# Formal methods in the Enhancement of The Data Security Protocols of 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-0520**

**December 2005**

**THE UNIVERSITY OF NEW SOUTH WALES**

## Abstract

The security of data gathered by mobile agents is crucial to the success of e-commerce and formal methods play an important role in the verification of the data security protocols. This paper demonstrates the effectiveness of formal methods in the analysis and design of security protocols. In this paper, we implement a formal method in the analysis and the rectification of a recent mobile agent security protocol by Maggi and Sisto [14] named "*Configurable Mobile Agent Data Protection Protocol*". We use STA (Symbolic Trace Analyzer), a formal verification tool that is based on symbolic techniques, in the analysis of the protocol. The analysis revealed a flaw in the security protocol. An adversary can impersonate the genuine initiator, and thus can breach the privacy of the gathered data. The protocol does not detect the malicious act. In addition [14] states that the protocol does not achieve strong data integrity in case two hosts conspire or a malicious host is visited twice. We rectify the protocol so it prevents the malicious acts, and then use STA tool to analyze a reasonably small instance of the protocol in key configurations. The analysis shows that the repaired protocol is free of flaws. Moreover, we reason about the security of a general model of the repaired protocol. To our knowledge, we are the first to repair the protocol, and analyze it formally.

**Keywords**: formal methods, security specifications, security protocols, formal analysis and verification, mobile agents.

## 1. Introduction

Mobile agents are autonomous programs that have one or more goals. They traverse the Internet and are expected to run in heterogeneous environments. 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 vulnerable to direct security threats as they transfer through insecure channels and are expected to execute on non-trusted hosts, which would result in disclosure of sensitive information, erroneous credits, truncation of the gathered data, etc. Several cryptographic protocols were presented in the literature asserting the security of data gathered by mobile agents [8, 16].

Formal methods have been commonly used in reasoning about the correctness of security protocols [1, 3, 9, 15, 16]. The formal methods help in developing error-free security protocols. The verification of the existing protocols using formal methods has revealed subtle flaws in the protocols and showed their failure to accomplish some or all of the asserted properties [3, 9, 12, 15, 19]. Intruders and malicious hosts were able to spy out data, force unjust authenticity, truncate the gathered data, etc. The flaws were unforeseen at the design stage.

In this paper, we argue the correctness of the "Configurable Mobile Agent Data Protection Protocol" [14] using STA, a formal verification tool. The protocol aims to protect the data gathered by mobile agents from malicious acts of intruders and malicious hosts. The protection is based on binding the static part to the dynamic part of a mobile agent, and forcing the agent to securely store the addresses of next hosts to be visited. The Symbolic Trace Analyzer (STA) is an infinite-state exploration tool that analyzes the security properties of protocols based on symbolic techniques. It models a protocol as a system of concurrent processes, using a dialect of the Spi-calculus [10].

The analysis of the protocol using STA reveals that an adversary can impersonate the genuine initiator, hence can breach the privacy of the gathered data. Maggi and Sisto state that a malicious host 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 attack would not be detected by formal analysis. We repair the protocol by implementing the following security techniques: (a) utilizing two co-operating agents, (b) applying additional cryptographic functions, (c) carrying out verifications at the early execution of the agent at a visited host on the identity of the genuine initiator, and (d) requesting the current executing host to clear its memory from any data acquired as a result of executing the agent, which would lead to the prevention of data truncation and data alteration. Next, we analyze the protocol using STA that shows that the repaired protocol is free of flaws, particularly, the flaw revealed in the original protocol using STA.

The rest of the paper is organized as follows. Section 2 outlines the advantages of using the formal methods of verification and their applicability to the verification of mobile agent data protection protocols, and describes the STA tool. In section 3, we describe the protocol and define the claimed security properties. In section 4, we use STA to model the protocol, specify its properties, analyze the protocol formally, and show the flaw revealed in the protocol. In section 5, we repair the protocol and show that the repaired version is free of the flaws. In addition, we reason the correctness of a general model of the repaired protocol. In section 6, we summarize our contributions in a conclusion and discuss future directions of this work.

## 2. STA and Related Formal Methods

Formal methods have played an important role in specifying, verifying and revealing unforeseen flaws in security protocols. They have the following capabilities as summarized by Mai and Tsai [9]: (i) Characterize the system specifications precisely. (ii) Define accurately the desired security properties. (iii) Set clearly the interaction between the system and its environment. (iv) Identify security flaw/s in protocols if any exists. (v) Provide systematic and exhaustive analysis of protocols.

(vi) Provide a proof of security, if a system meets the desired properties. (vii) 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 [9]:

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

The formal methods have been successfully employed in the verification of the security properties of the classical message-based protocols, such as the authentication protocols, though the specification and verification of the security of mobile agent paradigms deal with new aspects namely locations and mobility as well as cryptography. There have been good advances in expressing mobility using process algebra [2, 6, 10, 11]. Some experiments used existing formal methods to verify the data protection protocols of mobile agents. The data integrity properties of mobile agents are verified using CSP-based tools Casper and FDR in [19], the security properties of mobile agents are analyzed using STA tool in [16], and a model checker that is based on symbolic data representation and uses Spi-calculus in [15]. In this paper, we utilize STA to analyze the "Configurable Mobile Agent Data Protection Protocol", which is specifically designed to detect attacks on the data gathered by mobile agents. STA [12, 17, 18] is a tool for automatic analysis of security protocols. It is a recent approach that takes advantage of the concepts derived from process calculi. It describes a protocol using process algebra. It expresses the security properties as traces the protocol generates and analyzes the execution traces of the protocol. The trace is a sequence of I/O actions that results from interaction between a process and its environment. The model is close in spirit to the Dolev-Yao model [20], where an intruder may intercept, delete, insert, append messages, or spy out confidential data. The Dolev-Yao model gives rise to infinitely execution traces due to the unbounded capabilities of intruders. STA implements a verification method based on symbolic relation $\rightarrow_s$ that avoids the state explosion problem induced by infinite execution traces. The symbolic model is finite, because each input action should be preceded by the corresponding action, for every generated trace. STA performs a complete exploration and verification of all possible execution traces [11, 12]. STA detects the flaws in Needham-Schroeder, Yahalom, Otway-Rees, and Kerberos protocols [11, 18].

We use STA in the analysis as it is characterized by the following: (i) It avoids state explosion problem as compared to model checking methods [12, 13]. Model checkers analyze systems by searching for an insecure state starting from an initial state, which may result in a search of infinite transitions. STA reduces the infinite transitions to a single symbolic relation, where every input action should be preceded by a corresponding output action. (ii) It does not require expert guidance as compared to theorem proving methods [7]. (iii) It does not need to model the intruder explicitly as compared to model checking methods [5, 11, 12]. It is sufficient to specify the intruder's initial knowledge such as public keys of participating hosts. Actually, modeling the intruder is often relatively complicated and time consuming, due to the unlimited capabilities and unpredictable behavior of the intruder. (iv) It does not require hand-written proofs as compared to methods based on process algebra or modal logic [7]. The analysis of Needham-Shroeder and Kerberos protocols with STA in [18] and with Murφ in [5] shows advantages of using the infinite-state exploration methods over the finite-state exploration methods. Modeling a security protocol [12, 13] in a finite-state model checker requires considering a reasonably sized finite state system to avoid the state explosion problem. It is necessary to pose limitations on the number of parallel runs of the protocol, and select a finite set of possible messages the intruder can generate and send to trusted hosts participating in the protocol [13]. Conversely, symbolic methods such as STA do not require any assumptions on the possible messages the intruder can generate. It is sufficient to specify the intruder's initial knowledge [13]. The symbolic methods implement symbolic transition relation. The relation reduces infinite standard transitions to a single symbolic relation, where every instance of an input/output action should be preceded by the corresponding input/output action.

According to Boreale and Buscemi [12], a protocol in STA is modeled as the parallel composition of the processes of honest hosts participating in the protocol. 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 behaviors of honest participants.

The transition between configurations represents interactions between s and *P*. It takes the form of (s, *P*) → (s′, *P*′), where → denotes the transition relation. The STA analyzes the execution traces of the system to detect possible faults of security properties of the protocol. Security properties are expressed in terms of the traces the protocol generates, particularly correspondence assertions. Given a configuration (s, *P*) and a trace s′, if action β occurs in the trace, then action α must have occurred earlier in the trace. The correspondence assertion is written as (α ⇜⇝β). Authentication 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 ⊥. The action in the formula ⊥⇜⇝ α means that an action α should never take place. The secrecy of some sensitive data *d* would be expressed by considering action α as a guardian *guard* that can at any time intercept a message *x* from the network. Nevertheless, it would not deduce the sensitive data *d* from *x.* 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 STA, the protocol can be specified using four kinds of declarations: identifiers, processes, configurations, and properties as described below. Table 1 summarizes STA declarations.

▪ *Identifiers* are names, variables, and labels. A name can be an encryption key, a host identity, or the data provided by a host. A variable is a received term, which might differ from the original transmitted term due to malicious acts of adversaries. A label is a name of an input/ output action. Declaration of identifiers should conform to the following rules: Names must begin with a capital letter, variables must begin with one of the letters u, x, y, w or z, and the rest of letters are for labels.

▪ *Process P* is the intended behaviors of honest participants. The declaration of a participating host named *Pr* would be such as val *Pr* = *P.*

▪ *Configuration* is the pair ⟨s, *P*⟩, where s is the current knowledge of an intruder and *P* is a description of processes at honest participants. The declaration is in the form of: val *Conf* = (*L* @ *Pr*), where *L* is the initial environment's knowledge representing the intruder's initial knowledge, *Pr* is a process or a parallel composition of processes which are previously declared.

▪ *Properties* are expressed as a correspondence assertion, i.e. in every execution trace of the protocol, every occurrence of an action should be preceded by the occurrence of the corresponding action. A property would be declared as follows: val *Prop* = (*A* ← *B*), where *Prop* is a property name such as *Auth*. The *A* and *B* are I/O actions. Suppose a host receives a variable *wX1* through the input action *b1*, then the authentication of *wX1* requires that the action should be preceded by the output action *a2* of the name *X1*. The declaration would be: val *Auth* = (*a2*!*X1* ← *b1*?*wX1*).

**Table 1. STA declarations**

| | |
|---|---|
| Identifiers | |
| *Names* | 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); |

The syntax of STA is depicted in Table 2. It follows closely the syntax of Spi-calculus [10] with a few minor differences.

**Table 2. Syntax of STA**

| | |
|---|---|
| A! M | Output action |
| A? x | Input action |
| Stop | Terminated process |
| >> | Sequence of actions |
| K new_in | Fresh name K |
| P1 \|\| P2 | Parallel composition of the processes P1 and P2 |
| (M)^+K | Encrypting the term M with the public key K |
| (M)^+sigK | Signing digitally the term M with the signing key sigK |
| (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) | Decrypting the message M with the private key –K, then binding the result to variable x |
| M pkdecr(-sigK, x) | Decrypting the message M with the signature verification key –sigK, then binding the result to variable x |

The command used to verify a property is in the form of: CHECK *Conf Prop*, where *Conf* is the configuration of the system; *Prop* is the property to check.

## 3. Protocol Description and Security Properties

The "Configurable Mobile Agent Data Protection Protocol" asserts a combination of security properties of the data gathered by mobile agents. The security properties [8, 14] are: data secrecy, data authenticity, data non-repudiatibility, strong data integrity, and origin confidentiality. The protocol can be configured to meet a particular combination of security properties. The protocol is based on a basic message with a set of fields that can be configured according to the required properties.

The protocol considers a mobile agent that is initiated by a host called an initiator and is denoted as $i_0$. The agent starts its itinerary at the initiator $i_0$, then visits $k$ hosts denoted as $i_1$, …, $i_k$ and finally returns to the initiator $i_0$ where it terminates. The agent is assumed to collect some data at each visited host and can visit a non-initiator host more than once. The agent is composed of a static part $\prod$ and a dynamic part M. The static part includes both the agent code and any constants used by the agent, and the dynamic part contains the actually collected data and any scratch variables used during the agent's execution. It is assumed that the actually collected data are cryptographically protected and transferred within the agent during its move from one host to another, whereas the scratch variables are cleared during each move. The protocol is based on: (a) binding the static data to the dynamic data of a mobile agent to enable the detection of an attack that replaces the agent's dynamic data with data of similar agents, and (b) forcing the agent to securely store the addresses of the next hosts to be visited to enable the detection of data truncation attack.

The agent movement from host $i_j$ to host $i_{j+1}$ ($0 \le n \le k$) is modeled by a message that consists of the agent's static part $\prod$, and the protected data of the agent's dynamic part M. The static part $\prod$ is paired with a timestamp $t$, and then the pair is signed by the initiator to obtain $\prod_0$ that uniquely identifies an agent's instance. The protected data of the agent's dynamic part M is such as $\{M_0, \dots, M_k\}$, where $M_k$ is the protected data of the agent's dynamic part that is acquired at host $i_k$.

In the protocol description [14], the encryption of plaintext $m$ into a cipher is written as $\{m\}_{K_{i_n}^+}$, where $K_{i_n}^+$ is the public key of host $i_n$ that encrypted the data. A digital signature is written as encryption with a private signing key $S_{i_n}^{-1}$. The bare signature is computed by signing digitally a hash of data $m$, 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 terms $m_1$ and $m_2$ is denoted as $m_1 \hat{} m_2$, where concatenation

refers to appending a term $m_2$ to another term $m_1$. The hashing of data $m$ is denoted h($m$). The *Configurable Mobile Agent Data Protection Protocol* is described as follows:

$i_n \rightarrow i_{n+1}$: $\prod_0, \{M_0, \ldots, M_k\}$
where, $\prod_0 = \{\prod, t\} S_{i_0}^{-1}$
$M_n = D_n \hat{} C_n$ for ( $0 \le n \le k$)

The notations of the protocol are summarized in Table 3.

In this paper, we consider the protocol configuration that aims to preserve strong data integrity, data authenticity, and data secrecy. The properties are defined as follows:

▪ *Strong data integrity*: upon the agent's return, the initiator $i_0$ can detect if an adversary has tampered with the actually collected data ($d_1, \ldots, d_k$) by the deletion and/or modification of one or more data terms, or the illegitimate insertion of data into ($d_1, \ldots, d_k$).

▪ *Data authenticity*: the initiator $i_0$ can determine for sure the identity of the host $i_n$ that appended $C_n$ to the agent's dynamic part.

▪ *Data secrecy*: the actually collected data of the agent ($d_1, \ldots, d_k$) can only be read at the initiator $i_0$.

The protocol can be configured for the required security properties as follows:

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

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

**Table 3. Protocol's notations**

| | |
|---|---|
| $\prod_0$ | Static code $\prod$ and timestamp $t$ signed by initiator $i_0$ |
| $M_n$ | Cryptographically protected data of the agent's dynamic part at host $i_n$ |
| $d_n$ | Actual data provided by host $i_n$ |
| $P_o$ | Agent's Initial itinerary |
| $P_n$ | New hosts appended to the agent's initial itinerary $P_o$ by host $i_n$ |
| $D_n$ | Pair of Data $d_n$ encrypted with the public key of the initiator, and $P_n$ |
| $C_n$ | Computed checksum at host $i_n$ |
| $i_n$ | Identity of an executing host |
| $k$ | Number of visited host |

## 4. Modeling and Analyzing the Protocol

### 4.1. Modeling the protocol

The execution time of verification slows down as the number of participants, and so of possible data values increases. The model size should be reasonable and susceptible to various malicious acts of intruders, especially colluding attacks. The minimum reasonable model size would be four hosts including the initiator, which would allow for colluding attacks. The model we consider consists of an initiator *I* and three executing hosts *A*, *B*, and *C* respectively, as depicted in Figure 1. Hosts A and C might be colluding hosts trying to amend the data they already provided to the agent, delete the data acquired at the intermediate host B, spy out the gathered data, or insert arbitrary data into the gathered data. Also, hosts A, B, and C can be malicious hosts which try individually to truncate trailing data, append arbitrary data, or impersonate the genuine initiator to spy out confidential data. The selected model size would improve efficiency in terms of execution time, memory occupation, and execution traces.

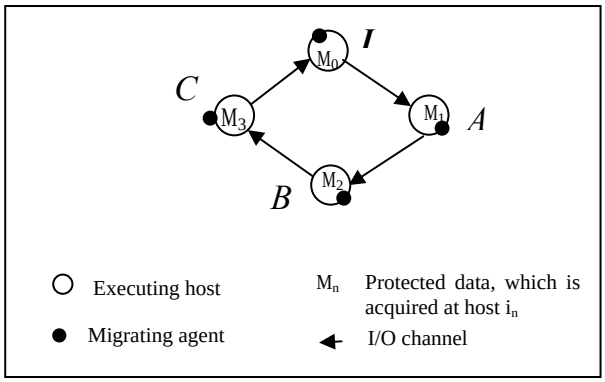


**Fig.1 Instance of the protocol**

The STA script of the protocol consists of the declarations of: identifiers, processes, system configuration, and properties. The identifiers are denoted as follows: identities of participating hosts as *I*, *A*, *B*, and *C*. The static part $\prod_0$ as *SP*. The timestamp t as *T*. The agent's initial itinerary $P_o$ as *P0*. The new hosts $P_1$, $P_2$, and $P_3$ appended to the agent's initial itinerary at host *A*, *B*, and *C* as *P1*, *P2*, and *P3* respectively. The data $d_1$, $d_2$, and $d_3$ acquired at hosts *A*, *B*, and *C* as *X*1, *X2*, and *X3*, respectively. The digital signatures of hosts *I*, *A*, *B*, and *C* as *sigI*, *sigA*, *sigB*, and *sigC* respectively. The public key of host *I* as *kI*. The signature verification keys of hosts: *I*, *A*, *B*, and *C* as -*sigI*, -*sigA*, -*sigB*, and -*sigC* respectively. The I/O actions at hosts *I*, *A*, *B*, and *C* as *i1*, *i2*, *a1*, *a2*, *b1*, *b2*, *c1*, and *c2* respectively. The second part of the agent's dynamic data $M_0$, $M_1$, $M_2$, *and* $M_3$ gathered at hosts *I*, *A*, *B*, and *C* as *C0*, *C1*, *C2*, and *C3*. Variables begin with *x*, *y*, *z*, or *w*. For example, a name *P0* is transmitted from host *I* to host *A*, is then received at *A* as a variable *wP0*. The hashing of data *m* as hsh(*m*). The configuration of the system (*Conf*) consists of the intruder's initial knowledge and the parallel composition of roles of participating hosts and their respective public keys (*Sys*). The intruder's knowledge consists of: (a) identifiers of an old agent (*SP*old, *T*old), (b) participating hosts' identities and the associated public keys and signature verification keys, and (c) an intercepted message.

Due to space limitations, we merely present the roles of hosts *I* and *A*, and the declarations of both the *Sys* and *Conf* of the protocol expressed in the syntax of STA in Figure 2. The full modeling and specifications of the protocol can be found at: http://www.cse.unsw.edu.au/~rjaljoli and are given in appendix A.

```
val iI =      T new_in P0 new_in SP new_in
              i1! (SP,T)^+SigI, P0, (hsh(P0,(SP,T)^+SigI,A))^+SigI >>
              i2?(wSP,wT)^+wSigI,wP0,wC0, (wX1)^+KI,wP1, wC1,(wX2)^+KI, wP2, wC2,
                  (wX3)^+KI,wP3,wC3 >> stop;

val rA =      X1 new_in P1 new_in
              a1?(xSP,xT)^+SigI, xP0, xC0 >>
              a2!(xSP,xT)^+SigI, xP0, xC0, (X1)^+KI, P1,
                  ((hsh(X1,P1,(xSP,xT)^+SigI,xC0, B))^+SigA)^+KI >>   stop;

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

val Conf =  ([disclose!(SPold, Told, I, A, B, C, T, +KI, +KA, +KB, +KC,  -SigI, -SigA, -SigB,-SigC,
              ((xSP,xT)^+SigI, xP0, (xP0,(xSP,xT)^+SigI,A,+SigI)))] @Sys);
```

**Fig. 2 Declarations of roles of hosts *I* and *A*, and *Sys* and *Conf* of the protocol in STA**

### 4.2. Formalizing the security properties

We are primarily concerned with strong data integrity, data authenticity, and data secrecy. Based on the examples in [18], the security properties are declared in STA as shown in Figure 3, and are explained as follows:

▪ The *authenticity* of the data returned to the initiator requires that each term has truly originated from the respective host. The authenticity properties of *X1*, *X2*, and *X3* are declared in STA as *Auth1*, *Auth2*, and *Auth3*. For example, the initiator in *Auth1* would authenticate the received variable *wX1* if it has truly originated from host *A*, e.g. sent through the output action *a2!* of host *A*.

▪ The *secrecy* of the data returned to the initiator is verified by assuming a guardian that can at any time pick a message and try to synthesize some secret data, e.g. *X1,* and then checking that the input action *guard*?*X1* never takes place. The secrecy properties of *X1*, *X2*, and *X3* are declared in STA as *Secrecy1*, *Secrecy2*, and *Secrecy3*.

▪ The *strong integrity* of the data returned to the initiator requires: (i) The signed static part (SP, T)^+SigI is received intact at hosts *A*, *B*, *C,* and *I* and the signer remains the genuine initiator of the agent *I* during the agent's transmission between hosts. If the verification fails, then the executing host might be communicating with an intruder impersonating the genuine initiator. The static part is verified during its transmission between executing hosts as in the declared properties: *Auth4*, *Auth5*, *Auth6*, and *Auth7*. The signed static part is the first in the order of the transmitted terms. It is sent as *w1* and received as *w1'.* (ii) The agent's initial itinerary *P0*, and the new sets of hosts *P1*, *P2* and *P3* appended to *P0* by hosts *A*, *B*, and *C* respectively that are returned to the initiator correspond to the originally generated terms by the respective hosts. The agent's initial itinerary and the appended itineraries are verified as in the declared properties: *Auth8*, *Auth9*, *Auth10*, and *Auth11*. (iii) The data returned to the initiator belong to the agent of concern, which is uniquely identified by (*SP*, *T*). The two terms are verified as in the declared property *Auth12*. (iv) The data returned to the initiator *X1*, *X2*, and *X3* are intact. Checking the authenticity of data in the declared properties *Auth1*, *Auth2*, and *Auth3* imply that the data are intact.

```
val Auth1 = (a2!X1 <-- i2?wX1);
val Auth2 = (b2!X2 <-- i2?wX2);
val Auth3 = (c2!X3 <-- i2?wX3);
val Auth4 = (i1!(w1, w2, w3) <-- a1?(w1', w2, w3, w4, w5, w6));
val Auth5 = (i1!(w1, w2, w3) <-- b1?(w1', w2, w3, w4, w5, w6, w7, w8, w9));
val Auth6 = (i1!(w1, w2, w3) <-- c1?(w1', w2, w3, w4, w5, w6, w7, w8 ,w9, w10, w11, w12));
val Auth7 = (c2!(w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12) <--
              i2?(w1', w2, w3, w4, w5, w6, w7, w8 ,w9, w10, w11, w12));
val Auth8 = (i1!P0 <-- i2?wP0);
val Auth9 = (a2!P1 <-- i2?wP1);
val Auth10 = (b2!P2 <-- i2?wP2);
val Auth11 = (c2!P3 <-- i2?wP3);
val Auth12 = (i1!(w1, w2, w3) <-- i2?(w1', w2, w3,w4, w5, w6, w7, w8, w9, w10, w11, w12));
val Secrecy1 = (Absurd <-- guard?X1);
val Secrecy2 = (Absurd <-- guard?X2);
val Secrecy3 = (Absurd <-- guard?X3);
```

**Fig.3 Declarations of security properties of the protocol in STA**

### 4.3. Analyzing the protocol

The analysis of strong data integrity, data authenticity, and data secrecy properties using STA revealed a failure upon checking the *Auth4* property as shown in Figure 4.

```
> val it ="An attack was found:
        disclose!(SPold,Told,I,A,B,C,T,+KI,+KA,+KB,+KC,-SigI,-SigA,
        -SigB,-SigC,[xSP,xT]+xSigI,w2,w3,w4,w5,w6).
        i1!([SP6,T4]+SigI,P05,[H(P05,[SP6,T4]+SigI,A)]+SigI). a1?([xSP,xT]+SigI,w2,w3,w4,w5,w6)
        2 symbolic configurations reached.": string
```

**Fig. 4 Detected attack on the protocol**

In the attack, an adversary can intercept the agent, decrypt the signed static part of the agent, sign the static part with its private key, and then send the agent impersonating the genuine initiator. Hence, it can breach the privacy of collected data. The protocol is susceptible to data truncation attack, though, it would not be detected by validating checksums if a malicious host replaces the agent's dynamic data that is current during the second visit of the agent to its host with the data that the agent had when it visited it first.

## 5. Protocol Repairing and Analysis

The implemented security techniques in [14] are not satisfactory for the following reasons:

- Initiators carry out verifications upon the agent's return based on both static data such as a timestamp that identifies the protocol run and dynamic data that are stored within the migrating agent, which are 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 as $\{\{\prod, t\} S_{i_0}^{-1}\} K_{i_{n+1}}^{+}$ within an *agent that resides at a trusted host* and co-operates with a major agent that traverses the Internet.

- Initiators validate the data returned with the agent if the checksums that they compute are consistent with the returned checksums that the agent stores. The verifications might not be accurate, especially in the case of colluding attacks. The attack takes place if a host conspires with a preceding host in the agent's itinerary and sends the agent back to it. The preceding host would then 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 first visited its host, if the host had already stored the dynamic data including the register contents 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. We propose to design the agent in a way that it requests the current *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. Hence, malicious hosts would not be able to tamper with the agent's dynamic data. 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 [4]. The technique requests an executing host to create and sign a trace of the execution of the agent at its host, 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 overburden the communication channels as traces are transmitted to the initiator upon request.

- 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 migrating 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*. The verifications would detect if an adversary is impersonating the genuine initiator for the purpose of breach of privacy of the collected data. The

cipher is $h(\prod, t, h(i_0))$ and would replace $\prod_0$ which is enclosed in $C_n$ and is equal to $\{\prod, t\}S_{i_0}^{-1}$ . Also we propose to encrypt the stand alone $\prod_0$ with the public key of the succeeding host in the agent's itinerary to prevent an adversary that intercepts the agent and attempts to decrypt $\prod_0$ and sign it with its private key so as to impersonate the genuine initiator and be able to breach the privacy of data to be gathered. The stand alone $\prod_0$ would be $\{\{\prod, t\} S_{i_0}^{-1}\} K_{i_{n+1}}^{+}$ , where $i_{n+1}$ is the succeeding host in the agent's itinerary. We will denote it as $\prod_{n+1}$ to indicate that the part is just readable at the next host in the agent's itinerary. The $\prod_{n+1}$ would be decrypted at each visited host to enable the agent's execution and the verification on the identity of the genuine initiator. The repaired protocol would be as follows:

$i_n \rightarrow i_{n+1}$: $\prod_{n+1}, \{M_0, \ldots, M_n\}$

where, $\prod_{n+1} = \{\{\prod, t\} S_{i_0}^{-1}\} K_{i_{n+1}}^{+}$

$M_n = D_n \| C_n$ for $(0 \le n \le k)$

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

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

Where, $\prod_0 = h(\prod, t, h(i_0))$

At the arrival of the agent at a visited host, the host decrypts $\prod_{n+1}$ with its private key so having $\prod'_{n+1}$, and then deduces the identity of the signer of $\prod'_{n+1}$. Next, it decrypts the term $\prod'_{n+1}$ with the signature verification key of the signer so having $\prod''_{n+1}$. If the decryption passes, then the agent computes $\prod'_0$ as a hash of the term $\prod''_{n+1}$ and $h(i_0)$. The computed $\prod'_0$ is compared with the term $\prod_0$ enclosed within $C_0$. If the verification passes, it ensures that the agent is signed by the genuine initiator. Hence, the host executes the agent and provides the agent with the requested data encrypted with the public key of the genuine initiator. Next, the host encrypts $\prod'_{n+1}$ with the public key of the succeeding host. Next, the agent requests the host to clear its memory from any data acquired as a result of executing the agent, and then dispatches the agent to the succeeding host. If any of the verifications fails the agent's execution is terminated.

At the return of the agent to the initiator $i_0$, the initiator compares the term $\prod_{n+1}$ returned with the migrating agent with the term received from the trusted co-operating agent. If the verification passes, it decrypts the term $\prod_{n+1}$ it received from the co-operating agent first with its private decryption key and then with its signature verification key. Next, it computes $\prod'_0$ as a hash of the decrypted term and $h(i_0)$ and compares it with corresponding term $\prod_0$, which is enclosed within $C_n$ for $(0 \le n \le k)$. If all the verifications pass, then it implies that the data were generated for the genuine initiator, and for the agent's code and timestamp of concern. The repaired protocol has an advantage over the flawed protocol: it verifies that the data were generated for the agent's code of concern as it compares the computed $\prod_0$ using the terms (SP, T) stored with the co-operating agent with that enclosed within $C_n$ for $(0 \le n \le k)$. In brief it is able to detect if the agent's code has been tampered with. Also, it can detect if the returned data belong to the protocol run of concern, which is identified by a timestamp.

Due to space limitations, we merely present the roles of hosts *I* and *A* of the repaired protocol expressed in the syntax of STA in Figure 5. The declarations of both the *Sys* and *Conf* are the same as of the original protocol. The new notations are as follows: $\prod_{n+1}$ as S, $\{\prod, t\}S_{i_0}^{-1}$ as IE, $\prod_0$ as D. The I/O actions at host I are as follows: i1 is an output action that sends initial verification data to the co-operating agent, i2 is an output action that sends initiating data to the first host in the agent's itinerary, i3 is an input action that receives the data gathered by the mobile agent upon its return to the initiator, i4 is an input action that receives the data stored within the co-operating agent. The full modeling and specifications of the repaired protocol can be found at: http://www.cse.unsw.edu.au/~rjaljoli and are given in appendix B.

```
val iI = T new_in P0 new_in SP new_in
         i1!(((SP,T)^+sigI)^+kI) >>
         i2!(((SP,T)^+sigI)^+kA), P0, ((P0,hsh(SP,T,hsh(I)),A)^+sigI) >>
         i3?wS,wP0,((wP0,wD,A)^+sigI),
           (wX1)^+kI,wP1,((wX1,wP1,xD,  ((wP0,xD,A)^+sigI),B)^+sigA)^+kI,
           (wX2)^+kI,wP2,((wX2,wP2,yD,((wX1,wP1,yD,
           ((wP0,yD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI,
           (wX3)^+kI,wP3,((wX3,wP3,zD,((wX2,wP2,zD,((wX1,wP1,zD,
           ((wP0,zD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI, I)^+sigC)^+kI >>
           (wS pkdecr (-kI, wIE)) >>
         i4?xSP >> (xSP pkdecr (-kI, wSP)) >> (wIE is wSP) >>
           (xD is hsh(SP,T,hsh(I)))  >>  (yD is hsh(SP,T,hsh(I)))  >>
           (zD is hsh(SP,T,hsh(I)))  >>  (wD is hsh(SP,T,hsh(I)))  >>
           Accept!((A,X1), (B,X2), (C,X3)) >>   stop;

val rA =   X1 new_in P1 new_in
           a1?xS, xP0, ((xP0,xD,A)^+sigI) >> (xS pkdecr (-kA, xIE)) >>
              (xIE pkdecr (-sigI, xID)) >> (xD is hsh(xID, hsh(I))) >>
           a2!(xIE)^+kB, xP0, ((xP0,xD,A)^+sigI),
              (X1)^+kI, P1, ((X1,P1,xD,((xP0,xD,A)^+sigI),B)^+sigA)^+kI >>  stop;
```

**Fig. 5 Declarations of roles of hosts *I* and *A*, and *Sys* and *Conf* of the repaired protocol in STA**

Here, only the verifications of data X1 acquired at host *A* are explained and shown expressed in STA. The verifications are of two types as follows.

1. *At the early execution of the agent at host A*: the host checks the identity of the genuine initiator. It decrypts the term xS it receives with its private decryption key and binds the result to variable xIE, and then it deduces the identity of the signer of xIE. Next, it decrypts xIE with the signature verification key of the host its identity is deduced. It would then bind the result to variable xID. If the decryption passes, it computes $\prod'_0$ as hsh(xID, hsh(I)) and then compares it with the corresponding term $\prod_0$ enclosed with $C_0$, which is received as variable xD. If the verification passes, it would ensure that the deduced identity is the identity of the genuine initiator and would encrypt the data it provides to the agent with the public key of the genuine initiator. The verification is as follows.

(xS pkdecr (-kA, xIE)) >>  (xIE pkdecr (-sigI, xID)) >> (xD is hsh(xID, hsh(I))) >>

2. *Upon the agent's return*: the initiator performs the two verifications given below.

▪ Verifies that the offer of host *A* was generated for the genuine initiator.

(xD is hsh(SP,T,hsh(I)))

▪ Verifies that the returned data were generated for the agent's code and timestamp of concern. It checks the static part of the agent, which is securely stored with the co-operating agent and would be communicated to it through the input action a4?, with the corresponding term returned with the agent.

(wS is xSP) >>

We used STA tool to check the security properties of the gathered data. The security properties of the repaired protocol are expressed in STA as given in Figure 6.

```
val Auth1 = (a2!X1 <-- i3?wX1);
val Auth2 = (b2!X2 <-- i3?wX2);
val Auth3 = (c2!X3 <-- i3?wX3);
val Auth4 =  (i2!((SP,T)^+sigI) <-- a2!(xIE));
val Auth5 =  (i2!((SP,T)^+sigI) <-- b2!(yIE));
val Auth6 =  (i2!((SP,T)^+sigI) <-- c2!(zIE));
val Auth7 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(xD));
val Auth8 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(yD));
val Auth9 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(zD));
val Auth10 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(wD));
val Auth11= (a2!(xIE)^+kB <-- b1?yS);
val Auth12= (b2!(yIE)^+kC <-- c1?zS);
val Auth12= (c2!(zIE)^+kI <-- i3?wS);
val Auth13 = (i2!P0 <-- i3?wP0);
val Auth14 = (a2!P1 <-- i3?wP1);
val Auth15 = (b2!P2 <-- i3?wP2);
val Auth16 = (c2!P3 <-- i3?wP3);
val Secrecy1 = (Absurd <-- guard?X1);
val Secrecy2 = (Absurd <-- guard?X2);
val Secrecy3 = (Absurd <-- guard?X3);
```

**Fig. 6 Declarations of security properties of the repaired protocol in STA**

Due to space limitations, we only explain the verification of the data *X1* acquired at host *A* as follows.
1. Verify the authenticity of the data *X1* acquired at host *A* and returned with the agent as given below.

```
val Auth1 = (a2!X1 <-- i3?wX1);
```

2. Verify that host *A* sends out the static part, which was originally signed by the genuine initiator, intact to host B as given below.

```
val Auth4 =  (i2!((SP,T)^+sigI) <-- a2!(xIE));
```

3. Verify that the data *X1* acquired at host *A* were generated for the genuine initiator, and for the agent's code and timestamp of concern as given below.

```
val Auth7 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(xD));
```

4. Verify that an intruder would not able to tamper with the static part, which was originally signed by the genuine initiator, during the agent's move from host *A* to host *B* as given below.

```
val Auth11= (a2!(xIE)^+kB <-- b1?yS);
```

5. Verify that the set of new hosts that host *A* appended to the agent's initial itinerary are maintained intact during the agent's lifecycle as given below.

```
val Auth14 = (a2!P1 <-- i3?wP1);
```

6. Verify the secrecy of the data *X1* acquired at host *A* as given below.

```
val Secrecy1 = (Absurd <-- guard?X1);
```

We analyzed the repaired protocol in four key configurations for the same analyzed instance of the original protocol. The runs are: (i) A single run with an intruder. (ii) A single run of the protocol with host *A* as a malicious host. (iii) A single run with hosts *A* and *C* as conspiring hosts. (iv) Two parallel runs of the protocol, each with an initiator and a particular static part $\prod_0$. The analysis of the security

properties of the data that are gathered by mobile agents using STA shows that the protocol is free of attacks, particularity the attack revealed in the original protocol as shown in Table 4.

**Table 4. Results of analysis of the repaired protocol for different runs**

| Protocol's configuration | Results of analysis |
|---|---|
| A single run with an intruder | No attack was found 328 symbolic configurations reached |
| A single run of the protocol with host A as a malicious host | No attack was found 67 symbolic Configurations reached |
| A single run with hosts A and C as conspiring hosts | No attack was found 18 symbolic Configurations reached |
| Two parallel runs of the protocol, each with an initiator and a particular static part: agent's code, and timestamp | No attack was found 986413 symbolic configurations reached |

We are concerned with the security of a general model of the repaired protocol. We consider the attacks the original protocol failed to prevent or detect, and carefully reason the correctness of a general model of the repaired protocol. The reasoning is given in Table 5. The reasoning shows that the security scheme of the repaired protocol has sufficient measures that are capable of preventing or at least detecting the attacks which the flawed protocol fails to prevent or detect.

**Table 5. Verifying a system of arbitrary size**

| Attack | Protocol's security techniques |
|---|---|
| Intruder intercepts the agent and tries to impersonate the genuine signer of the static part of the agent. | The static part is encrypted with the public key of the succeeding host in the agent's itinerary, and can only be decrypted with a key that is private to the succeeding host. |
| Malicious host tries to impersonate the genuine signer of the static part of the agent. | The succeeding host deduces the identity of the signer of the static part $i_k$, and then computes h ($\prod$, t, h ($i_k$)), and compares it with the cipher enclosed within $C_0$. If the verification fails, the agent execution terminates. |
| Malicious host tries to truncate the gathered data and alter the data it has provided to the agent at an earlier time, by replacing the agent's current dynamic part with the part that the agent had when it first visited it. | The host has to have the dynamic part of the agent when the agent first visited it. The host does not have the part, since the protocol requests the current 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. The denial of request would be revealed in the execution trace it should store and forward to the initiator upon request. |

## 6. Conclusion and Future Work

This paper demonstrates the effectiveness of formal methods in the analysis of data security protocols of mobile agents. In this paper we rectified the "Configurable Mobile Agent Data Protection Protocol" [14] and analyzed the repaired version using STA (Symbolic Trace Analyzer). We analyzed a security module of the protocol that aims to preserve data authenticity, data secrecy, and strong data integrity. STA is as an infinite-state exploration formal method that is used to analyze the security properties of protocols. It analyzes execution traces of the protocol based on symbolic techniques. The symbolic model is finite, because each input action should be preceded by the corresponding action for every generated trace, whereas the analysis of execution traces in model checkers suffer from state-explosion problem and the testing equivalences between processes in process algebra suffers from the infinite

universal quantifications as having infinitely many such processes [7]. It takes less than half an hour to write the STA script of a protocol run. The execution is automatic, though, it slows down as the number of participants and consequently the number of associated terms increases. According to Fiore and Abadi [13] the symbolic analysis is proven sound and complete. Detecting an attack on the symbolic model would imply that an attack exists in the infinite standard model and vice versa.

We analyzed a reasonably small instance of the protocol [14]. The analysis of the protocol using STA revealed a security flaw. We proposed a repair of the protocol, which implements the following security techniques: utilization of co-operating agents, carrying out verification on the identity of the genuine initiator at the early execution of the agent at a visited host, requesting 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 in the agent's itinerary so as to prevent data truncation or alteration, and amending the encryption scheme. Next, we analyzed a reasonably small instance of the repaired protocol in four key configurations. The selected configurations would generate the common attacks, and particularly the attacks the flawed protocol fails to prevent/ detect. The analysis with STA reported no security flaw, and hence would guarantee that the repaired protocol is secure as regards the modeled instance and configurations. The results of analysis provide a motivation for a proof of correctness of the repaired protocol for a general model. Moreover, we reasoned about the security of a general model of the repaired protocol and showed that the protocol is capable of preventing or at least detecting the malicious acts the original protocol fail to prevent or at least detect.

The future direction of this work is to verify the security properties of the protocol using other formal methods, such as model checking and theorem proving so as to compare the methods in terms of simplicity, usability, accuracy and efficiency.

## 7. Acknowledgements

Catherine Meadows, Michele Boreale, Amjad Hudaib, Maria Buscemi, Nandan Parameswaran, Kai Engelhardt, and Ralf Huuch have provided me with valuable and thorough comments on the analysis of security protocols of mobile agents using formal methods that are highly appreciated.

## 8. References

[1] B. Aziz, D. Gray, G. Hamilton, F. Oehil, J. Power, and D. Sinclair, "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.

[2] B. Blanchet, and B. Aziz, "A Calculus for Secure Mobility", In *18th Asian Computing Science Conference (ASIAN'03)*, volume 2896 of LNCS, Springer-Verlag, 2003, pp. 188-204.

[3] C. Meadows, "Formal Verification of Cryptographic Protocols: A Survey", In *Advances in Cryptography – ASIACRYPT'94*, pp. 135-150.

[4] G. Vigna, "Cryptographic Traces for Mobile Agents. In Mobile Agent Security", Vigna, G., editor, volume 1419 LNCS, Springer-Verlag, 1998, pp. 137-153.

[5] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated Analysis of Cryptographic Protocols Using Murφ", In *Proceedings of Symposiums on Security and Privacy*, IEEE Computer Society Press, 1997, pp. 141-153.

[6] J. Vitek, and G. Gastagna, "Seal: A Framework for Secure Mobile Computations", In *Internet Programming Language ICCL'98* Workshop, volume 1686 of LNCS, Springer-Verlag, May 1999, pp. 47-77.

[7] L. Durante, R. Sisto, and A. Valenzano, "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, pp. 155-170.

[8] L. Fischer, "*Protecting Integrity and Secrecy of Mobile Agents on Trusted and Non-trusted Agent Places*", Diploma Dissertation, University of Bremen, Germany, April 2003.
Available at:http://www.sec.informatik.tu- armstadt.de/ lang_neutral/ diplomarbeiten/docs/fischer_diplom.pdf.

[9] L. Ma and J. J. P. Tsai, "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

[10] M. Abadi, and A. D. Gordon, "A Calculus for Cryptographic Protocols: The Spi-calculus", In *Information and Computation*, 148(1): 1-70, 1999.

[11] M. Boreale and D. Gorla, "Process Calculi and the Verification of Security Protocols, In *Journal of Telecommunications and Information Technology–Special Issue on Cryptographic Protocol Verification (JTIT)*, Warsaw, Poland, 2002.

[12] M. Boreale, and M. Buscemi, "Experimenting with STA: a Tool for Automatic Analysis of Security Protocols", *ACM Symposium on Applied Computing 2002*, ACM Press, 2002.

[13] M. Fiore, and M. Abadi, "Computing Symbolic Models for Verifying Cryptographic Protocols", In *Proceedings of the 14th IEEE ComputerSecurity Foundations Workshop (CSFW 2001)*, IEEE Computer Society Press, Washington, pp. 160-173.

[14] P. Maggi, and R. Sisto, "A Configurable Mobile Agent Data Protection Protocol", In *Proceedings of AAMAS'03*, ACM Press, New York, NY, USA, 2003, pp. 851–858.

[15] P. Maggi, and R. Sisto, "Experiments on Formal Verification of Mobile Agent Data Integrity Properties", Available at:
http://www.labic.disco.unimib.it/woa2002/papers/15.pdf

[16] R. Jaljouli, "Boosting m-Business Using a Truly Secured Protocol for Data Gathering Mobile Agents", In the *Proceedings of ICMB 2005*, IEEE Computer Society Press, Sydney.

[17] STA Documentation, available at: http://www.dsi.unifi.it/~boreale/documentation.html

[18] 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

[19] X. Hannotin, P. Maggi, and R. Sisto, "Formal Specification and Verification of Mobile Agent Data Integrity Properties: A Case Study", volume 2240 of LNCS, Springer-Verlag, 2001, pp. 42-53.

[20] Y. Yao, and D. Dolev, "On the Security of Public Key Protocols", In *IEEE Transactions on Information Theory*, 29(2): 198-208, March 1983.

## Appendix A
## Cracking the protocol

The STA script of a single run of the protocol. The agent is initiated by host *I*.

```
(* A single run of the proposed protocol
I: initiator; A: 1st executing host, B: 2nd executing host,
C: 3rd executing host, T: timestamp, Told: old timestamp,
SP: the agent's static part, SPold: an old agent's static part, P0: agent's initial itinerary,
P1, P2, and P3: hosts appended to the agent's initial itinerary at hosts A, B, and C respectively,
X1: data provided by host A, X2: data provided by host B, X3: data provided by 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 verify the integrity, secrecy, and authenticity of the data: X1, X2, and X3 acquired at hosts A, B, and C respectively and returned to it.

```
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, which communicates
the initial verification data to major agent,
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 the 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, disclose, guard $;
DeclName $ SPold, Told, SP, T, SigI, SigA, SigB, SigC, I, A, B, C,
              X1, X2, X3, KI, KA, KB, KC, P0, P1, P2, P3 $ ;
DeclVar $     yX1, zX1, zX2, xSP, ySP, zSP, wSP, xT, wT,
              xP0, yP0, zP0, wP0, yP1, zP1, wP1, zP2, wP2, wP3,
              w1, w1', w3, w2', w3', w4', w2, yT, zT, wX1, wX2, wX3,
              w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14, w15,
              yC1, zX1, zC1, zX2, zC2, yC2, y,
              xSigI, ySigI, zSigI, wSigI, wSigA, wSigB, wSigC,
              xSigA, ySigB, zSigC  $;

(*The process at the initiator I is declared as iI)
val iI =      T new_in P0 new_in SP new_in
              i1!(SP,T)^+SigI, P0, (hsh(P0,(SP,T)^+SigI,A))^+SigI >>
              i2?(wSP,wT)^+wSigI,wP0,wC0,
              (wX1)^+KI,wP1, wC1,(wX2)^+KI, wP2, wC2,
              (wX3)^+KI,wP3,wC3 >> stop;

 (*The process at the 1st executing host A is declared as rA)
val rA =      X1 new_in P1 new_in
              a1?(xSP,xT)^+SigI, xP0, xC0 >>
              a2!(xSP,xT)^+SigI, xP0, xC0,
              (X1)^+KI, P1,
              ((hsh(X1,P1,(xSP,xT)^+SigI,xC0, B))^+SigA)^+KI >>
              stop;
```

```
(*The process at the 3rd executing host B is declared as rB)
val rB =    X2 new_in P2 new_in
            b1?(ySP,yT)^+ySigI, yP0, yC0,
            yX1, yP1, yC1 >>
            b2!(ySP,yT)^+ySigI, yP0, yC0,
            yX1, yP1, yC1,
            (X2)^+KI, P2, ((hsh(X2,P2,(ySP,yT)^+ySigI, yC1, C))^+SigB)^+KI >> stop;

(*The process at the 3rd executing host C is declared as rC)
val rC =    P3 new_in X3 new_in
            c1?(zSP,zT)^+zSigI, zP0, yC0,
            zX1, zP1, zC1, zX2, zP2, zC2 >>
            c2!(zSP,zT)^+zSigI, zP0, yC0,
            zX1, zP1, zC1, zX2, zP2, zC2,
            (X3)^+KI, P3, ((hsh(X3, P3,(zSP,zT)^+zSigI,zC2,I))^+SigC)^+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!(SPold, Told, I, A, B, C, T, +KI, +KA, +KB, +KC,
                    -SigI, -SigA, -SigB,-SigC,((xSP,xT)^+xSigI, xP0, xC0))] @Sys);



(*Checks the integrity of data X1 that is provided by host A)
val Auth1 = (a2!X1 <-- i2?wX1);

(*Checks the integrity of data X2 that is provided by host B)
val Auth2 = (b2!X2 <-- i2?wX2);

(*Checks the integrity of data X3 that is provided by host C)
val Auth3 = (c2!X3 <-- i2?wX3);

(*Checks the integrity of the two identifiers: (i) Agent's static part SP. (ii) Timestamp T that identify
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), as follows.
val Auth4 =     (i1!(w1, w2, w3) <-- a1?(w1', w2, w3, w4, w5, w6));
val Auth5 =     (i1!(w1, w2, w3) <-- b1?(w1', w2, w3, w4, w5, w6, w7, w8, w9));
val Auth6 =     (i1!(w1, w2, w3) <-- c1?(w1', w2, w3, w4, w5, w6, w7, w8 ,w9, w10, w11, w12));
val Auth7 =     (c2!(w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12) <--
                i2?(w1', w2, w3, w4, w5, w6, w7, w8 ,w9, w10, w11, w12));

(*Checks the integrity of agent's initial itinerary P0 and agent's partial itineraries P1, P2, and P3 that
are appended to P0 at hosts A, B, and C respectively)
```

```
val Auth8 = (i1!P0 <-- i2?wP0);
val Auth9 = (a2!P1 <-- i2?wP1);
val Auth10 = (b2!P2 <-- i2?wP2);
val Auth11 = (c2!P3 <-- i2?wP3);

(*Checks the integrity of the two identifiers: (i) Agent's static part SP. (ii) Timestamp T that identify
the protocol run of concern and the genuine initiator of the agent for whom data would be provided, as
the agent starts its itinerary as an output action at the initiator I and returns back as an the input action
at the initiator I. It verifies that an intruder did not tamper with the two identifiers as the agent is sent
out to executing hosts and returns to the initiator)
val Auth12 = (i1!(w1, w2, w3) <-- i2?(w1', w2, w3,w4, w5, w6, w7, w8, w9, w10, w11, w12));

val Auth13 =    (a1?(w1, w2, w3) <-- a2!(w1', w2, w3, w4, w5, w6));
val Auth14 =    (b1?(w1, w2, w3, w4, w5, w6) <-- b2!(w1', w2, w3, w4, w5, w6, w7, w8, w9));
val Auth15 =    (c1?(w1, w2, w3, w4, w5, w6, w7, w8, w9) <--
                c2!(w1', w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12));

(*Checks the secrecy of the data X1, X2, and X3 provided by hosts A, B, C, and I respectively)
val Secrecy1 = (Absurd <-- guard?X1);
val Secrecy2 = (Absurd <-- guard?X2);
val Secrecy3 = (Absurd <-- guard?X3);
```

## Appendix B
## Verifying the repaired protocol

**Single run of the protocol with an intruder**
The STA script for a single run of the protocol. The agent is initiated by host *I* and an intruder is assumed.

(* A single run of the proposed protocol
*I*: initiator; *A*: 1st executing host, *B*: 2nd executing host,
*C*: 3rd executing host, T: timestamp, Told: old timestamp,
SP: the agent's static part, SPold: an old agent's static part,P0: agent's initial itinerary,
P1, P2, and P3: hosts appended to the agent's initial itinerary at hosts *A*, *B*, and *C* respectively,
X1: data provided by host *A*, X2: data provided by host *B*, X3: data provided by 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 verify the integrity, secrecy, and authenticity of the data: X1, X2, and X3 acquired at hosts A, B, and C respectively and returned to it.

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, which communicates the initial verification data to major agent,
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 the 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 $      SPold, Told, SP, T, sigI, sigA, sigB, sigC, I, A, B, C,
                X1, X2, X3, kI, kA, kB, kC, P0, P1, P2, P3 $;
DeclVar $       yX1, zX1, zX2, xSP, ySP, zSP, wSP, xT, wT,
                xD, yD, zD, wD,xS, yS, zS, wS, xID, yID, zID, wID, xIE, yIE, zIE, wIE,
                xP0, yP0, zP0, wP0, yP1, zP1, wP1, zP2, wP2, wP3,
                w1, w1', w3, w2', w3',w2, yT, zT, wX1, wX2, wX3,
                w4, w5, w6, yC1, zX1, zC1, zX2, zC2, yC2, y, xSP, wSP  $;
```

(*The process at the initiator *I* is declared as iI)

```
val iI =    T new_in P0 new_in SP new_in
            i1!(((SP,T)^+sigI)^+kI) >>
           i2!(((SP,T)^+sigI)^+kA), P0, ((P0,hsh(SP,T,hsh(I)),A)^+sigI) >>
           i3?wS,wP0,((wP0,wD,A)^+sigI),
           (wX1)^+kI,wP1,((wX1,wP1,xD, ((wP0,xD,A),+sigI),B)^+sigA)^+kI,
            (wX2)^+kI,wP2,((wX2,wP2,yD,((wX1,wP1,yD,
            ((wP0,yD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI,
           (wX3)^+kI,wP3,((wX3,wP3,zD,((wX2,wP2,zD,((wX1,wP1,zD,
           ((wP0,zD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI, I)^+sigC)^+kI >>(wS pkdecr (-kI,
    wIE)) >>
            i4?xSP >> (xSP pkdecr (-kI, wSP)) >> (wSP is wIE) >>
           (wIE is (SP,T)^+sigI) >>
```

```
(xD is hsh(SP,T,hsh(I)))  >>
(yD is hsh(SP,T,hsh(I)))  >>
(zD is hsh(SP,T,hsh(I)))  >>
(wD is hsh(SP,T,hsh(I)))  >>
Accept!((A,X1), (B,X2), (C,X3)) >>
stop;
```

(*The process at the 1st executing host *A* is declared as rA)

```
val rA =  X1 new_in P1 new_in
          a1?xS, xP0, ((xP0,xD,A)^+sigI) >> (xS pkdecr (-kA, xIE)) >> (xIE pkdecr (-sigI, xID)) >>
          (xD is hsh(xID, hsh(I))) >>
          a2!(xIE)^+kB, xP0, ((xP0,xD,A)^+sigI),
          (X1)^+kI, P1, ((X1,P1,xD,((xP0,xD,A)^+sigI),B)^+sigA)^+kI >>
          stop;
```

(*The process at the 3rd executing host *B* is declared as rB)

```
val rB =  X2 new_in P2 new_in
          b1?yS, yP0, ((yP0, yD,A)^+sigI),
          yX1, yP1, yC1 >>  (yS pkdecr (-kB, yIE)) >> (yIE pkdecr (-sigI, yID)) >>
          (yD is hsh(yID, hsh(I))) >>
          b2!(yIE)^+kC, yP0, ((yP0, yD,A)^+sigI),   yX1, yP1, yC1,
          (X2)^+kI, P2, ((X2,P2,yD, yC1, C)^+sigB)^+kI >> stop;
```

(*The process at the 3rd executing host *C* is declared as rC)

```
val rC =  P3 new_in X3 new_in
          c1?zS, zP0, ((zP0, zD,A)^+sigI),
          zX1, zP1, zC1, zX2, zP2, zC2 >> (zS pkdecr (-kC, zIE)) >> (zIE pkdecr (-sigI, zID)) >>
          (zD is hsh(zID, hsh(I))) >>
          c2!(zIE)^+kI, zP0, ((zP0, zD,A)^+sigI),
          zX1, zP1, zC1, zX2, zP2, zC2,
          (X3)^+kI, P3, ((X3, P3,zD,zC2,I)^+sigC)^+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!(SPold, Told, I, A, B, C, +kI, +kA, +kB, +kC, -sigI, -sigA, -sigB,-sigC)] @Sys);
```

(*Checks the integrity of data X1 that is provided by host *A*)

```
val Auth1 = (a2!X1 <-- i3?wX1);
```

(*Checks the integrity of data X2 that is provided by host *B*)

```
val Auth2 = (b2!X2 <-- i3?wX2);
```

(*Checks the integrity of data X3 that is provided by host *C*)

```
val Auth3 = (c2!X3 <-- i3?wX3);
```

(*Checks that the signed static part that a visited host sends out is the same as the static part, which was originally sent out by the initiator)
val Auth4 = (i2!((SP,T)^+sigI) <-- a2!(xIE));
val Auth5 = (i2!((SP,T)^+sigI) <-- b2!(yIE));
val Auth6 = (i2!((SP,T)^+sigI) <-- c2!(zIE));

(*Checks that the hash of the static part and the identity of the initiator that the initiator receives is the same as the corresponding term, which the initiator has originally sent out)
val Auth7 = (i2!hsh(SP,T,hsh(I)) <-- i3?(xD));
val Auth8 = (i2!hsh(SP,T,hsh(I)) <-- i3?(yD));
val Auth9 = (i2!hsh(SP,T,hsh(I)) <-- i3?(zD));
val Auth10 = (i2!hsh(SP,T,hsh(I)) <-- i3?(wD));

(*Checks the integrity of the two identifiers: (i) Agent's static part SP. (ii) Timestamp T that identify 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), as follows.
val Auth11= (a2!(xIE)^+kB <-- b1?yS);
val Auth12= (b2!(yIE)^+kC <-- c1?zS);
val Auth13= (c2!(zIE)^+kI <-- i3?wS);

(*Checks the integrity of agent's initial itinerary P0 and agent's partial itineraries P1, P2, and P3 that are appended to P0 at hosts *A*, *B*, and *C* respectively)
val Auth14 = (i2!P0 <-- i3?wP0);
val Auth15 = (a2!P1 <-- i3?wP1);
val Auth16 = (b2!P2 <-- i3?wP2);
val Auth17 = (c2!P3 <-- i3?wP3);

(*Checks that the signed term that host A receives is the same as the static part which the initiator has originally sent out)
val Auth20 = (i2!(w1, w2, w3, w4, w5) <-- a1?(w1', w2, w3, w4, w5));

(*Checks the secrecy of the data X1, X2, and X3 provided by hosts *A*, *B*, *C*, and *I* respectively)
val Secrecy1 = (Absurd <-- guard?X1);
val Secrecy2 = (Absurd <-- guard?X2);
val Secrecy3 = (Absurd <-- guard?X3);

**Single run of the protocol with a malicious host**

The STA script for a single run of the protocol. The agent is initiated by host *I*. Host A is a malicious host that participates in the protocol.

(* A single run of the proposed protocol
*I*: initiator; *A*: 1st executing host, *B*: 2nd executing host,
*C*: 3rd executing host, T: timestamp, Told: old timestamp,
SP: the agent's static part, SPold: an old agent's static part,P0: agent's initial itinerary,
P1, P2, and P3: hosts appended to the agent's initial itinerary at hosts *A*, *B*, and *C* respectively,
X1: data provided by host *A*, X2: data provided by host *B*, X3: data provided by 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 verify the integrity, secrecy, and authenticity of the data: X1, X2, and X3 acquired at hosts A, B, and C respectively and returned to it.

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, which communicates the initial verification data to major agent,
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 the 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 $      SPold, Told, SP, T, sigI, sigA, sigB, sigC, I, A, B, C,
                X1, X2, X3, kI, kA, kB, kC, P0, P1, P2, P3 $;
DeclVar $       yX1, zX1, zX2, xSP, ySP, zSP, wSP, xT, wT,
                xD, yD, zD, wD,xS, yS, zS, wS, xID, yID, zID, wID, xIE, yIE, zIE, wIE,
                xP0, yP0, zP0, wP0, yP1, zP1, wP1, zP2, wP2, wP3,
                w1, w1', w3, w2', w3',w2, yT, zT, wX1, wX2, wX3,
                w4, w5, w6, yC1, zX1, zC1, zX2, zC2, yC2, y, xSP, wSP $;
```

(*The process at the initiator *I* is declared as iI)

```
val iI =    T new_in P0 new_in SP new_in
            i1!(((SP,T)^+sigI)^+kI) >>
            i2!(((SP,T)^+sigI)^+kA), P0, ((P0,hsh(SP,T,hsh(I)),A)^+sigI) >>
            i3?wS,wP0,((wP0,wD,A)^+sigI),
            (wX1)^+kI,wP1,((wX1,wP1,xD,
            ((wP0,xD,A)^+sigI),B)^+sigA)^+kI,
            (wX2)^+kI,wP2,((wX2,wP2,yD,((wX1,wP1,yD,
            ((wP0,yD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI,
            (wX3)^+kI,wP3,((wX3,wP3,zD,((wX2,wP2,zD,((wX1,wP1,zD,
            ((wP0,zD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI,
            I)^+sigC)^+kI >>(wS pkdecr (-kI, wIE)) >>
             i4?xSP >> (xSP pkdecr (-kI, wSP)) >> (wSP is wIE) >>
            (wIE is (SP,T)^+sigI) >>
            (xD is hsh(SP,T,hsh(I)))  >>
```

```
        (yD is hsh(SP,T,hsh(I)))  >>
        (zD is hsh(SP,T,hsh(I)))  >>
        (wD is hsh(SP,T,hsh(I)))  >>
        Accept!((A,X1), (B,X2), (C,X3)) >>
        stop;

(*The process at the 2nd executing host B is declared as rB)
val rB =   X2 new_in P2 new_in
           b1?yS, yP0, ((yP0, yD,A)^+sigI),
           yX1, yP1, yC1 >>  (yS pkdecr (-kB, yIE)) >> (yIE pkdecr (-sigI, yID)) >>
           (yD is hsh(yID, hsh(I))) >>
           b2!(yIE)^+kC, yP0, ((yP0, yD,A)^+sigI), yX1, yP1, yC1,
           (X2)^+kI, P2, ((X2,P2,yD, yC1, C)^+sigB)^+kI >> stop;

(*The process at the 3rd executing host C is declared as rC)
val rC =   P3 new_in X3 new_in
           c1?zS, zP0, ((zP0, zD,A)^+sigI),
           zX1, zP1, zC1, zX2, zP2, zC2 >> (zS pkdecr (-kC, zIE)) >> (zIE pkdecr (-sigI, zID)) >>
           (zD is hsh(zID, hsh(I))) >>
           c2!(zIE)^+kI, zP0, ((zP0, zD,A)^+sigI),
           zX1, zP1, zC1, zX2, zP2, zC2,
           (X3)^+kI, P3, ((X3, P3,zD,zC2,I)^+sigC)^+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 || 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!(SPold, Told, I, A, B, C, +kI, +kA, +kB, +kC, -sigI, -sigA, -sigB,-sigC)] @Sys);

(*Checks the integrity of data X2 that is provided by host B)
val Auth1 = (b2!X2 <-- i3?wX2);

(*Checks the integrity of data X3 that is provided by host C)
val Auth2 = (c2!X3 <-- i3?wX3);

(*Checks that the signed static part that a visited host sends out is the same as the static part, which
was originally sent out by the initiator)
val Auth3 =  (i2!((SP,T)^+sigI) <-- b2!(yIE));
val Auth4 =  (i2!((SP,T)^+sigI) <-- c2!(zIE));

(*Checks that the hash of the static part and the identity of the initiator that the initiator receives is the
same as the corresponding term, which the initiator has originally sent out)
val Auth5 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(yD));
val Auth6 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(zD));
val Auth7 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(wD));

(*Checks the integrity of the two identifiers: (i) Agent's static part SP. (ii) Timestamp T that identify
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), as follows.
```

```
val Auth8= (b2!(yIE)^+kC <-- c1?zS);
val Auth9= (c2!(zIE)^+kI <-- i3?wS);

(*Checks the integrity of agent's initial itinerary P0 and agent's partial itineraries P2, and P3 that are
appended to P0 at hosts A, B, and C respectively)
val Auth10 = (i2!P0 <-- i3?wP0);
val Auth11 = (b2!P2 <-- i3?wP2);
val Auth12 = (c2!P3 <-- i3?wP3);


(*Checks the secrecy of the data X1, X2, and X3 provided by hosts A, B, C, and I respectively)
val Secrecy1 = (Absurd <-- guard?X2);
val Secrecy2 = (Absurd <-- guard?X3);
```

**Single run of the protocol with two malicious hosts**

The STA script for a single run of the protocol. The agent is initiated by host *I*. Host *A* and host *C* are two malicious hosts participating in the protocol.

```
(* A single run of the proposed protocol
I: initiator; A: 1st executing host, B: 2nd executing host,
C: 3rd executing host, T: timestamp, Told: old timestamp,
SP: the agent's static part, SPold: an old agent's static part,P0: agent's initial itinerary,
P1, P2, and P3: hosts appended to the agent's initial itinerary at hosts A, B, and C respectively,
X1: data provided by host A, X2: data provided by host B, X3: data provided by 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 verify the integrity, secrecy, and authenticity of the data: X1, X2, and X3 acquired at hosts A, B, and C respectively and returned to it.

```
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, which communicates
the initial verification data to major agent,
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 the 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 $ SPold, Told, SP, T, sigI, sigA, sigB, sigC, I, A, B, C,
                X1, X2, X3, kI, kA, kB, kC, P0, P1, P2, P3 $;
DeclVar $      yX1, zX1, zX2, xSP, ySP, zSP, wSP, xT, wT,
                xD, yD, zD, wD,xS, yS, zS, wS, xID, yID, zID, wID, xIE, yIE, zIE, wIE,
                xP0, yP0, zP0, wP0, yP1, zP1, wP1, zP2, wP2, wP3,
                w1, w1', w3, w2', w3',w2, yT, zT, wX1, wX2, wX3,
                w4, w5, w6, yC1, zX1, zC1, zX2, zC2, yC2, y, xSP, wSP  $;

(*The process at the initiator I is declared as iI)
val iI =    T new_in P0 new_in SP new_in
            i1!(((SP,T)^+sigI)^+kI) >>
            i2!(((SP,T)^+sigI)^+kA), P0, ((P0,hsh(SP,T,hsh(I)),A)^+sigI) >>
            i3?wS,wP0,((wP0,wD,A)^+sigI),
             (wX1)^+kI,wP1,((wX1,wP1,xD, ((wP0,xD,A)^+sigI),B)^+sigA)^+kI,
             (wX2)^+kI,wP2,((wX2,wP2,yD,((wX1,wP1,yD,
             ((wP0,yD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI,
             (wX3)^+kI,wP3,((wX3,wP3,zD,((wX2,wP2,zD,((wX1,wP1,zD,
             ((wP0,zD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI, I)^+sigC)^+kI >>(wS pkdecr (-kI,
            wIE)) >>
            i4?xSP >> (xSP pkdecr (-kI, wSP)) >> (wSP is wIE) >>
             (wIE is (SP,T)^+sigI) >>
             (xD is hsh(SP,T,hsh(I)))  >>
             (yD is hsh(SP,T,hsh(I)))  >>
             (zD is hsh(SP,T,hsh(I)))  >>
```

```
        (wD is hsh(SP,T,hsh(I)))  >>
        Accept!((A,X1), (B,X2), (C,X3)) >>
        stop;

(*The process at the 2nd executing host B is declared as rB)
val rB = X2 new_in P2 new_in
        b1?yS, yP0, ((yP0, yD,A)^+sigI), yX1, yP1, yC1 >>
        (yS pkdecr (-kB, yIE)) >> (yIE pkdecr (-sigI, yID)) >>
        (yD is hsh(yID, hsh(I))) >>
        b2!(yIE)^+kC, yP0, ((yP0, yD,A)^+sigI), yX1, yP1, yC1,
        (X2)^+kI, P2, ((X2,P2,yD, yC1, C)^+sigB)^+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 || 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!(SPold, Told, I, A, B, C, +kI, +kA, +kB, +kC,  -sigI, -sigA, -sigB,-sigC)] @Sys);


 (*Checks the integrity of data X2 that is provided by host B)
val Auth1 = (b2!X2 <-- i3?wX2);

(*Checks that the signed static part that a visited host sends out is the same as the static part, which
was originally sent out by the initiator)
val Auth2 =  (i2!((SP,T)^+SigI) <-- b2!(yIE));

 (*Checks that the hash of the static part and the identity of the initiator that the initiator receives is the
same as the corresponding term, which the initiator has originally sent out)
val Auth3 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(yD));
val Auth4 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(wD));

(*Checks the integrity of agent's initial itinerary P0 and agent's partial itineraries P1, P2, and P3 that
are appended to P0 at hosts A, B, and C respectively)
val Auth5 = (i2!P0 <-- i3?wP0);
val Auth6 = (b2!P2 <-- i3?wP2);

(*Checks the secrecy of the data X2, and X3 provided by host B)
val Secrecy1 = (Absurd <-- guard?X2);
```

**Two parallel runs of the protocol, each with a particular initiator**

The STA script of two parallel runs of the protocol. The first run is initiated by host *I* and the second run is initiated by host *I′*. The first run is identified by (SP, T), and the second run is identified by (SP′, T′).

(* A single run of the proposed protocol
*I*: first initiator; *I′*: second initiator, *A*: 1st executing host, *B*: 2nd executing host,
*C*: 3rd executing host, T: timestamp of 1st run, Told: old timestamp, T′: timestamp of 2nd run,
SP: the agent's static part of the 1st run, SPold: an old agent's static part,P0: agent's initial itinerary,
SP′: the agent's static part of the 2nd run, P0′: agent's initial itinerary,
P1, P2, and P3: hosts appended to the agent's initial itinerary P0 at hosts *A*, *B*, and *C* respectively,
P′1, P′2, and P′3: hosts appended to the agent's initial itinerary P′0 at hosts *A*, *B*, and *C* respectively,
X1: data provided by host *A*, X2: data provided by host *B*, X3: data provided by host *C* in the 1st run,
X1′: data provided by host *A*, X2′: data provided by host *B*, X3′: data provided by host *C* in the 2nd run,

Host *I* sends a mobile agent identified by the static part (SP, T) to gather information from hosts *A*, *B*, and *C*. At the end of the protocol, host *I* needs to verify the integrity, secrecy, and authenticity of the data: X1, X2, and X3 acquired at hosts A, B, and C respectively and returned to it.
Host *I′* sends a mobile agent identified by the static part (SP′, T′) to gather information from hosts *A*, *B*, and *C*. At the end of the protocol, host *I* needs to verify the integrity, secrecy, and authenticity of the data: X1′, X2′, and X3′ acquired at hosts A, B, and C respectively and returned to it.

Notations used in declaring identifiers are as given below.
i1!: output action at host *I* that belongs to the 1st run, i2?: input action at host *I* that belongs to the 1st run ,
i3?: input action at host *I* that belongs to the 1st run and follows the output action of the secondary agent, which communicates the initial verification data to major agent,
i′3?: input action at host *I* that that belongs to the 2nd run and follows the output action of the secondary agent, which communicates the initial verification data to major agent,
i′1!: output action at host *I* that belongs to the 2nd run, i′2?: input action at host *I* that belongs to the 2nd run,
a1?: input action at host *A* that belongs to the 1st run, a2!: output action at host *A* that belongs to the 1st run,
a′1?: input action at host *A* that belongs to the 2nd run, a′2!: output action at host *A* that belongs to the 2nd run,
b1?: input action at host *B* that belongs to the 1st run, b2!: output action at host *B* that belongs to the 1st run,
b′1?: input action at host *B* that belongs to the 2nd run , b′2!: output action at host *B* that belongs to the 2nd run,
c1?: input action at host *C* that belongs to the 1st run, c2!: output action at host *C* that belongs to the 1st run,
c′1?: input action at host *C* that belongs to the 2nd run, c′2!: output action at host *C* that belongs to the 2nd run,
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 of the 1st run that satisfy the intended security properties to host *I* upon the agent's return,
accept2!: an output action that outputs the execution results of the 2nd run that satisfy the intended security properties to host *I* upon the agent's return,
KA, KB, KC, KI, KI′: public keys of hosts *A*, *B*, *C*, *I* and *I′* respectively,
SigA, SigB, sigC, SigI, SigI′: private signing keys of hosts *A*, *B*, *C*, *I* and *I′* respectively)

```
DeclLabel $   a1, a2, b1, b2, c1, c2, i1, i2, i3, i4, Accept1,
              a'1, a'2, b'1, b'2, c'1, c'2, i'1, i'2, i'3, i’4, disclose, guard, Accept2 $;
DeclName $ SPold, Told, SP, T, SP', T', sigI, sigI', sigA, sigB, sigC, I, I', A, B, C,
              X1, X2, X3, X1', X2', X3', kI, kI', kA, kB, kC, P0, P1, P2, P3 , P'0, P'1, P'2, P'3 $;
DeclVar $     yX1, zX1, zX2, xD, yD, zD, wD,xS, yS, zS, wS,
              xID, yID, zID, wID, xIE, yIE, zIE, wIE,
              xP0, yP0, zP0, wP0, yP1, zP1, wP1, zP2, wP2, wP3,
              w1, w1', w3, w2', w3',w2, yT, zT, wX1, wX2, wX3,
              w4, w5, w6, yC1, zX1, zC1, zX2, zC2, yC2, y
              yX1', zX1', zX2', x'D, y'D, z'D, w'D, x'S, y'S, z'S, w'S,
              xID', yID', zID', xIE', yIE', zIE', wIE',
              xP'0, yP'0, zP'0, wP'0, yP'1, zP'1, wP'1, zP'2, wP'2, wP'3,
              wX1', wX2', wX3', yC'1, zX1', zC'1, zX2', zC'2,
              xSP’, ySP, zSP, wSP’, xT, wT, xSP, wSP, vS, zSP, xSP’, wSP’$;

(*The process at the initiator I is declared as iI)
val iI =    T new_in P0 new_in SP new_in
            i1!(((SP,T)^+sigI)^+kI) >>
            i2!(((SP,T)^+sigI)^+kA), P0, ((P0,hsh(SP,T,hsh(I)),A)^+sigI) >>
            i3?wS,wP0,((wP0,wD,A)^+sigI),
            (wX1)^+kI,wP1,((wX1,wP1,xD,((wP0,xD,A)^+sigI),B)^+sigA)^+kI,
            (wX2)^+kI,wP2,((wX2,wP2,yD,((wX1,wP1,yD,
            ((wP0,yD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI,
            (wX3)^+kI,wP3,((wX3,wP3,zD,((wX2,wP2,zD,((wX1,wP1,zD,
            ((wP0,zD,A)^+sigI),B)^+sigA)^+kI,C)^+sigB)^+kI, I),+sigC)^+kI >>
             (wS pkdecr (-kI, wIE)) >>
            i4?xSP >> (xSP pkdecr (-kI, wSP)) >> (wSP is wIE) >>
            (wIE is (SP,T)^+sigI) >>
            (xD is hsh(SP,T,hsh(I)))  >>
            (yD is hsh(SP,T,hsh(I)))  >>
            (zD is hsh(SP,T,hsh(I)))  >>
            (wD is hsh(SP,T,hsh(I)))  >>
            Accept1!((A,X1), (B,X2), (C,X3)) >>
            stop;


val iI' =   T' new_in P'0 new_in SP' new_in
            i'1!(((SP,T)^+sigI)^+kI) >>
            i’2!(((SP',T')^+sigI')^+kA), P'0, ((P'0,hsh(SP',T',hsh(I')),A)^+sigI') >>
            i'3?w'S,wP'0,((wP'0,w'D,A)^+sigI'),
            (wX1')^+kI',wP'1,((wX1',wP'1,x'D, ((wP'0,x'D,A)^+sigI'),B)^+sigA)^+kI',
            (wX2')^+kI',wP'2,((wX2',wP'2,y'D,((wX1',wP'1,y'D,
            ((wP'0,y'D,A)^+sigI'),B)^+sigA)^+kI',C)^+sigB)^+kI',
            (wX3')^+kI',wP'3,((wX3',wP'3,z'D,((wX2',wP'2,z'D,((wX1',wP'1,z'D,
            ((wP'0,z'D,A)^+sigI'),B)^+sigA)^+kI',C)^+sigB)^+kI', I'),+sigC)^+kI' >>
             (w'S pkdecr (-kI', wIE')) >>
            i4?xSP’ >> (xSP’ pkdecr (-kI, wSP’)) >> (wSP’ is wIE’) >>
            (wIE is (SP',T')^+sigI') >>
            (x'D is hsh(SP',T',hsh(I')))  >>
            (y'D is hsh(SP',T',hsh(I')))  >>
            (z'D is hsh(SP',T',hsh(I')))  >>
            (w'D is hsh(SP',T',hsh(I')))  >>
            Accept2!((A,X1'), (B,X2'), (C,X3')) >>
            stop;
```

(*The process at the 1st executing host *A* is declared as rA)

```
val rA =   X1 new_in P1 new_in
           a1?xS, xP0, ((xP0,xD,A)^+sigI) >> (xS pkdecr (-kA, xIE)) >>
           (xIE pkdecr (-sigI, xID)) >> (xD is hsh(xID, hsh(I))) >>
           a2!(xIE)^+kB, xP0, ((xP0,xD,A)^+sigI),
           (X1)^+kI, P1, ((X1,P1,xD,((xP0,xD,A)^+sigI),B)^+sigA)^+kI >> stop
           ||
           X1' new_in P'1 new_in
           a'1?x'S, xP'0, ((xP'0,x'D,A)^+sigI') >> (x'S pkdecr (-kA, xIE')) >>
           (xIE' pkdecr (-sigI', xID')) >> (x'D is hsh(xID', hsh(I'))) >>
           a'2!(xIE')^+kB, xP'0, ((xP'0,x'D,A)^+sigI'),
           (X1')^+kI', P'1, ((X1',P'1,x'D,((xP'0,x'D,A)^+sigI'),B)^+sigA)^+kI' >>
           stop;
```

(*The process at the 2nd executing host *B* is declared as rB)

```
val rB = X2 new_in P2 new_in

           b1?yS, yP0, ((yP0, yD,A)^+sigI),
         yX1, yP1, yC1 >>  (yS pkdecr (-kB, yIE)) >> (yIE pkdecr (-sigI, yID)) >>
         (yD is hsh(yID, hsh(I))) >>
         b2!(yIE)^+kC, yP0, ((yP0, yD,A)^+sigI),
         yX1, yP1, yC1,
         (X2)^+kI, P2, ((X2,P2,yD, yC1, C)^+sigB)^+kI >> stop
         ||
         X2' new_in P'2 new_in
         b'1?y'S, yP'0, ((yP'0, y'D,A)^+sigI'),
         yX1', yP'1, yC'1 >>  (y'S pkdecr (-kB, yIE')) >> (yIE' pkdecr (-sigI', yID')) >>(y'D is
         hsh(yID', hsh(I'))) >>
         b'2!(yIE')^+kC, yP'0, ((yP'0, y'D,A)^+sigI'),
         yX1', yP'1, yC'1,
         (X2')^+kI', P'2, ((X2',P'2,y'D, yC'1, C)^+sigB)^+kI' >> stop;
```

(*The process at the 3rd executing host *C* is declared as rC)

```
val rC = P3 new_in X3 new_in
           c1?zS, zP0, ((zP0, zD,A)^+sigI),
           zX1, zP1, zC1, zX2, zP2, zC2 >> (zS pkdecr (-kC, zIE)) >> (zIE pkdecr (-sigI, zID)) >>
           (zD is hsh(zID, hsh(I))) >>
           c2!(zIE)^+kI, zP0, ((zP0, zD,A)^+sigI),
           zX1, zP1, zC1, zX2, zP2, zC2,
           (X3)^+kI, P3, ((X3, P3,zD,zC2,I)^+sigC)^+kI >> stop
           ||
           P'3 new_in X3' new_in
           c'1?z'S, zP'0, ((zP'0, z'D,A)^+sigI'),
           zX1', zP'1, zC'1, zX2', zP'2, zC'2 >> (z'S pkdecr (-kC, zIE')) >> (zIE' pkdecr (-sigI', zID'))
           >>
           (z'D is hsh(zID', hsh(I'))) >>
           c'2!(zIE')^+kI', zP'0, ((zP'0, z'D,A)^+sigI'),
           zX1', zP'1, zC'1, zX2', zP'2, zC'2,
           (X3')^+kI', P'3, ((X3', P'3,z'D,zC'2,I')^+sigC)^+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 kI' new_in kA new_in kB new_in kC new_in iI || 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!(SPold, Told, I, I', A, B, C, +kI, +kI', +kA, +kB, +kC, -sigI, -sigI', -sigA, -sigB,-
sigC)] @Sys);

 (*Checks the integrity of data X1 that is provided by host A)
val Auth1 = (a2!X1 <-- i3?wX1);

(*Checks the integrity of data X2 that is provided by host B)
val Auth2 = (b2!X2 <-- i3?wX2);

(*Checks the integrity of data X3 that is provided by host C)
val Auth3 = (c2!X3 <-- i3?wX3);

(*Checks the integrity of the two identifiers: (i) Agent's static part SP. (ii) Timestamp T that identify
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), as follows.
val Auth4 =  (i2!((SP,T)^+SigI) <-- a2!(xIE));
val Auth5 =  (i2!((SP,T)^+SigI) <-- b2!(yIE));
val Auth6 =  (i2!((SP,T)^+SigI) <-- c2!(zIE));

(*Checks that the hash of the static part and the identity of the initiator that the initiator receives is the
same as the corresponding term, which the initiator has originally sent out)
val Auth7 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(xD));
val Auth8 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(yD));
val Auth9 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(zD));
val Auth10 =  (i2!hsh(SP,T,hsh(I)) <-- i3?(wD));


(*Checks the integrity of the two identifiers: (i) Agent's static part SP. (ii) Timestamp T that identify
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), as follows.
val Auth11= (a2!(xIE)^+kB <-- b1?yS);
val Auth12= (b2!(yIE)^+kC <-- c1?zS);
val Auth13= (c2!(zIE)^+kI <-- i3?wS);


(*Checks the integrity of agent's initial itinerary P0 and agent's partial itineraries P1, P2, and P3 that
are appended to P0 at hosts A, B, and C respectively)
val Auth14 = (i2!P0 <-- i3?wP0);
val Auth15 = (a2!P1 <-- i3?wP1);
val Auth16 = (b2!P2 <-- i3?wP2);
val Auth17 = (c2!P3 <-- i3?wP3);


(*Checks the secrecy of the data X1, X2, and X3 provided by hosts A, B, C, and I respectively)
val Secrecy1 = (Absurd <-- guard?X1);
val Secrecy2 = (Absurd <-- guard?X2);
val Secrecy3 = (Absurd <-- guard?X3);
```