

Pirates et sécurité informatique : la logique à la rescousse !

Jean Goubault-Larrecq

LSV/CNRS UMR 8643, ENS Cachan

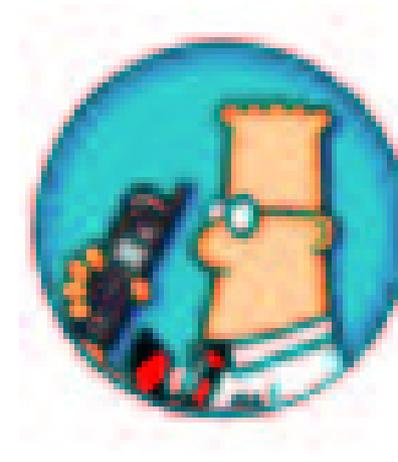
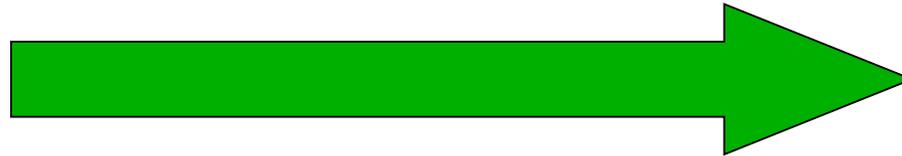
Plan

1. Comment envoyer des messages **secrets** ?
2. Une étude de quelques **attaques**.
3. Un modèle **logique** pour la **vérification** de protocoles cryptographiques
4. Une méthode de **vérification** (logique et automates)

Alice veut envoyer un message à Bob...



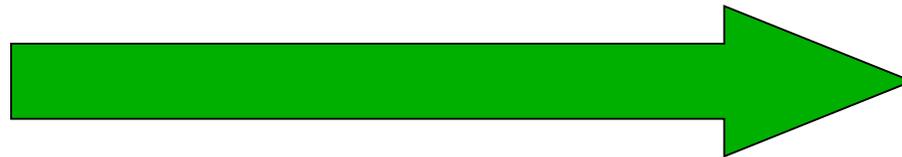
solution : 2 ; 3 ; 1 ; 1



Alice veut envoyer un message à Bob...



solution : 2 ; 3 ; 1 ; 1

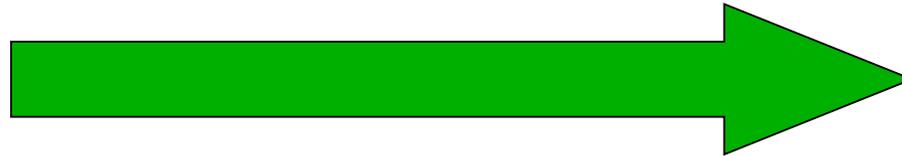


Mais Charlie peut espionner le message : pas de **secret**.

La solution standard : le chiffrement



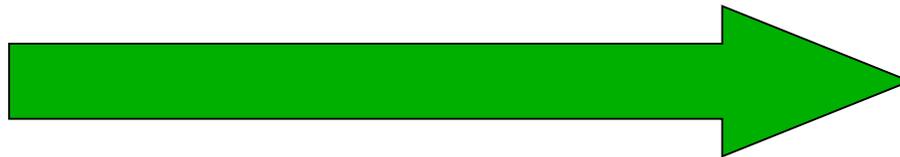
vroxwlrq = 5 > 6 > 4 > 4



La solution standard : le chiffrement



vroxwlrq = 5 > 6 > 4 > 4

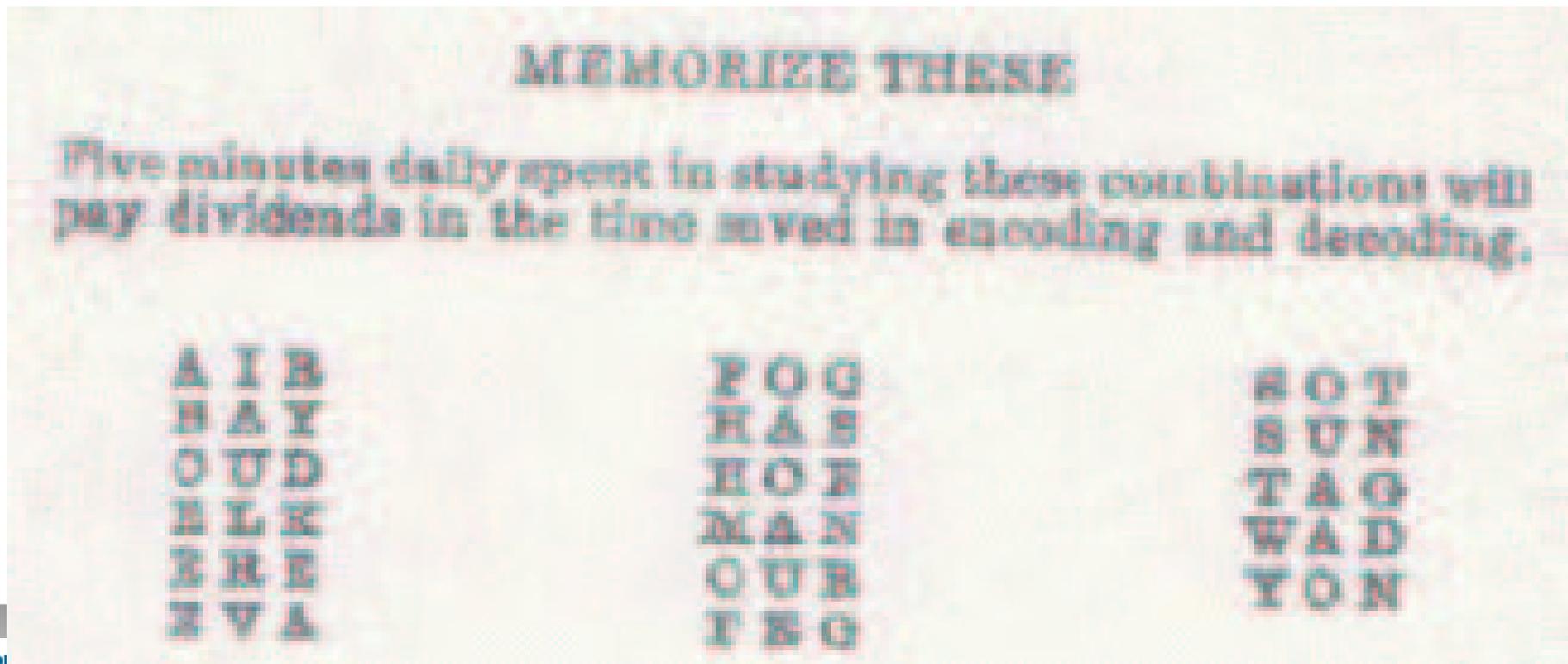


Mais l'algorithme de chiffrement est trop simple...

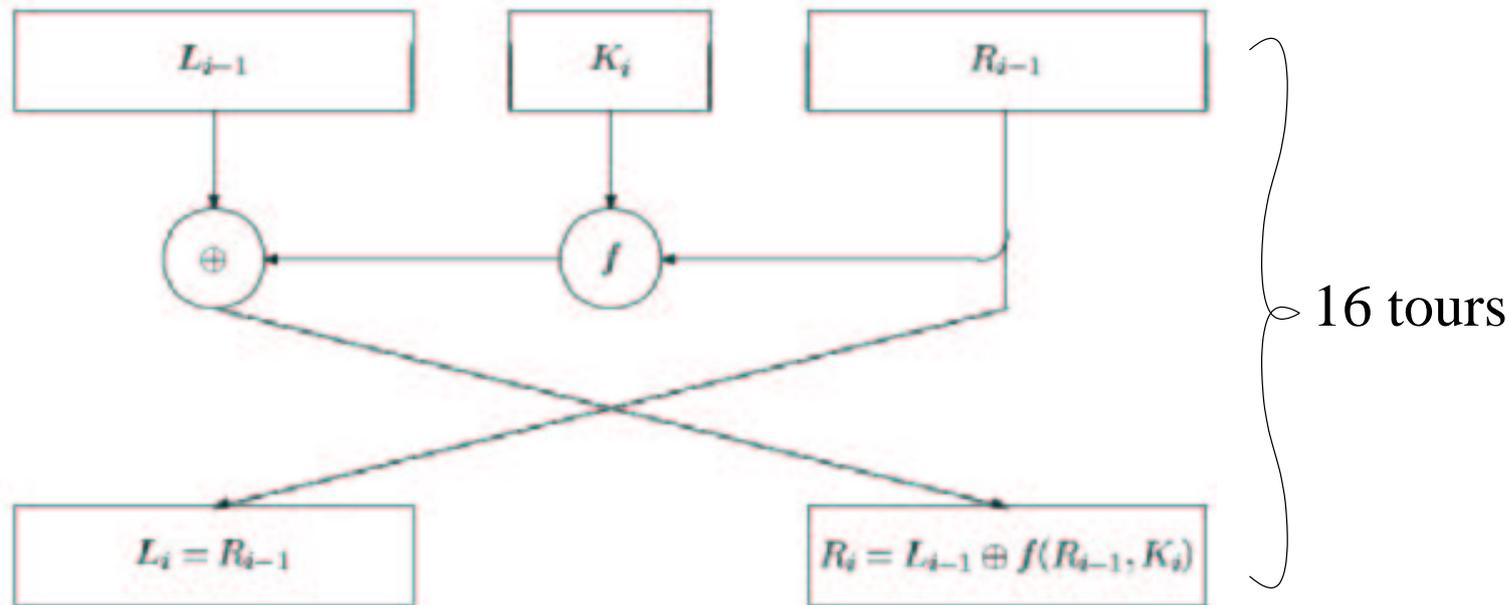
Un meilleur système de chiffrement

Envoyer k ème lettre \oplus k ème lettre d'un **one-time pad**, $1 \leq k \leq$ longueur :

(\oplus =ou exclusif, ou bien addition mod 26, etc.)



Et il y en a d'autres, le DES par exemple...



Un petit point de vocabulaire

En Français, on dit :

- **chiffrer**, pas “crypter” (to encrypt)
- **déchiffrer**, pas “décrypter” (to decrypt)

Un petit point de vocabulaire

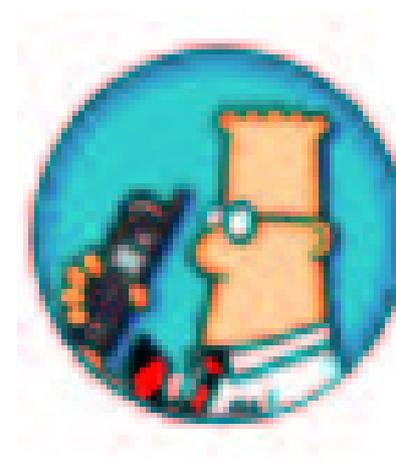
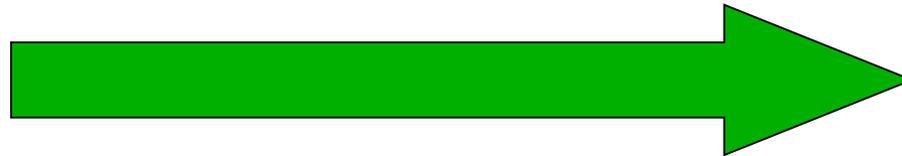
En Français, on dit :

- **chiffrer**, pas “crypter” (to encrypt)
- **déchiffrer**, pas “décrypter” (to decrypt)
- **décrypter**... (to decrypt, to crack)

Avec un chiffrement amélioré...



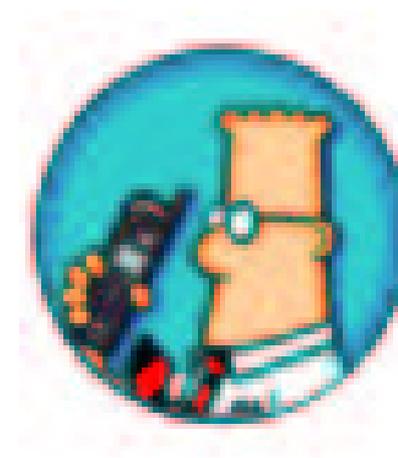
sw2z7o61 ; JB EM ≥ 4
= {solution : 2 ; 3 ; 1 ; 1}_K



... Charlie peut encore nuire !



sw2z7o61 ; IB DM @ = 4
= {solution : 1 ; 2 ; 3 ; 1} _K



L'intégrité du message est compromise.

Une solution : éviter le chiffrement par blocs

Par ex., CBC : Découper le messages en blocs (de 64 bits pour DES par ex.) :

B_1	B_2	...	B_n
-------	-------	-----	-------

Envoyer :

	$\{V_0I \oplus B_1\}_K$	$\{IV_1 \oplus B_2\}_K$...	$\{V_nL_1 \oplus B_n\}_K$
V_0I	$= IV_1$	$= IV_2$		$\forall_n I$

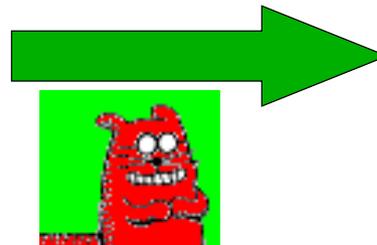
Empêche de tronquer ou remplacer des bouts.

Intégrité : inclure des sommes de contrôle (hash).

On n'arrête pas Charlie !



jm ss :o1t =hs
= {je ne sais pas}_K

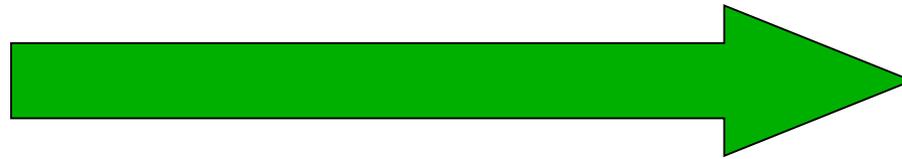


Charlie peut rejouer un vieux message, même sans connaître la clé K .

Alice inclut un temps dans son message...

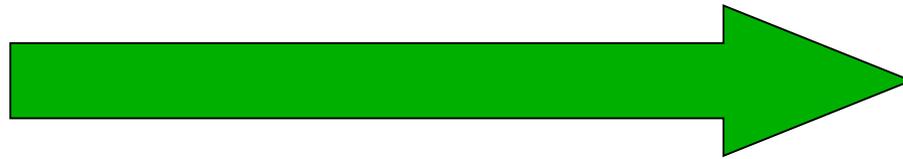


sw2z7o61 ; JB EM >= 4 - C9@5=
= {solution : 2 ; 3 ; 1 ; 1 - 09 :12} K

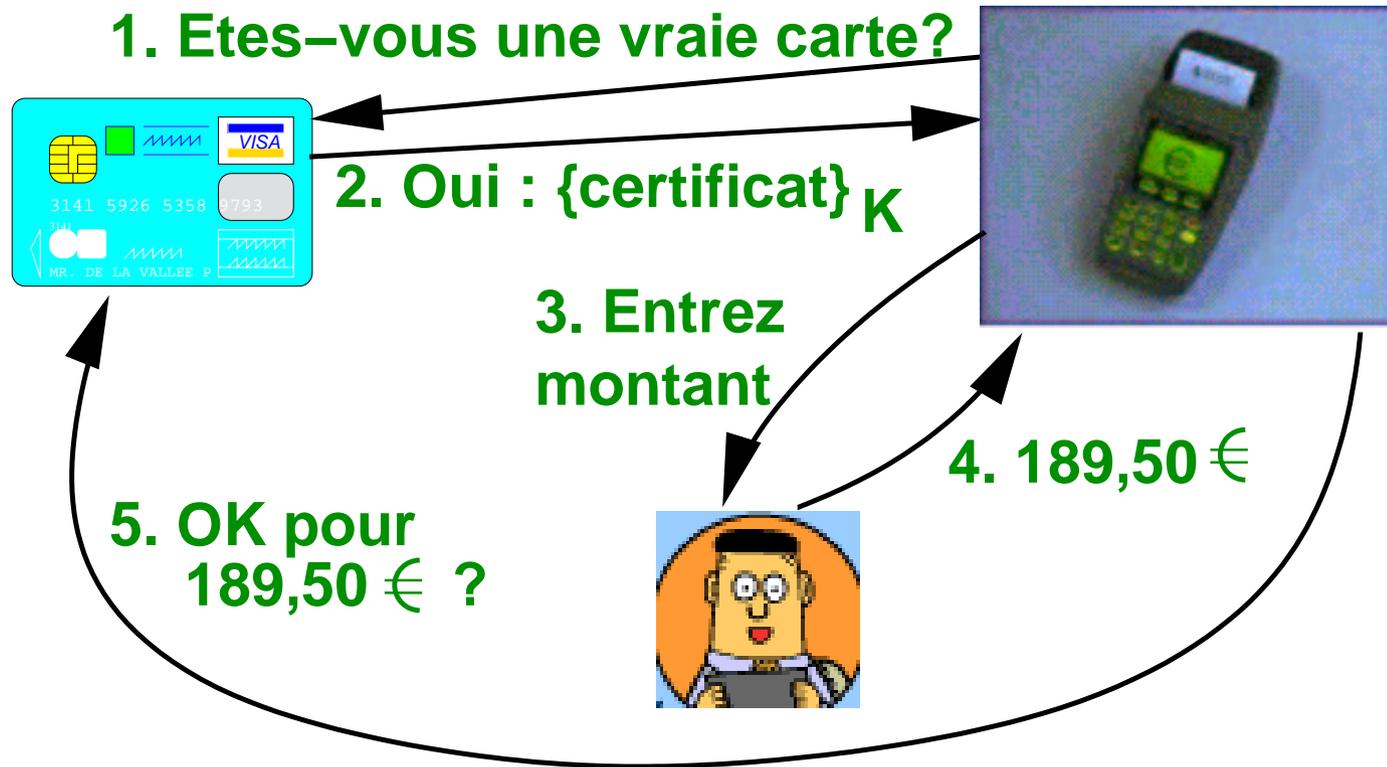


...Charlie ne peut plus rejouer de vieux messages.

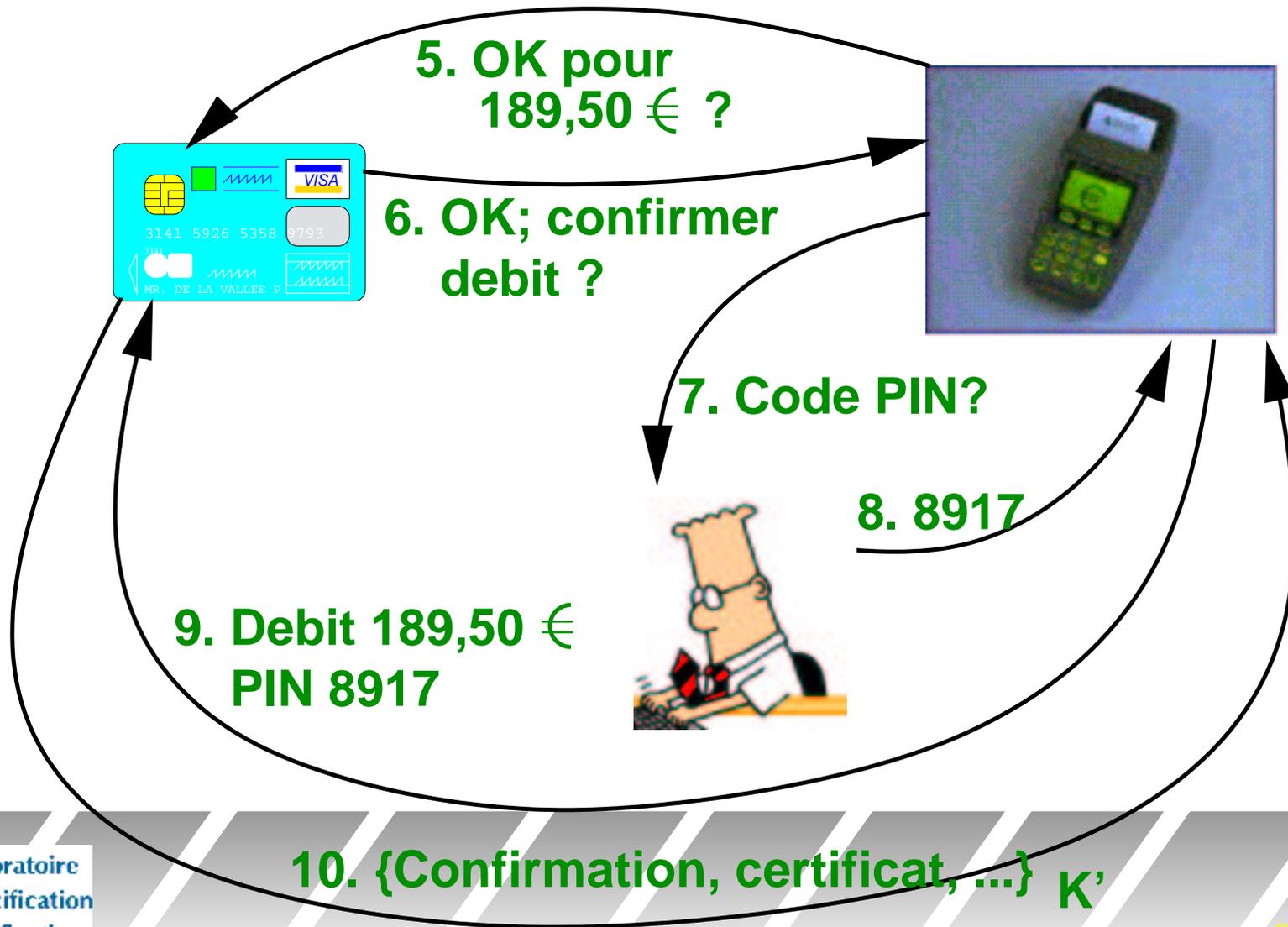
sw2z7o61 ; IB DM @ = 4 - C8@4@
= {solution : 1 ; 2 ; 3 ; 1 - 08 :05} K



Un exemple concret : les cartes à puce



Les cartes à puce, suite



En résumé

Alice a dû utiliser :

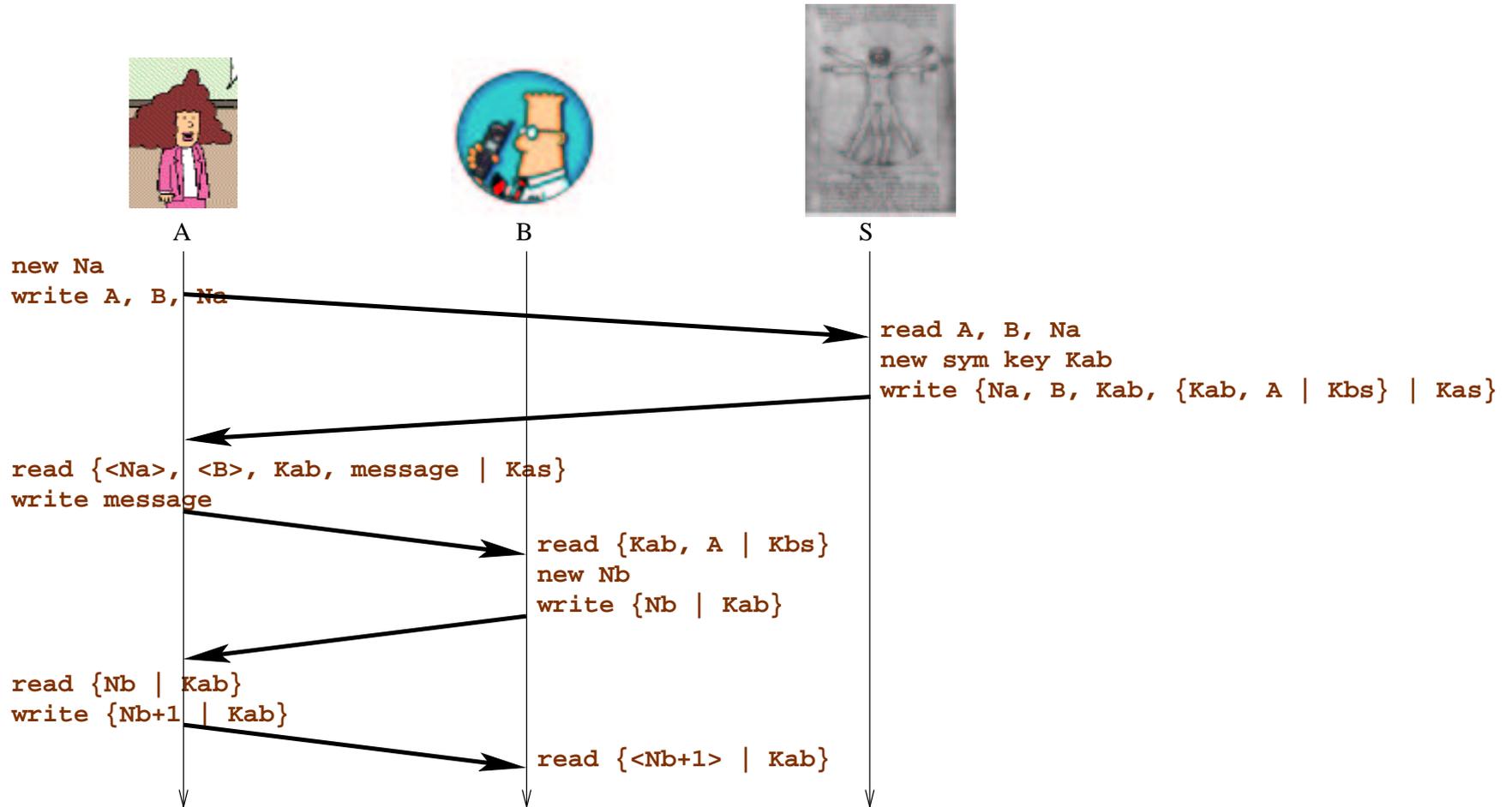
- un algorithme de **chiffrement fort** **secret** ;
- des **sommes de contrôle** **intégrité** ;
- des **estampilles** temporelles **fraîcheur** ;
- plus quelques autres précautions de codage
(CBC, types explicites, ...)
- plus divers **authenticité, unicité, ...**

Toutes ces précautions assurent-elles que Charlie n'arrivera pas à ses fins ?

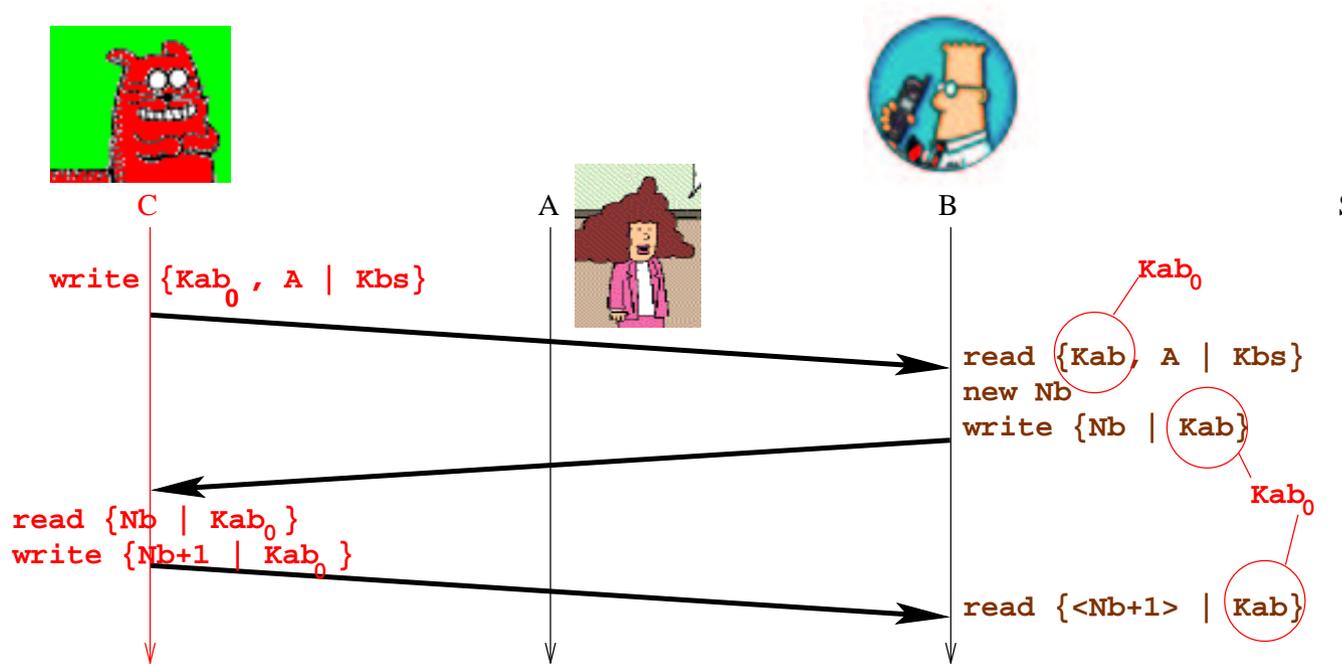
2. Une étude de quelques attaques

Étudios un exemple

[NeedhamSchroeder78]



Une attaque (secret, authentication)



Miracle : la cryptographie à clé publique

A fabrique K_a, K_a^{-1} , publie K_a , garde K_a^{-1} .

Pour **chiffrer** M , B utilise la clé **publique** K_a :

$$M' = \{M\}_{K_a}.$$

A **déchiffre** avec sa clé **privée** : $M = \{M'\}_{K_a^{-1}}.$

Miracle : la cryptographie à clé publique

A fabrique K_a, K_a^{-1} , publie K_a , garde K_a^{-1} .

Pour **chiffrer** M , B utilise la clé **publique** K_a :

$$M' = \{M\}_{K_a}.$$

A **déchiffre** avec sa clé **privée** : $M = \{M'\}_{K_a^{-1}}$.

Arg : M' n'est pas forcément **authentique** (K_a est publique...) \implies B **signe** avec sa clé **privée** K_b^{-1} .

Miracle : la cryptographie à clé publique

A fabrique K_a, K_a^{-1} , publie K_a , garde K_a^{-1} .

Pour **chiffrer** M , B utilise la clé **publique** K_a :

$$M' = \{M\}_{K_a}.$$

A **déchiffre** avec sa clé **privée** : $M = \{M'\}_{K_a^{-1}}$.

Arg : M' n'est pas forcément **authentique** (K_a est publique...) \implies B **signe** avec sa clé **privée** K_b^{-1} .

Secret+authenticité : B envoie $\underbrace{\{M, \{h(M)\}_{K_b^{-1}}\}}_{M'} K_a$

A vérifie ensuite

$$h(M) = \{M'\}_{K_b}.$$

La cryptographie à clé publique, c'est bien mais...

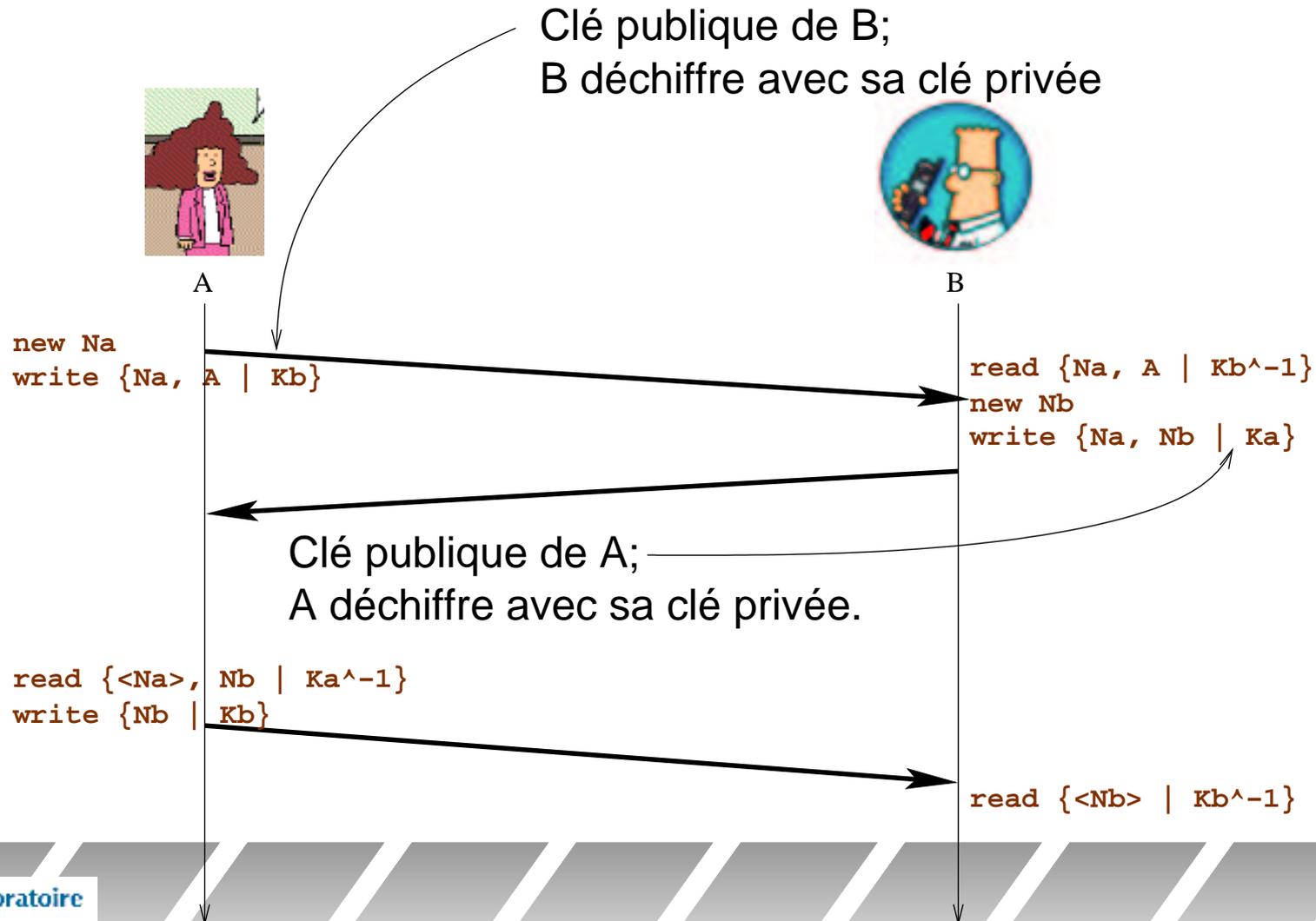
- C'est lourd et **lent** (RSA est typiquement 1000 fois plus lent que DES).

La cryptographie à clé publique, c'est bien mais...

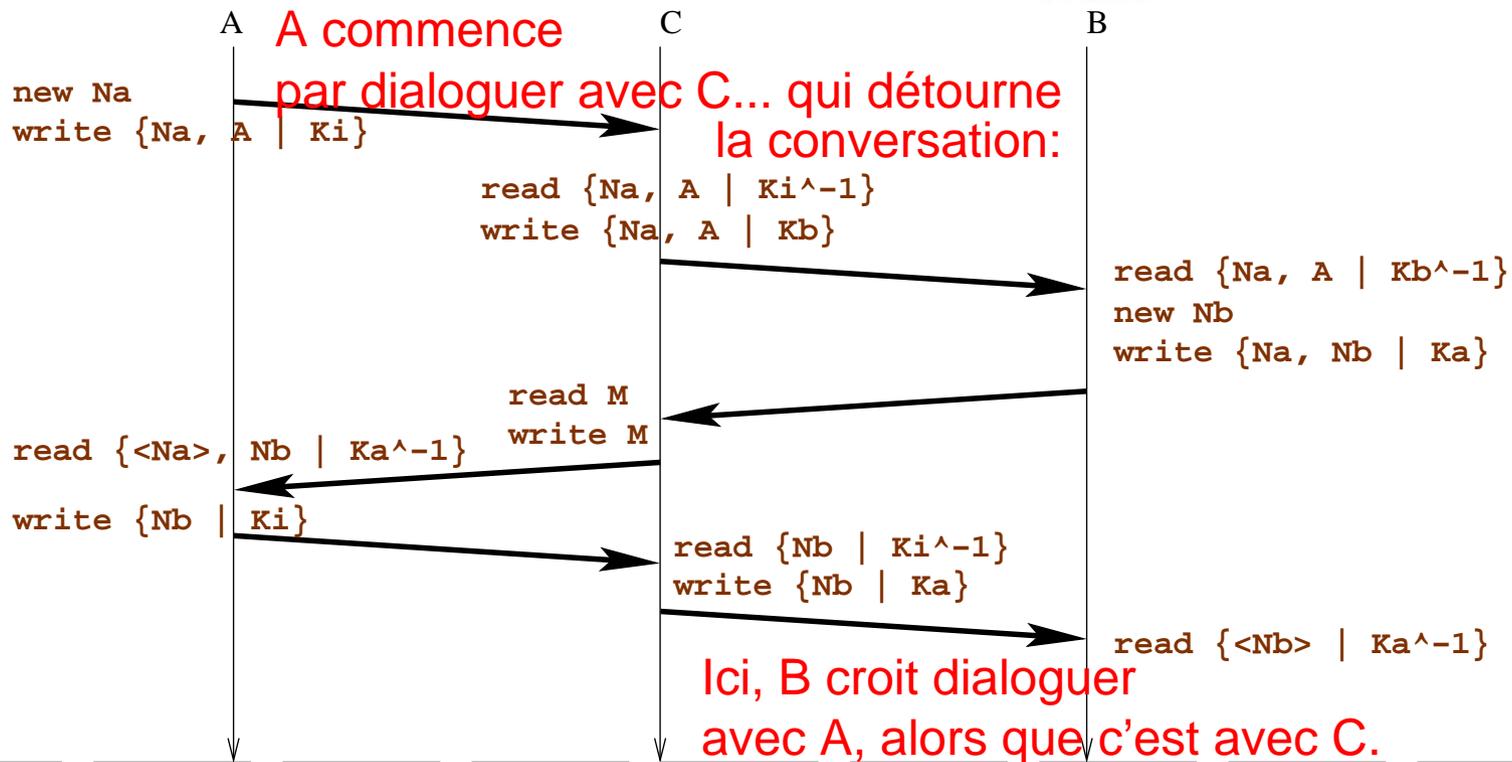
- C'est lourd et **lent** (RSA est typiquement 1000 fois plus lent que DES).
- Ça ne change rien au fait que les protocoles cryptographiques contiennent des bugs **purement logiques** !

Un autre exemple classique

[NeedhamSchroeder78] [aussi !]



Caramba, encore raté !



Quelle proportion de protocoles publiés sont faux ?

- Les 2 protocoles de [NeedhamSchroeder78] : **faux**.
- [OtwayRees87] : OK si exécution séquentielle, **faux** si un principal peut jouer le rôle de A et de B dans deux sessions différentes en même temps.
- [DenningSacco81], Yahalom [BAN89], Wide Mouthed Frog [BAN89], CCITT X.509 [BAN89], Andrew Secure RPC [Satyanarayanan89], SPLICE [YamaguchiOkayamaMiyahara90], [Sneekenes92], station-to-station protocol

[DiffieVanOorschotWiener92], KSL
[KehneSchönwälderLandendörfer92],
[NeumanStubblebine93], [WooLam94] : **faux** (à
des degrés divers).

Il y a donc besoin de méthodes **systematiques**
(automatiques ?) de vérification de protocoles
cryptographiques.



Vérification exhaustive

Un moyen direct : énumérer toutes les traces d'exécution possibles du protocole, et tester à chaque état les propriétés P (secret, etc.)

Vérification exhaustive

Un moyen direct : énumérer toutes les traces d'exécution possibles du protocole, et tester à chaque état les propriétés P (secret, etc.)

NB Les propriétés P à tester sont d'**inaccessibilité** (d'un état où P échoue).

Vérification exhaustive

Un moyen direct : énumérer toutes les traces d'exécution possibles du protocole, et tester à chaque état les propriétés P (secret, etc.)

NB Les propriétés P à tester sont d'**inaccessibilité** (d'un état où P échoue).

Thm [EvenGoldreich83,DurginLincolnMitchell Scedrov99] (Sous quelques hypothèses) Ces propriétés P sont **indécidables**.

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à **états finis** (automate fini).

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à **états finis** (automate fini).

1. **messages**

de tailles non bornées

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à **états finis** (automate fini).

1. **messages**
2. **nonces**

de tailles non bornées
en nombre arbitraire

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à **états finis** (automate fini).

1. **messages**
2. **nonces**
3. **canaux**

de tailles non bornées
en nombre arbitraire
capacité illimitée de l'intrus

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à **états finis** (automate fini).

1. **messages** de tailles non bornées
2. **nonces** en nombre arbitraire
3. **canaux** capacité illimitée de l'intrus
4. **instances** principaux en nombre quelconque.

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à **états finis** (automate fini).

1. **messages** de tailles non bornées
2. **nonces** en nombre arbitraire
3. **canaux** capacité illimitée de l'intrus
4. **instances** principaux en nombre quelconque.

Pour une présentation unifiée, on écrira des **programmes Prolog** purs = ens. de **clauses de Horn**.

3. Un modèle logique pour les protocoles cryptographiques

Qu'est-ce qu'une modélisation ?

On cherche à décrire **formellement** ce que fait un protocole.

On ne cherche pas (encore) à vérifier automatiquement des propriétés sur ces protocoles (cf. partie 4...)

Les clauses de Horn

```
on_route(rome).
```

```
on_route(Place) :-  
    move(Place, Method, NewPlace),  
    on_route(NewPlace).
```

```
move(home, taxi, halifax).
```

```
move(halifax, train, gatwick).
```

```
move(gatwick, plane, rome).
```

```
?on_route(home, X).      [false :- on_route(home, X).]
```

Sources d'infini — 1. Les messages

Espace des messages

$M ::= D$	Données de base
K	Clés brutes
(M, \dots, M)	n-uplets
$\{M K\}$	Chiffrements

Taille des messages **non bornée**.

Codage de systèmes de transitions sous forme de clauses de Horn

États :

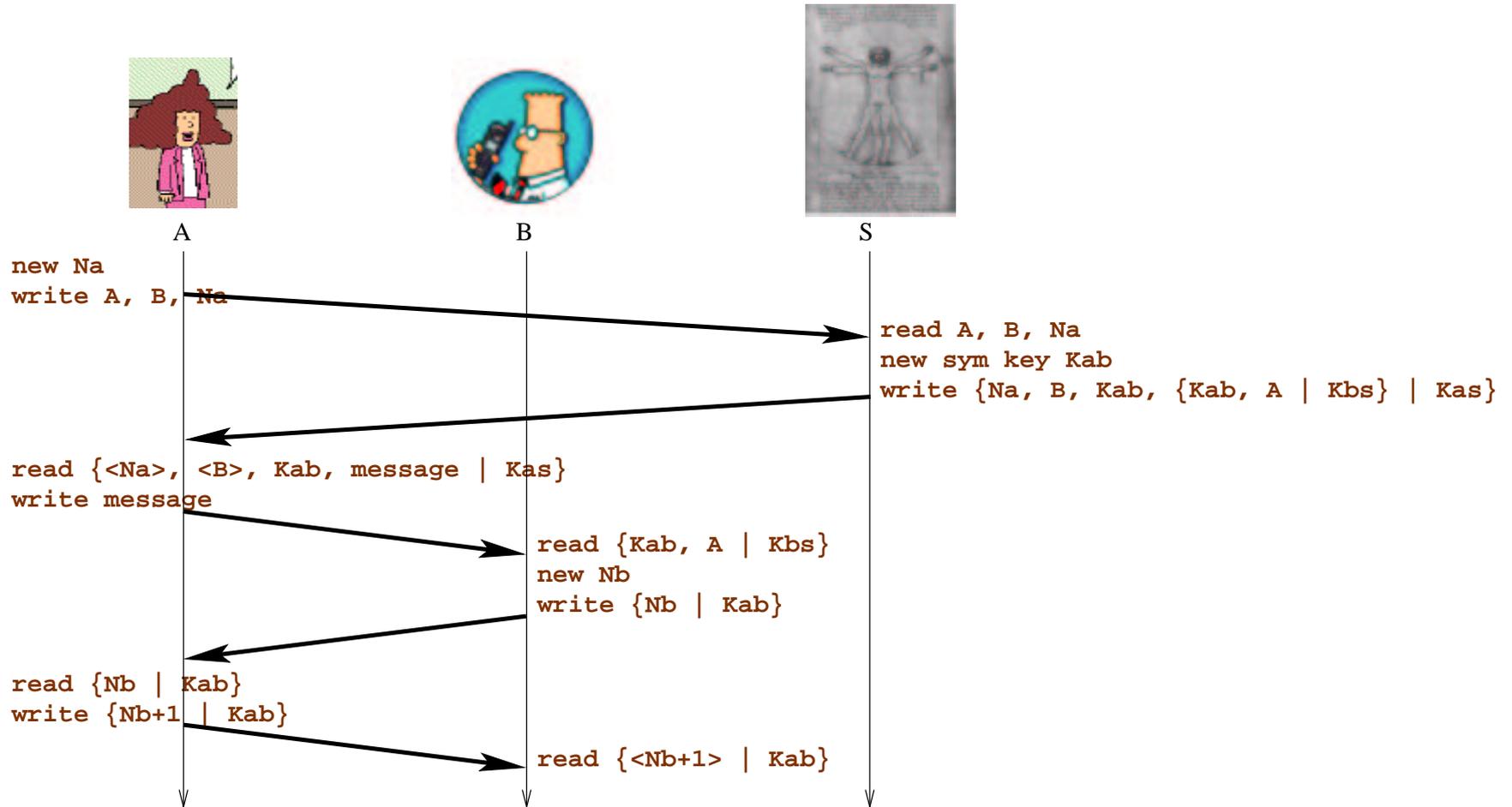
$(n + 1)$ -uplets $s = (\text{pc}, \text{valeur } x_1, \dots, \text{valeur } x_n)$.

Transitions :

$$s \xrightarrow{\text{condition}} s'$$

`trans(S, S') :- condition.`

Retour sur Needham-Schroeder à clés privées



Sources d'infini — 1. Les messages

État de A : $s_A = (pc, N_a, A, B, K_{as}, K, M)$.

Transitions :

```
transA((1,_,A,B,Kas,K,M), (2,Na,A,B,Kas,K,M)) :-  
    new(Na), write((A, B, Na)).
```

```
transA((2,Na,A,B,Kas,_,_), (3,Na,A,B,Kas,Kab,M)) :-  
    read({(Na, B, Kab, M) | Kas}).
```

```
transA((3,Na,A,B,Kas,Kab,M), (4,Na,A,B,Kas,Kab,M)) :-  
    write (M).
```

...

Mise en parallèle

Le prédicat trans_A décrit bien un principal A .
Mais on veut décrire $A \parallel B \parallel S$:

$\text{global}((\text{Etat}_A, \text{Etat}_B, \text{Etat}_S), (\text{Etat}_A', \text{Etat}_B, \text{Etat}_S)) :-$
 $\text{trans}_A(\text{Etat}_A, \text{Etat}_A')$.

$\text{global}((\text{Etat}_A, \text{Etat}_B, \text{Etat}_S), (\text{Etat}_A, \text{Etat}_B', \text{Etat}_S)) :-$
 $\text{trans}_B(\text{Etat}_B, \text{Etat}_B')$.

$\text{global}((\text{Etat}_A, \text{Etat}_B, \text{Etat}_S), (\text{Etat}_A, \text{Etat}_B, \text{Etat}_S')) :-$
 $\text{trans}_S(\text{Etat}_S, \text{Etat}_S')$.

Les exécutions globales

```
/* Cloture reflexive-transitive: */  
exec((EtatA,EtatB,EtatC), (EtatA,EtatB,EtatC)).  
exec((EtatA,EtatB,EtatC), (EtatA',EtatB',EtatC')) :-  
    global ((EtatA,EtatB,EtatC),  
            (EtatA1,EtatB1,EtatC1)),  
    exec ((EtatA1,EtatB1,EtatC1),  
          (EtatA',EtatB',EtatC')).
```

Accessibilité

- Demandons à Prolog si on peut atteindre
- un état global où la **propriété dangereuse** P est réalisée (ex : non-secret de K_{ab})
 - à partir d'un état vérifiant des **hypothèses initiales** H (secret de K_{as}, \dots)

```
? H ((EtatA0,EtatB0,EtatS0)),  
   exec((EtatA0,EtatB0,EtatS0), (EtatA,EtatB,EtatS)),  
   P((EtatA,EtatB,EtatS)).
```

(Reste à définir new, write, read...)

Sources d'infini — 2. Les nonces

Les **nonces** : `new(Na)` est censé créer une nouvelle donnée de base et la mettre dans `Na`.

Sources d'infini — 2. Les nonces

Les **nonces** : `new(Na)` est censé créer une nouvelle donnée de base et la mettre dans `Na`.

On fait une **approximation** : si deux nonces sont créés par le même processus au même pc, on les identifie.

→ remplacer la variable `Na` par une cste `na_A_2`.

Sources d'infini — 2. Les nonces

Les **nonces** : `new(Na)` est censé créer une nouvelle donnée de base et la mettre dans `Na`.
On fait une **approximation** : si deux nonces sont créés par le même processus au même pc, on les identifie.

→ remplacer la variable `Na` par une cste `na_A_2`.
→ **correct** pour la vérification de propriétés d'accessibilité.

(Autres approches : logique linéaire [Lincoln].)

Skolémiser les nonces, en Prolog

```
transA((1,_,A,B,Kas,K,M), (2,Na,A,B,Kas,K,M)) :-  
    new(Na), write((A, B, Na)).
```

...

→

```
transA((1,_,A,B,Kas,K,M), (2,na_A_2,A,B,Kas,K,M)) :-  
    write((A, B, na_A_2)).
```

...

Sources d'infini — 3. Les canaux

Les **canaux de communication** contiennent une quantité **non bornée** de messages en attente, de **tailles non bornées**.

Le problème est classique (automates à files, etc.), mais aucune solution classique ne s'applique.

Sources d'infini — 3. Les canaux

Les **canaux de communication** contiennent une quantité **non bornée** de messages en attente, de **tailles non bornées**.

Le problème est classique (automates à files, etc.), mais aucune solution classique ne s'applique.

Plus compliqué : l'intrus **détourne** les canaux, et :

- **intercepte** les messages,
- **fabrique** de nouveaux messages,
- chiffre, déchiffre, jette, duplique, ...

Le modèle de [Dolev Yao83]

On modélise le cas le pire pour la sécurité, où :

- tout message émis est envoyé directement à l'intrus ;
- tout message reçu est fabriqué directement par l'intrus ;
- l'intrus peut chiffrer, déchiffrer, etc., mais pas décrypter. (En particulier, il peut fabriquer des messages de taille **arbitraire**.)

En particulier, ceci subsume le comportement normal des canaux...

Capacité déductive des intrus

$$\frac{}{E, M \vdash M} (Ax)$$

$$\frac{E \vdash M \quad E \vdash K}{E \vdash \{M \mid K\}} (CryI)$$

$$\frac{E \vdash \{M \mid K\} \quad E \vdash K^{-1}}{E \vdash M} (CryE)$$

$$\frac{E \vdash M_1 \quad \dots \quad E \vdash M_n}{E \vdash (M_1, \dots, M_n)} (TupI)$$

$$\frac{E \vdash (M_1, \dots, M_n)}{E \vdash M_i} (TupE_i), 1 \leq i \leq n$$

La modélisation en clauses de Horn

- Écrire des clauses pour $\text{ded}(E, M)$ ($\equiv E \vdash M$);
- $\text{write}(M)$ = ajouter M à E ;
- $\text{read}(M)$ = imposer $\text{ded}(E, M)$.

```
transA((1,_,A,B,Kas,K,M), E,  
       (2,na_A_2,A,B,Kas,K,M), cons((A, B, na_A_2),  
transA((2,Na,A,B,Kas,_,_), E,  
       (3,Na,A,B,Kas,Kab,M), E) :-  
       ded(E, {(Na, B, Kab, M) | Kas})).  
transA((3,Na,A,B,Kas,Kab,M), E,  
       (4,Na,A,B,Kas,Kab,M), cons(M,E)).
```

...

Malheureusement,

Un intérêt de Prolog ici était de pouvoir **simuler** un protocole.

Malheureusement,

Un intérêt de Prolog ici était de pouvoir **simuler** un protocole.

Avec notre modélisation de l'intrus, ça ne fonctionne plus :

Prolog **boucle**

... même sans boucle dans le protocole ...

Malheureusement,

Un intérêt de Prolog ici était de pouvoir **simuler** un protocole.

Avec notre modélisation de l'intrus, ça ne fonctionne plus :

Prolog **boucle**

... même sans boucle dans le protocole ...

(La stratégie de recherche de preuve Prolog recherche des déductions par \vdash arbitrairement profondes.)

Sources d'infini — 4. Les instances

A , B , S sont des rôles (des spécifications), qui peuvent avoir un nombre arbitraire d'instances, les principaux.

Important :

- Les serveurs S opèrent en parallélisme non borné en pratique ;
- Certaines attaques n'apparaissent qu'à partir du moment où suffisamment de principaux sont en parallèle (Ex : Otway-Rees).

Parallélisme non borné et complicité

Une solution élégante

[DebbabiMejriTawbiYahmadi97,JGL00] :

approximer les comportements des principaux S en multi-session parallèle par de **nouvelles règles** pour \vdash .

Par ex., si S fait `read M ; ... ; write M'`, on ajoute :

$$\frac{E \vdash M}{E \vdash M'}$$

→ considère S comme un **complice** de l'intrus.

Exemple de complicité (Otway-Rees)

$$\begin{array}{l}
 \text{read}(M, A, B, \\
 \quad \{N_a, \langle M \rangle, \langle A \rangle, \langle B \rangle \mid K_{as}\}, \\
 \quad \{N_b, \langle M \rangle, \langle A \rangle, \langle B \rangle \mid K_{bs}\}) \sim \\
 \text{new_key}K_{ab}; \\
 \text{write}(M, \{N_a, K_{ab} \mid K_{as}\}, \\
 \quad \{N_b, K_{ab} \mid K_{bs}\});
 \end{array}
 \quad
 \begin{array}{l}
 E \mapsto (M, A, B, \\
 \quad \{N_a, M, A, B \mid K_{as}\}, \\
 \quad \{N_b, M, A, B \mid K_{bs}\}) \\
 \hline
 E \mapsto (M, \{N_a, k_{S_2} \mid K_{as}\}, \\
 \quad \{N_b, k_{S_2} \mid K_{bs}\})
 \end{array}$$

4. Comment vérifier un protocole cryptographique ?

Rappel

Notre protocole π

- + les hypothèses initiales H
- + la propriété P dangereuse à surveiller

(gros) ensemble S de clauses de Horn.

Fait P est fausse de π sous l'hypothèse H
(sécurité)

$\iff S$ non-contradictoire.

(enfin, au moins \Leftarrow ... c'est l'essentiel.)

Comment tester si un ensemble de clauses est contradictoire ?

Beaucoup de méthodes, c'est le thème de la **démonstration automatique** :

- Chercher un contre-exemple par énumération exhaustive [DavisPutnam60] ;
- Méthode de tableaux [Beth55, Hintikka55, Smullyan68], élimination de modèles [Loveland69], méthode des connexions [Bibel81, Andrews81], ...
- Résolution (et variantes) [Robinson65].

La résolution (pour les clauses de Horn)

Une seule règle (pensez clôture transitive) :

$$P(t) : -Q_1(t_1), \dots, Q_i(t_i), \dots, Q_m(t_m).$$

$$Q_i(u) : -R_1(u_1), \dots, R_n(u_n).$$

$$\frac{}{P(t\sigma) : -Q_1(t_1\sigma), \dots, R_1(u_1\sigma), \dots, R_n(u_n\sigma), \dots, Q_m(t_m\sigma).} \quad (Res)$$

$$\text{avec } \sigma = mgu(t_i, u).$$

La résolution (pour les clauses de Horn)

Une seule règle (pensez clôture transitive) :

$$P(t) : -Q_1(t_1), \dots, Q_i(t_i), \dots, Q_m(t_m).$$

$$Q_i(u) : -R_1(u_1), \dots, R_n(u_n).$$

$$\frac{}{P(t\sigma) : -Q_1(t_1\sigma), \dots, R_1(u_1\sigma), \dots, R_n(u_n\sigma), \dots, Q_m(t_m\sigma).} \text{ (Res)}$$

$$\text{avec } \sigma = mgu(t_i, u).$$

Thm [Correction, complétude] Un ensemble S de clauses de Horn est **contradictoire** \iff on peut **déduire false**. de S par la seule règle (*Res*).

Un exemple

`pair(0).` (1)

`pair(s(s(X))) :- pair(X).` (2)

`false :- pair(s10(0)).` (3)

Un exemple

`pair(0).` (1)

`pair(s(s(X))) :- pair(X).` (2)

`false :- pair(s10(0)).` (3)

$(1), (2) \implies$ `pair(s2(0)).` (4)

Un exemple

$$\text{pair}(0). \quad (1)$$

$$\text{pair}(s(s(X))) \quad :- \quad \text{pair}(X). \quad (2)$$

$$\text{false} \quad :- \quad \text{pair}(s^{10}(0)). \quad (3)$$

$$(1), (2) \implies \text{pair}(s^2(0)). \quad (4)$$

$$(2), (2) \implies \text{pair}(s^4(X)) \quad :- \quad \text{pair}(X). \quad (5)$$

Un exemple

$$\text{pair}(0). \quad (1)$$

$$\text{pair}(s(s(X))) \quad :- \quad \text{pair}(X). \quad (2)$$

$$\text{false} \quad :- \quad \text{pair}(s^{10}(0)). \quad (3)$$

$$(1), (2) \implies \text{pair}(s^2(0)). \quad (4)$$

$$(2), (2) \implies \text{pair}(s^4(X)) \quad :- \quad \text{pair}(X). \quad (5)$$

$$(5), (5) \implies \text{pair}(s^8(X)) \quad :- \quad \text{pair}(X). \quad (6)$$

Un exemple

$$\text{pair}(0). \quad (1)$$

$$\text{pair}(s(s(X))) \quad :- \quad \text{pair}(X). \quad (2)$$

$$\text{false} \quad :- \quad \text{pair}(s^{10}(0)). \quad (3)$$

$$(1), (2) \implies \text{pair}(s^2(0)). \quad (4)$$

$$(2), (2) \implies \text{pair}(s^4(X)) \quad :- \quad \text{pair}(X). \quad (5)$$

$$(5), (5) \implies \text{pair}(s^8(X)) \quad :- \quad \text{pair}(X). \quad (6)$$

$$(4), (6) \implies \text{pair}(s^{10}(0)). \quad (7)$$

Un exemple

`pair(0).` (1)

`pair(s(s(X))) :- pair(X).` (2)

`false :- pair(s10(0)).` (3)

(1), (2) \implies `pair(s2(0)).` (4)

(2), (2) \implies `pair(s4(X)) :- pair(X).` (5)

(5), (5) \implies `pair(s8(X)) :- pair(X).` (6)

(4), (6) \implies `pair(s10(0)).` (7)

(7), (3) \implies `false.`

Un exemple, modifié

`pair(0).`

`pair(s(s(X))) :- pair(X).`

`false :- pair(s11(0)).`

...

`pair(s2n(X)) :- pair(X).`

`pair(s2n(0)).`

`false :- pair(s2n+1(0)). (0 ≤ n ≤ 5)`

...

Que s'est-il passé ?

La résolution boucle \rightarrow nombre **infini** de clauses.

`false`. jamais engendré $\implies S$ n'est pas
contradictoire... mais on ne le saura jamais !

Que s'est-il passé ?

La résolution boucle \rightarrow nombre **infini** de clauses.

`false`. jamais engendré $\implies S$ n'est pas contradictoire... mais on ne le saura jamais !

Thm Le problème “un ensemble de clauses de Horn est-il contradictoire ?” est **indécidable**.

Preuve On peut même coder une machine de Turing universelle par un programme de la forme [DevienneLebègueRoutierWürtz94] :

`P(t0) . P(t2) : -P(t1) . false : -P(t3) .`

Que peut-on faire de mieux ?

Pour limiter l'espace de recherche de la résolution :

- **Raffinements (complets) de la résolution** : unit-résolution, input-résolution, résolution linéaire, hyperrésolution, résolution sémantique, résolution ordonnée, cg-résolution, lock-résolution, ...
Note : Prolog = S(election) L(inear) D(efinite)-résolution \sim input-résolution.
- **Stratégies d'effacement** : des tautologies, des clauses pures, des clauses subsumées, ...

La résolution ordonnée

Pareil qu'avant :

$$P(t) : -Q_1(t_1), \dots, Q_i(t_i), \dots, Q_m(t_m).$$

$$Q_i(u) : -R_1(u_1), \dots, R_n(u_n).$$

$$\frac{}{P(t\sigma) : -Q_1(t_1\sigma), \dots, R_1(u_1\sigma), \dots, R_n(u_n\sigma), \dots, Q_m(t_m\sigma).} \quad (Res)$$

... mais uniquement si $Q_i(t_i)$, $Q_i(u)$ **maximaux** dans leurs clauses (pour un ordre stable $>$).

Thm [Correction, complétude] S contradictoire \iff false. déductible.

L'exemple modifié, par résolution ordonnée

`pair(0).` (1)

`pair(s(s(X))) :- pair(X).` (2)

`false :- pair(s11(0)).` (3)

(2), (3) \implies `false :- pair(s9(0)).` (4)

(4), (3) \implies `false :- pair(s7(0)).` (5)

(5), (3) \implies `false :- pair(s5(0)).` (6)

(6), (3) \implies `false :- pair(s3(0)).` (7)

(7), (3) \implies `false :- pair(s1(0)).` (8)

Stop. \implies non-contradiction.

Et alors ?

La résolution ordonnée **termine** sur ce type d'exemples.

Peut-on identifier une forme de clauses pour laquelle la résolution ordonnée terminerait toujours ?

On va voir qu'il y a une réponse positive liée à la théorie des **automates**.

Sous-classes décidables de la logique du premier ordre

Observation Si \mathcal{C} est une classe d'ensembles de clauses S telle que :

l'ensemble des clauses déductibles de tout S fini
 $\in \mathcal{C}$ par résolution ordonnée est fini

alors \mathcal{C} est une classe **décidable**.

Ex : $\mathcal{C} = \{\text{clauses } P(\text{gros terme}) : - P_i(\text{petits})\}.$

Quelques résultats non triviaux

Corollaire la **classe monadique** [Ackermann54] est décidable. ($F ::= P(x) | F \wedge F | \neg F | \forall x \cdot F | \exists x \cdot F$.)

Preuve Mettre en forme clausale et utiliser l'observation précédente.

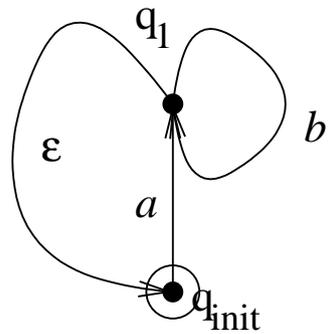
(Nécessite deux ou trois autres astuces...)

Corollaire [BachmairGanzingerWaldmann92] La satisfaction de **contraintes ensemblistes** est décidable.

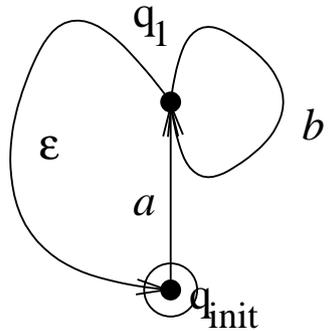
(Ensembles $E ::= X | E \cap E | \complement E | f(E, \dots, E) | f_i^{-1}(E)$,

Contraintes = conjonctions finies de $E \subseteq E$.)

Automates



Automates et clauses de Horn



`init(●).`

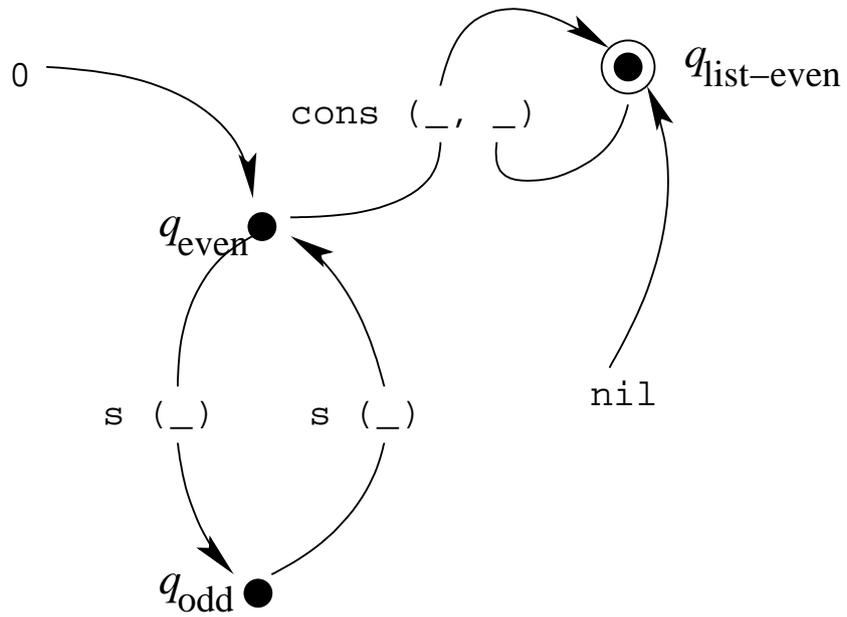
`q1(a(X)) :- init(X).`

`q1(b(X)) :- q1(X).`

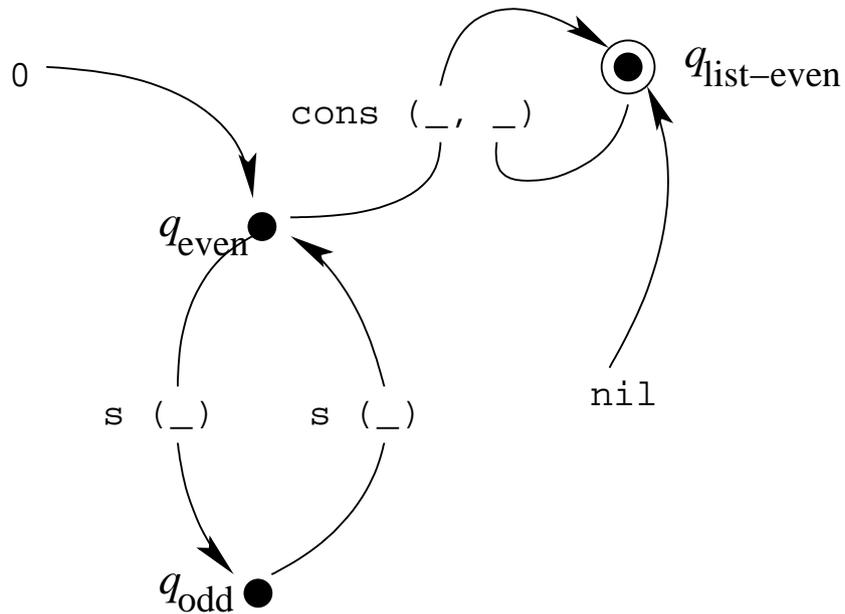
`init(X) :- q1(X).`

(mot $a_1 a_2 \dots a_n = a_n(\dots(a_2(a_1(\bullet))))\dots)$)

Automates d'arbres

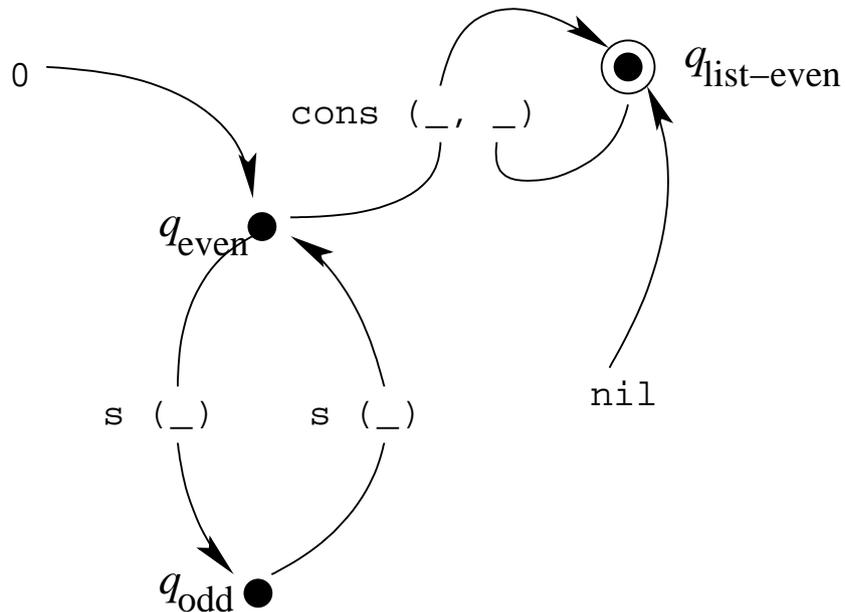


Automates d'arbres et clauses de Horn



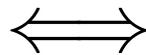
```
even(0).  
odd(s(X)) :- even(X).  
even(s(X)) :- odd(X).  
list-even(cons(X,Y)) :-  
    even(X), list-even(Y)  
list-even(nil).
```

Automates d'arbres et clauses de Horn



```
even(0).  
odd(s(X)) :- even(X).  
even(s(X)) :- odd(X).  
list-even(cons(X,Y)) :-  
    even(X), list-even(Y)  
list-even(nil).
```

Non-vacuité



Contradiction

(de $q_{\text{list-even}}$)

(avec $\text{false} \text{ :- list-even}(X)$)

Décidable !

La forme logique des automates

Un **automate d'arbres** est un ensemble de clauses de la forme

$$P(f(X_1, \dots, X_n) : \neg P_1(X_1), \dots, P_n(X_n))$$

La forme logique des automates

Un **automate d'arbres** est un ensemble de clauses de la forme

$$P(f(X_1, \dots, X_n) : \neg P_1(X_1), \dots, P_n(X_n))$$

ϵ -transitions :

$$P(X) : \neg Q(X)$$

La forme logique des automates

Un **automate d'arbres** est un ensemble de clauses de la forme

$$P(f(X_1, \dots, X_n) : \neg P_1(X_1), \dots, P_n(X_n))$$

ϵ -transitions : $P(X) : \neg Q(X)$

\wedge -transitions (**alternants**) : $P(X) : \neg Q_1(X), Q_2(X)$

La forme logique des automates

Un **automate d'arbres** est un ensemble de clauses de la forme

$$P(f(X_1, \dots, X_n) : \neg P_1(X_1), \dots, P_n(X_n))$$

ϵ -transitions : $P(X) : \neg Q(X)$

\wedge -transitions (**alternants**) : $P(X) : \neg Q_1(X), Q_2(X)$

pushdown : $P(X_i) : \neg Q(f(X_1, \dots, X_n))$

La forme logique des automates

Un **automate d'arbres** est un ensemble de clauses de la forme

$$P(f(X_1, \dots, X_n) : -P_1(X_1), \dots, P_n(X_n))$$

ϵ -transitions : $P(X) : -Q(X)$

\wedge -transitions (**alternants**) : $P(X) : -Q_1(X), Q_2(X)$

pushdown : $P(X_i) : -Q(f(X_1, \dots, X_n))$

égalités entre frères : poser $X_i \equiv X_j$

La forme logique des automates

Un **automate d'arbres** est un ensemble de clauses de la forme

$$P(f(X_1, \dots, X_n) : -P_1(X_1), \dots, P_n(X_n))$$

ϵ -transitions : $P(X) : -Q(X)$

\wedge -transitions (**alternants**) : $P(X) : -Q_1(X), Q_2(X)$

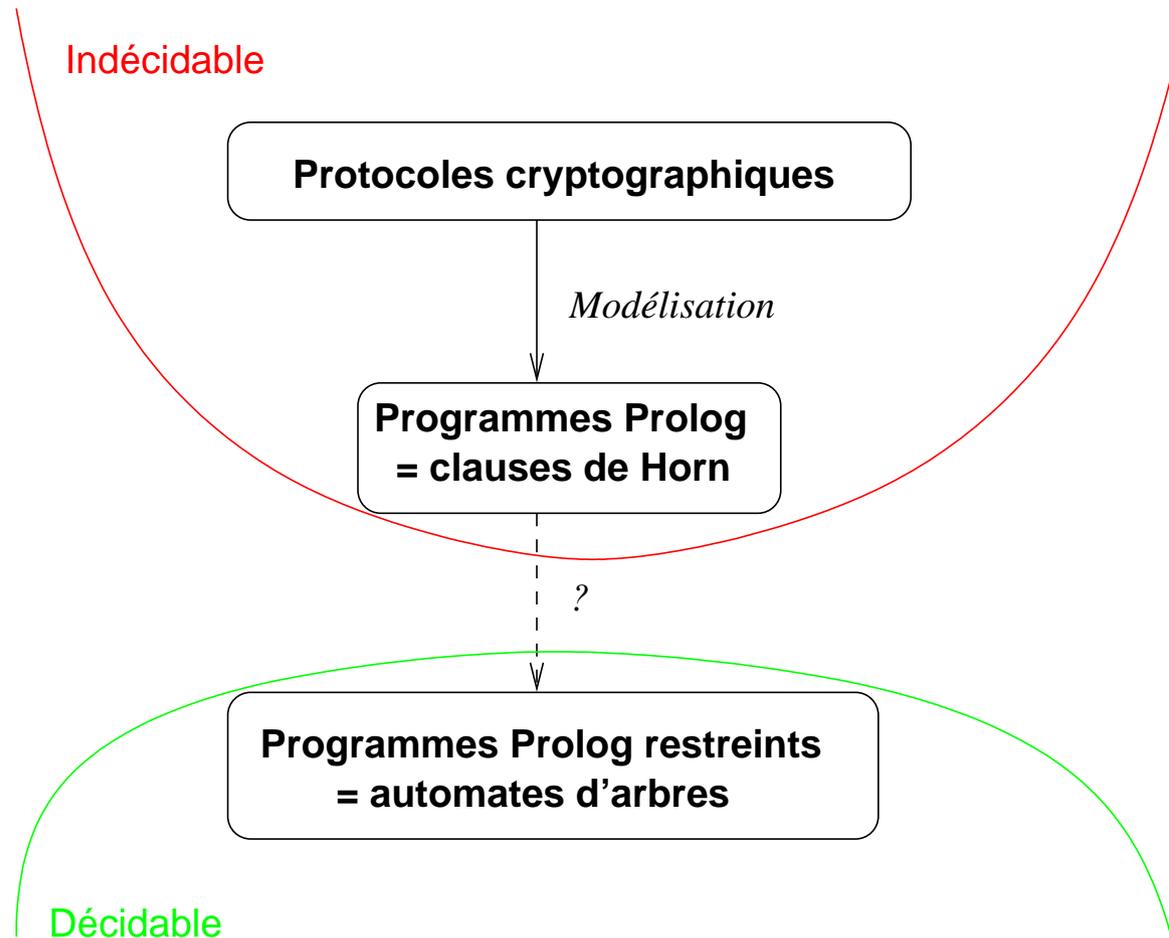
pushdown : $P(X_i) : -Q(f(X_1, \dots, X_n))$

égalités entre frères : poser $X_i \equiv X_j$

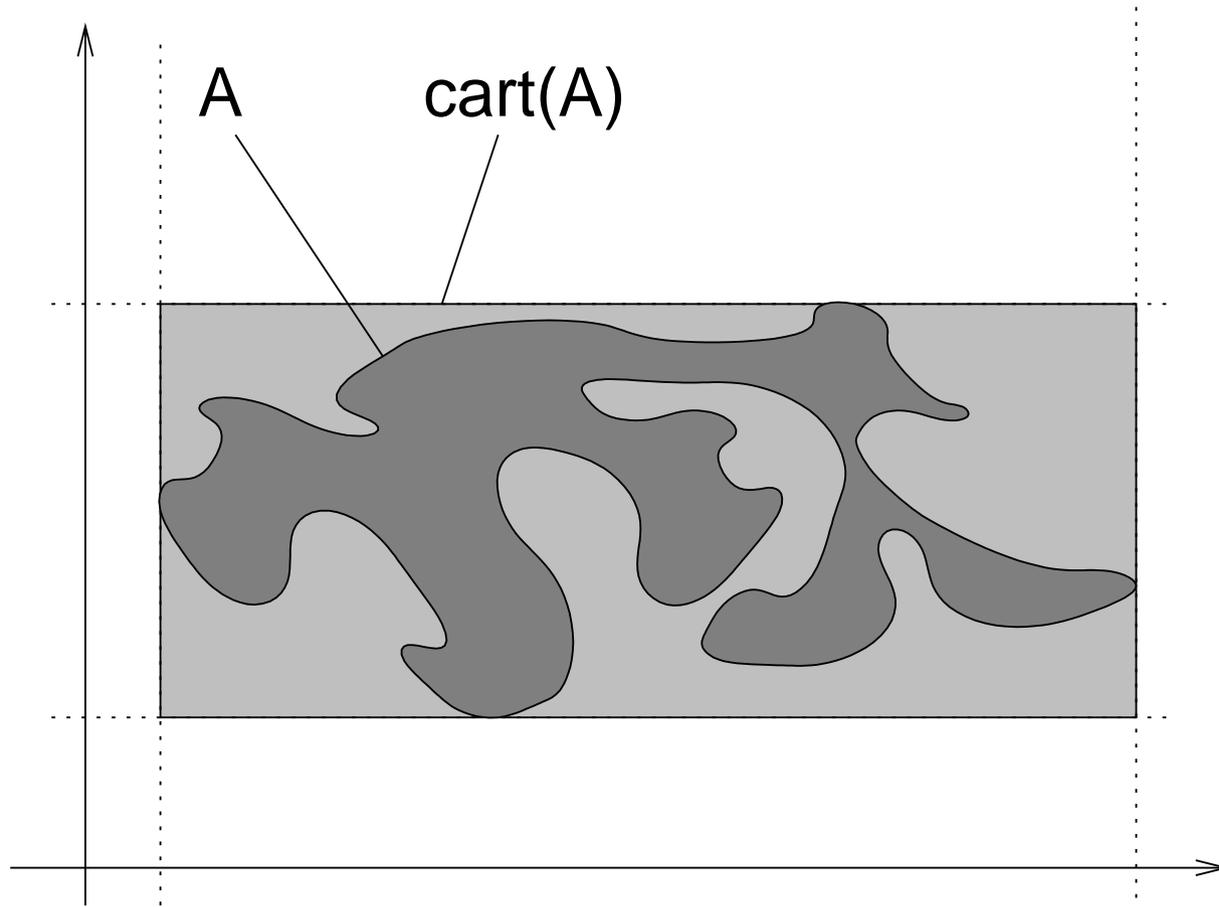
Thm La non-vacuité d'automates alternants avec contraintes d'égalité entre frères est décidable.

Preuve Par résolution ordonnée.

C'est très bien tout ça, mais où en sommes-nous ?



L'idée de l'approximation cartésienne



L'approximation cartésienne de clauses de Horn [CharatonikPodelski98]

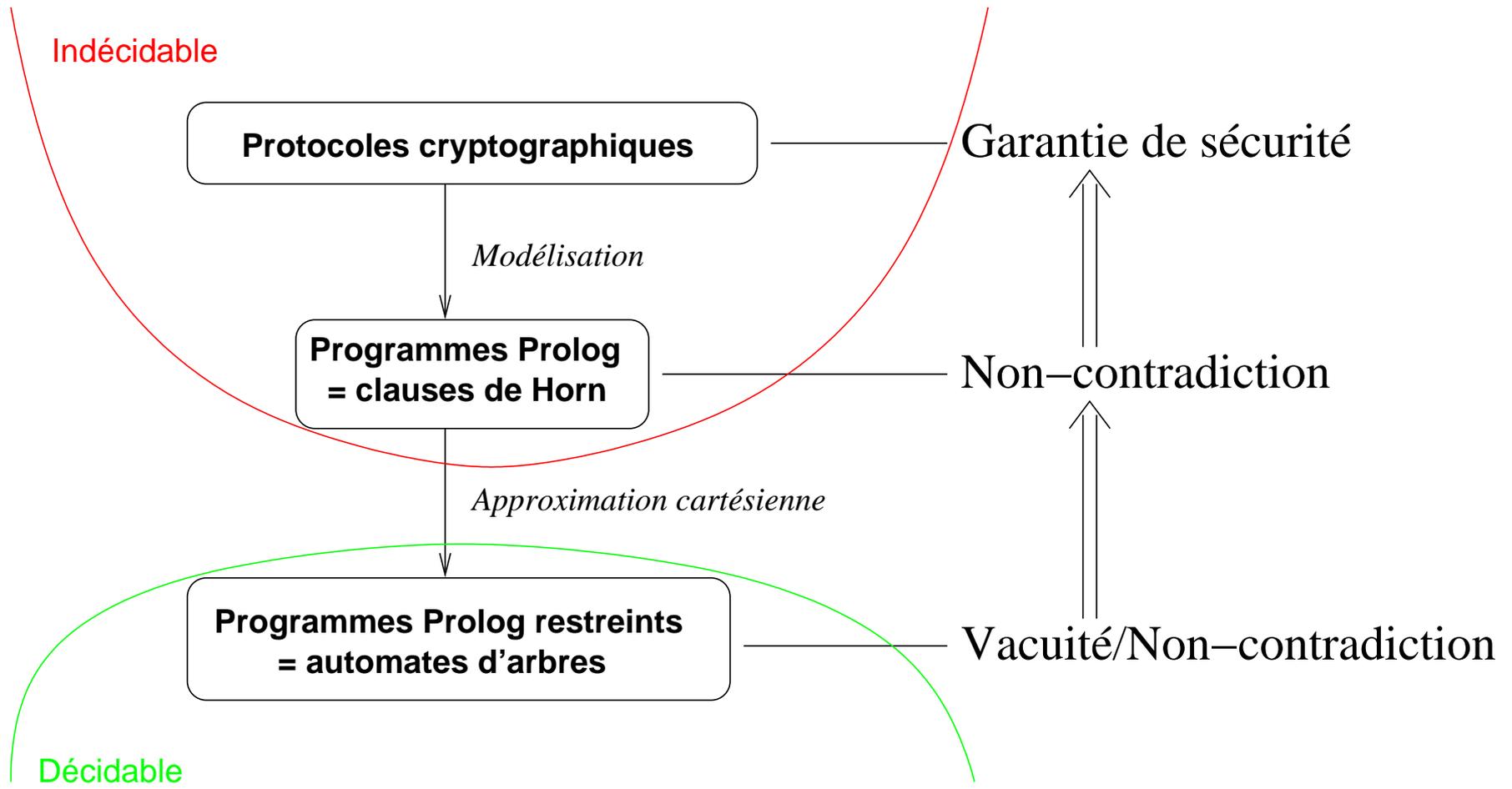
$$P(t[X_1, \dots, X_n]) : -Q(u)$$

↓

$$\left\{ \begin{array}{l} P(t[X_1, \dots, X_n]) \quad :- \quad \text{typeX1}(X_1), \dots, \text{typeXn}(X_n) \\ \text{typeX1}(X_1) \quad :- \quad Q(u) \\ \dots \\ \text{typeXn}(X_n) \quad :- \quad Q(u) \end{array} \right.$$

... est un **automate d'arbres**. (modulo quelques détails)

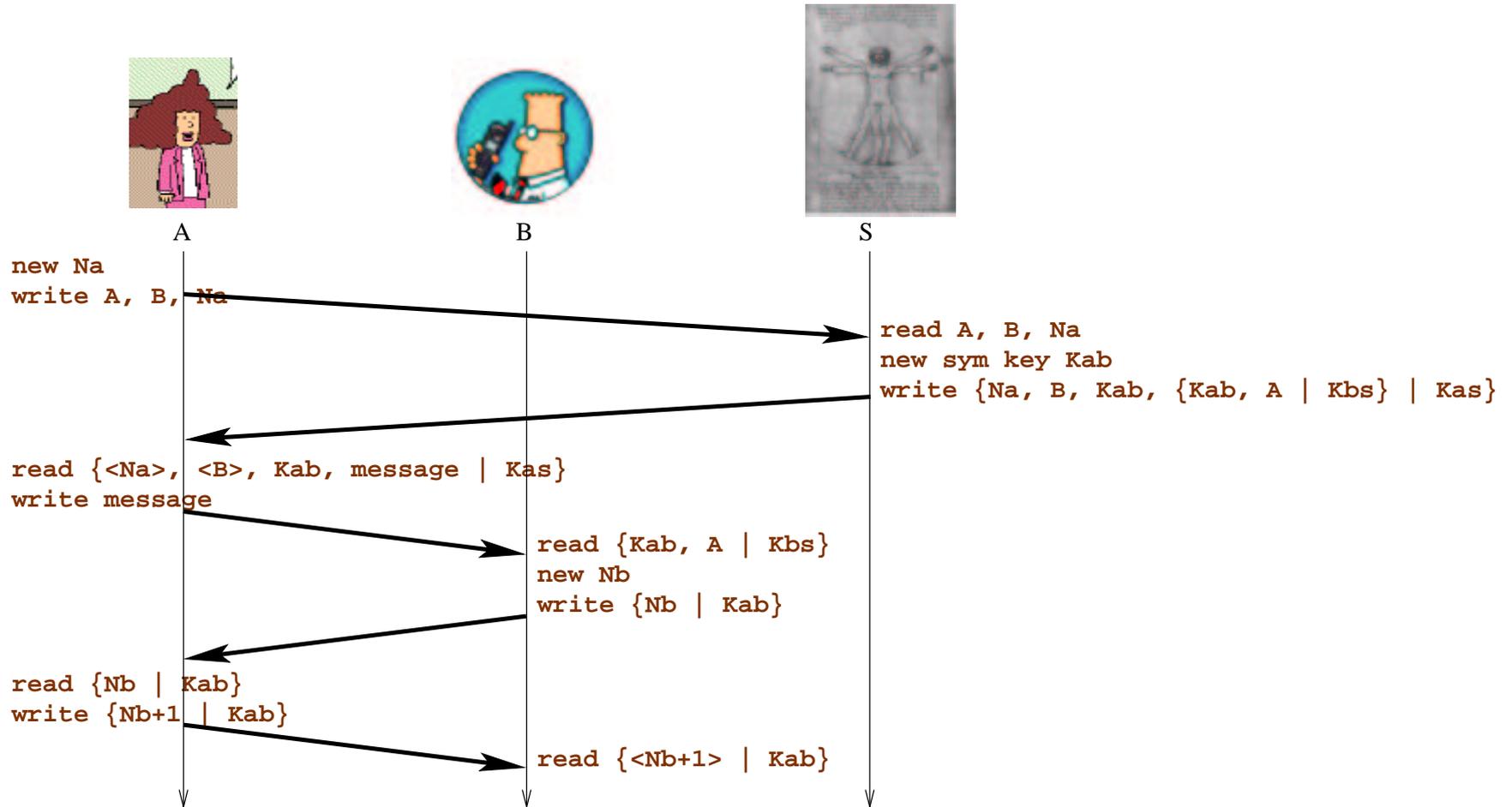
Et voilà !



Améliorations possibles

- Automates plus fins (**à une mémoire** [ComonCortierMitchell01] \implies sous-classe décidable large de protocoles crypto) ;
- Ne traiter par automates que les parties état initiaux et déductions de l'intrus (pas de perte d'information ici), et "exécuter symboliquement" le programme (**interprétation abstraite** [CousotCousot77], cf. [Monniaux99,JGL00]).

Testons tout ça [NeedhamSchroeder78]

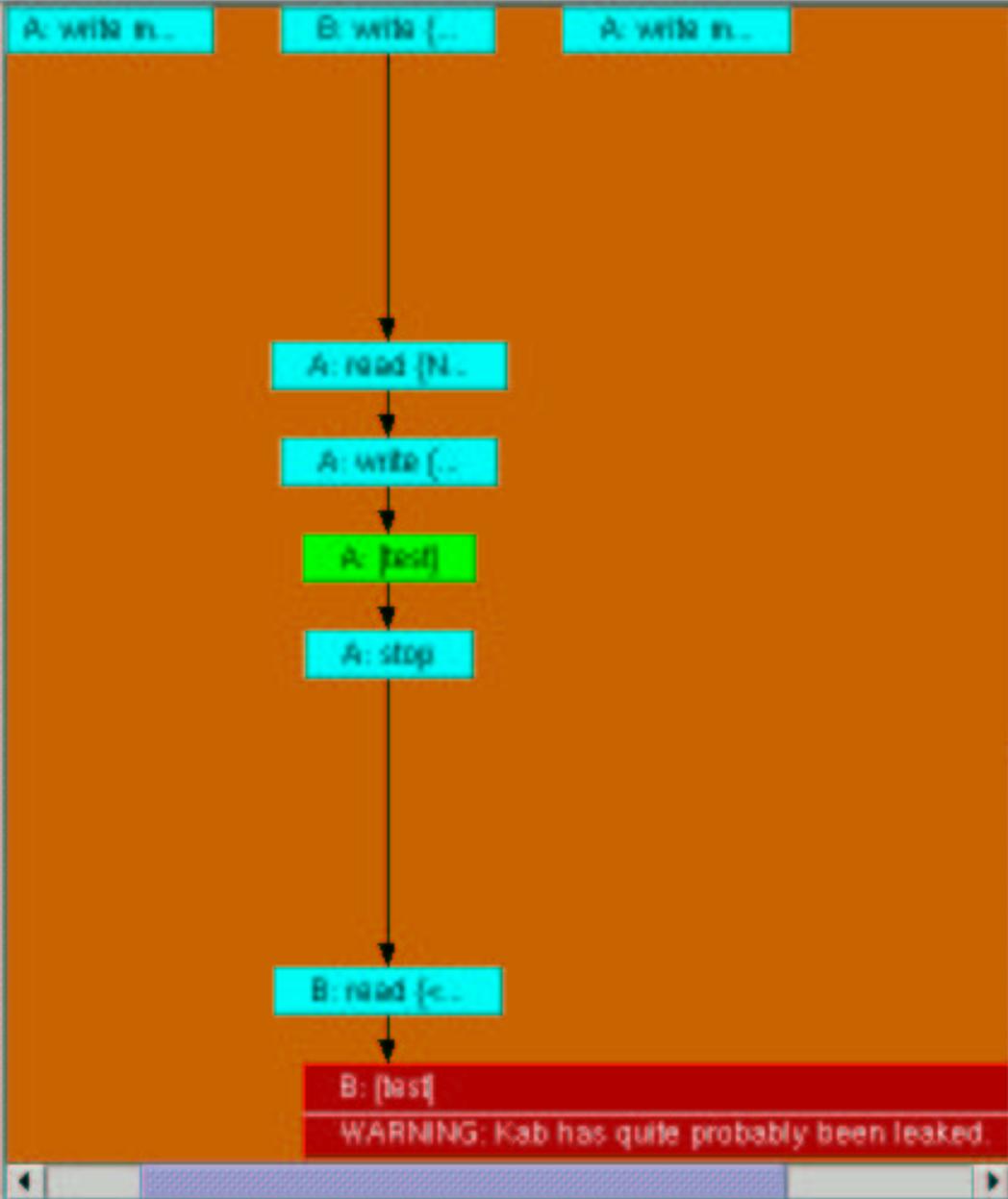


Applet

/jg/CPV/NeedhamSchroederSharedKeyProtocol.cpv

```
role A (me:D, B:D, S:D) {  
  new Na;  
  write me, B, Na; /* to S */  
  read {<Na>, <B>, Kab:key, message  
    | sk me/S}; /* from S */  
  [secret Kab];  
  write message; /* to B */  
  read {Nb:D | Kab}; /* from B */  
  write {nudge Nb | Kab}; /* to B */  
  [secret Kab]  
}
```

```
role B (me:D, S:D) {  
  read {Kab:key, A:D | sk me/S}; /* from A */  
  new Nb;  
  write {Nb | Kab}; /* to A */  
  read {<nudge Nb> | Kab}; /* from A */  
  [secret Kab]  
}
```



Finished in 0.77 s.

Buttons: About... Load... Save... Send

Applet started.

Le paysage final

