 Electronic Notes in Theoretical Computer Science 99 (2004) 339–359 

[www.elsevier.com/locate/entcs](http://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](#_bookmark9)

*Dipartimento di Informatica ed Applicazioni Universit`a 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](#_bookmark32)], give its formalization using the IOA model, and prove that some security properties are satisfied during the execution of the protocol.

# 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](mailto:carblu%2Ccimato%2Crobdep%2Cferrara@dia.unisa.it)

1571-0661 © 2004 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

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](#_bookmark37)] and Post Office Protocol [[10](#_bookmark38)], 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](#_bookmark30),[5](#_bookmark32),[14](#_bookmark39),[15](#_bookmark42)], the ttp is actively involved in each message exchange. In optimistic protocols [[1](#_bookmark28),[2](#_bookmark29),[9](#_bookmark36)], 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](#_bookmark32)], and present its formal model relying on the Input/Output Automaton [[7](#_bookmark34)], (IOA for brief), framework. IOA provides a framework allowing both a precise description of the code and the possibility of very detailed proofs [[6](#_bookmark33),[13](#_bookmark40)]. 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](#_bookmark35)], 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](#_bookmark28)] has been analyzed in [[11](#_bookmark41)], 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 Sec- tion [3](#_bookmark2) we describe Deng’s email protocol, and in Section [4](#_bookmark4) we present its IOA formalization. Finally, the correctness of the protocol is provided in Section [5](#_bookmark13), where non repudiation properties for the origin and destination and fair- ness properties are shown to hold with the help of invariant assertion proofs. Conclusions are drawn in Section [6](#_bookmark27).

# The Framework

We consider a distributed system consisting of *n* nodes (processors) P =

{1*,* 2*,..., 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 P∪ {ttp}.

* 1. *Cryptographic Primitives*

The cryptographic primitives used in this paper are:

* + - *SigA* (*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.
    - *P KB* (*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.
    - *Ek*(*m*): denotes the encryption of message *m* using the key *k* under some symmetric encryption algorithm.
  1. *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 associ- ated with named *actions*. Figure [1](#_bookmark1) shows an automaton that models a channel

channel*i,j*

State:

*Msgs*,a set of elements of M, initially empty

Actions:

input Send(*m*)*i,j* output Receive(*m*)*i,j*

Eff: add *m* to *Msgs* 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 vari- ables 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 repre- sents 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 transi- tions, 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*:

1. If *α* is finite, then *C* is not enabled in the final state of *α*.
2. 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](#_bookmark34)], Chapters 8 and 23, for more information about the IOA models.

# The CMPl Protocol

We now describe the CMP1 protocol for certified mail presented by Deng et al. in [[5](#_bookmark32)]. A concise representation of protocol message flow is provided in Figure [2](#_bookmark3).

Sender S

Thrusted Third Party TTP

Receiver R

*M*1 =*< S, R, ttp, h*(*m*)*,P Kttp* (*k*)*, Ek* (*SigS* (*S, R, ttp, m*)) *>*

*M*2 =*< SigR* (*S, R, ttp, h*(*m*))*,P Kttp* (*k*)*, Ek* (*SigS* (*S, R, ttp, m*)) *>*

*M*4 =*< Sigttp* (*SigR* (*S, R, ttp, h*(*m*))) *> M*3 *< Sigttp* (*SigR* (*S, R, ttp, h*(*m*)))*, R,m >*

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 *SigS* (*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 mes- sage *M*1 = ⟨*S, R, ttp, h*(*m*)*,P Kttp* (*k*)*, Ek*(*SigS* (*S, R, ttp, m*))⟩ 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 = ⟨*SigR* (*S, R, ttp, h*(*m*))*,P Kttp* (*k*)*, Ek*(*SigS* (*S, R, ttp, m*))⟩ to ttp. Upon receiving this message, the ttp first checks the validity of *SigR* (*S, R, ttp, h*(*m*)) using public key of *R*. Then, it decrypts *P Kttp* (*k*) using its private key, and decrypts *Ek*(*SigS* (*S, R, ttp, m*)) using *k*. Next, the ttp checks the validity of *SigS* (*S, R, ttp, m*) using *S*’s public key, computes *h*(*m*), and compares this *h*(*m*) with the one received in *SigR* (*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 = ⟨*Sigttp* (*SigR* (*S, R, ttp, h*(*m*)))*, R, m*⟩ corresponding to the proof-of-origin and *M*4 = ⟨*Sigttp* (*SigS* (*S, R, ttp, m*))⟩ 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.

# 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* ∈ 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* ∈ P∪

{*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* ∈ 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: channel*ttp,i*, and an unreliable one: UNREL CHANNEL*i,j*, for any *i* ∈ P and *j* ∈ P∪ {*ttp*}. The overall system is described by the composition of all the above automata. Figure [3](#_bookmark5) gives an overview of the automata that compose the system.

*Unreliable channeli,j*

*SENDERi*

*RECEIV ERj*

*Unreliable channeli,ttp*

*Channelttp,j*

*TTP*

*Channelttp,i*

*Unreliable channelj,ttp*

*RECEIV ERi*

*SENDERj*

*Unreliable channelj,i*

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

* 1. *IOA Code for the Sender*

The code of sender*i* is shown in Figure [4](#_bookmark6). 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 S = {idle*,* send*,* wait*,* done}

State:

for each *id* ∈ N

*StatusSnd*(*id*) ∈ S, initially idle *M1*(*id*) ∈ M, initially nil *M4*(*id*) ∈ 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(*M* 1(*id*)*, id*)*i,j* Pre: *StatusSnd*(*id*)= send Eff: *StatusSnd*(*id*):= wait

Tasks: {Send(*M* 1(*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 = ⟨*S, R, ttp, h*(*m*)*,P Kttp* (*k*)*,*

*Ek*(*SigS* (*S, R, ttp, m*))⟩ 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.

* 1. *IOA Code for the Receiver*

The code of receiver*i* is shown in Figure [5](#_bookmark7). As for the sender, state variables are indexed by a session id. Again, the state variable *StatusRcv* is the “pro- gram counter”. Variables *M1*, *M2* and *M3* are used to store the corresponding messages of the protocol.

receiver*i*

Let S = {idle*,* received*,* wait*,* discarded*,* done}

State:

for each *id* ∈ N

*StatusRcv*(*id*) ∈ S, initially idle *M1*(*id*) ∈ M, initially nil *M2*(*id*) ∈ M, initially nil *M3*(*id*) ∈ 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 = ⟨*SigR* (*S, R, ttp, h*(*m*))*,P Kttp* (*k*)*, Ek*(*SigS* (

*S, R, ttp, m*))⟩ 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.

* 1. *IOA Code for the Trusted Third Party*

ttp

Let S = {idle*,* received*,* send rcv*,* send snd*,* corrupt*,* done}

State:

for each *id* ∈ N

*StatusTtp*(*id*) ∈ S, initially idle *Rcv*(*id*) ∈ P, initially nil

*M2*(*id*) ∈ M, initially nil *Snd*(*id*) ∈ P, initially nil

*M3*(*id*) ∈ M, initially nil Hcheck(*id*) ∈ B, initially no

*M4*(*id*) ∈ M, initially nil HTtpToRcv(*id*) ∈ 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](#_bookmark8). 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 valid- ity of *SigR* (*S, R, ttp, h*(*m*)) using public key of *R*, then it decrypts *P Kttp* (*k*) using its private key, and decrypts *Ek*(*SigS* (*S, R, ttp, m*)) using *k*. Next, the ttp checks the validity of *SigS* (*S, R, ttp, m*) using *S*’s public key, computes *h*(*m*), and compares this *h*(*m*) with the one received in *SigR* (*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](#_bookmark10) 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 *StatusT tp* 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 = ⟨*Sigttp* (*SigR* (*S, R, ttp, h*(*m*)))*, R, m*⟩ and *M*4 = ⟨*Sigttp* (*SigS* (*S, R, ttp, m*))⟩ 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 noth- ing 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.

* 1. *IOA Code for Channels*

The code for UNREL CHANNEL*i,j* is shown in Figure [7](#_bookmark11). 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 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*

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

internal Lose(*m, id*)*i,j*

Pre: (*m, id*) is in *Msgs*

Eff: remove (*m, id*) from *Msgs*

Fig. 7. Automaton UNREL CHANNEL*i,j* ,

The code for channel*ttp,i* is shown in Figure [8](#_bookmark12). The automaton is described in section [2.2](#_bookmark0). 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* ∈ N

*Msgs*, a set of elements of M, initially empty

HChanSnd(*id*) ∈ B, initially no HChanRcv(*id*) ∈ 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*

# 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](#_bookmark2). 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](#_bookmark2), 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.

* 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.*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*) /= 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.channelttp,S .*HChanRec(id) = yes.

* *s*'*.StatusSnd*(*id*) /= 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* au- tomaton. 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.channelttp,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* au- tomaton. 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* au- tomaton. 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](#_bookmark16) 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.StatusT tp*(*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 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*.channel*ttp,S .*HChanSnd(id) = no, the invariant is true. Thus, assume that *s*.channel*ttp,S .*HChanSnd(id) = yes. We have to distinguish the following two cases:

* *s*'*.*channel

*ttp,S*

*.*HChanSnd(id) = yes*.*

From the inductive hypothesis *s*'*.StatusT tp*(*id*) =done.

Since *StatusT tp*(*id*) once set to done never changes any longer, it holds that *s.StatusT tp*(*id*) =done.

* *s*'*.*channel*ttp, .*HChanSnd(id) = yes*.* There exists only one enabled action that sets *s*.channel*ttp,S .*HChanSnd(id) to yes: *π* is the the input action Send(*m, id*)*ttp,S* of the channel*ttp,S* automaton. This input action is controlled by the output action Send(*m, id*)*ttp,S* of the ttp automaton. Since this action sets *StatusT tp*(*id*) to done, it follows that *s.StatusT tp*(*id*) =done. 

*S*

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.*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 fol-

lowing two cases:

* *s*'*.StatusRcv*(*id*) =done*.* From the inductive hypothesis it holds that

*s*'.channel*ttp,*

*R*

*.*HChanRec(id) = yes.

Since channel*ttp,R .*HChanRec(id) once set to yes, never changes any longer,

it holds that *s*.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 Receive(*m, id*)*ttp,R* of the au- tomaton receiver*R*. This input action is controlled by the output action Receive(*m, id*)*ttp,R* of the channel*ttp,R* .

Since this action sets channel*ttp,R .*HChanRec(id) to yes, it follows that

*s*.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.*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 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*.channel*ttp,R .*HChanSnd(id) = no, then the invariant is true. Thus, assume that *s*.channel*ttp,R .*HChanSnd(id) = yes. We have to distinguish the following two cases:

* *s*'*.*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*'*.*channel*ttp, .*HChanSnd(id) = no*.* There exists only one enabled action that sets

*R*

*s*.channel*ttp,R .*HChanSnd(id) to yes:

*π* is the input action Send(*m, id*) *ttp,R* of channel*ttp,R* . This input action is con- trolled by the output action 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](#_bookmark19) 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.StatusT tp*(*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. *StatusT tp*(*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.StatusT tp*(*id*) /= done, the invariant is true.

Thus, assume that *s.StatusT tp*(*id*) = done. We have to distinguish the fol-

lowing two cases:

* *s*'*.StatusT tp*(*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*'*.StatusT tp*(*id*) /= done*.* There exists only one action enabled that sets *s.StatusT tp*(*id*) to done: *π* is the output action Send(*m*(*id*))*ttp,Snd*(*id*) of ttp automaton. The precondition of this action claims:

*StatusT tp*(*id*) = send-snd. The only action that sets *StatusT tp*(*id*) to send-snd is the output action 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.StatusT tp*(*id*) ∈ {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 *StatusT tp*(*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.StatusT tp*(*id*) /∈

{send-rcv, send-snd, done}, the invariant is true. Otherwise, we have to dis- tinguish the following two cases:

* *s*'*.StatusT tp*(*id*) ∈ {send-rcv*,* send-snd*,* done}*.* From the inductive hy- pothesis *s*'.Hcheck(id) = yes. Since, Hcheck(id) once set to yes, never changes any longer, it follows that *s*.Hcheck(id) = yes.
* *s*'*.StatusT tp*(*id*) /∈ {send-rcv*,* send-snd*,* done}*.* There exists only one enabled action that sets *StatusT tp*(*id*) toa value in {send-rcv, send-snd, done}: *π* is the internal action 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.StatusT tp*(*id*) ∈ {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*'*.StatusT tp*(*id*) ∈{send-snd,done}. If *s*'*.StatusT tp*(*id*)= send-snd there exists only one enabled action that modifies the value of

*StatusT tp*(*id*): *π* is the output action Send(*m, id*) *ttp,S* of ttp automaton. This action sets *StatusT tp*(*id*) to done. Moreover, *StatusT tp*(*id*) once set to done never changes any longer.

It follows that *s.StatusT tp*(*id*) ∈ {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 Send(*m, id*) *ttp,R* of the ttp automaton. This action also sets *StatusT tp*(*id*) to send-snd. Therefore, *s.StatusT tp*(*id*) ∈ {send-snd,done}.
  1. *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 M* 4 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](#_bookmark14) it holds that *s*.HChanRec*ttp,S* = yes. From Invariant [5.2](#_bookmark15) it holds that *s*.HChanSnd*ttp,S* = yes. Moreover, from Invariant [5.3](#_bookmark16) it holds that *s.StatusT tp*(*id*) = done. Finally, from Invariant [5.7](#_bookmark20) *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](#_bookmark22) and Lemma [5.2](#_bookmark23).

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

* 1. *Non Repudiation of Origin Property*

The variable *M*3 of the *RECEIV ERR* 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 M* 3 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](#_bookmark17) it holds that

*s*.HChanRec*ttp,R* = yes. From Invariant [5.2](#_bookmark15) it holds that *s*.HChanSnd*ttp,R* = yes.

Moreover, from Invariant [5.5](#_bookmark18) and Invariant [5.8](#_bookmark21) it holds that *s.StatusT tp*(*id*) ∈ {send-snd, done}. Finally, from Invariant [5.7](#_bookmark20) *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](#_bookmark24) and Lemma [5.5](#_bookmark25).

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

* 1. *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](#_bookmark17), [5.2](#_bookmark15) and [5.5](#_bookmark18) 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 ac- tion 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](#_bookmark14), Invariant [5.2](#_bookmark15) and Invariant [5.3](#_bookmark16), it holds that

*s*'*.StatusT tp*(*id*)

= done. From Invariant [5.6](#_bookmark19) 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](#_bookmark26).

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

# Conclusions

Describing and reasoning about asynchronous distributed systems is often a difficult and error prone task. The I/O Automaton [[7](#_bookmark34)] 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](#_bookmark31)].

# 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.