Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 99 (2004) 339–359
www.elsevier.com/locate/entcs

Modeling A Certified Email Protocol using I/O Automata

C. Blundo, S. Cimato, R. De Prisco, A. L. Ferrara [1]

Dipartimento di Informatica ed Applicazioni
Università di Salerno
84081 Baronissi (SA)

Abstract

Describing and reasoning about asynchronous distributed systems is often a difficult and error prone task. In this paper we experiment the Input/Output Automata framework as a tool to describe and reason about cryptographic protocols running in an asynchronous distributed system. We examine a simple certified email protocol [5], give its formalization using the IOA model, and prove that some security properties are satisfied during the execution of the protocol.

1 Introduction

With the spreading diffusion of the Internet and the World Wide Web, our society is becoming more and more dependent on communication data which are transmitted over computer networks. A large number of transactions involving a growing number of people has been actually replaced by their digital analogues, in which electronic "objects" are exchanged among two or more parties. An example comes from the diffusion of the electronic mail service which allows users to exchange messages containing text or multimedia files.

Because of its features, such as low cost, rapidity and accessibility, the email service is increasingly used in place of ordinary mail. In many cases, email messages are recognized as receipts or evidences of online transactions,

[1] Email: {carblu,cimato,robdep,ferrara}@dia.unisa.it

1571-0661 © 2004 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2004.02.015

such as buying airline tickets, or submissions of papers for publications in conferences or journals, and so on. However the use of email poses some problems, since in its simplest form the email service does not have many features that are usually required in such cases. The standard email service is based on the Simple Mail Transfer Protocol [12] and Post Office Protocol [10], which do not offer guarantees on the delivery and the integrity of the messages. Messages are usually stored and transmitted in plain text allowing a malicious adversary to tap the connection during the transfer and making him able to access sensible data.

In order to provide some form of protection, cryptographic techniques have been employed to obtain additional guarantees on the email service. A number of certified email protocols has been presented in literature, ensuring that the message exchange procedure provides the participants with different security properties. Usually such protocols involve a trusted third party (TTP for brief) which controls the behavior of the participants, helping them in the message exchange, and resolving any dispute if necessary. According to the role played by the TTP, protocols have been classified as *inline* or *optimistic*. In inline protocols [3,5,14,15], the TTP is actively involved in each message exchange. In optimistic protocols [1,2,9], the sender and the receiver perform the message exchange without the intervention of the TTP but they can invoke the TTP to resolve any dispute, caused for example by a cheating attempt from one of the party.

In this paper we analyze Deng's certified email protocol [5], and present its formal model relying on the Input/Output Automaton [7], (IOA for brief), framework. IOA provides a framework allowing both a precise description of the code and the possibility of very detailed proofs [6,13]. The aim of the work is to use the IOA as a tool to describe and to reason about cryptographic protocols running in an asynchronous distributed system. In this perspective, to perform the analysis, we consider a scenario in which the participants to the protocol are modeled as interacting nodes in a distributed system. Their behaviour is then described through IOA automata. The IOA model is then used to prove that some security properties are satisfied during the execution of the protocol.

The IOA formalism has been previously employed for the modeling and the analysis of security protocols in [8], where the correctness of a simple shared key communication protocol and the Diffie-Helmann key distribution protocol has been proved. The security of Asokan's certified email protocol [1] has been analyzed in [11], where a formal model relying on simulatability and probabilistic state-transition machines is employed.

This paper is organized as follows. In the next section, we introduce the

framework we consider to analyze the protocol, presenting the setting and cryptographic primitives used during the execution of the protocol. In Section 3 we describe Deng's email protocol, and in Section 4 we present its IOA formalization. Finally, the correctness of the protocol is provided in Section 5, where non repudiation properties for the origin and destination and fairness properties are shown to hold with the help of invariant assertion proofs. Conclusions are drawn in Section 6.

2 The Framework

We consider a distributed system consisting of n nodes (processors) $\mathcal{P} = \{1, 2, \ldots, n\}$ and a special node, namely the Trusted Third Party (TTP for brief) which is delegated by the participants to control the behavior of the parties, assist them during the exchange of messages and resolve any dispute if necessary. The TTP is a fully trusted party, meaning that the senders and the receivers have complete trust in it. Moreover, there is a communication channel between each node of the set $\mathcal{P} \cup \{\text{TTP}\}$.

2.1 Cryptographic Primitives

The cryptographic primitives used in this paper are:

- $Sig_A(m)$: denotes the digital signature of the message m using the private key of user A under a public-key signature algorithm;
- $h(m)$: indicates the hash of message m using some collision resistant hashing scheme. A collision resistant hash function maps arbitrary length messages to constant size messages such that it is computationally infeasible to find any two distinct messages hashing to the same value.
- $PK_B(m)$: denotes the encryption of message m using the public key of user B in some public-key encryption algorithm. The algorithm should provide non-malleability, i.e., given a ciphertext it is impossible to generate another ciphertext such that the respective plaintexts are related.
- $E_k(m)$: denotes the encryption of message m using the key k under some symmetric encryption algorithm.

2.2 IOA Automata

In order to help the reader not familiar with IOA to understand the code we briefly explain how to read IOA code using a simple example.

An IOA is a simple type of state machine in which transitions are associated with named *actions*. Figure 1 shows an automaton that models a channel

CHANNEL$_{i,j}$

State:
$Msgs$, a set of elements of $\mathcal{M}$, initially empty

Actions:

input Send$(m)_{i,j}$
Eff: add m to $Msgs$

output Receive$(m)_{i,j}$
Pre: m is in $Msgs$
Eff: remove m from $Msgs$

Tasks: {**Receive**$(m)_{i,j}$}

Fig. 1. Automaton CHANNEL$_{i,j}$

for communication from node i to node j. The state is a list of all the variables that describe the state of the automaton. For this channel the state is completely described by a variable that contains the messages still in transit on the channel.

The channel has an *input* action **Send**$(m)_{i,j}$ which is controlled by another (unspecified in the example) automaton A, modeling node i, which has the same action **Send**$(m)_{i,j}$ as an *output* action. Whenever automaton A executes this action also the channel executes the action (at the same time), we will say that the action **Send** of A controls the action **Send** of CHANNEL$_{i,j}$. In this case the effect of the action, in the channel automaton, is to add a message in the set of in transit messages.

The channel has an *output* action **Receive**$(m)_{i,j}$ which has a precondition (a boolean condition) specifying when the action is enabled, that is when the action can be executed. An output action can be executed whenever it is enabled. Moreover, all other automata that have such an action as input will execute it. There will be an automaton B, modeling node j, that has **Receive**$(m)_{i,j}$ as an input action.

There are also *internal* actions that are similar to output actions (i.e., have a precondition and an effect) with the difference that they do not interact with other automata (i.e., several automaton may have internal actions with the same name and they are all independent). We use the notation *name.var* to indicate variable *var* of automaton *name*, for example CHANNEL.*Msgs* refers to variable *Msgs* of automaton CHANNEL$_{i,j}$.

Each IOA comes equipped with a partition of its locally controlled actions (output and internal actions); each equivalence class in the partition represents some task that the automaton is supposed to perform. In order for the input/output interaction to happen automata describing a system have to be composed together. The composition of several IOA is one single IOA. The execution of an IOA consists of a sequence of alternating states and transitions, beginning from a starting state. An execution is called *fair* if each task

gets infinitely many opportunities to perform one of its actions. Formally, an execution fragment α of an IOA A is said to be fair if the following conditions hold for each class C of tasks of A:

(i) If α is finite, then C is not enabled in the final state of α.

(ii) If α is infinite, then α contains either infinitely many events from C or infinitely many occurrences of states in which C is not enabled.

We refer the reader to [7], Chapters 8 and 23, for more information about the IOA models.

3 The CMPl Protocol

We now describe the CMP1 protocol for certified mail presented by Deng et al. in [5]. A concise representation of protocol message flow is provided in Figure 2.

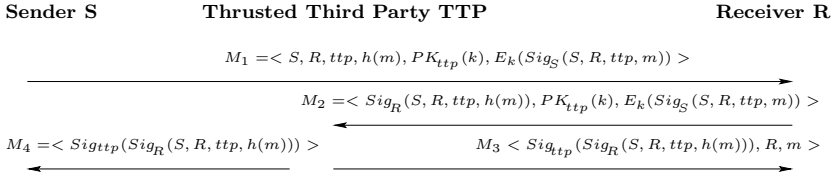

Fig. 2. A concise representation of protocol message flows in CMPl.

To send a mail message containing m to the *Receiver* R, the *Sender* S first digitally signs (S, R, ttp, m) with his private key to produce $Sig_S(S, R, ttp, m)$. Then, S generates a session key k and encrypts the signed data under k using a symmetric key cryptosystem. Finally, S computes $h(m)$ and sends the message $M_1 = \langle S, R, ttp, h(m), PK_{ttp}(k), E_k(Sig_S(S, R, ttp, m))\rangle$ to R. The clear text part (i.e., $S, R, ttp, h(m)$) in this message serves as the mail identifier. This message informs R that there is a certified mail from S to him. After receiving this message, R has two choices. He may ignore the message. In this case, the protocol is aborted. He may choose to receive the message. In this case, he signs $(S, R, ttp, h(m))$ using his private key and sends the message $M_2 = \langle Sig_R(S, R, ttp, h(m)), PK_{ttp}(k), E_k(Sig_S(S, R, ttp, m))\rangle$ to TTP. Upon receiving this message, the TTP first checks the validity of $Sig_R(S, R, ttp, h(m))$ using public key of R. Then, it decrypts $PK_{ttp}(k)$ using its private key, and decrypts $E_k(Sig_S(S, R, ttp, m))$ using k. Next, the TTP checks the validity of $Sig_S(S, R, ttp, m)$ using S's public key, computes $h(m)$, and compares this $h(m)$ with the one received in $Sig_R(S, R, ttp, h(m))$. If the two values match, the TTP knows that m is the mail content that S wanted to send to R, and that R is willing to receive m. In this case, the TTP is able to compute the messages

$M_3 = \langle Sig_{ttp}(Sig_R(S, R, ttp, h(m))), R, m\rangle$ corresponding to the proof-of-origin and $M_4 = \langle Sig_{ttp}(Sig_S(S, R, ttp, m))\rangle$ corresponding to the proof-of-delivery and sends them to R and to S, respectively.
In the next sections, by using the IOA model, we show that the protocol CMP1 meets the following requirements:

- Non-repudiation of origin. The protocol provides the recipient of an email with an irrefutable proof that the mail content received was the same as the one sent by the originator. This proof-of-origin can protect against any attempt by the originator to falsely deny sending that message.
- Non-repudiation of delivery. The protocol provides the mail originator with an irrevocable proof that the mail content received by the recipient was the same as the one sent by the originator. This proof-of-delivery can protect against any attempt by the recipient to falsely deny receiving the message.
- Fairness. Proper execution of the protocol ensures that the proof-of-delivery from the mail recipient and the proof-of-origin from the mail originator are available to the mail originator and recipient, respectively. Moreover, the protocol must be fail-safe. That is, incomplete execution of the protocol will not result in a situation where the proof-of-delivery is available to the originator but the proof-of-origin is not available to the recipient, or vice versa.

4 Description of CMP1 using IOA Model

In this section we provide a detailed description of CMP1 protocol by using the IOA model. We use an automaton SENDER_i to model the sender part on node i and an automaton RECEIVER_i to model the receiver part on node i. Hence, each node $i \in \mathcal{P}$ is modeled with the composition of automata: SENDER_i and RECEIVER_i. The TTP is modeled with a single automaton and, for each $i, j \in \mathcal{P} \cup \{ttp\}$ there is an automaton which models the channel between the node i and the node j.

We assume that the channel from the TTP to any node $i \in \mathcal{P}$ is reliable, i.e., we assume that these channels do not lose or alter in transit messages. Therefore, we distinguish two type of channels, a reliable one: $\text{CHANNEL}_{ttp,i}$, and an unreliable one: $\text{UNREL\_CHANNEL}_{i,j}$, for any $i \in \mathcal{P}$ and $j \in \mathcal{P} \cup \{ttp\}$. The overall system is described by the composition of all the above automata. Figure 3 gives an overview of the automata that compose the system.

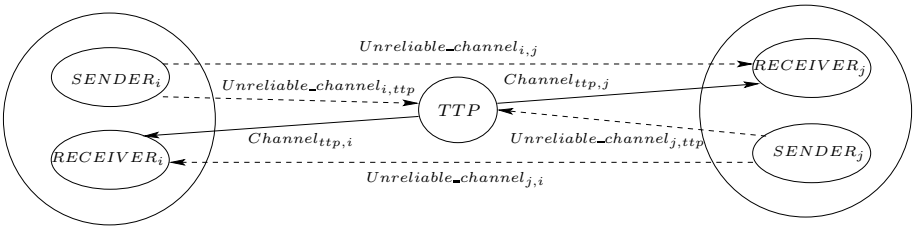

Fig. 3. Overview of the system modeled as IOA.

4.1 IOA Code for the Sender

The code of SENDER_i is shown in Figure 4. For each session, the sender keeps the following information: the *StatusSnd* is the "program counter" that goes through the steps of the normal protocol; variables *M1* and *M4* are used to store the corresponding messages of the protocol.

SENDER_i
Let $\mathcal{S} = \{\texttt{idle}, \texttt{send}, \texttt{wait}, \texttt{done}\}$
State:
for each $id \in \mathcal{N}$
$StatusSnd(id) \in \mathcal{S}$, initially `idle`
$M1(id) \in \mathcal{M}$, initially `nil`
$M4(id) \in \mathcal{M}$, initially `nil`

Actions:
input Deliver$(m, j)_i$
Eff: $id := Getuniqid(m, j)$
$M1(id) := Constr\_M_1(m, id)$;
$StatusSnd(id) :=$ `send`

output Send$(M1(id), id)_{i,j}$
Pre: $StatusSnd(id) =$ `send`
Eff: $StatusSnd(id) :=$ `wait`

Tasks: {**Send**$(M1(id), id)_{i,j}$}

input Receive$(m, id)_{ttp,i}$
Eff: if $(StatusSnd(id) =$ `wait`$)$
$M4(id) := m$
$StatusSnd(id) :=$ `done`

Fig. 4. Automaton SENDER_i

We can now start with the description of the automaton actions, and will proceed by looking at each of them in the order they appear in the code from top to bottom, left column first. This order corresponds to the *logical* order in which the actions are executed. Notice the use of the unique identifier *id*: it is attached to all the messages concerning a particular email: this is just to avoid interference with possible delayed messages from other sessions.

We assume that the environment tells the automaton when to send an email m to a recipient j; this is modeled by the input action **Deliver**$(m, j)_i$. A new session id is created for this email by means of the function *Getuniqid* and this id is used to identify all the communication related to this request. The

first step in the processing of a request for an email m is simply to construct the first message of the protocol $M_1 = \langle S, R, ttp, h(m), PK_{ttp}(k), E_k(Sig_s(S, R, ttp, m))\rangle$ where k is a session key, by using the function $Constr\_M_1$. Variable *StatusSnd* is set to `send` so that the only (non-input) action that is enabled is the **Send** action. This action interacts with the channel to the recipient j and sends the message stored in *M1*. The program counter goes into a wait state `wait`. All the non input actions are not enabled now. The execution proceeds when a message is received from the TTP. When this message is received, it is stored into variable *M4*. The program counter is updated to `done`. At this point the protocol has terminated successfully and nothing else has to be done. The output action `Send` is in a task, so in a fair execution it has infinitely many opportunities to be performed.

4.2 IOA Code for the Receiver

The code of RECEIVER$_i$ is shown in Figure 5. As for the sender, state variables are indexed by a session id. Again, the state variable *StatusRcv* is the "program counter". Variables *M1*, *M2* and *M3* are used to store the corresponding messages of the protocol.

RECEIVER$_i$
Let $\mathcal{S} = \{$`idle`, `received`, `wait`, `discarded`, `done`$\}$
State:
for each $id \in \mathcal{N}$
$StatusRcv(id) \in \mathcal{S}$, initially `idle`
$M1(id) \in \mathcal{M}$, initially `nil`
$M2(id) \in \mathcal{M}$, initially `nil`
$M3(id) \in \mathcal{M}$, initially `nil`

Actions:

input Receive$(m, id)_{j,i}$
Eff: $M1(id) := m$
$StatusRcv(id)$:= `received`

output Send$(M2(id), id)_{i,ttp}$
Pre: $StatusRcv(id)$ = `received`
$M2(id) = Constr\_M_2(M1(id), id)$
Eff: $StatusRcv(id)$:= `wait`

Tasks: {**Send**$(m, id)_{i,ttp}$, **Discard**$(id)_i$}

output Discard$(id)_i$
Pre: $StatusRcv(id)$ = `received`
Eff: $StatusRcv(id)$:= `discarded`

input Receive$(m, id)_{ttp,i}$
Eff: if($StatusRcv(id)$ = `wait`)
$M3(id) := m$
$StatusRcv(id)$:= `done`

Fig. 5. Automaton RECEIVER$_i$

We next describe the actions, top to bottom, left to right. The **Lose** action models the delivery of a corrupt message. The program counter *StatusRcv* is set to `discarded` and the protocol is aborted. The first **Receive** action takes a message from the channel and starts processing the incoming message.

Variable *M1* is used to store the message itself. The program counter *StatusRcv* is set to `received` so that the enabled actions are **Send** and **Discard**. The automaton non-deterministically executes one of these actions. If it executes the **Discard** action the program counter *StatusRcv* is set to `discarded` and nothing else has to be done. Otherwise, using the function $Constr\_M_2$ the message $M_2 = \langle Sig_R(S, R, ttp, h(m)), PK_{ttp}(k), E_k(Sig_S(S, R, ttp, m))\rangle$ is constructed and it is sent to TTP. The automaton goes into a waiting state (no internal or output action is enabled) by setting *StatusRcv* to `wait`. The automaton exits from this waiting state upon reception of a message from TTP. When this message is received, it is stored into variable *M3*. The program counter is updated to `done`. At this point the protocol has terminated successfully. The `done` state for this session, means that the receiver has the original email. The **Send** and **Discard** actions are in the same task, hence, in a fair execution this task gets infinitely many opportunities to perform one of these actions.

4.3 IOA Code for the Trusted Third Party

TTP

Let $\mathcal{S} = \{$`idle`, `received`, `send_rcv`, `send_snd`, `corrupt`, `done`$\}$

State:

for each $id \in \mathcal{N}$

- $StatusTtp(id) \in \mathcal{S}$, initially `idle`
- $M2(id) \in \mathcal{M}$, initially `nil`
- $M3(id) \in \mathcal{M}$, initially `nil`
- $M4(id) \in \mathcal{M}$, initially `nil`
- $Rcv(id) \in \mathcal{P}$, initially `nil`
- $Snd(id) \in \mathcal{P}$, initially `nil`
- `Hcheck`$(id) \in \mathcal{B}$, initially `no`
- `HTtpToRcv`$(id) \in \mathcal{B}$, initially `no`

Actions:

input Receive$(m, id)_{i,ttp}$
Eff: if($StatusTtp(id)$ = `idle`)
$M2(id) := m$
$Rcv(id) := ExtractRcv(m)$
$Snd(id) := ExtractSnd(m)$
$StatusTtp(id)$:= `received`

internal Check$(M_2(id), id)_{ttp}$
Pre: $StatusTtp(id)$ = `received`
Eff: if($CheckSignHash(M_2(id), id)$)
$StatusTtp(id)$:= `send_rcv`
$M3(id) := Constr\_M_3(M_2(id), id)$
$M4(id) := Constr\_M_4(M_2(id), id)$
`Hcheck`(id):=`yes`
else $StatusTtp(id)$:= `corrupt`

output Send$(M3(id))_{ttp,Rcv(id)}$
Pre: $StatusTtp(id)$ = `send_rcv`
Eff: $StatusTtp(id)$:= `send_snd`
`HTtpToRcv`(id) := `yes`

output Send$(M4(id))_{ttp,Snd(id)}$
Pre: $StatusTtp(id)$ = `send_snd`
Eff: $StatusTtp(id)$:= `done`

Tasks: {**Check**$(M_2(id), id)_{ttp}$}, {**Send**$(M3(id))_{ttp,Rcv(id)}$}, {**Send**$(M4(id))_{ttp,Snd(id)}$}

Fig. 6. Automaton TTP

The code of the TTP is shown in Figure 6. For each session, the TTP keeps the following information: the *StatusTtp* is the "program counter"; variables *Snd* and *Rcv* store the sender and the receiver for the session; variables M_2, M_3 and M_4 are used to store the corresponding messages of the protocol. By using the *CheckSignHash(m, id)* function the TTP first checks the validity of $Sig_R(S, R, ttp, h(m))$ using public key of R, then it decrypts $PK_{ttp}(k)$ using its private key, and decrypts $E_k(Sig_S(S, R, ttp, m))$ using k. Next, the TTP checks the validity of $Sig_S(S, R, ttp, m)$ using S's public key, computes $h(m)$, and compares this $h(m)$ with the one received in $Sig_R(S, R, ttp, h(m))$. If the two values match, the TTP knows that m is the mail content that S wanted to send to R and that R is willing to receive m. In this case the function *CheckSignHash(m, id)* returns `true` and the TTP is able to construct the proof-of-origin and the proof-of-delivery by using the functions $Constr\_M_3$ and $Constr\_M_4$, respectively. Moreover, we also use two history variables [2] `Hcheck` and `HTtpToRcv`. The variable `Hcheck` is set to `yes` if the TTP is able to construct the message M_3 and M_4 corresponding to the proof-of-origin and to the proof-of-delivery, respectively, whereas, the value of the history variable `HTtpToRcv` is `yes` if the TTP has sent the message M_3 to the receiver. We are now ready to describe the actions of automaton TTP top to bottom, left to right. The **Receive**$(m, id)_{i,ttp}$ action takes a message from the channel and stores it into variable M_2. The program counter *StatusTtp* is set to `received` so that the enabled action is the internal action **Check**$(M_2(id),id)_{ttp}$. The **Check**$(M_2(id),id)_{ttp}$ action checks wether it may construct the proofs of delivery and origin with the *CheckSignHash(m, id)* function. If this is not possible, the program counter *StatusTtp* is set to `corrupt` and the protocol is aborted. Otherwise, the TTP constructs the messages $M_3 = \langle Sig_{ttp}(Sig_R(S, R, ttp, h(m))), R, m\rangle$ and $M_4 = \langle Sig_{ttp}(Sig_S(S, R, ttp, m))\rangle$ and the program counter *StatusTtp* is set to `send_rcv` so that the enabled action is the **Send** to the receiver. Finally, the **Send**$(M3(id))_{ttp,Rcv(id)}$ action sets the program counter *StatusTtp* to `send_snd` and the **Send** action to the sender can be executed. The message M_4 is sent to the sender so that the program counter *StatusTtp* is update to `done`. The `done` state for this session means that this session has completed and nothing else has to be done. Actions **Check**$(M_2(id),id)_{ttp}$, **Send**$(M3(id))_{ttp,Rcv(id)}$ and **Send**$(M4(id))_{ttp,Snd(id)}$ are in three different tasks, hence, in a fair execution they get infinitely many opportunities to be executed.

[2] An history variable is a variable that is used only for the proofs but it is not necessary in the real code.

4.4 IOA Code for Channels

The code for UNREL_CHANNEL$_{i,j}$ is shown in Figure 7. The state is described by variable *Msgs* that contains the messages still in transit on the channel. It has an *input* action **Send**$(m, id)_{i,j}$ whose effect is to add a message in the set of in transit messages. Non-deterministically the automaton can execute one of the two actions in the task: {**Receive**$(m, id)_{i,j}$, **Lose**$(m, id)_{i,j}$}. The **Receive**$(m, id)_{i,j}$ action models the delivery of the message, whereas the **Lose**$(m, id)_{i,j}$ action models the loss or the alteration of a message in transit on the channel.

UNREL_CHANNEL$_{i,j}$

State:
$Msgs$, a set of elements of $\mathcal{M}$, initially empty

Actions:

input Send$(m, id)_{i,j}$
Eff: add (m, id) to $Msgs$

output Receive$(m, id)_{i,j}$
Pre: (m, id) is in $Msgs$
Eff: remove (m, id) from $Msgs$

internal Lose$(m, id)_{i,j}$
Pre: (m, id) is in $Msgs$
Eff: remove (m, id) from $Msgs$

Task: {**Receive**$(m, id)_{i,j}$, **Lose**$(m, id)_{i,j}$}

Fig. 7. Automaton UNREL_CHANNEL$_{i,j}$,

The code for CHANNEL$_{ttp,i}$ is shown in Figure 8. The automaton is described in section 2.2. We have only added the two history variables: `HChanSnd`(id) and `HChanRcv`(id). The history variable `HChanSnd`(id) models the mailing of a message from the TTP to i whereas, the variable `HChanRcv`(id) models the delivery of the message.

CHANNEL$_{ttp,i}$
State:
for each $id \in \mathcal{N}$
$Msgs$, a set of elements of $\mathcal{M}$, initially empty
`HChanSnd`$(id) \in \mathcal{B}$, initially `no`
`HChanRcv`$(id) \in \mathcal{B}$, initially `no`

Actions:

input Send$(m, id)_{ttp,i}$
Eff: add (m, id) to $Msgs$
`HChanSnd`(id)=YES

output Receive$(m, id)_{ttp,i}$
Pre: (m, id) is in $Msgs$
Eff: remove (m, id) from $Msgs$
`HChanRcv`(id)=YES

Tasks: {**Receive**$(m, id)_{ttp,i}$}

Fig. 8. Automaton CHANNEL$_{ttp,i}$

5 Correctness of CMP1 Protocol

In this section we analyze the CMP1 protocol by using the IOA model, in particular we prove that the protocol satisfies the properties shown in section 3. In the following we denote by S and R the indices corresponding to the processes which represent the sender and the receiver, respectively.
During the **Check**$(m,id)_{ttp}$ action, the TTP executes the function *CheckSignHash(m,id)* which returns `yes` if the TTP is able to construct the proof-of-origin and the proof-of-delivery corresponding to the messages M_3 and M_4, respectively. Hence, in order to show that the protocol CMP1 satisfies the properties in section 3, we have to prove the following three informal assertions:

- The sender eventually receives the message M_4 corresponding to the *proof-of-delivery* constructed by the TTP.
- The receiver eventually receives the message M_3 corresponding to the *proof-of-origin* constructed by the TTP.
- The sender eventually receives the *proof-of-delivery* if and only if the receiver eventually receives the *proof-of-origin*.

In the following we will prove several invariants that will be used to prove the above statements.

5.1 Invariants

The first invariant shows that if $StatusSnd(id)$= `done` the message M_4 of the protocol has been delivered to the sender.

Invariant 5.1 *In any reachable state s, if $s.StatusSnd(id)$=* `done` *then* $s.\text{CHANNEL}_{ttp,S}$`.HChanRec(id) = yes`.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially $StatusSnd(id)$ is `idle`. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s'. We need to prove that it is true in s for any possible step (s', π, s). If $s.StatusSnd(id) \neq$ `done`, the invariant is true. Thus, assume that $s.StatusSnd(id) =$ `done`. We have to distinguish the following two cases:

- $s'.StatusSnd(id) =$ `done`. From the inductive hypothesis it holds that $s'.channel_{ttp,S}$`.HChanRec(id) = yes`. Since $HChanRec(id)$ once set to `yes`, never changes any longer,
  it holds that $s.channel_{ttp,S}$`.HChanRec(id) = yes`.

- $s'.StatusSnd(id) \neq$ `done`. There exists only one enabled action that sets $StatusSnd(id)$ to `done`: π is the **Receive**$(m, id)_{ttp,S}$ action of the SENDER$_S$ automaton. This input action is controlled by the output action **Receive**$(m, id)_{ttp,S}$ of the CHANNEL$_{ttp,S}$ automaton.
  Since this action sets CHANNEL$_{ttp,S}$.`HChanRec(id)` to `yes`, it follows that $s.channel_{ttp,S}$.`HChanRec(id)` = `yes`. □

The next invariant states that if i receives a message from the TTP the message has been sent by the TTP.

Invariant 5.2 *In any reachable state s, if* $s.$CHANNEL$_{ttp,i}$.`HChanRec`$_{ttp,i}(id) =$ `yes` *then* $s.$CHANNEL$_{ttp,i}.HChanSnd(id) =$ `yes`.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially we have that CHANNEL$_{ttp,i}$.`HChanRec`$_{ttp,i}(id) =$ `no`. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s'. We need to prove it is true in s for any possible step (s', π, s). If we have that $s.$CHANNEL$_{ttp,i}$.`HChanRec(id)` = `no`, then the invariant is true. Thus, assume that $s.$CHANNEL$_{ttp,i}$.`HChanRec(id)` = `yes`. We have to distinguish the following two cases:

- $s'.$CHANNEL$_{ttp,i}$.`HChanRec(id)` = `yes`.
  From the inductive hypothesis it holds that $s'.$CHANNEL$_{ttp,i}$.`HChanSnd(id)` = `yes`. Since CHANNEL$_{ttp,i}$.`HChanSnd(id)` once set to `yes`, never changes any longer, it holds that $s.$CHANNEL$_{ttp,i}$.`HChanSnd(id)` = `yes`.
- CHANNEL$_{ttp,i}$.`HChanRec(id)` = `no`.
  There exists only one enabled action that sets $s.$CHANNEL$_{ttp,i}$.`HChanRec(id)` to `yes`: π is the output action **Receive**$(m, id)_{ttp,i}$ of the CHANNEL$_{ttp,i}$ automaton. The precondition of this action states that the message m is in CHANNEL$_{ttp,i}$.Msgs. There is only one action that inserts a message in CHANNEL$_{ttp,i}$.Msgs: the input action **Send**$(m, id)_{ttp,i}$ of the CHANNEL$_{ttp,i}$ automaton. This action also sets CHANNEL$_{ttp,i}$.`HChanSnd(id)` to `yes`.
  Since CHANNEL$_{ttp,i}$.`HChanSnd(id)` once set to `yes`, never changes any longer, it follows that $s.$CHANNEL$_{ttp,i}$.`HChanSnd(id)` = `yes`. □

Invariant 5.3 states that if the TTP has sent a message to the sender it has completed the protocol.

Invariant 5.3 *In any reachable state s, if* $s.$CHANNEL$_{ttp,S}$.`HChanSnd(id)` = `yes` *then* $s.StatusTtp(id) =$`done`.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially we have tha $\text{CHANNEL}_{ttp,S}$.`HChanSnd(id)` = `no`. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s'. We need to prove it is true in s' for any possible step (s', π, s). If $s.\text{CHANNEL}_{ttp,S}$.`HChanSnd(id)` = `no`, the invariant is true. Thus, assume that $s.\text{CHANNEL}_{ttp,S}$.`HChanSnd(id)` = `yes`. We have to distinguish the following two cases:

- $s'.\text{CHANNEL}_{ttp,S}$.`HChanSnd(id)` = `yes`.
  From the inductive hypothesis $s'.StatusTtp(id)$ =`done`.
  Since $StatusTtp(id)$ once set to `done` never changes any longer, it holds that $s.StatusTtp(id)$ =`done`.
- $s'.\text{CHANNEL}_{ttp,S}$.`HChanSnd(id)` = `yes`. There exists only one enabled action that sets $s.\text{CHANNEL}_{ttp,S}$.`HChanSnd(id)` to `yes`: π is the the input action $\mathbf{Send}(m, id)_{ttp,S}$ of the $\text{CHANNEL}_{ttp,S}$ automaton. This input action is controlled by the output action $\mathbf{Send}(m, id)_{ttp,S}$ of the TTP automaton. Since this action sets $StatusTtp(id)$ to `done`, it follows that $s.StatusTtp(id)$ =`done`. □

The next invariant shows that if the receiver has completed the protocol, it received the message M_3.

Invariant 5.4 *In any reachable state s, if $s.StatusRcv(id)$ =* `done` *then we have $s.\text{CHANNEL}_{ttp,R}$.*`HChanRec(id)` = `yes`.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially we have that $StatusRcv(id)$ is `idle`. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s'. We need to prove it is true in s for any possible step (s', π, s). If it holds that $s.StatusRcv(id)$ ≠`done`, the invariant is true.
Thus, assume that $s.StatusRcv(id)$ =`done`. We have to distinguish the following two cases:

- $s'.StatusRcv(id)$ =`done`. From the inductive hypothesis it holds that $s'.\text{CHANNEL}_{ttp,R}$.`HChanRec(id)` = `yes`.
  Since $\text{CHANNEL}_{ttp,R}$.`HChanRec(id)` once set to `yes`, never changes any longer, it holds that $s.\text{CHANNEL}_{ttp,R}$.`HChanRec(id)` = `yes`.
- $s'.StatusRcv(id)$ ≠ `done`. There exists only one enabled action that sets $StatusRcv(id)$ to `done`: π is the input action $\mathbf{Receive}(m, id)_{ttp,R}$ of the automaton RECEIVER_R. This input action is controlled by the output action $\mathbf{Receive}(m, id)_{ttp,R}$ of the $\text{CHANNEL}_{ttp,R}$.

Since this action sets $\text{CHANNEL}_{ttp,R}$.HChanRec(id) to yes, it follows that $s.\text{CHANNEL}_{ttp,R}$.HChanRec(id) = yes. □

The next invariant states that if the message is in transit on the channel from the TTP to the receiver, the message was sent by the TTP.

Invariant 5.5 *In any reachable state s, if $s.\text{CHANNEL}_{ttp,R}$.HChanSnd(id) = yes then s.HTtpToRcv(id) = yes.*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially we have that $\text{CHANNEL}_{ttp,R}$.HChanSnd(id) = no. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s'. We need to prove it is true in s for any possible step (s', π, s). If we have that $s.\text{CHANNEL}_{ttp,R}$.HChanSnd(id) = no, then the invariant is true. Thus, assume that $s.\text{CHANNEL}_{ttp,R}$.HChanSnd(id) = yes. We have to distinguish the following two cases:

- $s'.\text{CHANNEL}_{ttp,R}$.HChanSnd(id) = yes. From inductive hypothesis it holds that s'.HTtpToRcv(id) = yes. Since HTtpToRcv(id) once set to yes never changes any longer, it holds that s.HTtpToRcv(id)=yes.
- $s'.\text{CHANNEL}_{ttp,R}$.HChanSnd(id) = no. There exists only one enabled action that sets
  $s.\text{CHANNEL}_{ttp,R}$.HChanSnd(id) to yes:
  π is the input action $\mathbf{Send}(m, id)_{\,ttp,R}$ of $\text{CHANNEL}_{ttp,R}$. This input action is controlled by the output action $\mathbf{Send}(m, id)_{\,ttp,R}$ of the TTP automaton. Since this action sets HTtpToRcv(id) to yes, it follows that s.HTtpToRcv(id)=yes. □

Invariant 5.6 shows that if the TTP completed the protocol, then it has sent message M_3 to the receiver.

Invariant 5.6 *In any reachable state s, if $s.StatusTtp(id)$ = done then we have s.HTtpToRcv(id) = yes.*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. $StatusTtp(id)$ = idle. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s'. We need to prove it is true in s for any possible step (s', π, s). If $s.StatusTtp(id) \neq$ done, the invariant is true.
Thus, assume that $s.StatusTtp(id)$ = done. We have to distinguish the following two cases:

- $s'.StatusTtp(id)$ = `done`. From the inductive hypothesis, it follows that s'.`HTtpToRcv(id)` = `yes`. Since `HTtpToRcv(id)` once set to `yes` never changes any longer, it holds that s.`HTtpToRcv(id)`=`yes`.
- $s'.StatusTtp(id) \neq$ `done`. There exists only one action enabled that sets $s.StatusTtp(id)$ to `done`: π is the output action $\mathbf{Send}(m(id))_{ttp,Snd(id)}$ of TTP automaton. The precondition of this action claims:
  $StatusTtp(id)$ = `send-snd`. The only action that sets $StatusTtp(id)$ to `send-snd` is the output action $\mathbf{Send}(m(id))_{ttp,Rcv(id)}$ of TTP automaton. This action also sets `HTtpToRcv(id)` to `yes`. Since `HTtpToRcv(id)` once set to `yes` never changes any longer, it holds that
  s.`HTtpToRcv(id)`=`yes`. □

The next invariant shows that the TTP executes the internal action `Check` before executing its `Send` actions.

Invariant 5.7 *In any reachable state s, if $s.StatusTtp(id) \in$ {`send-rcv`, `send-snd`, `done`} then s.`Hcheck(id)` = `yes`.*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially $StatusTtp(id)$ is `idle`. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s'. We need to prove it is true in s for any possible step (s', π, s). If $s.StatusTtp(id) \notin$ {`send-rcv`, `send-snd`, `done`}, the invariant is true. Otherwise, we have to distinguish the following two cases:

- $s'.StatusTtp(id) \in$ {`send-rcv`, `send-snd`, `done`}. From the inductive hypothesis s'.`Hcheck(id)` = `yes`. Since, `Hcheck(id)` once set to `yes`, never changes any longer, it follows that s.`Hcheck(id)` = `yes`.
- $s'.StatusTtp(id) \notin$ {`send-rcv`, `send-snd`, `done`}. There exists only one enabled action that sets $StatusTtp(id)$ to a value in {`send-rcv`, `send-snd`, `done`}: π is the internal action $\mathbf{Check}(m,id)_{ttp}$ of the TTP automaton. This action also sets `Hcheck(id)` to `yes`. Therefore, s.`Hcheck(id)` = `yes`. □

Finally, the following invariant states that once the TTP has sent message M_3 to the receiver, in order to complete the protocol it only needs to send message M_4 to the sender.

Invariant 5.8 *In any reachable state s, if s.`HTtpToRcv(id)` = `yes` then we have $s.StatusTtp(id) \in$ {`send-snd`, `done`}.*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Initially s.`HTtpToRcv(id)`

= no. Hence, the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s'. We need to prove it is true in s for any possible step (s', π, s). If s.HTtpToRcv(id) = no, the invariant is true. Thus, assume that s.HTtpToRcv(id) = yes. We have to distinguish the following two cases:

- s'.HTtpToRcv(id) = yes. From the inductive hypothesis, it holds that $s'.StatusTtp(id) \in$ {send-snd,done}. If $s'.StatusTtp(id)$ = send-snd there exists only one enabled action that modifies the value of $StatusTtp(id)$: π is the output action $\mathbf{Send}(m, id)_{ttp,S}$ of TTP automaton. This action sets $StatusTtp(id)$ to done. Moreover, $StatusTtp(id)$ once set to done never changes any longer.
  It follows that $s.StatusTtp(id) \in$ {send-snd,done}.
- s'.HTtpToRcv(id) = no. There exists only one enabled action that sets s.HTtpToRcv(id) to yes: π is the output action $\mathbf{Send}(m, id)_{ttp,R}$ of the TTP automaton. This action also sets $StatusTtp(id)$ to send-snd. Therefore, $s.StatusTtp(id) \in$ {send-snd,done}. □

5.2 Non Repudiation of Destination Property

The variable M_4 of SENDER_S automaton contains a message received by the TTP. We will prove that the TTP sends this message after that the controls made by the *CheckSignHash(m,id)* function has been executed and the TTP is able to construct the proof of delivery for the sender by using the $Constr\_M4$ function. Recall that the value of the history variable Hchek(id) is yes only if the TTP may send the proof of delivery to the sender. Hence, we have to prove the following lemma:

Lemma 5.1 *In any reachable state s, if $s.StatusSnd(id)$ = done then we have that s.Hcheck(id) = yes.*

Proof. If $s.StatusSnd(id)$ = done, from Invariant 5.1 it holds that $s.\text{HChanRec}_{ttp,S}$ = yes. From Invariant 5.2 it holds that $s.\text{HChanSnd}_{ttp,S}$ = yes. Moreover, from Invariant 5.3 it holds that $s.StatusTtp(id)$ = done. Finally, from Invariant 5.7 s.Hcheck(id) = yes. □

If the history variable Hcheck(id) is set to yes, then the TTP is able to send the *proof-of-delivery* corresponding to the message M_4 to S. The next lemma says that if the TTP sends the *proof-of-delivery* to S, eventually S receives it.

Lemma 5.2 *In any fair execution, if there exists a state s' for which it holds that s'.Hcheck(id) = yes, then there exists a reachable state s such that $s.StatusSnd(id)$ = done.*

Proof. The variable `Hcheck(id)`, initially is equal to `no`. There exists only one action that sets it to `yes`: **Check**$(m,id)_{ttp}$. Since the execution is fair, the output actions **Send**$(m,id)_{ttp,R}$ and **Send**$(m,id)_{ttp,S}$ of the TTP will be executed. Moreover, the **Send**$(m,id)_{ttp,S}$ action controls the input action of the channel between TTP and the sender S. From the fair property also the output action of the channel will be executed. Finally, the `Receive` action of the channel controls the `Receive` action of the automaton SENDER$_S$. Since, this action sets $StatusSnd(id)$ to `done`, it follows that there exists a state s such that $s.StatusSnd(id) =$ `done`. □

The variable $StatusSnd$ is set to `done` after that the sender received the message M_4 from the TTP. Therefore, the next theorem easily follows from Lemma 5.1 and Lemma 5.2.

Theorem 5.3 *The CMP1 protocol satisfies the non repudiation of destination property.*

5.3 Non Repudiation of Origin Property

The variable M_3 of the $RECEIVER_R$ automaton contains a message received by the TTP. We will prove that the TTP sends this message after that the controls made by the *CheckSignHash(m,id)* function has been executed and the TTP is able to construct the proof of delivery for the sender by using the $Constr\_M3$ function. Recall that the value of the history variable `Hcheck(id)` is `yes` only if the TTP may send the proof of origin to the sender. Hence, we have to prove the following lemma:

Lemma 5.4 *In any reachable state s, if $s.StatusRcv(id) =$* `done` *then we have that $s.$*`Hcheck(id)` $=$ `yes`.

Proof: If $s.StatusRcv(id) =$ `done`, from Invariant 5.4 it holds that $s.$`HChanRec`$_{ttp,R}$ $=$ `yes`. From Invariant 5.2 it holds that $s.$`HChanSnd`$_{ttp,R} =$ `yes`. Moreover, from Invariant 5.5 and Invariant 5.8 it holds that $s.StatusTtp(id) \in \{$`send-snd`, `done`$\}$. Finally, from Invariant 5.7 $s.$`Hcheck(id)` $=$ `yes`. □

If the history variable `Hcheck(id)` is set to yes, then the TTP is able to send the *proof-of-origin* corresponding to the message M_3 to R. The next lemma says that if the TTP sends the *proof-of-origin* to R, eventually R receives it.

Lemma 5.5 *In any fair execution, if there exists a state s' for which it holds that $s'.$*`Hcheck(id)` $=$ `yes`*, then there exists a reachable state s such that $s.StatusRcv(id) =$* `done`.

Proof. The variable `Hcheck(id)`, initially is equal to `no`. There exists only one action that sets it to `yes`: **Check**$(m,id)_{ttp}$. Since the execution is fair, the output actions **Send**$(m,id)_{ttp,R}$ of the TTP will be executed. Moreover, this action controls the input action of the channel between TTP and the sender S. From the fair property also the output action of the channel will be executed. Finally, the `Receive` action of the channel controls the `Receive` action of the automaton RECEIVER$_R$. Since, this action sets $StatusRcv(id)$ to `done`, it follows that there exists a state s such that $s.StatusSnd(id) =$ `done`. □

The variable $StatusRcv$ is set to `done` after that the receiver received the message M_3 from the TTP. Therefore, the next theorem easily follows from Lemma 5.4 and Lemma 5.5.

Theorem 5.6 *The CMP1 protocol satisfies the non repudiation of origin property.*

5.4 Fairness Property

The next lemma states that the sender receives the prof-of-delivery if and only if the receiver receives the proof-of-origin.

Lemma 5.7 *In any fair execution, there exists a state s for which it holds that $s.StatusRcv(id) =$* `done` *if and only if there exists a state s' such that $s'.StatusSnd(id) =$* `done`*.*

Proof: Assume that there exists a state s such that $s.StatusRcv(id) =$ `done`. From Invariants 5.4, 5.2 and 5.5 it holds that $s.$`HTtpToRcv(id)` = `yes`. This variable, initially is equal to `no` and there exists only one action that sets it to `yes`: **Send**$(m,id)_{ttp,R}$. Since the execution is fair, the output action **Send**$(m,id)_{ttp,S}$ of the TTP will be executed. Moreover, this action controls the input action of the channel between TTP and the sender S. From the fair property also the output action of the channel will be executed. Finally, the `Receive` action of the channel controls the `Receive` action of the automaton SENDER$_S$. Since, this action sets $StatusSnd(id)$ to `done`, it follows that there exists a state s' such that $s'.StatusSnd(id) =$ `done`.

Conversely, assume that there exists a state s' such that $s'.StatusSnd(id)$ = `done`. From Invariant 5.1, Invariant 5.2 and Invariant 5.3, it holds that $s'.StatusTtp(id)$
= `done`. From Invariant 5.6 it holds that $s'.$`HTtpToRcv(id)` = `yes`. This variable, initially equal to `no`, is set to `yes` in the `Send` action of the TTP to the receiver R. This action controls the input action of the channel between TTP and the receiver R. Since the execution is fair also the output action of the channel will be executed. Finally, the `Receive` action of the channel controls

the `Receive` action of the automaton RECEIVER_R. Since this action sets $StatusRcv(id)$ to `done`, there exists a state s such that $s.StatusRcv(id) =$ `done`. □

The next theorem easily follows from Lemma 5.7.

Theorem 5.8 *The CMP1 protocol satisfies the fairness property.*

6 Conclusions

Describing and reasoning about asynchronous distributed systems is often a difficult and error prone task. The I/O Automaton [7] provides a framework allowing both a precise description of the code and the possibility of very detailed proofs. In this paper we carry out a simple experiment in using the IOA as a tool to describe and to reason about cryptographic protocols running in an asynchronous distributed system. We showed the feasibility of the approach by examining the security properties of the Deng's certified email protocol and proving its correctness. We are planning to extend these ideas to the modeling of more complex protocols for certified email [4].

References

[1] Asokan, N., M. Schunter and M. Waidner, *Optimistic protocols for fair exchange*, in: *ACM Conference on Computer and Communications Security*, 1997, pp. 7–17.

[2] Asokan, N., V. Shoup and M. Waidner, *Asynchronous protocols for optimistic fair exchange*, in: *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1998, pp. 86–99.

[3] Bahreman, A. and J. D. Tygar, *Certified electronic mail*, in: D. Nesset and R. Shirey, editors, *Proceedings of the Symposium on Network and Distributed Systems Security*, Internet Society, San Diego, CA, 1994, pp. 3–19.

[4] Blundo, C., S. Cimato and R. D. Prisco, *Certified email: Design and implementation of a new optimistic protocol*, in: *Proceedings of the 8th IEEE Symposium on Computers and Communications (ISCC'03)*, 2003, pp. 828–838.

[5] Deng, R. H., L. Gong, A. A. Lazar and W. Wang, *Practical protocols for certified electronic mail*, Journal of Network and System Management **4** (1996).

[6] Fekete, A., N. Lynch and A. Shvartsman, *Specifying and using a partitionable group communication service*, ACM Transactions on Computer Systems **19** (2001), pp. 171–216.

[7] Lynch, N., "Distributed Algorithms," Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[8] Lynch, N. A., *I/O automaton models and proofs for shared-key communication systems*, Technical Report MIT/LCS/TR-789, MIT Laboratory for Computer Science, Cambridge, MA, USA (1999).

[9] Micali, S., *Certified email with invisible post offices*, Technical report, available from author; an invited presentation at the RSA '97 conference (1997).

[10] Myers, J. and M. Rose, *Rfc 1939 - post office protocol - version 3*, URL http://www.ietf.org/rfc/rfc882.txt.

[11] Pfitzmann, B., M. Schunter and M. Waidner, *Provably secure certified mail*, Technical Report RZ 3207 (#93253), Universitat des Saarlandes (2000).

[12] Postel, J. B., *Rfc 821 - simple mail transfer protocol*, URL http://www.ietf.org/rfc/rfc882.txt.

[13] Prisco, R. D., B. Lampson and N. Lynch, *Fundamental study: Revisiting the paxos algorithm*, Theoretical Computer Science **243** (2000), pp. 35–91.

[14] Riordan, J. and B. Schneier, *A certified E-mail protocol with no trusted third party*, in: *Proceedings of the 13th Annual Computer Security Applications Conference*, 1998, pp. 347–352.

[15] Zhou, J. and D. Gollmann, *A fair non-repudiation protocol*, in: *Proceedings of the IEEE Symposium on Research in Security and Privacy* (1996), pp. 55–61.