DIC), yIS, hsh(y'R))^+KC >> stop;

(*The process at the 3rd executing host is declared as rC)
val rC = M3 new_in
          c1?(zM2, zM1, zM0, zDIC, zIS, zR)^+KC >> (zID pkdecr (-SigI, zIS))
           >>(zID is hsh(I)) >>
          c2!(((M3, zID, I, hsh(zR))^+SigC)^+KI,
                zM2, zM1, zM0, hsh(M3, zDIC), zIS, hsh(zR))^+KI >> stop
          ||
          M'3 new_in
           c'1?(zM'2, zM'1, zM'0, z'DIC, zIS, z'R)^+KC >> (zID pkdecr (-SigI, zIS))
            >>(zID is hsh(I)) >>
           c'2!(((M'3, zID, I, hsh(z'R))^+SigC)^+KI,
                  zM'2, zM'1, zM'0, hsh(M'3, z'DIC), zIS, hsh(z'R))^+KI >> stop;

(*The whole system is declared as: (1) the parallel composition of the role of hosts
participating in the protocol and their respective public keys; (2) a 'guardian' that can
detect if the environment learns some piece of sensible information, like y)
val Sys = KI new_in KA new_in KB new_in KC new_in iI || rA || rB || rC ||
          guard?y >> stop;

(*The initial configuration consists of: (1) the environment's initial knowledge where the
disclose channel leaks information to the environment, such as the public encryption
keys and signature verification keys of hosts participating in the protocol; (2) the role of
the system Sys)
val Conf = ( [disclose!(Rold, I, A, B, C, +KI, +KA, +KB, +KC, KE, -SigE, -SigI,
               -SigA,  -SigB, -SigC) ]@Sys);

(*Checks integrity of the nonce that uniquely identifies the protocol run)
val Auth1 = (a1?xR <-- a2!hsh(xR));
val Auth2 = (b1?yR <-- b2!hsh(yR));
val Auth3 = (c1?zR <-- c2!hsh(zR));

(*Checks the integrity of the two identifiers: (1) identifier of the genuine initiator; (2)
identifier of the protocol run of concern, as the agent migrates through the output action
of a preceding host and is received at the input action of a host, i.e. an intruder did not
tamper with the two identifiers as the agent migrates through communication channels)
val Auth4 = (i1!(w4, w5, w6, w7) <-- a1?(w4, w5, w6', w7'));
val Auth5 = (a2!(w3, w4, w5, w6, w7) <-- b1?(w3, w4, w5, w6', w7'));
val Auth6 = (b2!(w2, w3, w4 ,w5, w6, w7) <-- c1?(w2, w3, w4, w5, w6', w7'));
```

```
val Auth7 = (c2!(w1, w2, w3, w4, w5 ,w6 ,w7) <-- i2?(w1, w2, w3, w4, w5, w6', w7'));

(*Checks the integrity of the execution result M1 provided by host A)
val Auth8 = (a2!(w3, w4, w5, w6, w6, w7) <-- i2?(w1, w2, w3', w4, w5, w6, w7));

(*Checks the integrity of the execution result M2 provided by host B)
val Auth9 = (b2!(w2, w3, w4, w5, w6, w7) <-- i2?(w1, w2', w3, w4, w5, w6, w7));

(*Checks the integrity of the execution result M3 provided by host C)
val Auth10 = (c2!(w1, w2, w3, w4, w5, w6, w7) <-- i2?(w1', w2, w3, w4, w5, w6, w7));

(*Checks the integrity of the dummy data M0 generated at host I)
val Auth11 = (i1!(w4, w5, w6, w7) <-- i2?(w1, w2, w3, w4', w5, w6, w7));

(*Checks the integrity of the identifier of genuine host as the agent is executed at a host
participating in the protocol, i.e a host does not tamper with the identifier)
val Auth12 = (a1?(w4, w5, w6, w7) <-- a2!(w3, w4, w5, w6', w7));
val Auth13 = (b1?(w3, w4, w5, w6, w7) <-- b2!(w2, w3, w4, w5, w6', w7));
val Auth14 = (c1?(w2, w3, w4, w5, w6, w7) <-- c2!(w1, w2, w3, w4, w5, w6', w7));

(*Checks the secrecy of the execution results M1, M2, M3, and the dummy data
provided by hosts A, B, C, and I respectively)
val Secrecy1 = (Absurd <-- guard?M0);
val Secrecy2 = (Absurd <-- guard?M1);
val Secrecy3 = (Absurd <-- guard?M2);
val Secrecy4 = (Absurd <-- guard?M3);
```

Appendix C

*(*The STA script of two parallel runs of the protocol with R, and R' as identifiers of the first and the second protocol runs respectively. The agent's itinerary in the first run is I, A, B, C, I, whereas the itinerary in the second run is I, E, B, C, I. Hosts A and C are two malicious hosts, which co-operate with each other, in the first run and host E is a malicious host in the second run. The roles of the malicious hosts A, C, and E are not explicitly modeled.*

Notations used in declaring identifiers are as given below.

i1!: output action at host I, i2?: input action at host I
i3?: input action at host I that follows the output action of the secondary agent $\mathcal{A}s$ which communicates the initial verification data to major agent $\mathcal{A}$
b1?: input action at host B, b2!: output action at host B
disclose!: an output action that leaks information to the environment
guard?: an input action 'guard' such that a guardian learns some secret data
accept1!: an output action that outputs the execution results which satisfy the intended security properties to host I upon agent's return
KA, KB, KC, KI: public keys of hosts A, B, C, and I respectively
SigA, SigB, sigC, SigI: private signing keys of hosts A, B, C, I respectively)

```
DeclLabel $   b1, b2, i1, i2, i3, i'1, i'2, b'1, b'2, disclose, guard, Accept1 $;
DeclName  $  Rold, R, SigI, SigA, SigB, SigC, SigE, I, A, B, C, E,
             M0, M1, M2, M3, KI, KA, KB, KC, KE, M'0, M'1, M'2, M'3, R', IC $ ;
DeclVar   $   xM0, yM0, yM1, zM2,zM1, zM0, wM3, wM2, wM1, wM0, wM'0,
             wM'1, wM'2, wM'3, y, w1, w2, w3, w4, w5, w6, w7, w7',
             w1', w2', w3', w4', w6', xIC, yIC, zIC, wIC, yDIC, zDIC, wDIC,
             xR, yR, zR, wR0, wR1, wR2, wR3, y, wM'3 , w'R3, w'R2,
             w'R1, w'DIC, yM'1, yM'0, y'DIC, y'R, zM'2, zM'1, zM'0, z'DIC,
             z'R, xIS, xID, yIS, yID, zIS, zID, wID, wIS $;

(*The process at the initiator is declared as iI)
val iI =   M0 new_in R new_in
      i1! ((M0)^+KI, hsh (I), (hsh (I))^+SigI, R)^+KA >>
      i2? (((wM3, wID, I, wR3)^+SigC)^+KI,
          ((wM2, wID, C, wR2)^+SigB)^+KI,
          ((wM1, wID, B, wR1)^+SigA)^+KI,
           (wM0)^+KI, wDIC, wIS, wR3)^+KI >> (wID pkdecr (-SigI, wIS)) >>
          (wID is hsh(I)) >>
      i3? (M0, A, R) >>
          (wDIC is hsh(wM3, hsh(wM2, hsh(wM1, hsh(I))))) >>
          (wR3 is hsh(hsh(hsh(R)))) >>
```

```
            (wR2 is hsh(hsh(R))) >>
            (wR1 is hsh(R)) >>
            (wID is hsh(I)) >>
            (wM0 is M0) >>
        Accept1!((A,M1), (B,M2), (C,M3)) >> stop
            ||
            M'0 new_in R' new_in
         i'1!((M'0)^+KI, hsh(I), (hsh(I))^+SigI, R')^+KE >>
         i'2?(((wM'3, wID, I, w'R3)^+SigC)^+KI,
            ((wM'2, wID, C, w'R2)^+SigB)^+KI,
            ((wM'1, wID, B, w'R1)^+SigE)^+KI,
            (wM'0)^+KI, w'DIC, wIS, w'R3)^+KI >> stop ;

(*The process at the 2nd executing host is declared as rB)
val rB =  M2 new_in
        b1?(yM1, yM0, yDIC, yIS, yR)^+KB >> (yID pkdecr (-SigI, yIS)) >>
             (yID is hsh(I)) >>
        b2! (((M2, yID, C, hsh(yR))^+SigB)^+KI,
             yM1, yM0, hsh(M2, yDIC), yIS, hsh(yR))^+KC >> stop
            ||
            M'2 new_in
       b'1?(yM'1, yM'0, y'DIC, yIS, y'R)^+KB >> (yID pkdecr (-SigI, yIS)) >>
             (yID is hsh(I)) >>
       b'2!(((M'2, yID, C, hsh(y'R))^+SigB)^+KI,
            yM'1, yM'0, hsh(M'2, y'DIC), yIS, hsh(y'R))^+KC >> stop;

(*The whole system is declared as: (1) the parallel composition of the role of hosts
participating in the protocol and their respective public keys; (2) a 'guardian' that can
detect if the environment learns some piece of sensible information, like y)
val Sys = KI new_in KA new_in KB new_in KC new_in iI || rB || guard?y >> stop;

(*The initial configuration consists of: (1) the environment's initial knowledge where the
disclose channel leaks information to the environment, such as the public encryption
keys and signature verification keys of hosts participating in the protocol; (2) the role of
the system Sys)
val Conf = ([disclose!(Rold, I, A, B, C, E, +KI, +KA, +KB, +KC, KE, -SigE,
              -SigI, -SigA, -SigB, -SigC) ]@Sys);

(*Checks the integrity of the nonce that uniquely identifies the protocol run)
val Auth1 = (b1?yR <-- b2!hsh(yR));

(*Checks the integrity of the execution result M2 provided by host B)
val Auth2 = (b2!(w2, w3, w4, w5, w6, w7) <-- i2?(w1, w2', w3, w4, w5, w6, w7));
```

```
(*Checks the integrity of the dummy data M0 generated at host I)
val Auth3 = (i1!(w4, w5, w6, w7) <-- i2?(w1, w2, w3, w4', w5, w6', w7));

(*Checks the integrity of the identifier of genuine host as the agent is executed at a host
participating in the protocol, i.e a host does not tamper with the identifier)
val Auth4 = (b1?(w3, w4, w5, w6, w7) <-- b2!(w2, w3, w4, w5, w6', w7));

(*Checks the secrecy of the execution result M0 and the dummy data provided by hosts
B, and I respectively)
val Secrecy1 = (Absurd <-- guard?M0);
val Secrecy2 = (Absurd <-- guard?M2);
```

Appendix D

*(*The STA script of two parallel runs of the protocol. The first run is initiated by host I1, and the second run is initiated by host I2. The first run is identified by R and the second is identified by R'. The agent's itinerary in the first run is I1, A, B, C, I1, whereas the itinerary in the second run is I2, E, B, C, I2.*

Notations used in declaring identifiers are as given below.

i1!: output action at host I1, i2?: input action at host I1
i3?: input action at host I1 that follows the output action of the secondary agent $\mathcal{A}s$ which communicates the initial verification data to major agent $\mathcal{A}$
i'1!: output action at host I2, i'2?: input action at host I2
a1?: input action at host A, a2!: output action at host A
a'1?: input action at host A, a'2!: output action at host A
b1?: input action at host B, b2!: output action at host B
b'1?: input action at host B, b'2!: output action at host B
c1?: input action at host C, c2!: output action at host C
c'1?: input action at host C, c'2!: output action at host C
disclose!: an output action that leaks information to the environment
guard?: an input action 'guard' such that a guardian learns some secret data
accept1!: an output action that outputs the execution results which satisfy the intended security properties to host I1 upon agent's return
KA, KB, KC, KI1, KI2: public keys of hosts A, B, C, and I1, and I2, respectively
SigA, SigB, sigC, SigI, SigI: private signing keys of hosts A, B, C, I1, and I2 respectively)

```
DeclLabel $   a1, a2, a'1, a'2, b1, b2, c1, c2, i1, i2, i3,
              i'1, i'2, b'1, b'2, c'1, c'2, disclose, guard, Accept1 $;
DeclName  $   Rold, R, SigI1, SigI2, SigA, SigB, SigC, I1, I2, A, B, C,
              M0, M1, M2, M3, KI1, KI2, KA, KB, KC, KE, M'0, M'1, M'2, M'3, R',
              IC $ ;
DeclVar   $   xM0, yM0, yM1, zM2,zM1, zM0, wM3, wM2, wM1, wM0, wM'0,
              wM'1, wM'2, wM'3, y, w1, w2, w3, w4, w5, w6, w7, w7',
              w1', w2', w3', w4', w6', xIC, yIC, zIC, wIC, yDIC, zDIC, wDIC,
              xR, yR, zR, wR0, wR1, wR2, wR3,
              wM'3 , w'R3, wM'2, w'R2, wM'1, w'R1, w'DIC, yM'1, yM'0, y'DIC,
              y'R, zM'2, zM'1, zM'0, z'DIC, z'R, x'IC, x'IS, x'R, xM'0, x'ID,
              x'IS, M'1, y'IS, y'ID, y'IS, y'R, M'2, z'IS, z'ID, M'3, w'ID,
              wM'0, w'IS, xIS, xID, yIS, yID, zIS, zID, wID, wIS $;

(*The process at the initiator is declared as iI1)
val iI1=   M0 new_in R new_in
           i1!((M0)^+KI1, hsh (I1), (hsh (I1))^+SigI1, R)^+KA >>
```

```
            i2?(((wM3, wID, I1, wR3)^+SigC)^+KI1,
                ((wM2, wID, C, wR2)^+SigB)^+KI1,
                ((wM1, wID, B, wR1)^+SigA)^+KI1,
                (wM0)^+KI1, wDIC, wIS, wR3)^+KI1 >> (wID pkdecr (-SigI1, wIS))
                 >>(wID is hsh(I1)) >>
            i3?(M0, A, R) >>
                (wDIC is hsh(wM3, hsh(wM2, hsh(wM1, hsh(I1))))) >>
                (wR3 is hsh(hsh(hsh(R)))) >>
                (wR2 is hsh(hsh(R))) >>
                (wR1 is hsh(R)) >>
                (wID is hsh(I1)) >>
                (wM0 is M0) >>
            Accept1!((A,M1), (B,M2), (C,M3)) >> stop;

(*The process at the initiator is declared as iI2)
val iI2=  M'0 new_in R' new_in
            i'1!((M'0)^+KI2, hsh(I2), (hsh(I2))^+SigI2, R')^+KA >>
            i'2?(((wM'3, w'ID, I2, w'R3)^+SigC)^+KI2,
                ((wM'2, w'ID, C, w'R2)^+SigB)^+KI2,
                ((wM'1, w'ID, B, w'R1)^+SigA)^+KI2,
                (wM'0)^+KI2, w'DIC, w'IS, w'R3)^+KI2 >> stop ;

(*The process at the 1st executing host is declared as rA)
val rA =  M1 new_in
            a1?(xM0, xIC, xIS, xR)^+KA >> (xID pkdecr (-SigI1, xIS)) >>
                (xID is hsh(I1)) >>
            a2!(((M1, xIC, B, hsh(xR))^+SigA)^+KI1, xM0,
                hsh(M1, xIC), xIS, hsh(xR))^+KB >> stop
            ||
            M'1 new_in
            a'1?(xM'0, x'IC, x'IS, x'R)^+KA >> (x'ID pkdecr (-SigI2, x'IS)) >>
                (x'ID is hsh(I2)) >>
            a'2!(((M'1, x'IC, B, hsh(x'R))^+SigA)^+KI2, xM'0,
                hsh(M'1, x'IC), x'IS, hsh(x'R))^+KB >> stop;

(*The process at the 2nd executing host is declared as rB)
val rB =  M2 new_in
            b1?(yM1, yM0, yDIC, yIS, yR)^+KB >> (yID pkdecr (-SigI1, yIS)) >>
                (yID is hsh(I1)) >>
            b2!(((M2, yID, C, hsh(yR))^+SigB)^+KI,
                yM1, yM0, hsh(M2, yDIC), yIS, hsh(yR))^+KC >> stop
            ||
            M'2 new_in
```

```
        b'1?(yM'1, yM'0, y'DIC, y'IS, y'R)^+KB >> (y'ID pkdecr (-SigI2, y'IS))
             >>(y'ID is hsh(I2)) >>
        b'2!(((M'2, y'ID, C, hsh(y'R))^+SigB)^+KI2,
             yM'1, yM'0, hsh(M'2, y'DIC), y'IS, hsh(y'R))^+KC >> stop;

(*The process at the 3rd executing host is declared as rC)
val rC =  M3 new_in
        c1?(zM2, zM1, zM0, zDIC, zIS, zR)^+KC >> (zID pkdecr (-SigI1, zIS))
             >>(zID is hsh(I1)) >>
        c2!(((M3, zID, I1, hsh(zR))^+SigC)^+KI1,
             zM2, zM1, zM0, hsh(M3, zDIC), zIS, hsh(zR))^+KI1 >> stop
        ||
        M'3 new_in
        c'1?(zM'2, zM'1, zM'0, z'DIC, z'IS, z'R)^+KC >>
             (z'ID pkdecr (-SigI2, z'IS)) >>(z'ID is hsh(I2)) >>
              c'2!(((M'3, z'ID, I2, hsh(z'R))^+SigC)^+KI2,
              zM'2, zM'1, zM'0, hsh(M'3, z'DIC), z'IS, hsh(z'R))^+KI2 >> stop;


(*The whole system is declared as: (1) the parallel composition of the role of hosts
participating in the protocol and their respective public keys; (2) a 'guardian' that can
detect if the environment learns some piece of sensible information, like y)
val Sys = KI new_in KA new_in KB new_in KC new_in iI1 || iI2 || rA || rB || rC ||
          guard?y >> stop;

(*The initial configuration consists of: (1) the environment's initial knowledge where the
disclose channel leaks information to the environment, such as the public encryption
keys and signature verification keys of hosts participating in the protocol; (2) the role of
the system Sys)
val Conf = ([disclose!(Rold, I1, I2, A, B, C, +KI1, KI2, +KA, +KB, +KC, KE, -SigI1,
            -SigI2, -SigA, -SigB, -SigC) ]@Sys);

(*Checks integrity of the nonce that uniquely identifies the protocol run)
val Auth1 = (a1?xR <-- a2!hsh(xR));
val Auth2 = (b1?yR <-- b2!hsh(yR));
val Auth3 = (c1?zR <-- c2!hsh(zR));

(*Checks the integrity of the two identifiers: (1) identifier of the genuine initiator; (2)
identifier of the protocol run of concern, as the agent migrates through the output action
of a preceding host and is received at the input action of a host, i.e. an intruder did not
tamper with the two identifiers as the agent migrates through communication channels)
val Auth4 = (i1!(w4, w5, w6, w7) <-- a1?(w4, w5, w6', w7'));
val Auth5 = (a2!(w3, w4, w5, w6, w7) <-- b1?(w3, w4, w5, w6', w7'));
```

```
val Auth6 = (b2!(w2, w3, w4 ,w5, w6, w7) <-- c1?(w2, w3, w4, w5, w6', w7'));
val Auth7 = (c2!(w1, w2, w3, w4, w5 ,w6 ,w7)  <--  i2?(w1, w2, w3, w4, w5, w6', w7'));

(*Checks the integrity of the execution result M1 provided by host A)
val Auth8 = (a2!(w3, w4, w5, w6, w6, w7) <-- i2?(w1, w2, w3', w4, w5, w6, w7));

(*Checks the integrity of the execution result M2 provided by host B)
val Auth9 = (b2!(w2, w3, w4, w5, w6, w7) <-- i2?(w1, w2', w3, w4, w5, w6, w7));

(*Checks the integrity of the execution result M3 provided by host C)
val Auth10 = (c2!(w1, w2, w3, w4, w5, w6, w7) <-- i2?(w1', w2, w3, w4, w5, w6, w7));

(*Checks the integrity of the dummy data M0 generated at host I)
val Auth11 = (i1!(w4, w5, w6, w7) <-- i2?(w1, w2, w3, w4', w5, w6, w7));

(*Checks the integrity of the identifier of genuine host as the agent is executed at a host
participating in the protocol, i.e a host does not tamper with the identifier)
val Auth12 = (a1?(w4, w5, w6, w7) <-- a2!(w3, w4, w5, w6', w7));
val Auth13 = (b1?(w3, w4, w5, w6, w7) <-- b2!(w2, w3, w4, w5, w6', w7));
val Auth14 = (c1?(w2, w3, w4, w5, w6, w7) <-- c2!(w1, w2, w3, w4, w5, w6', w7));

(*Checks the secrecy of the execution results M1, M2, M3, and the dummy data
provided by hosts A, B, C, and I respectively)
val Secrecy1 = (Absurd <-- guard?M0);
val Secrecy2 = (Absurd <-- guard?M1);
val Secrecy3 = (Absurd <-- guard?M2);
val Secrecy4 = (Absurd <-- guard?M3);

-
(*Checks integrity of the nonce that uniquely identifies the protocol run)
val Auth1 = (a1?xR <-- a2!hsh(xR));
val Auth2 = (b1?yR <-- b2!hsh(yR));
val Auth3 = (c1?zR <-- c2!hsh(zR));

(*Checks the integrity of the two identifiers: (1) identifier of the genuine initiator; (2)
identifier of the protocol run of concern, as the agent migrates through the output action
of a preceding host and is received at the input action of a host, i.e. an intruder did not
tamper with the two identifiers as the agent migrates through communication channels)
val Auth5 = (i1!(w4, w5, w6, w7) <-- a1?(w4, w5, w6', w7'));
val Auth6 = (a2!(w3, w4, w5, w6, w7) <-- b1?(w3, w4, w5, w6', w7'));
val Auth7 = (b2!(w2, w3, w4 ,w5, w6, w7) <-- c1?(w2, w3, w4, w5, w6', w7'));

(*Checks the integrity of the execution result M1 provided by host A)
```

```
val Auth8 = (c2!(w1, w2, w3, w4, w5 ,w6 ,w7) <-- i2?(w1, w2, w3, w4, w5, w6', w7'));

(*Checks the integrity of the execution result M2 provided by host B)
val Auth9 = (a2!(w3, w4, w5, w6, w6, w7) <-- i2?(w1, w2, w3', w4, w5, w6, w7));

(*Checks the integrity of the execution result M3 provided by host C)
val Auth10 = (b2!(w2, w3, w4, w5, w6, w7) <-- i2?(w1, w2', w3, w4, w5, w6, w7));

(*Checks the integrity of the dummy data M0 generated at host I)
val Auth11 = (c2!(w1, w2, w3, w4, w5, w6, w7) <-- i2?(w1', w2, w3, w4, w5, w6, w7));

(*Checks the integrity of the identifier of genuine host as the agent is executed at a host
participating in the protocol, i.e a host does not tamper with the identifier)
val Auth12 = (i1!(w4, w5, w6, w7) <-- i2?(w1, w2, w3, w4', w5, w6', w7));
val Auth14 = (a1?(w4, w5, w6, w7) <-- a2!(w3, w4, w5, w6', w7));
val Auth15 = (b1?(w3, w4, w5, w6, w7) <-- b2!(w2, w3, w4, w5, w6', w7));
val Auth16 = (c1?(w2, w3, w4, w5, w6, w7) <-- c2!(w1, w2, w3, w4, w5, w6', w7));

val Secrecy1 = (Absurd <-- guard?M0);
val Secrecy2 = (Absurd <-- guard?M1);
val Secrecy3 = (Absurd <-- guard?M2);
val Secrecy4 = (Absurd <-- guard?M3);
```