NTNU
Norwegian University of
Science and Technology

Identity Protection, Secrecy and Authentication in Protocols with compromised Agents

Anders Lindholm Båtstrand

Master of Science in Physics and Mathematics
Submission date: July 2009
Supervisor: Kristian Gjøsteen, MATH
Co-supervisor: Cas Cremers, ETH Zürich

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

This paper aims to extend a current model for verification of security protocols to include identity protection, secrecy and two forms of authentication in protocol sessions with compromised agents.

Assignment given: 27. January 2009
Supervisor: Kristian Gjøsteen, MATH

Preface

This paper is written in spring 2009 as the course TMA4900, Mathematics, Master Thesis, at the Norwegian University for Science and Technology (NTNU).

For some example protocols, the Scyther tool has been used. This is a computer program for automatic verification of security protocols, written and maintained by Dr. Cas Cremers[5].

This report is written using the document preparation system Latex. Protocol examples are written as message sequence charts using the Latex package msc[19].

The author wishes to thank Cas Cremers at the ETH Zurich for guidance and advice during the semester.

Tønsberg, July 30, 2009 *Anders Båtstrand*

Summary

The design of security protocols is given an increasing level of academic interest, as an increasing number of important tasks are done over the Internet. Among the fields being researched is formal methods for modeling and verification of security protocols. One such method is developed by Cremers and Mauw[7]. This is the method we have chosen to focus on in this paper.

The model by Cremers and Mauw specifies a mathematical way to represent security protocols and their execution. It then defines conditions the protocols can fulfill, which is called security requirements. These typically states that in all possible executions, given a session in which all parties are honest, certain mathematical statements hold.

Our aim is to extend the security requirements already defined in the model to allow some parties in the session to be under control of an attacker, and to add a new definition of identity protection. This we have done by slightly modifying the model, and stating a new set of security requirements.

Contents

1 Introduction

The goal of this thesis is to expand current formal methods for security protocols to model and reason about secrecy, authentication and identity protection with compromised agents in the session in which we want the property to hold. We will formalize security requirements for this intruder model in the trace model developed by Cremers and Mauw[7]. By doing this, we take the model a first step towards inclusion of privacy.

We perform a case study to analyze the usefulness of our generic definitions. This consists of proposing and verifying a protocol for contract signing. Our scenario is that a user wants to sign a contract a store offers. The digital signature of the user is stored at a notary, and the user wants to sign the contract without the intruder or the notary learning neither the contract nor the identity of the store.

1.1 Related work

Our work aims to build on the formal methods presented by Cremers et al.[7, 8, 9], by allowing compromised agents in the protocol sessions, and introduce a notion of anonymity called identity protection.

The terminology for privacy and anonymity has been changing over the years, but now it seems most authors agree on the definitions proposed by Pfitzmann and Köhntopp in 2001[21], which we also use. Identity protection, however, is not a part of this proposal.

Anonymity is identified as one of the current challenges in protocol verification[20], and in the last decades, privacy has been a topic for work among computer scientists. For instance, metrics to measure anonymity has been developed, such as the *anonymity set* by Chaum in 1988[4]. More recent works on the degree of anonymity are found in [12, 22, 11]. We will, however, consider the set of all users as the anonymity group.

In a symbolic model similar to the one we use, Mauw et al. have formalized a notion of anonymity using the mentioned anonymity set[18]. The main difference is that they work in a model in which the intruder can only eavesdrop messages, a so-called passive attacker. We have chosen to allow an active attacker, and also let him compromise some of the participants in the session in which we want identity protection to hold.

Recent work on intruder models with compromised agents has, in the model we use, been done by Basin and Cremers[2]. They introduce a hierarchy of how and to what degree protocol participants are compromised. We will only consider honest and fully compromised participants.

Our case study contains contract signing in which the involved parties sign the contract after each other. This is called asymmetric contract signing, and is a simpler variant as opposed to the symmetric one as studied by e.g. Ralf Küsters et al. in [15].

1.2 Thesis outline

We begin in Section 2 with an informal discussion of the case study. In particular, we discuss the functional requirements of the protocol, the threat model, and security requirements it should fulfill.

In Section 3, we present the model for our formal reasoning. This consists of most parts of the existing model, and changes and extensions necessitated by our new intruder model, and our notion of identity protection.

A proposed protocol for our case study is presented in Section 4.

The case study is carried out in Section 5. Here we formalize the threat model and the security requirements for our scenario, using the semantics of our framework. Then we validate some of our definitions by testing them on our proposed protocol.

The paper is concluded with a summary of achievements and suggestions for future work in Section 6.

1.3 Contributions

The model presented in Section 3 builds upon the one presented by Cremers et al.[1, 5, 7, 8, 9]. We have extended the model to enable our definition of identity protection, and we have added new versions of already existing security requirements. While the existing definitions required all parties in the protocol session in which the claim is tested to be honest, we have relaxed this requirement to require only a subset of the parties to be honest.

The following security requirements make up our contribution to the model:

- Group secrecy.
- Partial injective and partial non-injective agreement.
- Term agreement and partial term agreement.
- Partial injective and partial non-injective synchronization.
- Identity protection and group identity protection.

2 Requirements

We perform a case study to analyze the usefulness of our generic definitions. The scenario we consider is a user and a store signing a contract, using a notary as a trusted third party.

This section first discusses what a protocol designed for this scenario should do. Then we discuss the threat model, and set up some informal security requirements for the protocol. These we require to hold even when certain combinations of the store, the user or the notary are played by dishonest agents.

2.1 Functional requirements

A typical example for our scenario could be a person (the *user*) searching the Internet for short-time work. He finds an offer (from the *store*), and downloads the contract. Instead of signing it by hand, he wants to do it online. For this purpose he has a government-approved company (the *notary*), which can sign documents electronically on his behalf. He fills out his personal details and sends the contract to the store, which signs and returns it. Then he authenticates with his notary, and sends the contract there. The notary then signs it with the user's signature, and sends it back. The user now has a contract signed by both him and the store, a copy of which he sends to the store.

We generalize this scenario, and divide it into three phases:

Information phase Here the user acquires the contract, and tells the store he wants to sign it.

Signing phase The store signs the contract, and then the notary signs it on behalf of the user.

Distribution phase Here the signed contract is sent to the user and the store, leaving them both with a proof of the agreement.

The functional requirement is to enable the above described transaction. In Figure 1 we see an informal depiction of the scenario protocol.

2.2 Threat model

We here specify the threat model. We do this in three steps, in which the threat level increases. We also give examples of different threat scenarios.

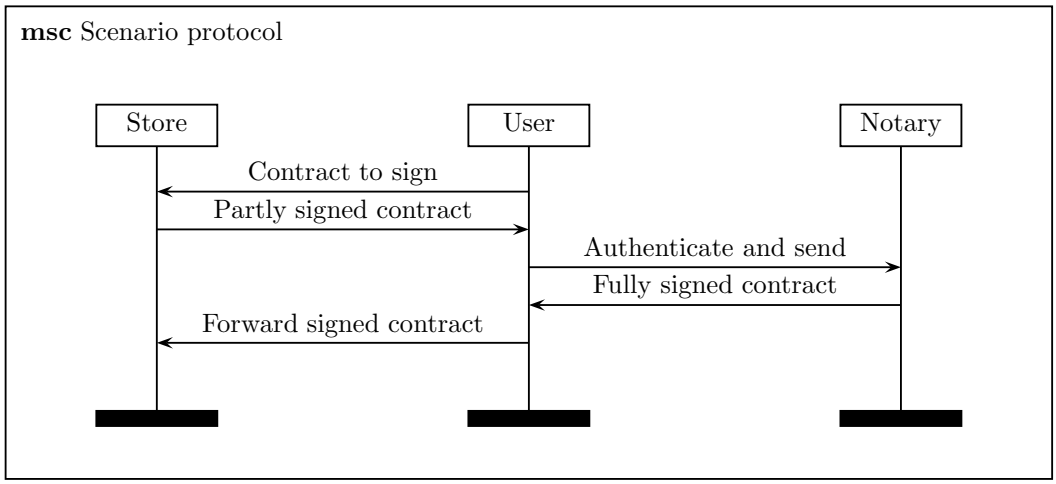


Figure 1: Informal message sequence chart of the scenario protocol.

2.2.1 Model 1: Dolev-Yao attack model

Considering that a natural environment for the protocol is the Internet, we allow a full Dolev-Yao attack model, as presented in [13]:

- The attacker has complete control over the network. This means he can see all messages, and block them if he chooses. He can also inject messages.
- The attacker is a legitimate user of the network, and honest users might choose to initiate protocol sessions with him.

In this level, we require no security if the attacker plays a role of in protocol session we inspect. However, he might do so someplace else, and use the knowledge he gains there. How this works will become clear as the paper progresses.

Attack scenarios in this model could be another store trying to get the content of the contract. The motivation could be to adjust the price he is offering for something similar.

2.2.2 Model 2: Compromised user and store

The next level in addition allows the attacker to play a role in the protocol session in which we want security requirements to hold. Which role he his allowed to play will depend on the security requirement we look at.

A dishonest store could try to trick the user into signing a contract different from what the user intend to sign. The same way, a dishonest user could also have interests in changing the contract.

2.2.3 Model 3: Partly compromised notary

In addition to the above two models, we add the possibility that the notary is partly compromised. We will not allow him to be fully compromised, as he controls the signature of the user. Instead, we require that the notary will always act according to the protocol, but allow some information he gains during the protocol execution to be leaked to the intruder.

An insider at the notary might conspire with the attacker, and try to gather information about the store or the content of the contract. For instance, he might want to know if someone is signing the contract of a competitor.

2.3 Security requirements

Security requirements are claims of what should not be possible. We here discuss informal requirements from the view of the three actors. This corresponds to how our model defines security requirements as part of a role specification.

Note that the time dimension is not considered, as it is not a part of the current model, and not in our additions. Typically, we would otherwise have required that only a certain time can pass between the store signs the contract, and it is signed by the notary.

2.3.1 The store

To be sure only the store can sign contracts with his signature, his signing key must be secret in all attack models.

The store wants to be sure that given an honest user, the contract is secret, also from a partly compromised notary.

Further, the store and the user should agree on the content of the contract. In no cases should it be possible for one of them to end up with a contract signed by both parties, different from a contract the other party has previously signed.

Another issue is that the store might want to know if a valid contract was produced in an unfinished session, since it might bind him to some actions. The store would want a copy of the signed contract, to present in case of a dispute. This relates to the concept of *fairness* in contract signing. We will not analyze this using our model, but we will briefly discuss it here.

Fairness means that among the two parties signing a contract, no one should be able to exclusively end up with a binding contract. To see how this might be a problem, consider a price being agreed upon, which in the future might rise or fall. Then it might be of interest to be the only one having the contract, to force through a transaction if desirable.

After the notary has signed the contract for the user, it might be binding even if the communication is stopped. That way, the user might keep the contract without giving it to the store. He could then choose to only use it when it gains him. A solution could be that the store, after the protocol has timed out, asks the notary if it did sign the contract, and eventually asks for a copy directly from him. We will, however, not discuss this problem further, and instead refer the reader to [3] and [15].

2.3.2 The user

The user wants to control how much information is available to the other parties. He might want to keep the content of the contract and the identity of the store secret, even to a partly compromised notary.

Further, the user wants to be in control of what is signed on behalf of him. That means that only the document he picked in the same session as he authenticated with the notary, should be signed in that session. To be sure only he can authenticate with the notary, the password should be secret. Both these requirements should hold with a compromised store.

After execution of the protocol, the user should agree with the store about the content of the contract. The user also wants it to be impossible for the notary to guess the contract used. For instance, the notary might want to take actions if the user signs the standard contract of a competitor notary. With this in mind, the notary could check some standard contracts against what he is asked to sign, as pointed out in a similar scenario in [14]. These two requirements should hold with a partly compromised notary.

2.3.3 The notary

The key used for signing must remain secret for all attack models.

The notary signs contracts on behalf of the user, and wants to be sure that the real user authenticated and indented to sign the contract that was actually signed. As a requirement for this, the notary wants the password they share to remain secret. This should hold with a compromised store.

3 Security protocol model

In this section we present the semantics in which we model security protocols. We begin with a presentation of the model and notation used, as presented by Cremers et al. in [7], [8] and [9], incorporating some of the changes done in [2].

Last in this section, we try to provide intuition for the model by discussing how it relates to actual protocols running over the Internet.

3.1 Formal semantics for security protocols

Protocols can be viewed as a specification of how agents playing roles interact over a communication network. The interaction is done by sending and receiving certain messages. Security protocols have in addition some claims about what should not be possible. These are called security requirements, and are the topic of Section 3.2.

We begin with notation for how to specify a protocol role. We then explain how the specification is realized into an actual run of the protocol. This takes us to different states of the system, and how the agents or the intruder change the state. One possible path of state transitions is called a trace, which gives us the semantics needed.

3.1.1 Role specifications

A protocol consists of a finite set of roles from the set *Role*. Each role is specified by a list of events from the set *RoleEvent*.

Roles interact in events by sending and receiving messages from the set *RoleMess*. The sending and receiving of the message m we denote by $send_l(m)$ and $recv_l(m)$. All events are tagged with a unique label l from the set *Label*, except for corresponding send and receive events, which is given the same label. The sender and recipient of the message is not required to be included as part of the message.

To facilitate discussion of security properties local to each role, we also include security claims in the event list. These we denote as $claim_l(c)$, where c is from the set *Claim*, which we will populate in Section 3.2.

Role messages are built up by combining terms from different sets. We define a set of assignable variables *Var*, a set of constants *Const*, a set of roles *Role*, and a set of functions *Func*. We denote the encryption of the term x by the key y as $\{\!|x|\!\}_y$, and can then define *RoleMess* recursively as:

$$\begin{aligned} \mathit{RoleMess} \equiv\ & \mathit{Var} \mid \mathit{Role} \mid \mathit{Const} \mid \mathit{Func}(\mathit{RoleMess}) \mid \\ & (\mathit{RoleMess}, \mathit{RoleMess}) \mid \{\!|\mathit{RoleMess}|\!\}_{\mathit{RoleMess}}. \end{aligned}$$

In this context we also introduce the subterm operator $\sqsubset$, defined in the obvious way based on the composition of terms. With this notation, a term is not a subterm of itself. If that is needed, we will use $\sqsubseteq$.

The functions we will use are $pk(R)$ and $sk(R)$ for the public and private (secret) key of the role R, respectively, $k(R, I)$ for the symmetric key known to roles R

and I, and $h(m)$ for the hash of m. The public keys of all agents are considered known to all parties of the protocol and the intruder.

For all terms we assume the existence of an inverse, where t^{-1} denotes the inverse key of t. In particular, we have that $pk(R)^{-1} = sk(R)$ and $sk(R)^{-1} = pk(R)$. For all other terms, $t^{-1} = t$ unless otherwise stated.

We then define a security protocols as a map *SecProt* from roles to lists of events:

$$\begin{aligned} \mathit{SecProt} &: \mathit{Role} \rightarrow \mathit{RoleEvent}^* \\ \mathit{RoleEvent} &\supseteq \{\mathit{send}_l(m), \mathit{recv}_l(m), \mathit{claim}_l(c) \mid \\ &\qquad l \in \mathit{Label}, m \in \mathit{RoleMess}, c \in \mathit{Claim}\} \end{aligned}$$

For a security protocol p, we will for readability denote the domain of *SecProt* as $\mathit{rolesof}(p)$.

We introduce three utility functions taking us from an event label in a protocol p, to the role having that event in the event list, denoted by $p(R)$ for role R:

$$\begin{aligned} \mathit{sendrole}(l) &= R \text{ if } \mathit{send}_l(\cdot) \in p(R) \\ \mathit{recvrole}(l) &= R \text{ if } \mathit{recv}_l(\cdot) \in p(R) \\ \mathit{claimrole}(l) &= R \text{ if } \mathit{claim}_l(\cdot) \in p(R) \end{aligned}$$

Here, and in the rest of the document, we use $\cdot$ to express that the arguments placed here does not matter in the current context. Above, it means that the actual message of the events does not matter for the utility functions defined.

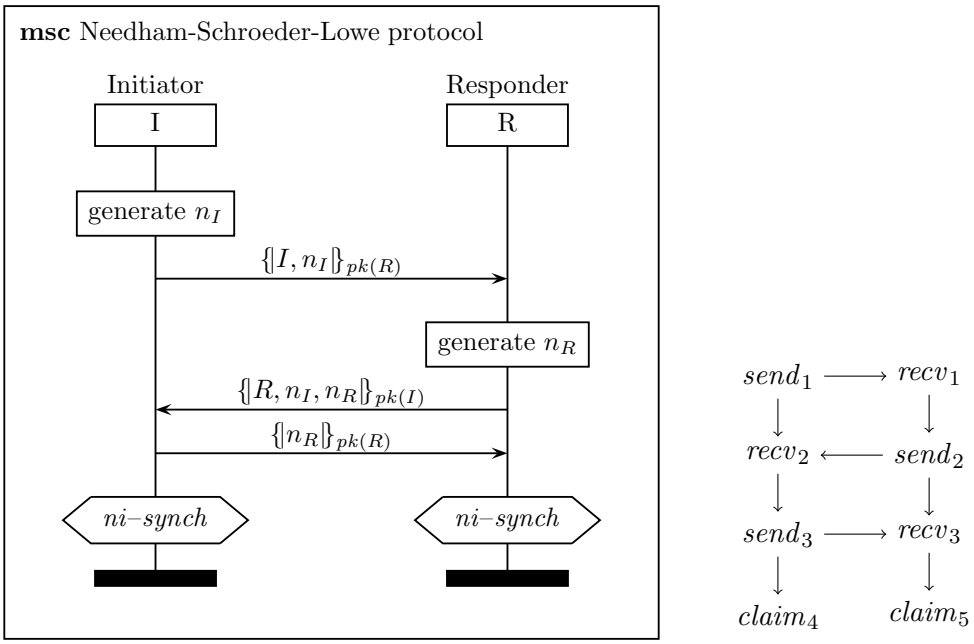
We also define a partial ordering of events in a protocol. First we introduce a *send-before-receive order*, $\prec^{sr}$, expressing that a send event happens before the corresponding receive event, $\mathit{send}_l \prec^{sr} \mathit{recv}_l$. We denote the ordering of events in the event list of the role R for $\prec_R$, and call it *role order*. We now define a partial ordering of events in a protocol as the transitive closure of the union of all role orders and the send-before-receive order, that is

$$\prec_p = \Big(\bigcup_{R \in \mathit{Role}} \prec_R \cup \prec^{sr} \Big)^+.$$

We denote a list of events for a protocol p which follows this ordering for $\mathit{elist}(p)$.

Example 1 (NSL protocol)**.** *As an example we use the Needham-Schroeder-Lowe protocol, denoted by NSL, as depicted with its partial ordering in Figure 2. In a role specification where $I, R \in \mathit{Role}$, $n_I, n_R \in \mathit{Const}$, $v_I, v_R \in \mathit{Var}$, this results in the following list of events for the initiator role:*

$$\begin{aligned} \mathit{NSL}(I) = \; & \mathit{send}_1(I, R, \{\!|I, n_I|\!\}_{pk(R)}) \cdot \mathit{recv}_2(R, I, \{\!|R, n_I, v_I|\!\}_{pk(I)}) \cdot \\ & \mathit{send}_3(I, R, \{\!|v_I|\!\}_{pk(R)}) \cdot \mathit{claim}_4(\mathit{ni\text{–}synch}). \end{aligned}$$



(a) Message sequence chart.

(b) Partial ordering.

Figure 2: The Needham-Schroeder-Lowe protocol.

For the responder, we get the following:

$$NSL(R) = recv_1(I, R, \{\!|I, v_R|\!\}_{pk(R)}) \cdot send_2(R, I, \{\!|R, v_R, n_R|\!\}_{pk(I)}) \cdot recv_3(I, R, \{\!|n_R|\!\}_{pk(R)}) \cdot claim_5(ni\text{–}synch).$$

In this example, we follow an often used practice of including the sender and the recipient as the first two parts of the messages.

3.1.2 Protocol instantiations

We have in the last section defined how to specify a protocol. When parties use this specification to run the protocol, we call it an *execution* of the protocol. The parties executing the protocol are called *agents*, and taken from the set *Agent*. When an agent follows a role specification, it is called a *run*.

A run of a role is in our model also called an instantiation of a role specification. Each run is identified by a run identifier from the set *RunId*. During an instantiation, roles are given to concrete agents, constants are made unique, and variables assigned values. This we denote by $inst = (rid, \rho, \sigma)$ for $rid \in RunId$, $\rho : Role \Rightarrow Agent$ and $\sigma : Var \Rightarrow RunMess$. Here ρ keeps track of the role assignments, and σ stores variable assignments. These functions are local to

each run. We will differ between different role assignment functions by tagging them with a run identifier *rid* and a trace α, that is, as $\rho_{\alpha,rid}$. The same way we will tag the function σ. Traces are explained in Section 3.1.3.

We now introduce two utility functions, taking us from a run identifier to the agent that created the run, and to the role that agent plays in the run. These we denote $agentof : RunId \rightarrow Agent$ and $roleof : RunId \rightarrow Role$. When needed, we tag thee functions with a trace, and write, say $agentof_\alpha$.

A run is an instantiation together with a list of role events, following role order.

After an agent has taken a role, that agent starts to execute that role, following the list of events. This takes us from *RoleEvent* to *RunEvent*, which we define as follows:

$$\begin{aligned} RunEvent \supseteq \{&create(run), send_l(m)\sharp rid, recv_l(m)\sharp rid, claim_l(c)\sharp rid \mid \\ &rid \in RunId, l \in Label, m \in RunMess, c \in Claim\} \end{aligned}$$

We define members of the set *Claim* in Section 3.2.

Any agent can at any time choose to start a run of a role that starts with a send event. That happens with a create event, in which the agent assigns that role to himself, and is given a run identifier. Then the role message in the send event is instantiated, and given to the intruder. Instantiation of role messages is done as follows:

All constants are made unique by appending the run identifier to their name. Role names not previously assigned an agent are given one, by the choice of the agent doing the instantiation. Variables must first be present in a receive event, and are replaced by the values they were assigned then. We formalize this, including the composition of terms. Here is how instantiation works on a role term t:

$$(rid, \rho, \sigma)(t) = \begin{cases} \rho(r) & if \quad t \equiv r \in Role \\ c\#rid & if \quad t \equiv c \in Const \\ f((rid, \rho, \sigma)(t_1), \ldots, (rid, \rho, \sigma)(t_n)) & if \quad t \equiv f(t_1, \ldots, t_n) \\ \sigma(v) & if \quad t \equiv v \in Var \\ ((rid, \rho, \sigma)(t_1), (rid, \rho, \sigma)(t_2)) & if \quad t \equiv (t_1, t_2) \\ \{\!|(rid, \rho, \sigma)(t_1)|\!\}_{(rid,\rho,\sigma)(t_2)} & if \quad t \equiv \{\!|t_1|\!\}_{t_2} \end{cases}$$

From some terms, others can be inferred. For instance, the cipher text together with the decryption key gives knowledge of the encrypted message. This is expressed by the knowledge inference operator, $\vdash$, defined as the smallest relation satisfying the following, where the set of terms M denotes the knowledge of an

agent or the intruder:

$$\begin{aligned} t \in M &\Leftrightarrow M \vdash t \\ M \vdash t_1 \wedge M \vdash t_2 &\Leftrightarrow M \vdash (t_1, t_2) \\ M \vdash t_1 \wedge M \vdash t_2 &\Rightarrow M \vdash \{|t_1|\}_{t_2} \\ M \vdash \{|t_1|\}_{t_2} \wedge M \vdash t_2^{-1} &\Rightarrow M \vdash t_1 \\ M \vdash f \wedge f \notin \{sk, k\} \wedge \bigwedge_{0 \leq i < n} M \vdash t_i &\Rightarrow M \vdash f(t_0, \ldots, t_n). \end{aligned}$$

Here we see that knowledge of an agent name gives knowledge of his public key. Since all agent names are known, all parties in the protocol knows all public keys. This will change in Section 3.2.5, in which we introduce a set of secret agents.

The needed initial knowledge of an agent can be inferred from the event list, as the set of all atomic terms used in the role specification, except for variables. By atomic terms, we will mean terms with no subterms.

After instantiation, the message in a send event is put directly into the knowledge of the intruder, denoted by IK.

Each time an agent encounters a receive event in the role specification, it has to wait for the intruder knowledge to contain a message matching the receive pattern. For details on this matching, we refer to [7]. For our use, it suffices to say that the agent waits for a message for which instantiated values in the patterns equal the ones in the message, and other variables can be assigned the right type, unless otherwise stated. We will for a instantiation of a receive event, $(inst)(recv(pattern))$, denote this as a predicate $Match(inst, pattern, m, inst')$. Here $inst'$ denotes the resulting instantiation, and m is the message fed the agent from the intruder.

If the receive pattern includes role names not previously present in the role specification, any agent name is accepted, and the role assignment function of the receiving run is updated.

The list of events in a non-empty role specification can be separated into the first event and the rest, and we write this as $RoleEvent^* = event_l \cdot s$, where $event_l$ is the first event. As events are instantiated, they are removed from this list, and a new event will be the first, until the list is empty.

Note that that while an agent is executing a role, it is still ready to start any other role starting with a receive event if it sees a matching message. The same agent can also choose to start playing any role that starts with a send event.

When a protocol role is instantiated, the run is added to the set F, which contains the remaining steps of created runs. To get from the set of runs F, to

the set of run identifiers that identifies these runs, we introduce the function $runids(F)$.

We now introduce notation to remove an event directly from the set of remaining runs. When we remove $event_l$ from $run = (inst, event_l \cdot s)$ in the set, we replace run in F with the run $run' = (inst, s)$. This we denote $F[run'/run]$, as a short-hand for redefining F to be $(F \setminus \{run\}) \cup \{run'\}$.

We will partition the agents in two sets, trusted agents in $Agent_T$ and compromised in $Agent_U$. Compromised agents only differ from honest agents in that the intruder shares their knowledge. In Section 3.2.5 we will introduce a third set, but since that set only affects the definition of identity protection, we have omitted it here.

The knowledge of the intruder is denoted by IK, and is given by a set of run terms. The initial intruder knowledge is defined to be IK_0. This is terms the intruder is given before the protocol starts executing.

Definition 1 (Default initial intruder knowledge)**.** Unless otherwise stated, the initial intruder knowledge is given by:

$$IK_0 = \bigcup_{a \in Agent_U, b \in Agent} \{pk(a), sk(a), k(a,b), k(b,a)\} \cup h.$$

Here, h represents the hash function as mentioned in Section 3.1.1.

Now we define the state of the protocol as the instantiation progresses as the tuple:

$$State = (tr, IK, F).$$

The list in tr, the trace so far, represents the previous states of the protocol, and is explained in Section 3.1.3.

The initial state is given by $s_0 = (\langle\rangle, IK_0, \emptyset)$.

This state can be modified only by certain transition rules. In the formal notation, which equals the one used in [7], we use the following pattern:

$$[event\ name]\frac{requirements\ for\ the\ event}{old\ state \rightarrow new\ state}$$

There are four ways in which an agent can change the state of the system for a protocol p.

$$[create]\frac{run = (inst = (rid, \rho, \emptyset), elist(p)) \ \wedge \ rid \notin runids(F)}{(tr, IK, F) \rightarrow (\langle tr \cdot create(run)\rangle, IK, F \cup \{run\})}$$

$$[send]\frac{run = (inst, send_l(m) \cdot s) \in F}{(tr, IK, F) \rightarrow (\langle tr \cdot (inst)send_l(m)\rangle, IK \cup \{inst(m)\}, F[(inst, s)/run])}$$

$$[recv]\frac{run = (inst, recv_l(pattern) \cdot s) \in F \wedge IK \vdash m \wedge Match(inst, pattern, m, inst')}{(tr, IK, F) \rightarrow (\langle tr \cdot (inst) recv_l(pattern) \rangle, IK, F[(inst', s)/run])}$$

$$[claim]\frac{run = (inst, claim_l \cdot s) \in F}{(tr, IK, F) \rightarrow (\langle tr \cdot (inst) claim_l \rangle, IK, F[(inst, s)/run])}$$

3.1.3 Traces

We have now defined the transition rules in which the state is modified. A valid sequence of these transitions is called a trace, denoted by $\alpha = \alpha_0 \cdots \alpha_{n-1}$. The trace α from s_0 to s_n, $s_0 \xrightarrow{\alpha} s_n$, is the result of the transitions $s_0 \xrightarrow{\alpha_0} s_1 \cdots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$. The length of the trace is denoted $|\alpha|$. The set of all possible traces for a protocol p is denoted $Tr(p)$.

In a trace α, $M(\alpha_i)$ denotes the intruder knowledge right before the execution of transition α_i.

Example 2 (The Lowe-attack on Needham-Schroeder)**.** *To illustrate the notation for a trace, we here show the attack Lowe discovered on the short version of the Needham-Schroeder Protocol[16]. This is the same trace as found in [9], but updated to use our notation. Here, the agent e is compromised, and the intruder therefore knows sk(e). The same attack is viewed using a message sequence chart in Figure 3. Note that σ is not written down, but can be constructed from the trace given. The role assignments function ρ is only written down in the create rule, to make it clear who is doing the run under what runtime identifier. The rest can be constructed from the trace.*

$$\begin{aligned}
\alpha_1 &= create(run = (1, \{I \rightarrow a\}, \emptyset)) \\
\alpha_2 &= send_1(a, e, \{|a, n_I \sharp 1|\}_{pk(e)}) \sharp 1 \\
\alpha_3 &= create(run = (2, \{R \rightarrow b\}, \emptyset)) \sharp 2 \\
\alpha_4 &= recv_1(a, b, \{|a, n_I \sharp 1|\}_{pk(b)}) \sharp 2 \\
\alpha_5 &= send_2(b, a, \{|n_I \sharp 1, n_R \sharp 2|\}_{pk(a)}) \sharp 2 \\
\alpha_6 &= recv_2(e, a, \{|n_I \sharp 1, n_R \sharp 2|\}_{pk(a)}) \sharp 1 \\
\alpha_7 &= send_3(a, e, \{|n_R \sharp 2|\}_{pk(e)}) \sharp 1 \\
\alpha_8 &= claim_4(ni\text{–}synch) \sharp 1 \\
\alpha_9 &= recv_3(a, b, \{|n_R \sharp 2|\}_{pk(b)}) \sharp 2 \\
\alpha_{10} &= claim_5(ni\text{–}synch) \sharp 2
\end{aligned}$$

Here the last claim fails, since b thinks he is talking to a, while he was actually talking to e. This will become clear after the content of the claim ni–synch is explained in Definition 14.

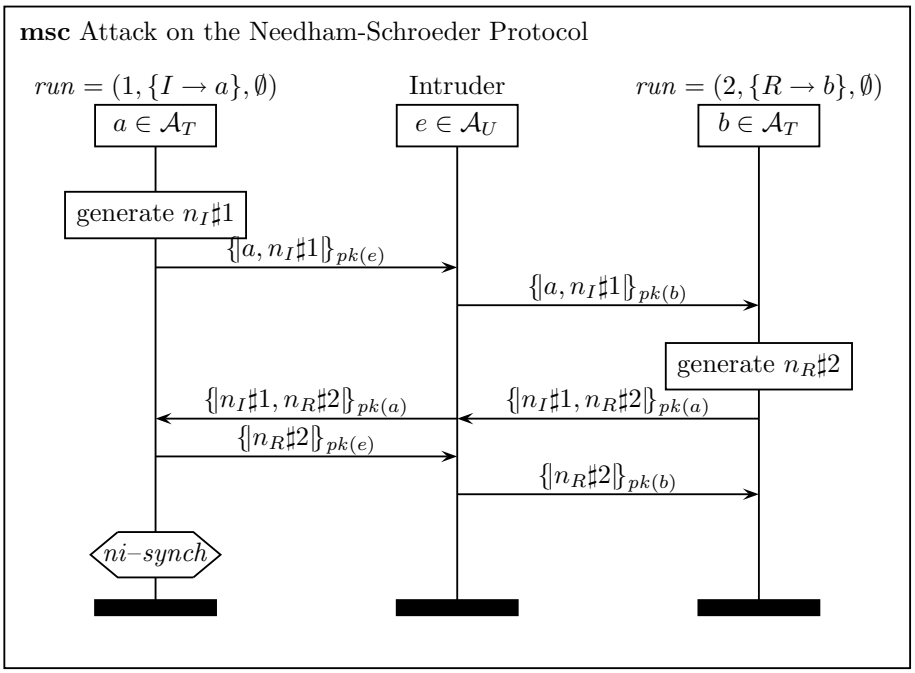


Figure 3: The Lowe-attack on the Needham-Schroeder Protocol, displayed with instantiated terms.

3.2 Security requirements

Protocols have something they want to achieve, and security protocols also have some requirements about what should not be possible. For example, a password login protocol should make it possible for users to log in, while also making it impossible for intruders to get hold of the password used.

In this chapter we develop and define new security requirements. Existing requirements for secrecy and two types of authentication, agreement and synchronization, are defined and modified to allow some of the roles to be given to dishonest agents. Last, a definition of identity protection is proposed in Section 3.2.5.

This section is intended to be a general extension of the model, and not specific to our scenario.

3.2.1 Secrecy

Secrecy is informally the property that the intruder does not know a certain term. We require that all roles in the run in question have been assigned to honest agents. In previous versions of the model, as found in [7, 9], all role

assignments are made as soon as a run is created. That might include roles that does not communicate with the role the creating agent is playing. We have slightly changed how the role assignment function works. In our version, roles are assigned when needed, or learned from received messages. Therefore, ρ might not be defined for all roles of the protocol, and we have updated that part in the definition to account for this.

For ρ, we require that it, prior to each instantiation of a role name, contains that role in its domain. The domain might be expanded right before a send event, or by receiving agent names in a receive event.

To require $\rho_{\alpha,rid}$ to map all assigned roles to honest agents, we will write $\rho_{\alpha,rid} : \mathit{Role} \rightarrow \mathit{Agent}_T$.

Definition 2 (Secrecy)**.** For a role in a protocol p, the predicate SECRET in a claim labeled l is given by:

$$
\begin{aligned}
&\mathit{SECRET}(p,l) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p), k \in N, a \in Agent, rid \in RunId, m \in RunMess} \\
&\qquad \alpha_k = \mathit{claim}_l(\mathit{secret}, m) \sharp \mathit{rid} \wedge \\
&\qquad \rho_{rid,\alpha} : \mathit{Role} \rightarrow \mathit{Agent}_T \Rightarrow \\
&\qquad m \notin M(\alpha_k).
\end{aligned}
$$

We now allow some of the roles to be given to compromised agents in the run, yet we still want the term to be secret.

The group of roles, $R_1, R_2, \ldots$, restricted to honest agents, we denote by a tuple $\mathit{TrustedRoles} = (R_1, R_2, \ldots)$, and we call the new security requirement for *group secrecy*, referring to the knowledge of the term remaining inside a group. We will always require that the claiming role is a part of the group $\mathit{TrustedRoles}$, and, trivially, that $\mathit{TrustedRoles}$ only includes roles of the protocol in question. When the roles are instantiated, we call the instantiated tuple of agents $\mathit{TrustedAgents}$. While a tuple of terms have order, this does not matter in this context, and we will treat the tuple as a set in our mathematical notation. By abuse of notation we will write $\mathit{TrustedAgents} \subseteq \mathit{Agent}_T$ to denote that all the agents in the tuple are honest.

Definition 3 (Group secrecy)**.** For a role in a protocol p, the predicate $\mathit{GROUP{-}SECRET}$ in a claim labeled l is given by:

$$
\begin{aligned}
&\mathit{GROUP{-}SECRET}(p,l) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p), k \in N, rid \in RunId, m \in RunMess} \\
&\qquad \alpha_k = \mathit{claim}_l(\mathit{group{-}secret}, m, \mathit{TrustedAgents}) \sharp \mathit{rid} \wedge \\
&\qquad \mathit{TrustedAgents} \subseteq \mathit{Agent}_T \Rightarrow \\
&\qquad m \notin M(\alpha_k).
\end{aligned}
$$

We then state a theorem for the relation between these two secrecy claims.

Theorem 1 (Group secrecy implies secrecy)**.** *The group secrecy requirement is at least as strong as secrecy. If a protocol p contains $claim_l(group\text{–}secret, m, TrustedRoles)$, then we have the following predicate implication, in which the protocol p' equals p, except we have replaced claim l by $claim_{l'}(secret, m)$:*

$$\forall_{p \in SecProt} : GROUP\text{–}SECRET(p, l) \Rightarrow SECRET(p', l')$$

Proof. Secrecy considers a subset of the traces considered by group secrecy. Therefore, if a term stays secret in the set of traces considered by group secrecy, it will hold in the subset of the traces considered by secrecy. □

3.2.2 Agreement

Agreement denotes that an agent agrees with the others regarding the content of messages sent and received. This is called *non-injective agreement*, which we develop in three steps.

The first step is agreement concerning one single pair of read and send events, identified by a label.

Definition 4 (One-label agreement)**.** For a given trace α, an upper bound $k \in N$ on the position in the trace to search, a label l, and two runs identified by rid_1 and rid_2, the single-label agreement predicate *1L–AGREE* is given by:

$$\begin{aligned} &1L\text{–}AGREE(\alpha, k, l, rid_1, rid_2) \Longleftrightarrow \\ &\quad \exists_{i,j,m \in RunMess} \ \wedge \ i < k, j < k \wedge \\ &\qquad \alpha_i = send_l(m) \sharp rid_1 \ \wedge \ \alpha_j = recv_l(m) \sharp rid_2. \end{aligned}$$

This predicate states that for a label l, the two given runs agree on the content of $send_l$ and $recv_l$.

We now introduce the notion of a cast, a terminology taken from a theater play with multiple performances with different actors playing different roles:

$$cast : RunId \times Role \rightarrow RunId$$

We will not need to know how this function works, as we will only need its existence. Typically, this map will take us from a run making a claim, and a role the agent playing that run communicated with, to the run identifier where an agent played that role.

We then define the next predicate, extending *1L–AGREE* to act upon a set of labels:

Definition 5 (Multi-label agreement)**.** For a given trace α, an upper bound $k \in N$ in the trace, a set of labels L, a run given by *rid* and a cast function *cast*, multi-label agreement is defined with the predicate:

$$\begin{aligned}&ML\text{–}AGREE(\alpha, k, L, rid, cast) \iff \\ &\quad \forall_{l \in L} \\ &\qquad 1L\text{–}AGREE(\alpha, k, l, cast(rid, sendrole(l)), cast(rid, recvrole(l))).\end{aligned}$$

The claim *ML–AGREE* is then extended to take into account the whole protocol. To do that, we need a way to get hold of relevant events to check. For an event $event_m$ in a protocol p, we denote the set of preceding receive event labels as

$$prec_p(m) = \{l \in \mathcal{L} \mid recv_l \prec_p e_l\}.$$

That is, all the receive events that could precede and therefore affect the event with the label given. With this we are ready for the last step.

Definition 6 (Non-injective agreement)**.** For a role in a protocol p, the predicate *NI–AGREE* in a claim labeled l is given by:

$$\begin{aligned}&NI\text{–}AGREE(p, l) \iff \\ &\quad \forall_{\alpha \in Tr(p), k \in N, rid \in RunId} \\ &\qquad \alpha_k = claim_l(ni\text{–}agree) \sharp rid \wedge \\ &\qquad \rho_{rid,\alpha} : Role \to Agent_T \Rightarrow \\ &\qquad \exists_{cast} : cast(rid, claimrole(l)) = rid \wedge \\ &\qquad ML\text{–}AGREE(\alpha, k, prec_p(l), rid, cast).\end{aligned}$$

Informally, this states that for all send and receive events of a role up until the claim, we have in all traces a (not necessarily unique) run were other agents are sending and receiving the same messages.

The notion of agreement requires all roles to be played by honest agents in the run of the claim. We here define *partial agreement*, allowing some roles to be assigned to compromised agents. Informally, we require agreement only for messages that is shared inside the trusted group. This will imply agreement of the terms sent between those agents.

We define a function that, given a label, returns the labels for the previous and relevant send and receive events that are between members of the group encoded in the tuple *TrustedRoles*. For the protocol p, we denote this $prec_{TrustedRoles,p}$, and we define it formally for $l \in Label$ as:

$$\begin{aligned}prec_{TrustedRoles,p}(l) = \{l' \in Label \mid &recvc_{l'} \prec_p event_l \wedge \\ &\rho(sendrole(l')) \in TrustedRoles \wedge \\ &\rho(recvrole(l')) \in TrustedRoles\}.\end{aligned}$$

We now use this instead of $prec_p$ in Definition 6, modify the part about which roles are trusted, and get:

Definition 7 (Partial agreement)**.** For a role in a protocol p, the predicate $PART\text{–}AGREE$ in a claim labeled l is given by:

$$
\begin{aligned}
&PART\text{–}AGREE(p,l) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p), k \in N, rid \in RunId} \\
&\qquad \alpha_k = claim_l(part\text{–}agree, TrustedAgents) \sharp rid \wedge \\
&\qquad TrustedAgents \subseteq Agent_T \Rightarrow \\
&\qquad \exists_{cast} : cast(rid, claimrole(l)) = rid \wedge \\
&\qquad ML\text{–}AGREE(\alpha, k, prec_{TrustedRoles,p}(l), rid, cast).
\end{aligned}
$$

Non-injective agreement guarantees the existence of runs by all communication partners in which the messages agree. The runs do not have to be unique. This enables replay attacks, in which an attacker can reuse old sessions. To disable this, we require the casting function to be injective, and get:

Definition 8 (Injective agreement)**.** For a role in a protocol p, the predicate $I\text{–}AGREE$ in a claim labeled l is given by:

$$
\begin{aligned}
&I\text{–}AGREE(p,l) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p), k \in N, rid \in RunId} \\
&\qquad \alpha_k = claim_l(i\text{–}agree) \sharp rid \wedge \\
&\qquad \rho_{rid,\alpha} : Role \rightarrow Agent_T \Rightarrow \\
&\qquad \exists_{cast \text{ injective}} : cast(rid, claimrole(l)) = rid \wedge \\
&\qquad ML\text{–}AGREE(\alpha, k, prec_p(l), rid, cast).
\end{aligned}
$$

Similarly, we define partial injective agreement:

Definition 9 (Partial injective agreement)**.** For a role in a protocol p, the predicate $PART\text{–}I\text{–}AGREE$ in a claim labeled l is given by:

$$
\begin{aligned}
&PART\text{–}I\text{–}AGREE(p,l) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p), k \in N, rid \in RunId} \\
&\qquad \alpha_k = claim_l(part\text{–}i\text{–}agree, TrustedAgents) \sharp rid \wedge \\
&\qquad TrustedAgents \subseteq Agent_T \Rightarrow \\
&\qquad \exists_{cast \text{ injective}} : cast(rid, claimrole(l)) = rid \wedge \\
&\qquad ML\text{–}AGREE(\alpha, k, prec_{TrustedRoles,p}(l), rid, cast).
\end{aligned}
$$

Agreement as defined here concerns the messages, and not the individual terms. When everyone agrees on all messages, they automatically also agree on the terms the messages contain, as stated in [8, Section 4.1].

This can be more complicated when we allow compromised agents in the claiming run. Some terms may pass through the compromised part, but we still want the honest agents to agree on them in the end. An example could be a signed term that is passed around. It might never be sent between two agents inside the trusted group, yet the intruder might not be able to change it, due to later receive events.

We therefore introduce agreement for individual terms. This is closer to the way Lowe defined agreement in [17, Section 2.3]. This has also been done in [1], under the name *data-agree*, with the requirement that all roles were given to honest agents.

To decide which terms should agree, we must specify a function along with the claim:

$$\gamma : Role \rightarrow RoleTerm.$$

Given a role, γ gives us the term that should agree with the one in the claim. This function is not instantiated. We allow the domain of γ, denoted by $dom(\gamma)$, to be smaller than the set of roles in the protocol.

Definition 10 (Term agreement)**.** For a role in a protocol p, the predicate $PART{-}TERM{-}AGREE$ in a claim labeled l is given by:

$$\begin{aligned}
&TERM{-}AGREE(p,l) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p), k \in N, m \in RunMess, rid \in RunId} \\
&\qquad \alpha_k = claim_{S_6}(part{-}term{-}agree, m, \gamma) \sharp rid \wedge \\
&\qquad \rho_{rid,\alpha} : Role \rightarrow Agent_T \Rightarrow \\
&\qquad \forall_{r \in dom(\gamma)} : \exists_{rid' \in RunId}\ \sigma_{\alpha,rid}(\gamma(r)) = m.
\end{aligned}$$

We then define partial term agreement.

Definition 11 (Partial term agreement)**.** For a role in a protocol p, a trusted group of roles *TrustedRoles*, the predicate $PART{-}TERM{-}AGREE$ in a claim labeled l is given by:

$$\begin{aligned}
&PART{-}TERM{-}AGREE(p,l) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p), k \in N, m \in RunMess, rid \in RunId} \\
&\qquad \alpha_k = claim_l(part{-}term{-}agree, m, TrustedAgents, \gamma) \sharp rid \wedge \\
&\qquad TrustedAgents \subseteq Agent_T \Rightarrow \\
&\qquad \forall_{r \in dom(\gamma)} : \exists_{rid' \in RunId}\ \sigma_{\alpha,rid'}(\gamma(r)) = m.
\end{aligned}$$

We require that valid partial term agreement claims only consider a term that is instantiated by the claiming role at the point of the claim, and that all roles in the trusted group are intended to have the value of this term in the role message given by γ.

3.2.3 Synchronization

Synchronization denotes that what an honest agent thinks has happened, actually did happen. That means that everything that agent sent and received, was received and sent by the honest agents the agent thought he communicated with, in the intended order. The requirement of an ordering is the only change from agreement. We call this claim *non-injective synchronization*, and we develop it in three steps.

The first step is synchronization concerning one single pair of send and receive events, identified by a label.

Definition 12 (One-label synchronization)**.** For a given trace α, an upper bound $k \in N$ on the trace length to search, a label l, and two runs identified by rid_1 and rid_2, the single-label synchronization predicate *1L–SYNCH* is given by:

$$
\begin{aligned}
&\mathit{1L–SYNCH}(\alpha, k, l, rid_1, rid_2) \Longleftrightarrow \\
&\quad \exists_{i,j \in N, m \in \mathit{RunMess}}\ i < j < k \wedge \\
&\qquad \alpha_i = \mathit{send}_l(m) \sharp rid_1 \ \wedge\ \alpha_j = \mathit{recv}_l(m) \sharp rid_2.
\end{aligned}
$$

When we in Definition 4 only required the events to exist, we here also require an ordering, given by $i < j < k$.

Next we proceed the same way as with agreement, using this new predicate. Corresponding to Definition 5, we get:

Definition 13 (Multi-label synchronization)**.** For a given trace α, an upper bound $k \in N$ in the trace, a set of labels L, a run given by *rid* and a cast function *cast*, multi-label synchronization is defined with the predicate:

$$
\begin{aligned}
&\mathit{ML–SYNCH}(\alpha, k, L, rid, cast) \Longleftrightarrow \\
&\quad \forall_{l \in L} \\
&\qquad \mathit{1L–SYNCH}(\alpha, k, l, cast(rid, sendrole(l)), cast(rid, recvrole(l))).
\end{aligned}
$$

Corresponding to Definition 6, we arrive at the definition:

Definition 14 (Non-injective synchronization)**.** For a role in a protocol p, the predicate *NI–SYNCH* in a claim labeled l is given by:

$$
\begin{aligned}
&\mathit{NI–SYNCH}(p, l) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p), k \in N, rid \in \mathit{RunId}} \\
&\qquad \alpha_k = \mathit{claim}_l(\mathit{ni–synch}) \sharp rid \wedge \\
&\qquad \rho_{\alpha, rid} : \mathit{Role} \rightarrow \mathit{Agent}_T \Rightarrow \\
&\qquad \exists_{cast} : cast(rid, claimrole(r)) = rid \wedge \\
&\qquad \mathit{ML–SYNCH}(\alpha, k, prec_p(l), rid, cast).
\end{aligned}
$$

Informally, this states that for all send and receive events of a role preceding the claim, we have a (not necessarily unique) run were other agents are receiving and sending the same messages, and that all receive events are preceded by their corresponding send events.

The notion of non-injective synchronization requires all roles to be played by honest agents in the claiming run. We here define *partial synchronization*, allowing some roles to be compromised by the intruder. Informally, we require synchronization only for messages passed inside the trusted group. If a message is sent to or from an agent outside the group, we will ignore it.

Corresponding to Definition 7, this gives us:

Definition 15 (Partial non-injective synchronization)**.** For a role in a protocol p, the predicate $PART\text{–}NI\text{–}SYNCH$ in a claim labeled l is given by:

$$
\begin{aligned}
&PART\text{–}NI\text{–}SYNCH(p,l) \iff \\
&\quad \forall_{\alpha \in Tr(p), k \in N, rid \in RunId} \\
&\qquad \alpha_k = claim_l(part\text{–}ni\text{–}synch, TrustedAgents) \sharp rid \wedge \\
&\qquad TrustedAgents \subseteq Agent_T \Rightarrow \\
&\qquad \exists_{cast} : cast(rid, claimrole(l)) = rid \wedge \\
&\qquad ML\text{–}SYNCH(\alpha, k, prec_{TrustedRoles,p}(l), rid, cast).
\end{aligned}
$$

To visualize the difference between this and normal synchronization, imagine that we cut away the parts of the protocol that involves untrusted agents. In some protocols, there might be no messages left. In that case, partial synchronization is trivially fulfilled.

Non-injective synchronization guarantees the existence of runs by all communication partners in which the messages agree. The runs do not have to be unique, however. This enables replay attacks, in which an attacker can reuse old sessions. To disable this, we require the casting function to be injective, and get:

Definition 16 (Injective synchronization)**.** For a role in a protocol p, the predicate $I\text{–}SYNCH$ in a claim labeled l is given by:

$$
\begin{aligned}
&I\text{–}SYNCH(p,l) \iff \\
&\quad \forall_{\alpha \in Tr(p), k \in N, rid \in RunId} \\
&\qquad \alpha_k = claim_l(i\text{–}synch) \sharp rid \wedge \\
&\qquad \rho_{rid,\alpha} : Role \rightarrow Agent_T \Rightarrow \\
&\qquad \exists_{cast \text{ injective}} : cast(rid, claimrole(l)) = rid \wedge \\
&\qquad ML\text{–}SYNCH(\alpha, k, prec_p(l), rid, cast).
\end{aligned}
$$

Similar to Definition 15, we also get:

Definition 17 (Partial injective synchronization)**.** For a role in a protocol p, the predicate $PART\text{–}I\text{–}SYNCH$ in a claim labeled l is given by:

$$
\begin{aligned}
&PART\text{–}I\text{–}SYNCH(p,l) \iff \\
&\quad \forall_{\alpha \in Tr(p), k \in N, rid \in RunId} \\
&\qquad \alpha_k = claim_l(part\text{–}i\text{–}synch, TrustedAgents) \sharp rid \wedge \\
&\qquad TrustedAgents \subseteq Agent_T \Rightarrow \\
&\qquad \exists_{cast \text{ injective}} : cast(rid, claimrole(l)) = rid \wedge \\
&\qquad ML\text{–}SYNCH(\alpha, k, prec_{TrustedRoles,p}(l), rid, cast).
\end{aligned}
$$

3.2.4 Hierarchy of the authentication requirements

In [9], Cremers et al. states a theorem giving a hierarchy between the different definitions of agreement and synchronization. We present a corresponding hierarchy between our new definitions for these two types of authentication.

Similarly to the implication between secrecy and group secrecy in Theorem 1, we have the implications shown in Figure 4 for our new definitions. An arrow $c \rightarrow c'$ denotes that if a protocol p contains a claim c that is correct, then the protocol p', defined as p but where c is replaced by c', is also correct.

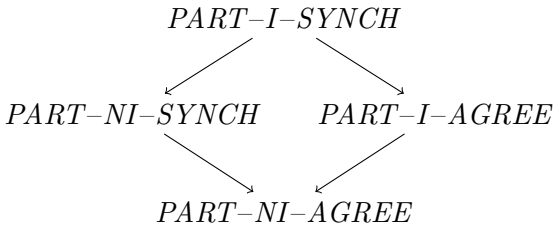


Figure 4: Hierarchy of the new authentication properties.

Theorem 2 (Hierarchy of partial authentication requirements)**.** *The security properties PART–I–SYNCH, PART–NI–SYNCH , PART–I–AGREE, and PART–NI–AGREE satisfy the inclusion relation as depicted in Figure 4.*

Proof. Straightforward from the definitions: Following the arrows, we are only removing requirements about injectivity or order. □

Implications like those above between the old definitions and their partial counterparts do not exist for general protocols. This we show by stating an example in Figure 5 showing that injective synchronization can not always imply partial non-injective agreement, and one in Figure 6 that shows partial injective synchronization can not always imply non-injective agreement.

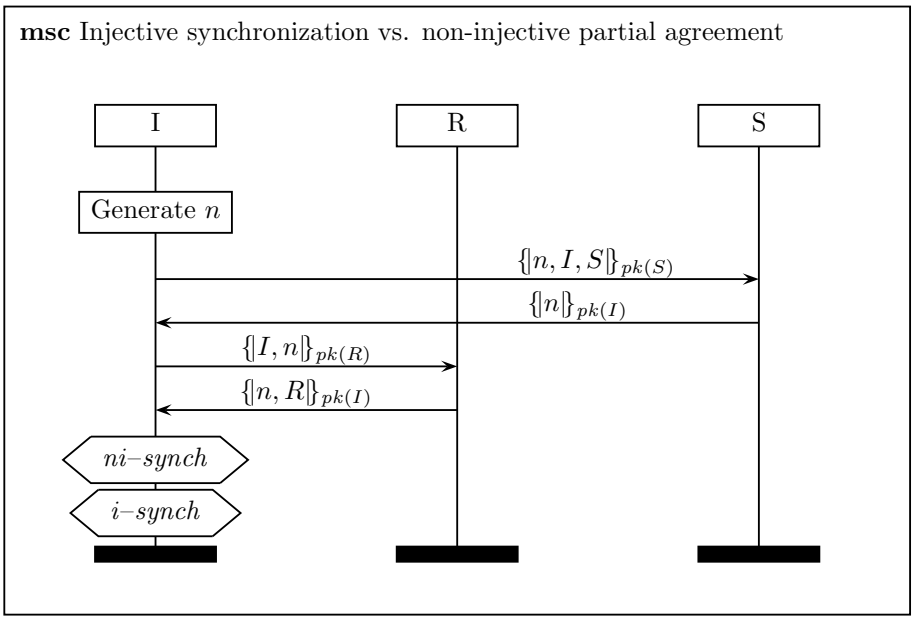


Figure 5: Example showing that injective synchronization does not imply partial non-injective agreement.

The protocol in Figure 5 is translated to a spdl-file in Appendix C, and we have proven in Scyther that the non-injective synchronization claim holds. Using the *loop*-property from [9, Section 4.3], which the claim in question trivially fulfills, we know that injective synchronization holds. Further details can be found in [9].

Partial non-injective agreement is trivially not fulfilled when only I and R make up the trusted group. With a compromised agent assigned the role S in the run with the claim, the intruder learns n and can generate the message sent from R to I,

In Figure 6, the initiator and the responder are, when we neglect the dummy role, executing the Needham-Schroeder protocol, which does give injective synchronization for the initiator [9, End of section 2]. When only they are in the trusted group, the messages to and from the dummy role are disregarded from the partial synchronization claim. Since the intruder gains no new knowledge from these messages, no new attacks emerge against the session between the initiator and the responder. Therefore, partial injective synchronization holds at the end of the initiator role, with the initiator and responder in the trusted group.

Non-injective agreement is trivially not fulfilled at the point where the injective partial synchronization claim is in the figure, as the agent playing the dummy does not have to be involved at all for the initiator to reach this point.

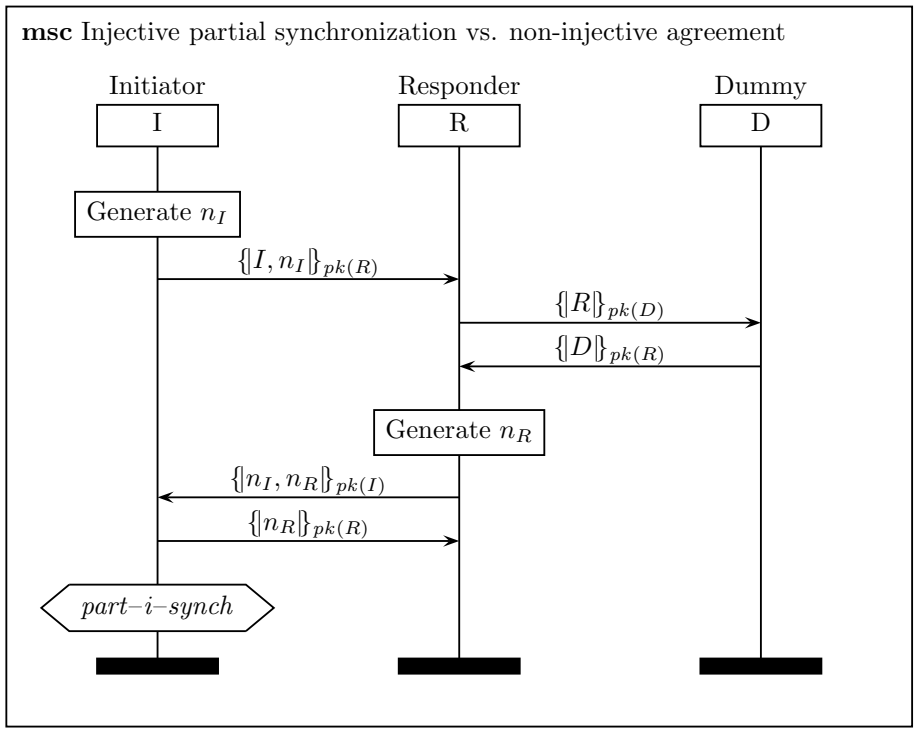


Figure 6: Example showing that partial injective synchronization (here for the initiator with him and the responder in the trusted group) does not imply non-injective agreement at the same point in the specification.

By stating these examples, we have shown that there exist no inclusion relations between the group of partial authentication requirements, and the old ones.

3.2.5 Identity protection

Common Criteria[10], an international standard for computer security certification, divides privacy into four different notions of requirements providing user protection against discovery and misuse of identity by other users. Anonymity is the first of these, and is defined to be the ability for a user to use a system without disclosing its user identity.

The current model does not include any notion of privacy. We will here focus on a notion of anonymity called identity protection.

We start with the definition of anonymity proposed by Pfitzmann and Köhntopp in [21]:

> Anonymity is the state of being not identifiable within a set of subjects, the anonymity set.

Since the agents in our model are only identified with their names, the anonymity set is the set of all honest agents.

Díaz et al.[12] distinguish between *data anonymity* and *connection anonymity*. Informally, data anonymity is about filtering any identifying information out of the data that is exchanged, while connection anonymity is about hiding the identities of the source and destination during the actual data transfer.

We will only look at data anonymity, and assume connection anonymity is present in our model. This means the intruder can not see who puts messages in his knowledge, or who is receiving messages from him.

The only way the intruder can find out that an agent is involved in a protocol, is to read that agent name as part of a message. That way, we are actually talking about *identity protection*.

We want to test if the intruder can learn the identity of an honest agent playing a certain role, yet all agents in $Agent_T$ are known to the intruder before the protocol starts to execute. Therefore we introduce a new set of agents, call them *secret agents* and denote the set by $Agent_S$. These agents work exactly like those in $Agent_T$, except they are not known to the intruder or any compromised agent. Although the secret agents are trusted in the way that they do not share their knowledge with the intruder, they are not members of the trusted set $Agent_T$.

We now require secret agents to remain secret, but we add two restrictions on the traces we consider:

- If an agent from the secret set is assigned a role, it has to be the role for which we claim identity protection.
- All role assignments made by secret agents must, where defined, map all other roles to honest agents.

With these requirements, we claim identity protection by requiring all agent names from $Agent_S$ to be secret. We formalize this in a claim:

Definition 18 (Identity protection)**.** For a role r in a protocol p, the predicate $IDPROT$ is given by:

$$
\begin{aligned}
&IDPROT(p, r) \Longleftrightarrow \\
&\quad \forall_{\alpha \in Tr(p)} \\
&\qquad \Big(\big(\forall_{rid \in RunId, r' \in Role} : \rho_{\alpha,rid}(r') \in Agent_S \Rightarrow r' = r \big) \wedge \\
&\qquad \big(\forall_{rid \in RunId} : agentof(rid) \in Agent_s \Rightarrow \\
&\qquad \forall_{r' \in dom(\rho_{\alpha,rid}) \setminus r} \ \rho_{\alpha,rid}(r') \in Agent_T \big) \Big) \Rightarrow \\
&\qquad Agent_S \cap M(\alpha) = \emptyset.
\end{aligned}
$$

This claim is represented by $claim_l(idprot)$ in a role specification. Note that because agent names are not unique to a run, where this claim is positioned in a role specification does not matter.

We then only require a subset of the roles to be assigned to honest agents in each run a secret agent is active:

Definition 19 (Group identity protection)**.** For a role r in a protocol p and a trusted group of roles $TrustedRoles$, the predicate $GROUP\text{–}IDPROT$ is given by:

$$
\begin{aligned}
&GROUP\text{–}IDPROT(p,r) \iff \\
&\quad \forall_{\alpha \in Tr(p)} \\
&\qquad \Big(\big(\forall_{rid \in RunId, r' \in Role} : \rho_{\alpha,rid}(r') \in Agent_S \Rightarrow r' = r \big) \wedge \\
&\qquad \big(\forall_{rid \in RunId} : agentof(rid) \in Agent_s \Rightarrow \\
&\qquad \forall_{r' \in \{dom(\rho_{\alpha,rid}) \cap TrustedRoles\} \setminus r} \ \rho_{\alpha,rid}(r') \in Agent_T \big) \Big) \Rightarrow \\
&\qquad Agent_S \cap M(\alpha) = \emptyset.
\end{aligned}
$$

This claim is represented by $claim_l(group\text{–}idprot, TrustedRoles)$ in a role specification.

We then present a theorem analogous to Theorem 1. Note that because the position of the claim does not change the outcome of the predicate, we don't have to reference claim labels in this theorem.

Theorem 3 (Group identity protection implies identity protection)**.** *The requirement group identity protection is at least as strong as identity protection. That is, we have the following predicate implication:*

$$\forall_{p \in SecProt, TrustedRoles \subseteq rolesof(p)} : GROUP\text{–}IDPROT(p,r) \Rightarrow IDPROT(p,r)$$

Proof. Identity protection requires secrecy of the secret agents in a subset of the traces for which group identity protection requires the same secrecy. Therefore, the implication holds. □

For examples of how to prove and disprove identity protection, we refer to Appendix A.

3.3 Model motivation

We here discuss how our abstract model maps to actual protocols run by computers over the Internet, and make clear some assumptions needed for applying it. The model is in no way restricted to this setting, but another network would require new justifications and a clarification of the assumptions needed.

We begin with the run terms. Agent names model IP-addresses. We have in the model assumed an infinite number of agents. The actual number of IP-addresses will limit the number of sessions with different agents the intruder can mix, and thus reduce the trace set. For claims of privacy, such as the identity protection requirement we have proposed, this might pose a problem. It could be possible for the intruder to guess an agent, and then try if an encryption with that agent matches one he sees, for instance $\{\!|a|\!\}_{pk(a)}$ for an agent a. We must either require encryptions to be non-deterministic, or the set of available IP-addresses large enough to make guessing infeasible.

Constants and keys correspond to random bit strings. We assume they are sufficiently long to be infeasible for the attacker to guess.

We further assume perfect cryptography, that is:

- It is not possible to learn anything of the encrypted text or the key by looking at the cipher text.
- It is not possible to forge an encryption. That is, only a valid cipher text will be accepted and decrypted by an agent.
- Hash functions are preimage and collision resistant.

The set of honest agents corresponds to users exactly following the protocol. Honest users not following the protocol, say, by mistake, must in our framework be modeled as compromised agents.

Untrusted agents are in our model fully compromised. That is, they could as well be the intruder himself.

The motivation for the set of secret agents is more technical. We want to test if the intruder can learn certain agent names during execution of the protocol, and therefore have to make it secret to him a priori. This could correspond to the fact that although an intruder knows all possible IP-addresses, we still want to measure if it is possible for him to see it in certain sequences of messages he intercepts.

When agents are executing a protocol specification, it corresponds to a thread on a computer executing a program. Putting a message in the intruder knowledge models sending it to the Internet, and receiving it from the intruder corresponds to receiving the message while listening on a port on the computer.

Although it is not necessary in the model, we have in this paper made the assumption that type flaw attacks are impossible. That is, we require a way to encode the terms in bits so that no confusion is possible when decoding them. For how to include type flaws in the model, we refer to the *Match* function discussed in [7].

We assume the intruder can not see who sends messages to, or receives mes-

sages from his knowledge. To achieve connection anonymity like this, we could require an onion router network. In this we must require that not all nodes are compromised, otherwise the anonymity would be impossible [23]. Onion routing could be used only between some of the agents in the protocol when needed. Note that if we require all the protocol messages to be on the form $(sender, receiver, message)$, as in [7] and [9], this assumption is not necessary.

4 Proposed protocol

In this section we propose a protocol for the case study scenario described in Section 2, using the protocol specification presented. For completeness we also include claim events, although they are first discussed in Section 5.2.

In the protocol specification that follows, we have some gaps in the numbering of messages. The meaning of this will become clear as they are filled in Section 5.1.3. To enable identity protection for the store, messages to and from him does not include the sender or the recipient.

Following naming conventions, we have that $U, S, N \in Role$, $k_1, k_2, c, r \in Const$, $V_i \in Var$, and h is a hash function. The term $k(U, N)$ denotes the shared key, the password, between the user and the notary.

$$
\begin{aligned}
U = \; & send_1(\{|k_1, U|\}_{pk(S)}) \\
& send_2(\{|U, c, N|\}_{k_1}) \\
& recv_3(\{|V_1, \{|h(c, U, S, V_1)|\}_{sk(S)}|\}_{k_1}) \\
& send_4(U, N, \{|k_2, U|\}_{pk(N)}) \\
& send_5(U, N, \{|k(U, N)|\}_{k_2}) \\
& send_6(U, N, \{|\{|h(c, U, S, V_1)|\}_{sk(S)}, U|\}_{k_2}) \\
& recv_9(N, U, \{|\{|\{|h(c, U, S, V_1)|\}_{sk(S)}, U|\}_{sk(N)}|\}_{k_2}) \\
& send_{10}(\{|\{|\{|h(c, U, S, V_1)|\}_{sk(S)}, U|\}_{sk(N)}|\}_{k_2}) \\
& claim_{U_1}(group\text{–}secret, k_1, (U, S)) \\
& claim_{U_2}(group\text{–}secret, k_2, (U, N)) \\
& claim_{U_3}(group\text{–}secret, k(U, N), (U, N)) \\
& claim_{U_4}(group\text{–}secret, c, (U, S, N)) \\
& claim_{U_5}(part\text{–}term\text{–}agree, c, (U, S, N), \gamma) \\
& claim_{U_6}(group\text{–}secret, V_1, (U, S, N)) \\
& claim_{U_7}(part\text{–}ni\text{–}synch, (U, N))
\end{aligned}
$$

$$
\begin{aligned}
S = \; & recv_1(\{|V_2, U|\}_{pk(S)}) \\
& recv_2(\{|U, V_3, N|\}_{V_2}) \\
& send_3(\{|r, \{|h(V_3, U, S, r)|\}_{sk(S)}|\}_{V_2}) \\
& recv_{10}(\{|\{|\{|h(V_3, U, S, r)|\}_{sk(S)}, U|\}_{sk(N)}|\}_{V_2}) \\
& claim_{S_1}(group\text{–}secret, sk(S), S) \\
& claim_{S_2}(group\text{–}secret, V_2, (U, S, N)) \\
& claim_{S_3}(group\text{–}secret, V_3, (U, S, N)) \\
& claim_{S_4}(part\text{–}term\text{–}agree, V_3, (U, S, N), \gamma) \\
& claim_{S_5}(group\text{–}secret, r, (U, S, N)) \\
& claim_{S_6}(group\text{–}idprot, (U, S, N))
\end{aligned}
$$

$$
\begin{aligned}
N = \; & recv_4(U, N, \{|V_4, U|\}_{pk(N)}) \\
& recv_5(U, N, \{|k(U, N)|\}_{V_4}) \\
& recv_6(U, N, \{|V_5, U|\}_{V_4}) \\
& send_9(N, U, \{|\{|V_5, U|\}_{sk(N)}|\}_{V_4}) \\
& claim_{N_1}(group\text{–}secret, sk(N), N) \\
& claim_{N_2}(group\text{–}secret, V_4, (U, N)) \\
& claim_{N_3}(group\text{–}secret, k(U, N), (U, N))
\end{aligned}
$$

The protocol is displayed in Figure 7 using a message sequence chart. We have in the specification used an encryption resembling a simplification of TLS, but have omitted it in the figure to make the messages easier to read.

We now explain the protocol, referring to messages in Figure 7. In addition the what we explain comes encryption, and the messages needed to establish the symmetric keys used for that. For this, we refer to the already given specification.

In the first message, the user tells the store the contract he wishes to sign, his name, and which notary he wants to use. This way it is possible for the store to know which notary to contact in case he did not receive a contract at the end of the session.

The store then hashes the received information, together with his name and a salt, and signs it. The salt r is added to the hash to make it impossible for the notary to test different combinations of known documents and stores.

The user receives the signed hash and the salt, and can verify it . He then authenticates with his notary, and sends the signed bundle to him.

The notary can not verify the signature, because he does not know which store signed it. Also the content of the hash is secret to him. All he can do, is to

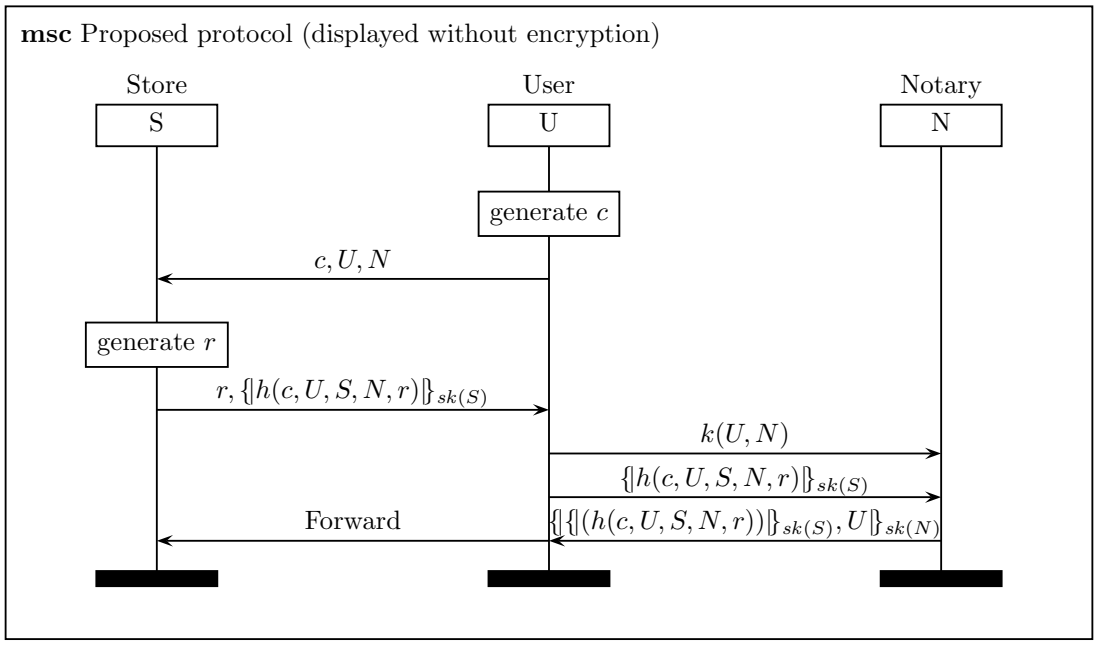


Figure 7: Message sequence chart of the proposed protocol, displayed without encryption. Note that although this way of depicting the protocol does not make it clear, the notary can not read the content of the hash he receives.

sign the bundle together with the name of the user. He includes the user name because he uses his own private key. Instead, he could have one special key for each user, and not have included the user name.

The user receives the bundle, and forwards it to the store. Both keep it for later reference.

The omission of the sender and receiver information in messages between the user and the store models an onion router network, in which at least the connection between the user and the network, and between the store and the network, is not compromised. This could be nodes controlled by an honest ISP.

The security protocol definition language file for this protocol can be found in Appendix D. Note that due to limitations in Scyther, we have there included the sender and recipient in each message. Since Scyther does not support our new security requirements, the file includes no claims.

5 Analysis

Here we apply the model and our new definitions to our scenario. First we formalize the threat model, and then the security requirements. Last, we verify

the proposed protocol by proving the claims in the requirements.

5.1 Formal threat model

We here formalize the threat model from Section 2.2 in the model put forth.

5.1.1 Model 1: Dolev-Yao attack model

The Dolev-Yao attack model is modeled by requiring all roles in the claiming run to be played by honest agents. In this model we can use the old security requirements.

5.1.2 Model 2: Compromised store and user

In this model we drop the requirement that the user and the store should be played by honest agents in the run with the claim. Depending on the requirement we test, we will allow one of them, or both, to be played by the intruder. The agent claiming a security requirement has to be honest.

Note that the status of an agent as honest or compromised does not change during the execution of the protocol. An attack model in which an agent is compromised from the beginning is always stronger than one in which the agent is compromised at some later point. Therefore, security in the first will imply security in the latter.

5.1.3 Model 3: Partly compromised notary

Our framework only allows an agent to be honest or under full control by the intruder. While this is changed in [2], we can in our setup use an alternative way to model a partly compromised notary.

Our solution is to add a fully compromised agent to the protocol, which receives leaked information from the notary. This way we can control what is leaked, and when, and look at what the intruder is able to do with this additional information.

In the formal notation we have presented, this results in the following role being added to the protocol listed in Section 4:

$$\begin{aligned} D = \; & recv_7(N, D, (V_6, U)) \\ & recv_8(N, D, \{\!|V_6, U|\!\}_{sk(N)}). \end{aligned}$$

The corresponding events are added to role of the notary, filling in the gap in the message numbering:

$send_7(N, D, (V_5, U))$
$send_8(N, D, \{|V_5, U|\}_{sk(N)})$.

5.2 Formal protocol requirements

We now try to identify the requirements our proposed signing protocol must fulfill, based on the informal discussion in Section 2.3.

All the listed claims can be found in the protocol specification in Section 4.

5.2.1 Secrecy of keys

The private keys used for signing must remain secret, even with compromised agents playing the other parts of the protocol.

Requirement 1 (Secrecy of the private keys)**.** *We include the following claims in the modified protocol specification described in Section 5.1.3, in the roles of the store and the notary, respectively:*

$claim_{S_1}(group\text{–}secret, sk(S), S)$
$claim_{N_1}(group\text{–}secret, sk(N), N)$

Further, the session keys the user generates should be secret, given an honest agent as the intended recipient.

Requirement 2 (Secrecy of session keys)**.** *We include the following claims in the protocol specification, in the role of the user:*

$claim_{U_1}(group\text{–}secret, k_1, (U, S))$
$claim_{U_2}(group\text{–}secret, k_2, (U, N))$

This requirement ensures that the messages the user sends using these keys, can not be read by any other than the intended recipient.

In the same way, the store and the notary want to be sure that the keys they receive are secret. This guarantees that the messages received are not generated by the intruder, and results in the following requirement.

Requirement 3 (Secrecy of received session keys)**.** *We include the following claims in the modified protocol specification described in Section 5.1.3, in the roles of the store and the notary, respectively:*

$claim_{S_2}(group\text{–}secret, V_2, (U, S, N))$
$claim_{N_2}(group\text{–}secret, V_4, (U, N))$

5.2.2 Secrecy of the password

The password is in our protocol modeled as a key shared between the user and the notary. We want them both to be sure that the password remains secret, even given a compromised store, and add the following requirement:

Requirement 4 (Secrecy of the password)**.** *For the user and the notary, we include the following in their respective protocol specifications:*

$$claim_{U_3}(group\text{–}secret, k(U,N), (U,N))$$
$$claim_{N_3}(group\text{–}secret, k(U,N), (U,N))$$

5.2.3 Requirements for the contract

As discussed, we want the contract to be secret against a partly compromised notary. For both the user and the store to be sure of this, we make the following requirement.

Requirement 5 (Secrecy of the contract)**.** *We include the following claims in the modified protocol specification described in Section 5.1.3, in the specification of the store and the user, respectively:*

$$claim_{S_3}(group\text{–}secret, V_3, (U,S,N))$$
$$claim_{U_4}(group\text{–}secret, c, (U,S,N))$$

In addition to being secret, the user and the store should agree on the content of the contract. That is, when one of them finishes his run, there should exist a run by the agent he thinks he communicated with, having the same value for the contract. This is partial term agreement.

Requirement 6 (Agreement on the contract)**.** *We include the following claims in the modified protocol specification described in Section 5.1.3, in the specification of the store and the user, respectively:*

$$claim_{S_4}(part\text{–}term\text{–}agree, V_3, (U,S,N), \gamma = (U \to c))$$
$$claim_{U_5}(part\text{–}term\text{–}agree, c, (U,S,N), \gamma = (S \to V_3))$$

In addition to being secret, the store and the user wants it to be impossible for the notary to test guesses of which contract is being signed. In the model, the contract is a nonce, and by definition impossible to guess. In the real world, we might imagine some contracts being often used, and we have therefore used salt in the hash to disable guessing. This requires the salt to be secret.

Requirement 7 (Secrecy of the salt)**.** *We include the following claims in the modified protocol specification described in Section 5.1.3, in the specification of the store and the user, respectively:*

$$claim_{S_5}(group\text{–}secret, r, (U, S, N))$$
$$claim_{U_6}(group\text{–}secret, V_1, (U, S, N))$$

5.2.4 Identity protection

The user and the store want the identity of the store to remain secret, even with a partly compromised notary.

Requirement 8 (Identity protection for the store)**.** *We include the following claim in the modified protocol specification described in Section 5.1.3, in the role of the store:*

$$claim_{S_6}(group\text{–}idprot, (U, S, N))$$

5.2.5 Authentication

The authentication of the user and the signing of the contract must happen in the same session, with the same understanding of the variables involved. This is ensured by partial non-injective synchronization between the user and the notary.

Requirement 9 (Partial synchronization between the user and the notary)**.** *We include the following claim in the specification of the user:*

$$claim_{U_7}(part\text{–}ni\text{–}synch, (U, N))$$

5.3 Results

We here verify the requirements put forth. The proof technique is in general to assume the requirement is broken, and arrive at a contradiction when investigating how this must have happened.

5.3.1 Secrecy of private keys

We first look at the private long term keys for the store and the notary, $sk(S)$ and $sk(N)$. None of them are inferable from any message in the protocol, and they remain therefore secret for all honest agents. This proves Requirement 1.

From this we know that all terms signed with one of these keys, must originate from the owner of the key, and that all terms encrypted with one of the corresponding public keys, $pk(S)$ and $pk(N)$, are only readable by their respective owners.

5.3.2 Secrecy of the session keys

We look at the session keys k_1 and k_2 generated by the user, starting with k_1.

The key k_1 is used between the user and the store. The only message from which this key is inferable, is $send_1(\{\!|k_1, U|\!\}_{pk(S)})$. This message is encrypted with the public key of the store. For an honest agent playing the role S, as required by the claim, the inverse is already shown to be group secret. Therefore the session key is also group secret.

The same argument as above holds for the session key k_2. This proves Requirement 2.

For later use, we make the following observation:

Lemma 1. *The session keys are group secret right after they are sent.*

In other words, the user does not need to reach the claim to be sure these keys are secret.

Proof. This follows since the role assignment of the recipient is done in the send event with the keys. Therefore we know at that point if the recipient is honest, and since the decryption key is group secret, we know only the recipient can read the key. □

5.3.3 Secrecy of the password

We assume there exists a run breaking the user's group secrecy claim for the password. That is:

$$\begin{aligned}&\exists_{\alpha \in Tr(p), k \in N, rid \in RunId, u,n \in Agent}\\ &\quad \rho_{\alpha, rid=\{U \to u, N \to n\}} \wedge\\ &\quad \alpha_k = claim_{U_3}(group\text{–}secret, k(u,n), (u,n))\sharp rid \wedge\\ &\quad u, n \in Agent_T \ \wedge \ k(u,n) \in M(\alpha_k).\end{aligned}$$

The password is only inferable from instantiations of the message $send_5$, sent from the user to the store. Therefore, there must exist a run with agent u playing the user, in which he sends this message to agent n assigned the notary role. That is, the following must hold for the trace:

$$\exists_{i<k} : \alpha_i = send_5(u, n, \{\!|k(u,n)|\!\}_{k_2 \sharp rid'})\sharp rid'$$

Note that because the password is not unique to a run, this could be a different run than the one with the claim.

For the intruder to be able to read the password from this message, he must know the encryption key $k_2 \sharp rid'$. This key is unique in each run, and the intruder must have learned it from the previous message sent by the user in that run. That is, for $h < i$:

$$\alpha_h = \ send_4(u, n, \{\!|k_2 \sharp rid', u|\!\}_{pk(n)}) \sharp rid'.$$

To infer the key from this message, the intruder must know the private key of the honest agent n. This we have in Section 5.3.1 shown is not possible, and hence is the password group secret.

We then look at the group secrecy of the password in the claim N_3 of the notary. For the term $k(U, N)$ to be known by the intruder, he must either have generated it himself, or learned it from an honest agent playing the user. The first is not possible as the notary already knows the password, and verifies it in $recv_5$. The second option is also not possible, because the encryption key the user used when sending the password is shown to be group secret in Section 5.3.2.

This proves Requirement 4.

5.3.4 Secrecy of received session keys

We first look at the session key received by the notary. We want to prove that, given an honest user, the value assigned to variable V_4 is secret when the notary is reaching the secrecy claim N_2. Here we allow the store to be compromised in the same run.

We assume the claim does not hold. That is:

$$\begin{aligned}
&\exists_{\alpha \in Tr(p), k \in N, rid \in RunId} \ \wedge \\
&\quad \rho_{\alpha,rid} = \{U \rightarrow u, N \rightarrow n\} \ \wedge \\
&\quad \alpha_k = claim_{N_2}(group\text{–}secret, \sigma_{\alpha,rid}(V_4), (u, n)) \sharp rid \ \wedge \\
&\quad u, n \in Agent_T \ \wedge \ \sigma_{\alpha,rid}(V_4) \in M(\alpha_k).
\end{aligned}$$

The key in V_4 is only inferable from $recv_4$. For the intruder to learn the key from this message, we must require the following for the trace, for $h < k$:

$$\alpha_h = \ recv_4(u, n, \{\!|\sigma_{\alpha,rid}(V_4), u|\!\}_{pk(n)}) \sharp rid.$$

Since this message is encrypted with the public key of an honest agent, the intruder can not decrypt and learn it. He could, however, have constructed it himself.

If the received key was constructed by the intruder, there exists no run in which the user is using that key. This is because nonces are by construction unique, and the session keys sent by the user are group secret right after sending, as stated in Lemma 1. Therefore, the intruder must have forged all messages the notary has received in this run. That is, for $h < i < j < k$:

$$\begin{aligned}\alpha_h &= recv_4(u, n, \{|\sigma_{\alpha,rid}(V_4), u|\}_{pk(n)})\sharp rid\\ \alpha_i &= recv_5(u, n, \{|k(u, n)|\}_{\sigma_{\alpha,rid}(V_4)})\sharp rid\\ \alpha_j &= recv_6(u, n, \{|\sigma_{\alpha,rid}(V_5), u|\}_{\sigma_{\alpha,rid}(V_4)})\sharp rid\end{aligned}$$

We here see that to forge the message received in α_i, the intruder must know the password $k(u, n)$. This is not possible, since for two honest agents we have already shown that the password is secret in Section 5.3.3. Therefore, upon receiving the password, the notary knows that the key stored in V_4 is group secret, even with a compromised store in the same session. In other words, the password authenticates the user to the notary, which is our intuitive understanding of what a password should do.

We then look at the session key received by the store. We want to prove that, given an honest user, the nonce assigned to variable V_2 is secret when the store is reaching the secrecy claim S_2. Here we allow a partly compromised notary, and use the modified protocol p', as specified in Section 5.1.3.

We assume the claim does not hold:

$$\begin{aligned}&\exists_{\alpha \in Tr(p'), k \in N, rid \in RunId} \wedge\\ &\quad \rho_{\alpha,rid} = \{U \to u, S \to s, N \to n\} \wedge\\ &\quad \alpha_k = claim_{S_2}(group\text{–}secret, \sigma_{\alpha,rid}(V_2), (u, s, n))\sharp rid \wedge\\ &\quad u, s, n \in Agent_T \ \wedge \ \sigma_{\alpha,rid}(V_2) \in M(\alpha_k).\end{aligned}$$

The key in V_2 is only inferable from $recv_1$. For the intruder to learn the key from this message, we must require the following for the trace, for $h < k$:

$$\alpha_h = \ recv_1(\{|\sigma_{\alpha,rid}(V_2), u|\}_{pk(s)})\sharp rid.$$

Since this message is encrypted with the public key of an honest agent, the intruder can not decrypt it. He could, however, have constructed it himself.

If the received key was constructed by the intruder, there exists no run in which the user is using that key. By the same argument as above, the intruder must then have forged all messages the store has received in this run. That is, for $h < i < j < k$:

$$\begin{aligned}\alpha_h &= recv_1(\{|\sigma_{\alpha,rid}(V_2), u|\}_{pk(b)})\sharp rid\\ \alpha_i &= recv_2(\{|u, \sigma_{\alpha,rid}(V_3), n|\}_{\sigma_{\alpha,rid}(V_2)})\sharp rid\\ \alpha_j &= recv_{10}(\{|\{|\{|h(\sigma_{\alpha,rid}(V_3), u, s, r\sharp rid)|\}_{sk(s)}, u|\}_{sk(n)}|\}_{\sigma_{\alpha,rid}(V_2)})\sharp rid.\end{aligned}$$

To construct the message in α_j, the intruder has to know the signed term he encrypted. Since the signing key of the notary is secret, we know that there must exist a run rid' of the agent n playing the notary in which he signs and sends the following message to a dummy agent d, in which $\sigma_{\alpha,rid'}(V_5) = \{\!|h(\sigma_{\alpha,rid}(V_3), u, s, r\sharp rid)|\!\}_{sk(s)}$. That is, for $j' < j$:

$$\alpha_{j'} = \ send_8(n, d, \{\!|\sigma_{\alpha,rid'}(V_5), u|\!\}_{sk(n)})\sharp rid'.$$

For the notary to arrive at this event, the following must have happened in the trace, for $f' < g' < h' < i' < j'$:

$$\begin{aligned}
\alpha_{f'} &= recv_4(u, n, \{\!|\sigma_{\alpha,rid'}(V_4), u|\!\}_{pk(n)})\sharp rid' \\
\alpha_{g'} &= recv_5(u, n, \{\!|k(u, n)|\!\}_{\sigma_{\alpha,rid'}(V_4)})\sharp rid' \\
\alpha_{h'} &= recv_6(u, n, \{\!|\sigma_{\alpha,rid'}(V_5), u|\!\}_{\sigma_{\alpha,rid'}(V_4)})\sharp rid' \\
\alpha_{i'} &= send_7(n, d, (\sigma_{\alpha,rid'}(V_5), u))\sharp rid'
\end{aligned}$$

Since the password $k(u, n)$ is shared between two honest users, we have shown it to be group secret in Section 5.3.3. Therefore, the intruder could not have made the encryption in $\alpha_{g'}$, and there must exist a run rid'' of the agent u in which $\sigma_{\alpha,rid'}(V_4) = k_2\sharp rid''$. The key k_2 we have already shown to be secret in Section 5.3.2. Therefore all messages encrypted with this key must stem from the run rid''. That is, for $c'' < d'' < e'' < f'' < g'' < h'' < h'$.

$$\begin{aligned}
\alpha_{c''} &= send_1(\{\!|k_1\sharp rid'', u|\!\}_{pk(s)})\sharp rid'' \\
\alpha_{d''} &= send_2(\{\!|u, \sigma_{\alpha,rid}(V_3), n|\!\}_{k_1\sharp rid''})\sharp rid'' \\
\alpha_{e''} &= recv_3(\{\!|V_1, \{\!|h(\sigma_{\alpha,rid}(V_3), u, s, r\sharp rid)|\!\}_{sk(s)}|\!\}_{k_1})\sharp rid'' \\
\alpha_{f''} &= send_4(u, n, \{\!|\sigma_{\alpha,rid'}(V_4), u|\!\}_{pk(n)})\sharp rid'' \\
\alpha_{g''} &= send_5(u, n, \{\!|k(u, n)|\!\}_{\sigma_{\alpha,rid'}(V_4)})\sharp rid'' \\
\alpha_{h''} &= send_6(u, n, \{\!|\{\!|h(\sigma_{\alpha,rid}(V_3), u, s, r\sharp rid)|\!\}_{sk(s)}, u|\!\}_{\sigma_{\alpha,rid'}(V_4)})\sharp rid''
\end{aligned}$$

We here see that $\alpha_{d''}$ and α_i contains the same value for the contract. That means, for the intruder to forge the message in α_i, he must know the value of the contract in this run by the user. This value is only inferable from the message sent in $\alpha_{d''}$, which is encrypted by $k_1\sharp rid''$. Since this key is shown to be group secret for an honest store, we have a contradiction.

We have now proved Requirement 3.

5.3.5 Secrecy of the contract and the salt

The contract is in our protocol the nonce c generated by the user, and stored in the variable V_3 by the store. It is only from $send_2$ that c can be inferred.

This message is, however, encrypted with the key k_1, already shown to be group secret in the modified protocol, in Section 5.3.2. We then know that the user, when assigning the store role to an honest agent, can be sure the contract stays group secret.

When it comes to the store, the same argument applies. He will upon finishing his protocol be sure that the key by which the messages he received was encrypted, is group secret. Therefore the agent could not have generated nor learned the contract. By this we have proven Requirement 5.

The salt is sent from the store as the nonce r, and received by the user in the variable V_1. The same argument as above holds: Because both the key used for encryption when sending and the one used for decryption when receiving are group secret, so is the salt inside the messages. This proves Requirement 7.

5.3.6 Partial term agreement for the contract

We here want to show that if the agent playing the user reaches the partial term agreement claim U_5, there exists a run in which the agent he assigned as the store, has the value of the instantiated nonce c, the contract, in his variable V_3.

This is trivially fulfilled in the modified protocol, since the agent playing the user at the point of the claim has received and verified the hash containing the contract, which is signed by the store agent, using the secret key $sk(S)$. As we noted in Section 5.3.1, this signature must stem from a run by the store.

We then want to show that when the store reaches claim S_4, there exists a run of the agent he assigned the user role, for which the instantiated nonce c equals the value in his variable V_3. We assume this is not the case for the modified protocol p':

$$
\begin{aligned}
&\exists_{\alpha \in Tr(p'), k \in N, rid \in RunId} \\
&\quad \rho_{\alpha,rid} = \{U \to u, S \to s, N \to n\} \wedge \\
&\quad \alpha_k = claim_{S_4}(part\text{–}term\text{–}agree, \sigma_{\alpha,rid}(V_3), (u,s,n), \gamma) \sharp rid \wedge \\
&\quad u, s, n \in Agent_T \wedge \\
&\quad \nexists_{rid' \in RunId} : \big(c \sharp rid' = \sigma_{\alpha,rid}(V_3) \ \wedge \ agentof(rid') = u\big).
\end{aligned}
$$

For agent s to reach this claim, we must have the following event in the trace, $j < k$:

$$\alpha_j = recv_{10}(\{\!|\{\!|\{\!|h(\sigma_{\alpha,rid}(V_3), u, s, r \sharp rid)|\!\}_{sk(s)}, u|\!\}_{sk(n)}|\!\}_{\sigma_{\alpha,rid}(V_2)}) \sharp rid$$

Since we have already shown that the key $\sigma_{\alpha,rid}(V_2)$ is secret when $u, s \in Agent_T$, this message must have been sent from agent u. That requires the following event

in the trace, $i < j$:

$$\alpha_i = send_{10}(\{|\{|\{|h(\sigma_{\alpha,rid}(V_3), u, s, r\sharp rid)|\}_{sk(s)}, u|\}_{sk(n)}|\}_{\sigma_{\alpha,rid}(V_2)})\sharp rid'$$

From this message we see that $c\sharp rid' = \sigma_{\alpha,rid}(V_3) \ \wedge \ agentof(rid') = u$, and we have a contradiction to out assumption.

This proves Requirement 6.

5.3.7 Synchronization

The user requires partial non-injective synchronization between him and the notary. We assume the claim fails, that is:

$$\begin{aligned}
&\exists_{\alpha \in Tr(p), k \in N, rid \in RunId} \\
&\quad \rho_{\alpha,rid} = \{U \to u, S \to s, N \to n\} \wedge \\
&\quad \alpha_k = claim_{U_7}(part\text{–}ni\text{–}synch, (u,n))\sharp rid \ \wedge \ u, n \in Agent_T \wedge \\
&\quad \nexists_{cast} : \big(cast(rid, u) = rid \ \wedge \ ML\text{–}SYNCH(\alpha, k, prec_{(u,n),p}(U_7), rid, cast)\big).
\end{aligned}$$

At the point of the claim, the user must have sent and received the following messages relevant to the claim. Here $g < h < i < j < k$:

$$\begin{aligned}
\alpha_g &= send_4(u, n, \{|k_2\sharp rid, u|\}_{pk(n)})\sharp rid \\
\alpha_h &= send_5(u, n, \{|k(u,n)|\}_{k_2\sharp rid})\sharp rid \\
\alpha_i &= send_6(u, n, \{|\{|h(c\sharp rid, u, s, \sigma_{\alpha,rid}(V_1))sk(c), a|\}_{k_2\sharp rid}|\})\sharp rid \\
\alpha_j &= recv_9(n, u, \{|\{|\{|h(c\sharp rid, a, s, \sigma_{\alpha,rid}(V_1))|\}_{sk(s)}, u|\}_{sk(n)}|\}_{k_2\sharp rid})\sharp rid
\end{aligned}$$

We then look at α_j. This message is encrypted with a key we have already shown is group secret. That is, only known to the user agent who made it, and the intended recipient. Therefore it must exist a run by the agent n, in which the following event happens, for $j' < j$:

$$\alpha_{j'} = send_9(n, u, \{|\{|\sigma_{\alpha,rid'}(V_5), u|\}_{sk(n)}|\}_{k_2\sharp rid})\sharp rid'$$

In this send event of the notary, all variables and role assignments are present, giving only one possible path that agent could have taken to this point. That path is, for $g' < h' < i' < j' < j$:

$$\begin{aligned}
\alpha_{g'} &= recv_4(u, n, \{|k_2\sharp rid, u|\}_{pk(n)})\sharp rid' \\
\alpha_{h'} &= recv_5(u, n, \{|k(u,n)|\}_{k_2\sharp rid})\sharp rid' \\
\alpha_{i'} &= recv_7(u, n, \{|\sigma_{\alpha,rid'}(V_5), u|\}_{k_2\sharp rid})\sharp rid' \\
\alpha_{j'} &= send_9(n, u, \{|\{|\sigma_{\alpha,rid'}(V_5), u|\}_{sk(n)}|\}_{k_2\sharp rid})\sharp rid'
\end{aligned}$$

We here manually inspect the messages, and finds that they all agree with those sent and received by agent u playing the user.

We have now proved that partial non-injective agreement is satisfied. To prove synchronization, we must also show that all the send events in rid happened before the corresponding receive events in rid'. We have already shown that $j' < j$.

Since constants are by definition unique, rid is the only run by the user with $k_2 \sharp rid$ as the key. Therefore the intruder can not get the messages encrypted with this key from different runs, and he has to wait for agent u to send the cipher texts before he can give them to agent n. Therefore we have that $g < g'$, $h < h'$ and $i < i'$.

This proves Requirement 9.

5.3.8 Identity protection for the store

Proving this is trivial: From none of the messages sent in the modified protocol can the agent name of the store be inferred. Therefore, it can not be learned by the intruder. This proves Requirement 8.

For a non-trivial example of how to prove and disprove identity protection, we refer to Appendix A.

6 Conclusion

We have in this paper expanded the model by Cremers et al. in two directions. In the direction of allowing a more powerful intruder, we have added new security requirements that allow the intruder to play a part in the protocol session in which we want the property to hold. Secondly, towards handling privacy, we have added a definition of identity protection. This requires some new assumptions:

For our notion of identity protection, we have assumed connection anonymity. We have further assumed the existence of a set of agents not known to the intruder. This gives us a way to test if the protocol leaks the identities in question.

As future work, we suggest looking into how other types of privacy could be modeled. Further, we also suggest testing the definition of identity protection against existing protocols promising different kinds of privacy.

References

[1] S. Andova, C.J.F. Cremers, K. Gjøsteen, S. Mauw, S.F. Mjølsnes, and S. Radomirović. A framework for compositional verification of security protocols. *Information and Computation*, 206:425–459, February 2008. http://arxiv.org/pdf/cs.CR/0611062.

[2] David Basin and C.J.F. Cremers. From Dolev-Yao to Strong Adaptive Corruption: Analyzing Security in the Presence of Compromising Adversaries. Cryptology ePrint Archive, Report 2009/079, 2009. http://eprint.iacr.org/2009/079.pdf.

[3] Michael Ben-Or, Oded Goldreich, Silvio Micali, and Ronald L. Rivest. A fair protocol for signing contracts. In *Automata, Languages and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 43–52. Springer Berlin / Heidelberg, 1985. http://www.springerlink.com/content/w42170257rw54474/.

[4] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptography*, 1(1):65–75, 1988. http://www.cs.cornell.edu/People/egs/herbivore/dcnets.html.

[5] C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006. http://people.inf.ethz.ch/cremersc/scyther/download-thesis.php.

[6] C.J.F. Cremers. Scyther 1.0 User Manual, April 2007. http://people.inf.ethz.ch/cremersc/scyther/index.html.

[7] C.J.F. Cremers and S. Mauw. Operational Semantics of Security Protocols. In S. Leue and T. Systä, editors, *Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers*, volume 3466 of *Lecture Notes in Computer Science*. Springer, 2005. http://www.win.tue.nl/~ecss/downloads/CrMa04b.pdf.

[8] C.J.F. Cremers, S. Mauw, and E.P. de Vink. Defining authentication in a trace model. In T. Dimitrakos and F. Martinelli, editors, *FAST 2003*, Proc. of the first international Workshop on Formal Aspects in Security and Trust, pages 131–145, Pisa, September 2003. IITT-CNR technical report. http://www.win.tue.nl/~ecss/downloads/cmv_defining_authentication.ps.

[9] C.J.F. Cremers, S. Mauw, and E.P. de Vink. Injective synchronisation: an extension of the authentication hierarchy. *Theoretical Computer Science*, pages 139–161, 2006. http://www.liacs.nl/~devink/research/Postscript/fast2003.pdf.

[10] Common Criteria. Common Criteria for Information Technology Security Evaluation v3.1 r2 - Part 2: Security functional components, September 2007. http://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R2.pdf.

[11] Claudia Dıaz, Joris Claessens, Stefaan Seys, and Bart Preneel. Information theory and anonymity. In *Proceedings of the 23rd Symposium on Information Theory in the Benelux*, pages 179–186, 2002. https://www.cosic.esat.kuleuven.be/publications/article-90.pdf.

[12] Claudia Díaz, Stefaan Seys, Joris Claessens, and Bart Preneel. *Towards Measuring Anonymity*, volume 2482/2003 of *Lecture Notes in Computer Science*, pages 184–188. Springer, 2003. http://www.cosic.esat.kuleuven.be/publications/article-89.pdf.

[13] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, Mar 1983. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01056650.

[14] Kristian Gjøsteen. *Weaknesses in BankID, a PKI-Substitute Deployed by Norwegian Banks*, volume 5057/2008 of *Lecture Notes in Computer Science*, pages 196–206. Springer, June 2008. www.math.ntnu.no/~kristiag/pki/europki.pdf.

[15] Detlef Kähler, Ralf Küsters, and Thomas Wilke. Deciding properties of contract-signing protocols. In *STACS 2005*, volume 3404 of *Lecture Notes in Computer Science*, pages 158–169. Springer, 2005. http://www.springerlink.com/content/m73q0pfkc6wdym01/.

[16] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.

[17] Gavin Lowe. A hierarchy of authentication specifications, 1997. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=596782.

[18] S. Mauw, J. Verschuren, and E.P. de Vink. A Formalization of Anonymity and Onion Routing. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Proc. Esorics 2004*, pages 109–124, Sophia Antipolis, 2004. LNCS 3193. http://www.springerlink.com/content/pjx13crve5xg4gf9/fulltext.pdf.

[19] Sjouke Mauw and Victor Bos. Drawing Message Sequence Charts with LaTeX. *TUGBoat*, 22(1-2):87–92, March/June 2001. http://satoss.uni.lu/mscpackage/.

[20] Catherine Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends, 2003. chacs.nrl.navy.mil/publications/CHACS/2003/2003meadows-issues.pdf.

[21] Andreas Pfitzmann and Marit Köhntopp. Anonymity, unobservability, and pseudeonymity — a proposal for terminology. In *International workshop on Designing privacy enhancing technologies*, pages 1–9, New York, NY, USA, 2001. Springer-Verlag New York, Inc. http://www.springerlink.com/content/xkedq9pftwh8j752/fulltext.pdf.

[22] Andrei Serjantov and George Danezis. *Towards an Information Theoretic Metric for Anonymity*, volume 2482/2003 of *Lecture Notes in Computer Science*, pages 259–263. Springer, 2003. http://www.springerlink.com/content/wwe2c7g3hmwn0klf/fulltext.pdf.

[23] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. Towards an analysis of onion routing security. In *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 96–114. Springer Berlin / Heidelberg, 2001. http://www.springerlink.com/content/5ec5ac6930q8m35p/.

A Identity protection - example proofs

Here we prove and disprove identity protection in two example protocols.

A.1 Example of proving identity protection

In Figure 8 we show a simple protocol in which two parties use a server to agree on a key. In this setup, only the server has a public key, and the initiator and the responder use a password shared with the server to authenticate. The protocol aims to achieve identity protection for the initiator.

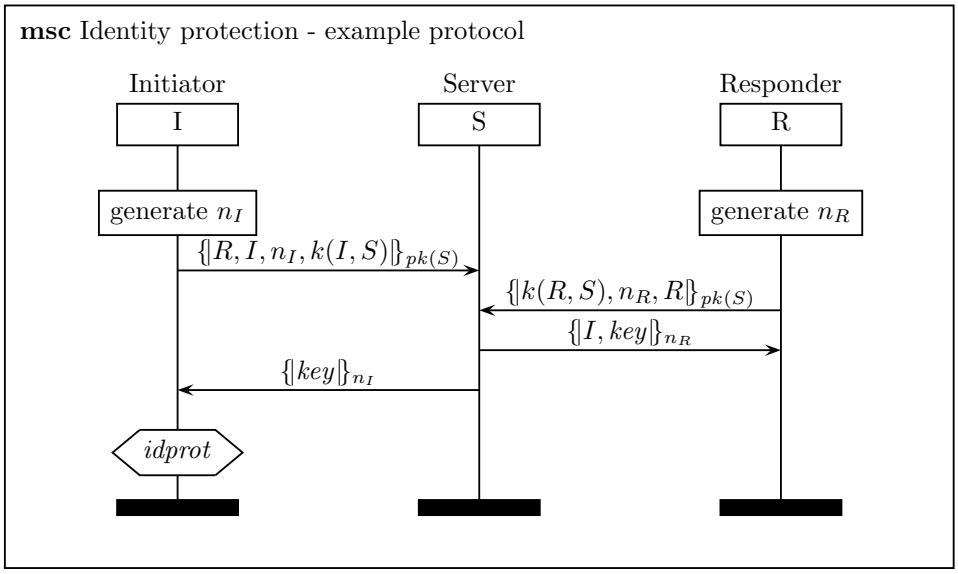


Figure 8: Example protocol in which we prove identity protection for the initiator.

We here list the protocol specification:

$$
\begin{aligned}
I = \; & send_1(\{|R, I, n_I, k(I,S)|\}_{pk(S)}) \\
& recv_4(\{|V_1|\}_{n_I}) \\
& claim_{I_1}(idprot)
\end{aligned}
$$

$$
\begin{aligned}
S = \; & recv_1(\{|R, I, V_2, k(I,S)|\}_{pk(S)}) \\
& recv_2(\{|k(R,S), V_3, R|\}_{pk(S)}) \\
& send_3(\{|I, key|\}_{V_3}) \\
& send_4(\{|key|\}_{V_2})
\end{aligned}
$$

$$R = send_2(\{|k(R,S), n_R, R|\}_{pk(S)})$$
$$\quad recv_3(\{|I, V_4|\}_{n_R})$$

In the first message, the initiator logs in to the server, gives a key n_I for the encryption of the response, and asks to communicate with the agent playing role R. The server then waits for this agent to log in.

When the responder logs in, he also sends a response encryption key. Back he gets the initiator's identity and a key. This key is also sent to the initiator, and I and R can use it for further communication.

Using Scyther with the input found in Appendix B, we have proved that this protocol satisfies non-injective synchronization at the end of all the roles, and that secrecy holds for all keys and nonces. We will now prove that the group identity protection claim holds.

We will do this by assuming that the claim does not hold, and then arrive at a contradiction. That is, denoting the protocol by p, we assume the following:

$$\begin{aligned}
&\exists_{\alpha \in Tr(p)} \\
&\quad \Big(\big(\forall_{rid \in RunId, r \in Role} : \rho_{\alpha,rid}(r) \in Agent_S \Rightarrow r = I \big) \wedge \\
&\quad \big(\forall_{rid \in RunId} : agentof_{\alpha}(rid) \in Agent_s \Rightarrow \\
&\quad \forall_{r \in dom(\rho_{\alpha,rid}) \setminus I} \ \rho_{\alpha,rid}(r) \in Agent_T \big) \Big) \wedge \\
&\quad Agent_S \cap M(\alpha) \neq \emptyset.
\end{aligned} \tag{1}$$

This means that the intruder must have read the name of at least one secret agent from a message in the trace. Since secret agents are, due to the requirements on the traces, only assigned to the role I, this must have been from a message containing an instantiation of the role name I. We have two possibilities, instantiations of $send_1(\{|R, I, n_I, k(I,S)|\}_{pk(S)})$ or $send_3(\{|I, key|\}_{V_3})$. Since messages are instantiated in a sequence, this means the intruder must have learned a secret agent name from one of them first.

We first assume the intruder first learned a secret agent name from $send_1$. That requires a run rid by a secret agent a playing the initiator, in which this message is instantiated. That is:

$$\begin{aligned}
&\exists_{i \in N, rid \in RunId} \\
&\quad \rho_{\alpha,rid} = \{I \rightarrow a, R \rightarrow r, S \rightarrow s\} \wedge \\
&\quad \alpha_i = send_1(\{|r, a, n_I \sharp rid, k(a,s)|\}_{pk(s)})
\end{aligned}$$

For the agent name a to be inferable from this message, the intruder must know the inverse of the encryption key, which is $sk(s)$. Since this key is never sent in

the protocol, this requires the agent s to be compromised. That means we have the following:

$$agentof_{\alpha}(rid) \in Agent_S \ \wedge \ \rho_{\alpha,rid}(S) \in Agent_U.$$

This is a contradiction to the requirement we put on the trace α in (1), and we conclude that the intruder could not first have learned a secret agent name from $send_1$.

The second possibility is for the intruder to first learn a secret agent name from an instantiation of $send_3(\{|I, key|\}_{V_3})$, done in a run of the server role. This requires the following for the trace α breaking the claim:

$$\begin{aligned}&\exists_{i\in N, rid\in RunId}\\ &\quad \rho_{\alpha,rid} = \{I \rightarrow a, R \rightarrow r, S \rightarrow s\} \wedge\\ &\quad \alpha_i = send_3(\{|a, key\sharp rid)|\}_{\sigma_{\alpha,rid}(V_3)})\sharp rid\end{aligned}$$

For the agent name to be inferable from this message, the intruder must know the encryption key, assigned to the variable V_3 by $\sigma_{\alpha,rid}$. This assignment happens for the server in $recv_2$, and the following must hold for the trace α:, for $j < i$:

$$\alpha_j = recv_2(\{|k(r,s), \sigma_{\alpha,rid}(V_3), r|\}_{pk(s)})\sharp rid$$

For the key in V_3 to be known by the intruder, we have two possibilities:

First, the key is inferable if the intruder knows the key $sk(s)$. Since this key is never sent, it must be a part of the initial knowledge of the intruder, requiring the agent s to be compromised.

We note that to reach this event, the agent playing the server, although compromised, must have instantiated $recv_1$. That is, for $k < j$:

$$\alpha_k = recv_1(\{|r, a, \sigma_{\alpha,rid}(V_2), k(a,s)|\}_{pk(s)})\sharp rid$$

Since the agent s is compromised, this means the intruder already knew a, as the agent s received it in $recv_1$. This contradicts our assumption that the intruder first learned the secret agent name in $send_3$.

Secondly, the message could have been constructed by the intruder. This also requires that the intruder already knew the secret agent name, and again contradicts with our assumption.

We have now proved that the identity protection claim holds for the example protocol.

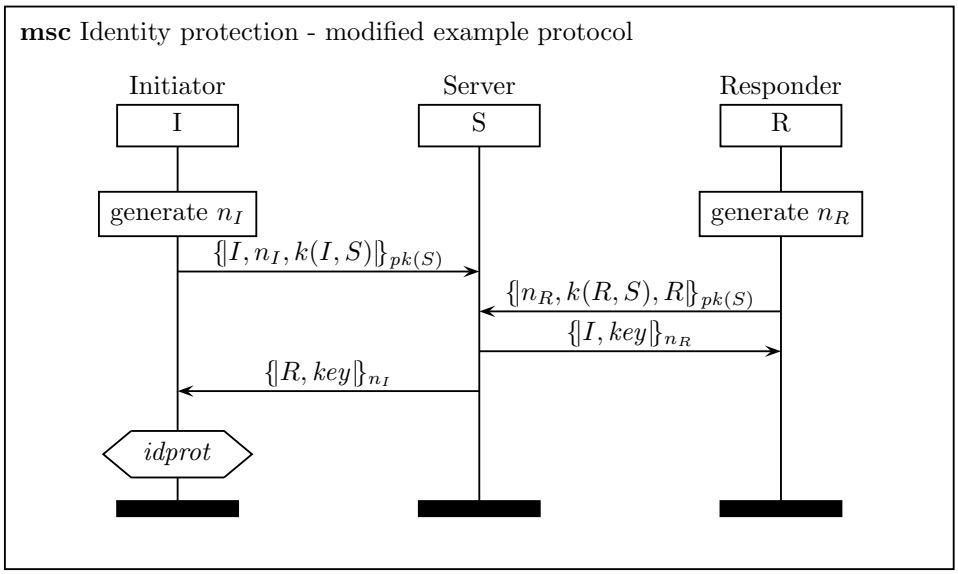


Figure 9: Example protocol in which we prove that identity protection does not hold.

A.2 Example of disproving identity protection

We now slightly modify the protocol in the previous proof, by moving the inclusion of the responder agent from the first to the last pair of send and receive events in the protocol. We show this modification in Figure 9.

As with the previous example protocol, Scyther proves that this protocol satisfies non-injective synchronization at the end of all the roles, and that secrecy holds for all keys and nonces. The change does only affect the identity protection claim.

We show that identity protection does not hold by giving the following trace, in which the secret agent a starts as the initiator. In this trace, $a \in Agent_S$, $s \in Agent_T$ and $r \in Agent_U$.

$$\alpha_1 = create(run = (1, \{I \rightarrow a\}, \emptyset))$$
$$\alpha_2 = send_1(\{|a, n_I\sharp 1, k(a,s)|\}_{pk(s)})\sharp 1$$
$$\alpha_3 = create(run = (2, \{S \rightarrow s\}, \emptyset))$$
$$\alpha_4 = recv_1(\{|a, n_I\sharp 1, k(a,s)|\}_{pk(s)})\sharp 2$$
$$\alpha_5 = create(run = (3, \{R \rightarrow r\}, \emptyset))$$
$$\alpha_6 = send_2(\{|k(r,s), n_R\sharp 3, r|\}_{pk(s)})\sharp 3$$
$$\alpha_7 = recv_2(\{|k(r,s), n_R\sharp 3, r|\}_{pk(s)})\sharp 2$$
$$\alpha_8 = send_3(\{|a, key\sharp 2|\}_{n_R\sharp 3})\sharp 2$$

If we nest up the role assignment function for the run of the secret agent a, we find that he has never assigned an agent to the role R, as he has not yet encountered the role name in the specification he is instantiating. We have:

$$agentof_\alpha(1) \in Agent_S \ \wedge \ \rho_{\alpha,1} : Role \rightarrow Agent_T$$

In the trace, no secret agent is assigned any other role than I, and therefore this trace should be considered by the claim. Yet, because agent r is compromised, the intruder can read the secret agent name a in α_8. Therefore the trace breaks the implication required by the claim, and the protocol does not satisfy identity protection for role I.

Using Theorem 3, we also know that the protocol does not satisfy group identity protection for the initiator role, for any group of trusted roles.

B Identity protection - spdl

We here include the spdl code for the protocol example of identity protection used in Figure 8. This is the file needed for analysis in the automatic verification tool Scyther. All claims have been proved, without bounds on the number of runs.

Note that we have included the sender and receiver in all messages, as this is required by Scyther. This does, however, not affect the claims proven here.

```
const pk: Function;
secret sk,k: Function;
inversekeys (pk,sk);

protocol idprot(I,S,R)
{
        role I
        {
```

```
            const nI: Nonce;
            var V1: Nonce;

            send_1( I, S, { R, I, nI, k(I, S) }pk(S) );
            read_4( S, I, { V1 }nI );

            claim_i1( I, Nisynch );
            claim_i2( I, Secret, V1 );
            claim_i3( I, Secret, k(I,S) );
        }
        role S
        {
            const key: Nonce;
            var V2,V3: Nonce;

            read_1( I, S, { R, I, V2, k(I, S) }pk(S) );
            read_2( R, S, { k(R, S), V3, R }pk(S) );
            send_3( S, R, { key }V3 );
            send_4( S, I, { key }V2 );
        }
        role R
        {
            const nR: Nonce;
            var V4: Nonce;

            send_2( R, S, {k(R, S), nR, R }pk(S) );
            read_3( S, R, { V4 }nR );

            claim_r1( R, Nisynch );
            claim_r2( R, Secret, V4 );
            claim_r3( R, Secret, k(R,S) );
        }
}

const Alice,Bob,Eve: Agent;
// An untrusted agent, with leaked information
untrusted Eve;
compromised sk(Eve);
```

C Hierarchy of authentication requirements - spdl

We here include the spdl code for the protocol example used in Section 3.2.4. This is the file needed for analysis in the automatic verification tool Scyther, in which the non-injective synchronization claim holds, without bounds on the number of runs.

```
const pk: Function;
```

```
secret sk: Function;
inversekeys (pk,sk);

protocol p(I, R, S)
{
        role I
        {
                const n: Nonce;

                send_1(I, S, { n, I, S }pk(S) );
                read_2(S, I, { n }pk(I) );
                send_3(I, R, { I, n }pk(R) );
                read_4(R, I, { n, R }pk(I) );

                claim_I1(I, Nisynch);
        }
        role R
        {
                var V1: Nonce;

                read_3(I, R, { I, V1 }pk(R) );
                send_4(R, I, { V1, R }pk(I) );
        }
        role S
        {
                var V2: Nonce;

                read_1(I, S, { V2, I, S}pk(S) );
                send_2(S, I, { V2 }pk(I) );
        }
}

const Eve, Alice, Bob, Charlie: Agent;
untrusted Eve;
compromised sk(Eve);
```

D Proposed protocol - spdl

We here include the spdl code for the protocol proposed in this thesis, as presented in Section 4. Using this file, the reader can run tests on the protocol in Scyther himself.

Note that we have included the sender and receiver in all messages, as this is required by Scyther, and that Scyther does not support our new security requirements.

The modified protocol presented in Section 5.1 is realized by uncommenting the

parts involving the dummy agent D.

We refer to the Scyther Manual[6] for information on the definition language.

```
const pk,h: Function;
secret sk,k: Function;
inversekeys(pk,sk);

// User types, to avoid type flaw attacks
usertype Contract;
usertype SessionID;
usertype Key;

protocol proto(U, S, N) // proto(U, S, N, D)
{
        role U // User
        {
                const k1 : Key;
                const k2 : Key;
                const c : Contract;
                var V1 : Nonce;

                send_1(U, S, { k1, U }pk(S) );
                send_2(U, S, { U, c, N }k1 );
                read_3(S, U, { V1, { h(c, U, S, V1) }sk(S) }k1 );
                send_4(U, N, { k2, U }pk(N) );
                send_5(U, N, { k(U, N) }k2 );
                send_6(U, N, { { h(c, U, S, V1) }sk(S), U }k2 );
                read_9(N, U, { { { h(c, U, S, V1) }sk(S), U}sk(N) }k2 );
                send_10(U, S, { { { h(c, U, S, V1) }sk(S), U}sk(N) }k1 );
        }
        role S // Store
        {
                const r : Nonce;
                var V2 : Key;
                var V3 : Contract;

                read_1(U, S, { V2, U }pk(S) );
                read_2(U, S, { U, V3, N }V2 );
                send_3(S, U, { r, { h(V3, U, S, r) }sk(S) }V2 );
                read_10(U, S, { { { h(V3, U, S, r) }sk(S),U}sk(N) }V2 );
        }
        role N // Notary
        {
                var V4 : Key;
                var V5 : Ticket;

                read_4(U, N, { V4, U }pk(N) );
                read_5(U, N, { k(U, N) }V4 );
                read_6(U, N, { V5, U }V4 );
```

```
        //      send_7(N, D, ( V_5, U) );
        //      send_8(N, D, { V_5, U }sk(N) );
                send_9(N, U, { { V5, U}sk(N) }V4 );
        }
        // role D // Dummy
        //{
        //      var V_6 : Ticket;
        //
        //      recv_7(N, D, ( V_6, U) );
        //      recv_8(N, D, { V_6, U }sk(N) );
        //}
}

const Mallory, Alice, Bob, Charlie: Agent;
// An untrusted agent, with leaked secret key:
untrusted Mallory;
compromised sk(Mallory);
compromised k(Mallory, Alice);
compromised k(Alice, Mallory);
compromised k(Mallory, Bob);
compromised k(Bob, Mallory);
compromised k(Mallory, Charlie);
compromised k(Charlie, Mallory);
```