Méthodes formelles et protocoles cryptographiques

Jean Goubault-Larrecq

G.I.E. Dyade, INRIA

- Méthodes formelles;
- Application aux protocoles cryptographiques.

Méthodes formelles

But : assurer qu'un programme est correct.

Idée de base : vérifier que le programme satisfait une propriété mathématique (appelée *spécification*).

$$\boxed{\text{Formule mathématique}} \xleftrightarrow{\text{Vérification}} \boxed{\text{Programme}}$$

Un exemple-jouet, le tri par fusion (en ML)

```
fun merge ([],l2) = l2
  | merge (l1,[]) = l1
  | merge (l1 as x1::l'1,
           l2 as x2::l'2) =
    if x1 < x2
       then x1::merge(l'1,l2)
    else x2::merge(l1,l'2)
```

```
fun sort1 (l, i, j) =
    if i>=j then []
    else if j=i+1 then [l sub i]
    else let val k = (i+j) div 2
         in merge (sort1 (l,i,k),
                   sort1 (l,k,j))
         end
```

```
fun sort l = sort1 (l, 0, length l);
```

Un exemple-jouet, le tri par fusion (en C)

```
void merge (int *l1, int n1,
            int *l2, int n2,
            int *res) {
  while (n1!=0 && n2!=0) {
    if (*l1 < *l2)
       { *res++ = *l1++; n1--; }
    else { *res++ = *l2++; n2--; }
  }
  while (n1--!=0) *res++ = *l1++;
  while (n2--!=0) *res++ = *l2++;
}
```

```
void sort1 (int *l, int n,
            int *res) {
  int k;

  if (n==0) return;
  if (n==1) { *res = *l; return; }
  k = n / 2;
  sort1 (l, k, res);
  sort1 (l+k, n-k, res+k);
  merge (res, k, res+k, n-k, l);
}
```

```
void sort (int *l, int n) {
  int *aux = (int *) malloc (n * sizeof (int));
  sort1 (l, n, aux);
  free (aux);
}
```

Spécification: qu'est-ce qu'un tri ?

Intuitivement, les deux programmes précédents font la même chose (trier), mais il y a des différences : lesquelles ? (regardez bien, le 2ème est faux. . . corrigez-le.)

À l'opposé, qu'y a-t-il de commun entre ces deux programmes ? Et avec le suivant ?

```
fun sort2 [] = []
  | sort2 (x::l) = case sort2 l of
                     [] => [x]
                   | y::l' =>
                     if x<=y then x::y::l'
                     else sort2 (y::x::l')
```

Spécification informelle du tri

"Un tri prend une liste l en entrée et retourne la liste des mêmes éléments en ordre croissant."

Les comportements suivants de ma fonction `sort3` sont-ils acceptables, selon vous ?

- `sort3 [1,2,3,2] = [1,2,3]`;
- `sort3 [1,2,3,2] = [1,2,2,3]`;
- `sort3 [1,2,3] = [1,2,2,3]`;
- `sort3 [1,2,3,2]` retourne parfois `[1,2,3]`, parfois `[1,2,2,3]`;
- `sort3 [1,2,3]` ne termine pas.

Spécifications formelles

Par exemple, on modélise le fait que `sort3` (l) retourne l' par une relation binaire $R(l,l')$, on veut donc vérifier :

$$\forall l:\ \texttt{int list}, l':\ \texttt{int list} \cdot R(l,l') \Rightarrow tri(l,l') \tag{1}$$

$$\forall l:\ \texttt{int list} \cdot \exists l':\ \texttt{int list} \cdot R(l,l') \tag{2}$$

sous les hypothèses (définitions, ici) :

$$\begin{aligned}&\forall l:\ \texttt{int list}, l':\ \texttt{int list}\cdot\\ &\quad tri(l,l') \Leftrightarrow (memes\_elements(l,l') \wedge ordre\_croissant(l'))\end{aligned} \tag{3}$$

$$\begin{aligned}&\forall l:\ \texttt{int list}, l':\ \texttt{int list}\cdot\\ &\quad memes\_elements(l,l') \Leftrightarrow sous\_liste(l,l') \wedge sous\_liste(l',l)\end{aligned} \tag{4}$$

$$\forall l : \texttt{int list}, l' : \texttt{int list}\cdot$$
$$sous\_liste(l, l') \Leftrightarrow (\forall i : \mathbb{N}\cdot$$
$$0 \leq i \wedge i < len(l) \Rightarrow$$
$$\exists j : \mathbb{N} \cdot 0 \leq j \wedge j < len(l') \wedge sub(l, i) = sub(l', j)) \tag{5}$$

$$\forall l : \texttt{int list}\cdot$$
$$ordre\_croissant(l) \Leftrightarrow (\forall i : \mathbb{N}, j : \mathbb{N}\cdot \tag{6}$$
$$0 \leq i \wedge i \leq j \wedge j < len(l) \Rightarrow sub(l, i) \leq sub(l, j)$$

$$len(\texttt{[]}) = 0 \tag{7}$$

$$\forall l : \texttt{int list}, x : \texttt{int} \cdot len(x\texttt{::}l) = len(l) + 1 \tag{8}$$

$$\forall l : \texttt{int list}, x : \texttt{int} \cdot sub(x\texttt{::}l, 0) = x \tag{9}$$

$$\forall l : \texttt{int list}, x : \texttt{int}, i : \texttt{int} \cdot sub(x\texttt{::}l, i + 1) = sub(l, i) \tag{10}$$

Est-ce la sécification que vous souhaitiez ?

Une autre (en logique d'ordre 2); remplacer (4) et (5) par :

$$
\begin{aligned}
&\forall l : \texttt{int list}, l' : \texttt{int list} \cdot \\
&\quad memes\_elements(l, l') \Leftrightarrow \exists \pi : \mathbb{N} \to \mathbb{N} \cdot \\
&\qquad (\forall j : \mathbb{N} \cdot \exists! i : \mathbb{N} \cdot \pi(i) = j) \wedge \\
&\qquad (\forall i : \mathbb{N} \cdot 0 \leq i \wedge i < len(l) \Rightarrow \\
&\qquad\quad 0 \leq \pi(i) \wedge \pi(i) < len(l') \wedge sub(l', \pi(i)) = sub(l, i))
\end{aligned}
\tag{11}
$$

Comment s'assurer que `sort3` est correct ?

Dépend du niveau de *confiance* que l'on veut atteindre :

- Test (indispensable, mais ne donne que peu de garanties);
- Relecture du code (si possible par un autre);
- Écriture de la spécification (détecte les ambiguïtés);
- Preuve de correction informelle (à la main);
- Preuve automatique de programmes (infaisable);
- Dérivation automatique de propriétés par interprétation abstraite;

- Preuve formelle (d'une idéalisation suffisamment poussée), c'est-à-dire *mécanisée*.

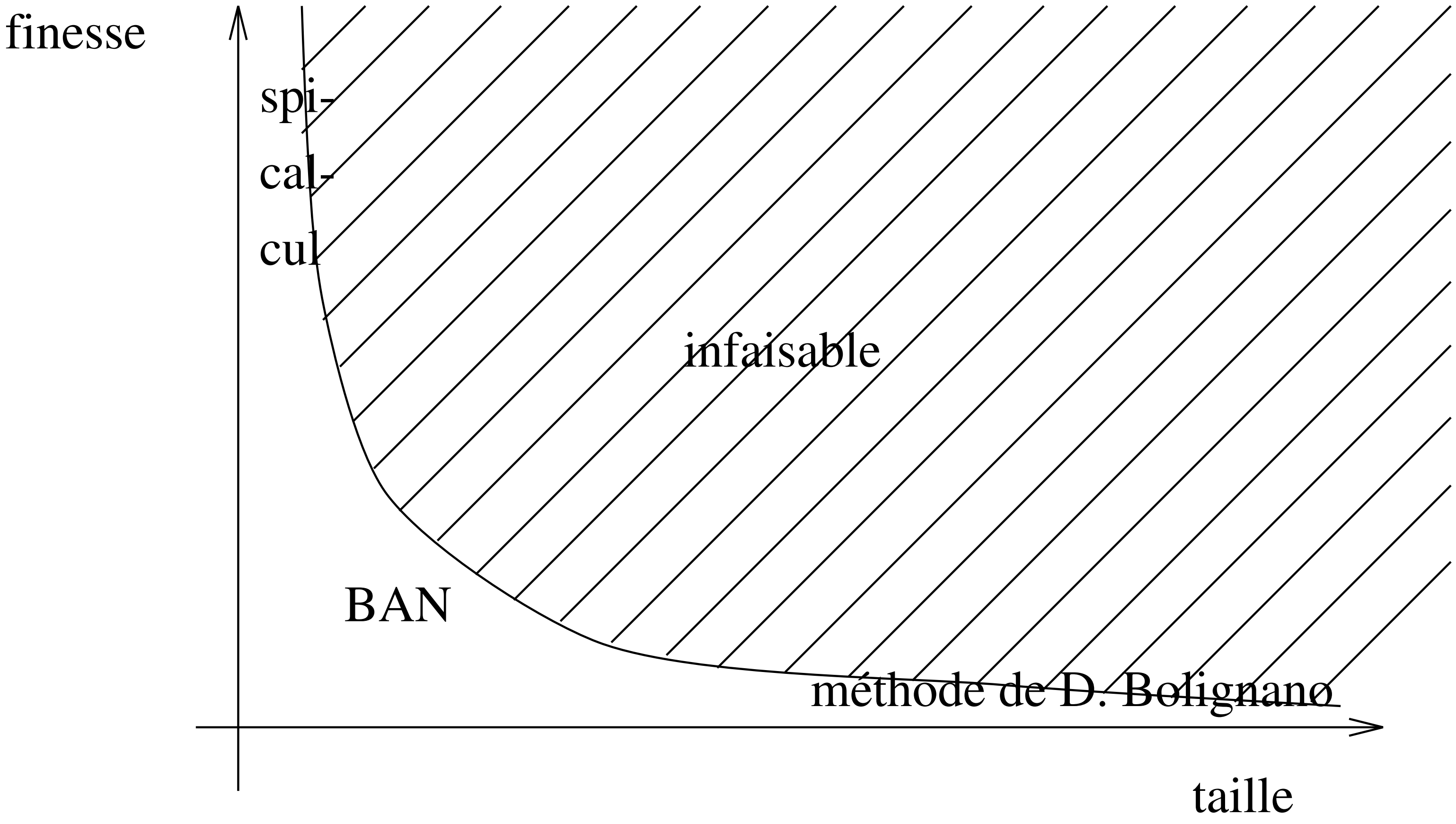
Autres méthodes formelles

- Méthodes orientées modèles : VDM, Z, B, Raise, etc.
- Méthodes d'états finis : *model-checking* de formules en logique temporelle (spécifications) sur des automates finis (les programmes).
- Extraction de programmes; en logique intuitionniste, toute preuve de $\forall x \cdot \exists y \cdot P(x, y)$ *est* un programme calculant y en fonction de x : Coq, NuPrl, LF, etc.
- Algèbres de processus (pour programmes parallèles), notions d'équivalence opérationnelle approchées par des *bisimulations*: CSP, CCS, π-calcul, etc.

Applications aux protocoles cryptographiques

- Logiques modales — la BAN;
- Algèbres de processus — le spi-calcul;
- Les méthodes orientées modèle — l'approche de D. Bolignano.

Compromis (à l'heure actuelle)



Le pionnier : la BAN (Burrows-Abadi-Needham)

Modèle très simple de données :

$$
\begin{array}{rcll}
m & ::= & K & \text{clés} \\
 & | & D & \text{autres messages de base} \\
 & | & m_K & \text{chiffrement} \\
 & | & m, m & \text{couples} \\
 & | & \ldots &
\end{array}
$$

Bizarrerie : messages $\equiv$ formules.

Logique modale avec opérateurs “croit”, “voit”, “a dit”, “a juridiction sur”, “est frais”, “est un secret entre”, etc.

Ambitions et défauts de la BAN

- Ambition modeste : trouver des bugs de nature logique, i.e. *indépendants* de la qualité des algorithmes cryptographiques utilisés.
- Résultat surprenant : de nombreuses attaques *sont de nature logique*.
- Défauts : ne permet pas de prouver qu'un protocole est correct; mauvaise notion de fraîcheur; faiblesse des postulats permettant de conclure qu'une donnée est secrète; nécessité d'une étape d'idéalisation; etc.

Règles

"Si P croit $P \stackrel{K}{\leftrightarrow} Q$, et P voit X_K, alors P croit que Q a dit X."

"Si P croit Y secret entre P et Q, et P voit X, Y, alors P croit que Q a dit X."

"Si P croit X frais, et que Q a dit X, alors P croit que Q croit X."

"P croit X, Y ssi P croit X et P croit Y."

"Si P croit que Q a dit X, Y, alors P croit que Q a dit X (resp. Y)."

"Si P voit X, Y, alors P voit X (resp. Y)."

"Si P croit $P \stackrel{K}{\leftrightarrow} Q$ et P voit X_K, alors P voit X."

"Si P croit X (resp. Y) frais, alors P croit X, Y frais."

Exemple : Needham-Schroeder à clés symétriques

1. $A \to S \quad : A, B, N_a$
2. $S \to A \quad : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3. $A \to B \quad : \{K_{ab}, A\}_{K_{bs}}$
4. $B \to A \quad : \{N_b\}_{K_{ab}}$
5. $A \to B \quad : \{N_b - 1\}_{K_{ab}}$

Needham-Schroeder idéalisé

$$
\begin{array}{lll}
1. & A \rightarrow S & : \cancel{A, B, N_a} \\
2. & S \rightarrow A & : \cancel{\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}} \\
 & & \{N_a, A \overset{K_{ab}}{\leftrightarrow} B, A \overset{K_{ab}}{\leftrightarrow} B \text{ frais}, \{A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}\}_{K_{as}} \\
3. & A \rightarrow B & : \cancel{\{K_{ab}, A\}_{K_{bs}}} \\
 & & \{A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}} \\
4. & B \rightarrow A & : \cancel{\{N_b\}_{K_{ab}}} \\
 & & \{N_b, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{ab}} \\
5. & A \rightarrow B & : \cancel{\{N_b - 1\}_{K_{ab}}} \\
 & & \{N_b, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{ab}}
\end{array}
$$

Needham-Schroeder analysé

Sous les hypothèses :

“A croit $A \stackrel{K_{as}}{\leftrightarrow} S$, B croit $B \stackrel{K_{as}}{\leftrightarrow} S$.”

“S croit $A \stackrel{K_{ab}}{\leftrightarrow} B$.”

“A croit que S a juridiction sur $A \stackrel{K}{\leftrightarrow} B$, $\forall K$.”

...

“B croit $A \stackrel{K}{\leftrightarrow} B$ frais, $\forall K$.” (*)

On déduit “B (resp. A, A croit que B, B croit que A) croit $A \stackrel{K_{ab}}{\leftrightarrow} B$.”
(Sans (*), seulement “B croit que S a dit $A \stackrel{K_{ab}}{\leftrightarrow} B$.”)

Le spi-calcul

Termes :

$$
\begin{array}{rcll}
m & ::= & n & \text{noms} \\
 & | & x & \text{variables} \\
 & | & 0 & \\
 & | & S(m) & \text{données de base} \\
 & | & \{m_1\}_{m_2} & \text{chiffrement} \\
 & | & (m_1, m_2) & \text{couples}
\end{array}
$$

Processus :

$$
\begin{array}{rcll}
P & ::= & \overline{m_1}\langle m_2\rangle.P & \text{émission} \\
 & | & m_1(x).P & \text{réception} \\
 & | & P_1 \mid P_2 & \text{parallèle} \\
 & | & (\nu n)P & \text{restriction} \\
 & | & !P & \text{réplication} \\
 & | & [m_1 \ is \ m_2]P & \text{test d'égalité} \\
 & | & \mathbf{0} & \text{nil} \\
 & | & let\ (x, y) = m\ in\ P & \text{décomposition} \\
 & | & case\ m_1\ of\ 0 : P\ S(x) : Q & \text{analyse de cas} \\
 & | & case\ m_1\ of\ \{x\}_{m_2}\ in\ P & \text{déchiffrement}
\end{array}
$$

Pas de logique, ce n'est qu'un langage de programmation.

Règles de calcul

$$(\overline{m}\langle m'\rangle.P) \mid (m(x).Q) \rightarrow P \mid (Q[m'/x]) \tag{12}$$

$$case\ 0\ of\ \ 0 : P\ S(x) : Q \rightarrow P \tag{13}$$

$$case\ S(m)\ of\ \ 0 : P\ S(x) : Q \rightarrow Q[m/x] \tag{14}$$

$$case\ \{m\}_{m'}\ of\ \ \{x\}_{m'}\ in\ P \rightarrow P[m/x] \tag{15}$$

etc.

Needham-Schroeder en spi-calcul

$A(F, K_{as}) =$

/* 1. $A \to S$ */

$(\nu N_a)\overline{c}_{AS}\langle id_A, id_B, N_a\rangle.$

/* 2. */

$c_{SA}(x).$

$case\ x\ of \{x'\}_{K_{as}}\ in$

$let\ (x_{N_a}, x_B, x_K, x'') = x'\ in$

$[x_{N_a}\ is\ N_a][x_B\ is\ id_B]$

/* 3. $A \to B$ */

$\overline{c}_{AB}\langle x''\rangle.$

/* 4. */

$c_{BA}(x''').$

$case\ x'''\ of \{x_{N_b}\}_K\ in$

/* 5. $A \to B$ */

$\overline{c}_{AB}\langle\{x_{N_b} - 1\}_K\rangle.F(K)$

$B(G, K_{bs}) =$

/* 3. */

$c_{AB}(z).$

$case\ z\ of \{z'\}_{K_{bs}}\ in$

$let\ (K, z_A) = z'\ in$

$[z_A\ is\ id_A]$

/* 4. $B \to A$ */

$(\nu N_b)\overline{c}_{BA}\langle\{N_b\}_K\rangle.$

/* 5. */

$c_{AB}(z'').$

$case\ z''\ of \{z'''\}_K\ in$

$[z'''\ is\ N_b - 1]G(K)$

$S(K_{as}, K_{bs}) =$

/* 1. */

$c_{AS}(y_A, y_B, y_{N_a}).$

/* 2. $S \to A$ */

$(\nu K_{ab})\overline{c}_{SA}\langle\{y_{N_a}, y_B, K_{ab}, \{K_{ab}, y_A\}_{K_{bs}}\}_{K_{as}}\rangle.$

Spécifications en spi-calcul

$$
\begin{array}{rcll}
Inst(F, G) & = & (\nu K_{as})(\nu K_{bs}) & \\
 & & (A(F, K_{as}) \mid B(G, K_{bs}) \mid !S(K_{as}, K_{bs})) & (16) \\
F_M(x) & = & \overline{c}_{AB}\langle\{M\}_x\rangle. & (17) \\
G(z) & = & c_{AB}(z').case\ z'\ of\ \{y\}_z\ in\ H(y) & (18)
\end{array}
$$

Équivalence observationnelle $\simeq$: $P \simeq Q$ ssi P et Q sont indistingables dans tout contexte.

K garantit une communication *secrète* ssi, pour tout H, pour tous M, M' tels que $H(M) \simeq H(M')$, $Inst(F_M, G) \simeq Inst(F_{M'}, G)$.

Authentification

$$
\begin{array}{rcl}
Inst(F, G) & = & (\nu K_{as})(\nu K_{bs}) \\
 & & (A(F, K_{as}) \mid B(G, K_{bs}) \mid !S(K_{as}, K_{bs})) \\
F_M(x) & = & \overline{c}_{AB}\langle\{M\}_x\rangle. \\
G_M(z) & = & c_{AB}(z').case\ z'\ of\ \{y\}_z\ in\ H(M)
\end{array} \qquad (19)
$$

K garantit l'*authenticité* des messages ssi, pour tout H, pour tout M, $Inst(F_M, G) \simeq Inst(F_M, G_M)$.

L'approche de Dominique Bolignano

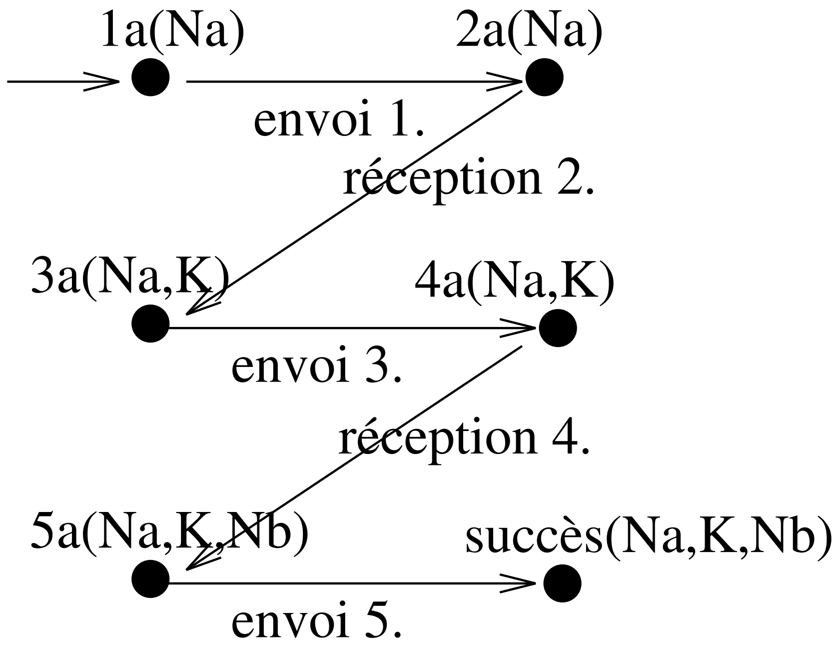
Attitude inverse du spi-calcul : on n'a plus que des formules (en logique du premier ordre). La sémantique du programme est décrite sous forme d'un ensemble de formules, exécutables par Prolog.

L'exécution du protocole est un *automate* infini.

La méthode est paramétrable par divers modèles de sécurités (des axiomes).

Dérivation de preuves formelles (à l'heure actuelle en Coq).

Le rôle de A en tant qu'automate



Modélisation du protocole

Le système ($A \mid B \mid S \mid I$, où I est l'intrus) est modélisé comme un automate produit; chaque état s est un quadruplet (s_A, s_B, s_S, E).

Pour ce qui est des propriétés de sécurité, on peut supposer que l'on est dans le pire des cas : tous les principaux parlent à l'intrus, reçoivent leurs messages de l'intrus.

À tout instant, l'intrus possède un ensemble E de messages (espionnés sur les canaux), et ne peut fabriquer que des messages m tels que $known\_in(E, m)$.

Le prédicat $known\_in$

...est défini inductivement par :

$$known\_in(E \cup \{m\}, m)$$

$$known\_in(E, m) \wedge known\_in(E, K) \Rightarrow known\_in(E, m_K)$$

$$known\_in(E, m_K) \wedge known\_in(E, K^{-1}) \Rightarrow known\_in(E, m)$$

$$known\_in(E, m_1) \wedge known\_in(E, m_2) \Rightarrow known\_in(E, (m_1, m_2))$$

$$known\_in(E, (m_1, m_2)) \Rightarrow known\_in(E, m_i)$$

Modélisation de la relation de transition

Écriture d'un prédicat P(état-avant, transition, message, état-après).

$$
\begin{aligned}
P(s,l,m,s') = \quad & (l = \text{émission-1.} \wedge P\_emission\_1(s,m,s')) \\
& \vee(l = \text{réception-1.} \wedge P\_reception\_1(s,m,s')) \\
& \vee \ldots
\end{aligned}
$$

$$
\begin{aligned}
& P\_emission\_1((s_A, s_B, s_S, E), m, (s'_A, s'_B, s'_S, E') = \\
& \qquad s_A = 1a(N_a) \wedge s'_A = 2a(N_a) \wedge s'_B = s_B \wedge s'_S = s_S \\
& \qquad \wedge E' = E \cup \{m\} \wedge m = (id_A, id_B, N_a) \wedge \neg known\_in(E, N_a) \\
& P\_reception\_1((s_A, s_B, s_S, E), m, (s'_A, s'_B, s'_S, E') = \\
& \qquad s_S = 1s() \wedge s'_S = 2s(x_A, x_B, x_{N_a}) \wedge s'_A = s_A \wedge s'_B = s_B \\
& \qquad \wedge E' = E \wedge m = (x_A, x_B, x_{N_a}) \wedge known\_in(E, m)
\end{aligned}
$$

Modélisation du protocole

Exécution :

$$Exec(s, s) \qquad P(s, l, m, s') \wedge Exec(s', s'') \Rightarrow Exec(s, s'')$$

Hypothèses de départ :

$$\neg known\_in(E_0, K_{as}) \qquad \neg known\_in(E_0, K_{bs})$$

$$Exec((s_{A0}, s_{B0}, s_{S0}, E_0), (s_{Afin}, s_{Bfin}, s_{Sfin}, E_{fin}))$$

Montrer :

$$\neg known\_in(E_{fin}, K_{ab})$$

$$\neg known\_in(E_{fin}, K_{as})$$

$$\neg known\_in(E_{fin}, K_{bs})$$

Quelques remarques

Souple : on peut changer le modèle de sécurité (ici, pour simplifier, on avait confiance en A, B et S, et S était en mono-session).

S'étend à de gros protocoles cryptographiques, y compris de commerce électronique.

L'exécution Prolog retourne soit “non”, soit une attaque. Dans le premier cas, la trace Prolog fournit l'argument de preuve.