RAPPORT TECHNIQUE EVA

Langage de spécification de protocoles cryptographiques de EVA: syntaxe concrète

Date	:	19 novembre 2001
Auteurs	:	Florent Jacquemard (Trusted Logic) Daniel Le Métayer (Trusted Logic)
Titre	:	Langage de spécification de protocoles cryptographiques de EVA: syntaxe concrète
Rapport No. / Version	:	1/ 3.17

TRUSTED LOGIC S.A.
5 rue du Bailliage
78000 Versailles, France
www.trusted-logic.fr

Laboratoire Spécification Vérification
CNRS UMR 8643, ENS Cachan
61, avenue du président-Wilson
94235 Cachan Cedex, France
www.lsv.ens-cachan.fr

Laboratoire Verimag
CNRS UMR 5104,
Univ. Joseph Fourier, INPG
2 av. de Vignate,
38610 Gières, France
www-verimag.imag.fr

Résumé : Cette note présente le langage de spécification de protocoles cryptographiques poposé dans le cadre du projet EVA, en tenant compte en particulier du contexte d'application envisagé pour le projet. Chacun des choix réalisés est justifié sur des exemples. Le lecteur intéressé uniquement par une référence sur la syntaxe concrète du langage pourra consulter directement la grammaire en annexe A.

Notes : Compatibilité avec les autres documents et outils EVA.
Cette version 3.17 du rapport EVA 1 est compatible avec:

- le rapport EVA 2, version 2.3 (*Langage de spécification de protocoles cryptographiques de EVA: syntaxe abstraite et sémantique*);
- le rapport EVA 4, version 1.15 (*EVA test base*);
- le traducteur version 2;
- la base de test, version 2.

Le rapport EVA 3 (*Les syntaxes et la sémantique du langage de spécification EVA*) présente une extension du langage de spécification défini ici. Les changements ont été motivés d'une part par le développement du traducteur (version 3) et d'autre part par la nécessité d'adapter le langage aux méthodes de vérification de EVA. Le dernier point se traduit essentiellement par un gain d'expressivité (avec compatibilité ascendante) pour la spécification des environnements d'exécution et des propriétés à vérifier.

Table des matières

1 Contexte et objectifs

Ce document définit une syntaxe concrète pour la spécification de protocoles. Il s'agit d'une proposition réalisée dans le cadre du projet RNTL EVA. Le contexte d'étude est celui des protocoles d'authentification (en général à deux participants ou deux participants et un serveur de confiance pour la distribution de clés par exemple) et des protocoles de commerce électronique (généralement tripartites).

On se place dans l'optique de la définition d'une syntaxe abstraite [GL01] (qui a la forme de programmes définissant les actions des participants au protocole) pour laquelle est définie une sémantique opérationnelle simple, et d'une correspondance (c'est à dire une procédure de traduction) entre syntaxes concrète et abstraite. Un certain nombre de problèmes évoqués ici font références la traduction, comme les connaissances des participants et de l'intrus, les actions, ou des hypothèses plus générales.

Certains paragraphes présentent plusieurs alternatives illustrées par des exemples de protocoles. Dans ce cas, la solution retenue (et argumentée) est présentée sous le titre « **résumé** » en fin de paragraphe.

Deux critères complémentaires (qui sont en quelque sorte les deux plateaux de la balance) motivent ici les choix concernant syntaxe proposée.

1. La facilité d'utilisation pour le rédacteur de spécifications :
   - Expressivité. Le langage doit être adapté aux cas d'études. En particulier, il faut tenir compte de spécificités des protocoles de commerce électronique.
   - Lisibilité. On veut un langage proche de ceux utilisés (informellement) dans la littérature académique.
   - Précision. Les spécifications contiennent nécessairement une part implicite (concernant les actions des participants au protocole) mais elles seront le moins ambiguës possible. La traduction vers la syntaxe abstraite doit être suffisamment naturelle pour permettre au rédacteur d'une spécification de lever les ambiguïtés restantes. On peut aussi envisager que l'outil de traduction fournisse un retour (mode verbeux) suffisant pour donner à l'utilisateur une indication précise des actions de son protocole, et des hypothèses liées à (la traduction vers) la syntaxe abstraite et à la sémantique.

– Réutilisation. On essaiera de s'inspirer de langages existant, notamment de Capsl [DMR00], même si cette syntaxe n'est pas exactement celle qui est préconisée dans cette note.

2. La possibilité de vérification du protocole spécifié. On doit s'abstenir d'ajouter de constructions syntaxiques qui sortent du cadre des méthodes de vérification que nous emploierons (on pense en particulier à la spécification des propriétés à vérifier, cf. § 12). Plus la syntaxe est restreinte, plus grandes sont les possibilités de vérification automatique.

L'annexe A résume la syntaxe concrète d'une spécification. L'annexe B reprend en intégralité les différents exemples de protocoles utilisé dans le document et les exprime dans la syntaxe proposée.

2 Terminologie et hypothèses

Une session d'un protocole est une exécution de celui-ci par plusieurs acteurs. On fait l'hypothèse qu'une session n'est complétée que si elle se passe sans problème du point de vue de chaque participant. C'est à dire que si un participant reçoit un message non conforme à ce qu'il attendait, il interrompt la session.

Les principaux sont les rôles du protocole (serveur, client, banque etc). Il ne faut pas les confondre avec les *acteurs* qui jouent ces rôles respectifs dans des sessions du protocole. Pour désigner les acteurs, on parle aussi de participants. Par abus, on pourra parfois parler d'un principal pour désigner l'ensemble des acteurs jouant le rôle de ce principal (en particulier lorsqu'il sera question des connaissances de participants, § 8).

Le tableau suivant résume les différentes terminologies pour les variables du protocole (rôles etc) et leur interprétation à l'exécution.

Protocole	Exécution
spécification	session
rôle, principal	acteur, participant
autres variables (clés, nonces etc)	valeurs

L'intrus peut participer de manière non réglementaire au protocole. L'intrus désigne un acteur (il n'y pas de rôle d'intrus dans le protocole). Il est unique : un modèle avec plusieurs intrus qui collaborent (hypothèse la pire) est équivalent à un modèle avec un seul intrus. Dans le modèle que l'on considère [Bol96], [Bol97], l'intrus représente l'environnement hostile, c'est à dire l'ensemble des canaux de communication. Tout message émis passe nécessairement par lui et est transmis suivant son bon vouloir. Il a la possibilité de faire de la rétention de messages (diverting) ou bien d'en altérer le contenu (voir de forger un nouveau message à partir du contenu de plusieurs anciens) ou encore de changer le nom de l'émetteur ou du destinataire (mascarade) etc. Cependant, il ne peut attaquer les fonctions cryptographiques.

Les clés cryptographiques. On distinguera les clés symétriques des clés asymétriques, qui vont par paires. À chaque clé, on peut associer son symétrique, avec la propriété qu'un texte chiffré avec une clé K peut être déchiffré avec la clé symétrique de K (on suppose pour simplifier que l'on applique la même fonction pour chiffrer et pour déchiffrer). Le symétrique d'une clé symétrique K est K elle-même.

Variables de session. Certaines variables sont locales aux exécutions du protocole, c'est à dire qu'elles sont générées durant une session et n'ont d'existence que durant cette session. C'est le cas par exemple des clés de session, ou des nombres aléatoires envoyés pour un challenge (ou *nonces*=only once dans la littérature anglo-saxonne). On parlera ici de variables fraîches § 8.2.

Fonctions. On pourra avoir recours à des fonctions de hachage non inversibles (MD5, sha) pour les signatures, certificats, la vérification de l'intégrité de données etc. Des fonctions inversibles (opérateurs arithmétiques) pourront aussi être utilisés dans les spécifications de protocoles.

Autres hypothèses. D'autres hypothèses concernant les aspects sémantiques sortent du cadre de cette note mais seront clairement énoncées dès la conception du traducteur vers la syntaxe abstraite, cf. rapport [GL01]. Par exemple, il peut être important de savoir si les principaux connaissent le texte du protocole, c'est à dire la structure de tous les messages, ou bien seulement la structure des messages qui les concernent, ou bien encore seulement la structure de ce qu'ils seront à même de lire dans les messages...

3 Langages existants

Une dex premières tâches du projet EVA a été de réaliser une étude comparative des langages de spécification utilisés dans trois systèmes les plus connus (Casper [Low], Capsl [DMR00], Casrul [JRV00]). On confondra parfois un système et le langage de spécification qu'il utilise. On ne reprendra ici que quelques remarques générales sur ce comparatif. Dans la suite du document, chacun de nos choix se justifie par rapport à ceux réalisés dans ces trois systèmes.

Sémantiques : ces systèmes reposent sur des sémantiques opérationnelles définies par traduction dans les formalismes respectifs suivants :

Casper : traduction en algèbres de processus (CSP), vérification avec FDR (model checker).

Casrul : traduction en un système de réécriture, vérification avec DaTac (déduction en logique du premier ordre), Spike (preuves automatiques par récurrence) ou Elan (interprète de règles de logique de réécriture avec stratégies).

Capsl : traduction dans un langage intermédiaire CIL ($\approx$ réécriture) et vérification avec Maude (logique de réécriture) et PVS (preuves par récurrence).

Les algorithmes de traduction permettent de définir une sémantique de traces mais ils ne sont pas toujours faciles à appréhender (il peuvent même ne pas être publics comme dans le cas de Casper).

Notez que l'on obtient un modèle d'état finis pour Casper et un modèle d'états infinis pour Casrul et Capsl.

Motivations : Les langages Casper et Casrul ont été définis dans un souci pratique. Ils ont été conçus de pair avec un traducteur pour automatiser des expériences et rendre les résultats plus fiables. Pour Capsl, il y a en revanche une apparente volonté de définir un standard très complet, les traducteurs ont apparemment été écrit après.

Spécialisation : ces systèmes sont plutôt concernés par les protocoles d'authentification (du moins au vu des expériences qu'ils ont permis de réaliser à ce jour). Casper et Casrul sont plus efficaces pour la détection d'erreurs dans les protocoles que pour la certification.

Résumé. Pour résumer grossièrement, le langage proposé ici est un sur-ensemble de Casrul et un sous-ensemble de Casper ; grosso-modo, il s'agit de Casper sans les déclarations propres à CSP/FDR. Un des buts visés est de laisser une grande latitude à l'utilisateur dans la définition des opérateurs et types pour permettre de couvrir un grand nombre de cas. Capsl semble un peu trop général, ce qui pourrait en rendre complexe l'utilisation dans le cadre du projet EVA. Aussi, la conception d'un traducteur de (l'ensemble de la syntaxe) Capsl vers la syntaxe abstraite choisie pourrait être une tâche assez lourde…

4 Structure de la spécification

Dans la littérature, les protocoles de sécurité sont souvent présentés comme une suite de messages de la forme suivante (à l'étape d'étiquette i, S_i envoie à R_i le message M_i) :

$$i.\ S_i \rightarrow R_i :\ M_i$$

On reprend le même schéma et on s'attachera donc à définir une syntaxe concrète pour les message M_i. Des informations supplémentaires devront compléter cette suite dans la spécification formelle d'un protocole (dans la littérature, elles sont généralement exprimées informellement en français ou en anglais). Il s'agit en particulier des caractéristiques des variables (type, fraîcheur) et des connaissances que possède chaque participant au début d'une session.

La structure proposée pour la spécification de protocoles correspond grosso-modo à celle des langages des trois systèmes cités. Les sections composant le plan d'une spécification sont données ici dans l'ordre dans lequel elles devront apparaître. Elle sont introduites dans un autre ordre dans cette note, pour des raisons de présentation.

1. déclaration des variables (§ 6)
2. déclaration des connaissances initiales de chaque principal et de l'intrus (§ 8)
3. suite des messages (§ 5, § 9, § 10)
4. définition d'un environnement d'exécution (§ 7) (facultatif)
5. définition de propriétés à vérifier (§ 12)

Un environnement d'exécution est la donnée de domaines de valeurs pour les variables du protocole et des *sessions* possibles (à exécuter en parallèle), c'est à dire des différentes affectations de valeurs aux variables (en

particulier affectations de noms d'acteurs aux rôles). Dans Casper et Casrul, cette section permet d'introduire des informations propres à la méthode de vérification – par exemple des renseignements sur les abstractions ou approximations.

NB On n'aura pas nécessairement dans la spécification du protocole *toutes* les informations sur les actions des principaux. Il faudra faire la part de ce qui est sous-entendu, c'est à dire commun à tous les protocoles (et qui devra être clairement défini par la sémantique) et de ce qui peut être explicité dans la spécification ; cf. par exemple § 6.2.4, la remarque sur l'évolution des connaissances au début du § 8 et § 8.5 (sous-entendu) et la déclaration `keyPair` § 5.3.4, § 6.2.3, § 8.1, § 10 (explicite).

5 Messages

La section décrivant les messages échangés est le cœur de la spécification d'un protocole. Nous présentons ici une syntaxe concrète pour les messages.

Capsl permet (et même exige) la définition de tous les types et opérateurs utilisés dans des `typespec`'s. Une telle généralité n'est pas indispensable au projet EVA. On préfère une syntaxe plus concise qui autorise à la fois des types et opérateurs prédéfinis (pour la facilité d'utilisation) et des types, variables et fonctions choisis par l'utilisateur (pour l'expressivité). Dans les paragraphes suivants, on trouvera les solutions retenues pour EVA.

5.1 Les opérateurs indispensables

Les deux opérateurs de base suivant seront nécessaires pour la plupart des protocoles.

5.1.1 Paire

On utilise l'opérateur binaire associatif $\langle \_, \_ \rangle$ pour construire les paires. La notation variadique $\langle a_1, \ldots, a_n \rangle$ est permise et, par abus, les n-uplets de longueur un $\langle a \rangle$ et zéro $\langle\rangle$ sont aussi autorisés. Les parenthèses '$\langle$' et '$\rangle$' peuvent être omises dans les spécifications (comme c'est souvent le cas dans la littérature).

Avec les notations choisies, toutes les fonctions d'arités ≥ 1 sont considérées comme des fonctions unaires appliquées à un n-uplet. Il ne faut pas confondre les $\langle\rangle$ avec les () dénotant l'application de fonctions (cf. § 5.2.2).

5.1.2 Chiffrement

On note toutes les fonctions de chiffrement à l'aide de l'opérateur binaire $\{\_\}\_$ appliqué de la manière suivante : $\{texte\}_{clef}$. On a donc la même notation pour plusieurs fonctions de chiffrement. Le type de la clé peut permettre de faire la différence, par exemple entre chiffrement à clé symétrique et chiffrement à clé publique.

Pour les cas où l'on veut être encore plus explicite, un troisième argument est autorisé, ce qui donne la notation suivante : $\{texte\}^{algo}_{clef}$, où *algo* est une variable de fonction représentant le nom de l'algorithme utilisé pour le chiffrement, comme par exemple dans : $\{N_a\}^{3DES}_{K}$, pour un chiffrement avec triple DES.

5.2 Variables

Tous les identificateurs qui ne sont pas un des deux opérateurs prédéfinis § 5.1.1, § 5.1.2 sont des variables.

5.2.1 Variables de "premier ordre"

Ces variables représentent de simple valeurs comme des principaux ou des nombres (nonces). On parlera de *variables du "premier ordre"*. Elle sont généralement notées avec une majuscule, mais ce n'est pas une obligation.

5.2.2 Variables de fonctions

Certaines variables représentent des fonctions d'arité supérieure à zéro, (*variables de fonction*), comme des fonctions de hachage, ou aussi des tables de clés cf. § 5.3.4. Leur interprétation peut varier (inversible, injective etc) suivant les déclarations cf. § 6.2.

Dans un message, la valeur retournée par une fonction F appliquée à V_1,...,V_n est notée $F(V_1, \ldots, V_n)$. Les caractères $'('$ et $')'$ sont réservés à cette notation des fonctions, cf. § 5.1.1

On peut aussi trouver dans un message une variable de fonction seule (sans arguments), c'est à dire un nom de fonction, cf. § 5.4.

5.3 Clés cryptographiques

Pour la représentation et la déclaration des clés, nous nous inspirons de certaines constructions des trois systèmes considérés (avec une préférence ici pour Casper), comme par exemple la déclaration de paires de clés, et de travaux de Jean Goubault et Dominique Bolignano pour la définition de registres de clés publiques.

5.3.1 Clefs et clés brutes

Nous considérons une notation de Jean Goubault [GL00] elle même inspirée de travaux de Dominique Bolignano. On suppose fixé un ensemble $\mathscr{K}$ de *clés brutes*, infini et disjoint des autres domaines de données considérés, et on se donne des fonctions qui injectent les clés brutes dans le domaine des clés symétriques et asymétriques : $K : \mathscr{K} \to keys$, $PK : \mathscr{K} \to keys$, $SK : \mathscr{K} \to keys$. L'inverse de $K(a)$ est $K(a)$, l'inverse de $PK(a)$ est $SK(a)$, et l'inverse de $SK(a)$ est $PK(a)$ (pour une interprétation dans les protocoles à clés publiques, $PK(a)$ est la clé publique et $SK(a)$ la clé privée). L'avantage d'une telle méthode est qu'elle évite d'avoir à spécifier explicitement des axiomes tels que $(x^{-1})^{-1} = x$.

Rem. C'est la solution retenue pour la syntaxe abstraite [GL01]. Nous proposons ici des extensions pour une syntaxe concrète à destination de l'utilisateur, plus conforme aux critères décrits § 1.

5.3.2 Domaine de définition arbitraire

Les clés de session doivent parfois être construites à partir d'autres éléments, ce qui est impossible à modéliser avec les clés brutes.

Exemple 1 Le protocole suivant de Diffie-Helman permet à deux principaux A et B de négocier mutuellement une clé symétrique secrète K_{ab}, qui est construite à partir de parties des messages échangés. Plus précisément, la construction de cette clé repose sur deux entiers que les principaux ont préalablement publiquement négocié : un entier premier P et $G < P$, et sur des nombres X_a et X_b choisis aléatoirement pour la circonstance et tenu secret.

1. $A \to B : G^{X_a} \bmod P$
2. $B \to A : G^{X_b} \bmod P$

Les termes $G^{X_a} \bmod P$ et $G^{X_b} \bmod P$ (notés respectivement Y_a et Y_b) sont construits par application d'une fonction unaire $G^{\_} \bmod P$ respectivement à X_a et X_b.

Après l'exécution du protocole, A et B partagent $K_{ab} = Y_a^{X_b} \bmod P = Y_b^{X_a} \bmod P$. La clé K_{ab} est donc calculée à partir des nombres X_a et X_b par application des opérateurs $G^{\_} \bmod P$ (unaire) et $(\_)^{\_} \bmod P$ (binaire). Ce protocole garantit la confidentialité de la clé établie mais pas son authenticité. ◇

Une solution peut être de permettre des ensembles de définition pour les opérateurs K, PK et SK plus généraux que le simple domaine des clés brutes $\mathscr{K}$. On peut par exemple choisir les domaines des n-uplets construits avec $\langle \_, \_ \rangle$ comme domaine de définition de K, PK et SK (on rappelle que par abus de notation des n-uplets $\langle a \rangle$ de longueur 1 sont autorisés). Une autre possibilité est de laisser à l'utilisateur le choix de la définition du typage de K, PK et SK (cf. § 6.3 pour le typage).

Exemple 2 Avec une autre fonction binaire $(\_)^{\_} \bmod P$, la clé de Diffie-Helman de l'exemple 1 serait dans notre syntaxe : $K\left(\left(G^{X_a} \bmod P\right)^{X_b} \bmod P\right) = K\left(\left(G^{X_b} \bmod P\right)^{X_a} \bmod P\right)$. On remarque qu'un axiome est nécessaire (cf. § 11). ◇

5.3.3 Registre de clés publiques

Casrul définit un type spécial pour les registres de clés publiques, et un opérateur pour y accéder. Ici, la fonction PK peut être vue comme un opérateur d'accès à un registre de clés auquel on applique des noms de principaux (dans cette interprétation, $PK(A)$ est la clé publique de A, $SK(A)$ est la clé privée de A).

Exemple 3 Le protocole de Needham-Schroeder à clés publiques [NS78] s'écrit dans cette syntaxe :

1. $A \to S : A, B$
2. $S \to A : \{PK(B), B\}_{SK(S)}$
3. $A \to B : \{N_a, A\}_{PK(B)}$
4. $B \to S : B, A$
5. $S \to B : \{PK(A), A\}_{SK(S)}$
6. $B \to A : \{N_a, N_b\}_{PK(A)}$
7. $A \to B : \{N_b\}_{PK(B)}$

◇

5.3.4 Multiples registres de clé publiques

Il est commun dans les protocoles de commerce électronique de supposer l'existence de plusieurs paires clés publiques / privées pour chaque principal. Par exemple dans SET [MV96], les clients ont une clé publique dite d'échange et une clé publique de signature.

Une solution pourrait être de passer à la fonction *PK* une paire $\langle cst, nom\ de\ principal \rangle$ (au lieu du nom de principal seul) où *cst* est une constante qui peut prendre les valeurs *data* ou *exch* (ces constantes prédéfinies sont déclarées comme fonctions, cf. § 6.2.2). Mais cette solution n'est pas très pratique pour une syntaxe utilisateur.

Une autre solution est de ne pas se limiter aux fonctions prédéfinies *PK* et *SK* et d'autoriser l'utilisateur à en déclarer d'autres, comme *dataPK*, *sigSK* etc (§ 6.2 pour les déclarations de variables de fonction). La déclaration de ces fonctions de constructeur de clés se fait à l'aide de l'instruction suivante :

$$\texttt{keyPair}\ K_1, K_2\ :\ Type$$
$$\texttt{keyPair}\ F_1, F_2\ (Type_1, \ldots, Type_n) : Type$$

où les identificateurs K_1, K_2 et F_1, F_2 sont respectivement :

- deux variables "premier ordre" de type *Type* ; auquel cas K_1 est le symétrique de K_2 et réciproquement,
- deux variables de fonctions, de signature $Type_1, \ldots, Type_n \rightarrow Type$ auquel cas leur appliquer les mêmes arguments retourne des clés qui sont inverses l'une de l'autre.

On notera que dans les deux cas, la déclaration `keyPair` fait aussi office de déclaration des identificateurs K_1, K_2, F_1, F_2. C'est en fait exactement la solution de Casper.

Résumé. Les clés sont des variables du "premier ordre" arbitraires ou bien sont construites avec des variables de fonctions arbitraires. La déclaration de clés symétriques se fait comme pour les autres variables, la déclaration de clés asymétriques (publiques et privées) se fait à l'aide de l'instruction `keyPair`. D'après les exemples des paragraphes précédents, c'est une solution qui satisfait des critères (cf. § 1) de souplesse d'utilisation et d'expressivité.

5.4 Passage de fonctions

On peut se demander s'il est judicieux d'autoriser le passage de (variables représentant des) fonctions dans les messages. Il n'est en effet pas courant de passer dans un protocole un algorithme cryptographique complet. En revanche il peut arriver d'échanger des noms d'algorithmes cryptographique dans des protocoles.

Exemple 4 Le protocole de handshake de SSL [Com96] permet à un client et un serveur de se mettre d'accord (entre autres) sur un nom d'algorithme cryptographique, pour la suite du protocole et des connexions sécurisée ultérieures. Nous y reviendront dans les exemple qui suivent. ◇

Exemple 5 Revenons à l'exemple de Diffie-Helman. Comme nous l'avons vu les principaux *A* et *B* doivent dans une première étape se mettre (publiquement) d'accord sur *P* et *G*. Pour être complet, cette première étape doit aussi être spécifiée. Une solution peut être d'utiliser une variable de fonction non inversible kap à un seul argument, définie par $kap(x) = G^x \bmod P$. On considère que transmettre *P* et *G* revient à transmettre kap.

1. $A \rightarrow B : kap$
1. $B \rightarrow A :$
2. $A \rightarrow B : kap(X_a)$
3. $B \rightarrow A : kap(X_b)$

◇

Résumé. On autorise le passage de variables de fonctions dans les messages.

6 Déclarations de variables

Tous les identificateurs autres que les opérateurs prédéfinis (§ 5.1.1 et § 5.1.2) sont des variables et doivent être déclarés. En particulier, les messages du protocole font référence à des *rôles* de principaux, par opposition aux *acteurs* qui se substituent à ces rôles lors des exécutions du protocole. On aura différentes formes de déclarations pour les variables du "premier ordre" et les variables de fonctions.

Résumé. Toutes les variables de la spécification (du premier ordre ou de fonctions) doivent être déclarées.

6.1 Déclaration de variables du “premier ordre”

Les déclarations ont la forme :

$$\mathit{Variable} : \mathit{Type}$$

Exemple 6 Dans l'exemple 3 ci-dessus (Needham-Schroeder) on aurait dû faire par exemple les déclarations de variables suivantes (entre autres) :

$$\begin{array}{ll} A, B, S : & \mathit{principal} \\ N_a, N_b : & \mathit{number} \end{array}$$

◇

Nous reviendrons sur les types § 6.3.

Rem. Capsl permet d'assigner des attributs supplémentaires aux variables telles que “FRESH”. La solution proposée en § 8.2 rend inutile ce genre de déclarations.

6.2 Déclaration de variables de fonctions

La syntaxe doit permettre de déclarer, en plus des variables du “premier ordre” de nouveaux opérateurs arithmétiques et des fonctions non inversibles (fonctions de hachage). En particulier, comme nous l'avons vu en § 3 et § 5.3.4, les clés peuvent être définies à l'aide de fonctions.

6.2.1 Déclarations

Les déclaration de fonctions ont la forme :

$$\mathit{Variable\ de\ fonction}(\ \mathit{Type}_1, \ldots, \mathit{Type}_n) : \mathit{Type}$$

6.2.2 Constantes

Parmi les fonctions, on peut avoir aussi des constantes.

Exemple 7 Dans le protocole de handshake de SSL 3 [Com96], un message dit `ChangeCipherSpec` a la forme : $C \rightarrow S : \{1\}_{Ms}$ qui signifie que le client C envoie au serveur S la constante 1 chiffrée avec la nouvelle clé Ms (master secret) sur laquelle S et C viennent de se mettre d'accord.

Dans ce protocole, les noms d'algorithmes cryptographiques sont aussi des constantes, cf. § 9.2.2. ◇

La différence essentielle entre une variable de fonction constante déclarée par 1() : *number* et une variable du “premier ordre” déclarée par N_1 : *number* est que la seconde (contrairement à la première) peut être *fraîche* cf. § 8.2.

6.2.3 Déclarations d'inversibilité

Les trois langages mentionnés permettent tous de faire des déclaration sur l'inversibilité des fonctions.

Fonctions non inversibles Nous avons vu que certains opérateurs comme PK ou kap (exemple 5) doivent être interprétés comme des fonctions non inversibles. C'est le cas aussi des fonctions de hachage utilisées pour les signatures et certificats.

Exemple 8 Dans le protocole SSL 3 le serveur envoie au client un certificat X.509 de la forme :

$$S \rightarrow C : \ \mathit{Cert}, \{h_s(\mathit{Cert})\}_{SK(CA)}$$

Cert est un nuplet contenant un certain nombre de valeurs, particulier le nom de la fonction de hachage h_s, et le nom de l'autorité de certificat *CA*. Il y a donc deux parties dans ce certificat : le corps du certificat lui même *Cert* et une signature $\{h_s(\mathit{Cert})\}_{SK(CA)}$ obtenue en chiffrant avec la clé privée $SK(CA)$ de l'autorité de certificat *CA* un digeste $h_s(\mathit{Cert})$ de *Cert*, calculé avec la fonction non inversible h_s. On notera que le nom de la fonction h_s est passé en clair dans le certificat *Cert*. ◇

Fonctions inversibles Il est aussi nécessaire dans certains cas d'avoir des opérateurs interprétés comme des fonctions inversibles.

Exemple 9 À des fin d'authentification, un nonce N peut être envoyé (sous forme chiffrée) à un participant qui doit retourner (chiffré aussi) $N-1$, pour montrer qu'il possède bien la clé qui permet d'accéder à N. C'est un mécanisme qui est utilisé par exemple dans les deux derniers message du célèbre protocole de Needham-Schroeder à clés symétriques [NS78] (pour la distribution d'une nouvelle clé K_{ab} partagée par A et B par l'in-

termédiaire d'un serveur S).

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

Si le contenu du message 5 était $\{N_b\}_{K_{ab}}$, l'authenticité ne serait bien sûr pas garantie car n'importe quel intrus ayant pu lire le message 4 pourrait composer le message 5 sans avoir besoin de la clé K_{ab} pour le déchiffrer. D'autre part, après réception du message 5, B doit être capable de retrouver la valeur de N_b à partir de $N_b - 1$, pour vérifier que c'est bien le nonce qu'il a envoyé à A. En d'autres termes, l'opérateur de décrémentation doit être interprété (et déclaré) comme une fonction inversible. ◇

Pour spécifier qu'une variable de fonction est interprétée comme une fonction non inversible, on ajoute un mot clé `Hash` à la fin de sa déclaration. Les autres variables de fonctions sont inversibles.

Exemple 10 Déclaration de *kap* dans l'exemple 5, clé de Diffie-Helman :
kap(*number*) : *number* `Hash` ◇

Résumé. Par défaut, les variables de fonctions déclarées représentent des fonctions inversibles. Pour qu'elles représentent des fonctions non inversibles, on ajoute le mot clé `Hash` à la fin de la déclaration.

6.2.4 Définition de fonctions

Par défaut, les fonctions sont interprétées dans un univers de Herbrand : la valeur de $F(V_1, \ldots, V_n)$ est le terme $F(V_1, \ldots, V_n)$, ou plus exactement $f(a_1, \ldots, a_n)$ où f, a_1,...,a_n sont des symboles instanciant les variables respectives F, V_1,...,V_n.

Dans Casper, on peut donner une définition alternative à une fonction : (*vf* = *variable de fonction* et *vpo* = *variable premier ordre*)

$$vf(vpo_1, \ldots, vpo_n) = vpo$$

Rem 1. Cette solution pourrait permettre de décrire les mise à jour de registres de clés (de bases de données de manière générale). Elle peut donc être d'un grand intérêt pour l'étude de protocoles de commerce électronique (dans un contexte d'infrastructure de clés publiques avec liste de clés révoquées).

Exemple 11 Un serveur S crée à la demande une nouvelle clé publique K_a pour A et lui distribue en mettant à jour son registre PK, qui pourra être interrogé comme par exemple dans les messages 1 et 2 du protocole de Needham-Schroeder à clés publiques (Exemple 3).

A, S : *principal*
K_a, K_{as} : *key*
PK(*principal*) : *key* `Hash`
$PK(A) = K_a$
SK(*principal*) : *key* `Hash`
`keyPair`(PK, SK)
A `knows` $S, K_{as}, PK(A), SK(A)$
S `knows` K_{as}, PK
1. $A \rightarrow S : A$
2. $S \rightarrow A : \{K_a\}_{K_{as}}$

Les déclarations de connaissances sont définies § 8. Elle signifient ici qu'au démarrage d'une session, A connaît le nom de S, la clé K_{as} qu'il partage avec S, et ses propres clés publique et privée $PK(A)$, $SK(A)$; S connaît la clé partagée K_{as} et l'ensemble des clés publiques par l'intermédiaire de PK. La variable K_a est fraîche (car elle n'apparaît pas dans les déclarations de connaissances, cf. § 8.2) : elle est créée par le serveur S en cours de session. Grâce à la déclaration $PK(A) = K_a$, la valeur de $PK(A)$ sera mise à jour « automagiquement » à la création de K_a. Plus précisément, dans l'état des participants jouant les rôles de S et A, la valeur de $PK(A)$ est liée à la valeur de K_a (cf. § 8.3 pour la définition d'états des principaux). ◇

Résumé. Bien qu'il semble que cette syntaxe permette de spécifier des mises à jour de bases de données de clés publiques (par exemple), sa sémantique est un peu compliquée, en raison des liens entre les valeurs de clés dans les états locaux des participants. Pour cette raison, on ne retient pas cette notation dans la syntaxe concrète.

6.2.5 Déclarations `keyPair`

Les déclarations `keyPair` sont présentées au § 5.3.4.

6.3 Typage

Pour les types des opérateurs et variables, deux possibilités :
- des types fixés, et un typage fixé des opérateurs prédéfinis $\langle \_, \_ \rangle$, $\{\_\}\_$, $K(\_)$, $PK(\_)$, $SK(\_)\ldots$
- choix des noms de types (et du typage des opérateurs, y compris prédéfinis) par l'utilisateur pour chaque spécification.

6.3.1 Typage prédéfini

On pourrait avoir les types *principal*, *key* et aussi *number* pour les nonces et le contenu des messages en général (on considère que toute structure complexe est un nombre, y compris les textes clairs, les textes codés, les paires...), avec les profils de fonctions :

$$\begin{array}{lll} \langle \_, \_ \rangle : & number \times number & \rightarrow number \\ \{\_\}\_ : & number \times key & \rightarrow number \end{array}$$

6.3.2 Typage libre

Casper n'impose pas de types prédéfinis, à l'exception de *TimeStamp* et *HashFunction* qui ont des sémantiques particulières. L'utilisateur peut donc choisir n'importe quel nom pour les types dans ses spécifications, le système garantissant la vérification de type. La souplesse offerte par cette méthode permet de considérer les attaques basées sur des erreurs de type.

Exemple 12 Le protocole d'Otway-Rees [OR87] spécifie la création et distribution d'une clé symétrique K_{ab} partagée par les principaux A et B, par l'intermédiaire d'un serveur de confiance S. Sa spécification dans notre syntaxe, sans les déclarations de variables (et en anticipant sur les déclarations de connaissances initiales § 8.1) est la suivante :

A `knows` : B, K_{as}
B `knows` : K_{bs}
S `knows` : K_{as}, K_{bs}

1. $A \rightarrow B$: $N, A, B, \{N_a, N, A, B\}_{K_{as}}$
2. $B \rightarrow S$: $N, A, B, \{N_a, N, A, B\}_{K_{as}}, \{N_b, N, A, B\}_{aK_{bs}}$
3. $S \rightarrow B$: $N, \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}}$
4. $B \rightarrow A$: $N, \{N_a, K_{ab}\}_{K_{as}}$

L'attaque suivante est décrite dans [BAN89] :

1. $A \rightarrow I(B)$: $N, A, B, \{N_a, N, A, B\}_{K_{as}}$
4. $I(B) \rightarrow A$: $N, \{N_a, N, A, B\}_{K_{as}}$

Le message 1 à destination de B est détourné par l'intrus (notation $\rightarrow I(B)$) et l'intrus se fait passer pour B en envoyant le message 4 (notation $I(B) \rightarrow$). En recevant le message 4, A va considérer que le triplet $\langle N, A, B \rangle$ (qui est alors connu de l'intrus) est la nouvelle la clé K_{ab}. Cette attaque est possible s'il n'y a pas vérification de type ou si le type du triplet peut être confondu avec le type de la clé K_{ab} attendue par A.

C'est le cas avec les déclarations suivantes :

$$\begin{array}{ll} A, B, S : & principal \\ K_{as}, K_{bs}, K_{ab} : & number \\ N, N_a, N_b : & number \\ \langle number, number \rangle : & number \\ \{number\}_{number} : & number \end{array}$$

◇

Les déclarations ci-dessus reviennent presque à considérer un environnement non typé. On notera que Casper donne la possibilité à l'utilisateur de définir (par surcharge) le typage des opérateurs prédéfinis $\langle \_, \_ \rangle$ et $\{\_\}\_$ dans les déclarations de variables et fonctions.

Une autre option moins élégante pourrait être la possibilité d'une déclaration du genre `no typing` (mot réservé) à un endroit déterminé de la spécification.

6.3.3 Relations de sous-types

Dans Casrul, les erreurs de types sont caractérisées à l'aide de relations de sous-typage, comme $key \subseteq number$. Ce genre de déclarations pourrait compliquer la procédure de vérification et, suivant le critère 2, § 1, on leur préférera la solution précédente (§ 6.3.2).

Résumé. Pour permettre des certifications qui prennent en compte les attaques basées sur des erreurs de typage (expressivité), on retient la solution suivante pour le typage :
- il existe trois noms de types prédéfinis, qui ont une interprétation particulière (pour des raisons de correspondance avec la syntaxe abstraite), voir à ce sujet le rapport [GL01]. Il s'agit des types *principal*, *number* et *algo* (pour les noms d'algorithmes de chiffrement).
- l'utilisateur a le libre choix des autres noms de types (sans relations de sous-sortes) et du typage des opérateurs (variables de fonctions) autre que les deux opérateurs prédéfinis de paire et chiffrement.

– les deux opérateurs prédéfinis sont considérés comme prenant des arguments de n'importe quel type et retournant le type *number*.

7 Environnement d'exécution

Dans cette section facultative de la spécification, on définit des sessions sur lesquelles va porter la vérification, comme dans Casperet Casrul.

Une session est définie par les valeurs qui peuvent être affectées aux variables du "premier ordre" du protocole. Pour spécifier les sessions possibles, on écrit une suite d'affectations d'une valeur à chaque variable :

$$\texttt{session} \quad var_1 : val_1, \ldots, var_n : val_n$$

Les valeurs données $val_1, \ldots, val_n$ sont des valeurs symboliques. $var_1, \ldots, var_n$ sont des variables du premier ordre. On n'attribue bien sûr pas de valeurs aux variables fraîches. On n'en attribue pas non plus aux variables de fonctions, dont l'interprétation se fait (sauf avis contraire) dans un univers de Herbrand (voir à ce sujet la discussion § 6.2.4).

Exemple 13 Pour la célèbre attaque de Lowe [Low95] sur le protocole de Needham-Schroeder à clés publiques (exemple 3), on aura l'environnement suivant :

$$\begin{array}{llll} \texttt{session} & A = a, & B = I, & S = s \\ \texttt{session} & A = a, & B = b, & S = s \end{array}$$

Notez l'usage du mot clé I qui désigne l'(acteur) intrus. L'attaque (avec les deux sessions en parallèle) est la suivante (en oubliant les messages 1, 2, 4,5)) :

$$\begin{array}{llll} 3 & a & \rightarrow & I : \quad \{N_a, a\}_{PK(I)} \\ 3' & I(a) & \rightarrow & b : \quad \{N_a, a\}_{PK(a)} \\ 6' & b & \rightarrow & I(a) : \quad \{N_a, N_b\}_{PK(a)} \\ 6 & I & \rightarrow & I(a) : \quad \{N_a, N_b\}_{PK(a)} \\ 7 & a & \rightarrow & I : \quad \{N_b\}_{PK(I)} \\ 7' & I(a) & \rightarrow & b : \quad \{N_b\}_{PK(b)} \end{array}$$

Un terme $I(a)$ représente :

– à gauche de la flèche : une mascarade de l'identité de a par l'intrus,
– à droite de la flèche : un détournement par l'intrus d'un message destiné à a.

Il y a une erreur d'authenticité (l'intrus s'est fait passer pour a dans la session prime). ◇

La description des sessions possibles est surtout intéressante pour la recherche d'erreurs dans les protocoles (c'est une manière de tricher en réduisant l'espace de recherche de l'erreur). Il faut noter que la syntaxe adoptée ne permet qu'un nombre fini de valeurs pour les variables affectées. Si les variables fraîches sont interprétées dans un domaine fini (comme c'est le cas avec Casper) on aura un modèle d'états fini pour l'exécution du protocole. Si on a un mécanisme de production de nombres aléatoires (comme avec Casrul) le modèle sera infini, même en se restreignant aux sessions définies dans la spécification.

Résumé. On autorise l'ajout de déclarations `session` dans les spécifications pour définir le système à exécuter dans la mesure ou cela peut être utile à certaines méthodes de vérification. Si la méthode de vérification considère un nombre infini de sessions parallèles, de telles déclaration seront simplement ignorées. Par convention, l'absence de déclarations `session`, est équivalente à une seule déclaration implicitement définie par :

$$\texttt{session} \quad var_1 : val_1, \ldots, var_n : val_n$$

où $val_1, \ldots, val_n$ sont des valeurs par défaut distinctes associées aux variables du premier ordre $var_1, \ldots, var_n$. (On suppose que l'on a un mécanisme qui associe de manière unique une valeur à une variable de protocole donnée.)

Si l'on veut se donner la possibilité d'une infinité de valeurs pour certaines variables, il faudrait introduire un mécanisme adéquat, par exemple avec des types prédéfinis représentant des ensembles infinis disjoints. Ce n'est toutefois pas le cas dans cette syntaxe concrète.

8 Connaissances

Le traducteur de la syntaxe concrète vers la syntaxe abstraite (c'est à dire d'une spécification vers un programme définissant les actions des principaux) devra calculer une représentation de l'ensemble des connaissances de chaque principal à chaque étape du protocole. Cela est essentiel par exemple pour évaluer la visibilité des messages reçus et construire les instructions décrivant explicitement les actions des principaux entre la réception d'un message et l'envoi de la réponse (mémorisation de valeurs, etc), cf. [GL00],[JRV00]. De plus, cette fonctionnalité du traducteur permettra de détecter par effet de bord (au vol lors de traduction) et signaler

à l'utilisateur qu'un protocole est impossible à exécuter car une information à inclure dans un message est inaccessible au participant émetteur du message (cf. la propriété d'inter-blocage § 12.1.2).

Dans cette optique, il est nécessaire de spécifier pour chaque principal quelle sont ses connaissances au début de l'exécution d'un protocole.

8.1 Connaissances initiales des principaux

Elles sont données sous la forme de déclarations :

Variable_premier_ordre knows : $Term_1$,...,$Term_n$

La *Variable_premier_ordre* doit être un principal. Les *Term*'s peuvent être des variables du premier ordre ou des variables de fonctions déclarées (d'arité 0 ou plus) ou encore l'application d'une variable de fonction à des arguments. On supposera d'un principal qui ne figure en *variable premier ordre* d'aucune déclaration de connaissance qu'il ne connaît que son propre nom.

Exemple 14 Dans l'exemple 3 (Needham-Schroeder) on déclare les connaissances initiales suivantes :

$$A \text{ knows} : A, B, S, K_1(A), K_2(A)$$
$$B \text{ knows} : B, S, K_1(B), K_2(B)$$
$$S \text{ knows} : S, K_1, K_2(S)$$

Chaque principal *A*, *B* ou *S* connaît ses propres clés publiques et privées. En revanche, *A* ne connaît pas initialement la clé publique de *B* (et pas sa clé privée non plus bien sûr) et réciproquement. Le serveur *S*, lui, connaît les clés publiques de tout le monde, c'est à dire qu'il connaît la fonction d'accès K_1 au registre de clés publiques. On notera que *S* ne connaît pas a priori les noms des deux participants au protocole. ◇

8.2 Variables fraîches

Les variables dites *fraîches* sont celles qui font référence à des valeurs créées durant l'exécution du protocole et qui n'ont pour durée de vie que la durée de la session. C'est donc le cas des nonces et clés de session. On les définit comme les variables déclarées qui n'apparaissent dans aucune déclaration des connaissances initiales de principaux. Avec cette définition, il n'est pas nécessaire (comme c'est le cas avec Capsl) de faire des déclarations explicites pour ce type de variables.

Il semble raisonnable d'exiger que toutes les variables fraîches soient du premier ordre (c'est à dire qu'on n'y trouve pas de fonction). En effet, on n'envisage pas dans ce projet d'étudier des protocoles dans lesquels une fonction arbitraire serait générée (et transmise) au vol...

8.3 Évolution des connaissances

Durant l'exécution d'un protocole, les participants acquièrent des connaissances, en associant des *valeurs* à des variables du protocole. Par la suite, la liste d'association variable / valeur propre à un participant est appelée *état* du participant. Dans le cas où un participant prend part à plusieurs sessions simultanément, il y aura un état du participant pour chaque session. En effet, certaines variables, comme les nonces, peuvent prendre des valeurs distinctes (du point de vue d'un participant) dans différentes sessions. Au début d'une session, les états des participants sont initialisés suivant :

- les déclaration à la section connaissances initiales (§ 8.1) ;
- et les environnements d'exécution (§ 7) s'il sont spécifiés, ou bien des valeurs par défaut.

L'intrus aussi peut participer à plusieurs sessions mais à la différence des participants honnêtes, il maintient un seul ensemble global de valeurs (ou messages, c'est à dire valeurs composites) qu'il pourra utiliser et mettre à jour dans toutes les sessions auxquelles il participe. C'est de cette manière par exemple que l'intrus peut perpétrer des attaques de fraîcheur, où il réutilise une valeur (par exemple une clé de session) glanée dans une session antérieure. On appellera cet ensemble : *connaissances globales* de l'intrus, et on notera qu'il s'agit d'un simple ensemble de valeurs et non pas d'une liste d'association variable / valeur comme pour les participants.

En décrivant sommairement les connaissances locales des participants et de l'intrus, on suggère des aspects de sémantique opérationnelle – dans une note en principe consacrée à la définition d'une syntaxe concrète ; C'est en fait dans le seul but de justifier les choix syntaxiques des § 8.1, § 8.4, § 8.5. On notera que les aspects sémantiques mentionnés sont compatibles avec le rapport EVA [GL01].

8.4 Connaissances initiales de l'intrus

Il peut être important pour les outils de vérification d'avoir les connaissances initiales de l'intrus. D'après la forme des connaissances globales de l'intrus, les

connaissances initiales de l'intrus seront des valeurs et non pas des variables de protocole.

`Intruder knows` : $val_1,\ldots,val_n$

Pour les déclarations de domaines de valeurs, voir §7.

8.5 La syntaxe %

Comme cela a été souligné plus haut, on devra définir (pour les besoins de la traduction entre syntaxes) des règles de mise à jour des connaissances à partir des messages reçus. Ce sont elles en particulier qui détermineront ce qu'un principal peut lire dans un message (en fonction des clés qu'il connaît).

Les langages de Casper et Capsl donnent aussi la possibilité de définir explicitement dans la spécification du protocole ce qui est visible respectivement par l'émetteur et le récepteur d'un message donné. C'est la notation %, un message M étant noté :

"M vu par l'émetteur" % "M vu par le récepteur"

Exemple 15 Avec la notation %, le protocole de Needham-Schroeder à clés symétriques (exemple 9) devient :

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

L'idée est que dans le message 2, A ne peut lire le contenu du message codé $\{K_{ab}, A\}_{K_{bs}}$ (il ne connaît pas K_{bs}). Donc ce texte codé est vu par A comme une variable X, qu'il transmet à B au message 3. Mais B, à la réception du message 3, peut voir à l'intérieur de X. ◇

Abréviations La syntaxe % peut aussi être utilisée pour alléger les notations.

Exemple 16 Dans l'exemple 1, la notation % permet de remplacer la sous-expression $G^{X_b} \bmod P$ par la variable X_b dans l'expression de la clé de Diffie-Helman :

1. $A \rightarrow B : G^{X_a} \bmod P \% X_a$
2. $B \rightarrow A : G^{X_b} \bmod P \% X_b$
3. $A \rightarrow B : \{1\}_{Y_b^{X_a} \bmod P}$

Dans le message 3, A utilise la clé de Diffie-Helman associée à X_a, X_b pour chiffrer la constante 1 (cf. message le message ChangeCipherSpec de SSL 3 [Com96], exemple 7). ◇

Alias L'usage fait du % dans le paragraphe précédent peut paraître biaisé. On lui préférera la possibilité de définir des alias dans la section de déclaration de variables :

`alias` *variable premier ordre = expression*

Résumé. La syntaxe % est autorisée dans les spécifications. Elle est facultative, et une interprétation par défaut de la visibilité des messages par les participants est définie à partir des règles sur l'évolution des connaissances (cf. § 8 et [GL01]). Des alias peuvent être définis à l'aide du mot clé `alias`.

9 Contrôle, branchement

Certains protocoles définissent différents cas suivant, par exemple, la méthode cryptographique choisie par les participants.

Exemple 17 Dans le protocole de handshake de SSL [Com96], un client et un serveur commencent par se mettre d'accord sur un (nom d') algorithme cryptographique, puis ils établissent une clé symétrique partagée pour d'ultérieures sessions sécurisés. Si l'algorithme choisit est *rsa*, alors ils commenceront un sous-protocole P_{rsa} pour l'échange de la clé symétrique en utilisant leurs clés publique et privée et RSA. Si l'algorithme est *df* (Diffie-Helman), ils commenceront un autre sous-protocole P_{dh} avec des règles similaires à celle présentées dans les exemples 1, 5 et 7 (ChangeCipherSpec) etc. ◇

9.1 Sous-protocoles et précompilation

Dans Capsl, on donne une étiquette à tout protocole. Le branchement (inconditionnel) à un protocole P est fait par une instruction `include` P. Ce branchement se fait au début de P, et il n'y a pas de possibilité de retour vers le protocole appelant.

Résumé. Nous retenons cette solution dans le langage EVA. L'instruction `#include` *label* est destinée à être remplacée par les messages d'un protocole dont l'étiquette est *label*. Cette inclusion se fait lors d'une précompilation cf [GL01] ; par convention, les nom de l'étiquette d'un protocole est le nom du fichier contenant ce protocole.

9.2 Branchement conditionnel

9.2.1 Capsl : `if-then-else`

Le branchement vers un sous-protocole peut aussi être conditionnel en Capsl, avec un `if-then-else` entre deux messages, de la forme :

`if` *Condition*
`then include` label_1
`else include` label_2

La syntaxe des conditions est présentée au § 10.3.

Exemple 18 Dans l'exemple 17 sur SSL, après le choix de la méthode cryptographique *M*, on aura en Capsl les branchements suivants. La première ligne est le message dans lequel le serveur *S* propose son choix de méthode au client *C*.

⋮
5. $S \rightarrow C : M$
`if` $M = rsa$
`then include` L_{rsa}
`else if` $M = dh$
`then include` L_{dh}
`else if` $M = dms$
`then include` L_{dms}

Les identificateurs *rsa*, *dh* et *dms* sont déclarées comme des constantes (variables de fonctions sans arguments, cf. § 6.2.2). ◇

9.2.2 EVA : `switch-case` et précompilation

On choisit pour les branchements conditionnels une instruction :

`switch` *terme*
`case` $terme_1$ $rest_1$
⋮
`case` $terme_n$ $rest_n$

où $rest_1$,…,$rest_n$ sont des suites de messages de la forme $i : S \rightarrow R : M$. On peut combiner cette instruction de branchement avec une inclusion, c'est à dire avoir un $rest_i$ de la forme `#include` *label*, auquel cas la suite de messages suivie par le branchement sera celle contenue dans le fichier *label*.

Exemple 19 Pour SSL (exemple 17), on introduit un type *method* pour les trois méthodes possibles (déclarées comme variables de fonctions constantes) et pour la variable *M* de choix.

rsa() : *method*
dh() : *method*
dms() : *method*
M : *method*
⋮
5. $S \rightarrow C : M$
`switch` *M*
`case` *rsa*() : `#include` P_{rsa}
`case` *dh*() : `#include` P_{dh}
`case` *dms*() : `#include` P_{dms}

Le protocole se termine après ce `switch`. Cela correspond donc à la spécification de trois protocoles distincts dérivés de la spécification ci-dessus en remplaçant *M* par les constantes respectives de *method*'s (*rsa*, *dh*, *dms*) et en remplaçant le bloc `switch` par les sous-protocoles correspondant (d'étiquettes respective P_{rsa}, P_{dh}, P_{dms}). ◇

Résumé. On autorise des branchements conditionnels à des sous-protocoles dans la mesure où cette construction est nécessaire pour la spécification de certains protocoles comme SSL. On préfère une instruction de branchement de la forme `switch-case` au `if-then-else` de Capsl, pour éviter d'avoir à définir des expressions Booléennes.

10 Actions, conditions

Il est essentiel pour certifier les protocoles de définir précisément quelles sont les actions et tests qu'un principal exécute entre le moment où il reçoit un message et l'envoi de sa réponse. Cela comprend la lecture du message, la vérification que les éléments contenus sont en adéquation avec les valeurs précédentes de la session (par exemple pour les nonces), la vérification de signatures ou certificats, la récupération de nouveaux éléments (par exemple une clé) pour composer la réponse *etc*.

Les actions et tests sont souvent implicites dans la description d'un protocole par l'utilisateur. Ils doivent être rendus explicites dans la syntaxe abstraite et des règles génériques permettant de déduire les actions devront être implantées dans le traducteur. Le traducteur pourra fournir le résultats du calcul par ces règles en retour à l'utilisateur, ce qui permettra à ce dernier d'être sûr des actions que son protocole défini. On présente

dans les § 10.2 et 10.3 les notations pour les conditions et actions retenues dans le langage du projet EVA.

10.1 Capsl

Dans Capsl, des équations peuvent être ajoutées entre deux messages, de la forme :

$$X = e$$

où e est un terme construit avec des opérateurs (variables de fonctions) et des variables du premier ordre. Elles définissent, suivant le cas, ou bien un test ou bien une affectation effectuée par un principal.

1. $A \rightarrow B : M_1$
   $X = e$
2. $B \rightarrow C : M_2$

- Si B connaît la valeur de X, B teste si X==e. En cas d'échec, l'exécution est interrompue.
- Si B ne connaît pas X, B fait l'affectation (dans son état local) $X := e$.

Cette double interprétation d'un même opérateur = peut apparaître comme une source de confusion, et on lui préférera des opérateurs explicites := (action, affectation) et == (comparaison). On garde néanmoins le principe des affectations de variables dans les états locaux des participants (cf. § 8.3 pour la définition des états).

Exemple 20 Typiquement, dans SSL (exemple 17,) on ajoute une instruction pour que les participants mettent tour à tour à jour dans leur état local la valeur de M qui représente la méthode cryptographique adoptée. ◇

10.2 Actions

Pour les actions, on retiendra une syntaxe étendue de celle de Capsl :

1. $A \rightarrow B : M_1$
   $U : X := e$
2. $B \rightarrow C : M_2$

où U est ou bien A ou bien B. L'action associée est la mise à jour (:=) de X à e dans l'état de U (dans la session concernée).

Ainsi, les actions permettent de décrire des mises à jour des états de l'émetteur ou du récepteur d'un message. Plusieurs actions sont autorisées entre deux messages.

10.3 Conditions

Les conditions sont des comparaisons dont la syntaxe est similaire à celle des actions ci-dessus.

1. $A \rightarrow B : M_1$
   $U : X == e$
2. $B \rightarrow C : M_2$

où U est ou bien A ou bien B. Cette notation définit la comparaison (==) de la valeur de X dans l'état de U (dans la session courante) avec avec l'expression e', qui est obtenu de e par remplacement des variables du premier ordre par les valeurs correspondantes dans l'état de U. La comparaisons peut se faire modulo les équations éventuellement spécifiées (cf. § 11), suivant la sémantique adoptée.

À l'issue de la comparaison il y a ou bien interruption définitive de la session en cas d'échec ou bien la continuation du protocole (au message suivant).

Résumé. Pour les actions et les condition, on retient les notations définies aux § 10.2 et § 10.3.

11 Axiomes

L'introduction dans la spécification d'équations sur les fonctions utilisées peut compliquer la vérification. Cependant, il existe de nombreux exemples où certaines propriétés arithmétiques sont nécessaires à la description (et au déroulement) du protocole.

Exemple 21 Dans l'exemple 2 (Diffie-Helman), la propriété $\left(G^{X_a} \bmod P\right)^{X_b} \bmod P = \left(G^{X_b} \bmod P\right)^{X_a} \bmod P$ est nécessaire. ◇

Exemple 22 RSA

1. $A \rightarrow B : \{X\}_{PK(A)}$
2. $B \rightarrow A : \{\{\{X\}_{PK(A)}\}_{PK(B)}$
3. $A \rightarrow B : \{X\}_{PK(B)}$

Sans l'hypothèse $\{\{L\}_{K_a}\}_{K_b} = \{\{L\}_{K_b}\}_{K_a}$ A ne peut pas extraire $\{X\}_{PK(B)}$ de $\{\{X\}_{PK(A)}\}_{PK(B)}$. Cet axiome est donc nécessaire à l'exécution du protocole. ◇

De plus, des trous de sécurité de RSA proviennent de certaines propriétés algébriques de la fonction de chiffrement (telles que l'équation ci-dessus), et il semble

donc important d'inclure ces équations dans la spécification pour la vérification formelle de tels protocoles.

Résumé. On autorise l'ajout d'axiomes (sur les propriétés des opérateurs) dans les spécifications, sous la forme d'équations :

`axiom` $s = t$

où s et t sont des termes construits avec des variables de fonction, des variables du premier ordre et des variables muettes (implicitement quantifiées universellement). Il est à noter que ces axiomes ne sont pas pris en compte dans la sémantique définie en [GL01], ils sont spécifiés pour information et pour un usage éventuel dans les procédures de vérification.

12 Propriétés à vérifier

Il paraît judicieux d'inclure dans la spécification du protocole les propriétés à vérifier. En effet, cela permet de faire référence (dans la définition des propriétés) aux variables du protocole, ou même aux indices (étiquettes) de messages, ce qui est nécessaire dans de nombreux cas. De plus, une propriété incluse dans la spécification sera automatiquement traduite en une formule (dans le formalisme des outils de vérification utilisés) pour être vérifiée, ce qui permet de gagner du temps et d'éviter des erreurs par rapport à la méthode qui consiste à écrire les formules «à la main».

Un troisième argument en faveur de l'ajout dans les spécifications de propriétés est que ces dernières peuvent influencer (dans une certaine mesure) la forme du programme abstrait associé (par traduction) à la spécification. Par exemple, comme nous l'avons vu, les systèmes Casrul[JRV00], Capsl[DMR00]), traduisent une spécification de protocole en syntaxe concrète vers un système de réécriture (syntaxe abstraite). La forme du système de réécriture n'est pas exactement la même suivant que l'on veut vérifier une propriété de confidentialité ou une propriété d'authenticité. En effet, dans le dernier cas, certaines constructions (avec marquage, flags etc) sont ajoutées au système de réécriture.

Cependant, il ne faut pas perdre de vue que la spécification des propriétés à vérifier dans le projet EVA nécessite la définition d'une syntaxe concrète spécifique plus riche que celles de Casper, Capsl, Casrul. En effet, les propriétés de protocoles de commerce électronique sont à la fois plus variées et plus complexes que celles de protocoles d'authentification, cf. [Bol99].

12.1 Propriétés temporelles

12.1.1 Sûreté et vivacité

À notre connaissance, la plupart des travaux théoriques se limitent à la vérification des propriétés de sûreté (non atteignabilité d'un état critique), par opposition aux propriétés de vivacité (par exemple le déni de service). On pourra envisager (selon la méthode de vérification choisie) de s'attaquer aussi à des propriété de vivacité. Dans ce cas, il faudra aussi introduire une syntaxe pour écrire des formules temporelles dans la spécification du protocole.

12.1.2 Blocage

Un protocole particulièrement mal conçu pourrait se révéler non exécutable par des participants honnêtes, même dans des conditions idéales (absence d'intrus). C'est par exemple le cas si un principal ne peut composer un message parce qu'un élément nécessaire lui est inconnu, ou bien parce qu'il n'a pas reçu cet élément ou bien parce qu'il l'a reçu sous forme chiffrée et n'a pas la clé pour le déchiffrer.

Comme nous l'avons souligné au § 8, il y aura analyse de l'évolution des connaissances des principaux lors de la phase de traduction de la spécification en syntaxe concrète vers une syntaxe abstraite (cf. [GL01]). Un éventuel inter-blocage dû à l'insuffisance de connaissances d'un principal pour composer un message qu'il doit envoyer sera détecté dans cette phase d'analyse, et signalé à l'utilisateur, terminant là la vérification.

12.2 Protocoles d'authentification

Les trois systèmes cités permettent (et même demandent) que soit attachée à une spécification une propriété de sécurité que l'on veut vérifier, parmi les suivantes (modulo variantes) :

- confidentialité
- authenticité

Les propriétés se présentent sous la forme d'un mot clé et d'une liste de variables du protocole.

La propriété spécifiée est traduite automatiquement dans le formalisme utilisé pour la vérification (cf. § 3). L'avantage de la traduction est d'éviter les erreurs de construction et de permettre la spécification du protocole *et* des objectifs de sécurité par un utilisateur non expert de la procédure de vérification.

Nous proposons ci-dessous des propriétés primitives de confidentialité et authenticité proches de celles des trois systèmes considérés. Une spécification pourra contenir une liste de ces propriétés, auquel cas la propriété vérifiée en sera la conjonction.

12.2.1 Confidentialité

- `Secret`$(A, V, [B_1, ..., B_n])$: la variable V est connue uniquement de A,B_1,...,B_n à tout moment de toute session.

L'expression "variable connue" fait référence ici aux états des principaux (cf. § 8.3 pour la notion d'état de principal). La variable V est dite connue d'un principal A (dans une session) si dans l'état local de A (correspondant à cette session), une valeur est attribuée à V.

La variable V est dite connue de l'intrus (et dans ce cas la propriété `secret` est invalidée) si la valeur associée à S dans la session concernée est dans les connaissances globales de l'intrus (cf. § 8.3). Par définition, la valeur *associée* à V dans une session est définie comme suit :

- si V est fraîche (définition § 8.2), c'est la valeur associée à V dans l'état du (principal) créateur de la variable (et dans la session concernée) ;
- si V n'est pas fraîche, c'est la valeur v associée à V dans la déclaration "`session` $V : v \ldots$" (§ 7) correspondant à la session concernée, ou bien, s'il n'existe pas de telle déclaration, la valeur par défaut pour V.

Avec cette définition, on a bien au plus une valeur associée à chaque variable dans une session, bien qu'il soit possible d'avoir des valeurs distinctes associées à une même variable (dans une même session) dans les états respectifs de deux principaux.

Dans les descriptions ci-dessus (et dans le § 12.2.2) on confond, par abus de langage, le rôle A avec l'acteur qui joue ce rôle dans une session. Formellement, on aurait dû dire «quand l'acteur a jouant le rôle A dans une session termine cette session, la variable V est connue uniquement des acteurs jouant respectivement les rôles de A,B_1,...,B_n dans la session». On suppose aussi implicitement que ces acteurs ne sont pas l'intrus.

12.2.2 Authentification

- `Agreement`(A, B, V_a, V_b) quand B termine une session avec A, A a aussi terminé (avec B) et la valeur de V_a dans l'état final de A (dans cette session) est égale à la valeur de V_b dans l'état final de B (dans cette session).
- `Aliveness`(A, B)! quand B termine une session, A a aussi terminé.

Par terminaison d'une session du protocole, on entend terminaison dans des conditions normales, par opposition à l'abandon de l'un des participants suite à une anomalie constatée (par exemple à la suite d'un échec de comparaison, cf.§ 10.3). Ce dernier cas de terminaison exceptionnelle n'entre pas dans le cadre des propriétés ci-dessus.

Dans `Agreement`(A, B, V_a, V_b), A et B peuvent être identiques. On notera que les propriétés ci dessus ne sont pas symétriques. Pour exprimer l'authentification mutuelle, on doit donc utiliser (une conjonction de) deux primitives. Les deux propriétés `Agreement` et `Aliveness` sont invalidées si l'intrus prend la place de l'un des participant, disons A (à l'insu de l'autre participant B), car dans ce cas, A ne pourra terminer la session.

12.3 Protocoles de commerce électronique

Les propriétés du § 12.2 restent pertinentes pour les protocoles de commerce électronique. Mais certaines propriétés décrites dans [Bol99] sont propres aux protocoles tripartites de commerce électronique. Il est donc nécessaire d'étendre les langages de Casper, Casrul ou Capsl pour vérifier ce type de protocoles (ce qui est un objectif du projet EVA). Nous avons pour cela deux possibilités :

- une solution ad-hoc qui consiste à ajouter un certain nombre de primitives à la syntaxe (une primitive par propriété) (§ 12.3.3) ;
- une solution générale qui consiste à définir un langage de spécification de propriétés à la manière de [Bol99] (§ 12.3.4).

Exemple 23 On s'appuiera par la suite sur un exemple librement inspiré d'un protocole de paiement à la SET

tiré de [Bol99] :

C, M, G : *principal*
P, Od, N : *number*
hash(*number*) : *(number)*
C knows $C, M, P, Od, G, N, PK(C), SK(C), PK(M), PK(G)$
M knows $M, PK(M), SK(M), PK(C), PK(G)$
G knows $G, PK(G), SK(G)$

1. $C \rightarrow M$: C, M
2. $M \rightarrow C$: Id
3. $C \rightarrow M$: $\{Id, P, C, M\}_{SK(C)}$, $\{P, Od\}_{PK(M)}$, $\{P, N\}_{PK(G)}$
4. $M \rightarrow G$: $\{Id, P, C, M\}_{SK(C)}$, $\{Id, P, C, M\}_{SK(M)}$, $\{P, N\}_{PK(G)}$
5. $G \rightarrow M$: $\{hash(Id, P, C, M)\}_{SK(G)}$
6. $M \rightarrow C$: $\{hash(Id, P, C, M)\}_{SK(G)}$

Après initialisation de la transaction et attribution d'un identificateur de transaction unique *Id* (1 et 2) le client *C* passe en 3 une commande au marchand *M* pour un objet *Od* au prix *P* en transmettant ses coordonnées bancaires *N*. *M* se charge ensuite (4) de demander à la banque *G* (en fait l'intermédiaire ou *Gateway*) le prélèvement de la somme convenue pour l'achat. Si le prélèvement se passe bien, la banque confirme au marchand (5) qui transmet ensuite au client (6). ◇

12.3.1 Confidentialité

Exemple 24 Dans l'exemple 23, on veut exiger que la description *Od* de l'achat de *C* reste inconnue de *G* et d'un éventuel intrus :

$$\texttt{Secret}(C, Od, [M])$$

et aussi que les coordonnées bancaires *N* ne soient pas lisibles par le marchand *M* et par l'intrus :

$$\texttt{Secret}(C, N, [G])$$

◇

12.3.2 Authentification

Exemple 25 Dans l'exemple 23, *M* veut être assuré à la fin de la session de l'authenticité de *G* et de la somme débitée :

$$\texttt{Agreement}(M, G, P, P)$$

◇

12.3.3 Nouvelles primitives

Dominique Bolignano décrit en [Bol99] une différence essentielle entre les protocoles d'authentification et les protocoles de commerce électronique (concernant les objectifs de sûreté). Pour les premiers, une preuve d'authentification garantit à l'acteur *a* (jouant le rôle de *A*) terminant une session qu'il a bien communiqué avec l'acteur *b* (jouant le rôle de *B*), sous réserve que *b* est *honnête*, c'est à dire qu'il suit les règles du protocole pour l'envoi des messages. En d'autres termes, si *a* termine la session, il a la garantie que l'intrus n'a pas pris la place de *b* dans cette session. Dans le cas du commerce électronique, on peut vouloir des garanties qui sont valables même lorsqu'un acteur malhonnête participe au protocole. Cela vient de la complexité plus grande des protocoles de commerce électronique (on sort du schéma «deux utilisateurs plus un serveur de confiance») et de problèmes juridiques intrinsèques (non répudiation, responsabilité etc). Dans les paragraphes suivants, on propose de nouvelles primitives pour deux exemples de propriétés.

Responsabilité

Exemple 26 Dans l'exemple 23, la banque *G*, en recevant le message 4 (qui est la demande d'autorisation de prélèvement) veut la garantie de la *responsabilité* de *M* dans l'envoi du message, pour prévenir des contestations ultérieures. C'est à dire que *G* veut la certitude que d'une part c'est bien *M* qui a envoyé cette requête (et pas un intrus se faisant passer pour *M*) et d'autre part que le message n'a pas été altéré. De plus, la preuve doit être valable même dans le cas où *C* n'est pas honnête. ◇

Une primitive pour la propriété de l'exemple précédent peut prendre une forme assez simple, comme :

$$\texttt{Responsability}(i)$$

qui affirme qu'à la réception du message *i* (4 dans l'exemple 26) la responsabilité de l'émetteur (supposé) est assurée. Le plus difficile est bien sûr la définition sémantique de cette propriété (voir la dernière suggestion page 18).

Cohérence Une autre propriété du commerce électronique est de garantir la cohérence entre les valeurs échangées. ◇

Exemple 27 Plaçons nous dans le cas où les trois participants au protocole de l'exemple 23 sont honnêtes. On veut assurer les propriétés suivantes : à la réception du message 4, la paire envoyée $\{Id, P, C, M\}_{SK(C)}, \{Id, P, C, M\}_{SK(M)}$ correspond bien à une transaction qui a eu lieu entre C et M, c'est à dire que les valeurs pour les variables Id, P, C, M coïncident dans les états locaux de C, M, et G à cet instant (à la réception de 4). ◇

La propriété de l'exemple 27 peut s'exprimer comme une conjonction de la variante suivante de l'accord :

$$\texttt{Agreement}(i, A, B, V_a, V_b)$$

qui s'énonce : à l'issue de la réception du message i, la valeur de V_a dans l'état de A est égale à la valeur de V_b dans l'état de B.

Exemple 28 La propriété pour l'exemple 27 s'écrit :

$$\begin{array}{l}
\texttt{Agreement}(4, G, C, Id, Id) \\
\texttt{Agreement}(4, G, C, P, P) \\
\texttt{Agreement}(4, G, C, C, C) \\
\texttt{Agreement}(4, G, C, M, M) \\
\texttt{Agreement}(4, G, M, Id, Id) \\
\texttt{Agreement}(4, G, M, P, P) \\
\texttt{Agreement}(4, G, M, C, C) \\
\texttt{Agreement}(4, G, M, M, M)
\end{array}$$

Rappelons qu'une suite de propriétés primitives est à interpréter comme la conjonction de ces propriétés. ◇

12.3.4 Langage de définition de propriétés

Dans [Bol99], Dominique Bolignano propose une méthode générique pour décrire des propriétés de commerce électronique. Pour résumer grossièrement, une propriété se définit comme la donnée d'une fonction binaire dite *filtre* et d'un automate déterministe. Par définition, le protocole satisfait la propriété si toute trace d'exécution finie, après filtrage avec la fonction, appartient au langage reconnu par l'automate. Cette méthode permet de donner des définitions formelles concises de propriétés a priori compliquées, et sa généricité nous épargne la tâche d'avoir à classer les propriétés en catégories prédéfinies («confidentialité», «authentification», «non répudiation» etc) et d'avoir à écrire des définitions compliquées et conceptuellement très éloignées pour chaque catégorie (comme aux § 12.2.1, 12.2.2, 12.3.3).

Dans [Bol99], les définitions de fonctions filtres et automates font référence à des traits sémantiques (système états / transitions, les transitions étant étiquetées par des messages). Par conséquent, la conception d'une syntaxe concrète analogue à [Bol99] pour la définition de ces objets dans les spécification exige certaines adaptations. Il s'agit d'une tâche importante qui n'est pas développée dans cette note.

Une solution alternative pourrait être la définition d'une logique temporelle restreinte dont les formules pourraient se traduire en une fonction de filtre et un automate définissant les propriétés associées. Une logique temporelle linéaire limitée aux propriétés de sûreté devrait être suffisante pour définir toutes les propriétés qui nous intéressent.

Résumé. Nous proposons la solution hybride suivante :

- On définit la syntaxe concrète de nouvelles primitives (de propriétés) quand le besoin s'en fait ressentir (par exemple `Responsability` et `Agreement` § 12.3.3) ; on a donc une syntaxe permissive, la seule contrainte sur les propriétés est d'avoir la forme *id TermeListe*. L'avantage est est la facilité d'utilisation et la lisibilité, les propriétés étant des primitives de haut niveau.
- On donne une définition de chaque primitive à l'aide de la méthode de [Bol99]. On garantit ainsi une définition formelle en nous épargnant la tâche laborieuse de redéfinir complètement toute propriété (voir même d'avoir à changer la sémantique pour permettre la définition).

Cette solution est possible car la sémantique retenue pour le projet EVA, et présentée dans le rapport [GL01], est suffisamment proche de celle de [Bol99], et le formalisme de définition de propriétés de [Bol99] s'adapte à la procédure de vérification.

13 Conclusion

Nous avons présenté dans ce rapport une syntaxe concrète qui permettre de spécifier les protocoles cryptographiques étudiés dans le cadre du projet EVA. On trouvera en annexe A un récapitulatif complet de cette syntaxe et en annexe B quelques exemples de spécification de protocoles d'authentification connus dans cette syntaxe.

Le rapport EVA-2 [GL01] présente une syntaxe abstraite, sous la forme de programmes, pour la description

de protocoles et la sémantique opérationnelle associée. Une procédure de traduction de la syntaxe concrète vers la syntaxe abstraite présentée dans [GL01] permet d'étendre la sémantique au langage défini dans le présent rapport.

Références

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society*, 426(1871) :233–271, 1989.

[Bol96] Dominique Bolignano. An approach to the formal verification of cryptographic protocols. In *3rd ACM Conference on Computer and Communication Security*, 1996.

[Bol97] Dominique Bolignano. Towards a mechanization of cryptographic protocol verification. In Orna Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV'97)*, pages 131–142. Springer LNCS 1254, 1997.

[Bol99] Dominique Bolignano. Toward the formal verification of electronique commerce protocols. In *10th IEEE Computer Security Foundations Workshop*, 1999.

[Com96] Netscape Communications. Secure socket layer protocol. Internet Draft, march 1996. Available at `http ://home.netscape.com/eng/ssl3/ssl-toc.html`.

[DMR00] Grit Denker, Jonathan Millen, and Harald Ruess. The capsl integrated protocol environment. Technical report, SRI International, 2000.

[GL00] Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *Workshop on Formal Methods in Parallel Programming, Theory and Applications*, volume 1800 of *Lecture Notes in Computer Science*. Springer Verlag, may 2000.

[GL01] Jean Goubault-Larrecq. Langage de spécification de protocoles cryptographiques de eva : syntaxe abstraite et sémantique. Technical report, EVA, 2001.

[JRV00] Florent Jacquemard, Micahel Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In *7th International Conference on Logic for Programming and Automated Reasoning*, Lecture Notes in Artificial Intelligence. Springer Verlag, november 2000.

[Low] Gavin Lowe. Casper : A compiler for the analysis of security. Available at `http ://www.mcs.le.ac.uk/~glowe/Security/Casper/index.html`.

[Low95] Gavin Lowe. An attack on the needham-schroeder public key authentification protocol. *Information Processing Letters*, 1995.

[MV96] Mastercard and VISA. Secure Electronic Transactions specification. Available at `http ://www.setco.org/`, june 1996. (Books 1, 2, 3).

[NS78] Roger M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12) :993–999, 1978.

[OR87] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1) :8–10, 1987.

Annexes

A Grammaire concrète

La grammaire suivante résume l'ensemble des solutions préconisées dans les paragraphes intitulés « résumé ». Les non-terminaux sont écrits en *italique*, alors que les mots réservés sont en caractères "`typewriter`". Le mot vide est noté ' '. Les identificateurs *label*, *id*, *type-id*, *val* sont des chaînes de caractères quelconques.

Structure globale de la spécification.

Spécification := *label Déclarations* { *Messages* } *Environnement Propriétés*

Préambule.

Déclarations := *Déclaration* | *Déclaration Déclarations*
Déclaration := *Variables1ordre* : *Type*
| *VariableFonction*(*TypeList*) : *Type OptHash*
| `KeyPair` *VariableFonction*,*VariableFonction* (*TypeList*) : *Type*
| `KeyPair` *Variable1ordre*, *Variable1ordre* : *Type*
| *Connaissance* | *ConnaissanceIntrus*
| *Alias* | *Axiome*
Variables1ordre := *id* | *id* , *Variables1ordre*
VariableFonction := *id*
TypeList := ' ' | *Type TypeList*
Type := *type-id*
OptHash := ' ' | `Hash`
OptAlgo := ' ' | ^*id*
Connaissance := ' ' | *Variable1ordre* `knows` *Tuple*
Connaissance-intrus := ' ' | `Intruder knows` *Tuple*
Alias := `alias` *Variable1ordre* = *Term*
Axiome := `axiom` *Term* = *Term*

Termes.

Terme := *Tuple*
Tuple := *TermeAtomique* | *TermeAtomique* , *Tuple*
TermeAtomique := *VariableFonction TermeListe* | *Variable1ordre* | *VariableFonction*
| ( ) | (*Terme*)
| { *Terme* } *TermeAtomique OptAlgo*
| *TermeAtomique* % *TermeAtomique*
TermeListe := ( ) | (*Tuple*)

Messages du protocole.

Messages := *Message Messages* | *Message*
Message := *label Variable-1ordre* -> *Variable-1ordre* : *Terme* | *Condition* | *Action* | *Switch-case*
Condition := *Variable-1ordre* : *Variable-1ordre* == *Terme*
Action := *Variable-1ordre* : *Variable-1ordre* := *Terme*
Switch-case := `switch` *Terme* { *Cases* }
Cases := ' ' | `case` *Terme* : *messages Cases*
Cases := ' ' | `case` *Fonction-déclarée* #`include` *label Cases*

Système.

Environnement := ' ' | *Session Environnement*
Session := *label* `session` *Affectations*
Affectations := ' ' | *Variable-1ordre* `=` *val*`,` *Affectations*

Propriétés.

Propriétés := ' ' | *id TermeListe Propriétés*

Les couples varid *TermeListe* définit dans ce rapport sont :

`Secret(` *Variable-1ordre* `,` *Variable-1ordre* `,` [*Variable-1ordre*,...,*Variable-1ordre*])
`Agreement(` *Variable-1ordre* `,` *Variable-1ordre* `,` *Variable-1ordre* `,` *Variable-1ordre*)
`Responsability(` *int*)
`Agreement(` *int*`,` *Variable-1ordre* `,` *Variable-1ordre* `,` *Variable-1ordre* `,` *Variable-1ordre*)

B Exemples

On trouvera ci-dessous les spécifications, dans la syntaxe proposée, des protocoles évoqués dans cette note.

Diffie-Helman

A, B :
P, G, X_a, X_b :
1() :
$kap(number, number, number)$:
axiom $kap(P, kap(P, G, X_b), X_a) = kap(P, kap(P, G, X_a), X_b)$
A knows : A, B, kap
B knows : B, kap
{
1. $A \rightarrow B$: P, G
2. $A \rightarrow B$: $kap(P, G, X_a)$
3. $B \rightarrow A$: $kap(P, G, X_b)$
4. $A \rightarrow B$: $\{1\}_{kap(P,kap(P,G,X_b),X_a)\%kap(P,kap(P,G,X_a),X_b)}$

}

On rappelle que l'interprétation des fonctions est : $kap(p, g, x) = g^x \bmod p$ et $kas(p, x, y) = x^y \bmod p$. Par définition, P, G, X_a et X_b sont des variables fraîches. On a ajouté un dernier message où A envoie la constante (1) chiffrée avec la clé de Diffie-Helman nouvellement d'être négociée.

Needham-Schroeder à clés publiques

La dernière partie de cette spécification fait référence à l'attaque de Lowe (cf. Exemple 13).

A, B, S : *principal*
N_a, N_b : *number*
keyPair PK, SK(*principal*) : *key*
A knows : $A, B, S, PK(A), SK(A)$
B knows : $B, S, PK(B), SK(B)$
S knows : $S, PK, SK(S)$
Intruder knows : s, b
{ *number* Hash
1. $A \rightarrow S$: A, B
2. $S \rightarrow A$: $\{PK(B), B\}_{SK(S)}$
3. $A \rightarrow B$: $\{N_a, A\}_{PK(B)}$
4. $B \rightarrow S$: B, A
5. $S \rightarrow B$: $\{PK(A), A\}_{SK(S)}$
6. $B \rightarrow A$: $\{N_a, N_b\}_{PK(A)}$
7. $A \rightarrow B$: $\{N_b\}_{PK(B)}$

}
session $A = a, \quad B = I, \quad S = s$
session $A = a, \quad B = b, \quad S = s$
Agreement$(A, B, [N_a, N_b])$

Needham-Schroeder à clés symétriques

A, B, S : *principal*
K_{as}, K_{bs}, K_{ab} : *key*
N_a, N_b : *number*
A knows : B, S, K_{as}
B knows : K_{bs}
S knows : K_{as}, K_{bs}
{
1. $A \rightarrow S$: A, B, N_a
2. $S \rightarrow A$: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3. $A \rightarrow B$: $\{K_{ab}, A\}_{K_{bs}}$
4. $B \rightarrow A$: $\{N_b\}_{K_{ab}}$
5. $A \rightarrow B$: $\{N_b - 1\}_{K_{ab}}$

}

Otway-Rees

Spécification pour détecter une attaque par erreur de typage, Exemple 12.

A, B, S : *principal*
K_{as}, K_{bs}, K_{ab} : *number*
N, N_a, N_b : *number*
A knows : A, B, K_{as}
B knows : B, S, K_{bs}
S knows : S, K_{as}, K_{bs}
{
1. $A \rightarrow B$: $N, A, B, \{N_a, N, A, B\}_{K_{as}}$
2. $B \rightarrow S$: $N, A, B, \{N_a, N, A, B\}_{K_{as}}$, $\{N_b, N, A, B\}_{K_{bs}}$
3. $S \rightarrow B$: $N, \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}}$
4. $B \rightarrow A$: $N, \{N_a, K_{ab}\}_{K_{as}}$

}

Secret($A, K_{ab}, [B]$)