Compositionality of security protocols

independence, message encoding, simulation and name attacks

Ceelen, P.

Award date:
2008

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Master's Thesis

Compositionality of Security Protocols

Independence, Message encoding, Simulation
and Name attacks

by
Pieter Ceelen

Supervisors:

prof.dr. J.C.M. Baeten (TU e)
Prof. dr. S. Mauw (UdL)

Eindhoven, December 2007

Abstract

Formal analysis of security protocols has been researched the last decades, recent developments introduced compositionality into the domain of security protocols. These developments enable verification of larger protocol sets. This thesis investigates improvements to the theories regarding compositionality of security protocols. Compositionality properties are studied and improved from various angles: the conditions under which compositional reasoning can be applied is weakened, enabling a wider application of compositional reasoning. The remodelling of security protocols such that they have better compositionality properties is investigated, and the expressive power of a security protocol is shown by a Turing completeness proof. Furthermore a subtlety in the assumptions used in the formal analysis of security protocols emerged during the research. Specially constructed protocols exploit this assumption which led to the discovery of new classes of attacks.

Preface

This Master's Thesis concludes my study at the Department of Mathematics & Computer Science at the Technische Universiteit Eindhoven. The majority of the work executed for this thesis has been done at the Université du Luxembourg, I worked in Luxembourg from February 2007 until the end of July 2007. I finalised this thesis after I returned to the Netherlands. The research done focusses around the concept of compositionality of security protocols. I tried to improve existing theories by looking at compositionality from different perspectives.

I would like to thank Sjouke Mauw and Saša Radomirović for the challenging assignment, the freedom they gave me when doing my research, the time they spent on discussions and the useful feedback. A word of gratitude towards Jos Baeten and the university of Luxembourg for enabling the opportunity do my master thesis abroad. Furthermore I would like to thank Cas Cremers, Kristian Gjøsteen, and Suzana Andova for the interesting discussions, helpful comments and the preview of working in academia. A word of thanks for the members of my Assessment Committee for the time they spent reading and judging my work.

I would also like to thank Hugo Jonker, Baptiste Alcalde, and all other people I met in Luxembourg for the fun we had and the fact that you all made me feel at home in Luxembourg. Finally, I would like to thank my girlfriend, Carolien, for her support during my master's project.

Contents

Chapter 1

Introduction

In current society, the ways of communication between humans have changed in recent years. The current possibilities of e-mail, instant messaging, and the Internet gives the opportunity to communicate with another human at any moment in any place. When using one of these communication means, the content will be transferred over the Internet; when using the Internet there is the risk of an adversary attacking your communication session. This is specifically a problem if one wants to transfer information with a security requirement (for example a secret) from A to B. To solve this problem *security protocols* have been designed.

A security protocol is a specification of the behaviour of honest agents such that the communication between these agents meets certain security requirements.

An example of a security protocol is the process executed when a person logs in to an Internet banking website. We can summarise the important steps as follows:

Go to Internet banking website and retrieve a number

Insert this number and PIN-code in hardware token

Insert response of hardware token on the website

The behaviour of the bank is divided into the following steps:

Pick a new random number (a challenge)

Transfer this number to the client

Retrieve response of client

Validate response of the client

One security requirement for this protocol is that the client is authenticated, which means that the bank is certain that you are the genuine owner of the bank account.

It is very hard to determine whether a security protocol meets the security requirements related to the protocol. The adversary can attack a protocol in numerous ways, yet excluding all currently known attacks does not prove achievement of a security requirement. To solve these problems numerous formal models have been developed which enable us to detect protocol attacks in a systematic way or deliver a proof of correctness.

One of the formalism's used to reason about the correctness of security protocols is the Cremers-Mauw operational semantics [14]. This formal model has been developed at the Technische Universiteit Eindhoven. This model is constructed in an intuitive manner, it is well documented and research is done in improving and extending this model.

One of the recent developments of this model is the work on compositionality. In general the formal models are aimed at the verification of small protocols, verifying a set of protocols (a protocol suite) requires some form of compositional reasoning. In [1] a framework is presented which extends the Cremers-Mauw semantics with compositional reasoning. A key ingredient of the presented framework is the notion of *independence* which expresses in which cases the compositional reasoning can be applied.

The purpose of this project is the following:

Study and improve the framework for compositional reasoning

The starting point for improvement of the formal framework is the notion of independence. One of the existing notions seems to be a rather ad-hoc solution; a more thorough study will probably improve this notion.

Another open question is related to the expressive power of the Cremers-Mauw semantics; it is not exactly clear what the limitations are on the security protocols used as input for these semantics.

To answer the main goal of the project, the following actions are defined:

Try to improve current independence notions.

Study what limitations there are on the protocols that can be verified.

During the work on the improvement of the independence notions questions emerged regarding the correctness of the Cremers-Mauw semantics; a more thorough study of certain exceptional settings lead to the discovery of a new class of attacks.

In Chapter 2 the basic formal model is introduced; Chapter 3 contains the work on the framework for compositional reasoning; it summarises the existing notions and ideas, and the research results on independence are described. In Chapter 4 a newly discovered class of attacks is introduced. Chapter 5 contains the research results regarding the expressiveness of the current model. Final conclusions are drawn in Chapter 6.

Chapter 2

Formal model

One of the first approaches towards a formal analysis of security protocols is the BAN–logic [11], this formalism uses postulates and definitions to prove protocols correct. In [26] Lowe illustrated an attack on a protocol proven correct in BAN–logic, this flaw was discovered using a tool called Casper/FDR. The flaw was not detected in the BAN–logic due to different assumptions on the intruders behaviour.

Since the publication of the attack various formal models and supporting tools have been developed. The Strand Spaces [39] approach provides a formal model which enables a rather elegant way of reasoning about protocol executions. The Cremers-Mauw semantics [15] take a similar approach as the Strand Spaces model. One of the differences between these models is the fact that in the Cremers-Mauw semantics the relation between a protocol and a protocol's execution is explicitly formalized, where the Strand Spaces model uses a partial order on events. An advantage of this approach is that the Cremers-Mauw protocol specifications are in general easier to read and understand.

In the succeeding section a general overview of the Cremers-Mauw semantics is given, Section 2.2 describes how a protocol is specified; in Section 2.3 it is shown how a protocol specification can be executed. Section 2.4 treats security properties and in Section 2.5 the model will be applied on an example protocol.

2.1 Overview

The actions of honest agents are specified in role specifications. A role specification is an ordered sequence of read, send and claim events defining the exact behaviour of honest agents. Claim events denote the fact that a certain security goal should be achieved at that moment in time.

A protocol is a set of role specifications expected to communicate together (e.g. one can have a role specification for an initiator and responder role, combining these two specifications resulting in a security protocol).

An agent is an entity able to execute a role specification, for instance Alice is an agent executing the role of the initiator. The actual execution of the role specification is

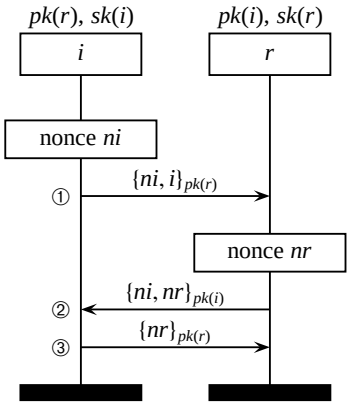

Figure 2.1: The Needham–Schroeder key exchange protocol

called a run. A run can be compared with a recording of all input and output messages in a protocol execution in the real world, all variables from the specification are replaced by values. A run is local to an agent, it only contains events of one agent. The communication between two agents is modelled by a buffer which is placed under full control of the adversary; the content of this buffer can be seen as the Dolev–Yao intruder knowledge.

All possible executions of the entire system are modelled by a set of traces; a trace is a sequence of events of the complete system. Reasoning about the set of traces allows us to determine whether a certain security claim is met in all possible executions.

The Needham–Schroeder protocol [34], as described in Figure 2.1, will be used as a running example throughout this chapter. This protocol is known to be flawed, in Section 2.5 it is illustrated how this flaw is detected by the formal model.

2.2 Protocol specification

A protocol is specified by a number of role specifications, each role specification is a list of events which specifies the behaviour of the agent. Each of these events contains role terms, which are used to specify message content. One can compare a protocol to the preparation instructions of a dinner. Role specifications are similar to the recipe of one dish, the role specification is a list of actions (the role events) which need to be executed in a certain order. In this comparison ingredients are analogous to role terms, since they are both inputs for the events.

In the remainder of this section we will formalise the notion of a security protocol, before we can do this formal definitions for role term, role event and role specification are introduced.

2.2.1 Role terms

Role terms are terms which represent the data used in events. They are used to specify the contents of messages.

Let $\mathcal{ID}$ be a set of identifiers, $\mathcal{R}$ a set of role names and $\mathcal{F}$ a set of functions. The identifiers can be divided into 3 types, constants, variables and parameters

The term (x, y) denotes the pairing of x and y, $\{x\}_y$ represents the encryption of term x with key y.

Definition 2.2.1. *The set of role terms is defined as:*

$$\begin{aligned} \mathit{RoleTerm} ::= \mathcal{ID} \mid \mathcal{R} \mid \mathcal{F}(\mathit{RoleTerm}\ ) \\ \mid (\mathit{RoleTerm}, \mathit{RoleTerm}) \mid \{\mathit{RoleTerm}\}_{\mathit{RoleTerm}} \end{aligned}$$

Some remarks need to be made regarding this definition:
The value of an identifier is local to a role, in the Needham–Schroeder example there is no global idea of the nonce *ni*. The formal model enables reasoning about the value of *ni* for the initiator role, or *ni* for the intruder role. This locality of identifiers implies that the value of a constant is initially only known by one role; which is exactly the behaviour of a nonce in a security protocol.

Functions with arity 0 are used to model constants whose value is known to all agents, these constants relate to constants defined in standards and RFC's etcetera.

The set $\mathcal{F}$ is a global set of all functions, we assume that their arity is respected in all terms. Functions are only used to model global constants or to model long term keys which are already established by a surrounding key infrastructure, typical examples of long term keys are public keys ($pk(r)$), private keys ($sk(i)$) and symmetric keys ($k(x, y)$). Short term session keys are represented by local constants.

Terms which are encrypted can only be decrypted by the inverse of the key, the inverse of the key can be the key itself (symmetric cryptography) or an inverse key (asymmetric encryption). Furthermore we assume that pairing is right associative, thus $(x, y, z) = (x, (y, z)), (x, y, z) \neq ((x, y), z)$

The subterm relation $\sqsubseteq$ on role terms is defined as the smallest transitive relation satisfying:

$$x_1 \sqsubseteq x_1, \quad x_1 \sqsubseteq (x_1, x_2), \quad x_2 \sqsubseteq (x_1, x_2), \quad x_1 \sqsubseteq \{x_1\}_{x_2}, \quad x_2 \sqsubseteq \{x_1\}_{x_2}.$$

for any roleterm x_1, x_2

2.2.2 Role events

Role events specify atomic actions of an agent. An event $send_\ell(r, r', x)$ represents the sending of the data x from agent r intended for agent r'. Read events are defined in a similar way, $read_\ell(r', r, x)$, which resemble the reading of data x by r', apparently sent by r. The third type of events are the claim events, $claim_\ell(r, c\,[, x])$, which claim that a certain security property c is true for agent r, security properties will be discussed in more detail in section 2.4. There are two special events for the start and end of a role, respectively $create_\ell(r)$ and $end_\ell(r)$, where r is the name of the role. The label ℓ is used to tag events; the purpose of this tagging is to make multiple occurrences of similar events non-ambiguous.

Role events are formally defined as:

$$\mathcal{E} = \{create_\ell(r), send_\ell(r, r', x), read_\ell(r', r, x), claim_\ell(r, c\,[, x]), end_\ell(r) \mid \ell \in \mathcal{L}, r, r' \in \mathcal{R}, x \in RoleTerm, c \in Claim\}$$

The sequential order of the list of events in a role R is represented by the ordering relation $\prec'_R$. Thus for 2 events $\__1$, $\__2$ in role R, $\__1 \prec'_R \__2$ if $\__1$ occurs before $\__2$.

A role specification can now be defined by the pair $(elist, type)$ where $elist \in \mathcal{E}$ is a list of events and $type : \mathcal{ID} \to \{const, param, variable\}$ is a function assigning the types to identifiers. We only consider lists *elist* starting with a *create* and having a *end* as the final event. The *create* and the *end* are not allowed at any other position in this list. Furthermore, we require that the role name in these events are the same. This role name must also be used in the claim events, and as the sender and recipient in send and read events respectively. This name is called the *role name*. If one of these requirements is not met, the specification is invalid. The set of all valid specifications is called *RoleSpec*.

2.2.3 Protocols

Security Protocols are partial mappings $\mathcal{R} \to RoleSpec$ that map a role name r to a role specification *elist*. Let P be a protocol such that $r \in dom(P)$ (r is a role in protocol P), we write $\ell \in P(r)$ if a label ℓ occurs in the event list of r and extend this notation such that we can use $\ell \in P$. $\mathcal{ID}(P)$ is used to denote the set of identifiers occurring in the protocol P. And *Prot* represents the universe of protocols .

The labels of two related read and send events in a protocol need to be the same, all other labels need to be unique for a given protocol set $\Pi \subseteq Prot$. This requirement does not limit the expressive power; one can always rename events in such a way that they are unique. We extend the $\prec'$ relation to a protocol P by taking the union of all the $\prec'_R$ relations for all roles R in P. This relation is extended with an ordering on related read and send events; the *send* events precede the corresponding *read* event. The transitive closure of the relation $\prec'$ is denoted with $\prec$ and this order resembles causalities in the events of the protocol.

Example 1. *The formal security protocol given here corresponds to the Needham–Schroeder protocol of Figure 2.1.*

$$\begin{aligned} NS(i) = (&create_1(i), send_2(i, r, \{ni, i\}_{pk(r)}), read_3(r, i, \{ni, nr\}_{pk(i)}) \\ &send_4(i, r, \{nr\}_{pk(r)}), claim_5(i, secret(nr)), end_6(i), \\ &\{ni \mapsto const, nr \mapsto variable\}) \\ NS(r) = (&create_7(r), read_8(i, r, \{ni, i\}_{pk(r)}), send_9(r, i, \{ni, nr\}_{pk(i)}) \\ &read_4(i, r, \{nr\}_{pk(r)}), claim_8(r, secret(nr)), end_9(r), \\ &\{nr \mapsto const, ni \mapsto variable\}) \end{aligned}$$

In this example the protocol consist of two roles, i and r; both of these roles have a role specification. It is clear that the role events in these role specifications have a unique label except for corresponding read and send events. Notice that in the role i the identifier ni is a constant, in the r ni is a variable.

2.3 Protocol execution

The transformation of role specifications to executions of that role are done by a process called *instantiation*, the execution of a role is called a *run*. Multiple runs can be combined into *traces*, traces are event sequences of the entire system; they contain, possibly interleaved, events of a collection of runs. In this section the instantiation process and the construction of the traces is defined.

On the abstract specification level the set *RoleTerm* was defined as an specification of message content. The instantiation of these role terms are defined the *run terms*, run terms contain the actual messages.

Let $\mathcal{A}$ be a set of concrete agent names, this set contains trusted as well as untrusted agents. $\mathcal{IT}$ denotes the set of intruder generated nonces. Furthermore, we assume the existence of a set run identifiers *Runid*.

$$RunTerm ::= \mathcal{A} \mid \mathcal{F}(RunTerm\ ) \mid \mathcal{ID}\sharp Runid \mid \mathcal{IT} \mid (RunTerm, RunTerm) \mid \{RunTerm\}_{RunTerm}$$

The addition of a run identifier to an $\mathcal{ID}$ makes each local constant unique, another difference between role and run terms is that the variables are instantiated by concrete values in the run terms. A subterm relation $\sqsubseteq$ is defined in a similar way on run terms as it was defined on role terms. The same symbol can be used for this relation since it is clear from the context which relation is intended.

The instantiation process is defined by a run identifier *rid* and two functions; the partial function $\quad : \mathcal{R} \rightarrow \mathcal{A}$ defines which agent is executing which role. The function $\quad : \mathcal{ID} \rightarrow RunTerm$ is a partial function which assigns run terms to identifiers. The actual value of an identifiers can be an arbitrary complicated run term. The instantiation function $inst_{(rid,\ ,\ )}$: $RoleTerm \rightarrow RunTerm$ is only defined if all the variables occurring in the role term are in the domain of $\quad$ and all occurring role names have a agent name assigned to it by $\quad$.

If both conditions are fulfilled than any role term x from a given specification can be transformed into a run term using the function inst.

Definition 2.3.1. *Let $rs = (elist, type)$ be a role specification, let $x_1, x_2..x_n$ be arbitrary role terms, instantiation has the following recursive definition:*

$$inst(x) = \begin{cases} (r) & \text{if } x \quad r \in \mathcal{R} \\ c\sharp rid & \text{if } x \quad c \in \mathcal{ID} \wedge type(c) = const \\ (x) & \text{if } x \in \mathcal{ID} \wedge type(x) \in \{param, variable\} \\ f(inst(x_1), \ldots, inst(x_n)) & \text{if } x \quad f(x, \ldots, x_n) \\ (inst(x_1), inst(x_2)) & \text{if } x \quad (x, x_2) \\ \{inst(x_1)\}_{inst(x_2)} & \text{if } x \quad \{x\}_{x_2} \end{cases}$$

The instantiation of the events of a role specification $rs = (elist, type)$ is named a *run*. A run is defined as the pair $(inst, elist')$ such that *inst* is an instantiation triplet and $elist'$ is a suffix of *elist*.

We do not require the event list $elist'$ to be the complete role specification, it is sufficient to have a list of the remaining events of a given role specification. This modelling expresses the focus on the current state of an agent. The actual values of variables and agent names are contained in the functions and of the instantiation $inst$. We use $Runs$ to denote the set of all runs. A *run event* is a pair $(inst, ev) \in Inst \quad \mathcal{E}$; run events are the actual events occurring in the execution of a system. A *trace* is a sequence of run events which represents the behaviour of the system. The set of all possible traces is denoted by $Traces$.

The set of runs that can possibly be created by a protocol P is defined by a function $runsof(P)$. A run $(inst, elist)$ is in the set $runsof(P)$ if and only if $dom(\ ) = dom(P)$ and $dom(\ ) = type\ ^{1}(param)$. This formal requirement expresses that the runs in $runsof(P)$ all have values assigned for all role parameters (names as well as variables).

We define the set of $runsof(\Pi)$ for a protocol set Π

$$runsof(\Pi) = \bigcup_{P \in \Pi} runsof(P).$$

The system that we consider consists of a number of runs executed by agents in parallel, the agents communicate with each other asynchronously through a network buffer. We use an intruder model which is based upon the Dolev–Yao model, thus we assume that the intruder has full control over the communication network. The intruder knowledge, denoted by M, is a set of run terms; the intruder is able to deduce new run terms by encrypting or decrypting terms if the key used for this operation is also in M.

$$\forall_{u,v \in M}(u, v) \in M \Rightarrow \{u\}_v \in \overline{M}$$
$$\forall_{u,v}\{u\}_v, v \ ^{1} \in M \Rightarrow u \in \overline{M}$$
$$\forall_{u,v}(u, v) \in M \Leftrightarrow u, v \in \overline{M}$$

$\overline{M}$ is used to denote the smallest closed superset of M.

During the execution of a system the intruder knowledge grows, the initial knowledge is denoted with M_0 and contains the names and public keys of agents and the secret keys of compromised agents. The initial knowledge can be derived from the protocol and the context the protocol is running in; in [15] this derivation is treated in more detail.

The buffer M is not only used to represent the intruder knowledge, it is also used to model the asynchronous communication between agents; when an agent send out a certain term, the intruder learns this term. Before an agent can read a term, this term has to be known by the intruder.

The system is defined by a state transition system, states are determined by the pair (M, F) where M is an intruder knowledge set and F a set of active runs. $runids(F)$ denotes the set of run identifiers appearing in F. We use $F[x/y]$ to denote the replacement of x by y. The transitions in the system are labelled with a run event.

The derivation rules of the system are given in Table 2.1. The *create* rule expresses that the run identifier used by a new run cannot be used by another run in the system. The *end* and *claim* rules express that these events have no prerequisites when executing

these events. The *send* rule states that when a run sends a term m, the instantiation of this term m is added to the intruder knowledge M and the executing run proceeds.

The *read* rule states that terms in the intruder buffer are accepted in a read event if there is a match on the pattern for some instantiation of the variables. The matching of a message t in the intruder knowledge on a role term m of the specification is done by a predicate *Match*; the idea of the predicate is that a new instantiation $inst'$ which extends $inst$ by assigning values to free variables such that the incoming message t equals the instantiation of m.

$$Match(inst, m, t, inst') \iff inst = (rid, \quad , \quad) \wedge inst \leftarrow (rid, \quad , ') \wedge \quad ' \wedge$$
$$dom(\quad ') = dom(\quad) \cup var_{rs}(m) \wedge inst'(m) = t \wedge WellTyped(\quad ').$$

The exact definition of the predicate $WellTyped$ is a parameter of the formal model. This parameter expresses whether type flaws are allowed. Type flaws occur when the type of a message is misinterpreted [10], for example an encryption term might be confused with a nonce. When the implementation of the protocol is such that these misinterpretations are prevented then we can use the following definition of WellTyped:

Definition 2.3.2. *We define the predicate WellTyped that expresses whether a substitution is well defined:*

$$WellTyped(\quad) = \forall v \in dom(\quad) : \quad (v) \in type(v)$$

where type is a function which returns a set of allowed values for a certain variable.

This definition expresses that if a variable v is of type "Agent name" then only agent names will be accepted.

In an environment where type-flaws are allowed a variable can match any arbitrary run term. The following definition of $WellTyped$ expresses this fact:

Definition 2.3.3. *Let be a substitution, then we set*

$$WellTyped(\quad) = True$$

to express that type-flaws are allowed.

In Chapter 3 and 4 type-flaws and the related attacks are treated in more detail.

The possible state transitions in the system are defined in table 2.1. All possible behaviour of a protocol can be derived by applying these derivation rules to the initial configuration $\Sigma_0 = (M_0, \emptyset)$.

Let be a trace of length $| \quad | = n$; $_i$ is used to denote the ith event in the trace . A trace is a *valid* trace if and only if there are states $\Sigma_1, ..\Sigma_n$ such that $\Sigma_0 \xrightarrow{0} \Sigma_1 .. \xrightarrow{n} \Sigma_n$ can be derived using the derivation rules of Table 2.1. The set of valid traces of a protocol Π is denoted by $Tr(\Pi)$.

$$[create]\frac{run = (inst, create_\ell(r) \quad elist) \in runsof(\Pi), inst = (rid, \quad , \quad), rid \notin runids(F)}{\langle M, F\rangle \xrightarrow{(inst, create_\ell(r))} \langle M, F \cup \{(inst, elist)\}\rangle}$$

$$[end]\frac{run = (inst, end(r)) \in F,}{\langle M, F\rangle \xrightarrow{(inst, end_\ell(r))} \langle M, F[(inst, \varepsilon)/run]\rangle}$$

$$[send]\frac{run = (inst, send_\ell(m) \quad elist) \in F}{\langle M, F\rangle \xrightarrow{(inst, send_\ell(m))} \langle \overline{M \cup \{inst(m)\}}, F[(inst, elist)/run]\rangle}$$

$$[read]\frac{run = (inst, read_\ell(m) \quad elist) \in F, t \in M, Match(inst, m, t, in's)}{\langle M, F\rangle \xrightarrow{(inst', read_\ell(m))} \langle M, F[(inst', elist)/run]\rangle}$$

$$[claim]\frac{run = (inst, claim_\ell(r, c\,[, x]) \quad elist) \in F}{\langle M, F\rangle \xrightarrow{(inst, claim_\ell(r, c\,[, x]))} \langle M, F[(inst, elist)/run]\rangle}$$

Table 2.1: Derivation rules.

2.4 Security Properties

In the derivation rules claim events are used to mark the positions in a trace where a certain claim should be valid. For instance a claim can express the fact that a certain term is secret; this fact should only be considered if a claim event is reached in a valid trace. Security properties express how to evaluate a claim event in a trace by defining a formal statement which is evaluated when a security claim is encountered in a trace. As an example we will discuss the secrecy claim. Detailed information on other security claims like authentication and data-agreement can be found in [14, Chapter 3].

Definition 2.4.1. *Let f_{cl} be a security property, let Π be a protocol set, and let ℓ be the label of a claim event with claim cl. We say that* Π satisfies *the claim ℓ, denoted by* $\mathrm{sat}(\Pi, \ell)$*, if*

$$\forall \quad \in Tr(\Pi) \forall i : {}_i = (inst, claim_\ell(r, cl, m)) \Rightarrow$$
$$f_{cl}(\Pi, claim_\ell(r, cl, m))(inst, \quad) \vee (inst = (\quad , \quad , \quad) \wedge im(\ {}_T).\not\in \mathcal{A}$$

A secrecy claim expresses that certain information is never revealed to an intruder; in the formal model this is expressed by demanding that the intruder knowledge M does not contain the secret information.

Definition 2.4.2. *The security property f_{secret} associates to the protocol set Π and the claim event $ev = claim_\ell(r, secret, m)$ the statement*

$$f_{secret}(\Pi, ev)(inst, \quad) \Leftrightarrow \quad \in Tr(\Pi) \wedge inst(m) \notin M_{|+1},$$

where the initial intruder knowledge is determined from Π.

2.5 Example

In Figure 2.2 a trace is constructed which is based upon the modelling of the Needham–Schroeder protocol (Example 1). This trace illustrates the famous attack on the Needham–Schroeder protocol. The secrecy claim of the responder role is violated, the intruder uses a man in the middle attack in which an honest initiator (executed by agent a) is used as a decryption oracle for the second message.

The trace is an example of the application of the formal model. When working with this model trace derivations and claim validation is supported by a tool which has implemented this formal model. This tool enables a fully automated verification of a security protocol.

instantiation	event	concrete term
$(1, {}_1, \{ni \mapsto u, nr \mapsto \bot\})$	$create_1(i)$	-
$(2, {}_2, \{ni \mapsto \bot, nr \mapsto v\})$	$create_7(r)$	-
$(1, {}_1, \{ni \mapsto u, nr \mapsto \bot\})$	$send_2(i, r, \{ni, i\}_{pk(r)})$	$\{u, a\}_{pk(eve)}$
$(2, {}_2, \{ni \mapsto \bot, nr \mapsto v\})$	$read_2(i, r, \{nr, i\}_{pk(r)})$	$\{u, a\}_{pk(b)}$
$(2, {}_2, \{ni \mapsto ni\sharp 1, nr \mapsto v\})$	$send_3(i, r, \{ni, nr\}_{pk(i)})$	$\{u, v\}_{pk(a)}$
$(1, {}_1, \{ni \mapsto u, nr \mapsto nr\sharp 2\})$	$read_3(i, r, \{ni, nr\}_{pk(i)})$	$\{u, v\}_{pk(a)}$
$(1, {}_1, \{ni \mapsto u, nr \mapsto nr\sharp 2\})$	$send_4(i, r, \{nr\}_{pk(i)})$	$\{v\}_{pk(eve)}$
$(2, {}_2, \{ni \mapsto ni\sharp 1, nr \mapsto v\})$	$read_4(i, r, \{nr\}_{pk(i)})$	$\{v\}_{pk(b)}$
$(2, {}_2, \{ni \mapsto ni\sharp 1, nr \mapsto v\})$	$claim_8(r, secret(nr))$	v

Figure 2.2: Example trace of the Needham–Schroeder protocol, ($_1 = \{i \mapsto a, r \mapsto eve\}$, $_2 = \{i \mapsto a, r \mapsto b\}$)

The solution for the problem indicated with this attack is the insertion of the name of the responder in the second message. This prevents the adversary from forwarding this message into another protocol run. The resulting Needham–Schroeder–Lowe protocol is depicted in Figure 2.3

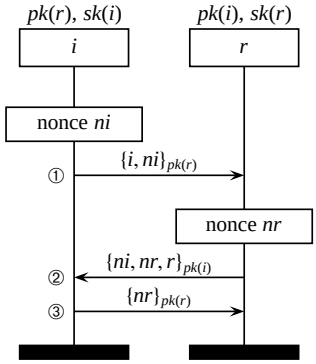

Figure 2.3: The amended Needham–Schroeder key exchange protocol

Chapter 3

Improving Independence Properties

In the previous chapter a formal framework is introduced which enables us to verify properties of security protocols. The verification process is a complex task and the complexity increases when verifying larger protocols. In general the time needed for verification using a tool grows exponentially in the number of messages in the protocol.

Consequently verification of large protocols is infeasible for the current tools. In [1] a framework for reasoning about compositionality of security protocols is introduced. This framework uses a divide and conquer approach which enables us to derive the correctness of properties of a large protocol suite by combining verification results of multiple smaller protocols.

In this chapter we will try to answer the following question:

Can we improve current concepts or ideas regarding the compositionality of security protocols?

Section 3.1 contains a short introduction to the framework and results of [1]. An improvement to the theory regarding compositionality is introduced in Section 3.2. Section 3.3 illustrates how remodelling of protocols can help in achieving better compositionality properties.

3.1 Current definitions

3.1.1 Independence

We say that two protocols are independent if no encryption term produced by the first protocol running in the context of the second protocol will be decrypted or verified by the second protocol, and vice versa. Formally:

Definition 3.1.1. *Let* Π_1, Π_2 *be two disjoint protocol sets. We say that* Π_1 and Π_2 are independent*, denoted* indep(Π_1, Π_2)*, if*

$$\begin{aligned}
&\forall \quad \in Tr(\Pi_1 \cup \Pi_2; \quad) \; \forall x, y, x', y' \in RoleTerm : \\
&\Big({}_i = (inst, send_\ell(m)) \wedge (\quad {}_j = (inst', read_{\ell'}(m')) \vee \quad {}_j = (inst', send_{\ell'}(m')) \vee \\
&\qquad\qquad {}_j = (inst', claim_{\ell'}(\quad , \quad))) \wedge \\
&\quad (\{x\}_y \sqsubseteq m \wedge \{x'\}_{y'} \sqsubseteq m' \wedge inst(\{x\}_y) = inst'(\{x'\}_{y'}))\Big) \\
&\Rightarrow (\ell, \ell' \in \Pi_1 \vee \ell, \ell' \in \Pi_2).
\end{aligned}$$

Intuitively, this definition expresses that two protocols are called independent if there is no encryption runterm produced by one of the protocols which can be read, send or claimed by the other protocol.

In [1] a number of interesting theorems are proved using this definition of independence, e.g.

Theorem 3.1.1. *Let* Π_1 *and* Π_2 *be two independent protocol sets. Let* ℓ *be the label of some secret claim event in* Π_1*. Then*

$$\text{sat}(\Pi_1, \ell) \Rightarrow \text{sat}(\Pi_1 \cup \Pi_2, \ell).$$

a proof of this theorem can be found in [1, Thm. 20]

Theorem 3.1.1 expresses that adding an independent protocol set Π_2 to the environment Π_1 is running in does not break a secrecy claim of Π_1.

3.1.2 Strong independence

Proving that two protocols are independent is non-trivial; in [1] the assumption is made that only nonce run terms can be assigned to variables and parameters. Using this assumption there is a form of strong independence defined as follows:

Definition 3.1.2. *Let* Π_0 *and* Π_1 *be two disjoint protocol sets. We say that* Π_0 and Π_1 are strongly independent*, denoted* s-indep(Π_0, Π_1)*, if for any* $b \in \{0, 1\}$*, any role specification* $(elist \quad send(m) \quad e, type)$ *in a protocol in* Π_b*, any role terms* x, y*, any role specifications* $(elist'' \quad send(m) \quad elist, type')$*,* $(elist'' \quad read(m) \quad elist, type')$ *or* $(elist'' \quad claim(r, c, m) \quad elist, type')$ *in protocols of* Π_{1-b}*, any map* s *on the set* $\mathcal{ID}$ *and any map* s' *on the set* $\mathcal{R}$*,*

$$\{x\}_y \sqsubseteq m \Rightarrow \{s(s'(x))\}_{s(s'(y))} \not\sqsubseteq m'.$$

This definition expresses that two protocols are called strongly independent if there is no encryption role term in one of the protocols which after renaming the variables (using the maps s and s') occurs in the other protocol.

Note that any map s on identifiers and s' on roles naturally induce maps on the set of role terms and we identify these maps with s and s'. Also note that since strong independence is a syntactical property, there is no need to consider traces in evaluating the strong independence criteria.

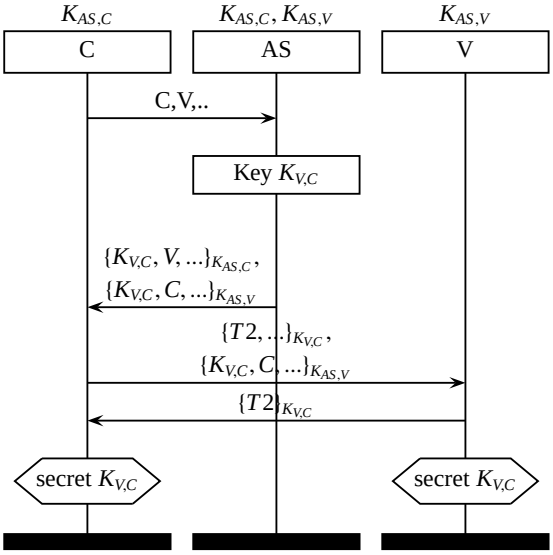

Figure 3.1: The Kerberos protocol

In [1, thm. 19] a proof is given which shows that strong independence implies independence.

As a consequence this compositionality framework enables us to detect whether two protocols are strong independent without evaluating all possible traces (it is a syntactical property). When two protocols are strong independent then they are independent according to the previously mentioned theorem. Theorem 3.1.1 shows that in order to evaluate the correctness of a secrecy claim it is sufficient to evaluate this claim within an environment where only the related protocol exists. Thus the traceset $Tr(\Pi_1 \cup \Pi_2)$ is not evaluated; this framework removes the problems related to the computational complexity related to the evaluation of larger protocol sets.

A problem with the strong independence definition as suggested above lies in the assumption that only nonce run terms are assigned to variables and parameters. There are numerous protocols using a form of *tickets*; in these protocols this assumption is not true. These protocol require an agent to forward a message part (the ticket) to another agent; this ticket usually is an encryption term which cannot be read by the agent forwarding the message. An example of such a protocol is the Kerberos protocol [35], as illustrated in Figure 3.1. This protocol is widely used in practise. For example, it is a basic building block of the Windows networking environment.

The Kerberos-protocol uses an authenticated server *AS* to establish a session key between agents C and V. Agent C receives a term encrypted with the key $K_{AS,V}$ as a part of the second message. The agent forwards this message part to agent V. This is called ticketing and should be modelled as a variable which is assigned a complex run term.

Another problem with the assumption used in the strong independence definition is the fact that type flaws are excluded. Type flaws [10] occur when one message is confused with another message which has another type. For example an encryption term might be interpreted as a nonce by another agent. There is a special class of attacks called the *type-flaw attacks* in which the adversary abuses confusion of the type

of a message to break a security protocol. There are mechanisms to prevent type flaws (see [19]), unfortunately not all implementations of security protocols incorporate these mechanisms due to limitations in message size and computational load for the agents.

3.2 Improving strong independence

The requirements for an improved definition of strong independence are the following:

It must be possible to prove independence using the improved definition

It should be a syntactical check

The assumptions used in the new definition should be less restrictive then the current assumption

Using this list of requirement we tried to look for an algorithm which did not require the assumption on the assignment to variables and parameters. A rather ad-hoc algorithm was sketched but after a more thorough study of literature it turned out that the independence definition is strongly related to the unification problem.

3.2.1 Unification

Unification is a fundamental theoretical computer science area with a long history. The unification problem answers the following question: given a term s containing free variables u and a term t with free variables v, is there an assignment to the variables in u and v such that s and t are syntactically the same.

Applications of unification theory appear in various areas of computer science, most notable it is used in theorem proving, areas of artificial intelligence and in term rewriting systems.

An algorithm which gives an answer to the unification problem would help proving the independence property; if a term $s = \{x\}_y$ and $t = \{x'\}_{y'}$ cannot be unified then $\nexists_{inst}\ inst(\{x\}_y) = inst'(\{x'\}_{y'})$. The following theorem expresses the relation between relation between unification and independence more formally:

Theorem 3.2.1.

$$\Big(\forall m_1 \in \Pi_1, m_2 \in \Pi_2\ \forall x_1, y_1, x_2, y_2 \in RoleTerm :$$
$$\{x_1\}_{y_1} \sqsubseteq m_1 \wedge \{x_2\}_{y_2} \sqsubseteq m_2\ : \neg Unify(\{x_1\}_{y_1}, \{x_2\}_{y_2})\Big)$$
$$\Rightarrow \text{indep}(\Pi_1, \Pi_2)$$

Proof. By the fact that there are no encryption terms from Π_1 which can be unified by an encryption term of Π_2 it is clear that the independence criterion $inst(\{x\}_y) = inst'(\{x'\}_{y'})$ can only be true if both terms $\{x\}_y$ and $\{x'\}_{y'}$ originated from the same protocol, which is exactly the independence requirement. □

3.2.2 Unification algorithm

In the previous section it is shown that an algorithm solving the unification problem would help in proving protocols independent. There are numerous algorithms for unification, Algorithm 3.2.1 illustrates a rather naive recursive descent approach. In [4] it is shown that this algorithm returns the most general unifier if the terms can be unified, otherwise it returns that the terms are not unifiable. Unfortunately the time and space needed for this algorithm can be exponential in the size of the terms. More complex algorithms ([37], [30]) have reduced this to a problem with linear complexity for instances with non associative/non commutative terms.

Algorithm 3.2.1: UNIFY(s, t)

global : substitution

if s is a variable
then s s
if t is a variable
then t t
if s is a variable $\wedge$ $s = t$
then {do nothing}
else if $(s = f(s_1, ..s_n) \wedge t = g(t_1, ..t_m)$
then
- **if** $f = g$
- **then for** i 1 **to** n
- **do** $Unify(s_i, t_i)$
- **else** exit function, non unifiable, symbol clash

else if s is not a variable
then $Unify(t, s)$
else if s occurs in t
then exit function, non unifiable, occurs check
else $\{s \mapsto t\}$

Algorithm 3.2.1 walks through both terms from left to right, a global substitution is maintained which is used to store the values of variables. If t or s is a variable then it is substituted by its substitution value in . When s and t are both functions then the arguments are unified and when s is a free variable then the substitution is extended with the replacement $s \mapsto t$.

3.2.3 Application of unification

In this section Theorem 3.2.1 is applied in a setting where strong independence cannot be used.

Consider a setting in which a Kerberos protocol (Figure 3.1) and the NSL-protocol Figure 2.3) occur in the same environment. Due to the ticket used in the Kerberos protocol strong independence cannot be used in this setting.

Since all the messages of NSL are encrypted with a key of the form $pk(A)$ and all encryptions in the Kerberos protocol are of the form $K(A, B)$ it might look if none of

Syntax	Size	Notes
PKM-REQ_Message_Format() {		
Management Message Type = 9	8 bits	
Code	8 bits	
PKM Identifier	8 bits	
TLV Encoded Attributes	*variable*	TLV specific
}		

Figure 3.2: A part of the WiMAX specification, (PKM request message format)

the encryption terms can be unified with a term of the NSL protocol. Unfortunately this argumentation is not exactly correct, consider the following exact modelling of the V-role of the Kerberos protocol:

$$\begin{aligned} Kerberos(V) \quad = \quad & create_1(V) \\ & read_2(C, V, \{T2, ...\}_{KVC}, \{KVC, C, ...\}_{K_{AS,V}} \\ & send_3(V, C, \{T2\}_{KVC}) \end{aligned}$$

In this role KVC is a variable which is read in the event $read_2$; thus the message sent out by $send_3$ is actually of the form $\{x\}_y$ where x and y are both variables. This message can be unified with the first message of the NSL-protocol ($\{Ni\}_{pk(r)}$). As a result these protocols cannot be proven independent using Theorem 3.2.1. In the succeeding section we study this problem further and we demonstrate how a remodelling of the protocols can help in achieving independence.

3.3 Improving independence using message encoding

In the previous sections the independence requirements are improved by relating independence to the unification of messages. However there are still settings where two protocols are theoretically not independent. In this section we will look at the modelling of protocols from the independence perspective. The goal of the succeeding sections is to give some theory, guidelines and examples on how to remodel protocols such that they are more likely to be independent.

The general concept behind the remodelling is the addition of details who are omitted in the original role specification. These details can be either information on the cryptography used or it is information related to the encoding of the message.

3.3.1 Concrete message specification

When implementing a security protocol detailed agreements must be made on how data is exchanged. Depending on the use of the protocol this is done in a design document or it is defined in a standard or RFC. These documents define message structures and the exact encoding of the contents; they specify which bits should be set and how the encryption algorithm should be applied. The specification of the messages at bit-level will be called the concrete message specification, Figures 3.2 and 3.3 illustrate parts of the concrete message specification as used in the WiMAX standard [36].

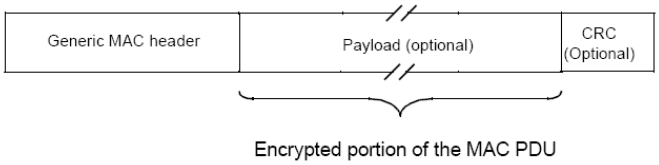

Figure 3.3: A part of the WiMAX specification, (MAC PDU encryption)

The concrete message specification contains lots of details, most of them are not of interest for security analysis purposes. These concrete specifications are interpreted and transformed into abstract message specification, which are message specifications as shown throughout this thesis, thus messages of the form $\{na\}_{pk(n)}$. The process of abstracting away unnecessary details is done manually and the aim of this abstraction process is to get a version of the protocol which is as short as possible while still having all properties relevant for the security of the protocol. Certain details might be omitted in the abstraction process which are interesting with respect to unification and independence. We introduce the concept of a *Unification preserving* abstraction which enables us to use properties of concrete message specifications on the abstract message specification level without introducing too many details on the abstract level.

Definition 3.3.1. *We call a protocol abstraction of protocol* Π Unification preserving *iff for any two concrete message specifications* cm_1 *and* cm_2 *used in* Π *with abstract message specifications* m_1, m_2 *respectively*

$$unify(cm_1, cm_2) \Leftrightarrow unify(m_1, m_2)$$

In the two succeeding sections we will look at protocol specification and improve abstract message specifications such that the abstraction process is unification preserving.

3.3.2 Cryptography

There are numerous cryptographic algorithms which are currently used, e.g. RSA, ECC and AES. These algorithms are based upon different mathematical properties and the results of these algorithms are in general incompatible with each other. A message encrypted with RSA cannot be decrypted with an ECC algorithm and vice versa.

However, when specifying a protocol using such an encryption algorithm there is only one operator available to specify an encryption. If we look at the example illustrated in Section 3.2.3 we see 2 messages which can be unified to each other although the two messages use different encryption techniques (Kerberos uses a symmetric encryption algorithm, NSL uses a public key algorithm).

Under the assumption that if two messages use different encryption algorithms there is no way that these messages can be unified, we can change the abstract messages such that they are non-unifiable. This is done using global constants; in the introduction of the formal model it was already mentioned that global constants are modelled by nullary functions. We introduce new global constants *sym_enc*() and *pub_enc*() which represent symmetric encryption and public encryption respectively.

The new abstract role specification for role V in the Kerberos protocol becomes :

$$
\begin{aligned}
Kerberos(V) \quad = \quad & create_1(V) \\
& read_2(C, V, \{sym\_enc(), T2, ...\}_{KVC}, \{sym\_enc(), KVC, C, ...\}_{K_{AS,V}} \\
& send_3(V, C, \{sym\_enc(), T2\}_{KVC})
\end{aligned}
$$

In a similar way the encryption terms in the NSL specification are extended with the nullary function *pub_enc*(). The abstraction is now unification preserving since the message $\{sym\_enc(), Ts\}_{KVC}$ cannot be unified with a message of the NSL protocol in the concrete as well as the abstract message specification. Using this modelling the protocols are independent by Theorem 3.2.1.

Instead of introducing global constants for symmetric and public key encryption one can also create a global constant for each encryption algorithm, thus *RSA_enc*(), *ECC_enc*(), *AES_enc*() etcetera.

This technique might be of particular use in settings where type flaws are permitted. Using different encryption schemes it is still possible to achieve independence between protocol sets.

3.3.3 Message encoding

In the previous section a demonstration is given on how to use properties of the encryption algorithm used to show that two protocols are independent. There are other properties of the concrete message specification which ensure that messages cannot be unified; in this section a commonly used encoding scheme is studied and it is shown how to take advantage of such a scheme in the role specifications.

As an example we will study a type-length-value(TLV) encoding scheme, in this scheme each data element is encoded as a type identifier, a length identifier, the actual data elements. The type and length identifiers have a fixed length (usually 1 to 4 bytes) and the length of the actual data elements is defined by the length identifier. There are many protocols which use a form of TLV encoding, TLV encoding is the basic building block of the Basic Encoding Rules (BER) [21] of the ASN.1 notation. ASN.1 is a widely accepted standard in the area of notation and encoding. The message encoding in WiMAX [36] is also based upon a TLV scheme.

As an example how we can take advantage of the encoding from the independence perspective we will consider a setting where 2 agents use Kerberos (Figure 3.1) to obtain a new key and they use this key to transfer data using the protocol of Figure 3.4

The Kerberos protocol and the data transfer protocol cannot be proven to be independent of each other since the last message of the key negotiation protocol $\{Ts\}_{KVC}$ can be unified with $\{data\}_{KVC}$.

If these protocols use a TLV encoding scheme the specification contains a table which defines identifiers for all the fields used in the protocol. An example of such a table is given in Table 3.1. If an agent needs to send the data ”1234” he will construct a message ”7/4/1234”, the receiving party sees that this is data (the message starts with a 7) and that the length of the data is 4 bytes.

In [38] it is proven that the composition of encoding and decoding using the Basic Encoding Rules (a commonly used form of TLV) yields a value which is equivalent

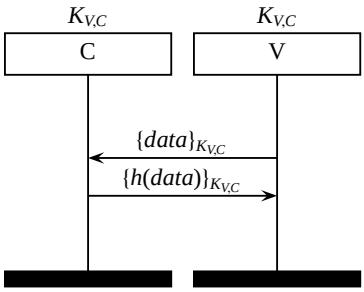

Figure 3.4: A data transfer protocol using a Kerberos key negotiation

Byte value	Field identifier
0 5	*reserved*
6	Ticket (Ts)
7	Data

Table 3.1: An example table of field identifiers

to the original. The TLV encoding prevents the confusion between a message containing the ticket and a message containing data by the fact that the first byte of the message content has a value of either a 6 or a 7. Using global constants *byteval*_6() and *byteval*_7() in the abstract specification we can express this property. The last message of the kerberos protocol becomes $\{byteval\_6(), Ts\}_{KVC}$ and first message of the data transfer protocol is modelled as $\{byteval\_7(), data\}_{KVC}$. Applying the unification algorithm immediately results in a symbol clash; the function *byteval*_6() and *byteval*_7() are not unifiable. Using this modelling of the protocols it is clear that messages from one protocol cannot be unified with messages of the other protocol and thus these two protocols are independent.

3.4 Conclusions

In this chapter we studied some aspects of compositionality of security protocols. We have shown that the problem of independence can be reduced to the unification problem. Using an unification algorithm we were able to prove that protocols are independent without the assumption used in the definition of strong independence . A basic unification algorithm was sufficient for the formal model used throughout this thesis, other formal models might need unification algorithms aimed at unifying commutative-associative functions or even first order unification depending on the specific properties of the input language.

In the second part of this chapter the relation between abstract and concrete message specifications was studied and it was shown how properties of the concrete specifications can be used to improve compositionality properties of a protocol.

Chapter 4

Unification & Name attacks

During the research on unification, compositionality and type flaws the question emerged which terms used in the model should be unifiable to which other terms. More specifically how does an agent name relate to role terms.

In the literature attacks existed where the adversary picks a nonce such that it conflicts with an agent name breaking a security requirement. An example of such an attack is the type flaw attack on a version of the Needham-Schroeder protocol as illustrated in Figure 4.1. In this variant of Needham-Schroeder the order of the arguments in the first message are swapped.

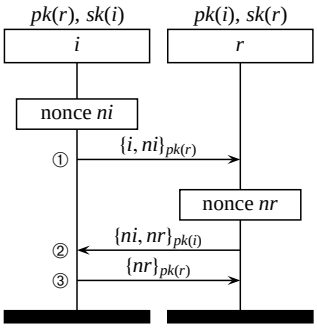

Figure 4.1: Needham-Schroeder public-key authentication protocol

In this attack (Figure 4.2) a malicious agent uses his name instead of a nonce in the first message in order to impersonate another agent. The attack requires two parallel sessions and is shown in detail in Figure 4.2. We write $a : r(b, a)$ to denote that agent a executes the responder role r assuming that the corresponding initiator role i is executed by agent b. Similarly, $eve(b) : i(b, a)$ means that the intruder *eve* impersonates b executing the initiator role in a session with agent a executing the responder role.

The type-flaw attack on the Needham-Schroeder protocol is an example of a situation where the adversary selects a nonce according to the value of an agent name. The question whether an adversary could change his name according to the value of a nonce lead to the insight that the current formal model excluded this possibility although this name-changing might be possible in certain scenarios.

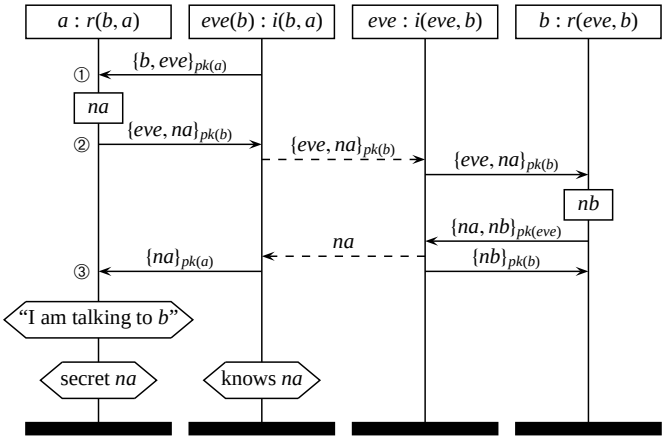

Figure 4.2: Type-flaw attack on Needham-Schroeder

Common to most security frameworks is the assumption that the adversary has complete control over the network and is only limited by the constraints of cryptographic operations. The first attempt to precisely formulate this idea was done in the early 1980s by Dolev and Yao [17].

Dolev and Yao's threat model assumes that the adversary is *a legitimate user of the network*, able to *be a receiver to any user on the network*, *obtain any message passing through the network*, and trying *everything he can in order to discover the plaintext* of an encrypted message.

There are mechanisms to ensure correctness of types throughout the execution of a protocol, for instance by message tags [19], therefore the absence of type-flaw considerations in some formalizations can be justified.

In this chapter, we introduce and discuss new intruder abilities concerning the dynamic selection of names for compromised agents and potentially even for honest agents. In contrast to the well-known type-flaw attacks, such as the one shown on the Needham-Schroeder protocol, which are based on the assumption that agent names are static, we allow the adversary to choose dynamically created terms as the name of an agent. Dynamic agent naming is typically not being considered by protocol verification frameworks and tools.

While it is easy to craft secure protocols that are susceptible to what we call the *chosen-name type-flaw attacks*, we don't expect these attacks to be found frequently in real-life protocols. However, to prove their existence, we also show a chosen-name type-flaw attack on a published protocol upon which previously no attacks were known. In the attack, using a dynamically created name, the intruder takes advantage of a type flaw to learn a shared key. This type-flaw attack would not be possible with a static name.

These new attacks and intruder abilities have, to the best of our knowledge, not been taken into account in any formalization. The achieved results have been published in [12].

4.1 Chosen-name attacks

4.1.1 Preliminaries

A *chosen-name attack* occurs when the intruder violates a security property by generating a suitable name or identity for an agent. Clearly, the less restricted an agent's name space is, the more likely this type of attack is to occur.

We distinguish between two types of chosen-name attacks. In a *selected-name attack*, the intruder can select arbitrary names for compromised agents only, while in an *assigned-name attack* the intruder may additionally assign arbitrary names to uncompromised, i.e. honest agents. Aside from the fact that assigned-name attacks are much more specialized than selected-name attacks, the two classes also have distinct targets. In selected-name attacks malicious agents choose their name to attack other agents, hence the veracity of security properties for these agents may be ignored, while in assigned-name attacks the victim may very well be the agent that is being assigned a name. Note that while the attacker may assign a name to an honest agent in order to attack another victim, in general these attacks can also be executed as selected-name attacks.

Although attacks that would fit our definition of chosen-name attacks have been known and described in the literature, they have not been treated as a class of their own, but have been rather occurring as instances of other classes, such as impersonation attacks, man-in-the-middle attacks, or relay attacks. The *chosen-name type-flaw* attacks, however, are new and did not receive attention before. In the rest of this section we will therefore restrict ourselves to chosen-name type-flaw attacks. We will discuss existing chosen-name attacks in Section 4.2.

4.1.2 Selected-name attacks

We consider attacks where the intruder dynamically selects the names of conspiring agents in such a way that a security claim fails due to a type flaw. We split these selected-name attacks into two subclasses. The first class consists of those attacks where an agent's selected name will be accepted without further scrutiny, while the second class requires the name to be confirmed or accepted by a third party, for instance by a certificate authority or a key server.

We begin by presenting and discussing a protocol vulnerable to an attack from the first class and then discuss Lowe's modification of the KSL protocol which is vulnerable to an attack from the second class. In both cases we only consider secrecy, even though our methods can be used to attack any security property.

A flawed key-establishment protocol

Consider the protocol in Figure 4.3. It is a fictitious key establishment protocol combined with a liveness check. The premise is that initiator i and server s share the secret key $k(i, s)$ and similarly responder r and server s share the secret key $k(r, s)$. On i's communication request to r in message ①, r contacts the server s in ②, who then distributes a fresh secret key kir to r and i in messages ③ and ④. When r receives the new

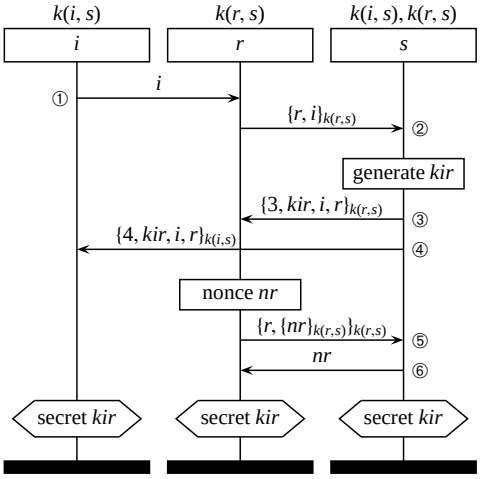

Figure 4.3: Key-establishment protocol

key *kir* he generates a nonce *nr* to check liveness of the server in ⑤ and ⑥. To demonstrate the selected-name attack, the liveness check is implemented in a non-standard manner. The security claim of this protocol for all three roles is that *kir* is secret, as indicated by the hexagons at the end of the protocol.

Since this is a specially crafted protocol, we first sketch why the protocol achieves its security goals in the absence of type-flaw attacks. The key *kir* is freshly produced by *s*, and transmitted only in messages ③ and ④, encrypted with $k(r, s)$ and $k(i, s)$, respectively. In both messages, *kir* is concatenated with *i* and *r* and encrypted with keys known only to *s* and one of *i* and *r*. Thus each of the messages binds *kir*, *i*, *r*, and *s* together. It follows that *kir* is secret for *i*, since all messages containing *kir* are bound to the intended agents, must have been produced by the intended agent, and are only readable by the intended agents. The same reasoning can be applied for *r* and *s*'s secrecy claim.

Next, we consider conventional type flaws. It is evident that messages ②, ③, and ④ cannot interfere with each other due to the message identifiers contained in messages ③ and ④. Messages ② and ⑤ are supposed to be distinguishable by the fact that one contains an agent name and the other one an encryption term. In a conventional type-flaw attack, an adversary may attempt replacing message ⑤ in a certain run by message ②, possibly from another run. In the present protocol, this attack would be futile, as it would, at best, lead to an encryption term $\{i\}_{k(r,s)}$ [1]. This term would not be useful for the adversary to break the secrecy of *kir*, since it does not fit the type of any other message.

Finally, in a selected-name attack, the replacement can be attempted in the other direction, too, i.e. message ② in a certain run may be replaced by message ⑤ from another run. Figure 4.4 shows a trace demonstrating this attack. Assume that the adversary controls two agents, an agent called *eve* who pretends to be *b* and an agent who will be named $\{3, kab, a, b\}_{k(b,srv)}$. Agent *eve* listens to a conversation between the honest agents *a*, *b*, and *srv*. When message ③ is sent from *srv* to *b*, *eve* intercepts this message and the agent with the name $\{3, kab, a, b\}_{k(b,srv)}$ is created. This agent initiates a

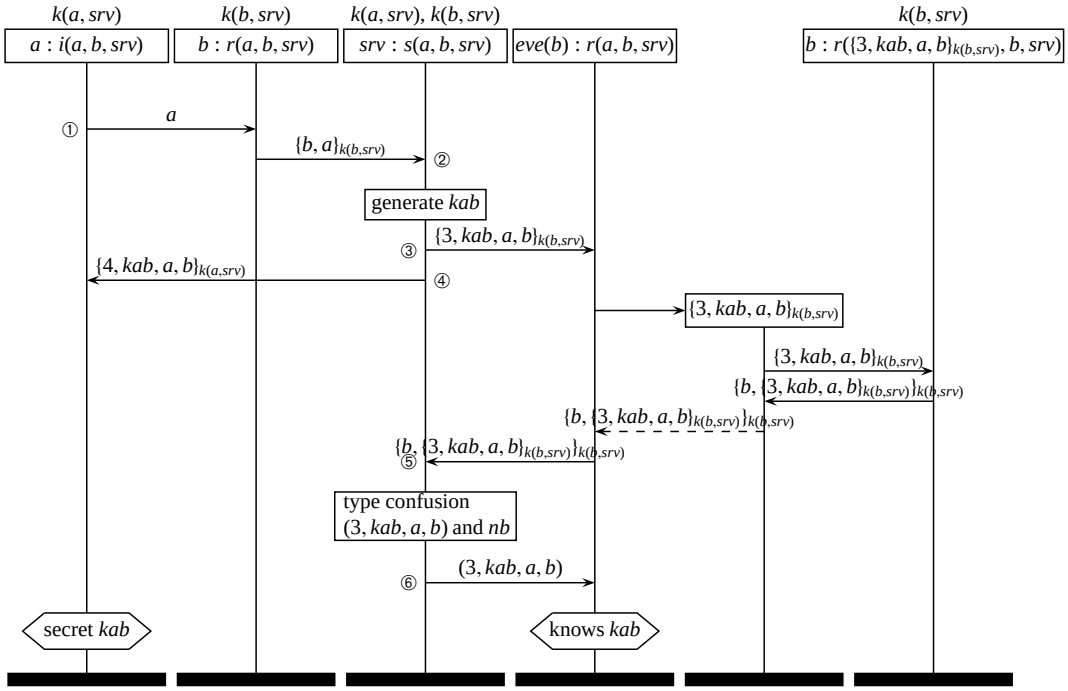

Figure 4.4: Attack on key-establishment protocol

session with a second run of b. Following the protocol, agent b constructs the message $\{b, \{3, kab, a, b\}_{k(b,srv)}\}_{k(b,srv)}$ which he sends to srv. The adversary intercepts this message and injects it into the first session as message ⑤, impersonating b to srv. Agent srv tries to decrypt the message and obtains $(3, kab, a, b)$ due to a type confusion. This quadruple is sent back in the clear allowing the adversary to learn kab.

Lowe's modified KSL

The selected-name attack presented in the previous section only required the adversary to select names for the agents he controls. Some selected-name attacks additionally require the conspiring agent to have his selected name accepted by a third party, for instance when obtaining a symmetric key associated with the name from a key server or a certificate from a certificate authority. The ability to obtain key material or certificates for a chosen name is plausible in identity-based encryption and signature schemes and in systems where users may have one or more pseudonyms.

As an example of a protocol vulnerable to a selected-name attack under the assumption that the chosen name is accepted, we consider the KSL protocol [22] including the modifications suggested by Lowe in [27]. In [29] an exact modelling of Lowe's modifications is provided. We will focus on the authentication phase of this protocol, shown in Figure 4.5, and omit the reauthentication protocol.

This protocol is similar to the key-establishment protocol discussed in the previous example in that in messages ① through ③ i contacts r, who in turn contacts the key

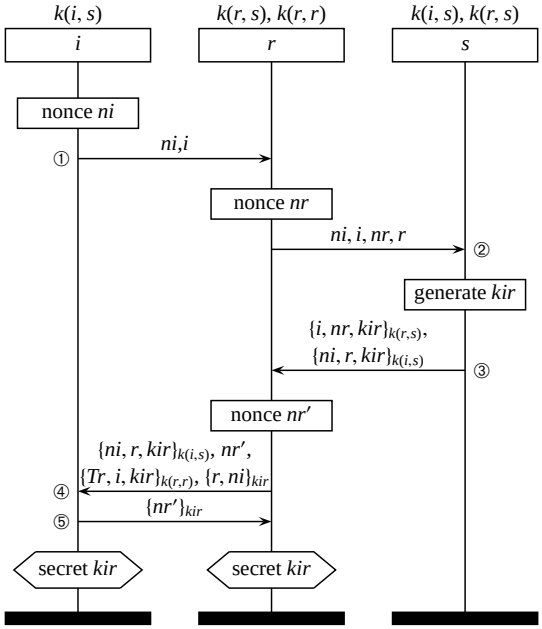

Figure 4.5: Lowe modified KSL

server s to obtain shared secret keys. Here, however, nonces are generated and sent in the first two messages, and neither of the first two messages is encrypted. Further, the server s does not deliver the encrypted shared secret key to i directly, but rather sends it to r in message ③, who forwards the encrypted key along with another fresh nonce nr', a ticket $\{Tr, i, kir\}_{k(r,r)}$, and i's original nonce encrypted under the shared secret key to i. Finally, i sends back nr' encrypted with the received shared secret key kir. The ticket in message ④, is only used for the reauthentication protocol which we have omitted. It is encrypted with the key $k(r, r)$ known only to r and contains a generalized time stamp, Tr, made with respect to r's local clock. The fact that the key $k(r, r)$ is only known to r prevents everybody but r to tamper with the ticket or create such a ticket. In the reauthentication protocol r uses Tr to check the validity of the ticket.

Until now, there have been no attacks known on this protocol. In fact, if our chosen-name attacks are disregarded, then the secrecy claims of the protocol can be shown to be correct using, for instance, the Cremers-Mauw semantics [15].

To carry out a selected-name attack, as described in Figure 4.6, the attacker waits for a to initiate a session with b and s. The adversary then creates an agent with the name nb which he observed in message ②. This agent obtains a valid key $k(nb, s)$ and pretends that b has initiated a session with him by sending the message (a, b, na, nb) to s. In this message s interprets a as a nonce and nb as a name and responds with a newly generated key, $k(b, nb)$, for b and nb. Agent nb can decrypt the first part of the message to learn the key $k(b, nb)$. He then reverses the order of the two parts of the message and forwards them to b. Agent b decrypts $\{a, nb, k(b, nb)\}_{k(b,s)}$ and thinks that $k(b, nb)$ is the freshly generated key that he should use in his session with a. He then forwards the ticket $\{b, na, k(b, nb)\}_{k(nb,s)}$ together with a newly created nonce nb' to a. The adversary intercepts this message and respond to it by encrypting the nonce nb' with the key

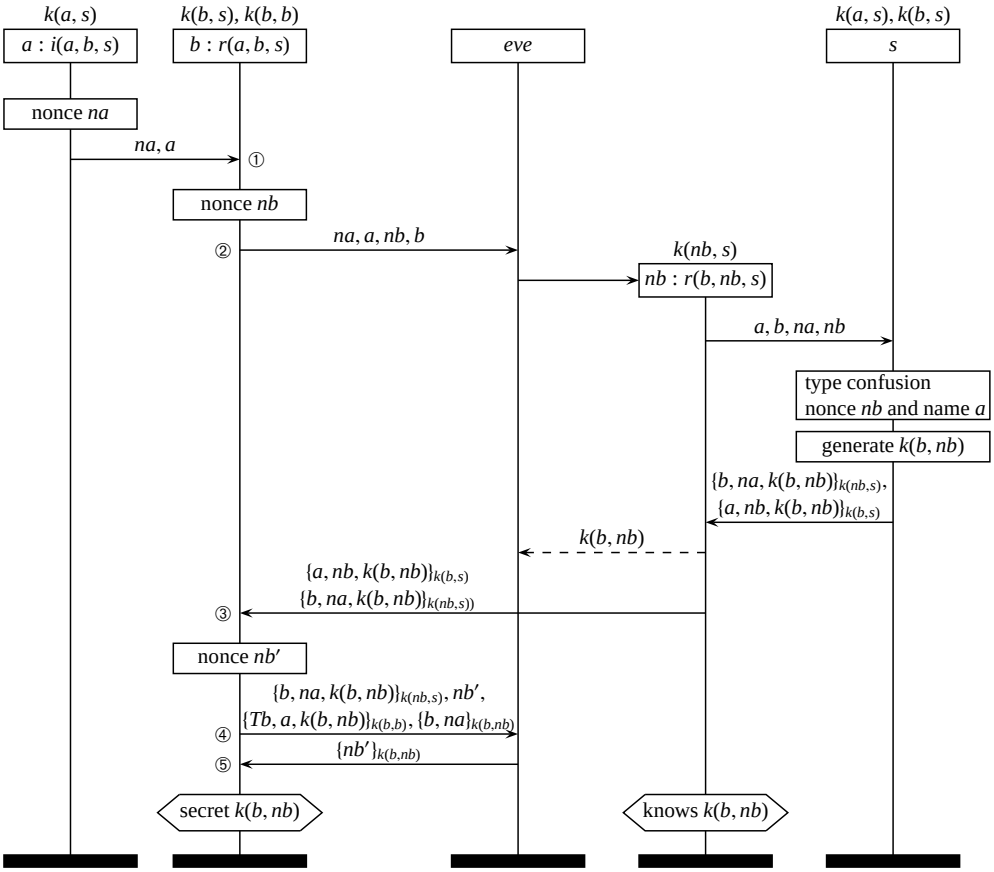

Figure 4.6: Attack on Lowe modified KSL

$k(b, nb)$ and impersonating a.

4.1.3 Assigned-name attacks

So far we have considered the adversary's ability to select the names of conspiring agents. In some settings, however, the adversary might even be able to assign names to honest agents. One example would be a compromised naming authority, another possibly more realistic example, would be a compromised DHCP server. In the latter scenario, a protocol which uses IP-addresses to identify agents could be vulnerable to an assigned-name attack.

Consider a variant of the Needham-Schroeder-Lowe (NSL) protocol where the nonces in the second message have been swapped as shown in Figure 4.7. The NSL protocol is a mutual authentication protocol that has originally been shown to be correct by Lowe [28] and since then by several other authors as well. The swapping of the two nonces has no influence on the correctness of the protocol even when conventional type flaws are taken into consideration. For simplicity, we are restricting ourselves to the secrecy claims of the protocol.

Figure 4.8 demonstrates an assigned-name attack on the NSL variant. An honest agent b starts a conversation with a malicious agent *eve* by sending $\{nb, b\}_{pk(eve)}$. The adversary then assigns the name (nb, e) to another honest agent. This honest agent starts a conversation with b and produces an encryption term of the form $\{nnbe, (nb, e)\}_{pk(b)}$. The conversation between the two honest agents continues and at the end of the protocol, (nb, e) and b agree on a secret value *nnbe*. The adversary takes the first message of this conversation and inserts it into the running session between b and *eve*. Agent b receives this message and confuses the name (nb, e) with nonce *nb* and name *eve* and responds with the message $\{nnbe\}_{pk(eve)}$ which enables the adversary to learn the value *nnbe*. Thus, the secrecy of *nnbe* claims of the honest agents (nb, e) and b are falsified by this attack.

This attack can be modified to impersonate b to *nb* and invalidate both secrecy claims of *nb* as follows. When (nb, e) sends out the first message of the protocol, the adversary can block the communication between the agents (nb, e) and b and inject the message $\{nnbe, (nb, e)\}_{pk(b)}$ into his run with b to learn *nnbe*. He then picks a nonce *ne* to construct the message $\{ne, nnbe, b\}_{pk((nb,e))}$. The adversary now knows both nonces and has furthermore impersonated b to *nb*. The security claims of b are not invalidated though, since b does not finish the protocol.

4.2 Related Work

The attacks we have described in this chapter belong to the intersection of two classes, namely chosen-name attacks and type-flaw attacks.

Chosen-name attacks have been known and described in the literature in various forms. For instance, it is known that in public key infrastructures a malicious or sloppy certificate authority would make it possible for an attacker to impersonate any user by registering under the user's name or a slight variation of the user's name. A particular instance of this attack, which is known as the *homograph* or *unicode* attack [18], is the registration of Internet domain names resembling well-known domain names. This attack became particularly popular when internationalized domain names became

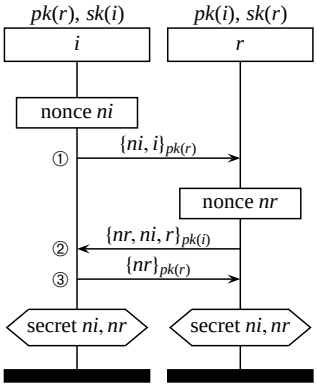

Figure 4.7: A variant of NSL

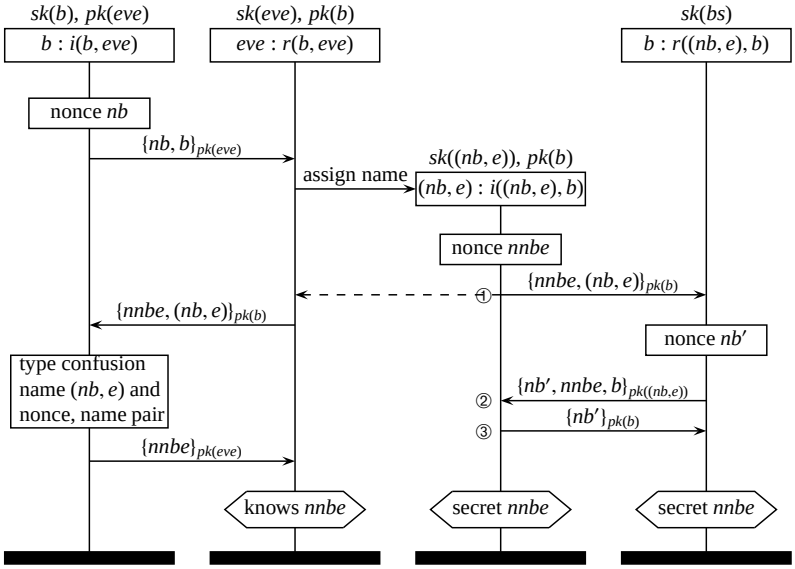

Figure 4.8: Assigned-name attack

available, since, for instance, several Cyrillic characters are identical to Latin characters allowing two distinct Internet domain names to have the same appearance.

A cryptographic impersonation attack, due to flawed key certification schemes, has been described by Lenstra and Yacobi [23]. In principle, such attacks can also be carried out on identity-based encryption schemes if the private key generation algorithm is weak. For instance, in Boneh and Franklin's scheme, it is easy to see that the possibility of a chosen-name attack hinges on the quality of the cryptographic hash function H_1 [9, Section 4.1].

Another source of chosen-name attacks are man-in-the-middle attacks on authentication protocols. A malicious agent seeking access to a resource would wait for an honest agent to initiate a vulnerable authentication protocol and consequently select the honest agent's name to perform the attack. In fact, the attack in Figure 4.2 is another example of a chosen-name man-in-the-middle attack. The attacker chooses and impersonates the agent b to obtain access to a. Similarly, in relay attacks, for instance on protocols running on radio frequency devices, a rogue device would forward the authentication challenge it receives to any victim it can find in the vicinity.

While all these attacks are well-known and have been extensively studied, they are different from the type-flaw attacks considered in this chapter, either in that they are not type-flaw attacks at all, or in that the chosen name is *static*.

Since the introduction of type flaws in security protocol analysis [10] various approaches have been used to detect and prevent type flaws. In [19] a tagging scheme is presented that prevents simple type flaws. Simple type flaws occur when one variable is unified with a complex term or a variable of another type.

More complex type-flaw attacks are described in [31]. These attacks emerge when tags are confused with terms or when parts of a term are confused with another term. The

detection of complex type flaws is formalized in [31, 32, 24, 25]. Research in this area focuses on the transitions from abstract message specification into concrete bit strings and vice versa.

Some of the formal frameworks aiming at verification of security protocols have included the concept of simple type-flaw attacks in their models, for instance [39, 2, 15]. We have investigated whether the tools based on these models, namely ProVerif [7], Scyther [14], the constraints solver in prolog [33], and the four tools of the Avispa project (CL-Atse[40], OFMC[6], SATMC [3], TA4SP [8]), are able to detect chosen-name attacks. These tools cover most of the modern techniques used in protocol verification, such as model checking, constraint solving, SAT-solving, and approximation. Since all of the selected tools provide a specification of the NSL protocol, only minor modifications were necessary to test the NSL variant in Figure 4.7. None of the selected tools were able to detect the selected-name attack described in Figure 4.8.

For Scyther and the Avispa tools it is easy to see why the attack could not be found. Scyther has a fixed domain out of which the names of agents are picked. The reason why none of the Avispa tools was able to find an attack is related to their input language, HLPSL. This language requires the user to define a set of concrete sessions under consideration. This set typically only contains sessions between agents with normal names. In order to find a chosen-name attack, one has to set up a session where the name of one of the agents is a concrete run term. Since the set of concrete run terms is infinite, it is not possible to list all potential chosen-name attack scenarios. This implies that for an Avispa tool using the HLPSL input language to find a chosen-name attack, the attack has to be known in advance. OFMC cannot find chosen-name type-flaw attacks, even in its native input language, due to an over-optimizing design choice in its symbolic session generation algorithm [6, §6.3].

We could not pinpoint the exact reason why the constraint solver in prolog did not find the assigned-name attack, as it seems that this formalism does not require a special domain for the names of honest agents. This formalism does however limit the names of the attacker, as a constant is used to represent his name, and thus precludes the detection of selected-name attacks. In ProVerif the default implementation of NSL uses the public key of an agent to identify the agent. Instead of sending $\{na, nb, b\}_{pk(a)}$ the second message is modeled by $\{na, nb, pk(b)\}_{pk(a)}$. Another way to model agent names in ProVerif is via the *host*() function, but even in that case, the attack could not be found.

Most formal models underlying tools for verification of security protocols can be extended to express chosen-name attacks. However, it will not necessarily be easy to extend the tools themselves. Especially tools that search through the state space of a given finite scenario will face the problem of having to choose appropriate agent names from an infinite domain.

4.3 Conclusion

In this chapter we have presented an intruder ability which was overlooked in common interpretations of the Dolev–Yao threat model and we demonstrated how this ability can be used to construct a special class of type-flaw attacks. We have identified a structure related to this intruder ability and classified the newly found attacks.

We have shown that Lowe's modified KSL protocol is vulnerable to a selected-name attack and that a mere reordering of two nonces renders the Needham-Schroeder-Lowe protocol vulnerable to an assigned-name attack.

Type-flaw attacks on a protocol are intimately related to the implementation of the protocol. The attacks presented in this chapter are infrequent, but as realistic as any other type-flaw attack and therefore should be taken into account by those tools and models which attempt to detect type flaws. Protocols vulnerable to this new class of attacks can be corrected like protocols vulnerable to type-flaw attacks by rearranging fields in messages, by adding extra information in vulnerable messages, such as was for instance done in messages ③ and ④ of the fictitious key-establishment protocol in Section 4.1.2, or by using tagging schemes such as those proposed in [19]. A way to prevent chosen-name type-flaw attacks in particular, is to precisely define the agent's name space and enforce strict name checking.

This work shows that the common Dolev–Yao interpretation is not complete with respect to the requirement that the adversary *tries everything he can* in order to learn a certain message. For instance, in [13] it is shown that from any attack on a secrecy claim involving n agents, an attack can be constructed which involves only two agents, assuming that agents may talk to themselves. The construction essentially maps all dishonest agents to one agent and all honest agents to the other agent. The attacks introduced in this chapter indicate that the security of a protocol can depend on the names of the agents. It is possible to construct protocols where an attack requires the adversary to select several names for dishonest agents. If one agent can only have one name, such an attack requires more than two agents. This shows that the results in [13] do not hold under the present intruder model. It is conceivable that there are other subtle assumptions made in the common interpretation of Dolev–Yao.

Chapter 5

Turing completeness of the framework

Formal modes of security protocols are only of interest if the protocols under inspection can express properties of a real world protocol. In Chapter 2 it is shown that this model can express interesting properties, illustrated by the attack on the Needham Schroeder protocol. Unfortunately the model has some limitations regarding the real world protocols it can handle. Certain cryptographic protocols require the use of an equational theory in their execution. Examples are protocols using the "exclusive or" operator or a Diffie–Hellman key negotiation [16]. The Diffie–Hellman protocol depends on the property that $g^{ab} = g^{ba}$.

The formal model used throughout this thesis has no support for equational theories, therefore there is a need for simulating these theories. There are other security models which incorporate the concept of an equational theory into their formal model, for instance OFMC [5]. In this chapter it is shown that an agent can perform any Turing computation which indicates that the framework can cope with any computable equational theory.

Proving Turing completeness is a non-trivial task, since there is no looping or recursion operator available in the role specification. However, the set of *Traces* of a given protocol might contain a trace with n executions of a certain role for any number n.

A Turing machine is an abstract computational device. The Church-Turing thesis asserts that any effective computation done on an abstract computer model (e.g. Random access machines) can also be accomplished by a Turing machine [20, p. 166]. We call a function $\mathcal{F}$ Turing computable if there is a Turing machine that computes it. The Church-Turing thesis asserts that all functions which can be expressed as an algorithm in an abstract computational device are Turing computable.

We call an arbitrary algebra, language or machine Turing complete, if everything that can be computed by a Turing machine can also be computed by the algebra, language or machine.

In this chapter we will try to answer the following question:

Is the presented framework Turing complete?

Turing completeness is usually discussed in the perspective of a computational system or a programming language. Its application to the field of security protocol analysis gives rise to the following questions:

What part of the security framework is Turing complete?

What does Turing completeness mean with respect to security protocols?

Can we use the Turing completeness results to simulate equational theory?

The goal for the Turing completeness we discuss here is to prove that honest agents are Turing complete. If we are able to construct a proof for this fact, we can conclude that any computation done by a Turing machine can be done by an agent. This agent must be able to execute the computation by itself, without interference of an adversary or another honest agent.

The Turing completeness of an agent should not be confused with the Turing completeness of the adversary as used in cryptography. In cryptography, systems are proven correct under assumption that the intruder is an arbitrary Turing machine; we assume perfect encryption, and thus limit the intruder in such a way that he can only decrypt a term when he knows the correct key.

We prove our framework Turing complete by showing that there is a way to map computational steps done by a Turing machine to related steps in the framework.

In Section 5.1 a Turing machine is described, Section 5.2 defines a mapping from Turing machines to protocols and roles. The exact proof obligation and a proof of correctness is given in 5.3, an example of the entire process if given in section 5.4. Section 5.6 describes the work related to the simulation of the Diffie–Hellman key exchange.

5.1 Turing machine

The Turing machine described below is based upon the Turing machine described in [20, p. 149].

A single-tape Turing machine is formally defined as the tuple $M = (Q, \Sigma, B, \ , q_0, F)$ where:

Q denotes a set of states

Γ denotes the set of tape symbols

B is a *blank symbol*; $B \in \Gamma$

Σ is a subset of Γ denoting the set of input symbols

denotes the *next move* transition function. $: Q \quad \Sigma \rightarrow Q \quad \Sigma \quad \{L, R\}$.

$q_0 \in Q$ is the initial state

$F \quad Q$ is the set of final states of the Turing machine

The computational state $C : \Sigma \quad Q \quad \Sigma$, of a Turing machine is defined as $C = (\ _1, q, \ _2)$, where:

$q \in Q$ denotes the active state

$_1 \in \Gamma$ is a list of elements on the left side of the reading head

$_2 \in \Gamma$ is a list of element on the right side of the reading head. The first element of $_2$ is the element under inspection of the reading head.

If $_2$ is a single symbol then the $_2$ will be extended with a blank symbol.

We define the move function of Turing machine TM as follows. Let $X_1, X_2....X_i, q, X_{i+1}, ...X_n$ be a computational state.

Suppose that $(q, X_i) = (p, Y, R)$ and $i = n + 1$ then $X_i = B$; this expresses the infinity of the tape on the right hand side. If $(q, X_i) = (p, Y, L)$ and $i = 1$ then there is no succeeding computational step. Thus the tape is limited on the left. if $i > 1$ then

$$X_1, X_2....X_{i-1}, q, X_i, X_{i+1}....X_n \vdash X_1, X_2....X_{i-2}, p, X_{i-1}, Y, X_{i+1}, ...X_n \qquad (5.1)$$

Alternatively if $(q, X_i) = (p, Y, R)$ then

$$X_1, X_2....X_{i-1}, q, X_i, X_{i+1}....X_n \vdash X_1, X_2....X_{i-1}, Y, p, X_{i+1}, X_{i+2}, ...X_n \qquad (5.2)$$

In case that $i = n + 1$ then X_{i+1} is taken to be B.

The symbol $\vdash$ denotes the repeated (0 or more) application of the move function. More formally:

Definition 5.1.1. $C \vdash C' \Leftrightarrow C = C' \vee C \vdash C_1 \vdash C_2 \vdash ....C_n \vdash C'$

where C_i denotes the computational state of a Turing machine after i steps.

Furthermore we define the state $C_0 = (\ , q_0, w)$ as the initial state when the Turing machine starts with word $w \in \Sigma$ on the tape. The accepted language for a Turing machine TM, denoted by $L(TM)$, is the set of all words $w \in \Sigma$ that let the Turing machine enter a final state.

Definition 5.1.2. *Let TM be a Turing machine,* $TM = (Q, \Gamma, B, \Sigma, \ , q_0, F)$
$L(TM) = \{w | w \in \Sigma \ , p \in F, \ _1, \ _2 \in \Sigma \ : q_0 w \vdash \ _1 p \ _2 \}$

At the moment a final step is reached, the Turing machine halts. There is no next move from a final state to another state. If a word is not accepted it is possible that a Turing machine never halts.

One can use a Turing machine to calculate a function by defining the machine in such a way that the value on the tape in a final position is a representation of the output values of the function.

5.2 Transformation and mapping

To prove Turing completeness of the security framework, we use two functions:

a function which relates the computational state of a Turing machine to role terms in the intruder knowledge.

a function Δ taking a Turing machine as input and producing a related protocol specification.

Our aim is to prove that

Theorem 5.2.1. *For the mappings* Δ, *and any computational state* C

$$C_0 \vdash C \Leftrightarrow \exists\ {}_{=1..\ n \in Tr(\Delta(TM))} \quad (C) \in M_n$$

If we can find the functions Δ and then the security framework can be used to simulate the execution of a computation by checking for all $q \in F$ whether there is a trace $= {}_1 .. \ {}_n$ such that $(1, q, {}_2) \in M_n$. Checking whether a certain role term occurs in the intruder knowledge can easily be done using the already existing secrecy claim.

The interpretation of a computational state of a Turing machine defined by the function is described in section 5.2.1. Section 5.2.2 defines the Turing machine transformation Δ.

5.2.1 State interpretation

The aim of the function is to relate a computational state to a role term. Recall that the goal of the simulation is that any agent A can perform the execution of a Turing machine. Assume that each agent A has a symmetric encryption key $K(A)$ which he does not share with anybody else.

Definition 5.2.1. *Let* $C = ({}_1, q, {}_2)$ *be a computational step of a Turing machine TM. The state interpretation function* $: \Sigma \quad Q \quad \Sigma \rightarrow$ *RoleTerm is defined by*

$$(1, q, {}_2) = \{ {}_1 {}^1, g_q, {}_1 \}_{K(A)}$$

where ${}_1 {}^1$ *denotes the reverse-ordered list of* ${}_1$, *and* g_q *is a global constant representing the state* q.

5.2.2 Turing machine Transformation

In this section a constructive definition of Δ is given. We will use $\Delta(TM)$ to denote the transformation of the Turing machine TM to the related protocol. We will use q to denote a state, X, Y as arbitrary length tuples of role terms and x and y to denote a single variable.

The intuition behind our Turing machine simulation is that we use roles to model state transitions; these roles read a value C and construct $'$ if $C \vdash C'$.

For each $q \in Q$ a global constant g_q is created.

For each $a \in \Sigma$ a global constant g_a is created.

For each element $(q, a) \in Q \quad \Sigma$ in the domain of , with index i, a role R_i is constructed as follows:

- if $(q, a) = (q\ a', L)$ then

$$Turing(R_i) = create_1(R_i) \quad read_2(R_i, R_i, \{(x, X), g_q, (g_a, Y)\}_{K(R_i)})$$
$$send_3(R_i, R_i, \{(X), g_{q'}, (x, g_{a'}, Y),\}_{K(R_i)})$$

- if $(q, a) = (q\ a', R)$ then

$$Turing(R_i) = create_1(R_i) \quad read_2(R_i, R_i, \{X, q, (a, Y)\}_{K(R_i)})$$
$$send_3(R_i, R_i, \{(a', X), q', Y\}\}_{K(R_i)})$$

By the fact that the Turing machine is deterministic it is evident that only one of these roles can be constructed for a given tuple (q, a).

Construct one role which models the infinity of the tape:

$$Turing(R_{inf}) = create_1(R_{inf}) \quad read_2(R_{inf}, R_{inf}, \{X, g_q, y\}_{K(R_{inf})})$$
$$send_3(R_{inf}, R_{inf}, \{X, g_q, (y, g_B)\}\}_{K(R_{inf})})$$

We construct a special role for the start and termination; let $w = w_1 w_2 .. w_n$ be the list of elements of the initial word of the Turing machine execution; we define $g_w = (g_{w_1}, gw_2, ..., g_{w_n})$

$$Turing(R_{init}) = create_1(R_{init}) \quad send_3(R_{init}, R_{init}, \{B, g_{q_0}, g_w\}_{K(R_{init})})$$
$$read_2(R_{init}, R_{init}, \{X, g_q, Y\}_{K(R_{init})})$$

and for each state $q_f \in F$ a claim is constructed:
$claim(R_{init}, secret, \{X, g_{q_f}, Y\}_{K(R_{init})})$

We use $\Delta(TM)$ to denote the protocol $P(R_{init}, R_{inf}, R_1, ...R_N)$

5.3 Proof of correctness

The functions Δ and defined above should fulfill the requirement stated in theorem 5.2.1; we prove this theorem in 2 directions.

Lemma 5.3.1. *For any honest agent A, computation state C, and trace* $\in Tr(\Delta(TM))$

$(C) \in M \Rightarrow \exists_{i<n} a_i = send_\ell(A, A, \quad (C))$

Proof. Observe that none of the roles constructed by $\Delta(TM)$ uses the key K as message content, as a consequence the adversary will not be able to learn the key of an honest agent. The adversary is not able to construct a term (C) himself, thus this term must have been sent by agent A at an event j. □

Theorem 5.3.1. *For the mappings* Δ, , *any computational state C and any honest agent A,*

$$\exists \; {}_{=1..\ n \in Tr(\Delta(TM))} \quad (C) \in M \Rightarrow C_0 \vdash C$$

Proof. Let $\in Tr(\Delta(TM))$ be a trace of length n such that $(C) \in M$

We use induction on the number of send events in the trace , and let $j(k)$ to denote the position of the kth send event in the trace.

The induction hypothesis is:
IH: $\forall_{k<K} \; {}_{j(k)} = send(A, A, \quad (C)) \Rightarrow C_0 \vdash C$

base: ${}_{j(1)} = send_\ell(A, A, \quad (C)) \Rightarrow C_0 \vdash C$;

${}_{j(1)}$ denotes the first send event of the trace. There is only one role which does not start with a read event. By construction of the roles, it is clear that these read events are not enabled when $M = M_0$. The only role which can add anything to M is the role R_{init}. R_{init} sends out a term $\{ {}_1{}^1, q_0 \; {}_2\}_{K(A,A)}$ which is exactly (C), where $C = ({}_1, q_0, \; {}_2)$; this C is the initial configuration C_0 of the Turing machine; by definition of $\vdash$, $C_0 \vdash C_0$.

Step: Assume the induction hypothesis holds for all $k < k'$. We have to show that ${}_{j(k')} = send_\ell(A, A, \quad (C)) \Rightarrow C_0 \vdash C$

If ${}_{j(k')}$ is not related to the protocol R_{init} then by construction of the roles, the send event k' is preceded by a read event, reading a value (C). By the fact that this send event occurs we are certain that there is an index i, $i < j(k')$ such that ${}_i$ is the read event preceding the send event. The fact that this ${}_i$ occurs implies that the read event was enabled, thus that $(C) \in M_i$. Combining this fact with lemma 5.3.1, the injectivity of , and the induction hypothesis implies that $C_0 \vdash C'$. By construction of the roles, it is clear that a role can only exist if there is a transition in such that $C \vdash C$; thus $C_0 \vdash C$. □

Theorem 5.3.2. *For the mappings* Δ, , *any computational state C and any honest agent A,*

$$C_0 \vdash C \Rightarrow \exists \; {}_{=1..\ n \in Tr(\Delta(TM))} \quad (C) \in M$$

Proof. Let A denote an honest agent, let C' be a computational state such that $C_0 \vdash C'$. By definition of $\vdash$ we obtain that $C_0 \vdash C_1 \vdash C_2 \vdash ....C_n \vdash C'$

The remainder of the proof will use induction based upon the length of this list of computations, using the following induction hypothesis:
IH: $\forall_{i<j} \; C_0 \vdash C_i \; \Rightarrow \exists \; {}_{=1..\ n \in Tr(\Delta(TM))} \quad (C) \in M_n$.

Base: There is a trace which start with a create event of role R_{init} followed by the send event of this role. This send event sends out $\{ {}_1{}^1, \; {}_2, q_0\}_{K(A,A)}$ which is exactly (C)

Step: let $C_0 \vdash C_1 \vdash C_2 \vdash ....C_j \; {}_1 \vdash C_j$ be a Turing computation and let be a trace of length n whith $(C_{j-1}) \in M$

Let $C_j \; {}_1 = ({}_1, q, \; {}_2)$ and assume that ${}_2 =$. The fact that ${}_2 =$ implies that ${}_2$ in computational state $C_j \; {}_2$contained exactly 1 element. The role R_{inf} was thus enabled

in computational state C_j $_2$ There is a trace in which the events of the role R_{inf} occur after send event sending out $(\mathcal{C}_2)$. The events of R_{inf} only add a blank to the end of $_2$ thus a read event reading $(\mathcal{C}_2)$ is still enabled and will produce $(\mathcal{C}_1)$ but now $_2 = B$. Thus if $_2 =$ then there is a trace in which the $\{_1{}^1, g_q, B\}_{K(A)}$ occurs, thus, in the role related to C_j $_1 \vdash C_j$ the read event is enabled. Therefore we can extend the trace with the events of this role such that $(\mathcal{G} \in M_n$.

If $_2 \neq$ then we know by construction of the roles that there is a role R which reads $(\mathcal{C}_1)$ and sends $(\mathcal{G})$. The trace is extended with the events from the role R. The induction hypothesis ensures that the read event of the role R is enabled. The send event of the role R produces a term $(\mathcal{G})$, thus $(\mathcal{G} \in M$. □

5.4 Example

As an example Turing machine we use a very simple machine which decides whether an inputstring is in the regular expression 1 on $\Sigma = \{1, 0, B\}$; this Turing machine is defined in figure 5.1. The input for the example is the string "11", which clearly is an element of the language.

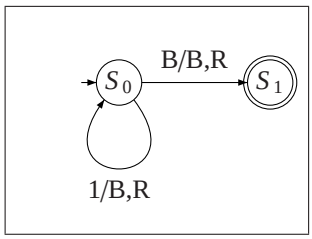

Figure 5.1: a simple Turing machine

If we apply the translation of a Turing machine to this protocol we end up with the protocol as described in appendix A

When evaluating this protocol Scyther immediately returns that the secrecy claim fails, thus there is a final state that is not secret; the Turing machine has terminated in that state. In figure 5.2 the trace which invalidates the security claim is shown, each run of a role relates to a transition, and the value of the tape in the final state can be seen in the claim event block. When running this protocol with an input that is not accepted by the Turing machine (e.g. "101") the secrecy claim will be valid.

5.5 Which part of the framework is Turing Complete

In the introduction of this chapter it was claimed that any honest agent should be able to perform a Turing computation by itself. It might seem that this requirement is not fulfilled since the intruder forwards the output of a run as input of another run.

If we would verify a Turing simulation in a setting with a eavesdropping adversary then the network would still act as a buffer. In such a setting it would be possible for

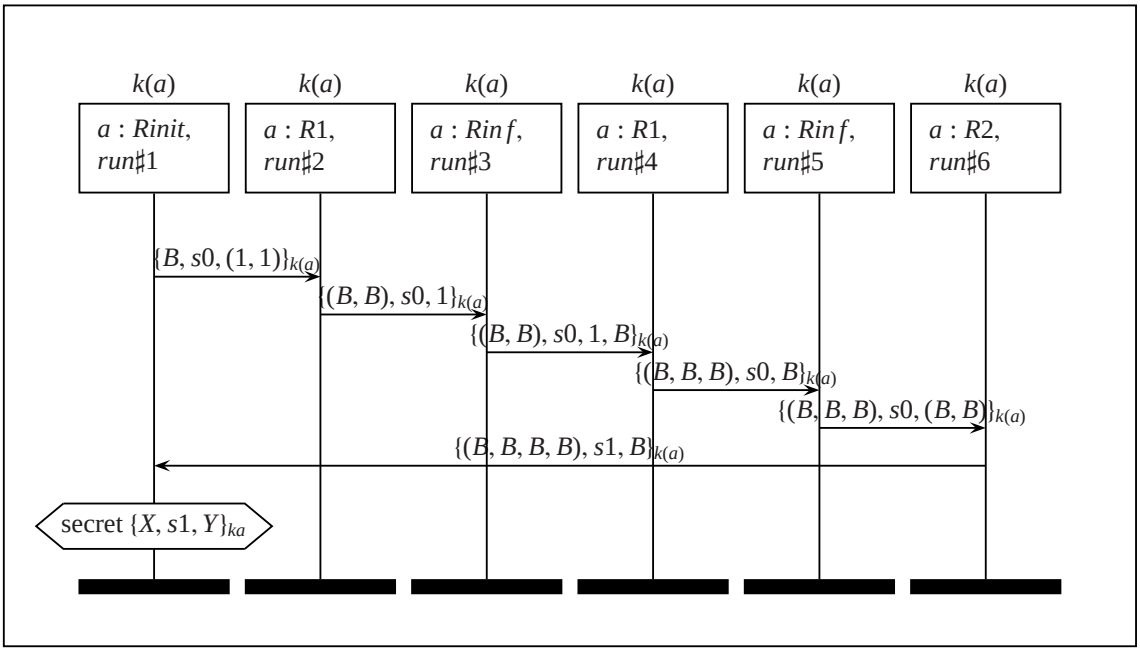

Figure 5.2: The trace invalidating the security claim

the agent to execute the Turing computation. Thus the intruder is not needed for a computation.

5.6 Simulation of equational theories

In the previous sections it is shown that an honest agent is Turing complete. This result is a first step towards the use of simulation of mathematical properties who cannot be expressed directly in the formal model.

In this section a first step is made in the simulation of the Diffie–Hellman key exchange protocol [16]. This protocol cannot be implemented in the formal model in a straightforward manner since it requires mathematical properties like associativity ($g^{ab} = g^{ba}$). The previously achieved Turing completeness result ensures us that this property can be expressed in the formal model.

As a starting point a list of basic requirements for the Diffie-Hellman simulation was constructed:

The key exchange protocol must work without interference of the adversary

There should be a relation between terms in the simulation and terms of the execution of the Diffie-Hellman key exchange protocol

No agent should be able to perform the decryption in the simulation which he could not do in a real Diffie-Hellman execution

The simulation should be reasonably efficient

These requirements express only some vital parts of the concept of simulation and this list is far from complete.

In Figure 5.3 one of the simulation attempts is illustrated. The idea underlying the protocol used in the simulation of Figure 5.3 is the introduction of a special agent N, the number theory agent, which executes all computations using the roles $N1$ and $N2$. An exponentiation is expressed by the encryption with the key $pk(n)$. To ensure the associativity requirement of Diffie-Hellman, agents do not compute the new key directly, instead they send out an intermediate result to the number theory agent. This number theory agent can execute either role $N1$ or $N2$ and in one of the roles the argument order is swapped. Thus there is always a trace where both honest agents receive the same term. The environment of this protocol ensures that only the imaginary agent N can obtain $pk(n)$ and thus the third requirement is not violated.

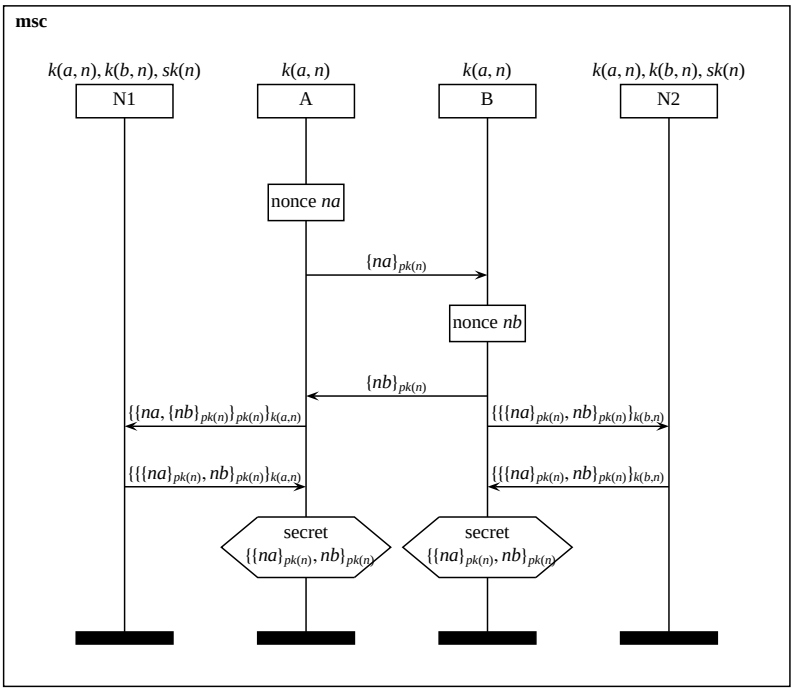

Figure 5.3: A trace of a Diffie–Hellman simulating protocol

Unfortunately it turned out that additional theory needs to be developed with respect to simulating an equational theory from a security perspective. One of the major problems encountered in this research direction was the question how we can exclude mathematical properties of a real execution in the simulation. The Diffie–Hellman key exchange is usually executed in a commutative group setting, should therefore all properties of such a group be modelled, or only the equation relevant to Diffie–Hellman? If not all properties are necessary, how can we prove that the properties modelled should be considered in the execution of Diffie–Hellman? A typical example of a property not included in the execution illustrated in figure 5.3 is the property that $g^{aeb} = g^{bea}$ which expresses that the adversary can exponentiate both public components with a term e

and both honest agent will still compute the same value for the key. Although this property cannot be used to break Diffie–Hellman directly it might be that the attained term can be used by the adversary to attack another protocol run.

Research in this area is still ongoing; as illustrated in the Diffie–Hellman example there are interesting challenges ahead and future research might lead to new insights in mixing the black-box cryptography assumption with certain real world mathematical properties of cryptographic schemes.

Chapter 6

Conclusions

In this thesis the research done in the area of formal analysis of security protocols is described. The research is focussed around the concept of "compositionality". Various directions are taken aiming at improving existing theoretical results.

An important component of the theory regarding compositionality are the exact constraints on the security protocols. Some of the strict requirements needed in the existing compositionality framework can be removed by the application of *unification theory*. The unification theory relaxed the requirements on the security protocols but it turned out that this was not sufficient to use the framework successfully in all cases. Research in the area of message encoding and the relation between abstract and concrete message specifications resulted in the concept of *unification preserving abstractions*. The results achieved in this area enable a broader application of the compositionality framework.

Another research direction considered the expressive power of the security protocols in the formal model. Equational theories are not supported by the formal model although they are of interest for numerous cryptographic applications. The *Turing completeness* proof shows that theoretically there is no need to extend the formal model with an equational theory. Besides the Turing completeness an attempt is made to model a protocol which simulates the Diffie–Hellman protocol.

Alongside the research related to compositionality and unification it emerged that there was a subtle assumption in the common interpretation of Dolev–Yao. Research in this area led to the discovery of *name attacks*, these attacks illustrate that verification results might be incorrect in certain environments. The work in this direction illustrates that there are minor differences between the scientific attacker models and an attacker in the real world.

6.1 Contributions

Improved definition of "strong independence" by introducing unification theory

Introduced "unification preserving abstraction" which enables remodelling of security protocols with improved compositionality properties

Detected a subtle missing intruder ability in the common Dolev–Yao interpretation

Constructed attacks related to the newly discovered intruder ability and classified these attacks

Proved Turing completeness of security protocols in the formal model, this is a first step towards simulation of equational theory (Diffie–Hellman)

The research related to the missing intruder ability, the related attacks and the attack classification have been published in [12].

6.2 Future work

Research hidden assumptions in formal models: In Chapter 4 a subtle hidden assumption is pointed out which is overseen in various formal models for security protocol evaluation. Future research in this direction might lead to new insights in formal attacker models, assumptions on protocol environments and the interpretation of protocol verification results.

Improve formal model and tool such that chosen name attacks can be detected: The name attacks introduced in this thesis were found by an exhaustive search of existing security protocols. Adjusting current models and tools such that they can cope with these attacks requires additional research and might even require a completely different strategy for formal security protocol analysis.

Research theory related to simulation: In this thesis it is shown that an agent can perform an arbitrary computation. Additional theory is needed on how to relate a real world cryptographic application to a protocol simulating parts of the behaviour of this application.

Appendix A

Protocol constructed for Turing completeness example

```
# Turing machine example
usertype Symbol;
usertype state;
secret const K : Function;

protocol Turing(R1,R2,Rinit, Rinf1, Rinf2)
{
   const B: Symbol;
   const 0: Symbol;
   const 1: Symbol;
   const s0,s1: state;

#the initializing role
 role Rinit
 {
    var X, Y:Ticket;
    var Q:state;
    send_Ri1(Rinit,Rinit,{B,(1,1),s0}K(Rinit,Rinit));
    read_!Ri2(Rinit,Rinit,{X,Y,Q}K(Rinit,Rinit));
    claim_Ri3(Rinit, Secret, {X,Y,s1}K(Rinit,Rinit));
  }

#  The roles for the transformations
  role R1
  {
    var X, Y:Ticket;
    read_!R11(R1,R1,{X,(1,Y),s0}K(R1,R1));
    send_R12(R1,R1,{(B,X),Y,s0}K(R1,R1));
  }

  role R2
```

```
  {
    var X, Y:Ticket;
    read_!R21(R2,R2,{X,(B,Y),s0}K(R2,R2));
    send_R22(R2,R2,{(B,X),Y,s1}K(R2,R2));
  }

# the roles for the infinity of the tape
    role Rinf1
    {
      var x: Symbol;
      var Y:Ticket;
      var Q:state;
      read_!inf11(Rinf1,Rinf1,{x,Y,Q}K(Rinf1,Rinf1));
      send_inf12(Rinf1,Rinf1,{(x,B),Y,Q}K(Rinf1,Rinf1) );
    }

    role Rinf2
    {
      var y: Symbol;
      var X:Ticket;
      var Q:state;
      read_!inf21(Rinf2,Rinf2,{X,y,Q}K(Rinf2,Rinf2));
      send_inf22(Rinf2,Rinf2,{X,(y,B),Q}K(Rinf2,Rinf2) );
    }
}

const Alice, Bob, Eve: Agent;
untrusted Eve;
```

Bibliography

[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*, 2007. To appear.

[2] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocol Analysis Tool. *Proceedings of CAV*, 2:349–354, 2002.

[3] A. Armando and L. Compagna. An Optimized Intruder Model for SAT-based Model-Checking of Security Protocols. *Proc. of ARSPA*, 2004, 2004.

[4] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[5] D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In *Proceedings of ESORICS*, pages 253–270. Springer, 2003.

[6] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[7] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[8] Y. Boichut, P.-C. Héam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay technique to automatically verify security protocols. In *Proc. Int. Ws. on Automated Verification of Infinite-State Systems (AVIS'2004), joint to ETAPS'04*, pages 1–11, Barcelona, Spain, April 2004.

[9] Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. *Lecture Notes in Computer Science*, 2139:213–229, 2001.

[10] C. Boyd. Hidden assumptions in cryptographic protocols. *Computers and Digital Techniques, IEE Proceedings-*, 137(6):433–436, 1990.

[11] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.

Reprinted from the Proceedings of the Royal Society, volume 426, number 1871, 1989.

[12] P. Ceelen, S. Mauw, and S. Radomirović. Chosen-name attacks: An overlooked class of type-flaw attacks. In *Proceedings of the 3rd International Workshop on Security and Trust Management*, to appear in ENTCS. Elsevier, 2007.

[13] H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. *Science of Computer Programming*, 50(1-3):51–71, 2004.

[14] Cas Cremers. *Scyther, Semantics and Verification of Security Protocols*. PhD thesis, Technische Universiteit Eindhoven, 2006.

[15] C.J.F. Cremers and S. Mauw. Operational semantics of security protocols. In S. Leue and T.J. Systä, editors, *Scenarios: Models, Algorithms and Tools (Dagstuhl 03371 post-seminar proceedings, September 7–12, 2003)*, volume 3466 of *LNCS*, pages 66–89, 2005.

[16] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[17] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

[18] Evgeniy Gabrilovich and Alex Gontmakher. The homograph attack. *Commun. ACM*, 45(2):128, 2002.

[19] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. *J. Comput. Secur.*, 11(2):217–244, 2003.

[20] J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory. *Languages, and Computation. Addison-Wesley*, 1979.

[21] X. ITU-T. 690: ITU-T Recommendation X. 690 (1997) Information technology-ASN. 1 encoding rules: Specification of Basic Encoding Rules (BER),Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). *Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*.

[22] Axel Kehne, Jürgen Schönwälder, and Horst Langendörfer. A nonce-based protocol for multiple authentications. *Operating Systems Review*, 26(4):84–89, 1992.

[23] Arjen K. Lenstra and Yacov Yacobi. User impersonation in key certification schemes. *J. Cryptology*, 6(4):225–232, 1993.

[24] Benjamin W. Long. Formal verification of type flaw attacks in security protocols. In *APSEC '03: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, page 415, Washington, DC, USA, 2003. IEEE Computer Society.

[25] B.W. Long, C.J. Fidge, and D.A. Carrington. Cross-layer verification of type flaw attacks on security protocols. In G. Dobbie, editor, *Proceedings of the 30th Australasian Computer Science Conference (ACSC 2007)*, pages 171–180, 2007.

[26] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[27] G. Lowe. Some new attacks upon security protocols. *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 162–169, 1996.

[28] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055, pages 147–166. Springer Verlag, 1996.

[29] LSV, ENS Cachan. Security Protocols Open Repository. `http://www.lsv.ens-cachan.fr/spore`.

[30] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[31] C. Meadows. Identifying potential type confusion in authenticated messages. In *Proceedings of Foundations of Computer Security*, 2002.

[32] C. Meadows. A procedure for verifying security against type confusion attacks. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003)*, pages 62–72, 2003.

[33] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 166–175, 2001.

[34] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[35] B.C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.

[36] IEEE 802.16 Working Group on Broadband Wireless Access. Ieee standard for local and metropolitan area networks, part 16: Air interface for fixed broadband wireless access systems, 2004.

[37] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.

[38] C. Rinderknecht. Proving a Soundness Property for the Joint Design of ASN. l and the Basic Encoding Rules. *System Analysis And Modeling: 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1-4, 2004: Revised Selected Papers*, 2005.

[39] F.J. Thayer Fàbrega, J.C. Herzog, and J.D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. 1998 IEEE Symposium on Security and Privacy*, pages 66–77, Oakland, California, 1998.

[40] M. Turuani. The CL-Atse Protocol Analyser. *17th international conference on term rewriting and applications-rta*, pages 277–286, 2006.