UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Depto. Sistemas Informáticos y Computación

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

MASTER THESIS

Protocol analysis modulo exclusive-or theories: a case study in Maude-NPA

CANDIDATE: Antonio González Burgueño

SUPERVISOR: Santiago Escobar Román

May 1, 2014

Departamento de Sistemas Informáticos y Computación

Universitat Politécnica de Valencia

Camino de Vera, s/n

46022 Valencia, Spain

Contents

Abstract

The development of this master thesis aims at verifying various existing security protocols using an advanced automated protocol verification tool, namely the Maude-NPA tool developed by Santiago Escobar (Universitat Politècnica de València) in collaboration with José Meseguer (University of Illinois at Urbana-Champaign, USA) and Catherine Meadows (Naval Research Laboratory, Washington, DC, USA). We focus on protocols using exclusive-or as the only cryptographic properties of symbols, apart of the standard cancellation of encryption and decryption. The protocols analyzed in this document are borrowed from the paper "Reducing Protocol Analysis with XOR to the XOR-Free Case in the Horn Theory Based Approach" by Ralf Küesters and Tomasz Truderung published in the Journal of Automated Reasoning, volume 46, pages 325-352, Springer 2011. These protocols are divided into two groups, those that can be specified in the Alice-Bob notation and those corresponding to an Application Programming Interface (API). We have proved the same security properties described in Küesters and Truderung paper, but we go beyond that paper in the sense that we have provided protocol specifications that meet all the requirements of the original protocols, whereas Küesters and Truderung paper uses simplified versions of these protocols without exclusive-or properties.

The main problem that we have encountered is to specify API protocols in Maude-NPA, since this was the first time that this kind of protocols were specified in the tool. A relevant contribution of this thesis is to confirm that complex protocols with exclusive-or can be verified in Maude-NPA.

Chapter 1

Introduction

The Maude-NPA is a tool and inference system for reasoning about the security of cryptographic protocols in which the cryptosystems satisfy different equational properties. The tool handles searches in the unbounded session model, and thus can be used to provide proofs of security as well as to search for attacks. It is the next generation of the NRL Protocol Analyzer [Meadows 96], a tool that supported limited equational reasoning and was successfully applied to the analysis of many different protocols.

The area of formal analysis of cryptographic protocols has been an active one since the mid 1980s. The idea is to verify protocols that use encryption to guarantee secrecy, and that use authentication of data to ensure security, against an attacker (commonly called the Dolev-Yao attacker [Dolev 83]) who has complete control of the network, and can intercept, alter, and redirect traffic, create new traffic on his/her own, perform all operations available to legitimate participants, and may have access to some subset of the longterm keys of legitimate principals. Whatever approach is taken, the use of formal methods has had a long history, not only for providing formal proofs of security, but also for uncovering bugs and security flaws that in some cases had remained unknown long after the original protocol's publication.

A number of approaches have been taken to the formal verification of cryptographic protocols. One of the most popular is model checking, in which the interaction of the protocol with the attacker is symbolically executed. Indeed, model-checking of secrecy (and later, authentication) in protocols in the bounded-session model (where a session is a single execution of a process representing an honest principal) has been shown to be decidable [Rusinowitch 01], and a num-

ber of bounded-session model checkers exist. Moreover, a number of unbounded model checkers either make use of abstraction to enforce decidability, or allow for the possibility of non-termination. The earliest protocol analysis tools, such as the Interrogator [Millen 87] and the NRL Protocol Analyzer (NPA) [Meadows 96] while not strictly speaking model checkers, relied on state exploration, and, in the case of NPA, could be used to verify security properties specified in a temporal logic language. Later, researchers used generic model checkers to analyze protocols, such as FDR [Lowe 96] and later Murphi [Mitchell 97]. More recently the focus has been on special-purpose model-checkers developed specifically for cryptographic protocol analysis, such as Blanchet's ProVerif [Proverif 10], the AVISPA tool [Armando 05], and Maude-NPA itself [Maude-NPA 09].

There are a number of possible approaches to take in the modeling of crypto algorithms used. In the simplest case, the free algebra model, crypto systems are assumed to behave like black boxes: an attacker knows nothing about encrypted data unless it has the appropriate key. This is the approach taken, for example, by the above-cited use of Murphi and FDR to analyze cryptographic protocols, and current tools such as SATMC [Armando 04] and TA4SP [Boichut 04], both used in the AVISPA tool. However, such an approach, although it can work well for protocols based on generic shared key or public key cryptography, runs into problems with algorithms such as Diffie-Hellman or algorithms employing exclusive-or, which rely upon various algebraic properties such as the law of exponentiation of products, associativity-commutativity and cancellation. Without the ability to specify these properties, one needs to rely on approximations of the algorithms that may result in formal proofs of secrecy invalidated by actual attacks that are missed by the analysis (see, e.g., [Paulson 98, Ryan 98, Stubblebine 00]). Thus there has been considerable interest in developing algorithms and tools for protocol analysis in the presence of algebraic theories [Abadi 06, Baudet 09, Bursuc 09, Chevalier 08, Ciobâca 09].

Another way in which tools can differ is in the number of sessions. A session is defined to be one execution of a protocol role by a single principal. A tool is said to use the bounded session model if the user must specify the maximum number of sessions that can be generated in a search. It is said to use the unbounded session model if no such restrictions are required. Secrecy is known to be decidable in the free theory together with the bounded session model [Rusinowitch 01], and undecidable in the free theory together with the unbounded session model [Durgin 04]. The same distinction between bounded

and unbounded sessions is known to hold for a number of different equational theories of interest, as well as for some authentication-related properties; see for example [Bursuc 09, Chevalier 08]. Thus, it is no surprise that most tools, whether or not they offer support for different algebraic theories, either operate in the bounded session model, or rely on abstractions that may result in reports of false attacks even when the protocol being analyzed is secure. Maude-NPA is a model-checker for cryptographic protocol analysis that both allows for the incorporation of different equational theories and operates in the unbounded session model without the use of abstraction. This means that the analysis is exact. That is, (i) if an attack exists using the specified algebraic properties, it will be found; (ii) no false attacks will be reported; and (iii) if the tool terminates without finding an attack, this provides a formal proof that the protocol is secure for that attack modulo the specified properties. However, it is always possible that the tool will not terminate. Maude-NPA is a backwards search tool, i.e., it searches backwards from a final insecure state to determine whether or not it is reachable from an initial state. This backwards search is symbolic, i.e., it does not start with a concrete attack state, but uses instead a symbolic attack pattern, i.e., a term with logical variables describing a general attack situation. The backwards search is then performed by backwards narrowing. Each backwards narrowing step denotes a state transition, such as a principal sending or receiving a message or the intruder manipulating a message, all in a backwards sense. Each backwards narrowing step takes a symbolic state (i.e., a term with logical variables) and returns a previous symbolic state in the protocol (again a term with logical variables). In performing a backwards narrowing step, the variables of the input term are appropriately instantiated in order to apply the concrete state transition, and the new previous state may contain new variables that are differentiated from any previously used variable to avoid confusion. To appropriately instantiate the input term, narrowing uses equational unification. As it is well-known from logic programming and automated deduction (see, e.g., [5]), unification is the process of solving equations $t = t'$. Standard unification solves these equations in a term algebra. Instead, equational unification (w.r.t. an equational theory E) solves an equation $t = t'$ in a free algebra for the equations E, i.e., modulo the equational theory E. In the Maude-NPA, the equational theory E used depends on the protocol, and corresponds to the algebraic properties of the cryptographic functions (e.g. cancellation of encryption and decryption, Diffie-Hellman, or exclusive-or).

In this thesis we focus in protocols with one exclusive-or operator. Exclusive-

or (XOR) is a binary operator with typical algebraic properties that has drawn a lot of interest. For example, XOR is often used in radio frequency identification (RFID) systems, which have become popular in recent years. The chosen exclusive-or operator (for example *)has the following four properties, where symbol 0 can also be chosen by the protocol user:

x * (y * z) = (x * y) * z : (associativity)
x * y = y * x : (commutativity)
x * 0 = x : (neutral element)
x * x = 0 : (nilpotence)

We work with different XOR protocols (see Section 4), and in order to detect attacks on XOR-protocols, we need to model not only the actions of the protocol but also the intruder actions to denote the ability of exploring the above algebraic properties, in addition to the perfect cryptography assumption.

In this thesis we also work with Application Program Interfaces (see Section 5). For this thesis we work with the IBM 4758 Common Cryptographic Architecture API (CCA-API), where XOR is used extensively. In an attack discovered by Bond [Bond 01] the self-inverse property of XOR can be exploited, together with some other coincidences in the API transaction set, to reveal a customer's PIN. However, the combinatorial possibilities caused by the associative, commutative and self-inverse properties of XOR pose a significant challenge to formal analysis. We show in Section 5 how the Maude-NPA tool works with these set of protocols (the original CCA-0 protocol, and the different variants due to the recommendations of IBM) .

In the following table we can see a summary of the results of the different protocols that we use in this thesis comparing our results with the results of the reference paper. In it we can see first the name of the protocol, then if it is secure according to the analysis of Küesters and Truderung and in our analysis in the next column. After that, we show if the tool reached a finite space of solutions or not, the number of states that the tool obtains and finally the depth of the search space. We should note that we tested two different versions of some protocols because we specify the original and a modified version of the protocol, instead of one only modified version of the original protocol as they did in the paper of reference because they can not handle the entire exclusive-or theory.

Protocol	**Secure**	**Our Analysis**	**Finite Space**	**States**	**Depth**
XOR-NSL	No	No	Yes	459	7
XOR-NSL-Fix	Yes	Yes	Yes	9	5
RA	No	No	Yes	120	8
RA-Fix	Yes	Yes	Yes	307	26
Shoup-Rubin	Yes	Yes	Yes	365	17
SK3	Yes	Yes	Yes	3	3
CCA-0	No	No	Yes	37*	6
CCA-0-Küesters	No	No	Yes	2495	6
CCA-1A	Yes	Yes	Yes	21*	5
CCA-1B	Yes	Yes	Yes	46*	6
CCA-1B-Küesters	Yes	Yes	Yes	1	2
CCA-2B	Yes	Yes	Yes	50*	11
CCA-2C	Yes	Yes	No	131*	7
CCA-2C-Küesters	Yes	Yes	No	105	4
CCA-2E	Yes	Yes	No	385*	7

* This protocol analysis includes Never Patterns

Table 1.1: Experimental results.

Chapter 2

Background on Term Rewriting

We follow the classical notation and terminology from [TeReSe 03] for term rewriting and from [Meseguer 92, Meseguer 98] for rewriting logic and order-sorted notions. We assume an *order-sorted signature* Σ with a finite poset of sorts $(\mathtt{S}, \leq)$ and a finite number of function symbols. We assume an $\mathtt{S}$-sorted family $\mathcal{X} = \{\mathcal{X}_\mathtt{s}\}_{\mathtt{s}\in\mathtt{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathtt{s}$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_\mathtt{s}$ denotes the set of terms of sort $\mathtt{s}$, and $\mathcal{T}_{\Sigma,\mathtt{s}}$ the set of ground terms of sort $\mathtt{s}$. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding term algebras. We write $\mathcal{V}ar(t)$ for the set of variables present in a term t. The set of positions of a term t is written $Pos(t)$, and the set of non-variable positions $Pos_\Sigma(t)$. The subterm of t at position p is $t|_p$, and $t[u]_p$ is the result of replacing $t|_p$ by u in t.

A *substitution* σ is a sort-preserving mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. Application of substitution σ to term t is denoted by $t\sigma$. The restriction of σ to a set of variables V is $\sigma|_V$. The composition of two substitutions is $X(\sigma \circ \theta) = (X\theta)\sigma$ for $X \in \mathcal{X}$.

A Σ-*equation* is an unoriented pair $t = t'$, where $t \in \mathcal{T}_\Sigma(\mathcal{X})_\mathtt{s}$, $t' \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathtt{s}'}$, and s and s' are sorts in the same connected component of the poset $(\mathtt{S}, \leq)$. Given a set E of Σ-equations, order-sorted equational logic induces a congruence relation $=_E$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$; see [Meseguer 98]. Throughout this paper we assume that $\mathcal{T}_{\Sigma,\mathtt{s}} \neq \emptyset$ for every sort $\mathtt{s}$. We denote the E-equivalence class of a term $t \in \mathcal{T}_\Sigma(\mathcal{X})$ as $[t]_E$ and the E-equivalence classes of all terms $\mathcal{T}_\Sigma(\mathcal{X})$ and

$\mathcal{T}_\Sigma(\mathcal{X})_\mathtt{s}$ as $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ and $\mathcal{T}_{\Sigma/E}(\mathcal{X})_\mathtt{s}$ for a sort $\mathtt{s}$, respectively.

For a set E of Σ-equations, an *E-unifier* for a Σ-equation $t = t'$ is a substitution σ s.t. $t\sigma =_E t'\sigma$. A *complete* set of E-unifiers of an equation $t = t'$ is written $CSU_E(t = t')$. We say $CSU_E(t = t')$ is *finitary* if it contains a finite number of E-unifiers.

A *rewrite rule* is an oriented pair $l \to r$, where $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_\mathtt{s}$ for some sort $\mathtt{s} \in \mathtt{S}$. An *(unconditional) order-sorted rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with Σ an order-sorted signature, E a set of Σ-equations, and R a set of rewrite rules. A *topmost rewrite theory* (Σ, E, R) is a rewrite theory s.t. for each $l \to r \in R$, $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{State}}$ for a top sort $\mathtt{State}$, $r \notin \mathcal{X}$, and no operator in Σ has $\mathtt{State}$ as an argument sort.

The rewriting relation $\to_R$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \overset{p}{\to}_R t'$ (or $\to_R$) if $p \in Pos_\Sigma(t)$, $l \to r \in R$, $t|_p = l\sigma$, and $t' = t[r\sigma]_p$ for some σ. The relation $\to_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_E ; \to_R ; =_E$, i.e., $t \to_{R/E} s$ iff $\exists u_1, u_2 \in \mathcal{T}_\Sigma(\mathcal{X})$ s.t. $t =_E u_1 \to_R u_2 =_E s$. Note that $\to_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ induces a relation $\to_{R/E}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ by $[t]_E \to_{R/E} [t']_E$ iff $t \to_{R/E} t'$.

When $\mathcal{R} = (\Sigma, E, R)$ is a topmost rewrite theory, we can safely restrict ourselves to the rewriting relation $\to_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$, where the rewriting relation $\to_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \overset{p}{\to}_{R,E} t'$ (or $\to_{R,E}$) if $p \in Pos_\Sigma(t)$, $l \to r \in R$, $t|_p =_E l\sigma$, and $t' = t[r\sigma]_p$ for some σ. Note that $\to_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ induces a relation $\to_{R,E}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ by $[t]_E \to_{R,E} [t']_E$ iff $\exists w \in \mathcal{T}_\Sigma(\mathcal{X})$ s.t. $t \to_{R,E} w$ and $w =_E t'$.

The narrowing relation $\rightsquigarrow_R$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \overset{p}{\rightsquigarrow}_{\sigma,R} t'$ (or $\rightsquigarrow_{\sigma,R}$, $\rightsquigarrow_R$) if $p \in Pos_\Sigma(t)$, $l \to r \in R$, $\sigma \in CSU_\emptyset(t|_p = l)$, and $t' = \sigma(t[r]_p)$. Assuming that E has a finitary and complete unification algorithm, the narrowing relation $\rightsquigarrow_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \overset{p}{\rightsquigarrow}_{\sigma,R,E} t'$ (or $\rightsquigarrow_{\sigma,R,E}$, $\rightsquigarrow_{R,E}$) if $p \in Pos_\Sigma(t)$, $l \to r \in R$, $\sigma \in CSU_E(t|_p = l)$, and $t' = \sigma(t[r]_p)$. In this thesis we consider only equational theories $E = E' \uplus Ax$ such that the oriented equations E' are confluent, coherent, and terminating modulo axioms Ax such as commutativity (C), associativity-commutativity (AC), or associativity-commutativity plus identity (ACU) of some function symbols. We also require axioms Ax to be regular, i.e., for each equation $l = r \in Ax$, $\mathcal{V}ar(l) = \mathcal{V}ar(r)$. Note that axioms such as commutativity (C), associativity-commutativity (AC), or associativity-commutativity plus identity (ACU) are regular. The Maude-NPA has then both dedicated and generic algorithms for solving unification problems in such theories $E' \uplus Ax$ under appropriate conditions [Escobar 12].

The use of topmost rewrite theories provides several advantages; see [Thati 07]:

(i) as pointed out above the relation $\rightarrow_{R,E}$ achieves the same effect as the relation $\rightarrow_{R/E}$, and (ii) we obtain a completeness result between narrowing ($\rightsquigarrow_{R,E}$) and rewriting ($\rightarrow_{R/E}$), in the sense that a reachability problem has a solution iff narrowing can find an instance of it.

Chapter 3

Maude-NPA

In this Chapter we briefly present the Maude-NPA cryptographic analyzer tool. In Chapter 3.1 we present the Maude-NPA's execution model. After that, in Chapter 3.2, we show the syntax for protocol specifications in Maude-NPA's. In Chapter 3.3 we show how to specify the protocol syntax. Next, in Chapter 3.4 we show how to specify the algebraic properties of the protocol. Also, in Chapter 3.5 we show how to specify the protocol strands. And finally, in Chapter 3.6 we show how is performed a protocol analysis.

3.1 Maude-NPA's Execution Model

Given a protocol $\mathcal{P}$, a *state* in the protocol execution is an $E_{\mathcal{P}}$-equivalence class $[t]_{E_{\mathcal{P}}}$ with t a term of sort `State`, $[t]_{E_{\mathcal{P}}} \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}(X)_{\texttt{State}}$, where $\Sigma_{\mathcal{P}}$ is the signature defining the sorts and function symbols for the cryptographic functions and for all the state constructor symbols, and $E_{\mathcal{P}}$ is a set of equations specifying the *algebraic properties* of the cryptographic functions and the state constructors. A protocol $\mathcal{P}$ is specified with a notation derived from strand spaces. In a *strand*, a local execution of a protocol by a principal is indicated by a sequence of messages $[msg_1^-,\ msg_2^+,\ msg_3^-, \ldots,\ msg_{k-1}^-,\ msg_k^+]$ where each msg_i is a term of sort Msg (i.e., $msg_i \in T_{\Sigma_{\mathcal{P}}}(X)_{\mathsf{Msg}}$). Strand items representing input messages are assigned a negative sign, and strand items representing output messages are assigned a positive sign. We write $m^{\pm}$ to denote m^+ or m^-, indistinctively. We often write $+(m)$ and $-(m)$ instead of m^+ and m^-, respectively. For each positive message msg_i in a sequence of messages $[msg_1^{\pm},\ msg_2^{\pm},\ msg_3^{\pm}, \ldots,\ msg_i^+,$

$\ldots,\ msg_{k-1}^{\pm},\ msg_k^{\pm}]$ the non-fresh variables (see below) occurring in an output message msg_i^+ must appear in previous messages $msg_1, msg_2, msg_3, \ldots, msg_{i-1}$.

In Maude-NPA [Escobar 06, Escobar 09c], strands evolve over time and thus we use the symbol | to divide past and future in a strand, i.e., $[nil, msg_1^{\pm}, \ldots, msg_{j-1}^{\pm} \mid msg_j^{\pm}, msg_{j+1}^{\pm}, \ldots, msg_k^{\pm}, nil]$, where $msg_1^{\pm}, \ldots, msg_{j-1}^{\pm}$ are the past messages, and $msg_j^{\pm}, msg_{j+1}^{\pm}, \ldots, msg_k^{\pm}$ are the future messages ($msg_j^{\pm}$ is the immediate future message). We often remove the nils for clarity, except when there is nothing else between the vertical bar and the beginning or end of a strand. We write $\mathcal{P}$ for the set of strands in a protocol, including the strands that describe the intruder's behavior.

Maude-NPA uses a special sort `Msg` of messages that allows the protocol specifier to describe other sorts as subsorts of the top sort `Msg`. The specifier can make use of a special sort `Fresh` in the protocol-specific signature Σ for representing fresh unguessable values, e.g., nonces. We make explicit the `Fresh` variables $r_1, \ldots, r_k \quad (k \geq 0)$ generated by a strand by writing $:: r_1, \ldots, r_k :: [msg_1^{\pm}, \ldots, msg_n^{\pm}]$, where $r_1, \ldots, r_k$ appear somewhere in $msg_1^{\pm}, \ldots, msg_n^{\pm}$. Fresh variables generated by a strand are unique to that strand.

To illustrate this we use the well-known Needham-Schroeder-Lowe (NSL) protocol [Lowe 96]. We reproduce the NSL protocol in the following.

1. $A \rightarrow B : \{N_A, A\}_{pk(B)}$
2. $B \rightarrow A : \{N_A, N_B, B\}_{pk(A)}$
3. $A \rightarrow B : \{N_B\}_{pk(B)}$

where $\{M\}_{pk(A)}$ means message M encrypted using the public key of principal with name A, N_A and N_B are nonces generated by the respective principals, and we use the comma as message concatenation.

The specification of the strands of the participants is as follows:

$:: \text{r} :: [(pk(B, n(A, r); A))^+, (pk(A, n(A, r); NB; B))^-, (pk(B, NB))^+]$
$:: \text{r'} :: [(pk(B, NA; A))^-, (pk(A, NA; n(B, r'); B))^+, (pk(B, n(B, r')))^-]$

A *state* is a set of Maude-NPA strands unioned together by an associative and commutativity union operator `_&_` with identity operator $\emptyset$, along with an additional term describing the intruder knowledge at that point. The *intruder knowledge* is represented as a set of facts unioned together with an associative and commutativity union operator `_,_` with identity operator $\emptyset$. There are two

kinds of intruder facts: *positive* knowledge facts (the intruder knows message m, i.e., $m{\in}\mathcal{I}$), and *negative* knowledge facts (the intruder does not yet know m but will know it in a future state, denoted by $m{\notin}\mathcal{I}$).

When new strands are not introduced into the state, the rewrite rules $R_{\mathcal{P}}$ obtained from the protocol strands $\mathcal{P}$ are as follows[1], where L, L' are variables of the sort for lists of input and output messages ($+m$,$-m$), IK is a variable of the sort for sets of intruder facts ($m{\in}\mathcal{I}$,$m{\notin}\mathcal{I}$), SS is a variable of the sort for sets of strands, and M is a variable of sort `Msg`:

$$SS \;\&\; [L \mid M^-, L'] \;\&\; (M{\in}\mathcal{I}, IK) \rightarrow SS \;\&\; [L, M^- \mid L'] \;\&\; (M{\in}\mathcal{I}, IK) \tag{3.1}$$

$$SS \;\&\; [L \mid M^+, L'] \;\&\; IK \rightarrow SS \;\&\; [L, M^+ \mid L'] \;\&\; IK \tag{3.2}$$

$$SS \;\&\; [L \mid M^+, L'] \;\&\; (M{\notin}\mathcal{I}, IK) \rightarrow SS \;\&\; [L, M^+ \mid L'] \;\&\; (M{\in}\mathcal{I}, IK) \tag{3.3}$$

In a *forward execution* of the protocol strands, Rule (3.1) synchronizes an input message with a message already in the channel (i.e., learned by the intruder), Rule (3.2) accepts output messages but the intruder's knowledge is not increased, and Rule (3.3) accepts output messages and the intruder's knowledge is positively increased. Note that Rule (3.3) makes explicit *when* the intruder learned a message M, which is recorded in the previous state by the negative fact $M{\notin}\mathcal{I}$. A fact $M{\notin}\mathcal{I}$ can be paraphrased as: "the intruder does not yet know M, but will learn it in the future".

New strands are added to the state by explicit introduction through dedicated rewrite rules (one for each honest or intruder strand). It is also the case that when we are performing a backwards search, only the strands that we are searching for are listed explicitly, and extra strands necessary to reach an initial state are dynamically added. Thus, when we want to introduce new strands into the explicit description of the state, we need to describe additional rules for doing that, as follows:

$$\text{for each } [\; l_1,\; u^+,\; l_2 \;] \in \mathcal{P} : SS \;\&\; [\; l_1 \mid u^+, l_2 \;] \,\&\, (u{\notin}\mathcal{I}, IK) \rightarrow SS \;\&\; (u{\in}\mathcal{I}, IK) \tag{3.4}$$

where u denotes a message, l_1, l_2 denote lists of input and output messages ($+m$,$-m$), IK denotes a variable of the sort for sets of intruder facts ($m{\in}\mathcal{I}$,$m{\notin}\mathcal{I}$),

[1] To simplify the exposition we omit the fresh variables at the beginning of each strand in a rewrite rule.

and SS denotes a variable of the sort for sets of strands. For example, intruder concatenation of two learned messages is described as follows:

$$SS \;\&\; [M_1^-, M_2^- \mid (M_1; M_2)^+] \;\&\; ((M_1; M_2) \notin \mathcal{I}, IK) \rightarrow SS \;\&\; ((M_1; M_2) \in \mathcal{I}, IK)$$

In summary, for a protocol $\mathcal{P}$, the set of rewrite rules obtained from the protocol strands that are used for backwards narrowing reachability analysis *modulo* the equational properties $E_{\mathcal{P}}$ is $R_{\mathcal{P}} = \{(3.1), (3.2), (3.3)\} \cup (3.4)$.

3.2 Syntax for Protocol Specification

In this Chapter, we briefly describe how to specify a protocol and all its relevant items in the current version of the Maude-NPA. For further information we refer the reader to [Escobar 09b]. Note that, since we are using Maude also as the specification language, each declaration has to be ended by a space and a period.

3.2.1 Specifying the Protocol Syntax

We begin by specifying sorts. In general, sorts are used to specify different types of data, that are used for different purposes. The Maude-NPA tool always assumes that the sort `Msg` is the top sort, but it allows user-defined subsorts of `Msg` that can be specified by the user for a more accurate protocol specification and analysis. To illustrate the definition of sorts, we use the Needham-Schroeder-Lowe (NSL) [Lowe 96]. For this protocol we need to define sorts to distinguish names, nonces and encrypted data. This is specified as follows:

```
sorts Name Nonce Enc .
subsort Name Nonce Enc  <  Msg .
subsort Name  <  Public .
```

We can now specify the different operators needed in NSL. A nonce generated by principal A is denoted by $n(A, r)$, where r is a unique variable of sort `Fresh`. Concatenation of two messages, e.g., N_A and N_B, is denoted by the operator $\_;\_$, e.g., $n(A, r); n(B, r')$. Encryption of a message M with the public key K_A of principal A is denoted by $pk(A, M)$, e.g., $\{N_B\}_{pk(B)}$ is denoted by $pk(B, n(B, r'))$. Encryption with a secret key is denoted by $sk(A, M)$. We begin with the public/private encryption operators.

```
op pk : Name Msg -> Enc .
op sk : Name Msg -> Enc .
```

Next we specify some principal names. For NSL, we have three constants of sort `Name`, a (for Alice), b (for Bob) and i (for the Intruder). We need two more operators, one for nonces and one for concatenation. The nonce operator is specified as follows.

```
op n : Name Fresh -> Nonce .
```

Note that the nonce operator has an argument of sort `Fresh` to ensure uniqueness. The argument of type Name is not strictly necessary, but it provides a convenient way of identifying which nonces are generated by which principal. This makes searches more efficient, since it allows us to keep track of the originator of a nonce throughout a search. Finally, we come to the message concatenation operator. In Maude-NPA, we specify concatenation via an infix operator _;_ defined as follows:

```
op _ ; _ : Msg  Msg  -> Msg [gather (e E)] .
```

Note that the gathering pattern of an operator [Maude 09] restricts the precedences of terms that are allowed as arguments. We give a (non-empty) sequence of as many `E`, `e`, or `&` values as the number of arguments in the operator, that is, one of these values for each argument position:

- `E` indicates that the argument must have a precedence value lower or equal than the precedence value of the operator,
- `e` indicates that the argument must have a precedence value strictly lower than the precedence value of the operatorand
- `&` indicates that the operator allows any precedence value for the corresponding argument. In fact, the precedence values work because of their combination with the gathering patterns.

3.2.2 Algebraic Properties

Next, we specify the algebraic properties of the symbols defined above for the NSL protocol. There are three types of algebraic properties in Maude-NPA: (i) *equational axioms*, such as commutativity, or associativity-commutativity,

called axiomsand (ii) *equational rules*, called *variant equations*, and (iii) *equational rules* for dedicated unification algorithms, called *dedicated equations* . *Variant* and *dedicated equations* are specified in the *PROTOCOL-EXAMPLE-ALGEBRAIC* module, whereas *axioms* are specified within the operator declarations in the *PROTOCOL-EXAMPLE-SYMBOLS* module, as illustrated in what follows. Note that combinations of all three different types of a*lgebraic properties* will be available in the future but in the current version only *axioms* and *variant equations* can be combined.

An equation is oriented into a rewrite rule in which the lefthand side of the equation is *reduced* to the righthand side. In NSL, we use two equations specifying the relationship between public and private key encryption, as follows:

```
var X : Msg . var A : Name .
eq pk(A,sk(A,X)) = X [variant] .
eq sk(A,pk(A,X)) = X [variant] .
```

Note that there are restrictions on the equations that can be included here, since the narrowing-based unification algorithm provided by the tool for those equations must be finitary, see [Escobar 09b]. For instance, the algebraic properties of exclusive-or symbol are specified as follows:

```
eq X * X * Y = Y [variant] .
eq X * X = null  [variant] .
eq X * null = X  [variant] .
```

Note that the redundant equational property X * X * Y = Y is necessary in Maude-NPA for coherence purposes; see [Escobar 09a].

3.2.3 Specifying the Strands

As explained in previous sections, the protocol itself and the intruder capabilities are both specified using strands. We use the keyword `STRANDS-PROTOCOL` for storing the principal strands. For example, the strands associated to the NSL protocol are specified as follows:

```
eq STRANDS-PROTOCOL =
  :: r ::
   [nil | +(pk(B,n(A,r) ; A)),
          -(pk(A,n(A,r) ; NB ; B)),
```

```
          +(pk(B,NB)), nil]
   &
  :: r ::
    [nil | -(pk(B,NA ; A)),
           +(pk(A,NA ; n(B,r) ; B)),
           -(pk(B,n(B,r))), nil] .
 [nonexec] .
```

The next thing to specify is the actions of the intruder, or Dolev-Yao rules [Dolev 83]. These specify the operations an intruder can perform. Each such action can be specified by an intruder strand consisting of a sequence of negative nodes, followed by a single positive node. If the intruder can (nondeterministically) find more than one term as a result of performing one operation (as in deconcatenation), we specify each of these outcomes by separate strands. Every operation that can be performed by the intruderand every term that is initially known by the intruder, should have a corresponding intruder strand. For each operation used in the protocol we should consider whether or not the intruder can perform itand specify a corresponding intruder strand that describes the conditions under which the intruder can perform it. For the NSL protocol, we have four operations: encryption with a public key (pk), decryption with a private key (sk), concatenation (_ ; _) and deconcatenation. We use the keyword `STRANDS-DOLEVYAO` for storing the intruder strands:

```
eq STRANDS-DOLEVYAO
  = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil] &
    :: nil :: [ nil | -(X ; Y), +(X), nil] &
    :: nil :: [ nil | -(X ; Y), +(Y), nil] &
    :: nil :: [ nil | -(X), +(sk(i,X)), nil] &
    :: nil :: [ nil | -(X), +(pk(Ke,X)), nil]
  [nonexec] .
```

3.3 Protocol Analysis

Next, we describe how to analyze a protocol in practice. First, we explain how a protocol state looks like and how an attack state is specified in the protocol. Then, we explain how the protocol analysis is performed.

In Maude-NPA, each state associated to the protocol execution (i.e., a backwards search) is represented by a term with five different components separated

by the symbol || in the following order: (1) the set of current strands, (2) the current intruder knowledge, (3) the sequence of messages encountered so far in the backwards execution, (4) some auxiliary data and (5) the never pattern, a technique to reduce the search space, associated to that state, i.e., a term of the following form.

Strands || Intruder Knowledge || Message Sequence || Auxiliary Data || Never Pattern.

The first component, the set of current strands, indicates in particular how advanced each strand is in the execution process (by the placement of the bar). The second component contains messages that the intruder already knows (we use symbol _inI for the notation m$\in \mathcal{I}$)) and messages that the intruder currently doesnt know (we use symbol _inI for the notation m$\notin \mathcal{I}$)) but will learn in the future. The third and fourth components are irrelevant for the purposes of this work, see [Escobar 09b]. The fifth component is used to specify negative conditions on terms or strands. Never patterns can also be used to cut down the search space (see [Escobar 09c] for more details).

An initial state is the final result of the backwards reachability process and is described as follows:

1. in an initial state, all strands have the bar at the beginning, i.e., all strands are of the form :: r_1, .., r_j :: $[nil \mid m_1^{\pm}, \, .. \, m_k^{\pm}]$;

2. in an initial state, all the intruder knowledge is negative, i.e., all the items in the intruder knowledge are of the form m$\notin \mathcal{I}$.

From an initial state, no further backwards reachability steps are possible. Attack states describe not just single concrete attacks, but *attack patterns* (or if you prefer *attack situations*), which are specified symbolically as terms (with variables) whose instances are the final attack states we are looking for. Given an attack pattern, Maude-NPA tries to either find an instance of the attack or prove that no instance of such attack pattern is possible. We can specify more than one attack state. Thus, we designate each attack state with a natural number.

When specifying an attack state, the user should specify only the first two components of the attack state: (i) a set of strands expected to appear in the attack and (ii) some positive intruder knowledge. The message sequence, auxiliary data components and never pattern should have just the empty symbol

`nil`. Note that the attack state is indeed a term with variables but the user does not have to provide the variables denoting the remaining strands, the remaining intruder knowledge and the two variables for the two last state components. These variables are symbolically inserted by the tool.

For example, to prove that the NSL protocol fixes the bug found in the Needham-Schroeder Public Key protocol (NSPK), i.e., the intruder cannot learn the nonce generated by Bob, we should specify the following attack state:

```
eq ATTACK-STATE(0) =
   :: r ::
    [nil, -(pk(b,a ; NA)),
          +(pk(a,NA ; n(b,r) ; b)),
          -(pk(b,n(b,r))) | nil]
    || n(b,r) inI
    || nil
    || nil
    || nil .
[nonexec] .
```

And the search space associated to this protocol is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 4 Solutions>> 0
reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 7 Solutions>> 0
reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 6 Solutions>> 0
reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 2 Solutions>> 0
reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 0 Solutions>> 0
```

which cannot reach an initial state and has a finite search space, proving it secure.

Maude-NPA also allows verification of authentication properties by using never patterns, i.e., the reachability analysis succeeds when none of the states in the reachability sequence is an instance of the never pattern. Never patterns can also be used to cut down the search space. Never patterns can share variables

with the attack pattern in order to have more specific patterns and the vertical bar is not included in strands of never patterns, since all the combinations of the vertical bar are taken into account. For instance, we can specify the following authentication attack pattern for NSL by including Bob's strand and adding never patterns for Alice's strand (note that we have to specify two never patterns because states may contain always partial strands):

```
eq ATTACK-STATE(1)
   = :: r ::
     [ nil, -(pk(b,a ; N)),
            +(pk(a,  N ; n(b,r))),
            -(pk(b,n(b,r))) | nil ]
     || empty
     || nil
     || nil
     || never *** for authentication
     (:: r' ::
     [ nil, +(pk(b,a ; N)),
            -(pk(a,  N ; n(b,r))) | +(pk(b,n(b,r))), nil ]
     & S:StrandSet
     || K:IntruderKnowledge)
  [nonexec] .
```

Chapter 4

Protocols with XOR

In this section we describe different protocols in which the main property that we use is the exclusive-or. To describe these protocols we use the textbook Alice-and-Bob notation. This notation is a way of expressing a protocol of correspondence between entities of a dynamic system, such as a computer network. In the context of a formal model, it allows reasoning about the properties of such a system.

The standard notation consists of a set of principals (traditionally named Alice, Bob, Charlie, and so on) who wish to communicate. They may have access to a server `S`, shared keys `K`, timestamps `T`, and can generate nonces `N` for authentication purposes.

A simple example might be the following: A $\rightarrow$ B : $\{X_1\}_{K_{AB}}$. This states that Alice intends a message for Bob consisting of a plaintext X_1 encrypted under shared key K_{AB}. Another example might be the following: B $\rightarrow$ A : $\{N_B\}_{PK_A}$. This states that Bob intends a message for Alice consisting of a nonce encrypted using the public key of Alice.

4.1 Needham-Schroeder Protocol with XOR algebraic properties (XOR-NSL)

This protocol is a variant of the well-known Needham-Schroeder-Lowe (NSL) protocol that involves the exclusive-or (XOR) algebraic property, as shown in [Chevalier 05]. We reproduce the XOR-NSL protocol below, using the textbook Alice-and-Bob notation.

1. A $\rightarrow$ B : $\{N_A,\}_{pk(B)}$
2. B $\rightarrow$ A : $\{N_B,N_A$* B$\}_{pk(A)}$
3. A $\rightarrow$ B : $\{N_B\}_{pk(B)}$

where A, B denote principal names, N_A and N_B denote nonces generated by participants, $\{M\}_K$ denotes encryption of message M using key K and * is the exclusive-or operator. This protocol is insecure; an attack similar to the one of the standard Needham-Schroeder Public Key Protocol (NSPK) was found in [Chevalier 05] exploiting in this case the algebraic properties of the XOR .

The informal description of this attack, is as follows:

1. I(A) $\rightarrow$ B : $\{N_B$,I$\}_{pub(sk_{I(A)})}$
2. B $\rightarrow$ I(A) : $\{N_A,N_B$ * I(A)$\}_{pub(sk_I)}$
3. I(A) $\rightarrow$ B : $\{N_B\}_{pub(sk_B)}$

where A, B are principal names, I is the intruder's name, $I(A)$ is when the intruder spoofs Alice's identity, N_A, N_B are nonces generated by A and $B$and sk_I is the public key of I. As we can see in this description, the intruder spoofs Alice's name. Bob thinks that he was talking to Alice and it doesn't, it was talking to the intruder.

4.1.1 Symbols

In the following we show how to specify the sorts and symbols for this protocol in the Maude-NPA's syntax. More specifically, we need sorts to denote names and nonces. This is specified as follows:

```
sorts Name Nonce Null NNSet .
subsort Name Nonce  <  NNSet  <  Msg .
subsort Name Null  <  Public .
```

We can now specify the different operators needed in XOR-NSL. A nonce generated by principal A is denoted by $n(A, r)$, where r is a unique variable of sort `Fresh`. Concatenation of two messages, e.g., N_A and N_B, is denoted by the operator ;, e.g., $n(A, r)$; $n(B, r')$. Encryption of a message M with the public key of principal A is denoted by pk(A,M), e.g., $N_{Bpk(B)}$ is denoted by

$pk(B, n(B, r'))$. Encryption with a secret key is denoted by $sk(A, M)$. The specification of these operators in the Maude-NPA's syntax is as follows:

```
op pk : Name Msg -> Msg [frozen].
op sk : Name Msg -> Msg [frozen].
```

Next, we specify some principal names. For XOR-NSL, we have three constants of sort `Name`, a (for Alice), b (for Bob) and i (for the Intruder). We need three more operators: one for nonces, one for concatenation and one for the XOR. The nonce operator is specified as follows:

```
op n : Name Fresh -> Nonce  [frozen].
```

Note that the nonce operator has an argument of sort `Fresh` to ensure uniqueness. The argument of type Name is not strictly necessary, but it provides a convenient way of identifying which nonces are generated by which principal. This makes searches more efficient, since it allows us to keep track of the originator of a nonce throughout a search. Finally, message concatenation is specified in Maude-NPA via the infix operator (_ ; _), which is defined as follows:

```
op _ ; _ : Msg Msg -> Msg [gather (e E) frozen].
```

Finally we define the characteristics of the XOR operator which is defined to be used with `NNSet` type variables. Also we indicate that this operator has the associative and commutative properties. This operator is specified as follows:

```
op _ * _ : NNSet NNSet  -> NNSet [assoc  comm frozen] .
```

4.1.2 Algebraic Properties

In this section we specify the algebraic properties of the symbols defined above for the XOR-NSL protocol using equations. In XOR-NSL, we use two equations specifying the relationship between public and private key encryption, as follows:

```
var X : Msg . var A : Name .
eq pk(A,sk(A,X)) = X [variant] .
eq sk(A,pk(A,X)) = X [variant] .
```

The properties of the XOR operator are denoted by the following equations:

```
eq X * X * Y = Y [variant] .
eq X * X = null  [variant] .
eq X * null = X  [variant] .
```

where X and Y are variables of sort `NNSet`. Note that the redundant equational property X * X * Y = Y is necessary in Maude-NPA for coherence purposes; see [Escobar 09].

4.1.3 Strand specification

In this section we describe the strands denoting the actions of the protocol's honest principals and the intruder capabilities. The strands of this protocol are specified as follows:

```
eq STRANDS-PROTOCOL
 = :: r ::   *** Alice ***
    [nil | +(pk(B, n(A,r) ; A)),
           -(pk(A, NB ; n(A,r) * B)),
           +(pk(B, NB)), nil]
&
 :: r' ::   *** Bob ***
   [nil | +(pk(B, NA ; A)),
          -(pk(A, n(B,r') ; NA * B)),
          +(pk(B, n(B,r'))), nil]
```

where A and B are variables denoting names and NA and NB are variables denoting nonces of A and B respectively.

The intruder capabilities are specified by the following strands:

```
eq STRANDS-DOLEVYAO
 = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
   :: nil :: [ nil | -(X ; Y), +(X), nil ] &
   :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
   :: nil :: [ nil | -(XN), -(YN), +(XN * YN), nil ] &
   :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
   :: nil :: [ nil | -(X), +(pk(A,X)), nil ] &
   ::  r  :: [ nil | +(n(i,r)), nil ] &
   :: nil :: [ nil | +(null), nil ] &
```

```
:: nil :: [ nil | +(A), nil ]
```

where variables X and Y denote messages, variable A denotes a sort name of the participant, variables XN and YN are of sort `NNSet`. The first strand denote concatenation of two messages, whereas the second and third strands denote the ability of deconcatenation. The fourth strand allows the intruder to perform the XOR. The fifth, sixth and seventh strands denote the ability of the intruder to perform private encryption and public encryption and to generate his own nonce, respectively. Finally, the two last strands allow the intruder to generate the XOR null element and any name, respectively.

4.1.4 Protocol analysis

This protocol is insecure as it shown in [Chevalier 05] since it is subject to an attack similar to the one of the original NSPK protocol that exploits the algebraic properties of the XOR.

This attack is found in Maude-NPA by searching backwards from the attack pattern below:

```
eq ATTACK-STATE(0)
   = :: r' ::   *** Bob ***
     [nil, +(pk(b, NA ; a)),
           -(pk(a, n(b,r') ; NA * b)),
           +(pk(b, n(b,r'))) | nil]
      || n(b,r') inI
      || nil
      || nil
      || nil
  [nonexec] .
```

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 20 Solutions>> 0
```

```
reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 48 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 72 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 93 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 110 Solutions>> 0

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 116 Solutions>> 1
```

For this attack pattern Maude-NPA finds an initial state. The sequence of exchanged messages of the attack is as follows:

```
 generatedByIntruder(a),
 generatedByIntruder(pk(b, (b * i) ; a)),
 -(pk(b, (b * i) ; a)),
 +(pk(a, n(b, #0:Fresh) ; i)),
 -(pk(a, n(b, #0:Fresh) ; i)),
 +(pk(i, n(a, #1:Fresh) ; a * n(b, #0:Fresh))),
 -(pk(i, n(a, #1:Fresh) ; a * n(b, #0:Fresh))),
 +(n(a, #1:Fresh) ; a * n(b, #0:Fresh)),
 -(n(a, #1:Fresh) ; a * n(b, #0:Fresh)),
 +(a * n(b, #0:Fresh)),
 -(a),
 -(a * n(b, #0:Fresh)),
 +(n(b, #0:Fresh)),
 -(n(b, #0:Fresh)),
 +(pk(b, n(b, #0:Fresh))),
 -(pk(b, n(b, #0:Fresh)))
```

where `generatedByIntruder` is produced by the Maude-NPA's optimization of the Super Lazy intruder (see [Escobar 14] for details), which means it is a

message that the intruder can trivially learn,i.e., it is not necessary to find out how it is generated. In this case, this happens with the name of one of the participants. After that, the intruder is able to use it with his own name and the other participant name to make a public encryption and use it in the attack.

4.2 Fixed Needham-Schroeder protocol with XOR algebraic properties (XOR-NSL-Fix)

Since the Needham-Schroeder Protocol with XOR algebraic properties (XOR-NSL protocol) of Section 5 has an attack, a fixed version of the protocol (XOR-NSL-Fix) is proposed in [Küesters 11] in which the message $\{N_B, N_A$ * B$\}_{pk(A)}$ is replaced by $\{N_B$,h(N_A, N_B) * B$\}_{pk(A)}$, using a hash function h$(\cdot)$.

The informal description of the fixed version of the protocol, is as follows:

1. A $\rightarrow$ B : $\{N_A$,A$\}_{pk(B)}$
2. B $\rightarrow$ A : $\{N_B$, h(N_A, N_B) * B$\}_{pk(A)}$
3. A $\rightarrow$ B : $\{N_B\}_{pk(B)}$

where A, B denote principal names, N_A and N_B denote nonces generated by participants, $\{$M$\}_K$ denotes encryption of message M using key K, * is the exclusive-or operatorand $h(M)$ denotes the hash function over a message M. This protocol is secure with the fix proposed. This fixed version of the XOR-NSL protocol is secure against the attack of [Chevalier 05] .

4.2.1 Symbols

All the symbols are the same that for the original version of the protocol except the one for the hash function. This is specified as follows:

```
sorts Hash .
subsort Hash  < NNSet < Msg .
```

We can now specify the hash operator as follows:

```
op h : Msg -> Hash [frozen] .
```

4.2.2 Algebraic properties

The algebraic properties of the XOR-NSL fixed protocol are the same of the XOR-NSL protocol, which are specified by the AC property of symbol _ * _ and the equations of Section 4.1.2.

4.2.3 Strand specification

As explained in previous section, the protocol itself and the intruder capabilities are both specified using strands. The principal strands are as follows:

```
  eq STRANDS-PROTOCOL
   = :: r ::   *** Alice ***
     [nil | +(pk(B, n(A,r) ; A)),
            -(pk(A, NB ; h(n(A,r) ; NB) * B)),
            +(pk(B, NB)), nil]
  &
   = :: r' ::   *** Bob ***
      [nil | +(pk(B, NA ; A)),
             -(pk(A, n(B,r') ; h(NA ; n(B,r') * B)),
             +(pk(B, n(B,r'))), nil]
[nonexec] .
```

where A and B are variables denoting names and NA and NB are variables denoting nonces of A and B respectively. The difference between the specification of the fixed version and the original version of this protocol (see 4.1.3) is that in the original version of the protocol the message "`pk(A, NB ; n(A,r) * B)`" is replaced by "`pk(A, NB ; h(n(A,r) ; NB) * B)`" for Alice' strand and "`pk(A, n(B,r') ; NA * B)`" is replaced by "`pk(A, n(B,r') ; h(NA ; n(B,r') * B)`" for Bob' strand.

The intruder capabilities are specified as in the XOR-NSL protocol in Section 4.1.3, the only difference is the inclusion of the ability of the intruder to apply the hash function, that is specified as follows:

```
      :: nil :: [ nil | -(XN), +(h(XN)), nil ] &
```

where XN denotes a sort sort of `Msg`.

4.2.4 Protocol analysis

We use an attack pattern similar to the one used in Section 4.1.4. for the original version of the protocol, to prove wether the fix makes the protocol secure.

```
eq ATTACK-STATE(0)
   = :: r' ::   *** Bob ***
    [nil,  +(pk(b, NA ; a)),
           -(pk(a, n(b,r') ; h(NA ; n(b,r') * b)),
           +(pk(b, n(b,r'))) | nil]
        || n(b,r') inI
        || nil
        || nil
        || nil
   [nonexec] .
```

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 4 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 2 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 4 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 2 Solutions>> 0

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 0 Solutions>> 0
```

The protocol is secure with this fix and, when this protocol is analyzed with Maude-NPA, it generates a finite search state space finding no initial state for the attack pattern above.

4.3 Bull Recursive Authentication Protocol (RA)

The Bull Recursive Authentication Protocol (RA) [Bull 97, Comon-Lundh 03], aims at establishing fresh session keys between a fixed number of participants (for instance 3) and a server: one key for each pair of agents. We reproduce the RA below, using textbook Alice-and-Bob notation.

A computes X_A = h(A,B,N_A), (A,B,N_A)
1. A $\rightarrow$ B : X_A
B computes X_B = h(B,C,N_B,Xa), (B,C,N_B,X_A)
2. B $\rightarrow$ C : X_B
C computes X_C = h(C,S,N_C,Xb), (C,S,N_C,X_B)
3. C $\rightarrow$ S : X_C
4. S $\rightarrow$ C : K_{AB} * h(N_A,K_{AS}),
K_{AB} * h(N_B,K_{BS}),
K_{BC} * h(N_B,K_{BS}),
K_{BC} * h(N_C,K_{CS})
5. C $\rightarrow$ B : K_{AB} * h(N_A,K_{AS}),
K_{AB} * h(N_B,K_{BS}),
K_{BC} * h(N_B,K_{BS})
6. B $\rightarrow$ A : K_{AB} * h(N_A,K_{AS})

where A, B, C, S denotes participants names, N_A and N_B, N_C denotes nonces generated by participants, K_{AB} and K_{BC} denote session keys between participants A and B and B and C respectively, K_{AS}, K_{BS} and K_{CS} denote encryption of principal names using the server's nonce, and * is the exclusive-or operator. This protocol is insecure; an attack similar to the one of the standard Needham Schroeder protocol was found in [Chevalier 05] exploiting in this case the algebraic properties of the XOR .

As we can see in [Comon-Lundh 03] this protocol is subject to an attack because secrecy of K_{AB} from C fails if the following constraint is satisfiable: $T = \{M_B, x, y, K_{AB} * h(K_A, N_A), K_{AB} * h(K_B, N_B), K_{BC} * h(K_B, N_B), K_{BC}$

$* \ h(x,y)\}$ implies K_{AB}, where variables x, y represent terms whose value can be chosen by C . We guess that x, y are not instantiated and that the subterms are derived in the following order: $h(x,y) < K_{BC} < K_{AB}$. Observe that $h(x,y)$ is derivable from T in one step by applying function symbol h to x and y, K_{BC} is derivable from $T \cup h(x,y)$ in one step, by (X), K_{AB} is derivable from $T \cup h(x,y) \cup K_{BC}$ in one step, by two application of (X) : $((K_{AB} * h(K_B, N_B)) * ((K_{BC} * h(K_B, N_B)) * K_{BC} = K_{AB}$. Therefore, after the second reduction step we obtain an empty constraint, proving that secrecy of K_{AB} is violated.

The informal description of this attack is as follows:

A computes X_A = h(A,B,N_A), (A,B,N_A)
1. A $\rightarrow$ B : X_A
B computes X_B = h(B,C,N_B,Xa), (B,C,N_B,X_A)
2. B $\rightarrow$ C : X_B
C computes X_C = h(C,S,N_C,Xb), (C,S,N_C,X_B)
3. C $\rightarrow$ S : X_C
4. S $\rightarrow$ C : K_{AB} * h(N_A,K_{AS}),
K_{AB} * h(N_B,K_{BS}),
K_{BC} * h(N_B,K_{BS}),
K_{BC} * h(N_C,K_{CS})
5. C $\rightarrow$ B : K_{AB} * h(N_A,K_{AS}),
K_{AB} * h(N_B,K_{BS}),
K_{BC} * h(N_B,K_{BS})
6. B $\rightarrow$ I : K_{AB}, h(N_A,K_{AS})
7. I(B) $\rightarrow$ A : K_{AB} * h(N_A,K_{AS})

4.3.1 Symbols

The symbols of the Bull Recursive Authentication Protocol (RA) are the same of the XOR-NSL-Fix protocol, which are specified by the equations of Section 4.2.1.

4.3.2 Algebraic properties

The algebraic properties of the Bull Recursive Authentication Protocol (RA) are the same of the XOR-NSL protocol, which are specified by AC and the equations of Section 4.1.2.

4.3.3 Strand specification

In this section we describe the strands denoting the actions of the protocol's honest principals and the intruder capabilities. First of all we define different variables of type Hash that will be used at the protocol specification.

1. HA represents h(A ; B ; N_A)
2. HA' represents h(K_A ; N_A)
3. HB represents h(B, C ; N_B ; (A ; B ; N_A ; h(A ; B ; N_A)))
4. HB' represents h(K_B, N_B)

The strands of this protocol are specified as follows:

```
eq STRANDS-PROTOCOL =
 :: r :: ***  Alice ***
 [nil | +(A ; B ; n(A,r) ; h(A ; B ; n(A,r))),
        -((SK * h(pkey(A,NS) ; n(A,r)))), nil ]
 &

 :: r ::  ***  Bob ***
 [nil | -(A ; B ; NA ; HA),
        +(B ; C ; n(B,r) ; (A ; B ; NA ; HA) ;
          h(B ; C ; n(B,r) ; (A ; B ; NA ; HA))),
        -((SK * HA') ;
          (SK * h(pkey(B,NS) ; n(B,r))) ;
          (SK' * h(pkey(B,NS) ; n(B,r)))),
        +(SK * HA'), nil]
 &

 :: r :: *** Charlie ***
 [nil | -(B ; C ; NB ; (A ; B ; NA ; HA) ; HB),
        +(C ; S ; n(C,r) ; (B ; C ; NB ; A ; B ; NA ; HA ; HB) ;
          h(C ; S ; n(C,r) ; ( B ; C ; NB ; A ; B ; NA ; HA ; HB))),
```

```
          -(( SK * HA') ;
            (SK * HB') ;
            (SK' * HB') ;
            (SK' * h(pkey(C,NS) ; n(C,r)))),
          +((SK * HA') ;
            (SK * HB') ;
            (SK' * HB')), nil]
   &

:: r,r' :: *** Server ***
   [nil | -(C ; S ; NC ; B ; C ; NB ; A ; B ; NA ; h(A ; B ; NA) ;
            h(B ; C ; NB ; A ; B ; NA ; h(A ; B ; NA)) ;
            h(C ; S ; NC ; B ; C ; NB ; A ; B ; NA ;
            h(A ; B ; NA) ; h(B ; C ; NB ; A ; B ; NA ; h(A ; B ; NA)))),
         +((seskey(A, B, n(s,r)) * h(pkey(A,n(s,r)) ; NA)) ;
           (seskey(A, B, n(s,r)) * h(pkey(B,n(s,r)) ; NB)) ;
           (seskey(B, C, n(s,r')) * h(pkey(B,n(s,r)) ; NB)) ;
           (seskey(B, C, n(s,r')) * h(pkey(C,n(s,r)) ; NC))), nil]
```

where A, B and C are variables denoting names and NA, NB and NC are variables denoting nonces of A, B and C respectively.

The intruder capabilities are specified by the following strands:

```
eq STRANDS-DOLEVYAO
 = :: nil :: [nil | -(X), -(Y), +(X ; Y), nil] &
   :: nil :: [nil | -(X ; Y), +(X), nil] &
   :: nil :: [nil | -(X ; Y), +(Y), nil] &
   :: nil :: [nil | -(XN), -(YN), +(XN * YN), nil] &
   :: nil :: [nil | -(XN), +(h(XN)), nil] &
   :: nil :: [nil | -(pkey(i,NX)), +(NX), nil] &
   :: nil :: [nil | -(NX), +(pkey(A,NX)), nil] &
   ::  r  :: [nil | +(n(i,r)), nil] &
   :: nil :: [nil | +(null), nil] &
   :: nil :: [nil | +(A), nil]
 [nonexec] .
```

where variables X and Y denote messages, variable A denotes a sort name of the participant, variables XN and YN denote `Msg`. The first strand denotes concatenation of two messages, whereas the second and third strands denote the ability of deconcatenation. The fourth strand allows the intruder to perform the

XOR. The fifth strand denote the ability of the intruder to generate his own hash function. The sixth strand allows the intruder to encrypt data with his public key. The seventh strand allows the intruder to encrypt a message of sort `Msg` with any public Key and the eight strand allows the intruder to generate his own nonce, respectively. Finally, the two last strands allow the intruder to generate the XOR null element and any name, respectively.

4.3.4 Protocol analysis

This attack is found in Maude-NPA by searching backwards from the attack pattern below:

```
  eq ATTACK-STATE(0)
   = :: r :: *** Alice ***
     [nil, +(a ; b ; n(a,r) ; h(a ; b ; n(a,r))),
           -(SK * h(pkey(a,NS) ; n(a,r))) | nil]
     || empty
     || nil
     || nil
     || nil
[nonexec] .
```

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 4 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 8 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 14 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 18 Solutions>> 0
```

```
reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 20 Solutions>> 0

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 24 Solutions>> 1

reduce in MAUDE-NPA : summary(7) .
result Summary: States>> 32 Solutions>> 1
```

For this attack pattern Maude-NPA finds an initial state as it was expected. The sequence of exchanged messages of the attack is as follows

```
+(a ; b ; n(a, #0:Fresh) ; h(a ; b ; n(a, #0:Fresh))),
-(a ; b ; n(a, #0:Fresh) ; h(a ; b ; n(a, #0:Fresh))),
 +(b ; n(a, #0:Fresh) ; h(a ; b ; n(a, #0:Fresh))),
 -(b ; n(a, #0:Fresh) ; h(a ; b ; n(a, #0:Fresh))),
 +(n(a, #0:Fresh) ; h(a ; b ; n(a, #0:Fresh))),
 -(n(a, #0:Fresh) ; h(a ; b ; n(a, #0:Fresh))),
 +(n(a, #0:Fresh)),
 generatedByIntruder(pkey(a, #2:Nonce)),
 -(pkey(a, #2:Nonce)),
 -(n(a, #0:Fresh)),
 +(pkey(a, #2:Nonce) ; n(a, #0:Fresh)),
 -(pkey(a, #2:Nonce) ; n(a, #0:Fresh)),
 +(h(pkey(a, #2:Nonce) ; n(a, #0:Fresh))),
 generatedByIntruder(#1:Sessionkey),
 -(#1:Sessionkey),
 -(h(pkey(a, #2:Nonce) ; n(a, #0:Fresh))),
 +(#1:Sessionkey * h(pkey(a, #2:Nonce) ; n(a, #0:Fresh))),
 -(#1:Sessionkey * h(pkey(a, #2:Nonce) ; n(a, #0:Fresh)))
```

4.4 Bull Recursive Authentication Protocol Fix (RA-Fix Protocol)

Since the RA of previous section has an attack, a fixed version of the protocol (RA-Fix) is proposed in [Küesters 11] in which the message $K_{AB} * h(N_A, key(A))$ sent by the key distribution server to A is replaced by $K_{AB} * h(N_A, B, key(A))$.

We reproduce the RA-fix protocol below, using textbook Alice-and-Bob notation.

A computes X_A = h(A,B,N_A), (A,B,N_A)
1. A $\rightarrow$ B : X_A
B computes X_B = h(B,C,N_B,Xa), (B,C,N_B,X_A)
2. B $\rightarrow$ C : X_B
C computes X_C = h(C,S,N_C,Xb), (C,S,N_C,X_B)
3. C $\rightarrow$ S : X_C
4. S $\rightarrow$ C : K_{AB} * h(N_A,B,K_{AS}),
K_{AB} * h(N_B,K_{BS}),
K_{BC} * h(N_B,K_{BS}),
K_{BC} * h(N_C,K_{CS})
5. C $\rightarrow$ B : K_{AB} * h(N_A,B,K_{AS}),
K_{AB} * h(N_B,K_{BS}),
K_{BC} * h(N_B,K_{BS})
6. B $\rightarrow$ A : K_{AB} * h(N_A,B,K_{AS})

where A, B, C, S denote participants names, N_A and N_B, N_C denote nonces generated by participants, Kab and Kbc denote session keys between participants A and B and B and C respectively, K_{AS}, K_{BS} and K_{CS} denote encryption of principal names using Server nonce, and * is the exclusive-or operator.

4.4.1 Symbols

The Symbols of the Bull Recursive Authentication Protocol Fix (RA-Fix Protocol) are the same of the RA, which are specified by the equations of Section 4.3.1.

4.4.2 Algebraic properties

The Algebraic properties of the Bull Recursive Authentication Protocol Fix (RA-Fix Protocol) are the same of the RA, which are specified by the equations of Section 4.3.2 .

4.4.3 Strand specification

In this section we describe the strands denoting the actions of the protocol's honest principals and the intruder capabilities. First of all we define different

variables of type `Hash` that will be used for clarity at the protocol specification.

1. HA represents h(A ; B ; NA)
2. HA' represents h(NA ; B ; KA)
3. HB represents h(B, C ; NB ; (A ; B ; NA ; h(A ; B ; NA)))
4. HB' represents h(NB ; KB)

And after that, we show the protocol strands in Maude-NPA:

```
eq STRANDS-PROTOCOL =
 :: r :: ***  Alice ***
 [nil | +(A ; B ; n(A,r) ; h(A ; B ; n(A,r))),
        -((SK * h(pkey(A,NS) ; B ; n(A,r)))), nil ]
 &

 :: r ::  ***  Bob ***
 [nil | -(A ; B ; NA ; HA),
        +(B ; C ; n(B,r) ; (A ; B ; NA ; HA) ;
          h(B ; C ; n(B,r) ; (A ; B ; NA ; HA))),
        -((SK * HA') ;
          (SK * h(pkey(B,NS) ; n(B,r))) ;
          (SK' * h(pkey(B,NS) ; n(B,r)))),
        +(SK * HA'), nil]
 &

 :: r :: *** Charlie ***
 [nil | -(B ; C ; NB ; (A ; B ; NA ; HA) ; HB),
        +(C ; S ; n(C,r) ;
          (B ; C ; NB ; A ; B ; NA ; HA ; HB) ;
          h(C ; S ; n(C,r) ;
          ( B ; C ; NB ; A ; B ; NA ; HA ; HB))),
        -(SK * HA') ;
         (SK * HB') ;
         (SK' * HB') ;
         (SK' * h(pkey(C,NS) ; n(C,r)))),
        +(SK * HA') ;
         (SK * HB') ;
```

```
           (SK' * HB'), nil]
   &

:: r,r' :: *** Server ***
   [nil | -(C ; S ; NC ; B ; C ; NB ; A ; B ; NA ;
            h(A ; B ; NA) ;
            h(B ; C ; NB ; A ; B ; NA ; h(A ; B ; NA)) ;
            h(C ; S ; NC ; B ; C ; NB ; A ; B ; NA ;
            h(A ; B ; NA) ; h(B ; C ; NB ; A ; B ; NA ;
            h(A ; B ; NA)))),
         +((seskey(A, B, n(s,r)) * h(pkey(A,n(s,r)) ; B ; NA)) ;
           (seskey(A, B, n(s,r)) * h(pkey(B,n(s,r)) ; NB)) ;
           (seskey(B, C, n(s,r')) * h(pkey(B,n(s,r)) ; NB)) ;
           (seskey(B, C, n(s,r')) * h(pkey(C,n(s,r)) ; NC))), nil]
```

where A, B and C are variables denoting names and NA, NB and NC are variables denoting nonces of A, B and C respectively. The intruder capabilities are specified as in the original protocol (see Section 4.3.3).

The difference between the specification of the fixed version and the original version of this protocol (see 4.3.3) is that in the original version of the protocol the message "`(SK * h(pkey(A,NS) ; n(A,r)))`" is replaced by "`(SK * h(pkey(A,NS) ; B ; n(A,r)))`" for Alice' strand and "`(seskey(A, B, n(s,r)) * h(pkey(A,n(s,r)) ; NA))`" is replaced by "`(seskey(A, B, n(s,r)) * h(pkey(A,n(s,r)) ; B ; NA))`" for Server' strand.

4.4.4 Protocol analysis

We use an attack pattern similar to the one used previously for the original version of the protocol in Section 4.3.4, to prove if the fix makes the protocol secure.

```
eq ATTACK-STATE(0)
 = :: r :: *** Alice ***
   [nil, +(a ; b ; n(a,r) ; h(a ; b ; n(a,r))),
         -(SK * h(pkey(a,NS) ; n(a,r) ; b)) | nil]
   || empty
   || nil
   || nil
```

```
     || nil
[nonexec] .
```

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 4 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 8 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 11 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 9 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 8 Solutions>> 1

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 10 Solutions>> 0

reduce in MAUDE-NPA : summary(7) .
result Summary: States>> 12 Solutions>> 0

reduce in MAUDE-NPA : summary(8) .
result Summary: States>> 11 Solutions>> 0

reduce in MAUDE-NPA : summary(9) .
result Summary: States>> 12 Solutions>> 0

reduce in MAUDE-NPA : summary(10) .
result Summary: States>> 17 Solutions>> 0

reduce in MAUDE-NPA : summary(11) .
result Summary: States>> 26 Solutions>> 0
```

```
reduce in MAUDE-NPA : summary(12) .
result Summary: States>> 33 Solutions>> 0

reduce in MAUDE-NPA : summary(13) .
result Summary: States>> 29 Solutions>> 0

reduce in MAUDE-NPA : summary(14) .
result Summary: States>> 18 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 11 Solutions>> 0

reduce in MAUDE-NPA : summary(16) .
result Summary: States>> 8 Solutions>> 0

reduce in MAUDE-NPA : summary(17) .
result Summary: States>> 10 Solutions>> 0

reduce in MAUDE-NPA : summary(18) .
result Summary: States>> 13 Solutions>> 0

reduce in MAUDE-NPA : summary(19) .
result Summary: States>> 14 Solutions>> 0

reduce in MAUDE-NPA : summary(20) .
result Summary: States>> 13 Solutions>> 0

reduce in MAUDE-NPA : summary(21) .
result Summary: States>> 11 Solutions>> 0

reduce in MAUDE-NPA : summary(22) .
result Summary: States>> 9 Solutions>> 0

reduce in MAUDE-NPA : summary(23) .
result Summary: States>> 6 Solutions>> 0
```

```
reduce in MAUDE-NPA : summary(24) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(25) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(26) .
result Summary: States>> 0 Solutions>> 0
```

The protocol is secure with this fix and, when this protocol is analyzed with Maude-NPA, it generates a finite search state space finding no initial state for the attack pattern above.

4.5 Shoup-Rubin Protocol

Shoup and Rubin's contribution is the design of a smart card protocol for session-key distribution, which is discussed within an extension of the Bellare-Rogaway's framework [Bella 03] [Shoup 96]. In the original version of the protocol, there were five participants of the protocol: Alice, Bob, the Server and smart cards of Alice and Bob. According to what the protocol says, it is assumed that the communication between smart cards and participants Alice and Bob is secure and therefore the attacker can not extract anything from that communication. We have made the following simplifications to this protocol:

- We have removed the strands of smart cards.
- Alice generates its nonce, rather than generate their smart card and send it to her.
- Bob generates its nonce, rather than generate their smart card and send it to him.
- Bob generates the session key and sends it to Alice. In the original version was one of the smart cards who generated it and is sending his honest participant and this corresponding forward it to the other honest participant.

We use these aliases to make easy the understanding of the specification:

- S_K *represents* $\{A, B\}_{N_B}$
- P_{AB} *represents* $K_{AB} * \{B\}_{K_A}$

– K_{AB} *represents* $\{A\}_{K_B}$

The specification of this protocol using textbook Alice-and-Bob notation is as follows:

1. A → S : A, B
2. S → A : $P_{AB}, \{P_{AB}, B\}_{K_A}$
3. A → B : A, N_A
4. B → A : $N_B, \{N_A, N_B\}_{K_{AB}}$
5. B → A : $\{S_K\}_{K_{AB}}$
6. A → B : $\{N_B\}_{K_{AB}}$

where A, B denote principal names, N_A and N_B denote nonces generated by participants, $\{\mathrm{M}\}_K$ denotes encryption of message M using key $K$and * is the exclusive-or operator.

We use this attack pattern to check if the session key is secure

```
eq ATTACK-STATE(0)
= :: r :: *** Bob ***
  [nil, -(a ; NCA),
        +(n(b,r) ; e(nse(symKey(b), a), NCA ; n(b,r))),
         *** Simulation: Bob sends session key to Alice
        +(e(nse(symKey(b), a), sesK(n(b,r), a, b))),
        -(e(nse(symKey(b), a), n(b,r))) | nil]
        || sesK(n(b,r), a, b) inI
        || nil
        || nil
        || nil
   [nonexec] .
```

As you can see in Section 4.5.4 this protocol is secure.

4.5.1 Symbols

In the following we show how to specify the sorts and symbols for this protocol in the Maude-NPA's syntax. More specifically, we need sorts to denote names, nonces, hash operator and Keys. This is specified as follows:

```
sorts Name Nonce Symkey Key Sessionkey NSymEnc Null .
```

```
subsort Name Nonce < Msg .
subsort Symkey Sessionkey NSymEnc <  Msg .
subsort Name < Public .
subsort Null < NSymEnc .
```

We can now specify the different operators needed in the Shoup-Rubin protocol. A nonce generated by principal A is denoted by n(A, r), where r is a unique variable of sort `Fresh`. Concatenation of two messages, e.g., N_A and N_B, is denoted by the operator " ; ", e.g., $n(A, r)$; n(B, r'). Next, we specify some principal names. For XOR-NSL, we have three constants of sort `Name`, a (for Alice), b (for Bob) and i (for the Intruder). We need three more operators: one for nonces, one for concatenation and one for the XOR. The nonce operator is specified as follows:

```
op n : Name Fresh -> Nonce  [frozen].
```

Note that the nonce operator has an argument of sort `Fresh` to ensure uniqueness. The argument of type `Name` is not strictly necessary, but it provides a convenient way of identifying which nonces are generated by which principal. This makes searches more efficient, since it allows us to keep track of the originator of a nonce throughout a search. Message concatenation is specified in Maude-NPA via the infix operator _ ; _, which is defined as follows:

```
op _ ; _ : Msg Msg -> Msg [gather(e E) frozen].
```

Next we define the characteristics of the XOR operator which is defined to be used with `NSymEnc` type variables. Also we indicate that this operator has the associative and commutative properties.

```
op _*_ :  NSymEnc  NSymEnc -> NSymEnc  [assoc comm frozen] .
```

Next we define diferents kinds of key generators. First we define the *sesK* operator which is defined to be used with the principal names and the nonce of one of the participants.

```
op sesK : Nonce Name Name -> Sessionkey [frozen] .
```

Second we define the *symKey* operator which is defined to be used with the participant name.

```
op symKey : Name -> Symkey [frozen] .
```

Also we define the *nse* operator which is defined as follows:

```
op nse : Symkey Msg -> NSymEnc [frozen] .
```

Next we define diferents kinds of encryption and decryption operators. We define the first encryption and decryption operators which are defined to be used with an arguments of sort `Key` and `Msg`.

```
op e  : Key Msg -> Msg [frozen] .
op d  : Key Msg -> Msg [frozen] .
```

Next we define the second encryption and decryption operators which are defined to be used with an arguments of sort `Symkey` and `Msg`.

```
op se : Symkey Msg -> Msg [frozen] .
op sd : Symkey Msg -> Msg [frozen] .
```

Next we define diferents kinds of key generators. We define the first key generator operator which are defined to be used with the principals names and the nonce of one of the participants

```
op sesK : Nonce Name Name -> Sessionkey [frozen] .
```

Next we define the second key generator operator which are defined to be used with the participant name .

```
op symKey : Name -> Symkey [frozen] .
```

And finally we define the third key generator operator which are defined to be used with a sort `Symkey` and the participant name.

```
op nse : Symkey Name -> NSymEnc [frozen] .
```

4.5.2 Algebraic properties

In this section we specify the algebraic properties of the symbols defined in the section 6.5.1, using equations apart of associative, commutative and identity properties. In Shoup-Rubin protocol, we use two equations specifying the relationship for each one of the two different types of encryption/decryption that we use. We specify as follows:

```
var NSEnc : NSymEnc .  var M : Msg .
eq d(NSEnc, e(NSEnc, M)) = M [variant] .
eq e(NSEnc, d(NSEnc, M)) = M [variant] .

var SKey : Symkey .
eq sd(SKey, se(SKey, M)) = M [variant].
eq se(SKey, sd(SKey, M)) = M [variant].
```

The properties of the XOR operator are denoted by the following equations:

```
var NSEnc NSEnc' : NSymEnc .
eq  null * NSEnc = NSEnc [variant] .
eq NSEnc * NSEnc = null  [variant] .
eq NSEnc * NSEnc * NSEnc' = NSEnc'  [variant] .
```

Note that the redundant equational property of the last equation is necessary in Maude-NPA for coherence purposes; see [Escobar 09].

4.5.3 Strand specification

In this section we describe the strands denoting the actions of the protocols honest principals and the intruder capabilities. The strands of this protocol are specified as follows:

```
:: r :: *** Alice ***
   [nil | +(A ; B),
          -((PAB * nse(symKey(A), B))  ;
           (se(symKey(A), (PAB * nse(symKey(A), B)) ; B))),
          +(A ; n(A,r)),
          -(NCB ; e(PAB,  n(A,r) ; NCB)),
          ***Simulation: Bob sends session key to Alice
          -(e(PAB, sesK(NCB, A, B))),
          +(e(PAB, NCB)), nil]
  &

  :: r ::  *** Bob ***
   [nil | -(A ; NCA),
          +(n(B,r) ; e(nse(symKey(B), A), NCA ; n(B,r))),
          *** Simulation: Bob sends session key to Alice
          +(e(nse(symKey(B), A), sesK(n(B,r), A, B))),
```

```
          -(e(nse(symKey(B), A), n(B,r))), nil]
  &

:: nil :: *** Server ***
  [nil | -(A ; B),
         +((nse(symKey(B), A) * nse(symKey(A), B)) ;
            se(symKey(A),
           (nse(symKey(B), A) * nse(symKey(A), B)) ; B)), nil]
```

where A and B are variables denoting names and NA and NB are variables denoting nonces of A and B respectively.
The intruder capabilities are specified by the following strands:

```
eq STRANDS-DOLEVYAO
 = :: nil ::[nil | -(X), -(Y), +(X ; Y), nil ] &
   :: nil :: [nil | -(X ; Y), +(X), nil ] &
   :: nil :: [nil | -(X ; Y), +(Y), nil ] &
   :: nil :: [nil | -(NSEnc), -(NSEnc'), +(NSEnc *  NSEnc'), nil ] &
   :: nil :: [nil | -(M), -(NSEnc), +(e(NSEnc,M)), nil ] &
   :: nil :: [nil | -(M), -(NSEnc), +(d(NSEnc,M)), nil ] &
   :: nil :: [nil | -(M), -(SK), +(se(SK,M)), nil ] &
   :: nil :: [nil | -(M), -(SK), +(sd(SK,M)), nil ] &
   :: nil :: [nil | -(SK), +(nse(SK, i)), nil ] &
   :: nil :: [nil | +(symKey(i)), nil ] &
   :: nil :: [nil | -(N), +(sesK(N,i,A)), nil ] &
   :: nil :: [nil | -(N), +(sesK(N,A,i)), nil ] &
   ::  r  :: [nil | +(n(i,r)), nil ] &
   :: nil :: [nil | +(null), nil ] &
   :: nil :: [nil | +(A), nil ]
[nonexec] .
```

where variables X and Y denote messages, variable A denotes a sort name of the participant, variable SK denotes `Symkey`, variable M denotes `Msg`, variable N denotes `Nonce`, and variables NSEnc and NSEnc' are of sort `NSymEnc`. The first strand denote concatenation of two messages, whereas the second and third strands denote the ability of deconcatenation. The fourth strand allows the intruder to perform the XOR. The fifth , sixth, seventh and eighth strands allows the intruder to perform different kinds of encryption and decryption. The ninth strand denotes the ability of the intruder to generate any kind of nse encryption with his name. The tenth strand denotes the ability of the intruder to generate his own symmetric key. The eleventh and twelfth strands denote the

ability of the intruder of generate a session key. Finally the three last strands allow the intruder to generate his own nonce, the XOR null element and any name, respectively.

4.5.4 Protocol analysis

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 4 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 12 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 28 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 51 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 74 Solutions>> 1

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 80 Solutions>> 0

reduce in MAUDE-NPA : summary(7) .
result Summary: States>> 70 Solutions>> 0

reduce in MAUDE-NPA : summary(8) .
result Summary: States>> 55 Solutions>> 0

reduce in MAUDE-NPA : summary(9) .
result Summary: States>> 45 Solutions>> 0

reduce in MAUDE-NPA : summary(10) .
result Summary: States>> 46 Solutions>> 0
```

```
reduce in MAUDE-NPA : summary(11) .
result Summary: States>> 36 Solutions>> 0

reduce in MAUDE-NPA : summary(12) .
result Summary: States>> 43 Solutions>> 0

reduce in MAUDE-NPA : summary(13) .
result Summary: States>> 21 Solutions>> 0

reduce in MAUDE-NPA : summary(14) .
result Summary: States>> 11 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(16) .
result Summary: States>> 0 Solutions>> 0
```

The protocol is secure against this attack and, when this protocol is analyzed with Maude-NPA, it generates a finite search state space finding no initial state for the attack pattern above.

4.6 Symmetric Key distribution protocol using Smart Cards (SK3)

The Symmetric Key distribution using Smart Cards (SK3) protocol [Bond 01] is a key distribution protocol for smart cards, which uses the XOR operator. This protocol is a version of the Shoup-Rubin protocol of the Section 4.5 . We have specified a simplified version of this protocol. In the original version of the protocol, there were five participants of the protocol: Alice, Bob, the Server and the smart cards of Alice and Bob. The informal description of the protocol in [Bond 01] assumes that the communication between smart cards and participants Alice and Bob is secure and therefore the attacker can not extract anything from that communication. We have made the following simplifications to this protocol:

- We have removed the strands of smart cards.
- Alice generates its nonce, rather than generate their smart card and send it to her.
- Bob generates its nonce, rather than generate their smart card and send it to him.
- Bob generates the session key and sends it to Alice. In the original version was one of the smart cards who generated it and is sending his honest participant and this corresponding forward it to the other honest participant.

We use these aliases to make easier the understanding of the specification:

- S_K *represents* $\{A, B, 1\}_{N_B}$
- P_{AB} *represents* K_{AB} * $\{B, 1\}_{K_A}$
- K_{AB} *represents* $\{A, 0\}_{K_B}$

The specification using textbook Alice-and-Bob notation would be as follows:

1. A $\rightarrow$ S : A, B
2. S $\rightarrow$ A : P_{AB}, $\{P_{AB}, B, 2\}_{K_A}$
3. A $\rightarrow$ B : A, N_A
4. B $\rightarrow$ A : N_B, $\{N_A, N_B, 1\}_{K_{AB}}$
5. B $\rightarrow$ A : $\{S_K\}_{K_{AB}}$
6. A $\rightarrow$ B : $\{N_B, 0, 1\}_{K_{AB}}$

where A, B denote principal names, N_A and N_B denote nonces generated by participants, $\{\mathrm{M}\}_K$ denotes encryption of message M using key K, * is the exclusive-or operator and 0 and 1 denote a sortrbitrary padding constants, known by all principals.

We use this attack pattern to check if the session key is secure

```
eq ATTACK-STATE(0)
= :: r :: *** Bob ***
   [nil, -(a ; NCA),
         +(n(b,r) ; e(nse(symKey(b), a), NCA ; n(b,r) ; 1)),
         *** Simulation: Bob sends session key to Alice
         +(e(nse(symKey(b), a), sesK(n(b,r), a, b, 1))),
```

```
          -(e(nse(symKey(b), a), n(b,r) ; 0 ; 1))  |  nil]
        || sesK(n(b,r), a, b, 1) inI
        || nil
        || nil
        || nil
[nonexec] .
```

As you can see in Section 4.6.4 this protocol is secure.

4.6.1 Symbols

The symbols that we use for this protocol are the same that we use for the Shoup-Rubin protocol in Section 4.5.1.

4.6.2 Algebraic properties

As in the previous section, we use the same equational properties that we use for the Shoup-Rubin protocol in Section 4.5.2.

4.6.3 Strand specification

In this section we describe the strands denoting the actions of the protocol's honest principals and the intruder capabilities. The strands of this protocol are specified as follows:

```
:: r :: *** Alice ***
[nil | +(A ; B),
        -((PAB * nse(symKey(A), B ; 1))  ;
          (se(symKey(A), (PAB * nse(symKey(A), B)) ; B ; 2))),
        +(A ; n(A,r)),
        -(NCB ; e(PAB,  n(A,r) ; NCB)),
         ***Simulation: Bob sends session key to Alice
        -(e(PAB, sesK(NCB, A, B, 1))),
        +(e(PAB, NCB ; 0 ; 1)), nil]
  &
  :: r ::  *** Bob ***
  [nil | -(A ; NCA),
         +(n(B,r) ; e(nse(symKey(B), A), NCA ; n(B,r) ; 1)),
         *** Simulation: Bob sends session key to Alice
         +(e(nse(symKey(B), A), sesK(n(B,r), A, B, 1))),
```

```
          -(e(nse(symKey(B), A), n(B,r) ; 0 ; 1)), nil]
  &
:: nil :: *** Server ***
[nil | -(A ; B),
       +((nse(symKey(B), A ; 0) * nse(symKey(A), B ; 1)) ;
          se(symKey(A),
          (nse(symKey(B), A ; 0) * nse(symKey(A), B ; 1) ; B ; 2), nil]
```

where A and B are variables denoting *names* and NCA and NCB are variables denoting *nonces* of A and B respectively and 0 and 1 denote a sort of arbitrary padding constants.

The intruder capabilities of the Symmetric key distribution using Smart Cards (SK3) are the same of the Shoup-Rubin protocol, which are specified by the strands of Section 6.5.3.

4.6.4 Protocol analysis

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 2 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 0 Solutions>> 0
```

The protocol is secure against this attack and, when this protocol is analyzed with Maude-NPA, it generates a finite search state space finding no initial state for the attack pattern above.

Chapter 5

The CCA series of protocols

CCA stands for Common Cryptographic Architecture API [IBM 08] as implemented on the hardware security module IBM 4758 (an IBM cryptographic coprocessor). As we can see in [Keighren 06], the CCA is a key management system, which provides commands that use encrypted keys to achieve desired functions. A 168-bit triple-DES key, known as the master key, is stored in the security module's tamper proof memory and is used to encrypt all other keys which are then kept on the host computer. These other keys, known as working keys, are used to perform the various functions provided by the CCA API, and have types associated with them. In all, the CCA supports the following functions and features:

- Encryption and decryption of data, using the DES algorithm [DES Encryption].
- Message authentication code (MAC) generation, and data hashing functions.
- Formation and validation of digital signatures.
- Generation, encryption, translation and verification of PINs and transaction validation messages.
- General key management facilities.
- Administrative services for controlling the initialization and operation of the security module.

As a number of the provided commands are particularly sensitive, the CCA enforces an access-control system, whereby certain commands are only available under specific circumstances. It is, however, the responsibility of the device administrator to ensure that the correct separation of duty provided is upheld.

The CCA API uses four main types for classifying DES working keys, each of which is further sub-divided into more specific and restrictive types. Each type takes the form of a control vector (a bit-string that is the same length as the associated working key). A working key is stored outside of the security module, encrypted under the exclusive-or of the device's master key and the control vector representing the type of the key. The main key types, and their uses, are as follows:

Data Keys
Keys of this type are used to encipher and decipher arbitrary data, as well as for the generation and verification of message authentication codes (MACs). Subtypes place greater restrictions on exactly which of these various functions a particular key can be used for.

PIN Keys
This type covers keys which are used for PIN block encryption, PIN block decryption, PIN generation and verification, and just PIN verification. A key cannot be of the general PIN type, but instead must be assigned a subtype that restricts its use to exactly one of the four operations mentioned.

Key Encryption Keys
These keys are used to encrypt and decrypt other working keys during transfer between security modules, and are divided into import and export types. Keys encrypted under an export key are referred to as external keys, as they must be imported into a security module before they can be used. Note that the transfer of a working key requires the same key encryption key to be present in both security modules (as an export key in one and as an import key in the other).

Key Generation Keys
The CCA API provides commands which generate DES keys, given an initial key, and will typically use the provided key to encrypt or decrypt a supplied piece of data. This type covers such initial keys and restricts them from being used with other commands (e.g. Encipher and Decipher) in order to prevent

the value of the generated key being discovered.

The typing mechanism restricts the working keys which can be used for a particular command, for example, the PIN derivation key used in the verification of a customer's PIN cannot be used with the Encipher command to encrypt arbitrary data.

The following terms are used in this section to represent the various control vectors and cryptographic keys:

- DATA → Control vector for data keys
- IMP → Control vector for import-type key encryption keys
- EXP → Control vector for export-type key encryption keys
- KP → Control vector indicating that a key is only a key part, and not a complete key
- KM → The security module-s master key
- ekek → An arbitrary key encryption key
- ek → An arbitrary cryptographic key
- T → An unknown, randomly generated, new cryptographic key An arbitrary key type control vector. We can say that T $\in \{DATA, IMP, EXP, PIN\}$.
- kpi → Key part i (used to build an arbitrary key)
- x → Arbitrary (unencrypted) data

The exact steps that the security module carries out for each command have not been included, since the process is virtually the same in all cases. The master key and all control vectors are known to the security module, and any additional information required is either passed as a plaintext parameter, or is encrypted under a known key. For example, in the case of the Key Import command, the security module knows both KM and IMP so is therefore able to obtain kek from the third parameter. This key encryption key is then XOR-ed with the second parameter, type, and used to obtain key from the first parameter. Finally, key is encrypted under the exclusive-or of KM and type to produce

the result that is returned by the command.

In the following we test nine protocols:

- CCA-0 Original (Section 5.1)
- CCA-0 version of Küesters and Truderung (Section 5.2)
- CCA-1A (Section 5.4)
- CCA-1B (Section 5.5)
- CCA-1B version of Küesters and Truderung Original (Section 5.6)
- CCA-2B (Section 5.8)
- CCA-2C (Section 5.9)
- CCA-2C version of Küesters and Truderung Original (Section 5.10)
- CCA-2E (Section 5.11)

5.1 CCA-0 Original

In this section we describe the specification and analysis of the original CCA protocol (CCA-0).

The informal description of the CCA API is as follows:

API Operator	**Description**
Encipher	x, $\{\text{eK}\}_{\{KM*DATA\}} \rightarrow \{\text{x}\}_{eK}$
Decipher	$\{\text{x}\}_{eK}$, $\{\text{eK}\}_{\{KM*DATA\}} \rightarrow$ x
Key Export	$\{\text{eK}\}_{(KM*T\}}$, T, $\{\text{ekek}\}_{\{KM*EXP\}} \rightarrow \{\text{eK}\}_{(ekek*T)}$
Key Import	$\{\text{eK}\}_{(kek*T\}}$, T, $\{\text{kek}\}_{\{KM*IMP\}} \rightarrow \{\text{eK}\}_{\{KM*T\}}$
Key Part Import First	km1, T $\rightarrow \{\text{km1}\}_{\{KM*KP*T\}}$
Key Part Import Middle	km2, $\text{km1}_{\{KM*KP*T\}}$, T $\rightarrow$ (km1 * km2)$_{\{KM*KP*T\}}$
Key Part Import Last	km3, $\text{km2}_{\{KM*KP*T\}}$, T $\rightarrow$ (km2 * km3)$_{\{KM*KP*T\}}$
Key Translate	$\{\text{eK}\}_{ekek1*T}$, T, $\{\text{ekek1}\}_{KM*IMP}$, $\{\text{ekek2}\}_{KM*EXP}$ $\rightarrow \{\text{eK}\}_{(ekek2*T)}$

Table 5.1: CCA API operators and description.

The `Encipher` rule corresponds to a data encryption command which allows data keys to be used to encrypt any given plaintext. `Decipher` allows data keys to be used for decryption. `Key Export` is used to encrypt a working key under a key encrypting key for transport to another HSM (hardware security modules). `Key Import` allows a key from another 4758 module, encrypted for transport under a `Key Encrypting Key (KEK)`, to be made into a working key for this HSM (Hardware Security Modules). The three `Key Part Import` commands can then be used one after the other, by three different security officers, each in possession of one key part, to create the working import key. It is this process that is subverted in Bond' attack to change the type of a key. The last API operator, `Key Translate`, is used to encrypt a key under a different key-encryption-key.

This series of commands builds up a working key from individual parts and can be used in one of two ways. Either the first, the middle and last commands can be used. In order to provide security through separation of duty, the commands are split into three groups, with individuals only allowed access to one, so the combination of these three commands allows more than one individual to obtain completed working keys. The other combination ensures that the people responsible for inserting the key parts cannot obtain a final key and the person who obtains the final key cannot modify it in any way.

In this specification X denotes a sort `Msg`, $eK, ekek, km1, km2, km3$ denote a sort `Key` and * is the exclusive-or operator.

This protocol is insecure. The attack that we found uses the same assumptions as Bond's attack [Bond 01] in terms of the role played by the intruder and his knowledge, and this attack is the same that found Küesters and Truderung in [Küesters 11]. As a result of the attack, the intruder obtains a PIN derivation key in clear, like in the IBM attack and hence, can compute PINs from bank account numbers. We use several strands in the attack pattern to prevent the increase of the number of generated states becomes this protocol into unhandled.

Since the sequence of exchanged messages of this initial state is quite long, we explain it below step by step.

First, the intruder receives $\{km1 * km2\}_{\{IMP*KP*KM\}}$, then he performs

`Key Part Import Last` with {k3 * pin} instead of {`k3`}. In this way he obtains $\{PIN * km1 * km2 * km3\}_{\{IMP*KM\}}$.

Next the intruder uses the same command again, this time with {k3 * pin * exp}, obtaining $\{PIN * km1 * km2 * km3\}_{\{IMP*KM\}}$.

Next, when {PDK} is imported, the intruder uses `Key Import` twice: The first time with input $\{PIN * km1 * km2 * km3\}_{\{IMP*KM\}}$ and $\{PDK\}_{\{PIN*km1*km2*km3\}}$ and the results is the message $\{PDK\}_{\{KM\}}$.

The second time the command `Key Import` is used with input $\{PIN * EXP * km1 * km2 * km3\}_{\{IMP*KM\}}$, and $\{PDK\}_{\{PIN*km1*km2*km3\}}$ and {type = exp}, which gives the message $\{PDK\}_{\{EXP*KM\}}$.

Now, using `Key Export` with input $\{PDK\}_{\{KM\}}$ and $\{PDK\}_{\{EXP*KM\}}$, the attacker obtains $\{PDK\}_{\{PDK\}}$.

Finally, using `Decipher` with input $\{PDK\}_{\{PDK\}}$ and $\{PDK\}_{\{KM\}}$, the attacker obtains the clear value of `PDK`, which can be then used to obtain the `PIN` for any account number: Given an account number, the corresponding `PIN` is derived by encrypting the account number under `PDK`.

5.1.1 Symbols

In the following we show how to specify the sorts and symbols for this protocol in the Maude-NPA's syntax. More specifically, we need sorts to denote names, different kinds of keys and encrypted data. This is specified as follows:

```
sorts Name Nonce Key Null type KPdk KData KPin KExp KImp
      KKek KMaster KKP .
subsort Name Nonce Key < Msg .
subsort KData KPin KExp KImp < type .
subsort KMaster type KKek Null KKP KPdk < Key .
subsort Null < KData .
subsort Name < Public .
subsort Null type < Public .
```

The specification in Maude-NPA of the different key operators is as follows:

```
op DATA : -> KData .
op PIN : -> KPin .
op EXP : -> KExp .
op IMP : -> KImp .
op KP : -> KKP .
op PDK : -> KPdk .
op KM : -> KMaster .
```

These operators were previously defined in Chapter 5, We can now specify the different operators needed in CCA-0. We specify some principal names. For XOR-NSL, we have three constants of sort `Name`, a (for Alice), b (for Bob) and i (for the Intruder). We need two more operators: one for concatenation and one for the XOR. Message concatenation is specified in Maude-NPA via the infix operator (_ ; _), which is defined as follows:

```
op _ ; _ : Msg Msg -> Msg [gather (e E) frozen].
```

Next we define the characteristics of the XOR operator which is need to be define twice, one to be used with variables of type `Msg` and another to be used with variables of type `Key`. Also we indicate that both operators have the associative and commutative properties.

```
op _ * _ : Msg Msg -> Msg [assoc comm frozen] .
op _ * _ : Key Key -> Key [ditto] .
```

Finally we define a customer's account number (PAN)), as a constant of sort `Msg` operator.

```
op PAN :  -> Msg .
```

5.1.2 Algebraic properties

In this section we specify the algebraic properties of the symbols defined above for the CCA-0 protocol, using equations apart of associative, commutative and identity properties. In CCA-0, we use two equations specifying the relationship between public and private key encryption, as follows:

```
var X : Msg . var A : Name .
eq pk(A,sk(A,X)) = X .
eq sk(A,pk(A,X)) = X .
```

The properties of the XOR operator are denoted by the following equations:

```
vars XN YN : Msg .
eq null * XN = XN [variant].
eq XN * XN = null [variant] .
eq XN * XN * YN = YN [variant] .
```

Note that the redundant equational property $XN * XN * YN = YN$ is necessary in Maude-NPA for coherence purposes; see [Escobar 09].

Finally as we can see in [Küesters 11] we use for DATA operator the following equation:

```
eq DATA = null [variant] .
```

We use two equations specifying the relationship between encryption and decryption, as follows:

```
eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [variant] .
eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [variant] .
```

5.1.3 Strand specification

For this protocol we have followed the original protocol specification instead of the version described in [Küesters 11], which is a simplified version of the original protocol.

Here we show the strand specification, of the original version of the protocol, in Maude-NPA.

```
    eq STRANDS-PROTOCOL
  ***Strands for the transaction
    = :: nil :: ***Encryption with data key
      [nil | -(X),
             -(e(KM * DATA, eK)),
             +(e(eK, X)), nil]
&
      :: nil :: ***Decryption with data key
      [nil | -(e(eK, X)),
```

```
              -(e(KM * DATA, eK)),
              +(X), nil]
&
       :: nil :: ***Key Import
       [nil | -(e(kek * T, eK)),
              -(T),
              -(e(KM * IMP, kek)),
              +(e(KM * T, eK)), nil]
&
        :: nil :: ***Key Export
        [nil | -(e(KM * T, eK)),
               -(T),
               -(e(KM * EXP, ekek)),
               +(e(ekek * T, eK)), nil]
&
        :: nil :: ***Key Part Import First
        [nil | -(km1),
               -(T),
               +(e(KM * KP * T, km1)), nil]
&
        :: nil ::  ***Key Part Import Middle
        [nil | -(km2),
               -(T),
               -(e(KM * KP * T, km1)),
               +(e(KM * KP * T, km1 * km2)), nil]
&
        :: nil ::  ***Key Part Import Last
        [nil | -(km3),
               -(T),
               -(e(KM * KP * T, km2)),
               +(e(KM * T, km2 * km3)), nil]
[nonexec] .
```

where X is a variable of sort `Msg`, $K, kek, km1, km2$ and $km3$ are variables of sort `Key` and T is a variable of sort `Type`.

The intruder capabilities are specified by the following strands:

```
  eq STRANDS-DOLEVYAO
    =   :: nil :: [nil | -(X), -(Y), +(X * Y), nil ] &
        :: nil :: [nil | -(X), -(K), +(e(K,X)), nil ] &
        :: nil :: [nil | -(X), -(K), +(d(K,X)), nil ] &
        :: nil :: [nil, +(Km3) | nil] &
        :: nil :: [nil, +(PIN) | nil] &
        :: nil :: [nil, +(IMP) | nil] &
        :: nil :: [nil, +(EXP) | nil] &
        :: nil :: [nil | +(null), nil] &
        :: nil :: [nil, +(e(KM * KP * IMP, Km1 * Km2)) | nil] &
        :: nil :: [nil, +(e(Km1 * Km2 * Km3 * PIN, PDK)) | nil]
[nonexec] .
```

where variables X and Y denote messages and variable K denotes a sort key. The first strand allows the intruder to perform the XOR. The second and third strands allows the intruder to perform encryption and decryption. The fourth, fifth, sixth and seventh strands allow the intruder to generate different kind of keys. The eight strand allows the intruder to generate the XOR *null* element. As we can see in [Küesters 11], the ninth and tenth strands show how the intruder can perform $KPILast$ and how the key $kek(Km1 * Km2 * Km3)$ is used to import a new PIN-derivation key (PDK) to the security module, respectively.

5.1.4 Protocol analysis

The specifications of the attack in Maude-NPA is as follows:

```
   eq ATTACK-STATE(0)
      = empty
        || e(PDK, PDK) inI
        || nil
        || nil
        || NP: NeverPatterns
[nonexec] .
```

where NP denotes a set of ?Never patterns? to reduce the search space (See Appendix A).

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 5 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 13 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 8 Solutions>> 0

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 7 Solutions>> 1
```

For this attack pattern Maude-NPA finds an initial state as it was expected. The sequence of exchanged messages of the attack is as follows:

```
+(e(PIN * Km1 * Km2 * Km3, PDK)),
+(e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3)),
-(e(PIN * Km1 * Km2 * Km3, PDK)),
-(EXP),
-(e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3)),
+(e(EXP * KM, PDK)),
+(e(IMP * KM, PIN * Km1 * Km2 * Km3)),
-(e(PIN * Km1 * Km2 * Km3, PDK)),
-(null),
-(e(IMP * KM, PIN * Km1 * Km2 * Km3)),
+(e(KM, PDK)),
-(e(KM, PDK)),
-(null),
-(e(EXP * KM, PDK)),
+(e(PDK, PDK))
```

5.2 CCA-0 version of Küesters and Truderung

In this section we describe the specification and analysis of the version of the original CCA-0 protocol they made Küesters and Truderung in [Küesters 11] (CCA-0 Küesters and Truderung).

The informal description of the CCA API is as follows:

API Operator	**Description**
Encipher	x, $\{eK\}_{\{KM*DATA\}} \rightarrow \{x\}_{eK}$
Decipher	$\{x\}_{eK}$, $\{eK\}_{\{KM*DATA\}} \rightarrow x$
Key Export	$\{eK\}_{(KM*T\}}$, T, $\{ekek\}_{\{KM*EXP\}} \rightarrow \{eK\}_{(ekek*T)}$
Key Import	$\{eK\}_{(kek*T\}}$, T, $\{kek\}_{\{KM*IMP\}} \rightarrow \{eK\}_{\{KM*T\}}$
KPI-First + KPI-Add/Middle	km12, T $\rightarrow$ {KM * KP * IMP}
Key Part Import Last	x, T, KM * KP * T $\rightarrow (x)_{\{KM*T\}}$ x, IMP $\rightarrow$ (X * km12)$_{\{KM*IMP\}}$

The `Encipher`, `Decipher`, `Key Export` and `Key Import` API operators are the same that the CCA-0 original protocol (see Section 5.1). The `KPI-First + KPI-Add/Middle` and `Key Part Import Last` API operators are the equivalent to the original `Key Part Import First` , `Key Part Import Middle` and `Key Part Import Last` API operators. They do these modifications to the originals API operators because they had problems to manage the entire exclusive-or theory. The way that this version of the protocol works is the same that for original version (see Section 5.1).

All the variables are the same that for the CCA-0 original protocol except the `km12` variable that denotes a sort `Key`.

This protocol is insecure as the original CCA-0 protocol. The attack is the same that we could found in the original version (see Section 5.1)

5.2.1 Symbols

All the symbols are the same that for the original version of the protocol except the one for the km12 variable . This is specified in Maude-NPA as follows:

```
sorts Key Msg .
subsort Key  <  Msg .
```

We can now specify the km12 variable as follows:

```
var km12 : Key .
```

The specification in Maude-NPA of the different key operators are the same that for the original version of the protocol (see Section 5.2)

5.2.2 Algebraic properties

The algebraic properties of this protocol version are the same than in the original version (see Section 5.1.2).

5.2.3 Strand specification

The strand specification is the same than in the original version (see Section 5.1.3) except for the Key Part Import Last operator. Here we show the specification of the Küesters and Truderung version of this API operator in Maude-NPA.

```
:: nil ::   *** Key Import Last
[nil |  -(X),
-(T),
-(KM * KP * T),
+(e(KM * T, X)), nil]
&
:: nil ::  *** Key Import Last
[nil |  -(X),
-(IMP),
+(e(KM * IMP, X * km12)), nil]
[nonexec] .
```

where X is a variable of sort `Msg`, $km12$ is a variable of sort `Key` and T is a variable of sort `Type`.

The intruder capabilities are specified by the following strands:

```
eq STRANDS-DOLEVYAO
  =   :: nil :: [nil | -(X), -(Y), +(X * Y), nil ] &
      :: nil :: [nil | -(X), -(K), +(e(K,X)), nil ] &
      :: nil :: [nil | -(X), -(K), +(d(K,X)), nil ] &
```

```
        :: nil :: [nil, +(Km3) | nil] &
        :: nil :: [nil, +(PIN) | nil] &
        :: nil :: [nil, +(IMP) | nil] &
        :: nil :: [nil, +(EXP) | nil] &
        :: nil :: [nil | +(null), nil] &
        :: nil :: [nil, +(e(KM * KP * IMP, Km12)) | nil] &
        :: nil :: [nil, +(e(Km1 * Km2 * Km3 * PIN, PDK)) | nil]
[nonexec] .
```

where variables X and Y denote messages and variable K denotes a sort key. The first strand allows the intruder to perform the XOR. The second and third strands allows the intruder to perform encryption and decryption. The fourth, fifth, sixth and seventh strands allow the intruder to generate different kind of keys. The eight strand allows the intruder to generate the XOR *null* element. The ninth strand shows how the intruder can perform the modification of the original protocol that made Küesters and Truderung of `Key Part Import First` and `Key Part Import Middle` and the tenth strand shows how the key $kek(Km1 * Km2 * Km3)$ is used to import a new PIN-derivation key (PDK) to the security module.

5.2.4 Protocol analysis

The specifications of the attack in Maude-NPA is as follows:

```
   eq ATTACK-STATE(0)
      = empty
        || e(PDK, PDK) inI
        || nil
        || nil
        || nil
[nonexec] .
```

Keep in mind one important thing, as this version is simplest than the original version of the protocol we do not need any never pattern to reduce the search space to obtain the desired result.

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 6 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 51 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 331 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 547 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 624 Solutions>> 1

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 936 Solutions>> 4
```

For this attack pattern Maude-NPA finds an initial state as it was expected. The sequence of exchanged messages of the attack is as follows:

```
+(e(PIN * Km1 * Km2 * Km3, PDK)),
-(null),
-(IMP),
+(e(IMP * KM, PIN * Km1 * Km2 * Km3)),
-(EXP),
-(IMP),
+(e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3)),
-(e(PIN * Km1 * Km2 * Km3, PDK)),
-(null),
-(e(IMP * KM, PIN * Km1 * Km2 * Km3)),
+(e(KM, PDK)),
-(e(PIN * Km1 * Km2 * Km3, PDK)),
-(EXP),
-(e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3)),
+(e(EXP * KM, PDK)),
-(e(KM, PDK)),
```

```
-(null),
-(e(EXP * KM, PDK)),
+(e(PDK, PDK))
```

5.3 IBM's first recommendations to avoid the attack

In order to prevent the attack of the CCA-0 protocol [Bond 01], IBM proposed two recommendation. At the first one, they argue the attack would have to be carried out by an insider and that the vulnerability is intrinsic to public key schemes. We suggest that access control should be used to restrict any single insider from having access to both the `PKA Symmetric Key Import`, `Key Import` and `Key Translate`. The `PKA Symmetric Key Import` takes an encrypted data block, containing the key to be imported and corresponding type information and returns the key encrypted under the local master key. The data block is encrypted under the public key that corresponds to the security modules private key.

We created two models, each one allowing access to only one of these functions and checked them with Maude-NPA. The different API operators for the IBM's first recommendation are as follows:

API Operator	CCA-1A	CCA-1B
Encipher	X	X
Decipher	X	X
Key Export	X	X
Key Import	X	
PKA Symmetric Key Import		X
Key Translate		X

Table 5.2: CCA API operators and description.

5.4 CCA-1A

In this section, we explain the specification and analysis of the CCA-1A protocol, that restricts the kind of commands that certain rules may perform. This

version of the protocol uses a subset of operations of the original CCA-0 protocol (See Section 5.1).

5.4.1 Symbols

The symbols that we use for this version of the protocol are the same that we use for the original one (see Section 5.1.1).

5.4.2 Algebraic properties

The algebraic properties of this protocol version are the same than in the original version (see Section 5.1.2).

5.4.3 Strand specification

The strand specification that we use for this subset of the API operators of the protocol are the same that we use for the original one (see Section 5.1.3).

The intruder capabilities that we use for this protocol are the same that we use for the CCA-0 protocol (see Section 5.1.3).

5.4.4 Protocol analysis

We use the same attack pattern of the CCA-0 protocol in Section 5.1.4, but in this case NP denotes a different set of ?Never patterns? to reduce the search space (See Appendix B).

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 5 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 9 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
```

```
result Summary: States>> 6 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 0 Solutions>> 0
```

The protocol is secure against this attack and when this protocol is analyzed with Maude-NPA, it generates a finite search state space finding no initial state for the attack pattern above.

5.5 CCA-1B

In this section, we explain the specification and analysis of the CCA-1B protocol, that restricts the kind of commands that certain rules may perform. This version of the protocol uses a subset of operations of the original CCA-0 protocol and a new one called **PKA Symmetric Key Import**. As we previously define, **PKA Symmetric Key Import** takes an encrypted data block, containing the key to be imported and corresponding type information and returns the key encrypted under the local master key. The data block is encrypted under the public key that corresponds to the security module private key.

The informal specification of this operator is as follows:

PKA Symmetric Key Import
$\{\text{eK ; T}\}_{PKA} \rightarrow \{\text{eK}\}_{(KM*T)}$

5.5.1 Symbols

The symbols that we use for this version of the protocol are the same that we use for the original one (see Section 5.1.1).

5.5.2 Algebraic properties

The algebraic properties of this protocol version are the same than in the original version (see Section 5.1.2).

5.5.3 Strand specification

The strand specification that we use for this subset of the API operators of the protocol are the same that we use for the original one (see Section 5.1.3), and a new one called **PKA Symmetric Key Import**.

The specification in Maude-NPA of the new operator is as follows:

```
:: nil :: ***PKA Symmetric Key Import
[nil | -(e(PKA, eK ; T)),
       +(e(KM * T, eK)), nil]
```

The intruder capabilities that we use for this protocol are the same that we use for the CCA-0 protocol (see Section 5.1.3).

5.5.4 Protocol analysis

We use the same attack pattern of the CCA-0 protocol in Section 5.1.4, but in this case NP denotes a different set of ?Never patterns? to reduce the search space (See Appendix B).

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 2 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 8 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 19 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 13 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 4 Solutions>> 0
```

```
reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 0 Solutions>> 0
```

The protocol is secure against this attack and, when this protocol is analyzed with Maude-NPA, it generates a finite search state space finding no initial state for the attack pattern above.

5.6 CCA-1B version of Küesters and Truderung

In this section, we explain the specification and analysis of the the version they made Küesters and Truderung in [Küesters 11] of the original CCA-1B protocol.

In this section we describe the specification and analysis of the version of the CCA-1B protocol they made Küesters and Truderung in [Küesters 11]. The difference between this version and the original version is that they cannot handle the entire exclusive-or theory and because of that should transform the original `Key Translate API operator`:

$\{\text{eK}\}_{ekek1*T}$, T, $\{\text{ekek1}\}_{KM*IMP}$, $\{\text{ekek2}\}_{KM*EXP} \rightarrow \{\text{eK}\}_{(ekek2*T)}$

into these two rules:

$\{\text{eK}\}_{ekek1*T}$, T, $\{\text{ekek1}\}_{KM*IMP} \rightarrow$ transf(eK,T)
transf(eK,T), $\{\text{ekek2}\}_{\{KM*EXP\}} \rightarrow \{\text{eK}\}_{(ekek2*T)}$

5.6.1 Symbols

The symbols that we use for this version of the protocol are the same that we use for the original one (see Section 5.1.1) except the one for the `transf` function. This is specified as follows:

```
op transf : Key Tipo -> Tipo [frozen] .
```

This operator is used in `Key Translate API operator` to translate a key from encryption under one KEK (of import type) to encryption under another (of export type).

5.6.2 Algebraic properties

The algebraic properties of this protocol version are the same than in the original version (see Section 5.1.2).

5.6.3 Strand specification

The strand specification that we use for this subset of the API operators of the protocol are the same that we use for the version of the original one (see Section 5.2.3) made by Küesters and Truderung, and two new API operators, one called **PKA Symmetric Key Import**, and a second one called **Key Translate**

The specification in Maude-NPA of the first one is as follows:

```
:: nil :: ***PKA Symmetric Key Import
 [nil | -(e(PKA, eK ; T)),
        +(e(KM * T, eK)), nil]
```

And the specification in Maude-NPA of the second one is as follows:

```
:: nil :: *** Key Translate
[nil | -(e(Kek1 * T, eK)),
       -(T),
       -(e(KM * IMP, Kek1)),
       +(transf(eK, T)), nil]

    &
    :: nil :: *** Key Translate
    [nil | -(transf(eK, T)),
        -(e(KM * IMP, Kek2)),
           -(e(Kek2 * T, eK)), nil]
          [nonexec] .
```

The intruder capabilities that we use for this protocol are the same that we use for the CCA-0 protocol (see Section 5.1.3).

5.6.4 Protocol analysis

We use the same attack pattern of the CCA-0 protocol in Section 5.1.4, to prove if the recommendation works as it should.

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 0 Solutions>> 0
```

The protocol is secure against this attack and, when this protocol is analyzed with Maude-NPA, it generates a finite search state space finding no initial state for the attack pattern above.

5.7 IBM's second recommendations to avoid the attack

In order to avoid the attack of the CCA-0 protocol explained in section 5.1.4, IBM suggested another recommendation. This recommendation use the Access Control System. Users of IBMs 4758 HSM are assigned to roles that determine which commands they are allowed to execute. The goal is to prevent one single individual from having access to all the commands required to mount Bonds attack. This is enforced using access controls. IBM provide an example of the KEK transfer process involving five roles (A -E) such that no single role is able to mount the attack, as we can see in the next table

Person	Responsibilities	Commands
A	the two clear key parts, using the Random Number Generate command, and the key verification pattern (KVP) for the complete key encryption key (KEK). The First key part is given to person B, the second to person C, and the verification pattern to persons C and E.	Not Applicable
B	Enters the first key part into the destination security module.	Key Part Import 1
C	Enters the two clear key parts, using the Random Number Generate command, and the key verification pattern (KVP) for the complete key encryption key (KEK). The First key part is given to person B, the second to person C, and the verification pattern to persons C and E.	Key Part Import 3 Key Test
D	Distributes the PIN derivation key (PDK) encrypted under KEK.	Not Applicable
E	Verifies that the KEK is correct, then imports the PDK.	Key Import. Key Test

Table 5.3: Roles described by IBM in their 2nd recommendation.

The different API operators and their description for the IBM's second recommendation are as follows:

API Operator	CCA-2B	CCA-2C	CCA-2E
Encipher	X	X	X
Decipher	X	X	X
Key Export	X	X	X
Key Import	X		X
Key Part Import First	X	X	
Key Part Import Last		X	
Key Test		X	X

Table 5.4: CCA Original API operators for the IBM's second recommendation.

API Operator	CCA-2B	CCA-2C	CCA-2E
Encipher	X	X	X
Decipher	X	X	X
Key Export	X	X	X
Key Import	X		X
Key Part Import First	X	X	
Key Part Import Last		X	
KPI-First + KPI-Add/Middle		X	X

Table 5.5: CCA version of Küesters and Truderung API operators for the IBM's second recommendation.

The `Key Test` API operator command is the equivalent to the `KPI-First + KPI-Add/Middle` API operator command of the version of the protocol of Küesters and Truderung. It's important to highlight that even though "Key Part Import Last" API operator has the same name in both versions, the specification is different. At the original version, the informal specification is as follows:

Key Part Import Last
km3, km2$_{\{KM*KP*T\}}$, T $\rightarrow$ (km2 * km3)$_{\{KM*KP*T\}}$

Meanwhile for the second one, the modified version of the protocol that made

Küesters and Truderung in [Küesters 11], the informal specification is as follows:

Key Part Import Last
x, T, KM * KP * T $\rightarrow$ (x)$_{\{KM*T\}}$
x, IMP $\rightarrow$ (X * km12)$_{\{KM*IMP\}}$

To implement this recommendation, additional access control needs to be assumed, to ensure that no single role is able to perform an attack. From Sections 5.8 to the 5.10 we follow these recommendations of the CCA protocol. For this second recommendation, we specify the original CCA-2B, CCA-2C and CCA-2E and a version of CCA-2C they made Küesters and Truderung of the original specification of the CCA-2C protocol. This is the only that has different specifications of the original protocol due to their problems to handle the entire exclusive-or theory as you could see in Section 5.7.

5.8 CCA-2B

In this section following the second recommendation of IBM we implement the role of person B, who enters the first key part into the destination security module as shown in Table 5.3.

5.8.1 Symbols

The symbols that we use for this version of the protocol are the same that we use for the original one (see Section 5.1.1).

5.8.2 Algebraic properties

The algebraic properties of this protocol version are the same than in the original version (see Section 5.1.2).

5.8.3 Strand specification

The strand specification that we use for this subset of the API operators of the protocol are the same that we use for the original one (see Section 5.1.3). The intruder capabilities that we use for this protocol are the same that we use for the CCA-0 protocol (see Section 5.1.3).

5.8.4 Protocol analysis

We use the same attack pattern of the CCA-0 protocol in Section 5.1.4, but in this case NP denotes a different set of ?Never patterns? to reduce the search space (See Appendix B).

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 10 Solutions>> 0

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 7 Solutions>> 0

reduce in MAUDE-NPA : summary(7) .
result Summary: States>> 7 Solutions>> 0

reduce in MAUDE-NPA : summary(8) .
result Summary: States>> 2 Solutions>> 0

reduce in MAUDE-NPA : summary(9) .
result Summary: States>> 10 Solutions>> 0

reduce in MAUDE-NPA : summary(10) .
```

```
result Summary: States>> 6 Solutions>> 0

reduce in MAUDE-NPA : summary(11) .
result Summary: States>> 0 Solutions>> 0
```

The protocol is secure against this attack and, when this protocol is analyzed with Maude-NPA, it generates a finite search state space finding no initial state for the attack pattern above.

5.9 CCA-2C

In this Section we describe a version of the CCA protocol fulfilling IBM' second recommendation. This protocol version implements the role of person C, enters the second key part and verifies that the completed `KEK` is correct by checking the `Key Verification Pattern (KVP)`. `Key Test` verifies that the completed `KEK` is correct by checking the `key verification pattern (KVP)`, that can be specified. The informal description of this operator is as follows:

Key Test
:: nil :: [nil | +(e(KM * KP * IMP, Km1 * Km2)), nil]

5.9.1 Symbols

The symbols that we use for this version of the protocol are the same that we use for the original one (see Section 5.1.1).

5.9.2 Algebraic properties

The algebraic properties of this protocol version are the same than in the original version (see Section 5.1.2).

5.9.3 Strand specification

The strand specification that we use for this subset of the API operators of the protocol are the same that we use for the original one (see Section 5.2.3), and a new one called **Key Test**.

The intruder capabilities that we use for this protocol are the same that we use for the CCA-0 protocol (see Section 5.1.3).

5.9.4 Protocol analysis

We use the same attack pattern of the CCA-0 protocol in Section 5.1.4, but in this case NP denotes a different set of ?Never patterns? to reduce the search space (See Appendix B).

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 15 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 31 Solutions>> 0

reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 38 Solutions>> 0

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 17 Solutions>> 0

reduce in MAUDE-NPA : summary(7) .
result Summary: States>> 26 Solutions>> 0
```

At the original version of the protocol (see Section 5.1.4), the attacker just needs seven steps to find an initial state for the attack-state, so we could assume that if in this version we reached the seventh step without finding an initial state, the tool won't find a solution for this attack, so the protocol is secure.

This bound could be set much higher, but informal analysis of the tool's output, shows that the intruder never would be able to obtain any useful new terms.

At TACAS's paper [Cortier 07] they do the following assumption for the same protocol:

Person	**Bound**	**Analysed States**	**Reachable States**	**Run-Time (s)**
C	3	413	68	58.22

where they do a similar assumption to ours to arrive to the same conclusion.

5.10 CCA-2C version of Küesters and Truderung

In this section we describe the specification and analysis of the version of the CCA-2C protocol they made Küesters and Truderung in [Küesters 11].

5.10.1 Symbols

The symbols that we use for this version of the protocol are the same that we use for the original one (see Section 5.2.1).

5.10.2 Algebraic properties

The algebraic properties of this protocol version are the same than in the original version (see Section 5.2.2).

5.10.3 Strand specification

The strand specification that we use for this subset of the API operators of the protocol are the same that we use for the original one (see Section 5.2.3), and a new one called `Key Test` using the same specification that in the original version for this API operator .
The difference between this version of Küesters and Truderung [Küesters 11] and the original version is the specification of the `Key Import Last` API operator as we could see in Section 5.7. You could see the specification of the `Key Import Last` API operator in Maude-NPA in Section 5.2.3.

The intruder capabilities that we use for this protocol are the same that we use for the CCA-0 protocol (see Section 5.2.3).

5.10.4 Protocol analysis

We use the same attack pattern of the CCA-0 protocol in Section 5.2.4, to prove if the recommendation works as it should.

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 2 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 6 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 26 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 48 Solutions>> 0
```

This bound could be set much higher, but informal analysis of the tool's output, shows that the intruder never would be able to obtain any useful new terms.

At TACAS's paper [Cortier 07] they do the following assumption for the same protocol:

Person	**Bound**	**Analysed States**	**Reachable States**	**Run-Time (s)**
C	3	413	68	58.22

where they do a similar assumption to ours to arrive to the same conclusion.

5.11 CCA-2E

In this section, we implement the role of person E, that verifies that the KEK is correct, then imports the PDK. This version of the protocol uses a subset of operations of the original CCA-0 protocol (See Section 5.1), and Key Test operator (See Section 5.8).

5.11.1 Symbols

The symbols that we use for this version of the protocol are the same that we use for the original one (see Section 5.1.1).

5.11.2 Algebraic properties

The algebraic properties of this protocol version are the same than in the original version (see Section 5.1.2).

5.11.3 Strand specification

The strand specification that we use for this subset of the API operators of the protocol are the same that we use for the original one (see Section 5.1.3 to see the strand specification and see Appendix B to see all the never patterns that we use in this specification for this recommendation) and Key Test operator (See Section 5.8.3). The intruder capabilities that we use for this protocol are the same that we use for the CCA-2B protocol (see Section 5.8.3).

5.11.4 Protocol analysis

We use the same attack pattern of the CCA-0 protocol in Section 5.1.4, but in this case NP denotes a different set of ?Never patterns? to reduce the search space (See Appendix B).

The number of generated states in the different levels with Maude-NPA tool is as follows:

```
reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 9 Solutions>> 0

reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 25 Solutions>> 0
```

```
reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 57 Solutions>> 0

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 133 Solutions>> 0

reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 157 Solutions>> 0
```

As in CCA-2C protocol (see Section 5.8.4), the original version of the protocol (see Section 5.1.4), the attacker just needs seven steps to find an initial state for the attack-state, so we could assume that if in the new version we reached the seventh step without solution, the tool won't find a solution for this attack, so the protocol is secure. This bound could be set much higher, but informal analysis of the tool's output, shows that the intruder never would be able to obtain any useful new terms.

At TACAS's paper [Cortier 07] they do the following assumption for the same protocol:

Person	**Bound**	**Analysed States**	**Reachable States**	**Run-Time (s)**
E	10	54	10	0.03

where they do a similar assumption to ours to arrive to the same conclusion.

Chapter 6

Conclusions

From this work we can draw some interesting conclusions. The first one is how the Maude-NPA can work effectively and appropriate with protocols with exclusive-or operator and its properties. This has been really useful to us, since previously existing collection of protocols that handle this property was very meager, being now far more extensive. By reference to the work of Ralf Küesters and Tomasz Truderung (see [Küesters 11]), we found another interesting thing about the Maude-NPA. This has happened on the analysis of the CCA API protocol. While they, unable to handle the full theory of XOR, were working on a modified version of the protocol, the Maude-NPA tool could handle both the modified version and the full version. This has also happened in the versions that have been made based on IBM recommendations. In these, Maude-NPA could handle the full versions of these variants, while the tool used by Küesters and Truderung in his paper could not handle it and must modify some of the versions recommended by IBM to perform the analysis .

This series of protocols (see Section 5), has also been very interesting because the Maude-NPA had never previously managed API protocols, thus it was also a challenge to check whether the tool was able to work with such protocols. Note that the behavior of the tool has been really good, allowing us to implement and work with it and to check the cryptographic properties we wanted to check into it.

This work has raised many challenges that have been solved satisfactorily, such as whether it can be handled the exclusive-or property completely or if it could work with API protocols using the Maude-NPA tool.

Bibliography

[IBM 08] CCA Basic Services Reference and Guide: CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764

[Abadi 06] Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. Theoretical Computer Science 367(1-2), 2-32 (2006)

[Armando 05] Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P.H., Heám, P.C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganø, L., Vigneron, L.: The AVISPA tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281-285. Springer, Heidelberg (2005)

[Armando 04] Armando, A., Compagna, L., Lierler, Y.: SATMC: A SAT-based model checker for security protocols. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS, vol. 3229, pp. 730-733. Springer, Heidelberg (2004)

[Baudet 09] Baudet, M., Cortier, V., Delaune, S.: YAPA: A generic tool for computing intruder knowledge. In: Treinen, R. (ed.) Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009), Brasilia, Brazil. LNCS.Springer, Heidelberg (to appear, 2009)

[Boichut 04] Boichut, Y., Héam, P.-C., Kouchnarenko, O., Oehl, F.: Improvements on the Genet and Klay technique to automatically verify security protocols. In: Proceedings of Automated Verification of Infinite States Systems (AVIS 2004). ENTCS (2004)

[Bursuc 09] Bursuc, S., Comon-Lundh, H.: Protocol security and algebraic properties: decision results for a bounded number of sessions. In: Treinen,

R. (ed.) Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA2009), Brasilia, Brazil. LNCS. Springer, Heidelberg 2009

[Chevalier 08] Chevalier, Y., Rusinowitch, M.: Hierarchical combination of intruder theories. Inf.Comput. 206(2-4), 352-377 (2008)

[Ciobâca 09] Ciobâca, S., Delaune, S., Kremer, S.: Computing knowledge in security protocols under convergent equational theories. In: Schmidt, R. (ed.) Proceedings of the 22nd International Conference on Automated Deduction (CADE 2009), Montreal, Canada. LNCS (LNAI). Springer, Heidelberg 2009

[Durgin 04] Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Multiset rewriting and the complexity of bounded security. Journal of Computer Security, 677-722 (2004)

[Millen 87] Millen, S.F.J.K., Clark, S.C.: The interrogator: Protocol security analysis. IEEE Transactions on Software Engineering, 274-288 (February 1987)

[Lowe 96] Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. Software Concepts and Tools 17, 93-102 (1996)

[Meadows 96] Meadows, C.: The NRL protocol analyzer: An overview. Journal of logic program-ming 26(2), 113-131 (1996)

[Mitchell 97] Mitchell,J.,Mitchell,M.,Stern,U.:Automated analysis of cryptographic protocols using Murphi. In: IEEE Symposium on Security and Privacy. IEEE Computer Society, Los Alamitos (1997)

[Paulson 98] Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6(1-2), 85-128 (1998)

[Rusinowitch 01] Rusinowitch, M., Turuani, M.: Protocol insecurity with a finite number of sessions and composed keys is NP-complete. In: 14th IEEE Computer Security Foundations Workshop, pp. 174-190 (2001)

[Ryan 98] Ryan, P.Y.A., Schneider, S.A.: An attack on a recursive authentication protocol. a cautionary tale. Inf. Process. Lett. 65(1), 7-10 (1998)

[Stubblebine 00] Stubblebine, S., Meadows, C.: Formal characterization and automated analysis of known-pair and chosen-text attacks. IEEE Journal on Selected Areas in Communications 18(4), 571-581 (2000)

[Küesters 11] Ralf Küesters, Tomasz Truderung: Reducing Protocol Analysis with XOR to the XOR-Free Case in the Horn Theory Based Approach. J. Autom. Reasoning 46(3-4): 325-352 (2011)

[Keifgren 07] Gavin Keifgren, Graham Steel and Alan Bundy. Analysing IBMs Common Cryptographic Architecture API with a Protocol Analysis Tool. ACM Journal Name, Vol. V, No. N, May 2007

[Bond 01] Bond, M.: Attacks on cryptoprocessor transaction sets. In: Cryptographic Hardware and Embedded SystemsCHES 2001. Third International Workshop. Lecture Notes in Computer Science, vol. 2162, pp. 220234. Springer, Heidelberg (2001)

[Shoup 96] Shoup, V., Rubin, A.: Session key distribution using smart cards. In: Advances in Cryptology EUROCRYPT 96, International Conference on the Theory and Application of Cryptographic Techniques. Lecture Notes in Computer Science, vol. 1070, pp. 321331. Springer, Heidelberg 1996

[Chevalier 05] Chevalier, Y., Ralf Küesters, R., Rusinowitch, M., Turuani, M.: An NP decision procedure for protocol insecurity with XOR. Theor. Comput. Sci. 338(1-3): 247-274, 2005

[Dolev 83] D. Dolev, A. Yao. On the security of public key protocols. IEEE Transaction on Information Theory, vol. 29, no. 2, pages 198208, 1983

[Escobar 09] S. Escobar, C. Meadows, J. Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Algebraic properties. In A. Aldini, G. Barthe, R. Gorrieri, editeurs, FOSAD 2008/2009 Tutorial Lectures, volume 5705 of LNCS, pages 150. Springer, 2009.

[Escobar 09a] S. Escobar, C. Meadows and J. Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In A. Aldini, G. Barthe and R. Gorrieri, editeurs, FOSAD 2008/2009 Tutorial Lectures, volume 5705 of LNCS, pages 1-50. Springer, 2009.

[Escobar 09b] Santiago Escobar, Catherine Meadows and José Meseguer. Maude-NPA, version 1.0. University of Illinois at Urbana-Champaign, March 2009. Available at http://maude.cs.uiuc.edu/tools/Maude-NPA.

[Escobar 14] S. Escobar, C. Meadows, J. Meseguer, S. Santiago. State Space Reduction in the Maude-NRL Protocol Analyzer, In Information and Computation, to appear, 2014

[Bull 97] Bull, J.A., Otway, D.J.: The authentication protocol. Technical Report. DRA/CIS3/PROJ/ CORBA/SC/1/CSM/436-04/03, Defence Research Agency, Malvern, UK, 1997

[Bella 03] Giampaolo Bella. Inductive verification of smart card protocols, in Journal of Computer Security 11 (2003) 87132 87 IOS Press

[Cortier 07] V. Cortier, G. Keighren and G. Steel. Automatic Analysis of the Security of XOR-based Key Management Schemes, in Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)

[Lowe 96] G. Lowe, Breaking and fixing the Needham-Schroeder public-key protocol using FDR, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1996), Lecture Notes in Computer Science, Vol. 1055, Springer, Berlin, 1996, pp. 147-166

[Maude 09] M.Clavel, F.Durán, S.Eker, P.Lincoln, N.Martí-Oliet J.Meseguer C.Talcott. Maude Manual (Version 2.4), October 2008 (Revised February 2009)

[Comon-Lundh 03] H. Comon-Lundh, V. Shmatikov. Intruder Deductions, Constraint Solving and Insecurity Decision in Presence of exclusive-or. LICS 2003: 271

[Küsters 08] Küsters, R., Truderung, T.: Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach. In: Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008), pp. 129-138. ACM, New York, 2008

[Meseguer 92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science, vol. 96, no. 1, pages 73-155, 1992

[TeReSe 03] TeReSe, editor. Term rewriting systems. Cambridge University Press, Cambridge,2003

[Thati 07] P. Thati and J. Meseguer. Symbolic reachability analysis using narrowing and its application verification of cryptographic protocols, J.

Higher-Order and Symbolic Computation, 2007, volume 20, number 1-2, pages 123-160

[Escobar 12] Santiago Escobar, Ralf Sasse and José Meseguer, Folding Variant Narrowing and Optimal Variant Termination,J. Log. Algebr. Program., 2012,

[Keighren 06] Gavin Keighren. Model Checking IBM's Common Cryptographic Architecture API. Informatics Research Report 862, SCHOOL of INFORMATICS, October 2006

[DES Encryption] Data Encryption Standard (DES), Dec. 1993. Federal Information Processing Standards publication 46-2. Available online at www.itl.nist.gov.

[Meseguer 98] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, Proc. WADT-97, pages 18-61. Springer LNCS 1376, 1998.

[Escobar 06] Escobar, S., Meadows, C., Meseguer, J., 2006. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. Theor. Comput. Sci. 367 (1-2), 162–202.

[Escobar 09c] Escobar, S., Meadows, C., Meseguer, J., 2009. Maude-NPA : Cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (Eds.), FOSAD 2008/2009 Tutorial Lectures. Vol. 5705 of LNCS. Springer, pp. 1-50.

[Maude-NPA 09] Maude-NPA : Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA, version 1.0. University of Illinois at Urbana-Champaign (March 2009)

[Chen 09] Xihui Chen, Ton van Deursen, Jun Pang: Improving Automatic Verification of Security Protocols with XOR. ICFEM 2009: 107-126

[Steel 05] Graham Steel: Deduction with XOR Constraints in Security API Modelling. CADE 2005: 322-336

[Proverif 10] ProVerif: Cryptographic protocol verifier in the formal model

Appendix A

Never Patterns in CCA-0 Protocol

```
(S:StrandSet || (e(#0:Key, KM * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (PDK inI,K:IntruderKnowledge))
(S:StrandSet || (e(EXP,PDK) inI,K:IntruderKnowledge))
(S:StrandSet || (e(Km1 * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(Km2 * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(Km3 * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(EXP * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(PDK * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(PIN * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((Km1 * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((Km2 * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((KM * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((KP * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((PDK * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(#0:Key, IMP * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * Km2 * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * EXP * Km2 * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * Km1 * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * EXP * Km1 * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (#0:Msg * e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3)) inI,K:IntruderKnowledge)
(S:StrandSet || (#0:Msg * e(IMP * KM, PIN * Km1 * Km2 * Km3)) inI,K:IntruderKnowledge)
(S:StrandSet || (#0:Msg * e(IMP * KP * KM, Km1 * Km2 * #1:Key)) inI,K:IntruderKnowledge)
(S:StrandSet || (#0:Msg * e(IMP * KP * KM, Km1 * Km2)) inI,K:IntruderKnowledge)
(S:StrandSet || (#0:Msg * e(IMP * KP * KM, Km1 * Km2 * #1:Key)) inI,K:IntruderKnowledge)
(S:StrandSet || (Km1 inI,K:IntruderKnowledge))
(S:StrandSet || (Km2 inI,K:IntruderKnowledge))
(S:StrandSet || (Km3 inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * Km1 * Km2 * #0:Key) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, Km1 * Km2 * #1:Key) inI,K:IntruderKnowledge))
(S:StrandSet || ((PIN * EXP) inI,K:IntruderKnowledge))
(S:StrandSet || ((EXP * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (#0:Msg * e(#0:Key, #1:Msg)) inI,K:IntruderKnowledge)
```

```
(S:StrandSet || (e(KP * KM, #0:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(#0:Key, null) inI,K:IntruderKnowledge))
(S:StrandSet || (e(PIN * EXP * Km1 * Km2 * Km3, PDK) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KM, #0:Tipo * #1:EKey) inI,K:IntruderKnowledge))
(S:StrandSet || (e(null, PDK) inI,K:IntruderKnowledge))
```

Appendix B

Never Patterns for IBM's recommendations

```
(S:StrandSet || (e(#0:Key, KM * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (PDK inI,K:IntruderKnowledge))
(S:StrandSet || (e(EXP,PDK) inI,K:IntruderKnowledge))
(S:StrandSet || (e(Km1 * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(Km2 * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(Km3 * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(EXP * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(PDK * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(PIN * KP * KM,#1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((Km1 * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((Km2 * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((KM * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((KP * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || ((PDK * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(#0:Key, IMP * #1:Msg) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * Km2 * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * EXP * Km2 * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * Km1 * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * EXP * Km1 * Km3) inI,K:IntruderKnowledge))
(S:StrandSet || (#0:Msg * e(IMP * KM, PIN * EXP * Km1 * Km2 * Km3)) inI,K:IntruderKnowledge)
(S:StrandSet || (#0:Msg * e(IMP * KM, PIN * Km1 * Km2 * Km3)) inI,K:IntruderKnowledge)
(S:StrandSet || (#0:Msg * e(IMP * KP * KM, Km1 * Km2 * #1:Key)) inI,K:IntruderKnowledge)
(S:StrandSet || (#0:Msg * e(IMP * KP * KM, Km1 * Km2)) inI,K:IntruderKnowledge)
(S:StrandSet || (#0:Msg * e(IMP * KP * KM, Km1 * Km2 * #1:Key)) inI,K:IntruderKnowledge)
(S:StrandSet || (Km1 inI,K:IntruderKnowledge))
(S:StrandSet || (Km2 inI,K:IntruderKnowledge))
(S:StrandSet || (Km3 inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, PIN * Km1 * Km2 * #0:Key) inI,K:IntruderKnowledge))
(S:StrandSet || (e(IMP * KP * KM, Km1 * Km2 * #1:Key) inI,K:IntruderKnowledge))
(S:StrandSet || ((PIN * EXP) inI,K:IntruderKnowledge))
(S:StrandSet || ((EXP * Km3) inI,K:IntruderKnowledge))
```