

# Travaux Pratiques

## Attaque et correction du protocole à clef publique de Needham et Schroeder par Gavin Lowe

**RESUME :** On se propose dans ces travaux pratiques de reproduire la démarche de Gavin Lowe, dans la mise en évidence d'une attaque sur le protocole à clef publique de Needham et Schroeder (à l'aide de FDR, un model checker pour CSP) et dans la construction d'un protocole corrigé, invulnérable à cette attaque.<sup>(0)</sup>

### PRE-REQUIS :

**CSP (Communicating Sequential Processes)**<sup>(1)</sup> : Une connaissance de *CSP* est nécessaire pour bien mener à bout la première activité.

### DEMARCHE :

Les travaux pratiques comporteront deux activités :

**Activité I - Etude formelle du Protocole à clef publique Needham-Schroeder**<sup>(2)</sup>:

Il s'agit de réaliser une étude formelle du protocole, débouchant à l'écriture en *CSP* d'une implémentation du système et une spécification de ses règles de sécurité.

**Activité II - Analyse et correction du Protocole à clef publique Needham-Schroeder :**

Il s'agit dans un premier lieu d'utiliser *FDR*<sup>(3)</sup>, un model checker de *CSP*, pour mettre en évidence une attaque sur le protocole.

Dans un second lieu on se propose de présenter une version corrigée du dit protocole.

### OUTILS :

**FDR (Failures Divergences Refinement Checker)** : Un model checker de *CSP*, utilisé dans notre cas dans sa version graphique, sous Linux, distribution : RedHat 9.

---

<sup>(0)</sup> Voir [1] dans la bibliographie.

<sup>(1)</sup> Voir [2] dans la bibliographie.

<sup>(2)</sup> Voir [3] dans la bibliographie. On désignera par NS, l'abréviation de Needham-Schroeder.

<sup>(3)</sup> Voir [4] dans la bibliographie, pour le téléchargement de la version d'essai et de sa documentation.

## Activité I

# Etude formelle du Protocole à clef publique Needham-Schroeder

### I.0 – Objectifs :

→ Modéliser le protocole à l'aide de CSP, et obtenir :

- Une implémentation du protocole NS en CSP.
- Une spécification des propriétés d'authentification que NS doit garantir.

### I.1 – Présentation du protocole à clef publique de Needham-Schroeder :

Ce protocole a été présenté par Roger Nedham et Michael Schroeder en 1978. Il a pour but d'établir l'authentification mutuelle entre deux intervenants :

- A l'initiateur qui demande l'établissement d'une session.
- B le répondeur qui accepte (éventuellement) la demande de l'initiateur.

Ce protocole utilise la cryptographie à clef publique et suppose l'existence d'une entité tierce, digne de confiance et parfaitement authentifiée auprès des intervenants, qui distribue les clefs publiques. Il utilise aussi des aléas ou 'nonces', utilisables chacun durant une et une seule session d'authentification et servant plus tard à établir des clefs de sessions de communication.

#### I.1.1 – Présentation informelle du protocole NS :

Le protocole NS peut être décrit en 3 étapes<sup>(4)</sup> :

Message (1)	A @ B :	A . B . {N <sub>a</sub> . A} <sub>PK(B)</sub>
Message (2)	B @ A :	B . A . {N <sub>a</sub> . N <sub>b</sub> } <sub>PK(A)</sub>
Message (3)	A @ B :	A . B . {N <sub>b</sub> } <sub>PK(B)</sub>

- (1) A, l'initiateur, choisit un nonce **N<sub>a</sub>**, le concatène à son identité **A**, crypte l'ensemble avec la clef publique de **B** : **PK(B)** et envoie le résultat à **B**.
- (2) B, le répondeur, reçoit le message **{N<sub>a</sub>. A}<sub>PK(B)</sub>**, il le décrypte – et croit savoir l'identité de l'initiateur, par la même –, choisit un nonce **N<sub>b</sub>**, qu'il concatène à **N<sub>a</sub>**, crypte le tout avec la clef publique de **A** : **PK(A)** et envoie le résultat à **A**.
- (3) A reçoit le message **B . A . {N<sub>a</sub>. N<sub>b</sub>}<sub>PK(A)</sub>**, en le décryptant, il compare le nonce **Na** reçu avec le nonce **Na** qu'il a envoyé en (1) et croit authentifier **B**, ensuite il récupère le nonce **N<sub>b</sub>**, le crypte avec la clef privé de **B** : **PK(B)**, et envoie le résultat à **B**.

---

<sup>(4)</sup> Le modèle original présente 7 étapes. Dans les 4 étapes omises les intervenants récupèrent les clefs publiques de leurs homologues auprès de l'entité tierce digne de confiance. Le modèle original peut être attaqué par réutilisation de clefs publiques périmées, cette attaque n'est pas étudiée dans ce document.

B reçoit le message  $A.B.\{N_b\}_{PK(B)}$  et en le décryptant il compare le nonce  $N_b$  reçu avec le nonce  $N_b$  qu'il a envoyé en (2) et croit authentifier A.

## I.2 – Implémentation en CSP du protocole NS : (voir en annexe ns3.csp)

On va supposer l'existence de 4 ensembles basiques :

- *Initiator* : L'ensemble des initiateurs.
- *Responder* : L'ensemble des répondeurs.
- *Key* : L'ensemble des clefs publiques.
- *Nonce* : L'ensemble des nonces.

### I.2.1 – Ensemble des messages du protocole NS :

On définit l'ensemble des messages *MSG* du protocole NS comme suit :

$$MSG\ 1 \circ \{ Msg1.a.b.Encrypt.k.n_a.a' \mid a,a'\hat{I}\ Initiator, b\hat{I}\ Responder, k\hat{I}\ Key, n_a\hat{I}\ Nonce \}$$

$$MSG\ 2 \circ \{ Msg2.b.b.Encrypt.k.n_a.n_b \mid a\hat{I}\ Initiator, b\hat{I}\ Responder, k\hat{I}\ Key, n_a,n_b\hat{I}\ Nonce \}$$

$$MSG\ 3 \circ \{ Msg3.a.b.Encrypt.k.n_b \mid a\hat{I}\ Initiator, b\hat{I}\ Responder, k\hat{I}\ Key, n_b\hat{I}\ Nonce \}$$

$$MSG \circ MSG1 \hat{E} MSG2 \hat{E} MSG3$$

Où la notation *Encrypt.k.m* représente le cryptogramme :  $\{m\}_k$  et où l'événement *comm.Msg1.A.B.Encrypt.k\_b.n\_a.A* représente le **message (1)** décrit plus haut.

### I.2.2 – Les canaux du protocole NS :

Ceci nous amènes à la définition des canaux :

- *comm* : le canal des communications standards.
- *fake* : le canal des émissions avec usurpation d'identité.
- *intercept* : le canal des interceptions non permises de messages.

Nous allons aussi introduire la notion d'intrusion ou d'attaque par usurpation d'identité ou par interception de messages.

*Channel comm, fake, intercept : MSG*

Nous définissons aussi des canaux constituant l'interface externe du protocole :

- *user* : l'événement *user.a.b* représente une demande d'un user ayant a comme initiateur, pour établir une communication avec le répondeur b.
- *session* : l'événement *session.a.b* représente une session apparemment réussie du protocole entre l'initiateur a et le répondeur b.
- *I\_running* : l'événement *I\_running.a.b* représente l'état où l'initiateur a croit qu'il est entrain d'exécuter une session du protocole avec le répondeur b.

- $R\_running$ : l'événement  $R\_running.a.b$  représente l'état où le répondeur  $b$  croit qu'il est entrain d'exécuter une session du protocole avec l'initiateur  $a$ .
- $I\_commit$ : l'événement  $I\_commit.a.b$  représente l'état où l'initiateur  $a$  croit qu'il a authentifié le répondeur  $b$ .
- $R\_commit$ : l'événement  $R\_commit.a.b$  représente l'état où le répondeur  $b$  croit qu'il a authentifié le répondeur  $a$ .

$Channel\ user, session, I\_running, R\_running, I\_commit, R\_commit : Initiator.Responder$

### I.2.3 – Les processus du protocole NS (sans intrus) :

#### I.2.3.1 – Le processus de l'initiateur :

$INITIATOR(a, n_a) \circ$

```

user.a.b @ I_running.a.b @
comm.Msg1.a.b.Encrypt.key(b).n_a.a @
comm.Msg2.b.a.Encrypt.key(a).n'_a.n_b @
if    n_a = n'_a
then  comm.Msg3.a.b.Encrypt.key(b).n_b @
      I_commit.a.b @ session.a.b @ Skip
else  Stop

```

#### I.2.3.2 – Le processus du répondeur:

$RESPONDER(b, n_b) \circ$

```

comm.Msg1.a.b.Encrypt.key(b).n_a.a @
R_running.a.b @
comm.Msg2.b.a.Encrypt.key(a).n_a.n_b @
comm.Msg3.a.b.Encrypt.key(b).n'_b @
if    n_b = n'_b
then  R_commit.a.b @ session.a.b @ Skip
else  Stop

```

### I.2.4 – Les processus du protocole NS (avec intrus) :

#### I.2.4.1 – Le processus de l'initiateur :<sup>(5)</sup>

$INITIATOR1(a, n_a) \circ$

```

INITIATOR(a, n_a)
[[comm.Msg1 → comm.Msg1, comm.Msg1 → intercept.Msg1,
 comm.Msg2 → comm.Msg2, comm.Msg2 → fake.Msg2,
 comm.Msg3 → comm.Msg3, comm.Msg3 → intercept.Msg3]]

```

<sup>(5)</sup> On a renommé  $INITIATOR$ , dans le sens que  $INITIATOR1$  est le processus qui peut exécuter soit  $comm.Msg1$  soit  $intercept.Msg1$ , quand  $INITIATOR$  exécute un  $comm.Msg1$  correspondant ...

### I.2.4.2 – Le processus du répondeur :

$$\begin{aligned} \text{RESPONDER1}(b, n_i) \circ \quad & \text{RESPONDER}(b, n_i) \\ & [[\text{comm.Msg1} \rightarrow \text{comm.Msg1}, \text{comm.Msg1} \rightarrow \text{fake.Msg1}, \\ & \text{comm.Msg2} \rightarrow \text{comm.Msg2}, \text{comm.Msg2} \rightarrow \text{intercept.Msg2}, \\ & \text{comm.Msg3} \rightarrow \text{comm.Msg3}, \text{comm.Msg3} \rightarrow \text{fake.Msg3}]] \end{aligned}$$

### I.2.4.3 – Le processus de l'intrus :<sup>(6)</sup>

Un des moyens de cerner l'état d'un intrus est de paramétrer le savoir qu'il a acquis :  
 $m1s, m2s$  et  $m3s$  représentent respectivement les ensembles des messages 1, 2 et 3 qu'il n'a pas su décrypter et  $ns$  l'ensemble des nonces qu'il connaît.

$$\begin{aligned} I(m1s, m2s, m3s, ns) \circ \quad & \text{comm.Msg1.a.b.Encrypt.k.n.a}' \textcircled{R} \\ & \text{if } k = K_i \\ & \text{then } I(m1s, m2s, m3s, ns \hat{E}\{n\}) \\ & \text{else } I(m1s \hat{E}\{Encrypt.k.n.a'\}, m2s, m3s, ns) \\ \dot{y} \text{ intercept.Msg1.a.b.Encrypt.k.n.a}' \textcircled{R} \\ & \text{if } k = K_i \\ & \text{then } I(m1s, m2s, m3s, ns \hat{E}\{n\}) \\ & \text{else } I(m1s \hat{E}\{Encrypt.k.n.a'\}, m2s, m3s, ns) \\ \dot{y} \text{ comm.Msg2.b.a.Encrypt.k.n.n}' \textcircled{R} \\ & \text{if } k = K_i \\ & \text{then } I(m1s, m2s, m3s, ns \hat{E}\{n, n'\}) \\ & \text{else } I(m1s, m2s \hat{E}\{Encrypt.k.n.n'\}, m3s, ns) \\ \dot{y} \text{ intercept.Msg2.b.a.Encrypt.k.n.n}' \textcircled{R} \\ & \text{if } k = K_i \\ & \text{then } I(m1s, m2s, m3s, ns \hat{E}\{n, n'\}) \\ & \text{else } I(m1s, m2s \hat{E}\{Encrypt.k.n.n'\}, m3s, ns) \\ \dot{y} \text{ comm.Msg3.a.b.Encrypt.k.n} \textcircled{R} \\ & \text{if } k = K_i \\ & \text{then } I(m1s, m2s, m3s, ns \hat{E}\{n\}) \\ & \text{else } I(m1s, m2s, m3s \hat{E}\{Encrypt.k.n\}, ns) \\ \dot{y} \text{ intercept.Msg3.a.b.Encrypt.k.n} \textcircled{R} \\ & \text{if } k = K_i \\ & \text{then } I(m1s, m2s, m3s, ns \hat{E}\{n\}) \\ & \text{else } I(m1s, m2s, m3s \hat{E}\{Encrypt.k.n\}, ns) \\ \dot{y} \text{ fake.Msg1.a.b.m:m1s} \textcircled{R} \quad & I(m1s, m2s, m3s, ns) \\ \dot{y} \text{ fake.Msg2.b.a.m:m1s} \textcircled{R} \quad & I(m1s, m2s, m3s, ns) \\ \dot{y} \text{ fake.Msg3.a.b.m:m1s} \textcircled{R} \quad & I(m1s, m2s, m3s, ns) \\ \dot{y} \text{ fake.Msg1.a.b.Encrypt.k.n:ns.a}' \textcircled{R} \quad & I(m1s, m2s, m3s, ns) \\ \dot{y} \text{ fake.Msg2.b.a.Encrypt.k.n:ns.n':ns} \textcircled{R} \quad & I(m1s, m2s, m3s, ns) \\ \dot{y} \text{ fake.Msg3.a.b.Encrypt.k.n:ns} \textcircled{R} \quad & I(m1s, m2s, m3s, ns) \end{aligned}$$

<sup>(6)</sup>  $\dot{y}$  : L'opérateur choix général.  $P \dot{y} Q$  est le processus qui commence par l'action possibles parmi la première action de  $P$  et celle de  $Q$ . Si les deux sont possibles le choix est non déterministe.

On considère alors un intrus ayant pour connaissance juste un nonce  $N_i$  de départ.

$INTRUDER \circ I(\mathcal{A}, \mathcal{A}, \mathcal{A}, \{N_i\})$

### I.2.5 – Un système CSP de NS avec intrus :<sup>(7)</sup>

$AGENTS \circ INITIATOR1 \parallel \{ |comm, session| \} \parallel RESPONDER1$   
 $SYSTEM \circ AGENTS \parallel \{ |fake, comm, intercept| \} \parallel INTRUDER$

## I.3 – Une spécification des propriétés d'authentification de NS :

Le bon sens stipule qu'un initiateur A conclut à une session réussie avec un répondeur B, alors forcément B a participé en tant que répondeur à une exécution du protocole : ceci signifie qu'un événement  $I\_commit.A.B$  doit obligatoirement être précédé d'un événement  $R\_running.A.B$ . Quant aux autres événements ils peuvent intervenir dans un ordre aléatoire.

$AR_0 \circ R\_running.A.B \textcircled{R} I\_commit.A.B \textcircled{R} AR_0$   
 $A_1 \circ \{ |R\_running, I\_commit| \}$   
 $AR \circ AR_0 \parallel \parallel RUN(\mathcal{S} \setminus A_1)^{(8)}$

Où  $\mathcal{S}$  représente l'ensemble de tous les événements :

$\mathcal{S} \circ \{ |comm, fake, intercept, user, session, I\_running, R\_running, I\_commit, R\_commit| \}$

Le même raisonnement est possible en considérant qu'un répondeur B ne peut conclure à une session réussie avec A, que si A a déjà participé à une exécution du protocole en tant qu'initiateur avec B. Autrement dit un événement  $R\_commit.A.B$  doit obligatoirement être précédé d'un événement  $I\_running.A.B$  dans l'exécution ; les autres événements peuvent intervenir dans un ordre aléatoire.

$AI_0 \circ I\_running.A.B \textcircled{R} R\_commit.A.B \textcircled{R} AI_0$   
 $A_2 \circ \{ |I\_running, R\_commit| \}$   
 $AI \circ AI_0 \parallel \parallel RUN(\mathcal{S} \setminus A_2)$

## I.4 – Résumé :

Dans cette activité nous avons :

- Présenté le protocole NS d'un manière informelle.
- Fourni une description formelle du protocole NS à l'aide de CSP, consistant en :
  - Une implémentation :  $SYSTEM$ .
  - Une spécification :  $AI$  ou  $AR$  de certaines de ses propriétés d'authentification.

<sup>(7)</sup> La notation employée diffère un peu de celle introduite par Hoare. D'une part on note  $P \parallel [A] \parallel Q$  la composition en parallèle des processus  $P$  et  $Q$  avec synchronisation sur l'ensemble des événements A. D'autre part on note  $\{ |c_1, c_2, \dots, c_n| \}$  l'ensemble des événements s'exécutant sur les canaux  $c_1, c_2, \dots, c_n$

<sup>(8)</sup>  $\parallel \parallel$  : Il s'agit de l'opérateur 'interleaving' (traduire en intercalage) : une mise en parallèle non synchronisée et non-déterministe.

## Activité II

# Analyse et correction du Protocole à clef publique Needham-Schroeder

### II.0 – Objectifs :

- Mettre en évidence une attaque par usurpation d'identité, sur le protocole NS.
- Proposer une correction pour la brèche observé.

### II.1 – Présentation de FDR (Failures Divergences Refinement Checker):

C'est un outil de model-checking pour une machine d'état., fondé sur la théorie de la concurrence introduite par Hoare et basée sur CSP.

Trois approches sont possibles avec FDR, l'apurement par traces, l'apurement par défaillances et l'apurement par défaillances-divergences.

Dans le cadre de cette étude la première approche a été adoptée, en voici une brève description :

#### L'apurement par traces :

On considère deux processus P et Q et on définit la relation d'apurement par traces par :

$$P \subseteq_T Q \quad \Leftrightarrow \quad \text{Traces}(Q) \subseteq \text{Traces}(P)$$

Nota Bene: Si on considère que P est une spécification qui représente les états corrects d'un système, et que Q est une implémentation de ce système, alors dire que  $P \dot{\subseteq}_T Q$ , revient à dire que Q est une implémentation correcte.

Autrement dit, pour effectuer l'analyse d'un protocole avec FDR, avec l'approche par apurement par traces, on doit fournir en entrée deux processus :

- Une implémentation Q, décrivant le protocole.
- Une spécification P, décrivant les états jugés corrects du protocole.

En sortie FDR doit être en mesure de renvoyer un ensemble de traces  $T_{\text{attaques}} = \{\text{traces } t \mid t \in \text{Traces}(Q) \text{ et } t \notin \text{Traces}(P)\}$  tel que :

- $T_{\text{attaques}} = \emptyset$ , auquel cas Q est un apurement par traces de P, ce qui revient à dire que P est une implémentation correcte, selon Q.
- $T_{\text{attaques}} \neq \emptyset$ , auquel cas Q n'est pas un apurement par traces de P, ce qui revient à dire que P est une implémentation incorrecte, selon Q.

## II.2 – Analyse du protocole NS à l'aide de FDR :

### II.2.1 – La réponse de FDR :

FDR a trouvé une attaque sur le protocole NS, en renvoyant la trace suivante :

```
<user.A.I, I_Running.A.I,  
  intercept.Msg1.A.I.Encrypt.Ki.Na.A,  
  fake.Msg1.A.B.Encrypt.Kb.Na.A,  
  intercept.Msg2.B.A.Encrypt.Ka.Na.Nb,  
  fake.Msg2.I.A.Encrypt.Ka.Na.Nb,  
  intercept.Msg3.A.I.Encrypt.Ki.Nb,  
  fake.Msg3.A.B.Encrypt.Ka.Nb>
```

En effet le système peut déclencher l'événement R\_commit.A.B, alors qu'aucun événement I\_running.A.B n'a été déclenché auparavant : SYSTEM n'est pas un apurement par traces de AI.

### II.2.2 – Interprétation de la réponse de FDR :

On peut réécrire cette attaque comme l'intercalage de deux exécutions de NS, qu'on désignera par  $\alpha$  et  $\beta$ .

Dans l'exécution  $\alpha$ , A essaie d'établir une session avec I, tandis que dans l'exécution  $\beta$ , I se fait passer pour A, pour mener une fausse session avec B.

On notera, par exemple  $\beta.2$  pour représenter le 2<sup>me</sup> message de l'exécution  $\beta$  et I(A) pour représenter l'intrus I se faisant passer pour A :

Message (a.1)	A @ I :	A .I. {N <sub>a</sub> .A} <sub>PK(I)</sub>
Message (b.1)	I(A) @ B :	A .B. {N <sub>a</sub> .A} <sub>PK(B)</sub>
Message (b.2)	B @ I(A) :	B.A. {N <sub>a</sub> .N <sub>b</sub> } <sub>PK(A)</sub>
Message (a.2)	I @ A :	I.A. {N <sub>a</sub> .N <sub>b</sub> } <sub>PK(A)</sub>
Message (a.3)	A @ I :	A.I. {N <sub>b</sub> } <sub>PK(I)</sub>
Message (b.3)	I(A) @ B :	A.B. {N <sub>b</sub> } <sub>PK(B)</sub>

Message (a.1) : A essaie d'établir une session avec I, en envoyant le nonce N<sub>a</sub> crypté avec la clef publique de I.

Message (b.1) : I se fait passer pour A et essaie d'établir une session avec B, en lui envoyant le nonce N<sub>a</sub>, récupéré dans Message (a.1).

Message (b.2) : B répond à A en rajoutant un nouveau nonce N<sub>b</sub>.

Message (a.2) : I ne sachant pas décrypter le message de B – car crypté avec la clef de A – le réachemine vers A, pour l'utiliser comme un oracle ...

*Message (a.3) :* A croit que I entame la deuxième étape du protocole et décrypte le message, y puise le nonce  $N_b$  et le renvoie à I, crypté par la clef publique de ce dernier.

*Message (b.3) :* Cette fois I, peut décrypter le nonce  $N_b$  et le renvoie à B, qui croit qu'il vient de mener une session réussie avec A.

### II.3 – Correction du protocole NS :

Une lecture détaillée de l'attaque, permet d'en repérer la pierre d'angle.

Quand A reçoit le message  $\alpha.2$  de I, il ne se doute pas que ce message provient en réalité de B ( message  $\beta.2$ ) et que I s'est contenté de le réacheminer ...

Si on modifiait le protocole NS, de telle sorte qu'on inclut l'identité de l'émetteur dans le deuxième message, l'attaque ne tiendrait plus.

La version revue du protocole NS, porte le nom du protocole NSL comme Needham-Schroeder-Lowe, en voici une description informelle :

<i>Message (1)</i>	$A \text{ @ } B :$	$A . B . \{ N_a . A \}_{PK(B)}$
<i>Message (2)</i>	$B \text{ @ } A :$	$B . A . \{ N_a . N_b . B \}_{PK(A)}$
<i>Message (3)</i>	$A \text{ @ } B :$	$A . B . \{ N_b \}_{PK(B)}$

Une fois réadaptée, la nouvelle implémentation du protocole NSL, s'avère être un apurement par traces de la spécification *AI*.

### II.4 – Epilogue :

L'analyse et la correction du protocole NS, dans ces travaux pratiques ont été restreintes au cas où le protocole s'exécuterait entre un unique initiateur et un unique répondeur, disposant chacun d'un seul nonce.

Gavin Lowe a poussé l'étude en montrant que toute attaque sur le protocole NS ( avec plusieurs sessions et plusieurs agents ), se ramène à une attaque sur ce système simplifié.

Ce complément d'étude pourrait faire le sujet d'une séance de travaux dirigés.

# WEBLIOGRAPHIE

- [1] “*Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR (1996)*”, Gavin Lowe  
<http://citeseer.ist.psu.edu/lowe96breaking.html>
  
- [2] “*Communicating Sequential Processes*”, C.A.R Hoare  
<http://www.usingcsp.com/cspbook.pdf>
  
- [3] “*Using encryption for authentication in large networks of computers*”, Roger Needham & Michael Schroeder
  
- [4] Formal Systems.  
[http://www.fsel.com/fdr2\\_download.html](http://www.fsel.com/fdr2_download.html)

# ANNEXE

## Ns3.csp : Le fichier csp utilisé par Gavin Low pour modéliser le protocole NS

```
-- CSP script produced using Casper version 1.3
-- -- Needham Schroeder Public Key Protocol, 3 message version
--
-- #Free variables
--
-- A, B : Agent
-- na, nb : Nonce
-- PK : Agent -> PublicKey
-- SK : Agent -> SecretKey
-- InverseKeys = (PK, SK)
--
-- #Processes
--
-- INITIATOR(A, na) knows PK, SK(A)
-- RESPONDER(B, nb) knows PK, SK(B)
--
-- #Protocol description
--
-- 0.   -> A : B
-- 1.   A -> B : {na, A}{PK(B)}
-- 2.   B -> A : {na, nb}{PK(A)}
-- 3.   A -> B : {nb}{PK(B)}
--
-- #Specification
--
-- Secret(A, na, [B])
-- Secret(B, nb, [A])
-- Agreement(A, B, [na, nb])
-- Agreement(B, A, [na, nb])
--
-- #Actual variables
--
-- Alice, Bob, Mallory : Agent
-- Na, Nb, Nm : Nonce
--
-- #Functions
--
-- symbolic PK, SK
--
-- #System
--
-- INITIATOR(Alice, Na)
-- RESPONDER(Bob, Nb)
--
-- #Intruder Information
--
-- Intruder = Mallory
-- IntruderKnowledge = {Alice, Bob, Mallory, Nm, PK, SK(Mallory)}
--
datatype Encryption =
  Alice | Bob | Mallory | Na | Nb | Nm | Garbage | PK_.Agent | SK_.Agent |
  Sq.Seq(Encryption) | Encrypt.(ALL_KEYS, Seq(Encryption)) |
  Hash.(HashFunction, Seq(Encryption)) | Xor.(Encryption, Encryption)

ALL_KEYS = PublicKey

HashFunction = {}
ATOM = {Alice, Bob, Mallory, Na, Nb, Nm, Garbage}

encrypt(m_, k_) = Encrypt.(k_, m_)
decrypt(Encrypt.(k1_, m_), k_) = if k_ == inverse(k1_) then m_ else Garbage
decrypt(_, _) = Garbage
decryptable(Encrypt.(k1_, m_), k_) = k_ == inverse(k1_)
decryptable(_, _) = false
nth(ms_, n_) = if n_ == 1 then head(ms_) else nth(tail(ms_), n_ - 1)

addGarbage_(S_) =
  if S_=={} then {Garbage}
  else Union({S_}, {Garbage | Encrypt._ <- S_},
            {Garbage | Hash._ <- S_},
            {Garbage | Xor._ <- S_})

-- Types in actual system
```

```

Agent = {Alice, Bob, Mallory}
Nonce = {Na, Nb, Nm}
PublicKey = {PK(arg_1_) | arg_1_ <- Agent}
SecretKey = {SK(arg_1_) | arg_1_ <- Agent}

inverse(PK_. arg_) = SK_. arg_
inverse(SK_. arg_) = PK_. arg_

PK(arg_1_) = PK_. (arg_1_)
SK(arg_1_) = SK_. (arg_1_)

datatype Labels =
  Msg1 | Msg2 | Msg3 | Env0

INPUT_INT_MSG3_BODY =
  {(Msg3, Encrypt.(PK(B), <nb>), <na>) |
   na <- Nonce, B <- Agent, nb <- Nonce}
INPUT_INT_MSG1_BODY =
  {(Msg1, Encrypt.(PK(B), <na, A>), <>) |
   A <- Agent, B <- Agent, na <- Nonce}
INPUT_INT_MSG2_BODY =
  {(Msg2, Encrypt.(PK(A), <na, nb>), <>) |
   A <- Agent, na <- Nonce, nb <- Nonce}

INPUT_INT_MSG_BODY =
  Uni on({INPUT_INT_MSG1_BODY, INPUT_INT_MSG2_BODY, INPUT_INT_MSG3_BODY})

OUTPUT_INT_MSG3_BODY =
  {(Msg3, Encrypt.(PK(B), <nb>), <na>) |
   na <- Nonce, B <- Agent, nb <- Nonce}

OUTPUT_INT_MSG1_BODY =
  {(Msg1, Encrypt.(PK(B), <na, A>), <>) |
   A <- Agent, B <- Agent, na <- Nonce}
OUTPUT_INT_MSG2_BODY =
  {(Msg2, Encrypt.(PK(A), <na, nb>), <>) |
   A <- Agent, na <- Nonce, nb <- Nonce}
OUTPUT_INT_MSG_BODY =
  Uni on({
    OUTPUT_INT_MSG1_BODY,
    OUTPUT_INT_MSG2_BODY,
    OUTPUT_INT_MSG3_BODY
  })

INPUT_MSG1_BODY = {rmb(m) | m <- INPUT_INT_MSG1_BODY}
INPUT_MSG2_BODY = {rmb(m) | m <- INPUT_INT_MSG2_BODY}
INPUT_MSG3_BODY = {rmb(m) | m <- INPUT_INT_MSG3_BODY}
OUTPUT_MSG1_BODY = {rmb(m) | m <- OUTPUT_INT_MSG1_BODY}
OUTPUT_MSG2_BODY = {rmb(m) | m <- OUTPUT_INT_MSG2_BODY}
OUTPUT_MSG3_BODY = {rmb(m) | m <- OUTPUT_INT_MSG3_BODY}
INPUT_MSG_BODY = Uni on({INPUT_MSG1_BODY, INPUT_MSG2_BODY, INPUT_MSG3_BODY})

OUTPUT_MSG_BODY = Uni on({OUTPUT_MSG1_BODY, OUTPUT_MSG2_BODY, OUTPUT_MSG3_BODY})

MSG_BODY = uni on(INPUT_MSG_BODY, OUTPUT_MSG_BODY)

ENVMSGO_BODY =
  {(Env0, B, <>) |
   B <- Agent}

ENVMSG_BODY = ENVMSGO_BODY

SenderType ((Msg1, _, _)) = Agent
SenderType ((Msg2, _, _)) = Agent
SenderType ((Msg3, _, _)) = Agent

ReceiverType((Msg1, _, _)) = Agent
ReceiverType((Msg2, _, _)) = Agent
ReceiverType((Msg3, _, _)) = Agent

ALL_PRINCIPALS = Agent

channel input1: ALL_PRINCIPALS. ALL_PRINCIPALS. INPUT_INT_MSG_BODY
channel output1: ALL_PRINCIPALS. ALL_PRINCIPALS. OUTPUT_INT_MSG_BODY
channel fake: ALL_PRINCIPALS. ALL_PRINCIPALS. INPUT_MSG_BODY
channel intercept: ALL_PRINCIPALS. ALL_PRINCIPALS. OUTPUT_MSG_BODY
channel env : ALL_PRINCIPALS. ENVMSG_BODY

datatype ROLE = INITIATOR_role | RESPONDER_role

ALL_SECRETS_0 = Nonce
ALL_SECRETS = addGarbage_(ALL_SECRETS_0)

datatype Signal =
  Claim_Secret. ALL_PRINCIPALS. ALL_SECRETS. Set(ALL_PRINCIPALS) |

```

```

Running1. ROLE. ALL_PRINCIPALS. ALL_PRINCIPALS. Nonce. Nonce |
Commi t1. ROLE. ALL_PRINCIPALS. ALL_PRINCIPALS. Nonce. Nonce |
Running2. ROLE. ALL_PRINCIPALS. ALL_PRINCIPALS. Nonce. Nonce |
Commi t2. ROLE. ALL_PRINCIPALS. ALL_PRINCIPALS. Nonce. Nonce

```

channel signal : Signal

-- Definitions of agents

```

INITIATOR_0(A, na) =
  [] B : Agent @ env. A. (Env0, B, <>) ->
  output1. A. B. (Msg1, Encrypt. (PK(B), <na, A>), <>) ->
  [] nb : Nonce @ input1. B. A. (Msg2, Encrypt. (PK(A), <na, nb>), <>) ->
  output1. A. B. (Msg3, Encrypt. (PK(B), <nb>), <na>) ->
  SKIP

```

```

INITIATOR(A, na) =
  INITIATOR_0(A, na)
  [[input1. B. A. m_ <- fake. B. A. rmb(m_) |
   B <- Agent, m_ <- INPUT_INT_MSG2_BODY]]
  [[output1. A. B. m_ <- intercept. A. B. rmb(m_) |
   B <- Agent, m_ <- OUTPUT_INT_MSG1_BODY]]
  [[output1. A. B. m_ <- intercept. A. B. rmb(m_) |
   B <- Agent, m_ <- OUTPUT_INT_MSG3_BODY]]

```

```

RESPONDER_0(B, nb) =
  [] A : Agent @ [] na : Nonce @
  input1. A. B. (Msg1, Encrypt. (PK(B), <na, A>), <>) ->
  output1. B. A. (Msg2, Encrypt. (PK(A), <na, nb>), <>) ->
  input1. A. B. (Msg3, Encrypt. (PK(B), <nb>), <na>) ->
  SKIP

```

```

RESPONDER(B, nb) =
  RESPONDER_0(B, nb)
  [[input1. A. B. m_ <- fake. A. B. rmb(m_) |
   A <- Agent, m_ <- INPUT_INT_MSG1_BODY]]
  [[input1. A. B. m_ <- fake. A. B. rmb(m_) |
   A <- Agent, m_ <- INPUT_INT_MSG3_BODY]]
  [[output1. B. A. m_ <- intercept. B. A. rmb(m_) |
   A <- Agent, m_ <- OUTPUT_INT_MSG2_BODY]]

```

-- Facts and deductions

```

Fact_1 =
  Union({
    {Garbage},
    Nonce,
    Agent,
    PublicKey,
    {Encrypt. (PK(B), <na, A>) |
     A <- Agent, B <- Agent, na <- Nonce},
    {Encrypt. (PK(A), <na, nb>) |
     A <- Agent, na <- Nonce, nb <- Nonce},
    {Encrypt. (PK(B), <nb>) |
     B <- Agent, nb <- Nonce}
  })

```

laws = {(Garbage, Garbage)}

```

external mtransclose
renaming = mtransclose(laws, Fact_1)

```

```

external relational_inverse_image
external relational_image
ren = relational_inverse_image(renaming)

```

```

-- renaming for facts
applyRenaming0(a_) =
  let S_ = ren(a_)
  within if card(S_)==0 then a_ else elsing(S_)

```

elsing({x\_}) = x\_

```

-- renaming for events
applyRenaming(Sq.ms_) =
  if member(Sq.ms_, Fact_1) then applyRenaming0(Sq.ms_)
  else Sq.<applyRenaming0(m_) | m_ <- ms_>
applyRenaming(a_) = applyRenaming0(a_)

```

```

rmb((l_, m_, extras_) =
(l_, applyRenaming(m_), <applyRenaming(e_) | e_ <- extras_>)

```

domain = {a\_ | (\_, a\_) <- renaming}

```

applyRenamingToSet(X_) =
  union({elsing(ren(a_)) | a_ <- inter(X_, domain)}, diff(X_, domain))

```

```

applyRenamingToDeductions(S_) =
  {(applyRenamingToSet(X_) | (f_, X_) <- S_)
-- Intruder's knowledge
unSq_(Sq, ms_) = set(ms_)
unSq_(m_) = {m_}
IK0 = {Alice, Bob, Mallory, Nm, SK(Mallory), Garbage}
unknown_(S_) = diff(S_, IK0)
-- Intruder's deductions
Deductions_0 =
  Union({SqDeductions, UnSqDeductions,
        EncryptionDeductions, DecryptionDeductions,
        VernEncDeductions, VernDecDeductions,
        UserDeductions, FnAppDeductions, HashDeductions})
SqDeductions =
  {(Sq, fs_, unknown_(set(fs_))) | Sq, fs_ <- Fact_1}
UnSqDeductions =
  {(f_, unknown_({Sq, fs_})) | Sq, fs_ <- Fact_1, f_ <- unknown_(set(fs_))}
EncryptionDeductions =
  {(Encrypt.(k_, fs_), unknown_(union({k_}, set(fs_))) |
   Encrypt.(k_, fs_) <- Fact_1}
DecryptionDeductions =
  {(f_, unknown_({Encrypt.(k_, fs_), inverse(k_)}) |
   Encrypt.(k_, fs_) <- Fact_1, f_ <- unknown_(set(fs_))}
VernEncDeductions =
  {(Xor.(m1_, m2_), unknown_(union(unSq_(m1_), unSq_(m2_))) |
   Xor.(m1_, m2_) <- Fact_1}
VernDecDeductions =
  {(m1_, union(unknown_(unSq_(m2_)), {Xor.(m1_, m2_)}) |
   Xor.(m1_, m2_) <- Fact_1, m1_ <- unSq_(m1_)}
UserDeductions = {}
FnAppDeductions =
  {(PK_.arg_1_, unknown_({arg_1_})) |
   PK_.arg_1_ <- Fact_1}
HashDeductions = {(Hash.(f_, ms_), set(ms_)) | Hash.(f_, ms_) <- Fact_1}
components_((_, Sq, ms_, _)) =
  if member(Sq, ms_, Fact_1) then {Sq, ms_} else set(ms_)
components_((_, m_, _)) = {m_}
-- close up knowledge and deductions
subset(A_, B_) = inter(A_, B_) == A_
Seeable_ = Union({unknown_(components_(m_)) | m_ <- MSG_BODY})
Close_(IK_, ded_, fact_) =
  let IK1_ =
    union(IK_, {f_ | (f_, fs_) <- ded_, subset(fs_, IK_)})
    ded1_ =
      {(f_, fs_) | (f_, fs_) <- ded_, not (member(f_, IK_)),
        subset(fs_, fact_)}
    fact1_ = Union({IK_, {f_ | (f_, fs_) <- ded_}, Seeable_})
  within
  if card(IK_)==card(IK1_) and card(ded_)==card(ded1_)
  and card(fact_)==card(fact1_)
  then (IK_, {(f_, diff(fs_, IK_)) | (f_, fs_) <- ded_}, fact_)
  else Close_(IK1_, ded1_, fact1_)
Deductions_1 = {(f_, fs_) | (f_, fs_) <- Deductions_0,
  not (member(f_, fs_))}
(IK1, Deductions, KnowableFact) =
  Close_(applyRenamingToSet(IK0),
        applyRenamingToDeductions(Deductions_1),
        applyRenamingToSet(Fact_1))
print IK1
print KnowableFact
print Deductions
-- The intruder
second_((_, m_, _)) = m_

```

```

INTRUDER_MSG_BODY = {second_(m_) | m_ <- MSG_BODY}
dummyDeds = {(Garbage, {Garbage})}
Deductions' = if Deductions=={} then dummyDeds else Deductions
-- Don't you hate hacks like this?
channel leak : addGarbage(ALL_SECRETS)
channel hear, say : INTRUDER_MSG_BODY
channel infer : Deductions'

IGNORANT(f_, ms_, fss_, ds_) =
  hear?m_:ms_ -> KNOWS(f_, ms_, ds_)
  []
  ([] fs_ : fss_ @ infer. (f_, fs_) -> KNOWS(f_, ms_, ds_))

KNOWS(f_, ms_, ds_) =
  hear?m_:ms_ -> KNOWS(f_, ms_, ds_)
  []
  say?m_:ms_ -> KNOWS(f_, ms_, ds_)
  []
  infer?(f1_, fs_) : ds_ -> KNOWS(f_, ms_, ds_)
  []
  member(f_, ALL_SECRETS) & leak.f_ -> KNOWS(f_, ms_, ds_)

f_ms_fss_ds_s =
  let rid_ = relational_image(Deductions)
  within {(f_,
    {m_ | m_ <- INTRUDER_MSG_BODY, member(f_, unSq_(m_))},
    rid_(f_),
    {x_ | x_@@(f_, fs_) <- Deductions, member(f_, fs_)}) |
    f_ <- diff(KnowableFact, IK1)}

AlphaL(f_, ms_, fss_, ds_) =
  Union({(if member(f_, ALL_SECRETS) then {leak.f_} else {}),
    {hear.m_, say.m_ | m_ <- ms_},
    {infer.(f_, fs_) | fss_ <- fss_},
    {infer.(f1_, fs_) | (f1_, fs_) <- ds_}})

transparent chase

INTRUDER_0 =
  ([] (f_, ms_, fss_, ds_) : f_ms_fss_ds_s @
    [AlphaL(f_, ms_, fss_, ds_)] IGNORANT(f_, ms_, fss_, ds_))
  \ { |infer|}

INTRUDER_1 =
  chase(INTRUDER_0)
  [[hear.(second_(m_)) <- intercept.A_.B_.m_ |
    m_ <- OUTPUT_MSG_BODY, A_ <- SenderType(m_), B_ <- ReceiverType(m_)]]
  [[say.(second_(m_)) <- fake.A_.B_.m_ |
    m_ <- INPUT_MSG_BODY, A_ <- SenderType(m_), B_ <- ReceiverType(m_)]]

SAY_KNOWN =
  ([] f_ : inter(IK1, ALL_SECRETS) @ leak.f_ -> SAY_KNOWN)
  []
  ([] m_ : {m_ | m_ <- OUTPUT_MSG_BODY, subset(components_(m_), IK1)} @
    let ST_ = SenderType(m_)
        RT_ = ReceiverType(m_)
    within
      (intercept?A_:diff(ST_, {Mallory})?B_:RT_!m_ -> SAY_KNOWN))
  []
  ([] m_ : {m_ | m_ <- INPUT_MSG_BODY, subset(components_(m_), IK1)} @
    let ST_ = SenderType(m_)
        RT_ = ReceiverType(m_)
    within
      (fake?A_:ST_?B_:RT_!m_ -> SAY_KNOWN))

INTRUDER =
  (INTRUDER_1 [|{|intercept.Mallory|}|] STOP) ||| SAY_KNOWN

-- Process representing Alice

Alpha_INITIATOR_Alice =
  Union({
    { |intercept.Alice.A_.m_ | A_ <- ALL_PRINCIPALS, m_ <- MSG1_BODY|},
    { |intercept.Alice.A_.m_ | A_ <- ALL_PRINCIPALS, m_ <- MSG3_BODY|},
    { |fake.A_.Alice.m_ | A_ <- ALL_PRINCIPALS, m_ <- MSG2_BODY|}
  })

INITIATOR_Alice = INITIATOR(Alice, Na)

Alpha_Alice = {|intercept.Alice.A_, fake.A_.Alice | A_ <- ALL_PRINCIPALS|}

AGENT_Alice =
  INITIATOR_Alice

-- Process representing Bob

Alpha_RESPONDER_Bob =

```

```

Union({
  { | intercept. Bob. A_. m_ | A_ <- ALL_PRINCIPALS, m_ <- MSG2_BODY | },
  { | fake. A_. Bob. m_ | A_ <- ALL_PRINCIPALS, m_ <- MSG1_BODY | },
  { | fake. A_. Bob. m_ | A_ <- ALL_PRINCIPALS, m_ <- MSG3_BODY | }
})

RESPONDER_Bob = RESPONDER(Bob, Nb)

Alpha_Bob = { | intercept. Bob. A_, fake. A_. Bob | A_ <- ALL_PRINCIPALS | }

AGENT_Bob =
  RESPONDER_Bob

-- Complete system

SYSTEM_0 =
  (AGENT_Alice
   |||
   AGENT_Bob)

SYSTEM = SYSTEM_0 [ | { | intercept, fake | } | ] INTRUDER

-- Systems specifications

Sigma = { | fake, intercept, env, leak | }

-- Secret specifications

SECRET_SPEC_0(s_) =
  signal.Claim_Secret?A!s?Bs_ ->
  (if member(Mallory, Bs_) then SECRET_SPEC_0(s_) else SECRET_SPEC_1(s_))
  []
  leak. s_ -> SECRET_SPEC_0(s_)

SECRET_SPEC_1(s_) = signal.Claim_Secret?A!s?Bs_ -> SECRET_SPEC_1(s_)

AlphaS(s_) =
  union({ | signal.Claim_Secret. A_. s_ | A_ <- ALL_PRINCIPALS | }, { leak. s_ })

Alpha_SECRETS = { | leak, signal.Claim_Secret | }

SECRET_SPEC = ( | | s_ : ALL_SECRETS @ [AlphaS(s_)] SECRET_SPEC_0(s_)

assert SECRET_SPEC [T= SYSTEM_S \ diff(Events, Alpha_SECRETS)

-- Authentication specifications

AuthenticateINITIATORToRESPONDERAgreement_na_nb(A) =
  signal.Running1.INITIATOR_role.A?B?na?nb ->
  signal.Commit1.RESPONDER_role.B.A.na.nb -> STOP

AlphaAuthenticateINITIATORToRESPONDERAgreement_na_nb(A) =
  { | signal.Running1.INITIATOR_role.A.B,
    signal.Commit1.RESPONDER_role.B.A |
    B <- Agent | }

AuthenticateRESPONDERToINITIATORAgreement_na_nb(B) =
  signal.Running2.RESPONDER_role.B?A?na?nb ->
  signal.Commit2.INITIATOR_role.A.B.na.nb -> STOP

AlphaAuthenticateRESPONDERToINITIATORAgreement_na_nb(B) =
  { | signal.Running2.RESPONDER_role.B.A,
    signal.Commit2.INITIATOR_role.A.B |
    A <- Agent | }

AuthenticateINITIATORAliceToRESPONDERAgreement_na_nb =
  AuthenticateINITIATORToRESPONDERAgreement_na_nb(Alice)

assert AuthenticateINITIATORAliceToRESPONDERAgreement_na_nb [T=
  SYSTEM_1 \ diff(Events, AlphaAuthenticateINITIATORToRESPONDERAgreement_na_nb(Alice))

AuthenticateINITIATORBobToRESPONDERAgreement_na_nb =
  STOP

assert AuthenticateINITIATORBobToRESPONDERAgreement_na_nb [T=
  SYSTEM_1 \ diff(Events, AlphaAuthenticateINITIATORToRESPONDERAgreement_na_nb(Bob))

AuthenticateRESPONDERAliceToINITIATORAgreement_na_nb =
  STOP

assert AuthenticateRESPONDERAliceToINITIATORAgreement_na_nb [T=
  SYSTEM_2 \ diff(Events, AlphaAuthenticateRESPONDERToINITIATORAgreement_na_nb(Alice))

AuthenticateRESPONDERBobToINITIATORAgreement_na_nb =
  AuthenticateRESPONDERToINITIATORAgreement_na_nb(Bob)

assert AuthenticateRESPONDERBobToINITIATORAgreement_na_nb [T=

```

```
SYSTEM_2 \ diff(Events, AlphaAuthenticateRESPONDERToINITIATORAgreement_na_nb(Bob))
```

```
SYSTEM_1 = SYSTEM
```

```
[[intercept.A.B.rmb((Msg3, Encrypt.(PK(B), <nb>), <na>))  
  <- signal.Running1.INITIATOR_role.A.B.applyRenaming(na).applyRenaming(nb),  
  fake.A.B.rmb((Msg3, Encrypt.(PK(B), <nb>), <na>))  
  <- signal.Commit1.RESPONDER_role.B.A.applyRenaming(na).applyRenaming(nb) |  
  A <- Agent, B <- Agent, na <- Nonce, nb <- Nonce]]
```

```
SYSTEM_2 = SYSTEM
```

```
[[intercept.B.A.rmb((Msg2, Encrypt.(PK(A), <na, nb>), <>))  
  <- signal.Running2.RESPONDER_role.B.A.applyRenaming(na).applyRenaming(nb),  
  intercept.A.B.rmb((Msg3, Encrypt.(PK(B), <nb>), <na>))  
  <- signal.Commit2.INITIATOR_role.A.B.applyRenaming(na).applyRenaming(nb) |  
  B <- Agent, A <- Agent, na <- Nonce, nb <- Nonce]]
```

```
SYSTEM_S = SYSTEM
```

```
[[intercept.A.B.rmb((Msg3, Encrypt.(PK(B), <nb>), <na>))  
  <- signal.Claim_Secret.A.na.{B},  
  fake.A.B.rmb((Msg3, Encrypt.(PK(B), <nb>), <na>))  
  <- signal.Claim_Secret.B.nb.{A}  
  | A <- Agent, B <- Agent, na <- Nonce, nb <- Nonce]]
```