Vérification de protocoles cryptographiques: la logique à la rescousse!

Jean Goubault-Larrecq[1]

1: LSV/CNRS UMR 8643 et ENS Cachan,
61, avenue du président-Wilson,
94235 Cachan CEDEX, France
goubault@lsv.ens-cachan.fr

Résumé

La vérification de propriétés de sécurité de protocoles cryptographiques est une activité délicate et importante. Après avoir passé en revue les principes de base de cette activité, nous montrerons que la logique du premier ordre permet à la fois une formalisation adéquate des protocoles cryptographiques et de certaines de leurs propriétés les plus courantes (secret, authentication), et se prête bien à des mécanismes d'abstraction *automatique* qui en permettent une vérification automatique aisée. Le lien entre logique du premier ordre et automates d'arbres y est ici fondamental.

1. Introduction

La sécurité des systèmes d'information est aujourd'hui un domaine en pleine effervescence, voire à la mode. Parmi les différents aspects de la sécurité informatique, la cryptologie et l'étude des protocoles cryptographiques a pris une importance considérable. La cryptologie est originellement l'étude des moyens permettant de transmettre des messages secrets, et remonte à l'antiquité. Depuis, la portée des méthodes cryptologiques s'est considérablement étendue, pour inclure des préoccupations de garantie d'authenticité et d'intégrité des messages reçus par exemple. Mais aussi, comme la sécurité des messages repose sur l'usage de clés de chiffrement ou de signature, la cryptologie a dû aussi traiter des problèmes de distribution sécurisée de clés à des tiers, de révocation de clés supposées peu fiables, de récupération de clés perdues par des utilisateurs réputés honnêtes, etc.

La cryptologie est donc un domaine riche [50], mais qui s'est longtemps limité essentiellement aux applications militaires. On peut considérer que le surgissement d'intérêt pour la cryptographie dans les années 1990 et 2000 est dû au développement du commerce électronique. Ce dernier terme recouvre pour commencer les échanges chiffrés d'information qui ont lieu lorsque l'on retire de l'argent dans un distributeur de billets à l'aide d'une carte bancaire, où l'ordinateur qu'est le distributeur de billets s'engage dans un dialogue avec la carte du client et possiblement avec un serveur de la banque pour vérifier que l'utilisateur de la carte est honnête, dispose de la somme demandée sur son compte, et ne pourra pas récupérer les billets demandés sans que son compte en soit débité, notamment. Le commerce électronique recouvre aussi, par exemple, les paiements par carte à l'aide de terminaux portables, dans les supermarchés ou les restaurants, où l'utilisateur doit confirmer son identité en entrant son code PIN à quatre chiffres. Toutes ces applications demandent des garanties de sécurité élevées, portant sur des propriétés de secret et d'authenticité comme ci-dessus, mais aussi de nombreuses autres propriétés, parmi lesquelles, la non-duplication (des factures, par exemple, dans l'intérêt du client), de non-révocation (des commandes, dans l'intérêt du commerçant), d'anonymat (dans le cadre des micro-paiements, où l'on souhaite offrir au client l'équivalent électronique du billet de banque, qui ne porte pas le nom de son utilisateur, contrairement au chèque par exemple), etc. La cryptologie aurait cependant pu rester un domaine relativement peu connu du grand public sans l'Internet, le World-Wide Web et en particulier le protocole SSL introduit par Netscape et son successeur TLS [18]. En 2000, il est possible de faire des achats depuis un browser Web, qui négocie un protocole garantissant un certain niveau de secret et d'authentification avec le serveur d'un commerçant, et l'utilise ensuite pour chiffrer les négociations de produits et de tarifs conduites par l'utilisateur du browser avec le commerçant.

Or il se trouve que si, depuis les années 1980, on dispose d'algorithmes cryptographiques suffisamment sûrs, il n'en est pas de même, loin s'en faut, des protocoles cryptographiques. Nous le montrons au travers d'un exemple en section 2, et y illustrons les particularités de la sécurité cryptographique. Nous montrons ensuite comment on peut modéliser et soit vérifier des protocoles cryptographiques, soit détecter des attaques, en section 3. Nous serons ici relativement partiaux, et nous contenterons de présenter une variante d'une approche due à Dolev et Yao, qui a le mérite d'être simple, de se prêter à l'automatisation, et de se concentrer sur les problèmes de sécurité principaux. Cette modélisation se prête cependant peu à l'automatisation, et nous montrons en section 4 comment formaliser différemment les mêmes protocoles, de sorte que des techniques d'approximation standard, inspirées de [24], permettent d'obtenir des preuves automatiques de sécurité. Nous concluons en section 5.

2. Un exemple de protocole cryptographique

2.1. Premières tentatives

Tentons d'imaginer comment permettre à deux principaux (ordinateurs, cartes à puce, éventuellement aussi êtres humains) A et B d'obtenir une clé K_{ab} qu'eux seuls connaissent, et qui leur permettra de chiffrer les messages qu'ils désirent échanger par la suite. Une solution simple est de faire créer K_{ab} par A, qui l'envoie en clair à B. Ceci a cependant de nombreux défauts. D'abord, la clé K_{ab} ne peut pas être considérée secrète : il est facile pour un intrus I d'espionner la ligne de communication entre A et B et de récupérer K_{ab}. Ensuite, la clé K_{ab} ne peut pas non plus être considérée comme authentique par B : il est aussi facile pour I de couper la ligne de communication entre A et B et de la rediriger de I vers B ; toute clé K_I envoyée par I à B sera non authentique.

Supposons donc que A et B ont déjà en commun une clé K_{long}, dite à long terme, et partagée uniquement entre eux deux. A peut alors créer la clé K_{ab} comme précédemment, mais au lieu de l'envoyer en clair, il la chiffre d'abord avec la clé K_{long}, au moyen d'un algorithme de chiffrement sûr (quel qu'il soit), obtenant ainsi le message chiffré $\{K_{ab}\}_{K_{long}}$. A envoie ensuite $\{K_{ab}\}_{K_{long}}$ à B. Si l'algorithme de chiffrement est suffisamment sûr, la seule façon pour un intrus de récupérer K_{ab} à partir de $\{K_{ab}\}_{K_{long}}$ est de connaître la clé K_{long} pour déchiffrer $\{K_{ab}\}_{K_{long}}$. Si K_{long} est maintenue secrète par A et B, l'intrus ne pourra donc pas récupérer la valeur de K_{ab} par l'espionnage seul des lignes de communication. Si de plus le chiffrement inclut suffisamment de redondance dans le message chiffré pour que le déchiffrement de $\{K_{ab}\}_{K_{long}}$ à l'aide d'une clé autre que K_{long} échoue, l'intrus ne pourra pas non plus passer un message de la forme $\{K_I\}_{K_{long}}$ à B, espérant que B utilisera une clé K_I qu'il aurait concoctée spécialement pour que B l'utilise sans le savoir à la place de la clé légitime K_{ab}. En effet, l'intrus ne sait pas chiffrer avec K_{long}, ne disposant pas de cette dernière.

Si ce protocole — car c'en est un, malgré sa simplicité — semble donc offrir de meilleures garanties de sécurité, grâce à l'utilisation de moyens de chiffrement, il ne semble par contre pas très utile. Le lecteur attentif aura sans doute remarqué qu'en principe ce protocole ne résout rien : pour obtenir une clé K_{ab} entre A et B, il faut déjà avoir une clé K_{long} entre A et B. En réalité, K_{long} est typiquement une clé extrêmement sûre, par exemple une clé à 1024 bits à utiliser avec l'algorithme RSA [50]. Mais cet avantage a un prix : chiffrer et déchiffrer avec K_{long} peut prendre de l'ordre de 1000 fois plus longtemps qu'avec un algorithme comme DES avec des clés de longueur effective 56 bits [50]. On préfère donc utiliser pendant peu de temps des clés relativement courtes, et renouveler ces clés régulièrement en utilisant la clé plus sûre K_{long} par un protocole comme celui ci-dessus. C'est pourquoi K_{long} est appelée une clé à long terme et K_{ab} est une clé de session. La distribution de la clé K_{long} elle-même peut être effectuée à l'aide du protocole de Diffie-Hellman [19], par exemple, qui repose sur des considérations de théorie des nombres.

En fait, ce protocole souffre de problèmes plus graves. Même s'il semble être sûr grâce à l'utilisation de méthodes de chiffrement, il n'en est rien. Rappelons qu'une clé de session est bien moins sûre qu'une clé à long terme comme K_{long}. Il est donc prudent de supposer qu'une clé de session K_0 utilisée il y a suffisamment longtemps a pu rentrer en la possession d'un intrus, que ce soit parce que l'intrus a eu tout le temps de trouver la clé K_0 à partir des différents messages chiffrés $\{M\}_{K_0}$ échangés entre A et B, par exemple par exploration brute de l'espace de toutes les clés possibles ; ou bien plus simplement parce que A ou B a de façon plus ou moins involontaire diffusé K_0 en clair après utilisation. L'intrus, qui avait d'autre part récupéré le message $\{K_0\}_{K_{long}}$ par espionnage des lignes de communication, peut maintenant passer $\{K_0\}_{K_{long}}$ à B, sachant que B va prendre la vieille clé K_0 pour la clé attendue K_{ab}. Tout message M que B chiffrera avec cette clé sera envoyé sous la forme $\{M\}_{K_0}$ et non $\{M\}_{K_{ab}}$, ce qui permettra à l'intrus de les déchiffrer et de récupérer M. Une telle attaque est appelée attaque par rejeu, car pour tromper B l'intrus a rejoué un message, $\{K_0\}_{K_{long}}$, d'une ancienne session du même protocole entre A et B.

Pour empêcher les attaques par rejeu, on complique le protocole pour inclure des garanties de fraîcheur dans les messages, autrement dit des données additionnelles qui permettent aux principaux de vérifier que les messages reçus sont récents, et donc pas des versions rejouées d'anciens messages. Une solution est d'apparier les messages avec une estampille temporelle T : A lit l'heure T à partir d'une horloge locale, et envoie le couple K_{ab}, T chiffré avec K_{long}, soit $\{K_{ab}, T\}_{K_{long}}$ à B, et B vérifie que le champ T du couple déchiffré est une heure suffisamment récente. Les estampilles temporelles sont d'un maniement délicat, la notion de "suffisamment récent" étant relativement vague pour commencer ; d'autre part pour que le mécanisme fonctionne, il faut encore que les horloges des différents principaux soient synchronisées, et ce par un protocole lui-même sécurisé, pour éviter qu'un intrus n'attaque le système en désynchronisant les horloges dans le but d'effectuer de nouveau des attaques par rejeu.

Une solution plus simple est l'utilisation de nonces. B, qui maintenant se méfie des attaques par rejeu, va initier la session et tirant un nombre aléatoire N_b (le *nonce*), et va l'envoyer en clair à A. A va ensuite créer la clé K_{ab}, et envoyer le message $\{K_{ab}, N_b\}_{K_{long}}$ à B. À réception de ce message, B va vérifier que le message se déchiffre correctement avec K_{long}, puis qu'il obtient ainsi un couple dont la deuxième composante est exactement le nonce N_b. Si tout s'est bien passé, B conclut que K_{ab} est une bonne clé récente pour communiquer avec A. En effet, avec très forte probabilité (et surtout si N_b est un nombre sur une grande largeur de bits, disons 128), le nombre N_b n'a jamais circulé que ce soit en clair ou chiffré sur les lignes de communication : N_b est une valeur fraîche. Si l'intrus avait pu construire $\{K_{ab}, N_b\}_{K_{long}}$, il n'avait donc pas pu le construire avant que N_b ne soit inventé par B, et la vérification par B que N_b est bien la deuxième composante du couple chiffré, sous l'hypothèse que la clé à long terme K_{long} est toujours inconnue de l'intrus, permet donc d'assurer que la paire K_{ab}, N_b, et donc la clé K_{ab} est bien récente et ne vient pas d'un message rejoué. On note en abrégé ce protocole :

1.	B $\longrightarrow$ A : N_b
2.	A $\longrightarrow$ B : $\{K_{ab}, N_b\}_{K_{long}}$

Pour autant, ce protocole est-il sûr ? Rien ne permet de l'affirmer. En fait, l'intrus pourrait jouer le rôle de A tout du long, et créer lui-même la clé K_{ab}. Les protocoles utilisés en réalité sont plus complexes, et utilisent en général un troisième principal de confiance. Ici, la solution classique est de faire intervenir un serveur de clés S, dont le rôle est de fabriquer une clé K_{ab} à la demande pour A et B, et de la leur communiquer dans une série de messages chiffrés avec des clés à long terme K_{as} et K_{bs} respectivement partagées entre A et S d'une part, entre B et S d'autre part.

2.2. Un protocole cryptographique réaliste

Comme exemple de cette famille plus réaliste de protocoles impliquant un tiers de confiance S, considérons le protocole suivant, dû à Needham et Schroeder [42]. Le but est que seuls A et B (et S, en toute rigueur) connaissent K_{ab} à la fin du protocole. Voici ce que font A, B et S dans ce protocole (voir aussi la figure 1).

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

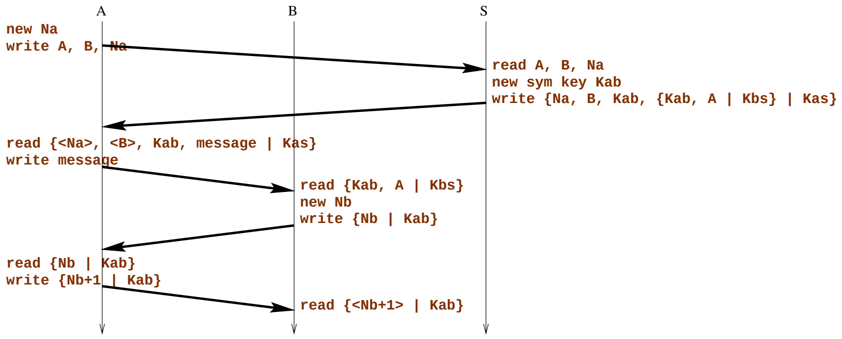

FIG. 1 – Le protocole de Needham-Schroeder à clés secrètes

En clair, voici comment se déroule une session du protocole. D'abord (message 1 ci-dessus, commandes `new Na` et `write A, B, Na` en haut à gauche de la figure 1, qui sont écrites dans le langage de l'outil CPV [26]), A envoie à S un message composé de sa propre identité, que l'on note A, de l'identité B de B, et d'un nonce N_a. (La création de ce dernier est effectuée par l'instruction `new Na` de la figure 1.) Les identités sont des identificateurs des principaux, par exemple des numéros de téléphone, des numéros de compte ou de carte bancaire, ou des URL. Ici, les identités de A et de B serviront au serveur S à retrouver les clés à long terme K_{as} et K_{bs} à l'étape 2. S les retrouve typiquement dans une base de données, à partir des identités de A et de B. Il est tentant de croire que les identités des principaux pourraient servir à authentifier les messages, mais ce n'est pas suffisant en général : il est facile de retrouver l'identité de n'importe qui, et n'importe quel intrus peut alors l'utiliser pour se faire passer pour n'importe qui.

Lorsque le message de l'étape 1 arrive à son destinataire S (`read A, B, Na` en figure 1), S renvoie le message $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$ à A. C'est un message un peu plus compliqué, qui a pour premier but d'envoyer la clé K_{ab} nouvellement créée (`new sym key Kab` en figure 1) à A et à B sous pli confidentiel. Pour ceci, S fabrique la paire consistant en K_{ab} et A, qu'il chiffre ensuite avec la clé K_{bs}, obtenant le message qu'on a noté $\{K_{ab}, A\}_{K_{bs}}$. Bien que ce message fasse partie d'un message que S envoie à A, A ne pourra pas accéder à K_{ab} en le déchiffrant, car seuls B et S connaissent la clé K_{bs} par hypothèse ; en fait, le message $\{K_{ab}, A\}_{K_{bs}}$ ne sert qu'à être récupéré par A pour le renvoyer à B à l'étape 3. C'est à l'étape 3 que B déchiffrera ce message et en retirera la valeur de K_{ab}. En attendant, S apparie ce message $\{K_{ab}, A\}_{K_{bs}}$ avec le nonce N_a envoyé par A, l'identité de B, et de nouveau la clé K_{ab}. (Le chiffrement d'un message M par la clé K, usuellement noté $\{M\}_K$, est `{M | K}` dans la notation de l'outil CPV [26].)

Revenons à l'étape 2 du protocole. A reçoit maintenant le message $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$, il utilise sa copie de la clé K_{as} pour le déchiffrer et ainsi récupérer le nonce N_a, la valeur de l'identité de B, la clé K_{ab} nouvellement créée par S, ainsi qu'un message qui est censé être $\{K_{ab}, A\}_{K_{bs}}$ et que A enverra à B à l'étape suivante. La figure 1 est plus précise quant à ce que A fait à cette étape (`read {<Na>, <B>, Kab, message | Kas}`) : A vérifie que la première composante du message déchiffré est exactement (la notation `<.>`) le nonce N_a que A avait créé en étape 1. Cette vérification assure A que le message $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$ est frais, et donc que la clé K_{ab} est elle-même fraîche.

A envoie ensuite à l'étape 3 le sous-message (`message` dans la figure 1) qu'il n'avait pas pu déchiffrer, et qui est censé être $\{K_{ab}, A\}_{K_{bs}}$. B le reçoit, obtient l'identité de A et la clé K_{ab}. Il s'engage ensuite dans deux étapes de protocole supplémentaires, 4. et 5., pour s'assurer qu'il possède la même clé K_{ab} que A. Pour ceci, B crée un nouveau nonce N_b, qu'il envoie chiffré par la clé K_{ab} à A. Si A est d'accord sur la valeur de K_{ab}, il pourra déchiffrer $\{N_b\}_{K_{ab}}$, fabriquer $N_b + 1$, le rechiffrer avec K_{ab} et le renvoyer à B. Noter qu'on pourrait modifier le protocole de sorte que A renvoie n'importe quel autre message fabriqué à partir de N_b et différent de N_b. Il ne peut pas renvoyer $\{N_b\}_{K_{ab}}$, sinon l'intrus pourrait intercepter le message 4 et le réinjecter directement à B.

Ce protocole est-il sûr ? La clé K_{ab}, que ce soit celle reçue par A ou celle reçue par B, est-elle véritablement connue uniquement de A, B, et S ? Les clés K_{ab} de A et B sont-elles les mêmes ? A peut-il avoir confiance en le fait que seul B (à part S) connaît K_{ab} ? Ce sont quelques-unes des questions que l'on peut se poser.

En fait, ce protocole n'est pas sûr. On peut montrer qu'il existe des attaques où les clés K_{ab} reçues par A et B sont différentes. Celle de A est toujours secrète (connue uniquement de A et S), mais celle de B peut provenir d'un rejeu. L'attaque typique est celle de la figure 2.

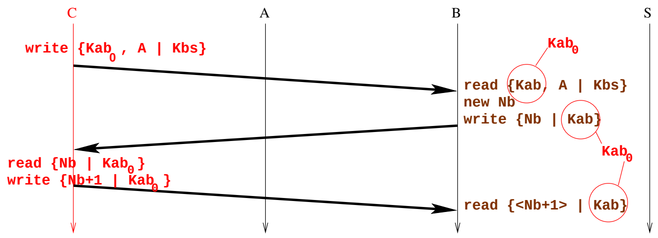

FIG. 2 – Une attaque contre le protocole de Needham-Schroeder à clés secrètes

Elle consiste pour l'intrus C à rejouer un message 3. d'une ancienne session, disons $\{K_{ab0}, A\}_{K_{bs}}$, suffisamment ancienne pour que C ait pu prendre connaissance de l'ancienne clé de session K_{ab0}. Il initie directement une session avec B à l'étape 3 (noter que B n'intervient pas avant cette étape), qui demande confirmation en créant le nonce N_b, et en envoyant le message $\{N_b\}_{K_{ab0}}$ à A. C déroute toutes les communications depuis B, et déchiffre lui-même ce dernier message, calcule $N_b + 1$ et le rechiffre avec K_{ab0}. Ainsi B dispose d'une vieille clé au lieu d'une fraîche. De plus cette clé est connue de l'intrus.

Le protocole à clés symétriques de Needham-Schroeder présenté ci-dessus est caractéristique des protocoles cryptographiques. Bien qu'il ait été publié, et ainsi soumis au regard critique de la communauté des chercheurs du domaine, il contient une faille de sécurité qui n'a été découverte que quelques années plus tard. Les failles sont parfois extrêmement subtiles. Dans le cas du protocole d'Otway-Rees [43], l'attaque [35] exige qu'un même principal joue deux rôles différents (ceux de A et de B) au cours d'une même session. Dans ce dernier cas, il a fallu six ans pour découvrir l'existence d'une attaque, mais il faut parfois plus longtemps. Un cas extrême est le protocole de Needham-Schroeder à clés publiques [42] (à ne pas confondre avec celui de la section 2.2), dont l'attaque a été découverte dix-sept ans plus tard [34].

En tout état de cause, ces quelques exemples ont permis à la communauté scientifique de prendre conscience de la nécessité de moyens systématiques de recherche d'attaques, ou a contrario de preuve formelle de sécurité des protocoles cryptographiques.

2.3. Si vous n'êtes pas encore convaincu...

L'attaque de la section précédente sur le protocole de Needham-Schroeder ne convainc pas toujours. Elle est en effet difficile à mener : l'intrus doit d'abord casser par force brute une clé $K_{ab\,0}$ d'une ancienne session, puis l'utiliser pour tromper B en se faisant passer pour un participant A honnête.

L'attaque sur le protocole de Needham-Schroeder *à clés publiques* est probablement plus convaincante de ce point de vue, en particulier parce qu'elle est systématique : elle réussit toujours, et il n'y a pas besoin de casser une clé par force brute auparavant. De plus, elle détruit le mythe communément répandu que la cryptographie à clés publiques fournirait une solution miracle à tous les problèmes de la cryptographie. La figure 3 présente un diagramme de la façon dont devrait se dérouler le protocole de façon normale.

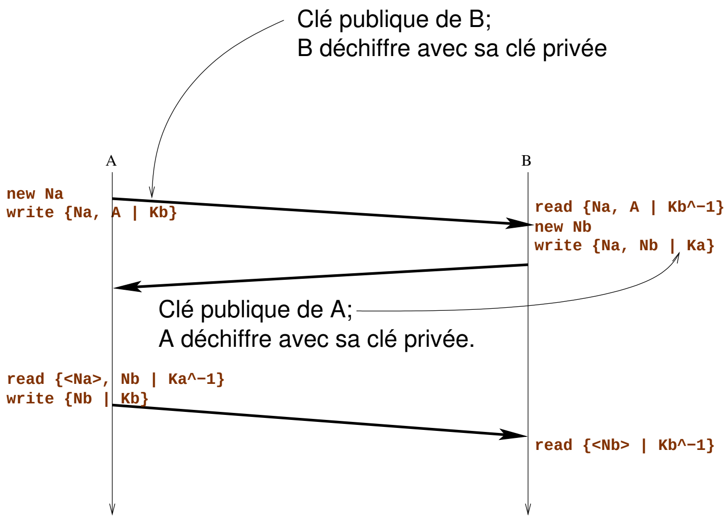

FIG. 3 – Le protocole de Needham-Schroeder à clés publiques

Le but de ce protocole est de permettre à A de transmettre un secret Na à B, et à B de transmettre un secret Nb à A, en s'assurant que la valeur de Na reçue par B provient authentiquement de A, et que la valeur de Nb reçue par A provient authentiquement de B. On note ici Kb^-1 la clé inverse de Kb.

La figure 4 présente l'attaque découverte par Lowe [34] dix-sept ans après l'invention du protocole.

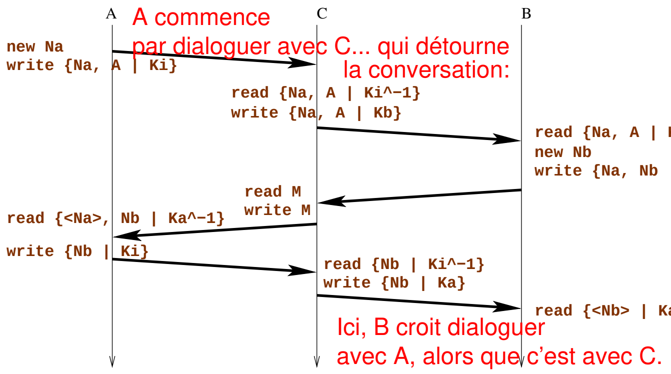

FIG. 4 – L'attaque de l'intermédiaire sur Needham-Schroeder à clés publiques

On notera que l'intrus C joue ici à la fois le rôle d'un participant (B) honnête avec A, et est par contre malhonnête dans une session jouée en parallèle avec B, auprès de qui il se fait passer pour A. En fait, C se sert des informations obtenues dans une session honnête avec A pour se faire passer pour A auprès de B.

Le fait que l'attaque nécessite que l'intrus joue *en même temps* dans deux sessions en parallèle, et qu'il est de plus à la fois honnête dans une session et malhonnête dans l'autre, explique probablement qu'on n'y ait pas pensé pendant autant de temps.

Pour sa simplicité, nous continuerons cependant à nous référer à l'exemple de protocole de Needham-Schroeder à clés secrètes de la section 2.1.

3. Le modèle de Dolev-Yao et ses applications

Nous allons ici expliquer le modèle de Dolev-Yao, et effectuer une première tentative de traduction des protocoles cryptographiques en formules de logique du premier ordre. Cette traduction a quelques défauts, que nous corrigerons en section 4.

3.1. Les principes fondamentaux

Un modèle simple servant de fondement à l'étude des protocoles cryptographiques est dû à Dolev et Yao [20]. Il connaît de nombreuses variantes et a servi de support à de nombreuses méthodes de vérification [37, 9, 10, 11, 45, 44, 17, 3, 31, 25, 40, 26, 22].

En premier lieu, il consiste à simplifier le modèle de communication entre principaux, en prenant en compte la quasi-omnipotence des intrus. Nul n'est besoin de préciser le modèle de communication (files, réseaux avec ou sans panne, avec ou sans duplication ou perte ou réordonnancement de messages, etc.) entre principaux : l'intrus pouvant espionner et dérouter toutes les lignes de communication, toute communication d'un message M entre deux principaux peut être simulée par un envoi de M à l'intrus, suivi d'un envoi de M par l'intrus au destinataire souhaité. Comme l'intrus peut simuler n'importe quel envoi et réception de message, on simplifie le modèle de communication en supposant que tout message envoyé est envoyé à l'intrus, et tout message reçu l'est de l'intrus.

En second lieu, le modèle de Dolev et Yao suppose que les messages envoyés et reçus ne sont pas ni des nombres ni des suites de bits, mais des éléments d'une algèbre de termes, éventuellement modulo une théorie équationnelle. Les messages sont engendrés par une grammaire de la forme :

$$M ::= D|K|pair(M, M)|p_1(M)|p_2(M)|encrypt(M, K)|decrypt(M, K)$$

où D parcourt un ensemble de données de base (entiers, réels, etc.), K parcourt un ensemble de clés, $pair(M_1, M_2)$ est le couple formé de M_1 et de M_2, $p_1(M)$ récupère la première composante du couple M, $p_2(M)$ récupère la deuxième composante, $encrypt(M, K)$ chiffre le message M avec la clé K, tandis que $decrypt(M, K)$ déchiffre le message chiffré M à l'aide de la clé K. Les messages sont à comprendre modulo la théorie équationnelle :

$$\begin{aligned} p_1(pair(M_1, M_2)) &= M_1 \\ p_2(pair(M_1, M_2)) &= M_2 \\ decrypt(encrypt(M, K), K) &= M \end{aligned}$$

En particulier, toute tentative de déchiffrer $\{M\}_K$ (codé ici par $encrypt(M, K)$) en utilisant une clé K' différente de K fournit un terme $decrypt(encrypt(M, K), K')$ qui est prouvablement différent de M dans la théorie équationnelle ci-dessus — plus précisément, qui est différent de M dans le modèle initial de cette théorie équationnelle.

Troisièmement et finalement, le modèle de Dolev et Yao représente l'intrus comme un système déductif, qui représente tous les messages qu'un intrus peut fabriquer par un ensemble de règles, qui définissent un prédicat $E \models\!\!\!\rightarrow M$ ("à partir de l'ensemble de messages E, l'intrus peut fabriquer le message M") :

1. $E, M \models\!\!\!\rightarrow M$: de tout ensemble E de messages augmenté d'un message M, l'intrus peut inférer M (rejeu simple) ;
2. Si $E \models\!\!\!\rightarrow M_1$ et $E \models\!\!\!\rightarrow M_2$, alors $E \models\!\!\!\rightarrow pair(M_1, M_2)$: l'intrus peut fabriquer tous les couples de messages formés de messages qu'il peut fabriquer ;
3. Si $E \models\!\!\!\rightarrow M$ alors $E \models\!\!\!\rightarrow p_1(M)$ et $E \models\!\!\!\rightarrow p_2(M)$: l'intrus peut extraire les composantes des couples ;

4. Si $E \mapsto M$ et $E \mapsto K$ alors $E \mapsto encrypt(M, K)$: l'intrus peut effectuer tous les chiffrements qu'il souhaite ;
5. Si $E \mapsto M$ et $E \mapsto K$ alors $E \mapsto decrypt(M, K)$: l'intrus peut tenter de déchiffrer tout message avec toute clé qu'il sait fabriquer ;
6. Si $E \mapsto M$ et $M = N$ dans la théorie équationnelle ci-dessus, alors $E \mapsto N$.

L'intrus peut produire des messages en utilisant ces règles en aussi grand nombre et aussi souvent que nécessaire. Par contre, l'intrus ne peut pas produire de message d'aucune autre façon. Notamment, l'intrus ne peut pas produire M à partir de $encrypt(M, K)$, à moins de ne pouvoir produire la clé K ou de ne pouvoir produire M sans partir de $encrypt(M, K)$.

3.2. Modélisation en logique du premier ordre

Sous ces hypothèses, il est maintenant raisonnablement facile de représenter une ou plusieurs sessions d'un protocole cryptographique, ainsi que les hypothèses initiales (dans l'exemple de la section 2.2, le fait que K_{as} et K_{bs} sont initialement secrètes, c'est-à-dire impossibles à produire par l'intrus) et les propriétés à prouver. À titre d'exemple, montrons comment le protocole de Needham-Schroeder à clés symétriques de la section 2.2 peut être décrit à l'aide d'une formule. Nous utilisons une notation à la Prolog pour bien montrer l'analogie avec l'écriture d'un programme : la notation `A :- B1, ., Bn` est la clause "`B1` et ... et `Bn` impliquent `A`". Nous conviendrons aussi que les identificateurs commençant par une minuscule sont des constantes, les variables commençant par des majuscules, comme en Prolog.

En premier, on va coder les comportements des principaux A, B, et S. Pour ceci, on représente chaque principal comme un automate, dont l'état est un n-uplet comportant un champ décrivant à quelle étape du protocole le principal se trouve (un compteur de programme), et autant de champs que nécessaires pour donner des valeurs aux variables locales. Par exemple, A maintiendra un champ pour son identité A, un pour celle de B, un pour la valeur du nonce N_a, un pour la clé K_{as}, un pour la clé K_{ab}, un pour le message `message` (que nous noterons `M`), et un pour le nonce N_b. Le prédicat

```
Pa ((S, A, B, Na, Kas, Kab, Nb), E, (S', A', B', N'a, K'as, K'ab, N'b), E')
```

exprimera que A peut passer de l'état

```
(S, A, B, Na, Kas, Kab, Nb)
```

à l'état

```
(S', A', B', N'a, K'as, K'ab, N'b)
```

alors que l'intrus disposait de tous les messages dans l'ensemble `E` et dispose après cette action de tous les messages dans l'ensemble `E'`. On écrira alors le rôle de A (la suite des actions qu'il exécute, cf. la colonne de gauche de la figure 1) par exemple comme suit.

Voici d'abord la clause décrivant la transition effectuée par A qui émet le message 1, passant de l'état `avant-1` (un symbole dénotant l'état où se situe A juste avant d'envoyer le message 1) à l'état `entre-1-et-2` (un autre symbole dénotant l'état où se trouve A une fois qu'il a envoyé le message 1, attendant de recevoir le message 2) :

```
Pa((avant-1, A, B, _, Kas, Kab, M, Nb), E,
    (entre-1-et-2, A, B, Na, Kas, Kab, M, Nb), E') :-
    Na = nonce-A-1,
    E' = add (pair (A, pair (B, pair (Na, nil))), E).
```

L'instruction `new Na` est ici approximée par la contrainte que la variable `Na` doive être égale à la constante `nonce-A-1`, qui représente toutes les valeurs possibles du nonce `Na`, au cours de n'importe quelle exécution du protocole par A. Cette approximation est correcte au sens où, si nous arrivons à prouver que le protocole ainsi codé en Prolog est sûr, alors il le sera effectivement. Nous verrons en section 4.2 une façon plus précise de modéliser les nonces.

L'envoi du message $[A, B, N_a]$ de A à S est tout simplement représenté par l'ajout de ce message (le terme du premier ordre `pair (A, pair (B, pair (Na, nil))))` à l'ensemble des connaissances `E` de l'intrus. Ceci fournit un nouvel ensemble de connaissances `E'` Nous laissons pour l'instant le terme `add` non spécifié. En toute rigueur, il doit obéïr à de nouvelles équations exprimant que `E` est un ensemble de messages, que nous ne précisons pas ici.

Définissons un prédicat binaire `knows` codant la relation $\not\mapsto$. Il est facile de le faire en Prolog (exercice : retranscrire les clauses 1.–6. de la section 3.1 en Prolog).

La deuxième transition de A consiste à recevoir le message $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$, puis à passer à l'état `entre-2-et-3`. Techniquement, A doit d'abord recevoir un message `M1` : dans le modèle de Dolev-Yao, un tel message est n'importe quel message que l'intrus peut fabriquer, autrement dit n'importe quel message tel que `knows(E,M1)` est déductible. A doit ensuite déchiffrer (en utilisant `decrypt`) le message `M1`, puis vérifier que le texte déchiffré est une liste $[N'_a, B', K_{ab}, M]$, où N'_a est exactement le nonce N_a créé à la précédente étape par A, et l'identité B' est exactement l'identité du correspondant souhaité B. On ne peut pas demander à A ni de vérifier le contenu de la clé K_{ab}, ni de déchiffrer le message M. Finalement, lors d'une telle transition on peut supposer que l'intrus n'apprend rien, et que donc son état de connaissances `E'` après la transition est le même que celui, `E`, avant la transition.

```
Pa((entre-1-et-2, A, B, Na, Kas, _, _, Nb), E,
   (entre-2-et-3, A, B, Na, Kas, Kab, M, Nb), E') :-
   knows(E, M1),
   decrypt(M1, Kas) = pair (N'a, pair (B', pair (Kab, pair (M, nil)))),
   N'a = Na, B' = B,
   E' = E.
```

Le reste de la modélisation est maintenant clair. Voici la troisième transition de A, qui consiste juste à envoyer le message M, censé être $\{K_{ab}, A\}_{K_{bs}}$, à B — donc directement à l'intrus :

```
Pa ((entre-2-et-3, A, B, Na, Kas, Kab, M, Nb), E,
    (entre-3-et-4, A, B, Na, Kas, Kab, M, Nb), E') :-
   E' = add (M, E).
```

Puis A reçoit le message $\{N_b\}_{K_{ab}}$:

```
Pa ((entre-3-et-4, A, B, Na, Kas, Kab, M, _), E,
    (entre-4-et-5, A, B, Na, Kas, Kab, M, Nb), E') :-
    knows(E, M2),
    decrypt(M2, Kas) = Nb,
    E' = E.
```

Enfin A envoie $\{N_b + 1\}_{K_{ab}}$:

```
Pa ((entre-4-et-5, A, B, Na, Kas, Kab, M, _), E,
    (après-5, A,  B, Na, Kas, Kab, M, Nb), E') :-
    E' = add (encrypt (Nb+1, Kab), E).
```

On écrit de même une série de clauses définissant le prédicat `Pb` caractérisant les actions possibles de B, dont l'état instantané est de la forme

```
(S, Kab, A, Nb, Kbs)
```

et le prédicat `Ps` caractérisant les actions possibles de S, dont l'état instantané est de la forme

```
(S, A, B, Na, Kab, Kas, Kbs)
```

On peut maintenant décrire la composition parallèle des trois principaux A, B, et S. Une configuration globale est un quadruplet `(Ea, Eb, Es, E)` comportant un état `Ea` de A (un septuplet `(S, A, B, Na, Kas, Kab, Nb)`), un état `Eb` de B (un quintuplet `(S, Kab, A, Nb, Kbs)`), un état `Es` de S (un septuplet `(S, A, B, Na, Kab, Kas, Kbs)`), et un ensemble de messages `E` récupérés par l'intrus. Définissons un prédicat `P` décrivant les actions possibles de la composition parallèle de A, B, et S :

```
P ((Ea, Eb, Es, E), (E'a, E'b, E's, E'))
```

décrit toutes les transitions possibles de l'état global `(Ea, Eb, Es, E)` vers l'état global `(E'a, E'b, E's, E')`, par entrelacement entre les actions de A, de B, et de S :

```
P((Ea, Eb, Es, E), (E'a, E'b, E's, E')) :-
   Pa  (Ea,  E,  E'a,  E'), Eb=E'b, Es=E's.

P((Ea, Eb, Es, E), (E'a, E'b, E's, E')) :-
   Ea=E'a, Pb(Eb,  E,  E'b, E'), Es=E's.

P((Ea, Eb, Es, E), (E'a, E'b, E's, E')) :-
   Ea=E'a,  Eb=E'b,  Ps(Es, E, E's, E').
```

Une exécution est une suite éventuellement vide de transitions, et `exec` peut être définie comme la clôture réflexive transitive de `P`, de sorte que

```
exec((Ea, Eb, Es, E), (E'a, E'b, E's, E'))
```

soit vrai si le système global peut évoluer de `(Ea, Eb, Es, E)` vers l'état `(E'a, E'b, E's, E')` en un nombre quelconque d'étapes :

```
exec((Ea, Eb, Es, E), (Ea, Eb, Es, E)).           % pas d'action.

exec((Ea, Eb, Es, E), (E''a, E''b, E''s, E'')) :- % une  transition,
                                                  % suivie d'une exécution:
     P((Ea, Eb, Es, E), (E'a, E'b, E's,  E')),
     exec((E'a,  E'b, E's, E'), (E''a, E''b, E''s, E'')).
```

Notons $Exec$ la conjonction de toutes ces clauses, préalablement universellement quantifiées. On a ainsi entièrement spécifié toutes les exécutions possibles du protocole cryptographique, incluant non seulement les exécutions normales, comme celle de la figure 1, mais aussi toutes les exécutions pathologiques, comme celle de la figure 2.

Notons $Intrus$ la modélisation du prédicat `knows` ($\models$) telle que décrite dans les points 1.-6. à la section 3.1, retranscrite en logique du premier ordre.

Vérifier que la valeur de K_{ab} que A possède à la fin du protocole est secrète, par exemple, en supposant que K_{as} et K_{bs} sont initialement secrètes revient à prouver la formule :

$$\left(\begin{array}{l} Exec \wedge Intrus \wedge \forall E_{init}, E_{a\ init}, E_{b\ init}, E_{s\ init} \cdot \\ \quad (\forall K_{as} \cdot E_{a\ init} = (\texttt{avant-1}, \_, \_, \_, K_{as}, \_, \_, \_) \Rightarrow E_{init} \not\models K_{as}) \\ \wedge (\forall K_{bs} \cdot E_{b\ init} = (\texttt{avant-1}, \_, \_, \_, K_{bs}) \Rightarrow E_{init} \not\models K_{bs}) \\ \wedge (\forall K_{as}, K_{bs}, K_{ab} \cdot E_{s\ init} = (\texttt{avant-1}, \_, \_, \_, K_{ab}, K_{as}, K_{bs}) \\ \qquad\qquad E_{init} \not\models K_{as} \wedge E_{init} \not\models K_{bs} \wedge E_{init} \not\models K_{ab}) \end{array}\right)$$

$$\begin{aligned}\Rightarrow\quad & \forall E_{fin}, E_{a\ fin}, E_{b\ fin}, E_{s\ fin}, K_{ab}\cdot\\ & \quad \texttt{exec}((E_{a\ init}, E_{b\ init}, E_{s\ init}, E_{init}), (E_{a\ fin}, E_{b\ fin}, E_{s\ fin}, E_{fin}))\\ & \wedge E_{a\ fin} = (\texttt{après-5}, \_, \_, \_, \_, K_{ab}, \_)\\ & \quad \Rightarrow E_{fin} \not\mapsto K_{ab}\end{aligned}$$

En clair, si les clés K_{as}, K_{bs}, K_{ab} ne sont pas connues dans l'état initial de connaissances E_{init} de l'intrus, alors pour tout état final où A est au compteur après-5, la version K_{ab} de la clé que A est censé partager avec B n'est pas connue de l'intrus dans son état final de connaissances E_{fin}.

Cette formule est prouvable en logique du premier ordre interprétée modulo la théorie équationnelle de la section 3.1 (à un détail près, lié à la nature inductive de la définition de $\mapsto$, que nous verrons en section 3.3). On peut pour s'en convaincre utiliser des assistants de preuve comme Coq [10, 11] ou Isabelle [45], ou des systèmes de démonstration automatique [31]. On peut ainsi vérifier des propriétés de sécurité sous diverses hypothèses initiales. Par exemple, dans la formule ci-dessus, on a pris des hypothèses relativement libérales ; notamment, on n'a pas demandé que les clés K_{as} ou K_{bs} aient les mêmes valeurs pour A et pour S, et pour B et pour S respectivement. En fait, A et B pourraient déjà avoir subi des attaques, avec comme conséquence que par exemple A et S ne sont pas d'accord sur la valeur de K_{as}. Ceci ne compromet pas le secret de K_{ab} à la fin de l'exécution de A, du moment que la clé K_{as} de A et la clé K_{as} de S sont toutes les deux secrètes initialement.

De même, on peut prouver que les différentes variantes des clés à long terme K_{as} et K_{bs} restent toujours secrètes à toute étape du protocole. Par contre, la formule similaire tendant à établir que le K_{ab} de B est secret à la fin de l'exécution de B est fausse et donc non prouvable : l'attaque de la figure 2 est un contre-exemple. La recherche de preuve permet de donner une idée du contre-exemple, néanmoins les systèmes de preuve automatique sont souvent incapables d'extraire un contre-exemple automatiquement. Des systèmes de model-checking comme FDR [34] sont plus directement orientés vers la détection d'attaque, mais ne fournissent aucune garantie de sécurité au cas où aucune attaque n'est trouvée, car ils travaillent sur des abstractions à états finis des protocoles cryptographiques, qui sont des systèmes à états infinis ; certains travaux montrent cependant que sous certaines hypothèses, des abstractions à suffisamment d'états finis suffisent pour obtenir des garanties de sécurité [53].

3.3. Commentaires et extensions

Prédicat $\mapsto$ et théories inductives. Le prédicat $E \mapsto M$ est défini comme étant défini inductivement par les clauses 1.-6. de la section 3.1. Autrement dit, l'ensemble des couples (E, M) tels que $E \mapsto M$ est le *plus petit* tel que les clauses 1.-6. sont vérifiées. L'aspect inductif permet en particulier de montrer des assertions de la forme $E \not\mapsto M$. Or la formalisation en logique du premier ordre ne permet pas d'inclure cet aspect inductif. Il y a plusieurs solutions à ce problème : utiliser un assistant de preuve incluant des principes inductifs, c'est-à-dire de récurrence [10, 11, 45], ou bien coder sous forme de formules de logiques du premier ordre quelques principes suffisants pour démontrer des formules de la forme $E \not\mapsto M$ [11]. Il se trouve qu'en fait un faible nombre de telles formules est nécessaire : El Kadhi [22] dégage une notion de théorème de transfert, de la forme "sous certaines hypothèses, si $E \not\mapsto M$ alors $E, M' \not\mapsto M$", qui permet de conclure qu'une donnée M secrète le reste même après l'écriture du message M', et montre qu'avec essentiellement trois théorèmes de transfert, on peut établir que certaines données secrètes le restent au cours d'un protocole. ([22] considère en fait le problème plus difficile de valider automatiquement un programme, par analyse statique, en ce qui concerne la non-divulgation de données sensibles en présence de primitives cryptographiques.)

Enrichissement de la théorie équationnelle. La théorie équationnelle des messages de la section 3.1 peut être enrichie pour prendre en compte certaines propriétés algébriques, notamment des algorithmes de chiffrement. Par exemple, le chiffrement à l'aide de l'algorithme RSA a de nombreuses propriétés, parmi lesquels $encrypt(M^n, K) = encrypt(M, K)^n$, qui mènent à des attaques. Pour établir de façon certaine la sécurité d'un protocole cryptographique, toutes les équations valides doivent être ajoutées à la théorie équationnelle. Un certain nombre de telles équations est fourni dans [1].

Appauvrissement de la théorie équationnelle. A contrario, les réalisations informatiques professionnelles de protocoles cryptographiques incluent de nombreuses précautions de codage (chiffrement en mode CBC plutôt que par blocs, inclusion d'information redondante de typage pour éviter les confusions de types, etc. [50]). Par exemple, au lieu de chiffrer M par K en utilisant RSA directement, on va d'abord encoder M par exemple en ASN.1 [32], puis appliquer RSA sur le message résultant avec la clé K. Ces précautions ont essentiellement pour effet d'invalider toutes les équations non triviales (autres que de la forme $x = x$) qui tenaient sur l'algorithme de chiffrement brut. On peut pousser l'élimination de la théorie équationnelle plus loin, dans ces conditions : à cause des informations de types incluses, on peut reconnaître si un message est un couple ou un chiffrement, avec faible probabilité d'erreur. Ceci permet de se passer intégralement des opérateurs p_1, p_2, et $decrypt$. Par exemple, au lieu d'exprimer que $E \models M_1$ et $decrypt(M_1, K_{as}) = pair(N'_a, pair(B', pair(K_{ab}, pair(M, \mathtt{nil}))))$ dans la modélisation du protocole de la section 3.2, on écrira directement que $E \models encrypt(pair(N'_a, pair(B', pair(K_{ab}, pair(M, \mathtt{nil})))), K_{as})$. On peut alors entièrement se passer de l'égalité dans la modélisation (comme dans [11, 26]), ce qui simplifie grandement la recherche de preuve ; notamment la recherche automatique de preuve au premier ordre sans égalité est beaucoup plus simple qu'avec égalité.

L'approche par réécriture. Une autre modélisation des protocoles cryptographiques utilise, elle, uniquement la notion de réécriture [31, 25]. L'état global du système est représenté comme un terme modulo la théorie équationnelle de la section 3.2 et l'associativité et la commutativité de l'opérateur d'union ensembliste utilisé pour former l'ensemble E des messages connus de l'intrus. L'exécution du protocole est représentée par la réécriture de ce terme, directement. Pour montrer que K_{ab} reste secret dans toute exécution du protocole (par exemple), on montre qu'il n'est pas déductible d'aucun état global accessible depuis les états initiaux, soit directement [31], soit par calcul d'un sur-ensemble de cet ensemble des états accessibles par une méthode qui termine toujours (par l'utilisation d'automates d'arbres, par exemple [25]).

Authentification. Nous n'avons traité jusqu'ici que des propriétés de secret. Une autre propriété fondamentale est l'authentification. Un moyen simple de traiter de l'authentification est de conserver dans la modélisation une information sur l'identité du créateur réel de toute donnée. On ajoute pour cela, par exemple, une variable de relation R entre messages et identités de participants parmi les composantes de l'état global : (M, X) est dans la relation R si et seulement si le participant X est capable de produire le message M. Formellement, on peut utiliser un prédicat ternaire $belongs(M, X, R)$ ("(M, X) est dans la relation R") et une constante $\mathtt{i}$ dénotant l'identité de l'intrus. Les deux propriétés à maintenir durant l'exécution du protocole sont, d'une part, que si $E \models M$ alors $belongs(M, \mathtt{i}, R)$, et d'autre part que lorsqu'un participant X construit un message M, alors $belongs(M', X, R)$ pour tout sous-message non variable M' de M. Le fait qu'un message M reçu par B, par exemple sous la forme $\{M\}_{K_{ab}}$, vienne authentiquement de A reviendra à vérifier que tout X tel que $belongs(M, X, R)$ est en fait égal à A. Ceci interdit en particulier que $belongs(M, \mathtt{i}, R)$, autrement dit que l'intrus ait pu fabriquer le message M, mais aussi que $belongs(M, C, R)$ par exemple, avec $C \neq A$, ce qui interdit que l'intrus ait pu rejouer un message $\{M\}_{K_{ab}}$ venant de C même sans avoir accès à M. Une telle approche de l'authentification est à l'œuvre, quoique sous une forme en apparence différente, dans [14, 25, 31]. Cette notion d'authentification inclut la notion d'intégrité, qui exprime que le message n'a pas été modifié depuis sa création. Différents auteurs utilisent des notions différentes d'authenticité, voir par exemple la notion d'"agreement" [54] qui exprime que si M est authentiquement de A, alors lorsque B termine sa session, A a lui-même auparavant commencé la partie correspondante de la session.

Multi-sessions. Il est important de comprendre dans combien de sessions un même principal peut participer. Dans le protocole de la section 3.2, par exemple, A et B vont typiquement participer à une session, puis éventuellement à une autre, et ainsi de suite, en séquence ; ce qu'on appelle le mode *multi-sessions séquentiel*. Notre modélisation considérait que A s'arrêtait après une session, ce qui est trop restrictif. Le serveur de clés S, lui, peut participer à de nombreuses sessions en parallèle (le mode *multi-sessions parallèle*). Plus subtilement, on peut envisager des modes de fonctionnement où un même principal endosse plusieurs rôles (celui de A, celui de B par exemple) soit à la suite, soit même en parallèle. Ainsi l'attaque de Paulson [46] sur le protocole d'Otway-Rees demande à ce qu'un même principal joue en même temps le rôle de A dans une session et le rôle de B dans une autre. L'attaque de Lowe [34] sur Needham-Schroeder à clés publiques nécessite que l'intrus joue à la fois le rôle du participant B honnête dans une session avec A, et le rôle d'un participant malhonnête A dans une autre session parallèle avec B.

Il existe des réponses partielles à ce genre de situations. Notamment, l'algorithme de [17] considère tous les principaux comme des complices de l'intrus : si A lit $\{N_b\}_{K_{ab}}$ à l'étape 4. du protocole de la section 3.2, et renvoie $\{N_b + 1\}_{K_{ab}}$ à l'étape 5., par exemple, alors on peut considérer que A agit comme un complice de l'intrus, qui permet à ce dernier de déduire $\{N_b+1\}_{K_{ab}}$ à partir de $\{N_b\}_{K_{ab}}$ même sans connaître K_{ab}. Ceci est utilisé dans [26] pour modéliser de façon efficace et raisonnablement précise le comportement des principaux en multi-sessions parallèle.

Algorithmes de chiffrement à clés publiques. Jusqu'ici nous n'avons traité que de méthodes de chiffrement et de déchiffrement à clés symétriques : pour déchiffrer un message chiffré avec une clé K, c'est encore K qu'on doit utiliser. Des algorithmes comme RSA utilisent une notion de clés asymétriques : on chiffre avec une clé K, et on déchiffre avec la clé inverse K^{-1}. La théorie équationnelle de la section 3.1 doit alors être modifiée, l'équation portant sur $encrypt$ et $decrypt$ devenant $decrypt(encrypt(M, K), K^{-1}) = M$. Il est aussi nécessaire d'ajouter l'équation $K^{-1^{-1}} = K$. Notons que même en présence de paires de clés inverses, il est possible de se passer d'équations, cf. [26].

Autres propriétés de sécurité et commerce électronique. Le commerce électronique pose des défis supplémentaires à relever [13]. D'abord, les propriétés de secret et d'authentification, si elles sont toujours fondamentales, ne sont plus les propriétés essentielles que l'on cherche à prouver. Des propriétés plus complexes sont à établir : non-duplication de messages (factures), non-révocation de messages (commandes), ainsi que nous l'avons déjà mentionné dans l'introduction. Ces propriétés plus complexes requièrent de travailler sur des abstractions du protocole, lesquelles peuvent être en grande partie calculées automatiquement [12, 6]. Un autre aspect complexe des protocoles électroniques est qu'ils incluent typiquement non pas deux principaux (plus un serveur S) de confiance, mais trois principaux (l'acheteur, le vendeur, la banque) tels que tout sous-ensemble de deux se méfie du troisième. Il est donc plus délicat de séparer le monde en deux zones, les principaux de confiance et l'intrus, tout principal étant considéré comme un intrus par les deux autres. Bolignano [13] montre comment moduler une modélisation dans le style de la section 3.2 pour inclure non seulement des hypothèses initiales de secret, mais aussi des hypothèses d'honnêteté des différents principaux du protocole. La méthode de [16] permet aussi de modéliser des secrets entre participants honnêtes qui ne soient pas connus d'autres participants honnêtes.

4. Logique du premier ordre, types, automates d'arbres et vérification approchée de protocoles cryptographiques

Nous allons donc changer légèrement le modèle, pour corriger les quelques défauts mentionnés en section 3.3 de la modélisation de la section 3.2. Ceci sera fait en section 4.1. Il s'agit en particulier du modèle adopté dans le cadre du projet EVA [28]. Nous montrons comment les processus EVA se traduisent en ensembles de clauses de Horn (en programmes Prolog) en section 4.2, raffinant ainsi la traduction de la section 3.2. En principe, il suffirait de lancer un interprète Prolog pour vérifier les protocoles cryptographiques EVA, mais ceci ne fonctionne pas, nous verrons pourquoi et comment contourner le problème en section 4.3.

Les approximations proposées dans cette section sont cependant trop grossières, et nous montrons comment les raffiner, par des transformations de programmes Prolog préservant leur sémantique en section 4.5. Nous verrons enfin pourquoi on a quand même besoin de théories équationnelles en section 4.6, et les problèmes techniques que ceci entraîne.

4.1. Le modèle EVA

Tout d'abord, nous allons nous passer de théorie équationnelle — en tout cas jusqu'en section 4.6. Ensuite, nous allons introduire des constructions permettant de parler de chiffrement tant à clés symétriques qu'à clés asymétriques. De plus, nous allons aussi spécialiser les opérateurs de chiffrement en fonction d'un paramètre additionnel définissant l'algorithme de chiffrement utilisé, ce qui nous permettra d'être plus fins dans la modélisation de protocoles utilisant plusieurs méthodes de chiffrement, comme par exemple le *handshake* SSL ou TLS [18], où l'algorithme de chiffrement utilisé in fine est négocié entre les principaux dans une première phase.

La grammaire des *termes* EVA est la suivante :

Term	: :=	id	variable
	$\mid$	`crypt` (Term, Term, Term)	chiffrement (algo, texte, clé)
	$\mid$	`tuple` (Term)	n-uplet
	$\mid$	`nil` ()	liste vide
	$\mid$	`cons` (Term, Term)	liste non vide
	$\mid$	`d` (Term)	conversion `*D*` $\rightarrow$ message
	$\mid$	`p` (Term)	conversion `principal` $\rightarrow$ message
	$\mid$	`a` (Term)	conversion `*algo*` $\rightarrow$ message
	$\mid$	`sa` (Term)	conversion `sym_algo` $\rightarrow$ `*algo*`
	$\mid$	`aa` (Term)	conversion `asym_algo` $\rightarrow$ `*algo*`
	$\mid$	`vanilla` ()	algorithme symétrique par défaut
	$\mid$	`apply` (id, Term*)	application de fonction définie
	$\mid$	`hash-apply` (id, Term*)	application de fonction one-way
	$\mid$	`secret-apply` (id, Term*)	application de fonction secrète
	$\mid$	`hash-secret-apply` (id, Term*)	application de fonction secrète one-way
	$\mid$	`apply-pubk` (Term, id, Term*)	appl. de constructeur de clé publique (algo, constructeur, arguments)
	$\mid$	`hash-apply-pubk` (Term, id, Term*)	appl. de constructeur de clé publique one-way (algo, constructeur, arguments)
	$\mid$	`apply-privk` (Term, id, Term*)	appl. de constructeur de clé privée (algo, constructeur, arguments)
	$\mid$	`hash-apply-privk` (Term, id, Term*)	appl. de constructeur de clé privée one-way (algo, constructeur, arguments)

On notera que les chiffrements $\texttt{crypt}(a, M, K)$, qui représentent $\{M\}_K$, ont maintenant un argument supplémentaire a, qui désigne le nom d'un *algorithme* de chiffrement. Intuitivement, a est un nom d'algorithme connu comme "DES", "RSA", "IDEA". Les contraintes de typage (que nous ne reproduirons pas ici [28]) font qu'un algorithme est toujours de la forme $\texttt{a}(\texttt{sa}(\ldots))$ — dénotant un algorithme à clés symétriques — ou de la forme $\texttt{a}(\texttt{aa}(\ldots))$ — un algorithme à clés publiques. Une originalité du modèle est que la décision de déchiffrer $\{M\}_K$ avec la clé K elle-même, ou bien avec un certain inverse de K, dépend ici non pas de la forme de K mais de l'algorithme utilisé pour déchiffrer (voir aussi [22]).

Les règles de typage font aussi que les seuls termes qui sont des *messages* sont d'une des formes :
- une variable x ;
- ou $\texttt{crypt}(a, M, K)$ où a est un nom d'algorithme et M et K des messages ; noter que, contrairement à d'autres approches, il n'y a pas de type spécifique pour les clés, en particulier les clés K peuvent être elles-mêmes des messages chiffrés ;

– ou $\texttt{tuple}(\texttt{cons}(M_1, \texttt{cons}(M_2, \ldots, \texttt{cons}(M_n, \texttt{nil}()) \ldots)))$, qui représente un n-uplet de messages, que nous noterons $[M_1, M_2, \ldots, M_n]$ par souci de concision ;
– ou $\texttt{d}(d)$, qui représente un objet d d'un type de base `*D*`, typiquement des chaînes de bits ;
– ou $\texttt{p}(p)$, qui représente une identité p, de type `principal` ;
– ou $\texttt{a}(\texttt{sa}(a))$ (algorithme a de chiffrement symétrique, de type `sym_algo`) ou $\texttt{a}(\texttt{aa}(a))$ (algorithme a de chiffrement asymétrique, de type `asym_algo`) ;
– ou $[\texttt{hash-}][\texttt{secret-}]\texttt{apply}(f, M_1, \ldots, M_n)$, où les crochets dénotent des parties optionnelles : il s'agit de l'application d'une fonction de nom f aux arguments M_1, …, M_n. Si le préfixe `hash` est présent, l'intrus est supposé ne pas pouvoir retrouver les arguments M_1, …, M_n à partir du message composé (c'est une fonction de *hachage*). Si `secret` est présent, l'intrus est supposé ne pas pouvoir *construire* le message composé à partir des arguments M_1, …, M_n ;
– ou $[\texttt{hash-}]\texttt{apply-}\{\texttt{pubk}, \texttt{privk}\}(a, f, M_1, \ldots, M_n)$, qui permet de construire soit une clé publique (si `pubk`) soit une clé privée (si `privk`) ; de plus, si le préfixe `hash` est présent, l'intrus n'est pas supposé pouvoir retrouver les arguments M_1, …, M_n à partir du message composé. Les clés $\texttt{apply-pubk}(a, f, M_1, \ldots, M_n)$ et $\texttt{apply-privk}(a, f, M_1, \ldots, M_n)$ sont supposées être inverses l'une de l'autre — pourvu que les arguments a, f, M_1, …, M_n soient les mêmes. De même avec `hash-apply-pubk` et `hash-apply-privk`.

4.2. Modélisation en clauses de Horn

La première chose à faire est de définir le prédicat `knows` que nous utilisions en section 3.2 pour modéliser les capacités déductives de l'intrus.

4.2.1. Conditions initiales

Un changement que nous apportons à la modélisation de l'intrus est que `knows(E,M)` va maintenant signifier, non pas que `M` est déductible à partir de la liste de messages `E`, mais que `M` est déductible à partir de `E` et d'un ensemble de messages initialement connus de l'intrus E_0, qui sera spécifié via un autre prédicat `knows0`.

Si l'on n'écrit pas de clause avec `knows0` en tête, alors E_0 sera vide, et on sera ramené au cas précédent. Plus intéressant est le fait que l'on peut décrire ainsi des ensembles E_0 *infinis*. Dans CPV ([26], section 4.4 de la version longue), il est notamment montré que l'on peut ainsi modéliser l'ensemble E_0 maximal des messages que peut connaître l'intrus, lorsque l'on suppose qu'il ne connaît pas certains messages. Par exemple, si l'on suppose que :

$(*)$ initialement l'intrus ne connaît aucune clé privée, c'est-à-dire aucun message commençant par $[\texttt{hash-}]\texttt{secret-apply}$ ou par $[\texttt{hash-}]\texttt{apply-privk}$

alors on spécifiera E_0 par le prédicat `knows0` défini comme suit. On introduit un prédicat auxiliaire `not-knows0` qui définit le complément de `knows0`, ainsi qu'un prédicat `exists0` qui définit l'ensemble des messages qui *existent initialement*. Ici, un message qui *existe initialement* est n'importe quel message qui peut être construit avec tous les symboles de fonction sauf `d`. L'intention est que les nonces créés ultérieurement dans le protocole seront des termes t de type `*D*`, et convertis en messages $\texttt{d}(t)$. La notion d'existence initiale permet de parler des messages qui ne mentionnent aucun nonce créé ultérieurement. (Ceci suppose que les conditions initiales ne mentionnent aucun nonce. On ne perd en fait pas en généralité en faisant cette hypothèse : si l'on souhaite que certains nonces soient en fait connus de l'intrus initialement, il suffit d'ajouter au protocole quelques transitions initiales divulguant les nonce souhaités à l'intrus, comme dans la section suivante.)

La notion d'existence initiale est formalisée directement par le prédicat `exists0` :

```
exists0(crypt(A, M, K)) :- exists0(A), exists0(M), exists0(K).
exists0(tuple(L)) :- exists0(L).
```

```
exists0(nil()).
exists0(cons(M, L)) :- exists0(M), exists0(L).
exists0(p(_)).
exists0(a(_)).
exists0(apply(F, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
exists0(hash-apply(F, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
exists0(secret-apply(F, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
exists0(hash-secret-apply(F, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
exists0(apply-pubk(A, F, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
exists0(hash-apply-pubk(A, F, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
exists0(apply-privk(A, F, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
exists0(hash-apply-privk(A, F, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
```

Pour formaliser l'hypothèse $(*)$, on commence par écrire :

```
not-knows0(secret-apply(_, ...)).
not-knows0(hash-secret-apply(_, ...)).
not-knows0(apply-privk(_, _, ...)).
not-knows0(hash-apply-privk(_, _, ...)).
```

ce qui exprime que l'intrus n'est effectivement pas capable de déduire les messages supposés secrets initialement.

On écrit maintenant à quelles conditions un message *chiffré* est inconnu initialement. (Nous nous contenterons de traiter des algorithmes de chiffrement symétriques par souci de simplicité.) Le message $\{M\}_K$ ne peut pas être connu de l'intrus si et seulement si M est lui-même inconnu et K est connue. En effet, si K était inconnue, alors $\{M\}_K$ doit être dans E_0 : sinon, intuitivement, on pourrait rajouter $\{M\}_K$ à E_0 sans violer la condition $(*)$ ci-dessus, contredisant la maximalité de E_0. Si K est connue, en revanche, alors M doit être lui-même inconnu sinon l'intrus pourrait déduire $\{M\}_K$ à partir de M et K. On écrit donc :

```
not-knows0(crypt(a(sa(_)), M, K)) :- not-knows0(M), knows0(K).
```

Un n-uplet est inconnu si et seulement si au moins une de ses composantes est inconnue, ce qu'on peut écrire comme suit :

```
not-knows0(tuple(L)) :- not-knows-list0(L).
not-knows-list0(cons(M, _)) :- not-knows0(M).
not-knows-list0(cons(M, L)) :- knows0(M), not-knows-list0(L).
```

Sachant ce qui est inconnu de l'intrus, nous pouvons maintenant définir ce qui est connu de l'intrus initialement. D'abord, l'intrus est supposé connaître toutes les identités `p(...)`, tous les algorithmes de chiffrement `a(...)`, toutes les clés publiques [`hash-`]`apply-pubk(...)`, mais aucune donnée de base `d(...)`, ni aucun des secrets mentionnés dans l'hypothèse $(*)$:

```
knows0(p(_)).
knows0(a(_)).
knows0(apply-pubk(_, _, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
knows0(hash-apply-pubk(_, _, M1, ..., Mn) :- exists0(M1), ..., exists0(Mn).
```

Pour ce qui est des chiffrements, l'intrus connaît $\{M\}_K$, autrement dit $\{M\}_K$ doit être dans E_0, si et seulement si M est dans E_0 et K est un message qui existe initialement, ou bien M est un message qui existe initialement et K est inconnu. En effet, supposons le contraire : M est initialement inconnu, et K est initialement connu (noter que comme $\{M\}_K$ est connu, il existe initialement, donc M et K aussi). Alors le raisonnement que nous avons effectué plus haut pour le prédicat `not-knows0` appliqué aux chiffrements montre que nécessairement $\{M\}_K$ sera inconnu initialement, une contradiction. On écrit donc :

```
knows0(crypt(a(sa(_)), M, K)) :- knows0(M), exists0(K).
knows0(crypt(a(sa(_)), M, K)) :- exists0(M), not-knows0(K).
```

Il reste à préciser la définition de `knows0` sur les autres symboles de fonction. C'est simple :

```
knows0(tuple(L)) :- knows-list0(L).
knows-list0(nil()).
knows-list0(cons(M, L)) :- knows0(M), knows-list0(L).
knows0(p(_)).
knows0(a(_)).
knows0(apply(F, M1, ..., Mn)) :- knows0(M1), ..., knows(Mn).
knows0(hash-apply(_, M1, ..., Mn)) :- exists0(M1), ..., exists0(Mn).
```

Noter que `apply` se comporte exactement comme `tuple`, alors que `hash-apply` se comporte comme un chiffrement avec une clé inconnue de l'intrus.

4.2.2. Déductions de l'intrus

La notion de conditions initiales de la section précédente est probablement la plus compliquée de toute la formalisation. La formalisation de `knows` est beaucoup plus transparente. D'abord, un message `M` est déductible de E_0 union `E` s'il est dans E_0, ou dans `E` :

```
knows(E, M) :- knows0(M).
knows(E, M) :- in-list(E,M).
in-list(cons(M, _), M).
in-list(cons(_, L), M) :- in-list(L, M).
```

Ensuite, on écrit les règles pour le prédicat $\models\!\!\rightarrow$, adaptées au modèle décrit informellement en début de section 4.1 :

```
knows(E, crypt(_, M, K)) :- knows(E, M), knows(E, K).
knows(E, M) :- knows(E, crypt(a(sa(_)), M, K)), knows(E, K).
knows(E, M) :- knows(E, crypt(a(aa(A)), M,
                          apply-pubk(a(aa(A)), F, M1, ..., Mn))),
               knows(E, apply-privk(a(aa(A)), F, M1, ..., Mn)).
knows(E, M) :- knows(E, crypt(a(aa(A)), M,
                          hash-apply-pubk(a(aa(A)), F, M1, ..., Mn))),
               knows(E, hash-apply-privk(a(aa(A)), F, M1, ..., Mn)).
knows(E, M) :- knows(E, crypt(a(aa(A)), M,
                          apply-privk(a(aa(A)), F, M1, ..., Mn))),
               knows(E, apply-pubk(a(aa(A)), F, M1, ..., Mn)).
knows(E, M) :- knows(E, crypt(a(aa(A)), M,
                          hash-apply-privk(a(aa(A)), F, M1, ..., Mn))),
               knows(E, hash-apply-pubk(a(aa(A)), F, M1, ..., Mn)).
```

Les quatre dernières clauses énoncent que si l'intrus connaît $\{M\}_K$ et la clé inverse K^{-1} pour l'algorithme de chiffrement asymétrique `A`, alors l'intrus connaît M aussi. On notera que la notion de clés inverses est définie par le fait que l'inverse de [`hash-`]`apply-pubk`(...) est [`hash-`]`apply-privk`(...) et réciproquement. (Exercice : revenir à la section précédente et traiter des conditions initiales pour les chiffrements à clés asymétriques. Il est nécessaire d'introduire des prédicats énonçant quand un message doit avoir un inverse pour un algorithme `A` donné qui est connu initialement, resp. inconnu initialement.)

Pour les n-uplets, on exprime que $[M_1, \ldots, M_n]$ est connu si et seulement si chaque M_i est connu, $1 \leq i \leq n$:

```
knows(E, tuple(L)) :- knows-list(E, L).
knows(E, nil()).
knows(E, cons(M, L)) :- knows(E, M), knows-list(E, L).
knows(E, M) :- knows-list(E, cons(M, L)).
knows-list(E, L) :- knows-list(E, cons(M, L)).
knows-list(E, L) :- knows(E, tuple(L)).
```

On peut découper l'ensemble de clauses ci-dessus en deux sous-ensembles. Les trois premières clauses expriment que l'intrus peut *fabriquer* des n-uplets, les trois dernières qu'il peut *décomposer* des n-uplets. Les constructions [`hash-`][`secret-`]`apply`$(f, M_1, \ldots, M_n)$ diffèrent précisément en fonction du fait que l'intrus peut ou non fabriquer de tels messages (`secret`), et qu'il peut ou non décomposer de tels messages (`hash`). On écrit donc :

```
knows(E, apply(F, M1, ..., Mn)) :- knows(E, M1), ..., knows(E, Mn).
knows(E, M1) :- knows (E, apply (F, M1, ..., Mn)).
    ...
knows(E, Mn) :- knows (E, apply (F, M1, ..., Mn)).

knows(E, hash-apply(F, M1, ..., Mn)) :- knows(E, M1), ..., knows(E, Mn).

knows(E, M1) :- knows (E, secret-apply (F, M1, ..., Mn)).
    ...
knows(E, Mn) :- knows (E, secret-apply (F, M1, ..., Mn)).
```

et il n'y a donc aucune clause portant sur `hash-secret-apply`.

Les autres clauses sont alors faciles à écrire :

```
knows(E, p(_)).
knows(E, a(_)).
knows(E, apply-pubk(A, F, M1, ..., Mn)) :- knows(E, M1), ..., knows(E, Mn).
knows(E, M1) :- knows (E, apply-pubk (A, F, M1, ..., Mn)).
    ...
knows(E, Mn) :- knows (E, apply-pubk (A, F, M1, ..., Mn)).

knows(E, hash-apply-pubk(A, F, M1, ..., Mn)) :-
      knows(E, M1), ..., knows(E, Mn).

knows(E, M1) :- knows (E, apply-privk (A, F, M1, ..., Mn)).
    ...
knows(E, Mn) :- knows (E, apply-privk (A, F, M1, ..., Mn)).
```

4.2.3. Modélisation des protocoles

Nous n'allons pas redire comment modéliser un protocole cryptographique comme celui de la section 3.2. Cependant la structure des termes représentant les messages est suffisamment différente pour que nous expliquions ici comment la modélisation en termes de clauses de Horn opère dans le nouveau contexte. En particulier, nous allons éviter d'utiliser le prédicat Prolog d'égalité, ce qui simplifiera notre travail de vérification dans la suite. Nous allons aussi éviter d'utiliser des primitives Prolog (éventuellement non spécifiées) comme nous l'avons fait en section 3.2.

La modélisation des envois de message ne change fondamentalement pas. Cependant, nous allons éviter d'utiliser une fonction externe `add` comme dans le passage de `avant-1` à `entre-1-et-2` pour l'agent A (cf. section 3.2), et nous allons changer la spécification de la création du nonce N_a de façon plus précise. En plus des composants A, B, N_a, K_{as}, K_{ab}, M, N_b et d'une constante dénotant la ligne du protocole que nous utilisions dans la modélisation de l'état de l'agent A en section 3.2, nous allons inclure un entier `Ses` dénotant le numéro de session où l'agent joue le rôle de A. (Un entier est ici un terme de la forme $\mathtt{S}(\ldots\mathtt{S}(\mathtt{0}())\ldots)$, représentant un entier en unaire.) Le nonce N_a nouvellement créé sera alors une fonction de ce numéro de session. En fait, pour des questions de fidélité des approximations que nous effectuerons à partie de la section 4.3, nous allons considérer que le nonce N_a est une fonction `nonce-A-1` non seulement de n mais aussi des valeurs courantes des autres variables de l'état de A :

```
Pa((avant-1, Ses, A, B, _, Kas, Kab, M, Nb), E
    (entre-1-et-2, Ses, A, B,
            nonce-A-1(Ses, A, B, Kas, Kab, M, Nb),
            Kas, Kab, M, Nb),
        cons ([A, B, nonce-A-1(Ses, A, B, Kas, Kab, M, Nb)], E)).
```

Ceci est exactement équivalent à la clause :

```
Pa((avant-1, Ses, A, B, _, Kas, Kab, M, Nb), E
    (entre-1-et-2, Ses, A, B, Na, Kas, Kab, M, Nb), E') :-
    Na=nonce-A-1(Ses, A, B, Kas, Kab, M, Nb),
    E'=cons ([A, B, Na], E).
```

en adoptant pour = la sémantique de l'égalité de Prolog. Quoique la deuxième forme soit plus lisible que la première, et plus proche de ce que nous avons écrit en section 3.2, nous adopterons le premier style d'écriture pour bien montrer que nous n'avons pas besoin de prédicat d'égalité.

Pour modéliser la réception du message $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$ par A dans la deuxième étape du protocole. Ici deux problèmes se posent : comme exprimer le déchiffrement de ce message sans utiliser de fonction `decrypt` comme en section 3.2, et comment vérifier que les composantes N_a et B du message une fois déchiffré sont égales aux composantes N_a et B stockées dans l'état de A, sans utiliser de prédicat d'égalité. Les deux problèmes sont très simples, en fait.

Dans le premier cas, pour déchiffrer le message reçu avec une clé K, et récupérer le texte en clair M il suffit d'exprimer que, si l'on rechiffrait M avec la clé K, on obtiendrait un terme déductible par l'intrus ; autrement dit, on doit avoir $\mathtt{knows}(E, \{M\}_K)$. Si la méthode de chiffrement était asymétrique, pour déchiffrer avec K nous demanderions que $\mathtt{knows}(E, \{M\}_{K^{-1}})$ soit satisfait, où K^{-1} est l'inverse de K pour l'algorithme de chiffrement considéré. (Si K n'est pas une clé pour l'algorithme de chiffrement donné, le déchiffrement est impossible, et la transition impliquant le déchiffrement ne sera pas franchie.)

Dans le deuxième cas, la vérification de l'égalité des deux occurrences de N_a s'opérera naturellement en plaçant la *même* variable logique N_a aux deux occurrences concernées (ci-dessous, en argument dans les états de A, et à l'intérieur du prédicat `knows`). On aboutit à la clause :

```
Pa((entre-1-et-2, Ses, A, B, Na, Kas, _, _, Nb), E,
    (entre-2-et-3, Ses, A, B, Na, Kas, Kab, M, Nb), E) :-
  knows(E, crypt(a(sa(vanilla())), [Na, B, Kab, M], Kas)).
```

Bien que nous ayons présenté la traduction en clauses de Horn sur un exemple, il est clair que le processus est en fait automatisable. Nous laissons le reste de la dérivation du protocole en exercice, ayant en fait exploré dans les deux clauses ci-dessus toutes les constructions nécessaires à modéliser les protocoles cryptographiques. En particulier, nous supposons que nous avons maintenant un ensemble $Exec$ de clauses de Horn, définissant toutes les exécutions possibles du protocole au travers d'un prédicat binaire `exec`, tel que $\mathtt{exec}(G, G')$ est déductible à partir de $Exec$ si et seulement s'il existe une exécution du protocole partant de la configuration globale G et aboutissant à la configuration globale G'.

4.2.4. Qu'est-ce qu'une preuve de sécurité ?

Nous avons dorénavant un ensemble de clauses de Horn décrivant précisément toutes les exécutions possibles d'un protocole donné. Nous avons encore à préciser comment nous modélisons les propriétés de sécurité. Il existe de nombreuses propriétés de sécurité auxquelles on peut s'intéresser, et nous allons nous intéresser uniquement à des propriétés d'*accessibilité*, c'est-à-dire des propriétés de la forme :

Peut-on atteindre, depuis une configuration globale initiale, une configuration globale vérifiant la propriété P ?

où P est une propriété des configurations globales. En guise d'exemple, nous allons considérer la propriété suivante :

Peut-on atteindre, depuis une configuration globale initiale, une configuration globale où l'agent A est dans l'état `entre-2-et-3` et possède une clé K_{ab} déductible par l'intrus ?

Autrement dit, si l'on en prend la négation, est-il toujours vrai que la clé K_{ab} que A reçoit à l'étape 2 du protocole est secrète ? Ici la propriété P est que, considérant une configuration globale de la forme `((entre-2-et-3, _, _, _, _, _, Kab, _, _), Eb, Es, E)`, on ait `knows(E, Kab)`.

Il est facile d'écrire une clause qui sera déclenchée dès que la propriété d'accessibilité est réalisée :

```
non-secret() :-
  exec(((avant-1, 0(), _, _, _, Kas-of-A, _, _),
        (avant-1, 0(), _, _, _, Kbs-of-B),
        (avant-1, 0(), _, _, _, Kab-of-S, Kas-of-S, Kbs-of-S),
        nil()),
       ((entre-2-et-3, _, _, _, _, _, Kab-of-A, _, _), _, _, E)),
  not-knows0(Kas-of-A),
  not-knows0(Kas-of-S),
  not-knows0(Kbs-of-B),
  not-knows0(Kbs-of-S),
  not-knows0(Kab-of-S),
  knows(E, Kab-of-A).
```

Littéralement, le nouveau prédicat `non-secret()` sera déductible de l'ensemble de clauses $Exec$ et de la clause ci-dessus si et seulement s'il existe une exécution du protocole, partant d'un état initial où l'intrus n'a encore récupéré aucun message (l'ensemble `E` initial est `nil()`), telle que les clés initiales K_{as} et K_{bs} des divers principaux ainsi que la clé K_{ab} créée par le serveur S sont secrètes (les conditions `not-knows0(...)`), et aboutissant à un état où A se trouve à l'état `entre-2-et-3` et dispose d'une clé K_{ab} qui est connue de l'intrus à ce point. En d'autres termes, `non-secret()` sera déductible si et seulement si la propriété d'accessibilité est vérifiée. (On notera ici qu'on n'a pas supposé que les différentes clés K_{as}, par exemple, de A et de S étaient les mêmes ; il n'est pas difficile d'imposer cette condition, cependant ceci affaiblit la portée de la propriété de sécurité vérifiée.)

Il est important de noter que le fait que `non-secret()` soit déductible signifie qu'un état où la clé K_{ab} n'est *pas* secrète est accessible depuis les conditions initiales. Autrement dit, qu'une *attaque* est possible.

Il est naturel de penser qu'une déduction, une *preuve* d'un prédicat tel que `non-secret()` devrait avoir quelque chose à voir avec une *preuve* de sécurité. Il est important de remarquer que les deux notions de preuves sont en fait complémentaires, et non identiques :
- Une *preuve* d'accessibilité (ici, de `non-secret()`) est une *attaque*.
- Une *preuve de sécurité* correspond à l'*absence* d'une preuve d'accessibilité.

En général, un moyen constructif de montrer qu'un preuve (d'accessibilité, ici) n'existe pas est de construire un contre-modèle. Cette observation cruciale est due à Peter Selinger [51]. Nous allons voir comment construire automatiquement de tels modèles, dans un grand nombre de cas, en sections 4.3 et suivantes.

4.2.5. Comment décider une propriété d'accessibilité ?

À ce point, le problème de l'accessibilité dans les protocoles cryptographiques est réduit au problème de la prouvabilité dans le fragment des clauses de Horn de la logique classique du premier ordre. Malheureusement, ce problème est indécidable.

Que la prouvabilité dans ce cadre logique soit indécidable peut signifier l'une des deux choses suivantes : ou bien la vérification de protocoles cryptographiques est intrinsèquement difficile, ou bien nous nous sommes trompés de langage pour traduire les protocoles cryptographiques et avons compliqué le problème inutilement. Il s'avère qu'en fait, la vérification de protocoles cryptographiques est effectivement difficile, en fait indécidable elle-même [21] sauf dans certains cas relativement limités (nombre de sessions borné et pas de boucle dans les rôles des principaux, état initial E_0 de l'intrus connu et fini, voir par exemple [49]). Nous allons voir que la formalisation au premier ordre offre des possibilités de vérification intéressantes, nonobstant son indécidabilité.

La première façon de décider d'un ensemble de clauses de Horn est d'utiliser un outil de démonstration automatique de formules du premier ordre. Parmi les plus adaptés aujourd'hui, on compte Otter [36], Vampire [47], SPASS [57], E-Setheo [41], notamment. Cette approche de la vérification des protocoles cryptographiques est notamment celle proposée par Weidenbach [55].

Le problème de la prouvabilité en logique du premier ordre est indécidable, mais les outils ci-dessus sont complets : s'il existe une preuve, ils la trouveront (en principe du moins). En conséquence, ces outils peuvent :
- trouver une preuve, c'est-à-dire une attaque ;
- ou terminer sans avoir trouvé une preuve, c'est-à-dire fournir une preuve de sécurité ;
- ou ne pas terminer, auquel cas le protocole est sûr, mais on ne le saura jamais.

Le deuxième cas est rare, et l'approche, telle quelle, n'aboutit que très peu souvent à une preuve de sécurité. De plus, même lorsqu'un de ces outils termine sans avoir trouvé de preuve, il est difficile d'en extraire un argument de sécurité (un modèle, donc) qui soit indépendamment vérifiable par d'autres sources — un besoin grandissant, notamment dans la certification CC [15].

Au passage, en principe la simple utilisation de Prolog serait elle-même envisageable. En pratique, Prolog réalise une stratégie de recherche de preuve incomplète, qui n'est donc même pas garantie de trouver une attaque en temps fini s'il en existe une, au contraire des outils ci-dessus.

Un cas intéressant est dû à Blanchet [7], qui montre qu'au lieu d'utiliser des techniques à base de résolution comme dans certains des outils ci-dessus (résolution ordonnée en SPASS, hyperrésolution en Otter), un raffinement légèrement différent où une forme de résolution ordonnée restreinte à certaines formes de clauses (la résolution gardée) est utilisé, pour compléter l'ensemble de clauses initiale. Si ce processus termine, alors on peut effacer un grand nombre de clauses devenues inutiles, et une autre forme de résolution (appelée recherche en arrière avec détection de cycles ; il s'agit d'hyperrésolution positive) produit alors une procédure de décision qui termine sur l'ensemble de clauses résultant. Blanchet applique cette méthode à la vérification du protocole Skeme [7]. Il est à noter que la phase de prétraitement par résolultion gardée ne termine pas toujours. De plus, le modèle de Blanchet est légèrement différent du nôtre.

4.3. Approximations et types descriptifs

Une autre approche de la vérification de protocoles cryptographiques, et en fait de n'importe quel problème indécidable, est par *approximation*. En général, l'idée est de transformer le problème P à résoudre en un problème P' qui soit décidable d'une part, et tel que d'autre part une réponse positive à P' implique une réponse positive à P.

L'approximation du problème de la prouvabilité est un problème difficile, et à l'heure actuelle non résolu de façon satisfaisante. La bonne nouvelle ici est l'observation de Selinger : vérifier des propriétés de sécurité (au moins celles d'accessibilité) est la négation du problème de la prouvabilité, à savoir le problème de la *satisfiabilité*, c'est-à-dire l'existence d'un modèle.

Or on connaît une famille extrêmement efficace d'approximations pour le problème de la satisfiabilité : les *typage descriptifs*. Il s'agit de familles d'algorithmes de typage de programmes Prolog, qui tentent de découvrir les types des variables qui mènent à une déduction réussie d'un but donné. Par exemple, considérons l'ensemble de clauses :

```
double(0(),0()).
double(s(X),s(s(Y))) :- double(X,Y).
```

Il est relativement facile de se convaincre que les seules formules dérivables de la forme $\texttt{double}(m, n)$ sont telles que m est un entier en unaire, et n est un entier pair, qui vaut le double de m.

Un mécanisme de typage descriptif est un moyen d'attribuer à chaque variable X de chaque clause un *type* τ, c'est-à-dire un ensemble de termes clos (sans variables), tel que toute utilisation de la clause dans une déduction instantiera X à un terme du type τ (plus précisément, un terme dont toutes les instances closes sont dans τ). Dans l'exemple ci-dessus, on peut ainsi attribuer à X dans la deuxième clause le type Nat des entiers unaires :

$$Nat ::= \texttt{0}()|\texttt{s}(Nat)$$

et à Y le type $Pair$ des entiers pairs :

$$Pair ::= \texttt{0}()|\texttt{s}(\texttt{s}(Pair))$$

Connaissant ces types, on constate que, si on ajoute aux clauses ci-dessus la clause

```
false :- double (X, s(0())).
```

il n'y a aucun moyen de déduire `false`. On peut bien sûr le voir en lançant un des outils de démonstration automatique mentionnés plus haut. L'exemple est suffisamment simple pour qu'un prouver à base de résolution ordonnée, comme SPASS, termine sans trouver de contradiction. Mais c'est encore plus visible par typage : si on pouvait déduire `false`, ç'aurait été avec une valeur du deuxième argument de `double` paire, par typage. Mais on ne peut déduire `false` qu'en passant la valeur un `(s(0()))`, qui n'est pas paire.

Il n'est a priori pas facile de trouver de tels types descriptifs. Frühwirth *et al.* [24] propose une méthode extrêmement élégante et complètement automatisée. Elle procède en trois étapes (nous prenons quelques libertés avec [24]), consistant à transformer l'ensemble de clauses initial S_0 en un nouvel ensemble de clauses S décrivant les types des variables :

– On crée un nouveau prédicat `type`, prenant un argument, et autant de symboles de fonctions f_P qu'il y a de symboles de prédicats P dans l'ensemble de clauses S_0 de départ. L'idée est que $\texttt{type}(f_P(t_1, \ldots, t_n))$ sera déductible de S si $P(t_1, \ldots, t_n)$ est déductible de S_0 : `type` décrit un sur-ensemble de toutes les formules $P(t_1, \ldots, t_n)$ déductibles de S_0.
– Pour toute clause C de S_0, de la forme

$$P(t_1, \ldots, t_n)\texttt{:-}P_1(t_{11}, \ldots, t_{1n_1}), \ldots, P_m(t_{m1}, \ldots, t_{mn_m})$$

et chaque variable libre X de C, on crée un prédicat $\texttt{type-}C\texttt{-}X$, prenant un argument. L'idée est que $\texttt{type-}C\texttt{-}X(t)$ sera déductible de S si t est une valeur possible pour la variable X dans une déduction utilisant la clause C, comme dans l'exemple ci-dessus. Soient $X_1, \ldots, X_k$ les variables libres dans la tête $P(t_1, \ldots, t_n)$, listées de gauche à droite. (Il est possible qu'une même variable soit donc listée plusieurs fois.) Soient $Y_1, \ldots, Y_k$ des variables fraîches, et distinctes deux à deux. Soient $\tilde{t}_1, \ldots, \tilde{t}_n$ les termes $t_1, \ldots, t_n$ où X_i est remplacée par Y_i, $1 \leq i \leq k$. On remplace alors C par les clauses :

$$\begin{array}{rcl}
\texttt{type}(f_P(\tilde{t}_1, \ldots, \tilde{t}_n)) & \texttt{:-} & \texttt{type-}C\texttt{-}X_1(Y_1), \ldots, \texttt{type-}C\texttt{-}X_k(Y_k) \\
\texttt{type-}C\texttt{-}X_1(X_1) & \texttt{:-} & \texttt{type}(f_{P_1}(t_{11}, \ldots, t_{1n_1})), \ldots, \texttt{type}(f_{P_m}(t_{m1}, \ldots, t_{mn_m})) \\
 & & \ldots \\
\texttt{type-}C\texttt{-}X_k(X_k) & \texttt{:-} & \texttt{type}(f_{P_1}(t_{11}, \ldots, t_{1n_1})), \ldots, \texttt{type}(f_{P_m}(t_{m1}, \ldots, t_{mn_m}))
\end{array}$$

Par exemple, en partant des clauses ci-dessus, on obtient :

```
type(f-double(0(), 0())).
type(f-double(s(X), s(s(Y)))) :- type-C2-X(X), type-C2-Y(Y).
type-C2-X(X) :- type(f-double(X,Y)).
type-C2-Y(Y) :- type(f-double(X,Y)).
```

On peut se demander quel est l'avantage de décrire les types des variables d'un ensemble de clauses S_0 par un autre ensemble de clauses. Il se résume en ceci : savoir si un type est vide est décidable — et le problème est DEXPTIME-complet [24].

Ceci nous mène à une procédure d'approximation, qui termine en temps exponentiel, pour examiner la sécurité des protocoles cryptographiques :

1. Traduire le protocole cryptographique en un ensemble $Exec$ de clauses, plus les clauses correspondant aux propriétés de sécurité, comme en section 4.2, obtenant un ensemble de clauses S_0.
2. Produire l'approximation par types descriptifs S.
3. Décider par l'algorithme de [24] si l'une des propriétés d'accessibilité est déductible de S. Si ce n'est pas le cas, alors le protocole cryptographique est sûr, par rapport aux propriétés demandées, sous les hypothèses initiales données, et dans le modèle de sécurité choisi. Sinon, on ne peut pas conclure.

On notera que les deux premiers étapes s'effectuent en temps polynomial. Bien que la dernière partie s'effectue en temps exponentiel, elle est souvent rapide (d'une fraction de seconde à quelques minutes).

Il est montré que l'algorithme de [24] produit en plus un automate d'arbres finis décrivant exactement un modèle, qui de plus est un modèle fini. Ce modèle est, d'après l'observation de Selinger, exactement ce que nous cherchions en termes de preuve de sécurité.

Dans l'exemple du prédicat `double` ci-dessus, cet algorithme essentiellement produit l'ensemble de clauses suivant pour le prédicat de typage `type-C2-Y` :

```
type-C2-Y(0()).
type-C2-Y(s(s(Y))) :- type-C2(Y).
```

qui est exactement le type des entiers unaires pairs, que nous avions écrit sous forme du type $Pair$ ci-dessus.

Nous avons donc trouvé une méthode automatique qui trouve des preuves de sécurité de propriétés d'accessibilité de protocoles cryptographiques. Cette méthode n'est pas complète : il y a des protocoles cryptographiques sûrs que cette méthode ne parviendra pas à prouver. Mais elle est correcte : un protocole présentant une attaque ne pourra pas être prouvé par cette méthode (il n'y a aucun modèle). En un sens, il s'agit bien d'un algorithme de typage : tout protocole typé est correct, mais il existe des protocoles corrects non typés ; c'est exactement la philosophie du typage de ML par exemple [39], où tout programme typé est correct au sens où il n'y aura pas d'erreur de types à l'exécution, mais il existe des programmes ne présentant aucune erreur à l'exécution et qui pourtant ne sont pas bien typés.

La question cruciale ici est : cette méthode de vérification est-elle suffisamment précise ? Autrement dit, la proportion de protocoles sûrs que cette méthode arrive à montrer corrects est-elle satisfaisante ? Nous allons répondre à cette question en section 4.5.

4.4. À propos de la relation entre clauses et automates d'arbres

En fait, les techniques de résolution ordonnée [23], qui forment déjà le cœur du prouveur SPASS [57], terminent sur la classe des clauses que l'on obtient par la procédure d'approximation ci-dessus. (On pourra aussi consulter [27] pour un traitement complet de ce genre de clauses, généralisés à l'ordre supérieur, par des techniques de résolution ordonnée.)

Fixons une signature Σ, c'est-à-dire un ensemble de symboles de fonction, équipés d'une *arité*, c'est-à-dire du nombre d'arguments que prend chaque symbole de fonction. Un *automates d'arbres* non déterministe $\mathcal{A}$ est un ensemble fini de clauses de la forme :

$$P(f(X_1, \ldots, X_n)) \mathbin{:\!-} P_1(X_1), \ldots, P_n(X_n) \tag{1}$$

Les clauses (1) sont appelées des *clauses montantes*, ou des transitions ordinaires d'automates d'arbres. Elles peuvent se lire : "si X_1 est un terme reconnu à l'état P_1, et ..., et X_n est reconnu à l'état P_n, alors le terme $f(X_1, \ldots, X_n)$ est reconnu à l'état P".

Par exemple, l'automate suivant reconnaît l'ensemble des listes d'entiers pairs :

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

On représente souvent les automates d'arbres par des hypergraphes, c'est-à-dire des graphes dont les arcs (les *transitions*) passent par deux ou davantage de sommets. L'automate ci-dessus se représente alors notamment comme en figure 5. Il y a notamment une transition étiquetée par le symbole de fonction `cons` qui va de la paire d'états `q-even`, `q-list-even` vers l'état `q-list-even`, et qui représente la dernière clause ci-dessus.

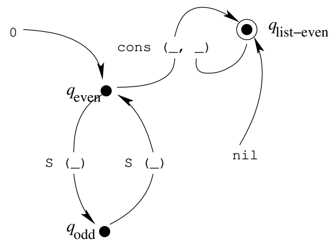

FIG. 5 – Un automate d'arbres reconnaissant les listes d'entiers pairs

En réalité, les clauses montantes ne sont des transitions normales d'automates d'arbres que quand les variables X_1, ..., X_n sont distinctes deux à deux. Dans le cas contraire, on obtient des automates dits à *contraintes d'égalité entre frères* [8].

On peut compliquer la notion d'automates d'arbres. Notamment, un automate d'arbres *alternant* a en plus des *clauses intersection* de la forme :

$$P(X) \mathbin{:\!-} P_1(X), \ldots, P_n(X) \tag{2}$$

Un automate *bidirectionnel* peut aussi inclure des *clauses descendantes*, de la forme :

$$P_i(X_i) \mathbin{:\!-} P(f(X_1, \ldots, X_n)), P_{i_1}(X_{i_1}), \ldots, P_{i_k}(X_{i_k}) \tag{3}$$

où $1 \leq i \leq n$, $1 \leq i_1 < \ldots < i_k \leq n$. Si $k = 0$, il s'agit d'une *clause descendante standard*, qui signifie "si $f(X_1, \ldots, X_n)$ est reconnu en P, alors X_i est reconnu en P_i", et qui est souvent appelée une *transition d'empilement* ("pushdown transition"). Les autres clauses descendantes sont dites *conditionnelles*. Par exemple,

$$P_1(x_1) \mathbin{:\!-} P(f(X_1, X_2)), P_2(X_2)$$

exprime que si $f(X_1, X_2)$ est reconnu en P, et pourvu que X_2 soit aussi reconnu en P_2, alors X_1 est reconnu en P_1. Par exemple, si la clause

```
knows(crypt(A, M, K)) :- knows(M), knows(K).
```

qui exprime que l'intrus peut toujours chiffrer, est une clause montante, la clause symétrique

```
knows(M) :- knows(crypt(A, M, K)), knows(K)
```

qui exprime que l'intrus peut chiffrer à condition de posséder la bonne clé, est une clause descendante conditionnelle.

Rappelons qu'un *atome* est l'application $P(t)$ d'un symbole de *prédicat* P à un terme t. Un *littéral* est un atome A (un littéral *positif*) ou la négation $\neg A$ (un littéral *négatif*) d'un atome A. Une *clause* est une disjonction $L_1 \vee \ldots \vee L_n$ de littéraux. Il s'agit d'une clause *de Horn* si et seulement si elle contient au plus un littéral positif. Nous avons constamment noté la clause de Horn $A \vee \neg B_1 \vee \ldots \vee \neg B_n$ jusqu'ici sous la forme $A\mathbin{:\!-}B_1, \ldots, B_n$.

Une *interprétation de Tarski* I est un triplet $(D, (I_f)_{f \text{ symbole de fonction}}, (I_P)_{P \text{ symbole de predicat}})$ où le *domaine* D de I est un ensemble non vide ; pour chaque symbole de fonction f d'arité n, I_f est une fonction de D^n vers D ; et pour chaque prédicat P, I_P est un sous-ensemble de D. L'interprétation d'un terme t dans un environnement ρ, qui envoie chaque variable vers un élément de D, est définie par $I\,[\![X]\!]\,\rho = \rho(X)$, $I\,[\![f(t_1, \ldots, t_n)]\!]\,\rho = I_f(I\,[\![t_1]\!]\,\rho, \ldots, I\,[\![t_n]\!]\,\rho)$. La relation $I, \rho \models F$ ("F est vraie dans l'interprétation I, dans l'environnement ρ) est définie sur les formules atomiques et les littéraux par $I, \rho \models P(t)$ ssi $I\,[\![t]\!]\,\rho \in I_P$, et $I, \rho \models \neg P(t)$ ssi $I\,[\![t]\!]\,\rho \notin P$. Pour chaque clause C, on dit que I *satisfait* C, en abrégé $I \models C$, si et seulement si pour tout environnement ρ, il y a un littéral L dans C tel que $I, \rho \models L$. Pour tout ensemble S de clauses, on pose $I \models S$, où S est un ensemble de clauses, ssi $I \models C$ pour toute clause C de S. S est *insatisfiable* ssi $I \models S$ pour aucune interprétation de Tarski I.

Une *interprétation de Herbrand* est un ensemble d'atomes clos (sans variables libres). Il s'agit d'un cas particulier d'une interprétation de Tarski, où D est figé et contraint à être l'ensemble des termes clos, et I_f envoie chaque n-uplet de termes clos $(t_1, \ldots, t_n)$ vers le terme clos $f(t_1, \ldots, t_n)$. Dans ces conditions, se donner une collection d'ensembles I_P de termes clos (une interprétation de Tarski) revient exactement à se donner une interprétation de Herbrand, à savoir l'ensemble des atomes vrais dans I ; et réciproquement.

Un résultat standard de théorie de la preuve est qu'un ensemble de clauses S est satisfiable si et seulement si S a un modèle de Herbrand, autrement dit une interprétation de Herbrand I telle que $I \models S$. Il est bien connu, et facile de voir, que tout ensemble satisfiable de clauses de Horn a un *plus petit* modèle de Herbrand, qui estjuste l'intersection de tous ses modèles de Herbrand. C'est en particulier vrai pour un automate d'arbres, possiblement alternant ou bidirectionnel.

On peut alors définir formellement l'ensemble des termes *reconnus* à l'*état* P par l'automate $\mathcal{A}$, pour chaque symbole de prédicat P, comme étant l'ensemble des termes clos t tels que $P(t)$ soit dans le plus petit modèle de Herbrand de $\mathcal{A}$. L'ensemble des termes reconnus à P forme le *langage* $L_P(\mathcal{A})$. On dit que l'état P est *vide* dans $\mathcal{A}$ si et seulement si $L_P(\mathcal{A})$ est vide.

Il n'est pas difficile de voir que P est vide dans $\mathcal{A}$ si et seulement si $\mathcal{A}$ augmenté de la clause $\neg P(X)$ est satisfiable. En effet, si P est vide alors le plus petit modèle de Herbrand de $\mathcal{A}$ ne contient aucun atome clos de la forme $P(t)$, donc rend $\neg P(X)$ vraie. Réciproquement, si $\mathcal{A}$ plus $\neg P(X)$ est satisfiable, alors son plus petit modèle de Herbrand ne contient certainement aucun atome de la forme $P(t)$. Comme tout modèle de $\mathcal{A}$ plus $\neg P(X)$ en est aussi un de $\mathcal{A}$, le plus petit modèle de Herbrand de $\mathcal{A}$ est inclus dans celui de $\mathcal{A}$ plus $\neg P(X)$, donc ne contient aucun atome clos de la forme $P(t)$ lui non plus. Donc P est vide dans $\mathcal{A}$.

Plus généralement, on peut considérer des clauses qui ne sont pas de Horn. Appelons un *bloc* toute clause de la forme :

$$\bigvee_{i=1}^{m} \pm_i P_i(X) \tag{4}$$

où les signes $\pm_i$ sont soit $+$ soit $-$ ($+A$ signifiant A, $-A$ signifiant $\neg A$). Il est à noter qu'un bloc a au plus une variable libre X. Écrivons $B(X)$ un bloc de variable libre X. Considérons des clauses qui sont de l'une des deux formes suivantes : les *clauses simples* sont les blocs, et les *clauses complexes* sont de la forme :

$$\bigvee_{i=1}^{m} \pm_i P_i(f(X_1, \ldots, X_n)) \vee \bigvee_{j=1}^{n} B_j(X_j) \tag{5}$$

où $B_1(X_1), \ldots, B_n(X_n)$ sont des blocs, et $m \geq 1$. Noter que f et les variables X_1, …, X_n sont les mêmes pour chaque i dans (5), et que les variables libres de la clause sont toutes libres dans $f(X_1, \ldots, X_n)$.

Les clauses montantes et les clauses descendantes sont toutes des clauses complexes. Les clauses simples incluent les clauses intersection et les clauses de la forme $\neg P(X)$. Mais ce format permet encore davantage de liberté, et il se trouve que ce format de clauses correspond essentiellement à un format de *contraintes ensemblistes positives*, ou encore à un format clausal pour la sous-classe monadique de la logique du premier ordre [2], comme observé par Bachmair et Ganzinger [4].

La technique de *résolution ordonnée* avec *splitting* est une méthode de preuve correcte et complète pour la logique du premier ordre en forme clausale : on consultera [23, 56] pour une introduction au domaine. Il s'agit de techniques inventées vers la fin des années 1960. La résolution ordonnée avec splitting termine sur des clauses de la forme (4) et (5), l'ordre sur les atomes étant pris comme l'ordre naturel sur la taille des atomes : en fait cette stratégie de recherche de preuve ne peut fabriquer qu'un nombre fini de clauses, et termine donc après les avoir toutes engendrées. Si l'ensemble de clauses restant contient la clause vide, alors l'ensemble de clauses de départ est insatisfiable, sinon il est satisfiable (il y a un modèle).

On notera que les clauses obtenus par l'algorithme de la section 4.3 sont presque des clauses ascendantes et des clauses descendantes. On peut en fait définir une procédure d'approximation de clauses de Horn générales en clauses d'automates bidirectionneles qui est très proche de celle de la section 4.3 comme suit. On suppose d'abord qu'on a transformé les clauses en clauses dont les symboles de prédicat prennent tous exactement un argument. Ensuite, tant qu'il y a un terme t dans le clause C qui n'est ni une variable ni de la forme $f(X_1, \ldots, X_n)$, disons $C = (C_0 \vee \pm P(t))$, alors t est de la forme $f(u_1, \ldots, u_n)$ en particulier. Créer n prédicats nouveaux P_1, …, P_n, et n variables fraîches X_1, …, X_n, et remplacer la clause C par :

$$C_0 \vee \pm P_1(u_1) \vee \ldots \vee \pm P_n(u_n) \tag{6}$$

$$\pm P(f(X_1, \ldots, X_n)) \vee \mp P_j(X_j) \qquad (1 \leq j \leq n) \tag{7}$$

où $\mp$ est le signe opposé de $\pm$ (toutes les occurrences de $\pm$ dénotent ici le *même* signe).

Une fois ces transformations terminées, on transforme toute clause C de la forme $C_0 \vee L_1 \vee L_2$, où L_1 et L_2 sont deux littéraux avec des ensembles de variables libres distincts mais non disjoints, en $C_0 \vee L_1 \vee C'_0 \vee L'_2$, où $C'_0 \vee L'_2$ est une version renommée de $C_0 \vee L_2$, c'est-à-dire dont toutes les variables libres ont été remplacées par de nouvelles. Ce processus termine, et fournit des disjonctions de clauses simples et complexes qui n'ont aucune variable en commun. Ceci se ramène aisément à un ensemble de clauses simples ou complexes, décidables par résolution ordonnée et splitting, par splitting. (Voir par exemple [27].)

Nous avons donc rebouclé la boucle : partant d'une modélisation en clauses de Horn, que l'on peut traiter par des techniques de résolution, nous avons défini une approximation qui est une légère extension d'un automate bidirectionnel alternant d'arbres, et de nouveau les techniques fondées sur la résolution ordonnée s'appliquent ; cette fois-ci elles terminent, et fournissent un algorithme de décision pour la satisfiabilité des clauses d'automates, donc pour la vacuité des langages définis par automates bidirectionnels alternants.

4.5. Transformations de programmes et améliorations de la précision

Revenons à la question posée en fin de section 4.3 : la méthode d'approximation des ensembles de clauses de Horn par des automates d'arbres est-elle suffisamment précise ? Il se trouve que non, cette méthode de vérification n'est pas suffisamment précise dans le cadre des protocoles cryptographiques. Cependant, certaines transformations préliminaires de l'ensemble de clauses S_0 améliorent grandement la précision de l'algorithme de typage de la section 4.3.

4.5.1. Inlining

Considérons les différents appels à `knows` dans la modélisation de la section 4.2. L'algorithme de typage descriptif de la section 4.3 va chercher à déterminer un type pour des termes de la forme $f_{\texttt{knows}}(E, M)$, c'est-à-dire à déterminer un sur-ensemble de tous les couples (E, M) tels que E est une liste de messages récupérés par l'intrus et M est un ensemble possible de messages déductibles de E. L'ensemble de tous ces couples sera décrit en un unique état (par un unique prédicat) de l'automate. Dans de nombreux cas, ceci induit une perte d'information, et la technique de la section 4.3 va faire un amalgame entre des couples (E, M) venant de l'étape 1 du protocole avec des couples venant de l'étape 2, avec des couples venant de l'étape 3 ; elle sera alors insuffisante pour montrer qu'un protocole est effectivement sûr.

Une façon simple de contourner ce problème est de définir des versions spécialisées du prédicat `knows` pour chaque étape du protocole. En d'autres mots, si l'on considère deux clauses différentes utilisant `knows`, comme celles définissant `Pa` en section 4.2.3, on souhaite les réécrire en remplaçant le prédicat `knows` dans chaque par un nouveau prédicat, `knows-1` dans le première, `knows-2` dans la seconde, et ainsi de suite. On remplacera donc des clauses de la forme :

```
... :- ..., knows(E, M), ...
... :- ..., knows(E, M), ...
```

par

```
... :- ..., knows-1(E, M), ...
... :- ..., knows-2(E, M), ...
```

et l'on créera autant de clauses définissant `knows-1`, resp. `knows-2`, qu'il y avait de clauses définissant `knows` (voir section 4.2.2). Autrement dit, on créera les clauses :

```
knows-1(E, M) :- knows0(M).
knows-1(E, M) :- in-list-1(E,M).
in-list-1(cons(M, _), M).
in-list-1(cons(_, L), M) :-
   in-list-1(L, M).
knows-1(E, crypt(_, M, K)) :-
   knows-1(E, M), knows-1(E, K).
```

```
knows-2(E, M) :- knows0(M).
knows-2(E, M) :- in-list-2(E,M).
in-list-2(cons(M, _), M).
in-list-2(cons(_, L), M) :-
   in-list-2(L, M).
knows-2(E, crypt(_, M, K)) :-
   knows-2(E, M), knows-2(E, K).
```

et ainsi de suite.

Formellement, pour *inliner* un prédicat P (ici, `knows`), on liste les clauses $C_1, \ldots, C_n$ qui ont un atome de la forme $P(...)$ dans leur corps (à droite de `:-`), mais pas dans leur tête (à gauche de `:-`). On crée n nouveaux prédicats $P_1, \ldots, P_n$, et on remplace P par P_i dans C_i, pour chaque i entre 1 et n. La seconde étape consiste à créer les clauses définissant les nouveaux prédicats P_i. Pour ceci, soit $\mathcal{P}(P)$ le plus petit ensemble de prédicats contenant P et tel que si $Q \in \mathcal{P}(P)$ et il existe une clause de la forme $Q(...)\texttt{:-}Q_1(...), \ldots, Q_m(...)$, alors $Q_1, \ldots, Q_m \in \mathcal{P}(P)$. Soit aussi $\mathcal{C}(P)$ l'ensemble des clauses dont la tête est de la forme $Q(...)$ avec $Q \in \mathcal{P}(P)$. Pour définir P_i il suffit de, d'abord, créer de nouveaux prédicats Q^i distincts deux à deux, pour chaque $Q \in \mathcal{P}(P_i)$, $1 \leq i \leq n$, ensuite de créer pour chaque clause $C \in \mathcal{C}(P)$, la clause obtenue en remplaçant tout prédicat Q apparaissant dans C par Q^i.

Cette méthode est appelée inlining, parce qu'elle correspond exactement à la technique d'inlining de procédures en compilation, ici adaptée à Prolog. L'inlining produit un ensemble de clauses S_1 à partir d'un ensemble de clauses S_0, tel que, pour tout prédicat *non inliné* P, $P(t_1, \ldots, t_n)$ est déductible de S_0 si et seulement s'il est déductible de S_1 : c'est en cela que l'inlining est correct.

4.5.2. Partitionnement

Un autre cas de perte d'information se présente notamment dans la gestion des numéros de session. Rappelons que nous avons introduit des numéros de session `Ses` en section 4.2.3. Nous ne nous en sommes pas servis ; typiquement, leur besoin se fait sentir lorsque les principaux tournent en boucle. Ici nous avons considéré que A effectuait les étapes 1, 2, 3, 4, 5, puis s'arrêtait. Dans la réalité il est plus raisonnable de penser que A va ensuite boucler et revenir à l'étape 1, par exemple. Il est alors raisonnable d'incrémenter le numéro de session. Ceci permet notamment de distinguer deux nonces créés en deux étapes identiques du protocole, mais dans deux sessions différentes. Notamment, les nonces créés par A à l'étape 1 dans la session initiale seront de la forme :

```
nonce-A-1(0(), ...)
```

alors que ceux créés dans les sessions suivantes seront de la forme :

```
nonce-A-1(s(_), ...)
```

On peut raffiner la précision de l'approximation de la section 4.3 en distinguant les deux cas. Par exemple, la transition pour A passant de `avant-1` à `entre-1-et-2` de la section 4.2.3 sera découpée en deux clauses selon que la session est la session initiale ou pas :

```
Pa((avant-1, 0(), A, B, _, Kas, Kab, M, Nb), E
    (entre-1-et-2, 0(), A, B,
                nonce-A-1(0(), A, B, Kas, Kab, M, Nb),
                Kas, Kab, M, Nb),
            cons ([A, B, nonce-A-1(0(), A, B, Kas, Kab, M, Nb)], E)).
Pa((avant-1, s(Ses'), A, B, _, Kas, Kab, M, Nb), E
    (entre-1-et-2, s(Ses'), A, B,
                nonce-A-1(s(Ses'), A, B, Kas, Kab, M, Nb),
                Kas, Kab, M, Nb),
            cons ([A, B, nonce-A-1(s(Ses'), A, B, Kas, Kab, M, Nb)], E)).
```

Cette transformation est générale, et nous l'appelons *partitionnement*, ici de la variable `Ses` dans la clause donnée.

Cette transformation nécessite cependant pour être plus précise que les termes soient typés. Bien que nous n'ayons pas insisté sur ce point, c'est le cas dans la formalisation utilisée ici, voir le début de la section 4.1 : pour chaque variable X que nous souhaitons partitionner dans une clause C, considérons son type τ (message, `*D*`, `*algo*`, etc.), et enfin la liste f_1, …, f_n des symboles de fonctions dont le type résultat est τ. Nous fabriquons alors n clauses, C dans laquelle X est remplacée par $f_1(X_{11}, \ldots, X_{1m_1})$, …, enfin C dans laquelle X est remplacée par $f_n(X_{n1}, \ldots, X_{nm_n})$. Ici les X_{ij} sont des variables distinctes deux à deux, et nouvelles.

Le partitionnement préserve tous les plus petits modèles de Herbrand, mais pas nécessairement les autres modèles.

4.5.3. Ensembles magiques

Une autre transformation classique en Prolog est celle des *ensembles magiques*. Elle est utile en particulier en conjonction avec la transformation par partitionnement, qui a tendance à engendrer de nombreuses clauses qui ne servent à rien dans aucune dérivation. La technique des ensembles magiques permet de guider la recherche de preuves [5].

La technique est relativement simple encore une fois. Supposons que nous ayons une clause A `:-` $B_1, \ldots, B_n$. Créons de nouveaux symboles de prédicats (que nous représenterons en ajoutant le préfixe magic-). La formule magic-A intuitivement sera vraie si on estime avoir besoin de prouver A. On modifie alors la clause ci-dessus en la clause

$$A :- \texttt{magic-}A, B_1, \ldots, B_n$$

qui exprime que l'on n'aura le droit d'utiliser cette clause que si magic-A est vraie, c'est-à-dire si on a demandé à prouver A. La technique est élégante, dans la mesure où l'on peut ainsi guider la recherche de preuve de sorte qu'elle ne cherche à dériver que des formules qui ont une chance de servir à la preuve, sans avoir à se plonger dans des détails algorithmiques : le formalisme logique se charge de tout.

On doit aussi rajouter des clauses exprimant les dépendances entre formules. On écrira notamment :

$$\begin{array}{rcl} \texttt{magic-}B_1 & :- & \texttt{magic-}A \\ \texttt{magic-}B_2 & :- & \texttt{magic-}A \\ & \ldots & \\ \texttt{magic-}B_n & :- & \texttt{magic-}A \end{array}$$

Autrement dit, si on a besoin de dériver A, on a a priori à dériver B_1, …, B_n. Ce sont les *ensembles magiques* (“magic sets”). Le raffinement des *motifs magiques* (“magic templates”) introduit une interaction non triviale avec la recherche de preuve proprement dite. Au lieu des clauses ci-dessus, on écrit :

$$\begin{array}{rcl} \texttt{magic-}B_1 & :- & \texttt{magic-}A \\ \texttt{magic-}B_2 & :- & \texttt{magic-}A, B_1 \\ & \ldots & \\ \texttt{magic-}B_n & :- & \texttt{magic-}A, B_1, \ldots, B_{n-1} \end{array}$$

Autrement dit, si on a besoin de dériver A on doit dériver B_1. Mais on n'a à dériver B_2 que si d'une part on a besoin de dériver A, mais aussi si d'autre part on a réussi à prouver B_1.

Il est finalement nécessaire d'ajouter une clause à l'ensemble de clauses transformé exprimant quel est le résultat que nous souhaitons démontrer. Dans l'exemple de la section 4.2, si l'on souhaite dériver non-secret(), on ajoutera à l'ensemble des clauses transformées par ensembles ou motifs magiques, la clause :

```
magic-non-secret().
```

Ainsi la recherche de preuve sera spécialisée à la recherche d'une attaque sur le secret de K_{ab} du principal A à l'étape 2, à l'exclusion de toute autre dérivation. On consultera [5] pour le cas de formules du premier ordre (avec variables). Ce n'est pas spécialement plus compliqué.

Ces transformations ne préservent la dérivabilité que des atomes $P(t_1, \ldots, t_n)$ pour lesquels on a ajouté la clause magic-$P(t_1, \ldots, t_n)$.

4.6. À propos du besoin de théories équationnelles

Il est parfois besoin de raisonner modulo une *théorie équationnelle*, c'est-à-dire en considérant que certaines égalités entre termes doivent être valides. Jusqu'ici, deux termes (deux messages) étaient égaux si et seulement s'ils étaient deux termes identiques, ce qui cadrait bien avec notre vision d'une cryptographie idéale. Cependant, un certain nombre de primitives cryptographiques vérifient des équations supplémentaires.

Nous donnons un exemple dans le domaine des schémas d'accord de clé en groupe (“group key agreement”). Considérons le protocole d'accord de clé initial IKA.1 [52] (anciennement GDH.2). Ce protocole est utilisé pour créer une clé de groupe initial dans la suite de protocoles CLIQUES, et fonctionne comme suit.

On suppose qu'on dispose d'une fonction de hachage cryptographique e, et une opération $\oplus$, qui est *associative* et *commutative*, et a un élément neutre $\mathbf{0}$. Autrement dit, on considère des messages qui sont des termes construits à l'aide des symboles de fonction e, $\oplus$ et $\mathbf{0}$, et modulo la théorie équationnelle AC1 :

$$(M_1 \oplus M_2) \oplus M_3 = M_1 \oplus (M_2 \oplus M_3) \qquad M_1 \oplus M_2 = M_2 \oplus M_1 \qquad M \oplus \mathbf{0} = M$$

Typiquement, ces fonctions sont réalisés par une exponentielle modulaire, comme dans le protocole de Diffie-Hellman [19]. Autrement dit, on a fixé deux constantes entières α et N, et $e(M)$ vaut $\alpha^M \bmod N$. L'opération $\oplus$ est alors la multiplication et $\mathbf{0}$ est l'entier 1. Nous utilisons aussi la fonction binaire `cons` et la constante $\texttt{nil}()$ pour représenter des listes, et nous abrégeons $\texttt{cons}(M_1, \texttt{cons}(M_2, \ldots, \texttt{cons}(M_n, \texttt{nil})\ldots))$ par la notation $[M_1, M_2, \ldots, M_n]$. Par souci de simplicité, nous supposons qu'il y a trois membres dans le groupe $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$. Le protocole IKA.1 est défini pour un nombre arbitraire de membres. Le but est que tous les membres obtiennent une clé commune, connue d'eux seuls. De plus, aucun ne doit pouvoir influer sur le choix de la clé, typiquement imposer une clé qu'il aurait choisie aux autres membres.

On suppose que quiconque possède $e(M)$ et M' peut calculer $e(M \oplus M')$. C'est certainement le cas dans la réalisation ci-dessus : si j'ai $e(M) = \alpha^M \bmod N$, alors je peux le mettre à la puissance M' modulo N, et obtenir $(\alpha^M)^{M'} = \alpha^{MM'} \bmod N = e(M \oplus M')$.

Passons à la description du protocole proprement dit. En premier, IKA.1 effectue une phase *montante* (“upflow”) : $\mathcal{M}_1$ envoie à $\mathcal{M}_2$ la paire $[e(\mathbf{0}), e(N_1)]$, où N_1 est un nonce frais ; nous allons modéliser ici N_1 comme une simple constante. Ensuite $\mathcal{M}_2$ envoie $[e(N_2), e(N_1), e(N_1 \oplus N_2)]$ à $\mathcal{M}_3$, où N_2 est un autre nonce frais (une autre constante $N_2 \neq N_1$). Ceci est possible, grâce à l'hypothèse que quiconque peut fabriquer $e(M \oplus M')$ à partir de $e(M)$ et M'.

Une fois ceci fait, $\mathcal{M}_3$ commence la phase *descendante* (“downflow”), et diffuse à tous le message $[e(N_2 \oplus N_3), e(N_1 \oplus N_3)]$, où N_3 est un troisième nonce créé par $\mathcal{M}_3$. À partir de ce message, tous les membres peuvent calculer la clé de groupe $e(N_1 \oplus N_2 \oplus N_3)$.

On peut modéliser toutes les exécutions du protocole via des clauses de Horn comme en section 4.2, cette fois-ci modulo la théorie AC1. Écrivons quelques-unes de ces clauses. La technique de typage descriptif de la section 4.4 s'adapte au cas des théories équationnelles. En fait, on peut même modéliser IKA.1 sous forme de clauses de Horn modulo AC1 qui sont directement des clauses d'automates d'arbres bidirectionnels (non alternants), voir [29]. Le problème de la vacuité des automates d'arbres bidirectionnels *alternants* modulo AC1 est indécidable, mais en absence d'alternance, c'est-à-dire de clauses intersection, et de clauses descendantes conditionnelles, ce problème est de nouveau décidable ; en fait, même le problème de la vacuité de l'intersection d'un nombre fini de langages définis sans alternance est décidable [30], ce qui permet de traiter le cas du protocole IKA.1 [29]. L'algorithme de décision est cependant non trivial.

Le lecteur sera peut-être curieux de savoir que la propriété de secret de la clé de groupe échoue. Autrement dit, il y a une attaque, qui consiste pour l'intrus à rejouer le premier message lors du troisième échange de messages [38]. Il est à noter qu'IKA.1 a en fait été conçu pour un modèle où l'intrus est un *voyeur pur*, c'est-à-dire que l'intrus ne peut en fait qu'espionner le trafic réseau, et non fabriquer des messages, ni envoyer des messages, ni détourner des lignes de communication. La version authentifiée d'IKA.1 semble éviter cette attaque. Sa modélisation nécessite l'utilisation d'automates modulo AC1 non alternants mais avec des clauses descendantes conditionnelles. Le problème de la décidabilité de tels automates est à l'heure actuelle ouvert.

5. Conclusion

Le domaine de la modélisation et de la vérification des protocoles cryptographiques a connu une véritable explosion dans les années 1990. On dispose à l'entrée des années 2000 de toute une gamme de modèles et de méthodes. Les modèles dérivés de celui de Dolev et Yao semblent être les plus avancés en ce qui concerne l'expressivité et l'automatisation, mais ils sont encore relativement abstraits, comparés notamment au modèle plus opérationnel du spi-calcul et du pi-calcul appliqué, et surtout au modèle réaliste des machines de Turing probabilistes cher aux cryptologues. Il reste encore de nombreuses améliorations à apporter aux modèles à la

Dolev-Yao (automatisation, gestion des différents modes multi-sessions, extension aux problèmes propres au commerce électronique ou à d'autres domaines, par exemple le vote anonyme, en particulier). A côté de cela, il est nécessaire aujourd'hui de formaliser les rapports entre les différents modèles, par exemple pour justifier formellement certains choix apparemment arbitraires du modèle de Dolev-Yao (par exemple, pourquoi l'intrus n'est-il pas autorisé à créer des nonces ? Ceci poserait des problèmes dans la vérification, et on peut supposer sans perte de généralité que tous les nonces de l'intrus ont été créés à l'avance — à condition de faire attention à ne pas imposer des conditions initiales trop contraignantes). Plus finement, on peut souhaiter intégrer des considérations probabilistes dans des modèles à la Dolev-Yao, dans le but d'évaluer les risques d'attaques dans le style du modèle à base de machines de Turing probabilistes. Quelques tentatives préliminaires ont déjà été effectuées [33].

Références

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.

[2] W. Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[3] D. Amadio, Roberto et Lugiez. On the reachability problem in cryptographic protocols. In *CONCUR'2000*, pages 380–394. Springer Verlag LNCS 1877, 2000.

[4] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93)*, pages 75–83. IEEE Computer Society Press, 1993.

[5] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the International Conference on Management of Data and Symposium on Principles of Database Systems*. ACM Press, 1986.

[6] A. Bidoit, Michel et Boisseau. Abstract interpretation : An algebraic approach. In *Workshop on Algebraic Data Types (WADT'01)*. Springer Verlag LNCS, 2001.

[7] B. Blanchet. Abstracting cryptographic protocols by Prolog rules (invited talk). In P. Cousot, editor, *Proceedings of the 8th International Static Analysis Symposium (SAS'2001)*, pages 433–436, Paris, France, 2001. Springer-Verlag Lectures Notes in Computer Science 2126.

[8] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *Proceedings of Symp. on Theoretical Aspects of Computer Science (STACS '92)*, pages 161–172. Springer-Verlag LNCS 577, 1992.

[9] D. Bolignano. Une approche à la vérification formelle des protocoles cryptographiques. *Technique et Science Informatique*, 1995.

[10] D. Bolignano. Vérification formelle de protocoles cryptographiques à l'aide de Coq. In *Actes des journées GDR*, 1995.

[11] D. Bolignano. Formal verification of cryptographic protocols. In *Proceedings of the ACM Conference on Computer and Communication Security*, 1996.

[12] D. Bolignano. Towards a mechanization of cryptographic protocol verification. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV'97)*, pages 131–142. Springer LNCS 1254, 1997.

[13] D. Bolignano. Towards the formal verification of electronic commerce protocols. In *10th IEEE Computer Security Foundations Workshop*, 1997.

[14] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society*, 426(1871) :233–271, 1989.

[15] Common criteria, version 2.1, iso is 15408. `http://csrc.nist.gov/cc/ccv20/ccv2list.htm`.

[16] V. Cortier, J. Millen, and H. Ruess. Proving secrecy is easy enough. In *14th IEEE Computer Security Foundations Workshop*, 2001.

[17] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi. Formal automatic verification of authentication cryptographic protocols. In *1st IEEE International Conference on Formal Engineering Methods (ICFEM'97)*. IEEE, 1997.

[18] T. Dierks and C. Allen. The TLS protocol, IETF RFC 2246. `http://www.ietf.org/rfc/rfc2246.txt`, Jan. 1999.

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

[20] D. Dolev and A. C. Yao. On the security of pubic key protocols. *IEEE Transactions on Information Theory*, IT-29(2) :198–208, Mar. 1983.

[21] N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Proceedings of the FLOC Workshop on Formal Methods in Security Protocols*, 1999.

[22] N. El Kadhi. Automatic verification of confidentiality properties of cryptographic programs. *Networking and Information Systems*, 2001.

[23] C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. *Resolution Decision Procedures*, chapter 25, pages 1791–1849. Volume II of Robinson and Voronkov [48], 2001.

[24] T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *LICS'91*, 1991.

[25] T. Genet and F. Klay. Rewriting for cryptographic protocol verification (extended version). Technical report, CNET-France Telecom, 1999. `http://www.loria.fr/~genet/Publications/GenetKlay-RR99.ps`.

[26] J. Goubault-Larrecq. A method for automatic cryptographic protocol verification (extended abstract). In *Proceedings of the Workshop on Formal Methods in Parallel Programming, Theory and Applications (FMPPTA'2000)*, Lecture Notes in Computer Science LNCS 1800, pages 977–984. Springer Verlag, 2000. L'URL `http://www.dyade.fr/fr/actions/vip/jgl/cpv.ps.gz` pointe vers une version plus complète.

[27] J. Goubault-Larrecq. Higher-order positive set constraints. In *CSL'02*. Springer Verlag LNCS, 2002. Disponible en rapport de recherche LSV-02-6, Lab. Specification and Verification, ENS de Cachan, juillet 2002, `http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/rr-lsv-2002-6.rr.ps`.

[28] J. Goubault-Larrecq. Les syntaxes et la sémantique du langage de spécification EVA. Rapport technique EVA 3, LSV/CNRS UMR 8643 et ENS Cachan, July 2002. Version 6, projet RNTL EVA (Explication et Vérification Automatique de protocoles cryptographiques, `http://www.lsv.ens-cachan.fr/~goubault/EVA.html`).

[29] J. Goubault-Larrecq and K. N. Verma. Alternating two-way AC-tree automata. Research report, LSV, Sept. 2002. To appear. Submitted.

[30] J. Goubault-Larrecq and K. N. Verma. Alternating two-way AC-tree automata. Research Report LSV-02-1, LSV, January 2002.

[31] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In *Proceedings of the 7th Conference on Logic for Programming and Automated Reasoning (LPAR'2000)*, pages 131–160. Springer Verlag LNCS 1955, 2000.

[32] J. Larmouth. *Understanding OSI*, chapter 8. Available on the Web, 1994. `http://www.isi.salford.ac.uk//books/osi/chap8.html`.

[33] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, 1998.

[34] G. Lowe. An attack on the Needham-Schroeder public-key protocol. *Information Processing Letters*, 56(3) :131–133, 1995.

[35] W. Mao and C. Boyd. Towards formal analysis of security protocols. In *6th IEEE Computer Security Foundations Workshop*, pages 147–158, 1993.

[36] W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois, USA, 1994.

[37] C. A. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1) :5–36, 1992.

[38] J. Millen and G. Denker. CAPSL and muCAPSL. *Journal of Telecommunications and Information Technology*, 2002. To appear.

[39] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, 1978.

[40] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *6th International Static Analysis Symposium (SAS'99)*, pages 149–163. Springer-Verlag LNCS 1694, 1999.

[41] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – The CADE-13 Systems. *Journal of Automated Reasoning*, 18 :237–246, 1997.

[42] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12) :993–999, 1978.

[43] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1) :8–10, 1987.

[44] L. C. Paulson. Mechanized proofs for a recursive authentication protocol. In *10th IEEE Computer Security Foundations Workshop*, pages 84–95, 1997.

[45] L. C. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83, 1997.

[46] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6 :85–128, 1998.

[47] A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In *Proceedings of the first International Joint Conference on Automated Reasoning (IJCAR'01)*. Springer-Verlag LNAI 2083, 2001.

[48] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. North-Holland, 2001.

[49] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 174–190, 2001.

[50] B. Schneier. *Applied Cryptography : Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, 1994.

[51] P. Selinger. Models for an adversary-centric protocol logic. *Electronic Notes in Theoretical Computer Science*, 55(1) :73–87, July 2001. Proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification (LACPV'01), J. Goubault-Larrecq, ed.

[52] M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, 11(8) :769–780, 2000.

[53] S. Stoller. A reduction for automated verification of authentication protocols. In *Workshop on Formal Methods and Security Protocols*, 1999.

[54] S. Stoller. A bound on attacks on payment protocols. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, 2001.

[55] C. Weidenbach. Towards an automatic analysis of security protocols. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 378–382. Springer-Verlag LNAI 1632, 1999.

[56] C. Weidenbach. *Combining Superposition, Sorts and Splitting*, chapter 27, pages 1965–2013. Volume II of Robinson and Voronkov [48], 2001.

[57] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. SPASS version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*. Springer-Verlag LNAI 2392, 2002.