Aide à la conception et vérification de spécifications formelles de protocoles cryptographiques

Alix Trieu Amélie Royer Antoine Chatalic
Baptiste Tessiau Lucas Seguinot Siargey Kachanovich
prénom.nom@ens-rennes.fr

ÉNS Rennes - Université de Rennes 1

Résumé Les protocoles dits cryptographiques – c'est à dire reposant sur l'utilisation de primitives cryptographiques – sont très largement utilisés dans la vie courante, et en particulier dans de nombreux domaines critiques. L'utilisation d'outils de vérification automatique permet de prouver certaines propriétés de sécurité sur de tels protocoles. Cependant, exprimer ces propriétés de manière formelle n'est pas toujours aisé et lesdits outils ne sont pour la plupart pas faciles d'utilisation. Ce rapport présente certains de ces programmes de vérification, expose les principales difficultés rencontrées lors de l'écriture de spécifications formelles et émet plusieurs suggestions en vue du développement d'un ensemble de modules facilitant la conception et la vérification de protocoles cryptographiques.

Introduction

Le bon déroulement d'une communication entre plusieurs agents repose sur la formalisation commune que constitue le protocole utilisé ; celui-ci décrit l'ordre selon lequel les messages peuvent êtres envoyés par les différents participants, ainsi que la structure des messages échangés. Un protocole qui utilise des primitives cryptographiques – comme des fonctions de chiffrement ou de hachage – pour sécuriser la communication est lui-même dit cryptographique. De très nombreux domaines, parfois critiques, font appel à de tels protocoles ; nous en utilisons par exemple lors de chaque paiement par carte bancaire.

Il est le plus souvent nécessaire d'assurer certaines propriétés de sécurité sur les protocoles conçus – par exemple, assurer qu'un message sensible ne peut pas être lu par quelqu'un d'autre que la personne à qui il est destiné. De nombreux outils de vérification ont été conçus dans ce but ; l'enjeu est de rendre automatique la preuve de telles propriétés, et d'exhiber des traces d'attaques lorsqu'une propriété est mise en défaut. Cependant, la plupart de ces outils sont incomplets et ne sont pas simples d'utilisation. Ce travail constitue le résultat d'une étude préliminaire au développement de tels outils en réponse à ce problème.

Nous ne nous intéresserons pas aux opérations arithmétiques utilisées pour chiffrer les messages – algorithme RSA par exemple –, mais nous contenterons d'utiliser les propriétés générales établies autour de ces méthodes de chiffrement. Il existe des modèles de chiffrement asymétriques et symétriques. Dans le premier cas, chaque agent du protocole doit posséder deux clés. L'une est dite publique car elle est connue de tous les participants du protocole ; nous la noterons `pkey` (*public key*). L'autre est dite secrète, car connue seulement par son propriétaire, et est notée `skey` (*secret key*). Un message crypté à l'aide d'une clé publique (resp. privée) peut être déchiffré à l'aide de la clé privée (resp. publique) associée. Dans un cadre de chiffrement symétrique, une clé est partagée par plusieurs agents (généralement 2) ; nous noterons `symk(A, B)` une telle clé partagée par les agents `A` et `B`. Un message peut alors être chiffré et déchiffré avec cette même clé. Un message `m` chiffré par une clé `s` sera noté $\{\texttt{m}\}_{\texttt{s}}$ par la suite.

Notons que la sécurité des modèles de chiffrement précédemment décrits repose le plus souvent sur la difficulté de résolution de problèmes mathématiques. Nous considèrerons par la suite qu'il est nécessaire pour déchiffrer un message de connaître la clé associée et nous nous intéresserons seulement aux propriétés liées à la succession des envois de messages. Plus exactement nous travaillerons sous les 3 hypothèses suivantes, qui définissent le modèle de Dolev-Yao.

– Il est possible d'exécuter un nombre infini de sessions, c'est-à-dire d'exécutions distinctes du protocole avec différents agents, et ce de manière parallèle et/ou séquentielle. Il est souvent intéressant de considérer l'exécution paral-

lèle de plusieurs sessions car c'est souvent de là que viennent les attaques, l'intrus pouvant par exemple combiner des informations issues de deux sessions différentes et/ou les utiliser dans une troisième.
- L'intrus peut intercepter et modifier tous les messages transitant sur le réseau, choisir ou non de les laisser parvenir à leur destinataire ou encore les mémoriser pour une utilisation ultérieure – à l'exception des messages envoyés sur canaux privés, cf. section 2.3. Cela est parfois traduit par l'expression « l'intrus est le réseau ». Il connaît en particulier toutes les données qui transitent en clair, et peut également participer à n'importe quelle session d'un protocole, dans n'importe quel rôle. Les fonctions de génération de messages (génération de nonces, chiffrement, déchiffrement...) sont aussi à sa disposition.
- Les attaques sur la méthode de chiffrement sont impossibles ; cette hypothèse est dite « du chiffrement parfait ». Ainsi, pour déchiffrer un message, il est nécessaire de connaitre la clé de déchiffrement correspondante. Cela exclut en particulier les attaques de type « force brute », c'est-à-dire que l'intrus ne peut pas essayer de déchiffrer un message en utilisant successivement un grand nombre de clés, générées par exemple par énumération ou encore aléatoirement.

Notons que la connaissance initiale d'un intrus se limite aux clés publiques de tous les agents, à sa clé privée, aux identités des agents et aux éventuelles données initialement publiques supplémentaires propres au protocole.

D'un point de vue syntaxique, il est possible de décrire un protocole cryptographique sous la forme d'une séquence d'échanges du type « `A` envoie un message `m` à `B` » ; un tel échange est conventionnellement représenté `A` $\rightarrow$ `B` : `m` ; nous parlerons de formalisme « Alice&Bob ».

Nous détaillons différents types de propriétés de sécurité qui peuvent être définies sur un protocole et présentons plusieurs programmes de vérification automatique utilisés à ce jour. Nous proposons ensuite plusieurs pistes pour le développement de modules facilitant l'utilisation de ces outils, et plus généralement la conception de protocoles cryptographiques, ce qui constituera notre objectif pour le second semestre.

1 Classes de propriétés applicables aux protocoles cryptographiques

Les propriétés existant sur les protocoles cryptographiques sont très nombreuses, et parfois très spécifiques. Nous nous contenterons d'étudier deux grandes classes de propriétés : nous distinguons essentiellement les propriétés dites « de secret », qui s'intéressent à l'ensemble des agents ayant connaissance d'un message, des propriétés « d'authentification » dont la vocation est d'assurer la cohérence de l'exécution du protocole avec les attentes de chacun des agents qui l'exécutent vis-à-vis des autres agents et des messages reçus. Dans chaque cas, il est possible de distinguer des propriétés locales et globales. Ces dernières doivent être vérifiées sur des exécutions complètes du protocole, tandis que les premières font référence à un échange bien particulier dans le protocole. Nous souhaitons le plus souvent assurer des propriétés globales sur un protocole, mais les propriétés locales ont leur utilité car elles permettent d'avoir une vision plus fine du fonctionnement du protocole : nous pouvons par exemple vouloir qu'un message soit secret jusqu'à un certain point, ce qui ne peut être exprimé qu'avec des propriétés locales. Certaines propriétés seront préférées à d'autres selon le protocole à l'étude. À titre d'exemple, un protocole d'authentification d'agents – c'est à dire conçu dans le but d'assurer à chacun des agents participant qu'il communique bien avec l'autre – fera naturellement appel à des propriétés d'authentification, tandis qu'un protocole visant à protéger une communication amène à écrire des propriétés de confidentialité.

Nous définissons par la suite les propriétés usuellement rencontrées et les appliquons sur des protocoles minimalistes, puis détaillons l'exemple classique du protocole d'authentification de Needham-Schroeder.

1.1 Propriétés de secret

La première propriété que l'on peut attendre d'un protocole cryptographique est de permettre la transmission d'un message sans que celui-ci ne soit intercepté par l'intrus. Cela se traduit au moyen de propriétés de secret, qui sont générales dans le sens où elles se réfèrent au secret d'un message à l'échelle de l'exécution du protocole, que nous distinguons ici de la propriété de confidentialité qui est relative à un échange de message précis du protocole.

Secret d'un message Un message est dit secret si l'intrus n'en a pas connaissance à la fin de l'exécu-

tion du protocole. Cette propriété est parfois appelée « secret faible », par opposition à la propriété dite de « secret fort » [1], qui repose sur la notion d'indistinguabilité : un message $\mathtt{m}$ est dit secret au sens fort si l'exécution du protocole jouée avec $\mathtt{m}$ est indistinguable pour l'intrus de toute autre exécution jouée avec un message $\mathtt{m}'$ différent. Cette propriété, qui est en réalité une propriété d'équivalence observationnelle entre sessions, est toutefois beaucoup plus complexe à démontrer ; dans la suite nous ferons référence à la propriété de secret faible lorsque nous parlerons de propriété de secret.

Une seconde variante qu'il nous paraît utile d'introduire est le secret restreint à un certain nombre d'agents : Ainsi un message est dit secret pour les agents $\mathtt{A}_1 \dots \mathtt{A}_n$ si et seulement si à la fin de toute session le message est connu des agents $\mathtt{A}_i$ et d'eux seuls. Cette propriété est donc plus précise que le secret faible car elle fait mention précise des agents. Elle peut entre autres être utilisée dans le cadre d'un protocole de vote (Helios [2] par exemple), dans la mesure où seul le votant doit connaitre le contenu de son vote.

Confidentialité d'un message La confidentialité d'un message est une propriété de secret plus restreinte, dans le sens où elle est liée à un échange précis dans le protocole ; nous pouvons aussi parler de secret local. Ainsi l'échange $\mathtt{A} \to \mathtt{B} : \mathtt{m}$ est dit confidentiel si $\mathtt{A}$ est certain que seul $\mathtt{B}$ recevra cette instance du message $\mathtt{m}$.

Exemples Nous présentons ici quelques exemples de protocoles très simples, permettant de bien faire la distinction entre les différentes notions introduites.

$$\mathtt{A} \to \mathtt{B} : \{\mathtt{m}\}_{\mathtt{pk(B)}}$$

est confidentiel par rapport à $\mathtt{m}$ (car seul $\mathtt{B}$ peut décrypter le message pour obtenir $\mathtt{m}$).

$$\mathtt{A} \to \mathtt{B} : \{\mathtt{m}\}_{\mathtt{sk(A)}}$$

n'est pas confidentiel par rapport à $\mathtt{m}$, car l'intrus peut intercepter le message, et, puisqu'il détient tous les clés publiques (en particulier celle de $\mathtt{A}$), le déchiffrer avec $\mathtt{pk(A)}$ pour obtenir $\mathtt{m}$.

$$\begin{aligned} \mathtt{A} &\to \mathtt{B} : \{\mathtt{m}\}_{\mathtt{symk(A,\ B)}} \\ \mathtt{A} &\to \mathtt{B} : \mathtt{symk(A,\ B)} \end{aligned}$$

La propriété de confidentialité du message $\mathtt{m}$ dans le premier échange est ici respectée (même si l'intrus intercepte $\{\mathtt{m}\}_{\mathtt{symk(A,\ B)}}$, il ne connaît pas la clé $\mathtt{symk(A,\ B)}$ et ne peut donc pas déchiffrer le message). Cependant, le message $\mathtt{m}$ n'est pas secret, car la clé $\mathtt{symk(A,\ B)}$ est ensuite échangée en clair : $\mathtt{I}$ peut donc l'intercepter, déchiffrer le message $\{\mathtt{m}\}_{\mathtt{symk(A,\ B)}}$ qu'il a appris précédemment et donc obtenir $\mathtt{m}$.

1.2 Propriétés d'authentification

Il est possible de définir des propriétés d'authentification portant aussi bien sur les messages que sur les agents. Celles-ci permettent selon le cas de confirmer à un agent l'origine du message qu'il reçoit ou bien l'identité de son interlocuteur.

Authentification de message Un message $\mathtt{m}$ est dit authentifié par $\mathtt{B}$ dans la communication $\mathtt{A} \to \mathtt{B} : \mathtt{m}$ si $\mathtt{B}$ peut s'assurer que le message $\mathtt{m}$ qu'il reçoit a été généré par $\mathtt{A}$. Il est également possible de parler d'intégrité du message ; en effet, cette propriété assure que le message $\mathtt{m}$ envoyé par $\mathtt{A}$ n'a pas été modifié jusqu'à sa réception par $\mathtt{B}$. Toutefois, $\mathtt{B}$ ne peut pas confirmer que cette instance de $\mathtt{m}$ vient de lui être envoyée par $\mathtt{A}$ directement.

Considérons le protocole suivant :

$$\mathtt{A} \to \mathtt{B} : \{\mathtt{m}\}_{\mathtt{pk(B)}}$$

Le message $\{\mathtt{m}\}_{\mathtt{pk(B)}}$ n'est pas authentifié par $\mathtt{B}$ puisque la clé $\mathtt{pk(B)}$ est connue de tous les agents et $\mathtt{m}$ est quelconque : le message aurait pu être créé par n'importe qui. En revanche, dans le protocole

$$\mathtt{A} \to \mathtt{B} : \{\mathtt{m}\}_{\mathtt{symk(A,\ B)}},$$

le message $\{\mathtt{m}\}_{\mathtt{symk(A,\ B)}}$ est bien authentifié par $\mathtt{B}$, car mis à part $\mathtt{B}$, seul $\mathtt{A}$ connaît la clé symétrique $\mathtt{symk(A,\ B)}$ et a pu chiffré le message.

Authentification d'agent locale La propriété suivante ne porte plus sur le message échangé mais sur l'agent qui l'envoie ; nous parlons alors d'authentification d'agent. Nous commençons ici par définir l'authentification d'agent locale, c'est-à-dire relative à un échange : $\mathtt{A}$ est ainsi dit authentifié par $\mathtt{B}$ dans la communication $\mathtt{A} \to \mathtt{B} : \mathtt{m}$ si $\mathtt{B}$ peut s'assurer que le message $\mathtt{m}$ lui a été directement envoyé par $\mathtt{A}$.

À titre d'exemple, dans le protocole

$$\mathtt{A} \to \mathtt{B} : \{\mathtt{m}\}_{\mathtt{symk(A,\ B)}},$$

l'agent $\mathtt{A}$ n'est pas authentifié par $\mathtt{B}$. En effet, l'intrus $\mathtt{I}$ peut exécuter deux sessions à la suite, en interceptant le message adressé à $\mathtt{B}$ puis en le réutilisant pour

se faire passer pour A. Nous noterons I(A) lorsque l'intrus I participe à une session du protocole avec un interlocuteur pensant que I est A. Cette attaque correspond à la trace suivante :

$$\begin{aligned} \mathtt{A} &\rightarrow \mathtt{I} : \{\mathtt{m}\}_{\mathtt{symk(A,\ B)}} \\ \mathtt{I(A)} &\rightarrow \mathtt{B} : \{\mathtt{m}\}_{\mathtt{symk(A,\ B)}} \end{aligned}$$

La réutilisation d'un message intercepté mène à désigner cela en termes d'attaque « par rejeu ».

Authentification d'agent globale Il est possible d'étendre la notion précédemment définie à une exécution entière d'un protocole. Cette nouvelle propriété permet par exemple de vérifier qu'aucun intrus n'usurpe l'identité de A au cours du protocole. L'agent A est donc dit authentifié par B au cours d'un protocole si lorsque B pense avoir fini une session du protocole avec A, alors A a effectivement démarré une session.

Prenons l'exemple du protocole suivant :

$$\begin{aligned} \mathtt{B} &\rightarrow \mathtt{A} : \{\mathtt{N_B}\}_{\mathtt{pk(A)}} \\ \mathtt{A} &\rightarrow \mathtt{B} : \{\mathtt{s}, \mathtt{N_B}\}_{\mathtt{pk(B)}} \end{aligned}$$

Notons tout d'abord l'apparition de la notion de nonces, souvent représentés comme ici par la lettre N : ceux-ci sont des nombres générés aléatoirement lors de chaque session. Ici l'agent A est authentifié par B. La sécurité de ce protocole vient du fait qu'un message intercepté dans une session donnée ne pourra pas être rejoué lors d'une session ultérieure, car les nonces auront alors changé. Ainsi, même si l'intrus peut apprendre certains messages au cours d'une session du protocole, ceux-ci seront inutilisables à la session suivante, et il n'aura alors pas assez d'informations pour prendre l'identité de A auprès de B.

Unification Les deux notions locales présentées précédemment (authentification de messages et authentification d'agents locale) sont très similaires. Cependant, l'authentification locale d'agents est plus forte dans le sens où elle implique l'authentification de messages. Plus précisément, pour l'exécution de la communication $\mathtt{A} \rightarrow \mathtt{B} : \mathtt{m}$ sur un nombre infini de sessions, l'authentification locale d'agent requiert une correspondance injective (un à un) entre les envois de message de A et les réceptions de B (cf. figure 1). Il est d'ailleurs possible trouver des références à ces propriétés sous le nom de « accord non injectif » et « accord injectif » [3].

1.3 Étude d'un exemple : le protocole de Needham-Schroeder

Le protocole de Needham-Schroeder asymétrique (cf. figure 2) vise à assurer l'authentification de deux agents sur tout le protocole. L'idée est la suivante : l'agent initiateur A envoie un nonce et son identité dans un message que seul B, le destinataire, pourra déchiffrer. À la réception, B prend connaissance du nonce $\mathtt{N_A}$ et le renvoie à A accompagné d'un nonce $\mathtt{N_B}$ qu'il a lui-même généré. En recevant $\mathtt{N_A}$, A identifie son nonce et renvoie alors $\mathtt{N_B}$ à B pour finaliser l'authentification.

$$\begin{aligned} \mathtt{A} &\rightarrow \mathtt{B} : \{\mathtt{N_A}, \mathtt{A}\}_{\mathtt{pk(B)}} \\ \mathtt{B} &\rightarrow \mathtt{A} : \{\mathtt{N_A}, \mathtt{N_B}\}_{\mathtt{pk(A)}} \\ \mathtt{A} &\rightarrow \mathtt{B} : \{\mathtt{N_B}\}_{\mathtt{pk(B)}} \end{aligned}$$

Figure 2. Protocole de Needham-Schroeder

Secret. Ici, le nonce $\mathtt{N_A}$ est secret (de même pour $\mathtt{N_B}$). En effet, ce nonce est généré par A et ne peut être obtenu qu'en déchiffrant le premier message échangé (resp. le second) avec la clé sk(B) (resp. sk(A)), qui est inconnue de l'intrus.

Authentification des messages

- $\mathtt{A} \rightarrow \mathtt{B} : \{\mathtt{N_A}, \mathtt{A}\}_{\mathtt{pk(B)}}$: ce message n'est pas authentifié, car initialement l'intrus connaît tous les éléments pour pouvoir générer un tel message (quitte à créer un nonce).
- $\mathtt{B} \rightarrow \mathtt{A} : \{\mathtt{N_A}, \mathtt{N_B}\}_{\mathtt{pk(A)}}$: le message est authentifié par A, *i.e.* A sait que B a créé ce message. En effet pour connaitre $\mathtt{N_A}$ il faut soit être A (qui l'a généré), soit posséder sk(B) (pour déchiffrer le message précédent), dont seul B a connaissance.
- $\mathtt{A} \rightarrow \mathtt{B} : \{\mathtt{N_B}\}_{\mathtt{pk(B)}}$ Pour des raisons similaires, ce message est authentifié par B.

Authentification des agents. Dix-sept ans après sa création, et malgré plusieurs preuves manuelles de correction, G.Lowe a exhibé une attaque du protocole sur l'authentification d'agent, qui est pourtant l'objectif principal de ce protocole [4] (cf. figure 3).

L'intrus peut en effet jouer deux sessions du protocole (l'une initiée par A, l'autre en tant qu'initiateur avec B) et entremêler les deux exécutions. Ainsi A envoie des messages chiffrés par pk(I) à I. Celui-ci

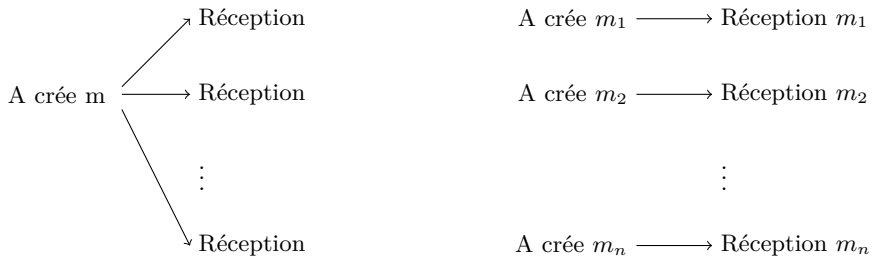

Figure 1. Authentification de messages (non-injective) et authentification locale d'agent (injective).

$$
\begin{array}{lll}
\mathtt{A} \to \mathtt{I} : \{\mathtt{N_A}, \mathtt{A}\}_{\mathtt{pk(I)}} & & \\
 & \mathtt{I(A)} \to \mathtt{B} : \{\mathtt{N_A}, \mathtt{A}\}_{\mathtt{pk(B)}} & \\
 & \mathtt{B} \to \mathtt{I(A)} : \{\mathtt{N_A}, \mathtt{N_B}\}_{\mathtt{pk(A)}} & \\
\mathtt{I} \to \mathtt{A} : \{\mathtt{N_A}, \mathtt{N_B}\}_{\mathtt{pk(A)}} & & \\
\mathtt{A} \to \mathtt{I} : \{\mathtt{N_B}\}_{\mathtt{pk(I)}} & & \\
 & \mathtt{I(A)} \to \mathtt{B} : \{\mathtt{N_B}\}_{\mathtt{pk(B)}} &
\end{array}
$$

Figure 3. Attaque sur Needham-Schroeder

peut donc obtenir le contenu en clair du message (car il connait `sk(I)`) puis le rechiffrer par `pk(B)` avant de l'envoyer à `B`, puisque la clé `pk(B)` est dans sa connaissance : l'attaque se fonde sur les rejeux des messages que `I` peut utiliser à sa guise d'une session à l'autre.

`A` est donc conscient d'interagir avec `I`, mais `B` pense jouer le protocole avec `A`, ce qui n'est pas le cas : l'intrus joue un rôle d'intermédiaire, ce que l'on désigne parfois en terme d'attaque de type « homme du milieu ». Une version corrigée du protocole est présentée en figure 4.

$$
\begin{array}{l}
\mathtt{A} \to \mathtt{B} : \{\mathtt{N_A}, \mathtt{A}\}_{\mathtt{pk(B)}} \\
\mathtt{B} \to \mathtt{A} : \{\mathtt{N_A}, \mathtt{N_B}, \mathtt{B}\}_{\mathtt{pk(A)}} \\
\mathtt{A} \to \mathtt{B} : \{\mathtt{N_B}\}_{\mathtt{pk(B)}}
\end{array}
$$

Figure 4. Protocole de Needham-Schroeder corrigé

L'ajout de l'identité de `B` combiné aux nonces $\mathtt{N_A}$ et $\mathtt{N_B}$ rend impossible tout rejeu du second message. De plus l'intrus n'a pas moyen de créer le message $\{\mathtt{N_A}, \mathtt{N_B}, \mathtt{I}\}_{\mathtt{pk(A)}}$, car il ne connaît jamais directement $\mathtt{N_A}$ et $\mathtt{N_B}$: l'attaque précédente n'est ainsi plus valable.

Notons que les outils de vérification automatique détectent très bien cette attaque sur le protocole de Needham-Schroeder, alors qu'il a été prouvé manuellement et jugé correct pendant de nombreuses années : il n'est en effet pas toujours simple de vérifier à la main les propriétés recherchées, car il faut envisager toutes les possibilités d'action de l'intrus sur un nombre théoriquement infini de sessions jouées, et modéliser sa connaissance qui évolue au cours du temps, lui permettant de décrypter de plus en plus de messages ; c'est pourquoi nous avons besoin d'outils de vérification performants.

2 Quelques outils de vérification

De nombreux outils sont disponibles à ce jour pour effectuer de la vérification automatique de protocoles cryptographiques. Notons qu'il existe également des formalismes particuliers qui permettent de prouver des protocoles cryptographiques avec d'autres outils de vérification, parfois même issus d'autres domaines (par exemple Isabelle dans le domaine de la preuve formelle [5] ou Spin pour le *Model Checking* [6]). Nous avons choisi de nous concentrer sur trois outils conçus spécialement pour la vérification de protocoles cryptographiques qui sont souvent utilisés dans ce domaine.

ProVerif [7] est un outil français développé principalement par Bruno Blanchet. Il est très complet mais sa prise en main n'est pas évidente. Sa sortie est en effet assez verbeuse et l'expression correcte des propriétés recherchées n'est pas toujours aisée. De plus, il est nécessaire de définir manuellement la plupart des primitives cryptographiques désirées.

Scyther [8][9] est un outil suisse créé par Cas Cremers à visée principalement pédagogique. Depuis quelques temps, son développeur travaille sur un autre projet, Tamarin [10], mais celui-ci se rapproche plutôt d'un assistant de preuves (tels que Coq ou Isabelle).

Avantssar [11] est un outil européen qui s'inscrit dans une suite assez importante de logiciels destinés à la vérification de protocoles cryptographiques. Il est assez proche de ProVerif, avec une entrée et une sortie plus claires mais moins d'expressivité.

2.1 Critères d'évaluation

Afin de comparer les trois outils introduits ci-dessus, nous avons retenu principalement les critères d'évaluation suivants :

1. **Langage d'entrée** : chacun de ces outils possède un langage qui lui est propre. Certains de ces langages se veulent simples et proches du langage Alice&Bob que nous utilisons pour décrire les protocoles, tandis que d'autres sont bien plus complexes à prendre en main.
2. **Expressivité** : certains protocoles utilisent des théories équationnelles ou encore des canaux privés, qui ne sont pas modélisables avec tous les outils, comme le protocole de Diffie-Hellman qui utilise les propriétés de l'exponentiation par exemple ; ces derniers ne sont donc pas tous aussi expressifs. Ce critère étant assez général, il sera développé à la sous-section 2.3.
3. **Compréhension de la sortie** : chacun des outils possède une présentation des résultats différente : certains peuvent représenter les attaques sous forme de graphe, tandis que d'autres le font sous forme d'une trace d'attaque plus ou moins compréhensible.
4. **Propriétés** : nous avons implémenté une série de petits protocoles pour observer comment fonctionne l'aspect vérification de chacun de ces outils. Il est apparu qu'ils ne permettent pas tous de vérifier les propriétés qui nous intéressent.
5. **Documentation** : nous avons besoin de bien comprendre les mécanismes concernant la vérification de propriétés pour la suite. Nous avons donc pris en compte la documentation existante sur l'outil.
6. **Licence** : étant donné que nous serons peut-être amenés à modifier le code des outils, il est important de vérifier leur licence.

2.2 Aperçu

Nous présentons ici de manière succincte un aperçu du format d'entrée de chaque outil et la manière de représenter les propriétés que nous souhaitons vérifier sur le protocole.

ProVerif. Nous nous intéressons dans un premier temps à la manière de représenter un échange `A → B : {m}pk(B)`. En ProVerif, chaque rôle est d'abord défini comme une fonction. Le protocole est ensuite démarré à la fin du fichier : concrètement il s'agit de créer les connaissances initiales des agents (clés publiques, identités ...) puis d'appeler les fonctions de rôles correspondantes. Le « `!` » devant les appels de fonctions dénote l'exécution d'une infinité de sessions en parallèle ; le « `|` » correspond à l'exécution de deux fonctions (rôles) simultanément. Les envois (respectivement réceptions) de messages correspondent aux instructions `out` (respectivement `in`). Notons que lorsqu'un message est envoyé, il n'y a pas mention particulière des agents destinataires, il est simplement envoyé sur un canal `c` et tout agent ayant accès à ce canal peut récupérer le message.

```
        (* Role de A *)
let process_A(pkB : pkey, pkA:pkey) =
        new message:bitstring;
        out(c, aenc(message, pkB)).

        (* Role de B *)
let process_B(pkB:pkey, sKB:skey, pkA:pkey) =
        in(c, m:bitstring).

        (** Principal **)
process
new skB : skey; let pkB = pk(skB) in
out(c, pkB);
new skA : skey; let pkA = pk(skA) in
out(c, pkA);
(!process_A(pkB, pkA)|(!process_B(pkB,skB, pkA))
```

Figure 5. `A → B : {m}pk(B)` en ProVerif

Notons que le code ici donné est incomplet. Au dessus de ces définitions se trouvent normalement une dizaine de lignes qui permettent de définir les fonctions `pk` (qui associe une clé publique à chaque clé secrète), `aenc` (chiffrement asymétrique) et `adec` (déchiffrement) : il est nécessaire de redéfinir ces fonctions utiles à chaque nouveau fichier.

Enfin, ProVerif permet de définir explicitement la connaissance initiale de l'intrus : À la fin du protocole sont présents deux évènements `out(c, pkB)` et `out(c, pkA)` qui n'apparaissent pas dans la spécification Alice&Bob d'origine ; ces envois de messages sur un canal public permettent à l'intrus de prendre connaissance des clés `pkA` et `B`.

Représenter les propriétés. La propriété la plus simple à vérifier avec ProVerif est le secret d'une donnée : si `m` est un message du protocole dont nous voulons vérifier le secret, alors nous utilisons le mot clé `query attacker m`.

La vérification des propriétés d'authentification est plus complexe et repose sur l'utilisation d'événements. Ceux-ci sont des instructions n'ayant pas d'effet et qui peuvent être paramétrés par des données du protocole ; ils font office en quelque sorte de points de contrôle dans le code. Il est alors possible de vérifier des propriétés de causalité sur ces événements, et c'est cela qui permet de vérifier les propriétés d'authentification présentées en section 1.

Prenons le protocole $\texttt{A} \rightarrow \texttt{B} : \{\texttt{m}\}_{\texttt{pk(B)}}$ pour exemple, et supposons vouloir montrer que `B` authentifie `A` (propriété d'authnetification d'agent globale), c'est-à-dire que pour tout événement de réception du message de `B` il existe un et seul événement d'envoi du message par `A`. Cela se traduit très bien en termes d'événements, comme le montre la figure 6.

```
          (* Vérifier l'authentification *)
event send_message.
event receive_message.
query inj-event(receive_message)
        ==> inj-event(send_message)

        (* Role de A *)
let process_A(pkB : pkey, pkA:pkey) =
        event send_message;
        new message:bitstring;
        out(c, aenc(message, pkB)).

        (* Role de B *)
let process_B(pkB:pkey, sKB:skey, pkA:pkey) =
        in(c, m:bitstring);
        event receive_message.
```

Figure 6. Authentification d'agent en ProVerif

Avantssar est composé de différents outils de vérification (SATMC, OFMC, CL-Atse) utilisant un langage de spécification commun : Aslan++. Dans ce langage, une session est paramétrée par des agents dont elle définit les rôles. Chaque rôle est défini comme une entité qui contient des déclarations et le "corps" (c'est-à-dire les actions effectuées par l'agent qui remplit ce rôle). La syntaxe utilisée pour définir les rôles est très semblable à la notation Alice&Bob (cf. figure 7).

Représenter les propriétés. Les propriétés sont définies en ajoutant des étiquettes aux messages du protocole, et en indiquant les propriétés attendues

```
entity Session (A, B: agent) {
 entity A(Actor, B: agent) {
     symbols
      m: text;
     body
    { m := fresh();
      Actor -> B : {m}_pk(B); }
 }

 entity B(A, Actor: agent) {
     symbols
     m : text;
     body
    { A -> Actor : {m}_pk(Actor);}
 }

  body
 { new A(A,B);
   new B(A,B);}
}
```

Figure 7. $\texttt{A} \rightarrow \texttt{B} : \{\texttt{m}\}_{\texttt{pk(B)}}$ en Aslan++

sur les messages étiquetés. Par exemple, pour vérifier la propriété de secret du message `m` dans le protocole donné en figure 7, nous ajoutons une étiquette `secret_m` au message `m` en remplaçant la ligne `m := fresh()` par `secret_m:(m) := fresh()`. Nous ajoutons ensuite dans le code une section `goals` contenant `secret_m:(_) {A,B}`, indiquant que les messages étiquetés par `secret_m` doivent être connus uniquement par les agents `A` et `B`.

Les propriétés d'authentification peuvent être décrites de la même manière en étiquettant les messages et en complétant la section `goals`.

Scyther permet lui aussi de définir un protocole en fonction des rôles de ses participants.

```
protocol protocol1(I,R) {
   role I
  { fresh m: Nonce;
    send_1(I,R,{m}pk(B)); }

   role R
  { var m: Nonce;
    recv_1(I,R,{m}pk(B)); }
}
```

Figure 8. $\texttt{A} \rightarrow \texttt{B} : \{\texttt{m}\}_{\texttt{pk(B)}}$ en Scyther

La représentation de l'envoi et de la réception des messages est effectuée avec les mots clés `send_i` et `recv_i` qui prennent en paramètres les deux rôles intervenant dans l'échange décrit. Le label `i` indique que les deux évènements sont reliés, c'est-à-dire qu'à chaque envoi d'un message dans le protocole, correspond une réception du même message. Parmi les types de variables disponibles, nous disposons de nonces, d'agents et de clés, ainsi que des types définis par l'utilisateur. Les variables qui sont générées par les agents sont déclarées avec le mot clé `fresh`, les autres sont définies avec le mot clé `var`. Le code est très simple à comprendre car il est constitué d'une liste d'évènements d'envois et de réceptions, puis une liste de propriétés à vérifier. L'analogie avec Alice&Bob est visible d'autant que les clés sont représentées de la même façon (les clés publiques notées `pk(A)`, les clés privées notées `sk(A)` ...). Néanmoins cette syntaxe n'est pas assez riche ; par exemple la modélisation de l'indéterminisme ou des canaux privés n'est pas possible. Un point distinctif de Scyther est sa sortie graphique (cf. figure 9). Alors que les deux outils précédents ne présentent qu'une trace textuelle de l'attaque, Scyther produit un graphe pour schématiser les attaques. Les nœuds dans ce graphe représentent les événements de communication et les flèches l'ordre de ces évenements. Ce graphe est orienté verticalement, ce qui facilite la compréhension de la chronologie des événements lorsque plusieurs sessions se chevauchent.

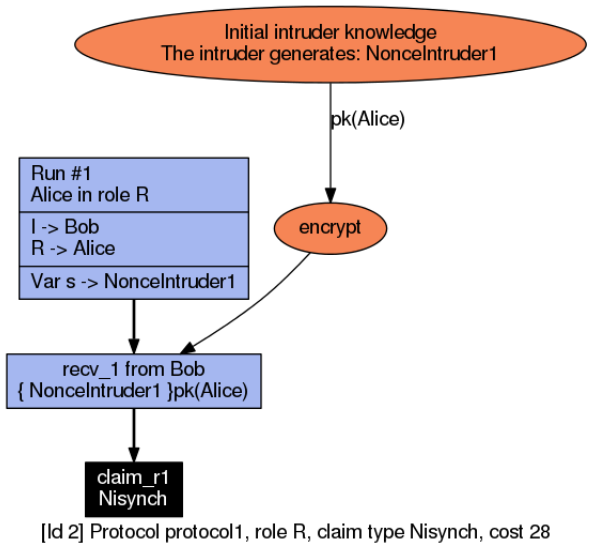

Figure 9. Un exemple de sortie de Scyther

Représenter les propriétés. Parmi les propriétés que nous avons définies, celles disponibles dans Scyther se limitent à l'authentification de message et au secret. La syntaxe générale pour la vérification d'une propriété est `claim(rôle, propriété, options)`. Par exemple `claim(R, Secret, N_A)` permet d'indiquer le secret d'un nonce N_A pour le rôle R. (cf. [12])

2.3 Estimer l'expressivité des outils

Nous avons ensuite étudié différentes classes de protocoles et la manière de les représenter dans chaque langage afin de mieux comprendre les possibilités et limites de chaque outil, ainsi que d'énumérer les mécanismes courants des protocoles cryptographiques que nous aurons besoin de représenter dans la seconde partie du projet.

Représenter les canaux privés La notion de canal privé est assez intuitive et est utilisée dans de nombreux protocoles. Elle permet de modéliser des canaux sécurisés, sur lesquels il est possible de faire transiter des données en les soustrayant à la vue de l'intrus.

Un premier exemple d'un tel protocole est le protocole de paiement par carte bancaire [13] (cf. figure 10). Lors de la « saisie du code à l'abri des regards indiscrets », il est supposé que le code transmis par l'utilisateur au terminal n'est pas intercepté : cet échange peut être modélisé en utilisant un canal privé.

$$
\begin{aligned}
&\texttt{T} \rightarrow \texttt{A} : \texttt{Authentification?}\\
&\texttt{C} \rightarrow \texttt{T} : \{\texttt{Nom}, \texttt{Prénom}, \texttt{Numéro de carte}\}_{\texttt{sk(Banque)}}\\
&\texttt{T} \rightarrow \texttt{A} : \texttt{Code?}\\
&\texttt{A} \rightarrow \texttt{T} : \texttt{1234}\\
&\texttt{T} \rightarrow \texttt{C} : \texttt{1234}\\
&\texttt{C} \rightarrow \texttt{T} : \texttt{ok}
\end{aligned}
$$

Figure 10. Protocole de carte bancaire simplifié : l'utilisateur `A` transmet son code (1234) au terminal bancaire `T` qui le transmet à la carte `C`. Si le code est bon, la carte clôture l'échange par un message de confirmation. Dans ce protocole, la propriété importante est l'authentification des agents, et il est clair que si l'intrus connaît le code de `A` il peut se faire passer pour lui, d'où la nécessité d'utiliser un canal privé.

En ProVerif et en Avantssar, la notion de canal privé est présente et se manipule très facilement. En Scyther en revanche, cette notion n'existe pas.

Indéterminisme Certains protocoles font appel à une notion de choix ou de hasard qui peut mener à plusieurs exécutions différentes : l'exécution du protocole est indéterministe. C'est un phénomène courant dans les protocoles multi-agents (plusieurs agents ayant un rôle similaire) où l'ordre de réponse des participants n'est pas fixé. De tels protocoles sont parfois représentés sous forme d'un automate. Le protocole Large Community of Device Protocol (LCDP) [14] en est un exemple ; il décrit de quelle manière un système de transfert pair-à-pair entre plusieurs agents peut être géré par un serveur centralisé. Un autre exemple est le protocole User Supervised Device Pairing (USDP) [15], qui vise à permettre l'appariement entre deux appareils. Ce type de protocoles est notamment utilisé sur des systèmes de communication wi-fi, bluetooth, etc.

U → A : initA	U → A : initA
U → B : initB	U → B : initB
A → B : Challenge	B → A : Challenge
B → A : ResponseB	A → B : ResponseA
A → B : ResponseA	B → A : ResponseB
A → U : okA	B → U : okB
B → U : okB	A → U : okA
U → A : okA	U → A : okA
U → B : okB	U → B : okB

Figure 11. Traces possibles d'USDP (Les messages Challenge, Response ... ne sont pas détaillés ici mais sont normalement construits à partir d'exponentiations

Le principe d'USDP est simple : l'utilisateur commence par allumer les appareils ; ceux-ci cherchent alors à entrer en contact avec d'autres appareils allumés. L'indéterminisme vient du fait qu'il est impossible de savoir quel appareil va initier la communication. Enfin, une fois les communications terminées, chaque appareil envoie un signal de fin à l'utilisateur. La figure 11 présente deux traces Alice&Bob simplifiées couvertes par ce protocole, avec un utilisateur U et deux appareils A et B.

ProVerif nous a semblé constituer le meilleur choix pour représenter des protocoles indéterministes. Il possède en particulier l'opérateur « | » qui permet d'effectuer les différentes exécutions possibles du protocole en parallèle.

2.4 Comparatif

La figure 12 résume synthétiquement les avantages et inconvénients des trois outils étudiés. Compte tenu de ce comparatif, nous avons décidé de concentrer notre travail du prochain semestre sur ProVerif. En effet, il s'agit de l'outil le plus expressif et le moins ergonomique des trois, et nous souhaitons donc concevoir et développer des mécanismes facilitant son utilisation. C'est sur cet outil qu'un tel travail semble le plus intéressant. Notons que ces dernières années, d'autres projets ont eu des directions similaires, comme SPAN [16] ou Prouvé [17] ; le premier a des objectifs proches des nôtres mais se concentre sur des outils de la famille Avispa (les prédécesseurs d'Avantssar), tandis que le second vise à définir rigoureusement un nouveau langage de description de protocoles et sa sémantique.

3 Propositions

Le formalisme Alice&Bob utilisé jusqu'ici manque d'expressivité pour traduire certaines notions, comme les canaux privés ou l'indéterminisme, et ne permet pas de retranscrire la connaissance des agents, ce qui mène à des comportements ambigus. Par exemple, la communication $\mathtt{A} \to \mathtt{B} : \{\mathtt{m}\}_{\mathtt{pk(B)}}$ peut signifier que B apprend le message m par l'intermédiaire de A, mais également qu'il connaît déjà m et doit vérifier que le contenu du message de A correspond bien.

Cependant, Alice&Bob reste très intéressant dans la mesure où il est concis et lisible. Nous proposons donc pour répondre aux problèmes évoqués précédemment de l'agrémenter de plusieurs améliorations afin d'obtenir un langage clair et expressif qu'il sera possible de convertir de manière automatique en ProVerif. Nous utiliserions ainsi les capacités de vérification de ProVerif sans que l'utilisateur n'ait besoin d'utiliser son langage de spécification.

Nous présentons désormais plusieurs éléments intéressants envisagés dans le cadre de cette extension du formalisme Alice&Bob puis précisons ceux que nous avons effectivement retenus.

3.1 Aperçu des extensions déjà existantes

Indiquer explicitement la génération de valeurs. L'introduction de valeurs fraichement générées rend parfois l'écriture des protocoles ambigüe,

Critères	Scyther	Avantssar	Proverif
Lisibilité de l'entrée (code)	✓	✓	
Lisibilité de la sortie	✓		
Expressivité			✓
Propriétés		✓	✓
Présence de canaux privés		✓	✓
Documentation		✓	✓
Open source	✓	✓	✓

Figure 12. Comparaison des fonctionnalités de Scyther, Avantssar et ProVerif.

et il est nécessaire de distinguer données créées pour un échange de celles issues des échanges précédents. À ce sujet, un article présentant une sémantique du langage Alice&Bob [18] propose d'étiqueter chaque échange du protocole avec la liste, entre parenthèses, des valeurs nouvellement créées. L'envoi par `A` à `B` du message `m` dépendant des données $\mathtt{N}_1 \ldots \mathtt{N}_p$ fraîchement générés serait ainsi noté :

$$\mathtt{A} \rightarrow \mathtt{B}\,(\mathtt{N}_1 \ldots \mathtt{N}_p) \;:\; \mathtt{m}.$$

Notons que ces valeurs sont le plus généralement des nonces, mais peuvent parfois être de nature différente, comme une clé symétrique générée pour une utilisation dans la suite du protocole (comme c'est le cas dans le protocole LCPD [14] par exemple). Dès lors, il sera peut-être nécessaire d'effectuer une différentiation syntaxique afin de respecter le typage de ProVerif.

Utiliser les canaux de communication. Nous introduisons ici plusieurs variantes de la notation $\rightarrow$ permettant de faire apparaître des informations supplémentaires sur la manière dont les échanges sont effectués. Formellement nous pouvons définir quatre types de canaux :

- Un canal est dit authentique ($\mathtt{A} \bullet\!\rightarrow \mathtt{B} : \mathtt{m}$) si et seulement si `B` peut s'assurer que seul `A` peut être à l'origine de ce message.
- Un canal est dit confidentiel ($\mathtt{A} \rightarrow\!\bullet\, \mathtt{B} : \mathtt{m}$) si et seulement si `A` peut s'assurer que seul `B` recevra ce message.
- Un canal est dit sûr ($\mathtt{A} \bullet\!\rightarrow\!\bullet\, \mathtt{B} : \mathtt{m}$) si et seulement si il est à la fois authentique et confidentiel.
- Un canal est dit frais ($\mathtt{A} \twoheadrightarrow \mathtt{B} : \mathtt{m}$) si et seulement si le message échangé sur ce canal ne peut pas être attaqué par rejeu. Autrement dit, même si l'intrus peut intercepter et apprendre `m`, il ne peut pas le réutiliser pour se faire passer pour `A` dans cet échange.

Il est important de comprendre que ces notations peuvent avoir plusieurs significations selon le contexte dans lequel elles sont utilisées. En effet, elles peuvent à la fois être utilisées pour effectuer des hypothèses sur les échanges d'un protocole – par exemple, dans le protocole de paiement par carte bancaire, le concepteur du protocole sait que l'on suppose la communication du code secret au terminal confidentielle – mais également pour coder des propriétés de sûreté que l'on cherche à vérifier sur le protocole. Ces deux utilisations sont possibles dans l'outil Avantssar et sont décrites dans différents articles [19] [20].

Composition de protocoles. Il existe aujourd'hui plusieurs résultats portant sur la composition de protocoles cryptographiques [21], c'est-à-dire sur leur utilisation conjointe, que ce soit de manière parallèle, séquentielle ou imbriquée. Il est par exemple possible, en étiquetant [22] des protocoles vérifiés de manière isolée, de garantir que ceux-ci conserveront les mêmes propriétés dans un environnement où plusieurs protocoles coexistent. Si ce cas s'éloigne un peu de notre travail dans la mesure où seuls des protocoles déjà écrits et fonctionnels sont concernés, notons que le cas de l'imbrication est particulièrement intéressant de notre point de vue. En effet, l'utilisateur peut souhaiter utiliser un protocole déjà existant et vérifié afin de sous-traiter une fonctionnalité – par exemple, pour générer une clé symétrique – dont il a besoin au sein du protocole en cours de conception. Il existe à ce sujet un résultat assez général reposant sur les théories équationnelles utilisées [23]. Toutefois, nous avons fait

le choix de ne pas se pencher sur cette fonctionnalité qui semble assez technique à implémenter.

3.2 Extension du formalisme Alice&Bob

Nous présentons ici notre proposition de syntaxe de spécification de protocole, ainsi que quelques pistes pour traduire cette syntaxe en ProVerif. Une spécification est décrite par des déclarations, une section `Protocol` et une section `Goals`.

Déclarer les connaissances des agents Comme nous l'avons déjà mentionné, une grande part de l'ambiguïté du langage Alice&Bob est de savoir exactement ce que connaît un agent à chaque échange. Une première amélioration possible est donc d'expliciter clairement les messages qui sont générés au cours du protocole (cf. paragraphe 3.1).

Une deuxième étape est de préciser la connaissance initiale des agents. En effet, certains protocoles utilisent des données initiales qui sont définies avant l'exécution et sont connues de certains agents. Nous verrons dans la suite (cf. sous-section 3.3) qu'il est possible d'inférer une partie de cette connaissance. Cependant, dans le cas où l'utilisateur voudrait rajouter des contraintes en spécifiant lui-même cette connaissance, nous rajoutons le mot clé `Knowledge(A)` qui permet de lister des connaissances initiales de l'agent `A` dans les déclarations.

Pour ce qui est des données initiales publiques, elles seront déclarées à l'aide du mot-clé `public`. Par défaut, l'ensemble des données publiques initiales contient déjà les clés publiques et identités de tous les agents.

Traduire les canaux privés Nous avons choisi de faire appel à la notion de canal confidentiel présentée précédemment et qui correspond bien aux canaux privés. Ainsi, dans la section `Protocol`, nous utiliserons la notation `A` $\rightarrow\bullet$ `B : m`, pour indiquer que `A` envoie un message à `B` sur un canal privé.

Indéterminisme Représenter un protocole non déterministe n'est pas une chose aisée. Nous nous sommes d'abord intéressés à une forme « simple » d'indéterminisme , qui correspond au cas des protocoles multi-agents où l'ordre d'interaction des agents est inconnu, et où tous les agents jouent un rôle identique. En effet, puisqu'en ProVerif un rôle est une fonction, il suffit, pour représenter plusieurs agents, d'appeler le même rôle avec des paramètres différents. La représentation que nous avons imaginée est la suivante :

```
Agents : A, D (5)
...
A → D? : M
D? → A : C
...
```

Cela se traduirait par : le protocole contient un rôle `A` et 5 rôles `D`, joués par des identités différentes mais dont les actions dans le protocole sont identiques. `A` envoie un message `M` à un des agents `D` (indéterminisme) qui lui répond par un message `C`.

Une autre particularité des protocoles multi-agents, est la notion de *broadcast*, c'est-à-dire communiquer un message à tous les agents du protocole. Ainsi, avec la syntaxe précédente, le message `A` $\rightarrow$ `D` correspondrait à : `A` envoie un message à tous les agents `D` dans un ordre indéterminé.

Enfin, nous proposons une forme plus générale pour représenter l'indéterminisme : un protocole indéterministe peut être vu comme un ensemble de sous-protocoles pour lesquels plusieurs choix d'exécution sont possibles. Nous décidons d'utiliser l'opérateur « | » pour représenter un choix. Ainsi,

```
Protocol :
  firstA | firstB.
  firstA { A -> B : m }
  firstB { B -> A : m }
```

représente un simple échange du message `m` entre `A` et `B` avec un indéterminisme sur l'identité de l'initiateur de la conversation.

Traduire les propriétés Nous nous concentrons ici uniquement sur les propriétés énumérées à la section 1. Les propriétés à vérifier seront décrites dans la section `Goals`.

Nous noterons simplement « `m secret` » pour indiquer que `m` doit rester inconnu de l'intrus (secret faible). Nous introduisons aussi la propriété de secret restreinte à certains agents par « `m secret of` $\mathtt{A}_1$ ... $\mathtt{A}_i$ » : à la fin d'une session du protocole, le message `m` est connu des agents $\mathtt{A}_j$ et uniquement d'eux.

La propriété de confidentialité d'un message est locale à un échange. Nous noterons « `i : m secret` » pour indiquer que l'échange se trouvant au label i est confidentiel par rapport au message `m`.

La propriété d'authentification de message est locale ; nous devons donc ici encore faire référence au

label correspondant à cette communication dans le protocole. Nous noterons « i : A $\bullet\!\!\rightarrow$ B : m » la propriété : le message m est authentifié par B dans la communication initiée par A à la ligne de label i. Nous utilisons ici la notion de canal authentique, qui a été présenté dans la sous-section précédente.

La notion d'authentification d'agent locale correspond à l'authentification de message à laquelle on rajoute une notion d'injectivité, d'où une grande similarité avec le point précédent. Nous noterons « i : A $\bullet\!\!\twoheadrightarrow$ B : m » la propriété : B peut s'assurer que le message m vient de lui être envoyé par A dans la communication initiée par A à la ligne de label i. Remarquons que cette propriété se traduit en combinant la notion de canal authentique et de canal frais : en effet si le canal est authentique, alors B a déjà l'authenticité du message, et si de plus le canal est frais, alors l'échange est protégé des rejeux donc aucun intrus n'a pu se placer en intermédiaire entre A et B pour faire passer le message ; d'où l'authentification de l'agent A.

La notion d'authentification globale concernant l'ensemble du protocole, nous noterons simplement « B authenticates A » pour indiquer que B authentifie A dans le protocole.

Exemples Nous présentons ici des descriptions de protocoles déjà introduits dans notre formalisme Alice&Bob amélioré.

```
Agents : A, B
Import : pk(), sk()

Protocol :
      A -> B (NA) : {NA}_pk(B)
  l1: B -> A (NB) : {NA, NB}_pk(A)
  l2: A -> B : {NB}_pk(B)

Goals :
  /*Secret*/
  NA secret
  NB secret
  /*Authentification de message*/
  l1 : B *-> A : NA
  l2 : A *-> B : NB
  /*Authentification d'agent globale*/
  A authenticates B
  B authenticates A
```

Figure 13. Protocole de Needham-Schroeder (cf. fig. 2)

```
Agents : U, D(2)
Import : pk(), sk(), exp()
Public : init, ok, end;
Protocol :
  U ->* D : init
  D?1 -> D?2 : Challenge
  D?2 -> D?1 : ChallengeResponse
  D?1 -> D?2 : Response
  D ->* U : end
  U ->* D : ok
Goals :
  s secret
  D?1 authenticates D?2
  D?2 authenticates D?1
```

Figure 14. Protocole USDP (cf. figure 11)

```
Agents : A, T, C, B
Import : pk(), sk()
Public : Ok, Code?, Authentification?
Protocol :
  T ->* A : Authentification?
  C -> T : {Data}_sk{B}
  T -> A : Code?
  A ->* T : code
  T -> C : code
  C -> T : Ok
Goals :
  code secret
  T authenticates A, T authenticates C
```

Figure 15. Protocole DDA (cf. figure 10)

3.3 Expliciter la traduction vers Proverif

Utiliser l'analyse syntaxique Une dernière incertitude réside dans la connaissance initiale des agents qui n'est pas précisée. Notre idée est de construire cette connaissance lors de la phase d'analyse syntaxique de la traduction du langage Alice&Bob vers ProVerif, complétée par les connaissances précisées par l'utilisateur à l'aide du mot-clé `Knowledge`. Ceci nous permettrait d'autre part de repérer les attaques évidentes sur le protocole en observant la connaissance de l'intrus (par exemple envoi d'un nonce en clair qui devrait être secret) sans avoir à passer par ProVerif.

Nous proposons donc d'utiliser cette phase d'analyse syntaxique pour construire pour chaque agent A l'ensemble de ses connaissances, `knows(A)` ainsi que l'ensemble de ses connaissances initiales, `init(A)`, c'est-à-dire l'ensemble des messages manipulés par A

qu'il n'apprend pas (par création de nonces, échanges de messages ...) au cours du protocole. En particulier `init(A)` $\subset$ `knows(A)`. Ces deux ensembles sont mis à jour au fur-et-à-mesure que l'analyse syntaxique évolue dans le protocole.

Générer le code ProVerif Bien que nous n'ayons pas encore commencé l'implémentation, nous avons déjà pu réfléchir à comment traduire les différents éléments clés de ce langage « Alice&Bob++ » vers ProVerif. Il serait ensuite intéressant de pouvoir montrer que le code ainsi compilé est bien correct.

Traduire un échange $\mathtt{A} \rightarrow \mathtt{B}(\mathtt{N}_1 \ldots \mathtt{N}_p) : \{\mathtt{m}\}_\mathtt{k}$

1. Si la communication a lieu sur un canal privé ($\mathtt{A} \rightarrow \bullet\, \mathtt{B} : \mathtt{m}$), il faut commencer par vérifier qu'un tel canal existe entre `A` et `B`, et le créer si ce n'est pas le cas.
2. **Rôle de A** Dans le rôle de l'initiateur, `A`, cet échange se traduira par la génération de p nonces puis le chiffrement et l'envoi du message ; dans le code ProVerif du rôle de `A`, nous ajoutons donc les lignes :

   ```
   ... new Ni:bitstring; ...
   out(c, enc(m, k))
   ```

   Ensuite, nous ajoutons les nonces à l'ensemble `knows(A)`. Puis, avant d'envoyer le message, nous vérifions qu'il est bien réalisable, c'est à dire que chaque élément nécessaire au chiffrement du message $\{\mathtt{m}\}_\mathtt{k}$ est bien contenu dans la connaissance de `A`, c'est-à-dire `knows(A)`. Si ce n'est pas le cas, nous choisissons de continuer l'analyse en ajoutant ces messages à la connaissance initiale de `A`, c'est-à-dire `init(A)`.
3. **Rôle de B** Dans le rôle du destinataire, l'échange en ProVerif se traduit par `in(c, message:bitstring)` : `B` apprend une donnée `message`, que nous ajoutons à l'ensemble `knows(B)`. Il reste ensuite à vérifier que le message reçu correspond bien à la forme attendue par l'agent, ce qui s'effectue par un `filtrage par motif` (*pattern-matching*). Tout d'abord, il est nécessaire de savoir si B peut déchiffrer le message ; il faut donc observer la fonction de chiffrement et tester si la clé de déchiffrement associée est dans la connaissance de `B`. Si c'est effectivement le cas, `B` déchiffre le message. Une fois le message en clair obtenu (quitte à en faire une procédure récursive dans le cas où le message contient lui-même des éléments chiffrés), il faut vérifier qu'il correspond à la forme renseignée dans le protocole : formellement, pour tous les éléments qui sont dans `knows(B)` à cette étape, et qui apparaissent dans le message de la description Alice&Bob, il est possible de tester leur égalité, ce qui correspond en ProVerif à : `let (..., =Q, ..., P:bitstring, ....) = dec(m,k)` ; où `Q` correspond aux messages qui sont dans `knows(B)` à ce moment, c'est-à-dire ceux que `B` peut reconnaître. À l'inverse les messages comme `P` correspondent aux messages appris par `B` en déchiffrant le message `m`.
4. Enfin, il faut mettre à jour les ensembles `knows()` après l'échange : en effet, chaque nouveau message appris par un agent peut lui permettre d'en déchiffrer de nouveaux et augmenter sa connaissance.

Traduire les propriétés Intéressons-nous désormais à la manière de traduire les différentes propriétés de vérification que nous avons présentées, en ProVerif. Nous avons déjà détaillé de manière assez précise comment exprimer les propriétés de secret dans la section 2.2. Pour ce qui est de la propriété de la confidentialité, cela peut se faire en utilisant le système de phases de ProVerif : nous pouvons définir des « phases » dans le protocole (à voir comme des points clés de l'exécution), et ensuite avec le mot clé `query attacker M phase n`, nous pouvons demander à ProVerif de vérifier la propriété du secret pendant la phase `n` du protocole.

De même pour la propriété d'authentification d'agent globale, nous avons déjà montré un exemple de spécification ProVerif pour les propriétés d'authentification. Concrètement, cela consiste à placer les mots-clés `event` au bon endroit du code, d'où l'avantage d'avoir une génération de code automatisée. L'authentification d'agents locale ($\mathtt{A} \bullet \twoheadrightarrow \mathtt{B} : \mathtt{m}$) repose sur le même principe que la propriété précédente, mis à part que les évènements seront localisés autour de l'échange particulier que l'on veut vérifier, et seront paramètrés par le message `m`.

Enfin, l'authentification de message ($\mathtt{A} \bullet \rightarrow \mathtt{B} : \mathtt{m}$) est semblable à la propriété d'authentification d'agents locale mais sans la notion d'injectivité, ce qui revient simplement à changer les mots-clés `query inj-event`, en `query event` pour traduire la non-injectivité.

3.4 Représenter les traces d'exécution

Le deuxième objectif de notre travail sera de faciliter la conception de protocoles. En effet, à partir du code ProVerif seul, il est difficile de se convaincre de ce que fait le protocole ; ainsi la description écrite peut ne pas correspondre à ce que l'utilisateur pensait signifier. Notre idée sur ce point est de pouvoir développer un protocole de manière interactive, c'est-à-dire que l'utilisateur pourrait exécuter le protocole pas-à-pas en choisissant l'échange à effectuer lorsque c'est possible. Cette exécution serait représentée sous la forme d'une séquence de messages.

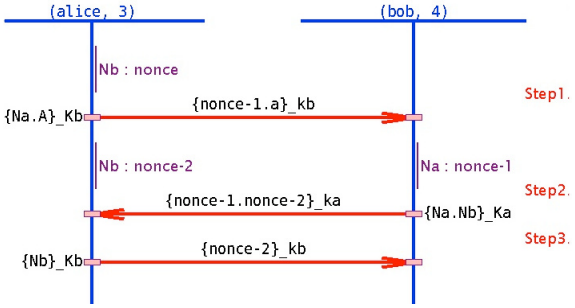

Figure 16. Un exemple de visualisation possible (outil SPAN)

Concrètement, la base du travail serait le code ProVerif généré. À partir de ce code, il est possible de retrouver les échanges de la trace Alice&Bob originale. Dans le cas où il y a plusieurs échanges possibles (car rappelons le, en ProVerif, les messages sont émis sur un canal sans préciser leur destinataire, d'où parfois une imprécision), ce serait à l'utilisateur de choisir l'exécution qu'il souhaite. Enfin, le protocole sera représenté sous forme d'un diagramme de séquences de messages (*message sequence chart*). Nous avons le choix entre représenter nous même l'aspect graphique du diagramme, ou bien nous contenter d'une représentation textuelle et utiliser des outils de génération déjà existants [24].

3.5 Clarifier la sortie de ProVerif

Enfin, la dernière partie de notre travail sera de présenter une sortie claire et lisible de l'outil de vérification. Actuellement, dans le cas où une attaque a été trouvée, ProVerif rend une trace d'attaque qui fait directement référence au formalisme du protocole en ProVerif (les évènements font référence aux lignes du code). Cette trace est compliquée à lire puisqu'il faut toujours se rapporter au code d'origine – en effet, la sortie ne fait pas mention du nom des agents, mais seulement des évènements `out` et `in`, de ce fait la trace seule a peu de sens. De plus la trace fait parfois référence à des messages générés pendant le protocole (nonces) avec des noms peu clairs.

Pour toutes ces raisons, nous voulons présenter une sortie compréhensible à partir de la sortie de ProVerif. Notre objectif est double, d'une part représenter la trace sous la forme d'un diagramme de séquence (ce qui rejoint le paragraphe précédent), d'autre part représenter l'attaque sous la forme d'un graphe, comme le fait Scyther par exemple.

Conclusion

Bien que les protocoles cryptographiques soient largement utilisés, les outils proposés à ce jour pour les vérifier manquent d'ergonomie ; il est compliqué de traduire un protocole depuis un langage simple, comme Alice&Bob, vers les langages plus formels qu'utilisent les vérificateurs. Il en va de même pour la spécification des propriétés de sécurité qui accompagnent les protocoles. Nous proposons de faciliter l'utilisation de ProVerif en implémentant trois modules complémentaires comme illustré en figure 17.

Le premier module est un compilateur permettant la traduction d'un langage Alice&Bob amélioré vers du ProVerif, le second permettra de visualiser de manière interactive le protocole pour en faciliter la conception, et le troisième permettra d'obtenir une sortie graphique claire du vérificateur. Nous pensons utiliser le langage de programmation `OCaml`, et ce en partie car c'est le langage dans lequel a été codé ProVerif. Les aspects graphiques nous mènerons à utiliser d'autres outils, comme peut-être `Html5/Javascript`.

Références

1. Véronique Cortier. Vérifier les protocoles cryptographiques. *TSI. Technique et science informatiques*, 24(1) :115–140, 2005.
2. Veronique Cortier and Ben Smyth. Attacking and fixing helios : An analysis of ballot secrecy. Cryptology ePrint Archive, Report 2010/625, 2010. http://eprint.iacr.org/.
3. Gavin Lowe. A hierarchy of authentication specifications. pages 31–43. IEEE Computer Society Press, 1997.

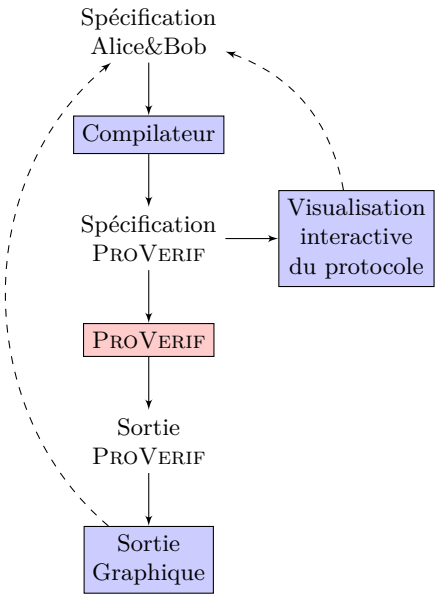

Figure 17. Chaîne de fonctionnement finale. Les blocs bleus (▭) correspondent aux outils proposés, les flèches pleines (⟶) représentent les entrées et sorties de ces outils, et les flèches en pointillés (- - ➤) traduisent le fait que l'utilisateur peut utiliser les informations qu'ils génèrent pour corriger la spécification de son protocole.

4. Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56 :131–133, 1995.
5. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 1998.
6. Paolo Maggi and Riccardo Sisto. Using spin to verify security properties of cryptographic protocols. In *In LNCS*, pages 187–204. Springer-Verlag, 2002.
7. Proverif. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.
8. C.J.F. Cremers. The Scyther Tool : Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.
9. Scyther tool. http://users.ox.ac.uk/~coml0529/scyther/.
10. Tamarin prover. http://www.infsec.ethz.ch/research/software/tamarin.
11. Avantssar : Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures. http://www.avantssar.eu/.
12. Casimier Joseph Franciscus Cremers. Scyther : Semantics and verification of security protocols, 2006.
13. Thomas Genet. Le protocole cryptographique de paiement par carte bancaire : Survey. http://interstices.info/protocole-cryptographique, 2008.
14. Olivier Heen, Thomas Genet, Stéphane Geller, and Nicolas Prigent. An industrial and academic joint experiment on automated verification of a security protocol. In Maryline Laurent-Maknavicius and Hakima Chaouchi, editors, *Mobile and Wireless Networks Security*, pages 39–53, 2008.
15. Olivier Courtay, Olivier Heen, Mohamed Karroumi, and Alain Durand. Secure device pairing under realistic conditions. *Applied Cryptography and Network Security (ACNS'06)*, pages 41–54, 2006.
16. Irisa. Projet span. http://www.irisa.fr/celtique/genet/span/.
17. Réseau National des Technologies Logicielles (RNTL). Projet "prouvé". http://www.lsv.ens-cachan.fr/Projects/prouve/.
18. Carlos Caleiro, Luca Viganò, and David Basin. On the semantics of alice&bob specifications of security protocols. *Theor. Comput. Sci.*, 367(1) :88–122, November 2006.
19. Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels. 5789 :337–354, 2009.
20. Ueli M Maurer and Pierre E Schmid. A calculus for security bootstrapping in distributed systems. *Journal of Computer Security*, 4 :55–80, 1996.
21. Véronique Cortier. Secure composition of protocols. In *Proceedings of the 2011 International Conference on Theory of Security and Applications*, TOSCA'11, pages 29–32, Berlin, Heidelberg, 2012. Springer-Verlag.
22. Véronique Cortier, Jérémie Delaitre, and Stéphanie Delaune. Safely composing security protocols. In *FSTTCS 2007 : Foundations of Software Technology and Theoretical Computer Science*, pages 352–363. Springer, 2007.
23. Stefan Ciobâca and Véronique Cortier. Protocol composition for arbitrary primitives. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 322–336. IEEE, 2010.
24. Mscgen. http://www.mcternan.me.uk/mscgen/.
25. Véronique Cortier and Ben Smyth. Attacking and fixing Helios : An analysis of ballot secrecy. *Journal of Computer Security*, 21(1) :89–148, 2013.