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 s'intéresse au développement d'un ensemble de modules visant à améliorer l'ergonomie du logiciel de vérification automatique ProVerif, ainsi qu'à développer l'interaction entre l'utilisateur et le logiciel.

Introduction

Le bon déroulement d'une communication entre plusieurs agents repose sur une formalisation commune que constitue le protocole utilisé ; celui-ci décrit l'ordre selon lequel les messages peuvent être envoyés par les différents participants, ainsi que la structure de ces messages. 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. Par la suite, nous nous placerons dans un cadre où un certain nombre d'agents, dits « honnêtes », communiquent entre eux dans un environnement où il existe un intrus, lui-même pouvant être aussi bien acteur qu'observateur de la communication, et cherchant à extraire des informations de l'exécution du protocole.

Il est le plus souvent nécessaire d'assurer certaines propriétés de sécurité sur les protocoles conçus – par exemple, assurer qu'un message sensible ne puisse pas être lu par quelqu'un d'autre que la personne à qui il est destiné. 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 des outils de vérification automatique que nous avons étudiés au premier semestre sont soit incomplets d'un point de vue spectre de propriétés vérifiables, soit difficiles d'utilisation.

Nous ne nous intéresserons pas au fonctionnement des opérations arithmétiques utilisées pour chiffrer les messages , mais nous contenterons d'utiliser les propriétés que vérifient ces opérations. Notamment, nous distinguerons deux modèles de chiffrement, asymétrique et symétrique. 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 notons `pk` ou `pkey` (*public key*). L'autre est dite secrète, car connue seulement par son propriétaire, et est souvent notée `sk` ou `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. Quant au fonctionnement du chiffrement symétrique, il repose sur le partage d'une clé entre 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 la même clé. Par la suite, un message `m` chiffré par une clé `s` sera noté $\{\texttt{m}\}_{\texttt{s}}$.

Notons que la sécurité des modèles de chiffrement précédemment décrits repose le plus souvent sur la difficulté de résolution de problèmes mathématiques. Dans notre approche, nous considérerons que les agents n'utilisent pas d' « astuce mathématique » pour retrouver un message, et qu'il donc est nécessaire, pour le déchiffrement, de connaître la clé associée : nous nous intéresserons seulement aux propriétés liées au déroulement du protocole en s'abstrayant des concepts arithmétiques sous-jacents. Plus exactement nous travaillerons sous les 3 hypothèses suivantes, qui définissent le modèle dit de Dolev-Yao :

Premièrement, 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 particulièrement intéressant de considérer l'exécution parallèle de plusieurs sessions car c'est souvent de là que viennent les attaques, l'intrus pouvant réutiliser les informations issues d'une session dans une autre. Pour distinguer les sessions, il existe un type de messages particuliers, appelés « nonces », (et souvent représentés par la lettre `N` suivie d'un identifiant) : ceux-ci sont des nombres générés aléatoirement lors de chaque session.

Deuxièmement, l'intrus peut intercepter et modifier tous les messages transitant sur le réseau, choisir ou non de les laisser parvenir à leur destinataire, tant que ces messages passent par un canal de communication public (visible par l'intrus). 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 (chiffrement, déchiffrement ...) sont aussi à sa disposition.

Enfin, 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 connaître 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 à d'éventuelles données supplémentaires initialement publiques propres au protocole.

D'un point de vue syntaxique, il est possible de décrire un protocole cryptographique sous la forme d'une séquence de *communications* du type « `A` envoie un message `m` à `B` ». Une telle communication est conventionnellement représentée par $\texttt{A} \rightarrow \texttt{B} : \texttt{m}$ et nous parlons de formalisme « Alice&Bob ».

Dans ce rapport, nous détaillons différents types de propriétés de sécurité qui peuvent être définies sur un protocole et présentons l'outil de vérification automatique sur lequel nous avons choisi de travailler, à savoir ProVerif. Puis nous présentons les modules facilitant l'utilisation de cet outil que nous avons développés, et enfin, discutons de leur validité.

1 Motivations et enjeux

1.1 La vérification de protocoles cryptographiques

Les propriétés spécifiant les protocoles cryptographiques sont très variées et dépendent pour beaucoup de l'usage que nous souhaitons faire du protocole. Dans cette sous-section, nous nous contentons d'étudier deux grandes classes de propriétés les plus communes : nous distinguons les propriétés dites « de secret », qui caractérisent la connaissance d'un terme par un ensemble d'agents, des propriétés « d'authentification » dont la vocation est de pouvoir confirmer à un agent l'identité de son interlocuteur ou bien l'origine d'un message reçu. Il est aussi possible de distinguer des propriétés locales et globales. Les premières font référence à une communication particulière dans le protocole, alors que les autres doivent être vérifiées sur des exécutions complètes. 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 vérifier des propriétés de secret.

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 à clé publique.

1.1.1 Propriétés de secret

Le premier type de propriétés souvent attendues d'un protocole cryptographique est de permettre la transmission d'un message sans qu'il ne soit visible par certain(s) agent(s) (en général, l'intrus). C'est ce que nous appelons des propriétés de secret.

Secret d'un message. Un message est dit secret si l'intrus n'en a pas connaissance à la fin de l'exécution 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 m est dit secret au sens fort si l'exécution du protocole jouée avec m est indistinguable pour l'intrus de toute autre exécution jouée avec un message 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.

Confidentialité d'un message. La confidentialité d'un message est une propriété de secret locale restreinte à un seul destinataire possible. Ainsi la communication $\mathtt{A} \rightarrow \mathtt{B} : \mathtt{m}$ est dite confidentielle si A est certain que seul B recevra cette instance du message m ; autrement dit A peut s'assurer à la fois que personne n'intercepte le message m et que le destinataire est bien b.

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

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

est confidentiel par rapport à m (car seul B peut décrypter le message pour obtenir m).

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

n'est pas confidentiel par rapport à m, car l'intrus peut intercepter le message, et, puisqu'il détient toutes les clés publiques, le déchiffrer avec pk(A) pour obtenir m.

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

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

1.1.2 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 les cas de confirmer à un agent l'origine du message qu'il reçoit ou bien l'identité de son interlocuteur.

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

Considérons par exemple le protocole suivant :

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

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

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

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

Authentification d'un agent. La propriété suivante ne porte plus sur le message échangé mais sur l'agent qui l'envoie ; nous parlons alors d'authentification d'agent. Cette nouvelle propriété permet par exemple de vérifier qu'aucun intrus n'usurpe l'identité de A au cours du protocole. Formellement, l'agent A est 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 avec l'agent B.

Prenons l'exemple du protocole suivant :

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

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 la valeur du nonce $\mathtt{N_B}$ aura 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.

Hiérarchie des propriétés d'authentification. Notons que l'authentification d'un agent A par un agent B est plus forte dans le sens où elle implique l'authentification de messages par B sur toutes les communications

de la forme $\mathtt{A} \rightarrow \mathtt{B} : \mathtt{m}$. Il est d'ailleurs possible de trouver des références à ces propriétés sous le nom d' « accord non injectif » et « accord injectif » [2]. Plus précisément, pour l'exécution d'un protocole sur un nombre infini de sessions, l'authentification d'agent requiert qu'à chaque communication de A vers B, l'agent B puisse s'assurer que A lui envoie directement ce message. Autrement dit, il faut une correspondance injective (un à un) entre les envois de messages de A et les réceptions de B (cf. figure 1). À l'inverse, l'authentification de message ne requiert que la création du message par l'agent A, mais B pourrait le recevoir par un autre intermédiaire.

1.1.3 Étude d'un exemple : le protocole de Needham-Schroeder Le protocole de Needham-Schroeder à clé publique (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. Si l'intrus ne fait qu'observer la communication entre A et B, 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. À l'inverse, si l'intrus joue le rôle d'un acteur de la communication, on verra dans la suite qu'il existe une attaque portant essentiellement sur l'authentification d'agents, et par laquelle l'intrus peut aussi obtenir connaissance des nonces $\mathtt{N_A}$ et $\mathtt{N_B}$.

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 (*i.e.* son interlocuteur) a créé ce message. En effet pour connaître $\mathtt{N_A}$ 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 [3] (cf. figure 3).

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

Figure 3. Attaque sur Needham-Schroeder

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 peut donc obtenir le contenu en clair du message (car il connaît sk(I)) puis le chiffrer à nouveau 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 nous désignons parfois en terme d'attaque de type « homme du milieu ». Une version corrigée du protocole est présentée en figure 4.

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

Figure 4. Protocole de Needham-Schroeder corrigé

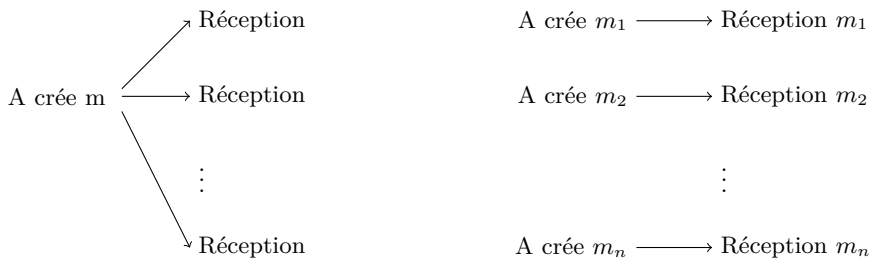

Figure 1. Authentification de messages (non injective) et authentification locale d'agent (injective).

L'ajout de l'identité de `B` combinée 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.

1.2 ProVerif : un outil de vérification complet

1.2.1 Présentation ProVerif est un outil de vérification automatique de protocoles cryptographiques développé par Bruno Blanchet à l'ENS Ulm [4] ; comparé à d'autres outils de vérification existant [5] [6], il possède de nombreuses fonctionnalités et une grande expressivité. Il permet notamment d'exprimer des outils de modélisation plus avancés des protocoles cryptographiques tels que les canaux privés (*i.e.* un canal de communications réservé à un certain nombre d'agents). Enfin, ProVerif couvre un très large spectre de propriétés de sécurité des protocoles, et notamment des propriétés de type équivalence observationnelle.

Cependant, cet outil reste assez complexe à prendre en main : bien que la documentation soit très complète, il est parfois difficile pour l'utilisateur de faire le lien entre sa vision intuitive du protocole sous sa forme Alice&Bob, et sa spécification ProVerif. De manière similaire, la sortie de ProVerif en cas d'attaque sur le protocole contient beaucoup d'informations sur la trace correspondante, mais est difficile à comprendre.

ProVerif est donc un outil essentiellement tourné vers la vérification de propriétés. Les objectifs de notre projet sont donc d'une part d'ajouter un aspect « simulation » à ProVerif en permettant à l'utilisateur d'exécuter son protocole pas à pas, et d'autre part de créer un langage d'entrée moins expressif mais plus clair, inspiré du langage Alice&Bob, ainsi qu'un compilateur de ce langage vers le langage d'entrée ProVerif, afin de pouvoir créer des spécifications de protocole ProVerif plus rapidement. Nous explorons dans la suite différents points du langage ProVerif, afin de donner l'occasion au lecteur de mieux comprendre le formalisme et ses subtilités.

1.2.2 Première approche de ProVerif

Syntaxe de ProVerif. Nous présentons ici un exemple de base d'un protocole réalisé en ProVerif, afin d'illustrer rapidement les éléments de base de la syntaxe. Nous nous intéresserons au protocole $\mathtt{A} \rightarrow \mathtt{B} : \{\mathtt{m}\}_{\mathtt{sk(A)}}$. La spécification ProVerif correspondante est présentée en figure 5.

Remarquons d'abord qu'un code ProVerif s'organise de la manière suivante. La phase de définitions permet de définir de manière publique des messages, des fonctions, des types ... à l'aide du mot-clé `free` : lorsqu'il est de plus accompagné de l'option `private`, il indique que seul les participants honnêtes (excluant ainsi l'intrus) ont accès à ces messages initialement . Analysons cette première partie sur notre exemple.

Tout d'abord, la définition du canal public `c`. Le type `channel` est un type de base ProVerif et est utilisé pour définir les canaux sur lesquels transiteront les messages.

Puis nous définissons les fonctions nécessaires au chiffrement, déchiffrement et à la création de messages. Dans ce cas, il s'agit simplement des fonctions d'encodage par clé publique (`aenc`) et décodage par clé privée (`adec`). Nous devons de plus créer deux types `pkey` (*clé publique)* et `skey` (*clé privée*), car il

```
(* Ouverture des canaux *)
free c : channel.

(* Définition des types clés secrètes
    et clés publiques *)
type pkey.
type skey.
fun pk(skey) : pkey.
fun aenc(bitstring , skey) : bitstring
    .
reduc forall x : bitstring, y : skey;
    adec(aenc(x, y), pk(y)) = x.

(* Propriété du secret *)
free m : bitstring [private].
query attacker (m).

(* Fonctions *)
let process_A(skA : skey) =
out(c, aenc(m, skA)).

let process_B(pkA : pkey) =
in(c, mess:bitstring);
let (=m) = adec(mess, pkA) in
0.

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

Figure 5. Exemple de spécification ProVerif pour le protocole $\texttt{A} \rightarrow \texttt{B} : \{\texttt{m}\}_{\texttt{sk(A)}}$

```
Definitions(fonctions, termes, canaux)
process
initialisation();
(fonction1)|...|(fonctionN)
```

Figure 6. Organisation schématique d'un code ProVerif

n'existe pas de types prédéfinis pour les clés, ainsi qu'une fonction `pk` qui retranscrit le fait qu'à toute clé privée est associée une et une seule clé publique.

Vient ensuite la partie vérification : ici la ligne `query attacker(m)` permet de vérifier que l'intrus ne peut jamais obtenir le message `m` en clair. Nous reviendrons sur la manière d'exprimer les propriétés formelles en ProVerif dans la sous-section suivante.

Pour clore ces définitions, il est possible de définir des fonctions. Ici, chaque fonction correspond au rôle d'un des agents (`process_X` représentant l'agent `X`). Remarquons que les paramètres de la fonction correspondent en quelque sorte à la connaissance initiale de l'agent nécessaire à l'exécution du protocole (clés publiques connues, clé privée de l'agent ...). Dans ces fonctions, nous voyons aussi apparaître les instructions de base pour modéliser les protocoles :

1. `out([canal], [terme typé])` correspond à l'émission d'un message sur un canal `c` ; le mot « terme » étant le nom de la structure interne avec laquelle ProVerif représente les messages.
2. `in([canal], [pattern])` correspond à la réception d'un message sur un canal. Remarquons que `in` ne prend pas en argument le message reçu : en effet, l'agent récepteur ne sait pas à l'avance ce qu'il va recevoir ; par contre, il prend en argument un `pattern` (*motif*). Ce pattern lui permet d'indiquer quelle « forme » de message il veut recevoir. Dans notre exemple, nous avons simplement écrit `in(c,m:bitstring)` : le pattern correspond ainsi à `:bitstring`, *i.e.* nous demandons à ce que le message reçu soit de type bitstring (ce qui en fait revient à ne faire aucun test à la réception du message car le type `bitstring` englobe tous les autres types).
3. l'instruction `new [terme]` (non présente dans l'exemple) permet la génération d'un nonce, c'est-à-dire d'un terme aléatoire dépendant de la session.
4. Enfin, une dernière instruction très utilisée est `let pattern = term in P else Q` qui permet de généraliser la notion de « pattern-matching ». ProVerif commence par évaluer le terme à droite de l'instruction ; si ce terme correspond au pattern à gauche, alors ProVerif affecte les variables dans le pattern selon la valeur du terme calculée, puis continue avec l'exécution de `P`. Sinon, il n'y a aucune affectation, et le futur du programme est `Q`. Le programme `Q` est optionnel, s'il n'est pas présent, alors le protocole s'arrête lorsque le pattern-matching échoue. Dans notre exemple, le pattern-matching est réalisé dans le rôle de `B` à la ligne : `let (=m) = adec(mess, pkB) in 0`.. Cette ligne permet donc de vérifier que le message `adec(mess,pkB)` retourne un terme valide de type `bitstring` et égal à `m`. Autrement dit, nous nous assurons que le message reçu est bien $mess = \{m\}_{sk(B)}$.

Une fois la partie contenant les définitions achevée, le mot-clé `process` indique où démarre exactement le protocole. Dans notre exemple, les deux premières lignes ont pour fonction de créer des clés privées `skA` et `skB` puis d'envoyer les clés publiques correspondantes sur un canal public, afin que l'intrus les apprenne Autrement dit, nous traduisons le fait que l'intrus connaît initialement les clés publiques de tous les agents, ce qui était implicite dans le formalisme Alice&Bob. Enfin, la dernière ligne, `(fonction1)...|(fonctionN)`, démarre le protocole entre les agents honnêtes du protocole : les N fonctions `(fonction1)` ... `(fonctionN)` sont démarrées et exécutées en parallèle. Dans notre exemple, le rôle de l'agent `A` et le rôle de l'agent `B` sont lancés de cette manière.

1.2.3 Représenter les protocoles en ProVerif

Communications asynchrones L'une des différences majeures entre le langage ProVerif et d'autres langages de formalisation de protocoles (Alice&Bob, Scyther [5], Avantssar [6]...) est que le langage ne comporte aucune mention explicite des agents acteurs du protocole.

Plus précisément, chaque agent peut être représenté par une fonction (son rôle) qui est appelée avec un certain nombre de paramètres (que nous pouvons voir comme la connaissance initiale de l'agent). À l'intérieur de ces rôles, les communications sont représentées par des émissions (`out([canal], [message])` puis lecture (`in([canal], [forme du message attendue])` de messages sur des canaux, éventuellement privés.

Remarquons que les émissions (*resp. réceptions*) de messages dans un rôle ne prennent pas en paramètres le nom de l'autre agent, récepteur (*resp. émetteur*) du message. Autrement dit, il n'y a pas d'informations explicites sur l'interlocuteur d'un agent dans son rôle en ProVerif.

Ainsi, pour retrouver les communications entre agents à partir de la spécification ProVerif, il nous faut d'abord arriver à mettre en correspondance les différents appels à `out` et `in` apparaissant dans le protocole. La tâche est d'autant plus compliquée qu'un protocole ProVerif peut souvent correspondre à plusieurs spécifications Alice&Bob (selon l'ordre dans lequel se font les communications par exemple). Pour cette raison, l'un des objectifs de notre outil est de reconstruire de manière interactive une exécution du protocole sous la forme d'une liste de communications entre agents.

Traduire les propriétés en ProVerif Nous nous intéresserons ici à la manière dont sont exprimées les propriétés de vérification en ProVerif. Nous pouvons d'abord distinguer deux types de propriétés de base présentes dans le langage, qui nous permettront ensuite d'en exprimer de plus complexes :

- Les propriétés de secret : elles sont utilisées pour vérifier qu'un terme donné ne sera jamais obtenu en clair par l'intrus, quelle que soit l'exécution du protocole jouée. En ProVerif, le fait que le message m puisse être obtenu par l'intrus dans une exécution du protocole se traduira par `attacker(m)`.
- Les propriétés d'atteignabilité : elles permettent de vérifier qu'un point précis sera atteint dans toutes les exécutions du protocole. Ces « checkpoint » dans le protocole, correspondent au type `event` (événement) de ProVerif. En pratique un objet de type événement est donc une fonction d'arité quelconque, mais n'ayant aucun effet lorsqu'elle est appelée, si ce n'est d'être présente dans le code et de servir de point de repère pour vérifier des propriétés d'atteignabilité.

Notons qu'il existe d'autres propriétés de base que nous ne détaillerons pas ici car elles ne nous sont pas utiles pour décrire les propriétés que nous avons présentées à la sous-section 1.1.

Ensuite, pour demander à ProVerif la vérification d'une propriété, nous utilisons le mot clé `query`. Plus précisément, si P est une propriété de base, la ligne `query P` revient à demander à ProVerif s'il existe une trace d'exécution du protocole au cours de laquelle la propriété P est atteinte. Par exemple pour demander à ProVerif de vérifier le secret d'un message m au cours de chaque exécution, nous utilisons la commande `query attacker(m)`.

Enfin, notons qu'à partir de ces propriétés de base, il est possible de construire de plus complexes en utilisant les constructeurs classique de la logique booléenne (&&, ||, $\implies$, *not*). Pour vérifier l'authentification par exemple, il n'y a pas de mots-clés prédéfinis, il faut passer par l'opérateur booléen $\implies$ et la notion d'événements. Rappelons nous par exemple la définition d'authentification d'un message que nous avions vu dans la sous-section 1.1.2 :

« L'agent `B` authentifie le message m envoyé par l'agent `A` si et seulement si lorsque B reçoit le message m, alors c'est que A a été créé et émis par `A` à un instant antérieur de l'exécution. ».

Si nous reformulons cette propriété sous forme d'événements, nous voyons apparaître l'événement `event arrivée(m) =` « B reçoit le message m » et `event`

`émission(m) =` « A crée et émet le message m. m peut donc être vu comme un paramètre de ces deux événements, et la propriété s'exprime maintenant sous forme booléenne : `event arrivée(m)` $\Longrightarrow$ `event émission(m)`, ce qui correspond à la manière dont nous formalisons cette propriété en ProVerif.

Quant à l'authentification d'agents, nous avons expliqué à la sous-section 1.1.2 qu'elle correspondait à l'authentification de messages à laquelle s'ajoute une notion d'injectivité. Ce caractère injectif existe aussi dans le formalisme ProVerif, et ainsi nous traduirons cette propriété par : `inj-event fin_role_B(A)` $\Longrightarrow$ `inj-event debut_role_A(B)`. Où l'événement `fin_role_B(A)` indique que B a fini d'exécuter son rôle avec l'agent `A` comme principal interlocuteur, et vice-versa pour `debut_role_A(B)`.

L'un des objectifs de notre projet étant de créer un langage de formalisation de protocoles plus proche du Alice&Bob (accompagné d'un compilateur vers ProVerif), nous avons choisi de ne pas utiliser le même formalisme que ProVerif pour définir les propriétés, car il alourdirait beaucoup la syntaxe. Nous avons préféré nous limiter à quelques propriétés (par exemple `A authenticates B` pour exprimer le fait que l'agent `A` veut authentifier l'agent `B`) que le compilateur traduira ensuite en ProVerif avec des événements, comme nous l'avons vu précédemment. Pour les utilisateurs plus expérimentés qui désireraient exprimer des propriétés plus complexes, ils pourront les rajouter eux-mêmes dans le code ProVerif rendu par le compilateur.

Résumé. La figure 7 résume les objectifs du projet : il se compose de 3 surcouches qui ont pour objectif de faciliter le développement de protocoles en ProVerif. Premièrement, un langage Alice&Bob amélioré accompagné d'un compilateur vers ProVerif, afin de pouvoir décrire des protocoles, certes limités par le manque d'expressivité du Alice&Bob, mais d'une manière plus rapide et plus simple que dans le formalisme ProVerif. Ensuite, nous proposons un outil de simulation d'un protocole décrit en ProVerif, pour permettre à l'utilisateur d'exécuter son protocole, et de mieux appréhender tous les scénarios possibles. Enfin, notre dernier ajout est une étape de traitement de la sortie ProVerif, qui est assez verbeuse et proche du formalisme ProVerif, afin de la transformer dans un langage plus proche du Alice&Bob. Cela permet à utilisateur peu expérimenté de plus facilement comprendre la trace d'attaque trouvée par ProVerif. Ces trois outils seront eux-mêmes intégrés au sein d'une interface graphique de notre conception.

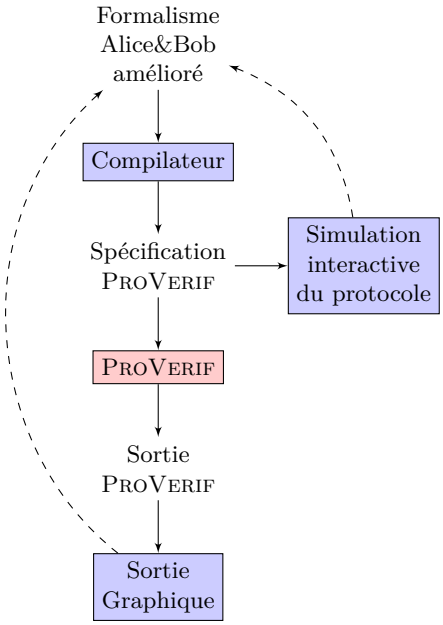

Figure 7. Chaîne de fonctionnement finale. Les blocs bleus (▭) correspondent aux outils proposés, les flèches pleines (⟶) représentent les entrées et sorties de ces outils, et les flèches en pointillés (⇢) traduisent le fait que l'utilisateur peut utiliser les sorties de nos outils pour corriger la spécification de son protocole.

2 Faciliter le développement de protocoles

2.1 Générer une spécification ProVerif à partir d'un langage d'entrée épuré

Notre premier objectif est de définir un langage de spécification intuitif, proche du Alice&Bob, mais plus complet et moins ambigu que celui-ci, accompagné d'un compilateur permettant de transformer ce langage en spécification ProVerif. Cela permettrait à l'utilisateur débutant de se familiariser avec la spécification de protocoles et de propriétés formelles, et aux utilisateurs ProVerif plus expérimentés de générer rapidement un squelette de code ProVerif fonctionnel. D'autre part, nous avons également ajouté un système de librairies à ce langage qui permet d'inclure des définitions ProVerif, ce qui nous a semblé pertinent dans le sens où l'un des défauts de ProVerif est qu'il contient très peu d'objets prédéfinis.

2.1.1 Un langage d'entrée épuré Nous présentons ici différentes extensions du formalisme

Alice&Bob qu'il est possible de retrouver dans la littérature et que nous avons choisi d'inclure dans notre langage.

Indiquer explicitement la génération de valeurs. L'introduction de valeurs fraîchement générées rend parfois l'écriture des protocoles ambiguë, et il est nécessaire de distinguer les données créées pour un échange de celles issues des échanges précédents. À ce sujet, un article présentant une sémantique du langage Alice&Bob [7] 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 générées est ainsi noté :

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

C'est sous cette forme que nous avons choisi d'exprimer une communication dans notre langage. Notons que ces valeurs générées 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 [8] par exemple).

Les connaissances publiques . 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.

Fonctionnalités avancées. Il existe une variante de la notation $\rightarrow$ [9] [10] permettant de faire apparaître des informations supplémentaires sur la manière dont les échanges sont effectués. Cependant, la manière d'exprimer ces notions est très difficile à transcrire en ProVerif, nous n'avons donc pas encore réussi à l'implémenter.

Exprimer l'indéterminisme en ProVerif s'est aussi avéré être trop complexe ; nous avons par conséquent décidé de laisser cette fonctionnalité à côté.

2.1.2 Implémentation du langage Une spécification dans notre langage se découpe en 3 parties, les déclarations, le protocole proprement dit et les buts de vérification. Dans la première sont déclarés les agents du protocole, la connaissance publique initiale ainsi que les librairies à importer. Nous pouvons définir des agents « de confiance », qui sont des agents dont l'intrus ne peut pas prendre le rôle dans l'exécution du protocole. Ensuite, nous définissons le protocole lui-même. Enfin, l'utilisateur peut indiquer des propriétés simples d'authentification et de secret à vérifier sur le protocole.

Un exemple d'une telle spécification, pour le protocole d'authentification de Needham-Schroeder (cf. 1.1.3) est présenté en figure 8.

```
Agents : A, B
Import : pk

Protocol :
  A -> B (Na) : {Na, A}_pk(B)
  B -> A (Nb) : {Na, Nb}_pk(A)
  A -> B : {Nb}_pk(B)

Goals :
  Na secret
  Nb secret
```

Figure 8. Le protocole de Needham-Schroeder écrit dans notre formalisme Alice&Bob.

Dans cet exemple, nous commençons par définir les agents `A` et `B` et importons la librairie `pk`, qui définit le chiffrement asymétrique. Le protocole se compose d'envois de messages entre `A` et `B`. Par exemple, la première ligne du protocole, `A -> B (Na) : {Na, A}_pk(B)` signifie que l'agent `A` génère un nonce `Na` et envoie à l'agent `B` le couple (`Na`, `A`) chiffré par la clé publique de l'agent `B`. Enfin, il est demandé comme but de vérification le secret des deux nonces. Notons que dans notre syntaxe, il existe aussi d'autres mots-clés, comme `Trusted` par exemple, qui permet de déclarer la liste des agents de confiance, *i.e.* ceux dont l'intrus ne peut usurper l'identité.

2.1.3 La structure de l'analyseur syntaxique La surcouche gérant la génération du code ProVerif depuis une entrée en Alice&Bob est constitué de plusieurs modules, dont l'organisation est présentée à la figure 9.

Nous commençons par le module de construction d'un arbre syntaxique (`AST`), reposant sur un lexeur et un parseur réalisés en `OcamlLex` et `OcamlYacc`. La création de l'`AST` s'effectue comme suit :

- Un arbre de syntaxe abstraite associé à la spécification du protocole Alice&Bob est construit.
- Une table des symboles est remplie au fur et à mesure avec les variables utilisées et leur type,

afin de vérifier, entre autres, les cohérences de nommage et de typage du fichier d'entrée.

- Par défaut, les variables générées au cours du protocole sont d'abord considérées comme des messages quelconques (`nonces`, type `bitstring` en ProVerif, puis nous spécifions leur type si besoin est. Par exemple, dans le protocole suivant, nous inférons que la variable `K` est une clé symétrique car elle est utilisées comme une clé de chiffrement (`Nb_K`) et qu'elle est générée au cours du protocole, donc ce ne peut être une clé publique ou privée (qui sont des données initiales).

```
A -> B (K) : {K}_pk(B)
B -> A (Nb) : {Nb}_K
```

L'étape suivant la construction de l'`AST` est la vérification basique des connaissances . Cette vérification permet à l'utilisateur d'éviter des erreurs liées à une mauvaise utilisation des variables définies dans la spécification du protocole. Prenons par exemple le protocole suivant :

```
A -> B (Na) : {Na}_pk(B)
C -> D : {Na}_pk(D)
```

Comme le nonce `Na` est créé par `A` pendant le déroulement du protocole, seuls `A` (puis, `B` qui le reçoit) le connaissent. L'agent `C` n'a donc pas le droit d'utiliser la variable `Na` puisqu'elle est attachée à un nonce qui lui est inconnu. Ce protocole renvoie donc une erreur à la compilation.

Il ne nous reste alors plus qu'à générer le code ProVerif à partir de l'`AST` construit, ce que nous détaillons dans la sous-section suivante.

2.1.4 Génération de spécifications ProVerif

Définitions initiales L'un des défauts de ProVerif est qu'il n'existe pas de système de librairies ou d'inclusions. Il est nécessaire, dans chaque fichier, de redéfinir les fonctions et types usuels (clés publiques, fonctions d'encodages...). Les seuls objets prédéfinis de ProVerif sont les types de base `bitstring` et `channel`, les fonctions n-uplets et les opérateurs booléens de base. Afin de générer les définitions initiales, nous commençons par établir la liste des librairies utilisées dans la spécification Alice&Bob indiquées par l'utilisateur après le mot-clé `Import`. Puis nous ajoutons les théories équationnelles correspondantes que nous avons nous-mêmes spécifiées en ProVerif.

Exemple :

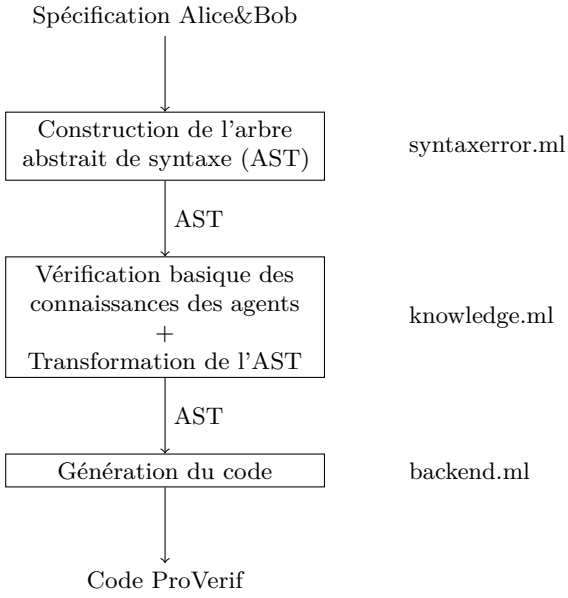

Figure 9. L'organisation du programme d'analyse syntaxique

`Import : pk` demande l'inclusion du type clé publique et des fonctions d'encodage et décodage, cela correspond à ajouter les lignes suivantes en ProVerif :

```
type pkey.
type skey.
fun pk(skey) : pkey.
fun saenc(bitstring, skey) :
    bitstring. (* Encodage par clé
    privée *)
reduc forall m : bitstring, k :
    skey; sadec(saenc(m, k), pk(k))
     = m.
fun paenc(bitstring, pkey) :
    bitstring. (* Encodage par clé
    publique  *)
reduc forall m : bitstring, k :
    skey; padec(paenc(m, pk(k)), k)
     = m.
```

La liste des agents nous permet ensuite d'établir les fonctions correspondant aux rôles des agents à écrire en `ProVerif`. Pour chaque agent, nous parcourons le protocole et déterminons les actions qu'il effectue.

Construire le rôle des agents Dans ce paragraphe, nous présentons de manière succincte de quelle manière nous traduisons les actions de base d'une communication en ProVerif.

- La génération d'un nonce `Na` correspond à écrire en ProVerif : **new Na : bitstring**.
- L'envoi du message `msg` sur le canal `c` correspond à **out(c, msg)**. Néanmoins, nous vérifions d'abord que l'agent peut bien constituer le message à partir de ses connaissances. Par exemple, dans le protocole suivant,

```
A -> S (Na) : {Na, B}_pk(A)
S -> B : {Na, A}_pk(B)
B -> A : {Na}_pk(A)
```

, l'agent `A` cherche à transmettre un nonce `Na` à l'agent `B` par l'intermédiaire d'un serveur de confiance `S`, mais l'utilisateur s'est trompé dans l'écriture du protocole. En effet, le message est crypté par `pk(A)` au lieu de `pk(S)` dans la première ligne. Ce qui a pour effet que la seconde ligne du protocole ne peut pas être exécutée car `S` ne peut décrypter le message reçu et ne connaît donc pas `Na`.
- La réception d'un message se traduit par `in(c, s)` où `s` est une chaîne de caractères unique que nous générons, indiquant le nom du message reçu. Ensuite, si ce message est crypté, nous le déchiffrons le plus possible en utilisant des instructions de pattern-matching `let ([pattern_1], ..., [pattern_r]) = f(s, ...)` de façon à ce que les connaissances de l'agent soit maximisées.

Constitution des messages Nous établissons aussi une table de connaissance pour chaque agent, qui évolue au cours du protocole, ce qui nous permet de déterminer quelles actions un agent peut effectuer et quels messages il peut effectivement construire. À l'état initial, ces tables contiennent entre autre toutes les variables globales déclarées ainsi que toutes les clés auxquelles chaque agent a accès (toutes les clés publiques et sa propre clé secrète par exemple).

Ces tables nous permettent d'une part de vérifier qu'un message peut-être construit par l'agent qui l'émet, et d'autre part, la connaissance des clés d'un agent nous permet de savoir si, et jusqu'à quel point, un agent peut déchiffrer un message qu'il reçoit. De plus, elles permettent de retenir les associations entre une variable Alice&Bob et son nom dans le protocole ProVerif. Par exemple, pour la génération ProVerif du protocole

```
A -> B (Na) : {Na}_pk(B)
```

il sera nécessaire de retenir que dans la spécification ProVerif pour l'agent `B` le nonce `Na` s'appellera `Na_known_by_B`, alors qu'il s'appellera `Na` pour l'agent `A`.

Constitution des buts Il reste enfin à traduire les buts de vérification demandés. Pour ce qui est de vérifier le secret en ProVerif, nous avons vu à la sous-section 1.2.3 que nous utilisions une instruction de la forme `query attacker(message)`, mais cela n'est possible que si la variable `message` est globale. Pour ce qui est du secret d'un nonce (*i.e.* un message généré *localement* dans un rôle) `N`, la méthode est de chiffrer de manière symétrique une variable globale témoin `secret_N` par le message `N` dont nous cherchons à assurer le secret. Ainsi, l'intrus n'a accès au message `secret_N` que s'il connaît la clé de déchiffrement, `N`. Le code ProVerif suivant est ainsi produit :

```
fun secretN_enc(bitstring,
    bitstring) : bitstring.
reduc forall m : bitstring, m' :
    bitstring; secretN_dec(
    secretN_enc(m, m'), m') = m.
free secretN : bitstring[private].
query attacker (secretN).
```

Quant à l'authentification d'agents, il est difficile de déterminer où doit être placé exactement l'événement de début de l'authentification (étant donné qu'il ne se situe pas nécessairement au début du protocole). Pour cette raison, nous n'avons donc malheureusement pas pu implémenter une traduction automatique en ProVerif de l'authentification d'agents.

2.2 Simuler l'exécution d'un protocole ProVerif

Le deuxième objectif de notre projet a été de créer une surcouche prenant un protocole ProVerif en entrée et permettant à l'utilisateur de simuler son protocole dans une interface graphique.

2.2.1 Représenter et retrouver les agents du protocole

La première étape afin de simuler l'exécution du protocole, est d'en construire l'état de départ, à savoir la liste des agents, leur rôle et leur connaissance initiale. Pour cela, nous avons d'abord extrait le plus d'information possible lors de l'étape d'analyse syntaxique du fichier ProVerif, puis nous avons réalisé une étape de pré-traitement sur la structure interne ProVerif du protocole (type `process`) pour en déduire les informations nécessaires restantes.

Rappelons qu'une spécification ProVerif a la forme suivante :

```
Definitions(fonctions, termes,
    canaux)
process
initialisation();
(fonction1)|...|(fonctionN)
```

Extraire les connaissances initiales. Les définitions initiales permettent par exemple de définir les canaux, les fonctions de chiffrement, les propriétés à vérifier etc. Tous les objets définis dans cette partie en tant que *privés* (option `[private]`) sont ajoutés à la connaissance initiale des agents honnêtes du protocole. Les autres objets sont donc publics, et sont ajoutés à la connaissance de tous les agents et à celle de l'intrus. Notons qu'on peut distinguer deux types de connaissance. D'une part les messages (ou `termes` ProVerif) : cette connaissance est évolutive car elle augmente à chaque fois que l'agent reçoit un nouveau message sur un canal. Ainsi chaque agent aura un champ `knowledge` contenant les termes qu'il connaît. D'autre part, il y a la connaissance des fonctions et des canaux : cette connaissance est statique car ces objets ne transitent pas, et ne peuvent donc pas être appris durant le protocole. De plus, leur caractère public ou privé est fixé dès l'initialisation et définit parfaitement de quels agents ils sont connus. Nous utiliserons donc des variables globales `public_functions` et `private_functions` pour mémoriser ces objets.

Enfin, cette phases de déclarations nous permet aussi d'obtenir des informations sur les propriétés de vérification. En effet, notre outil permet actuellement de détecter si une propriété simple (du type propriété de secret, `query attacker m` ou bien implication simple `query event(1)` $\implies$ `event(2)`) est violée au cours d'une simulation. Les informations relatives à cette propriété sont extraites lors de l'étape d'analyse syntaxique du fichier.

Extraire les rôles des agents. Après les définitions, ProVerif exécute la ligne `initialisation()` correspondant à la préparation de l'environnement du protocole en ProVerif, notamment la génération de clés et leur émission sur canal public pour permettre à l'intrus de les ajouter à sa connaissance. Nous représentons ces actions par le rôle d'un agent « particulier », dans le sens où son rôle s'exécute avant tous les autres, et non en parallèle avec eux. Par la suite, nous nommerons cet agent `network`.

Enfin, la dernière ligne représente l'exécution de `N` fonctions en parallèle. Nous avons décidé de considérer que chacune de ces fonctions représente un rôle, et donc un agent : le nom de l'agent est simplement le nom de la fonction appelée (avec éventuellement un renommage si le nom est déjà pris), et nous ajoutons les paramètres de la fonction à la connaissance initiale dudit agent. Notons que souvent, pour représenter le fait que la vérification s'effectue sur un nombre infini de sessions, les rôles des agents sont répliqués infiniment (opérateur `!` en ProVerif). Pour simplifier, nous considérerons pour l'instant qu'il n'y a qu'une seule instance pour chaque agent par exécution du protocole. Nous aborderons la paramétrisation du nombre d'agents et de sessions dans la section 2.2.3.

Cette étape nous permet donc d'extraire les `N` agents légitimes du protocole, dont les rôles s'exécutent en parallèle dans le même environnement. Un agent est principalement un type enregistrement comprenant un nom (`name`), un rôle (`role`) et les messages qu'il connaît (liste de termes ProVerif) (`knowledge`). À ceux-ci se rajoutent l'agent `network` représentant la phase d'initialisation du protocole, ainsi que l'intrus, qui est un agent particulier n'ayant pas de rôle.

2.2.2 Calcul des communications entre les agents

Gérer les émissions et réceptions de message. Étant donné un agent, nous souhaitons pouvoir exécuter son rôle, et faire correspondre entre eux les événements d'émission et de réception de messages du protocole pour inférer les communications entre agents.

Remarquons d'abord que, pour exécuter un rôle, nous avons besoin du choix de l'utilisateur dans trois cas seulement :

- `in(c, pattern)` : lors de la réception d'un message, c'est à l'utilisateur de choisir exactement quel message l'agent doit lire sur le canal `c`.
- `get(tables, t)` : il s'agit de la récupération d'un terme `t` depuis une table de hachage `tables`. En effet, lorsqu'un agent veut récupérer un message depuis la table, il vaut mieux mettre en pause son exécution, au cas où le message n'aurait pas encore été inséré (instruction `insert`) ce qui provoquerait une erreur dans le rôle l'agent. Dans la suite, ces instructions ne seront pas détaillées, mais pour il est possible de voir ce mécanisme comme une communication entre un agent et une table de hachage, ce qui nous ramène par analogie au cas

de la communication entre agents avec `in` et `out` ; cas que nous détaillerons.

- `R1 | R2` : lorsque le rôle contient un branchement, c'est-à-dire qu'il peut se continuer de deux manières possibles (`R1` ou `R2`). On laisse ce choix à l'utilisateur. À ce jour, notre outil ne permet pas d'inclure de l'indéterminisme dans un rôle.

Pour toutes les autres instructions apparaissant dans un rôle, nous pouvons les exécuter indépendamment des autres agents du protocole, ou des choix de l'utilisateur. Nous ne ferons pas ici une description complète de toutes ces instructions, mais nous pouvons notamment citer `out(c, m)` et `event(t)` qui sont les plus intéressantes, du point de vue de leur effet. `out(c, m)` correspond à l'émission du message `m` sur le canal `c`. En interne, un canal sera représenté par une association « nom de canal » - « listes de termes présents sur le canal » (on stocke aussi des informations complémentaires, telles que l'identité de l'agent qui a émis le message par exemple). Ces associations seront rangées dans une table de hachage. Quant à l'instruction `event(t)`, elle correspond à la lecture d'un événement `t` dans un protocole. Il nous suffit alors d'émettre un signal contenant l'information de l'événement et du rôle dans lequel il apparaît, qui sera ensuite récupérée par les applications graphiques afin de le représenter. On utilise aussi les signaux d'événements pour vérifier les propriétés simples d'implication demandées sur le protocole (par exemple lorsque l'événement `event 1` apparaît, et qu'on veut vérifier la propriété `query event(1)` $\implies$ `event(2)`, alors notre outil vérifie qu'il a bien vu l'événement `event 2` antérieurement. Si ce n'est pas le cas, la simulation s'arrête).

Finalement, nous calculons les choix de communications à chaque étape de la manière suivante :

- Pour chaque agent encore présent dans la communication (*i.e.* dont le rôle n'est pas vide), nous déroulons son rôle jusqu'à arriver à une instruction nécessitant un choix (`in` ou `get`) de l'utilisateur, ou bien jusqu'à arriver à la fin du rôle (dans ce cas nous pourrons supprimer l'agent de la liste des agents actifs).
- Une fois que chaque agent s'est « arrêté », les agents actifs restants sont donc tous bloqués sur une instruction `in` (attente d'un message) ou bien `get` (que nous ne développerons pas ici car la cas est très similaire avec celui de l'instruction `in` mis à part que l'agent « communique » avec une table de hachage et non avec un agent). Nous pouvons alors, pour chacun de ces agents, observer les messages présents sur les canaux, et en déduire des communications possibles en réalisant plusieurs tests de « compatibilité » (égalité des canaux, concordance de types de messages ...) que nous détaillerons dans le paragraphe 2.2.2.
- Enfin, une fois que l'agent a choisi sa communication, il faut mettre à jour les canaux (suppression du message reçu du canal), les connaissances de l'agent récepteur et de l'intrus (si le canal était public), et enfin faire avancer le rôle de l'agent récepteur d'un pas.

Cet algorithme est détaillé en annexe dans les pseudo-codes 3.1 à 3.4. Dans la suite, nous revenons plus en détail sur quelques points de l'implémentation.

Restreindre les choix de l'utilisateur Nous savons désormais inférer un ensemble de communications possibles à un instant donné de l'exécution du protocole ; cependant, selon la manière dont nous caractérisons une communication *possible*, leur nombre peut être assez important. Afin de garder un outil facile à utiliser, nous devons donc limiter au maximum le nombre de ces communications, et pour cela nous devons renforcer les tests que nous réalisons pour déterminer si une communication est « possible » ou non. Lorsque nous considérons un rôle `in(c, pat)`, que devons nous chercher dans un message `m` déjà émis pour qu'ils soient compatibles et forment une communication ? Dans le paragraphe précédent, nous ne vérifiions que deux critères : l'égalité des canaux entre `c`, le canal de réception, et le canal sur lequel est émis `m`, puis la correspondance entre le terme `m` et le pattern `pat`.

À cela peuvent se rajouter d'autres critères qui permettent de renforcer les tests, et d'éliminer des communications impossibles :

- Vérifier la validité de la communication dans le futur : il s'agit d'avancer dans le rôle de l'agent récepteur (de nouveau, jusqu'à arriver à une instruction `in` ou `get`), en supposant que le message `m` a été reçu. Si l'exécution du rôle échoue (à cause d'une erreur de pattern-matching par exemple), alors c'est que cette communication n'était pas possible, puisqu'elle amène à un blocage du rôle de l'agent.
- Vérifier l'égalité des phases : les phases en ProVerif sont définies par des points de changement de phase dans les rôles des agents, et sont telles que deux agents ne peuvent communiquer que s'ils sont dans la même phase du protocole, ce qui limite encore une fois les choix de communications.

2.2.3 Multiplier les agents : les sessions

Nous avons implémenté l'outil de telle sorte que l'utilisateur plus expérimenté puisse lui-même paramétrer le nombre de sessions du protocole s'exécutant en parallèle et le nombre d'instances d'agents par session.

En effet cela peut s'avérer utile dans certains cas où un agent doit exécuter plusieurs fois son rôle pour que le protocole s'exécute entièrement. Par exemple dans le protocole de Needham-Schroeder fourni avec ProVerif, un des agents est un tiers de confiance dont le rôle est d'être un serveur de clés : lorsqu'il reçoit un couple d'identités d'agents `X, Y`, de renvoyer à l'agent `X` la clé publique de `Y`. Ainsi son rôle doit être sollicité plusieurs fois pour simuler entièrement un protocole (à chaque fois que quelqu'un demande une clé, il intervient), et nous laissons donc le choix du nombre d'exécutions à l'utilisateur.

2.2.4 Modéliser l'intrus

Le rôle de l'intrus En interne, l'intrus `I` sera représenté par deux agents différents : `Intruder_In`, qui représente l'intrus récepteur de messages et `Intruder_Out` représentant l'intrus émetteur de messages. Cependant, ces deux agents partagent la même connaissance qui sera centralisée dans une variable globale partagée.

L'avantage d'avoir deux agents différents est que nous pouvons plus facilement régler le niveau de « pouvoir » de l'intrus. Nous avons donc par défaut un mode `Active` lorsque l'intrus peut librement envoyer et recevoir des messages ; puis un mode `Restricted` lorsqu'il ne peut qu'envoyer des messages qui ont été créés par d'autres agents et qu'il a obtenu sur un canal public. Enfin, nous avons ajouté un mode d'exécution `Passive`, lorsque l'intrus ne peut que recevoir des messages (autrement dit `Intruder_Out` est absent).

La connaissance de l'intrus Un enjeu critique de la modélisation de l'intrus dans les communications, est de pouvoir inférer ses connaissances dynamiquement au cours de l'exécution. En effet la connaissance d'un agent est ce qui lui permet de construire des messages, les déchiffrer… et dans le cas de l'intrus, sa connaissance nous permet en particulier de savoir quels messages ne sont plus secrets, et lesquels il peut construire à partir sa connaissance, et donc envoyer aux autres agents.

Il y a trois manières pour l'intrus d'augmenter sa connaissance.

- En récupérant les termes contenus dans la connaissance initiale publique du protocole (que nous avons déjà obtenus lors de la première étape de traitement du fichier ProVerif dans la sous-section 2.2.1).
- En observant les messages transitant sur les canaux publics, ce qui est aussi assez simple à mettre en place, dès lors que nous connaissons le nom des canaux publics.
- En déchiffrant des messages. C'est la partie la plus complexe à réaliser. En effet, pour les agents honnêtes du protocole (*i.e.* ceux dont le rôle est défini dans le fichier ProVerif), le décodage de termes est encodé directement dans le rôle, et cela fait donc simplement partie de la simulation de communications vue dans la sous-section précédente. À l'inverse, comme l'intrus n'a pas de rôle défini, il faut, à chaque fois qu'il reçoit un nouveau message (émission sur un canal public), que nous inférions quelles informations il peut tirer de cette nouvelle connaissance. Dans le reste de la sous-section, nous détaillerons la manière dont nous avons traité ce problème.

Le déchiffrement de messages se fait grâce au moyen de fonctions particulières appelées « destructeurs » déclarées par le mot clé `reduc`. Un destructeur comporte un certain nombre de règles qui permettent d'obtenir un nouveau terme, lorsqu'il est appelé avec les bons arguments.

Exemple : `reduc forall m : bitstring, k : skey; padec(paenc(m, pk(k)), k) = m`. La règle de destruction `padec` indique qu'il est possible de déchiffrer un message chiffré avec la fonction `paenc` et une clé publique, si nous possédons la clé privée associée.

Lors de l'analyse syntaxique du fichier ProVerif, nous récupérons les règles des destructeurs. Une première idée serait donc, pour un terme donné, de tester toutes les règles possibles pour essayer de le déchiffrer et d'en extraire un nouveau message. Le défaut principal de cet « algorithme » est qu'il nécessite beaucoup de tests inutiles ; nous l'avons donc amélioré sur plusieurs points afin de limiter le nombre de destructeurs à considérer.

Tout d'abord, il est important de définir les termes dits « en clair » : ce sont les variables et les fonctions constantes ; ces termes ci sont déjà déchiffrés. À l'inverse des termes en clair, les termes chiffrés sont de la forme d'une application de fonction, autrement dit ils s'écrivent `t = FunApp(f,l)` où `f` est un symbole de fonction et `l` une liste d'arguments. Notre objectif étant de limiter les tests de destructibilité sur un terme, nous avons décidé de conserver une

table de hachage `destructors` qui à chaque symbole de fonctions `f` associe toutes les règles de destructeurs pouvant s'appliquer à un terme de la forme `FunApp(f,liste)`.

Plus précisément, soit une règle de destruction de la forme $\mathtt{g}(\mathtt{M}_1, \ldots, \mathtt{M}_n) = \mathtt{M}_0$. Pour chaque terme de la forme `t = FunApp(f,l)` apparaissant dans l'un des arguments $\mathtt{M}_i$, nous commençons par regarder si l'un des termes dans `l` apparaît dans $\mathtt{M}_0$: dans ce cas, il est possible de dire que l'application du destructeur à $\mathtt{M}_i$ apporte de l'information sur l'un des termes de `l` ; nous pouvons donc associer à f la règle du destructeur `g`. Pour un symbole de fonction `f` donné, une règle de destructeur associée contient le nom du destructeur, ses arguments (sous la forme de pattern) ainsi que la position de l'argument où nous devons placer `t = FunApp(f,l)` (encore une fois sous forme de pattern) pour appliquer le destructeur.

Finalement, grâce à cette étape d'initialisation, nous pouvons construire une table de hachage `refined_destructors` qui à un symbole de fonction `f` donné associe toutes les règles de destruction pouvant apporter de l'information sur un terme de la forme `FunApp(f,l)` Ainsi lorsqu'un nouveau terme `m` arrive dans la connaissance de l'intrus, nous testons pour chacune des règles de destructeurs applicables, si l'intrus peut construire les arguments du destructeurs à partir des messages qu'il connaît et des symboles de fonction publiques :

- S'il n'existe aucun destructeur pouvant s'appliquer au terme `m`, alors c'est que le terme est clair ;
- Si tous les destructeurs applicables au terme ont pu être appliqués avec succès, le terme ne peut apporter d'informations supplémentaires, on n'aura donc plus besoin d'effectuer de test de destructeurs sur ce terme.
- Enfin, si l'un des destructeurs applicables au terme `m` n'a pas pu être appliqué, le terme `m` est susceptible de pouvoir encore être décrypté.

Enfin, nous appliquons à nouveau la procédure à tous les termes qui ont été créés par destruction afin de les ajouter eux-mêmes à la connaissance d'intrus. Puis, comme c'est un apport de l'information, nous devons répéter la procédure avec tous les termes encore susceptibles de pouvoir être déchiffrés dans la connaissance de l'intrus.

Exemple : supposons avoir comme destructeur le déchiffrement par clé privée : `reduc forall m : bitstring, k : skey; padec(paenc(m, pk(k)), k) = m`.

1. **Étape d'initialisation**. Étudions chaque application de fonction apparaissant dans les arguments du destructeur :
   - `paenc(m, pk(k))` : comme `m` apparaît dans le résultat du destructeur, nous avons un apport d'information. Nous pouvons donc lier le symbole de fonction `paenc` à la règle `padec, [args = paenc(m,pk(k)), k], [position = paenc(m, pk(k))]`
   - `pk(k)` : comme `k` n'apparaît pas dans le résultat du destructeur, nous en déduisons que l'application de `padec` n'apporte aucune information à partir du terme `pk(k)`. Nous ne lions pas cette règle à `pk`.
2. **Étape de mise à jour de la connaissance**. Ajoutons quelques termes à la connaissance de l'intrus pour voir comment se comporte cette connaissance.
   - Si l'intrus reçoit `skA` (clé secrète de `A`). C'est une variable, donc un terme en clair. Il n'y a donc aucun autre test à faire.
   - Si l'intrus reçoit `pk(skB)` (clé publique de `B`). Il y a le symbole de fonction `pk`, mais aucun destructeur associé ; encore une fois il n'y a rien à faire (rien à déchiffrer).
   - Si l'intrus reçoit `t = aenc(mess, pk(skA))`, il remarque qu'il connaît une règle de destructeur associée à `aenc` et dont l'argument en tant que pattern correspond à `t`. Il commence donc par associer le terme `t` au pattern de l'argument (dans notre exemple, le remplacement de variables donne $\mathtt{m} \leftarrow \mathtt{mess}$ et $\mathtt{k} \leftarrow \mathtt{skA}$), ce qui fixe les variables (pour les variables qui ne sont pas fixées, l'intrus est libre de choisir n'importe quelle variable de sa connaissance, du temps que le type est respecté). Ensuite, il observe les autres arguments du destructeurs avec ces variables fixées (donc ici l'argument $\mathtt{k} \leftarrow \mathtt{skA}$). S'il peut construire ces arguments (donc dans notre cas, s'il possède `skA`), alors il applique le destructeur pour obtenir un nouveau message (ici `mess`).

2.3 Simplifier la sortie ProVerif

seront d'abord interprétées comme une communication dont l'intrus est le récepteur. Puis, à chaque instruction de réception, nous commençons par chercher s'il existe une émission antérieure lui correspondant (c'est-à-dire égalité du canal de communication et du

message échangé). Si oui, alors nous avons l'émission et la réception, donc avons une communication entre deux agents valides du protocole. Si aucune émission antérieure ne correspond, nous en déduisons que c'est l'intrus l'émetteur du message, et nous pouvons aussi construire la communication. Remarquons que cette idée repose sur le fait que la trace renvoyée par ProVerif est chronologique : autrement dit, pour une instruction de réception donnée, nous avons seulement besoin de considérer les émissions antérieures dans la trace pour en déduire sans ambiguïté la communication associée à cette réception.

Enfin nous utiliserons un système de marquages pour distinguer d'une part les communications « finies », c'est-à-dire celles pour lesquelles nous avons déjà déterminé l'émetteur et le récepteur. A l'inverse nous ne marquerons pas les communications « partielles », *i.e.* celles qui ont été créées par la rencontre d'une instruction d'émission et dont nous n'avons pas encore déterminé le récepteur.

Le troisième et dernier objectif de notre outil est cette fois ci axé sur l'aspect vérification de ProVerif ; il s'agit de simplifier la sortie pour qu'elle soit plus claire aux yeux de l'utilisateur. Pour une propriété donnée, la sortie de ProVerif se compose de trois parties :

- Un résultat global. (*i.e.* est-ce que la propriété est vraie ou fausse ?)
- Dans le cas où la propriété est fausse, ProVerif recrée un contre-exemple (appelé une *dérivation*). La dérivation sera affichée dans la sortie sous une forme textuelle mais verbeuse.
- Enfin, ProVerif recrée une trace à partir de la dérivation : la différence vient du fait que la dérivation générée peut ne pas être réalisable à cause des approximations faites par le moteur de vérification ProVerif, alors que la trace recréée, elle, est exécutable.

Dans la suite, nous nous intéresserons à la simplification de la trace trouvée par ProVerif en cas d'attaque. Nous exposons en figure 10 la trace ProVerif de l'attaque du secret du message `m` dans le protocole $\texttt{A} \rightarrow \texttt{B} : \texttt{m}_{\texttt{sk(A)}}$. Bien que l'attaque soit évident (l'intrus intercepte le message et le déchiffre avec `pk(A)`, la trace obtenue, elle, est assez compacte. Au niveau du formalisme, nous voulons obtenir une trace plus proche du formalisme Alice&Bob.

Nous pouvons déjà remarquer que chaque ligne de la trace correspond à une instruction atomique du protocole, suivie du numéro de la ligne où l'instruction est exécutée, puis des informations sur la session : `instruction at [ligne] in copy [session]`

```
new skB creating skB_133 at {1}
out(c, pk(skB_133)) at {3}
new skA creating skA_132 at {4}
out(c, pk(skA_132)) at {6}
out(c, paenc(m,skA_132)) at {8} in copy
    a
The attacker has the message m.
A trace has been found.
```

Figure 10. Exemple de sortie ProVerif pour l'attaque de $\texttt{A} \rightarrow \texttt{B} : \texttt{m}_{\texttt{sk(A)}}$

2.3.1 Simplifier l'écriture La première chose que nous observons sur la syntaxe de la trace, c'est le nommage des termes : les indices (partie après le caractère _) sont attribués lors de la génération d'un terme pendant la recherche d'une attaque par ProVerif, et de ce fait, à l'affichage, ils alourdissent l'écriture. Nous avons choisi de tous les renommer en observant leur identifiant (la partie du nom avant le caractère _) puis en les numérotant à nouveau en partant de 0. Les correspondances entre termes et (re)numérotations seront mémorisées dans une table de hachage, pour éviter d'utiliser des noms déjà pris. Dans le même esprit, nous avons simplifié l'écriture des fonctions de chiffrement : par exemple le terme `paenc(m,key)` correspond, dans notre formalisme Alice&Bob au terme `m_key`. Ainsi, pour les sorties dont le fichier d'entrée était sous forme Alice&Bob, et que nous avons compilé en ProVerif, nous avons la correspondance entre la syntaxe Alice&Bob et la syntaxe ProVerif ; de sorte qu'il nous est facile de simplifier l'écriture ProVerif de la sortie.

2.3.2 Retrouver la trace Alice&Bob correspondante Le problème ici est assez similaire à celui de la sous section 2.2.2, et même plus simple dans la mesure où nous n'avons pas à gérer les choix de l'utilisateur. La sortie ProVerif est représentée en interne par une structure de trace qui consiste en une suite d'émission et réceptions de messages (ainsi que d'autres informations, telles que l'apparition d'événements au cours de l'exécution par exemple). Le seul point critique est donc d'inférer les communications entre agents à partir des émissions et réceptions.

Notons que chaque instruction apparaissant dans la trace est accompagnée d'un numéro, qui correspond en fait à la ligne du protocole où elle est exécutée : ce sont donc des instructions d'agents honnêtes. De ce fait les émissions et réceptions de l'intrus n'apparaissent pas

dans la trace et il faut donc les inférer. Notre première étape est d'associer chaque instruction de la trace à l'agent qui la réalise : pour cela, nous réutilisons la liste des agents obtenus à l'initialisation (*c.f.* sous section 2.2.1) ; nous pouvons alors associer à chaque agent l'intervalle de lignes correspondant à son rôle dans le protocole. Il nous est donc maintenant facile de faire la correspondance entre numéro de ligne et agent.

Vient ensuite la construction des communications, dont l'idée est la suivante : par défaut, toutes les émissions sur un canal seront d'abord interprétées comme une communication dont l'intrus est le récepteur. Puis, à chaque instruction de réception, nous commençons par chercher s'il existe une émission antérieure lui correspondant (c'est-à-dire égalité du canal de communication et du message échangé). Si oui, alors nous avons l'émission et la réception, donc avons une communication entre deux agents valides du protocole. Si aucune émission antérieure ne correspond, nous en déduisons que c'est l'intrus l'émetteur du message, et nous pouvons aussi construire la communication. Remarquons que cette idée repose sur le fait que la trace renvoyée par ProVerif est chronologique : autrement dit, pour une instruction de réception donnée, nous avons seulement besoin de considérer les émissions antérieures dans la trace pour en déduire sans ambiguïté la communication associée à cette réception.

Enfin nous utiliserons un système de marquages pour distinguer d'une part les communications « finies », c'est-à-dire celles pour lesquelles nous avons déjà déterminé l'émetteur et le récepteur. A l'inverse nous ne marquerons pas les communications « partielles », *i.e.* celles qui ont été créées par la rencontre d'une instruction d'émission et dont nous n'avons pas encore déterminé le récepteur.

Pour ce qui est des autres instructions apparaissant dans la trace, il s'agit essentiellement d'informations annexes sur le protocole (changements de phases, apparitions d'événements ...), et il suffit de les afficher telles quelles. L'algorithme 3.5 présenté en annexe résume la manière dont nous extrayons une trace Alice&Bob à partir d'une trace ProVerif.

2.4 Réalisation d'une interface graphique

Les trois outils précédemment décrits (compilateur, systèmes d'animation et de conversion de traces) ont été intégrés dans une interface graphique pour en rendre l'utilisation encore plus aisée. Celle-ci doit permettre à l'utilisateur de soumettre en entrée un protocole au format Alice&Bob (.ab) ou ProVerif (.pv), d'en effectuer la vérification et d'afficher une représentation graphique d'une trace d'attaque en cas de mise en défaut d'une propriété, et enfin de simuler le protocole en choisissant les envois de messages. Si le protocole entré est au format Alice&Bob, la phase de compilation est exécutée de manière transparente. Nous distinguons ainsi principalement deux modes de fonctionnement, l'un pour la vérification et l'autre pour l'animation. Dans les deux cas, l'objet principal présenté à l'utilisateur est une suite de communications entre agents, que nous représentons au moyen d'un diagramme de séquence de messages.

2.4.1 Choix de l'outil graphique

Plusieurs alternatives ont été envisagées pour produire l'interface graphique en question, dont deux ont retenu notre attention. La première est d'utiliser OCaml et LablGtk, la seconde de produire une interface JavaScript. Deux interfaces ont ainsi été développées en parallèle.

LablGtk est une API OCaml permettant de créer des interfaces graphiques en utilisant la bibliothèque GTK+. Bien que cette bibliothèque soit multiplateforme, son installation est complexe sur certaines systèmes d'exploitation (en particulier Windows et MacOSX). Il est possible de créer des paquets binaires contenant l'application et toutes les bibliothèques nécessaires pour ces systèmes, mais maintenir ces paquets risque de demander un travail important.

En utilisant le langage JavaScript, il est possible de créer et distribuer une interface multiplateforme et de s'affranchir de la plupart des difficultés liées à l'utilisation de LablGtk. Afin de pouvoir interagir avec le reste du programme, qui est codé en OCaml, nous avons choisi d'utiliser *js_of_ocaml* [11] qui est un compilateur de bytecode OCaml vers JavaScript. En pratique, l'application entière (c'est à dire à la fois ProVerif et nos modules supplémentaires) est compilée en un fichier Javascript qui est appelé au moyen d'une page HTML. Quelques fichiers supplémentaires ont été écrits directement en Javascript. Plusieurs librairies accompagnent `js_of_ocaml`, de manière à pouvoir programmer en OCaml de manière aisée les aspects propres à l'interface (manipulation du DOM, chargement de fichiers, gestion des évènements...) Pour tracer les diagramme de séquence, nous avons choisi d'utiliser la librairie KineticJs, qui permet de manipuler de manière aisée les canvas HTML5 [12].

3 Validation des outils

Dans cette section, nous présentons les méthodes que nous avons utilisées pour vérifier que nos outils avaient le comportement attendu.

3.1 Générer la trace Alice&Bob à partir du code ProVerif

Une première étape pour valider notre outil est d'utiliser la partie interactive (c'est-à-dire inférer toute les communications possibles à partir d'une spécification ProVerif) afin de générer les traces Alice&Bob couvertes par l'exécution du protocole de manière totalement automatisée. L'intérêt ici, est double : d'une part, cela permet de vérifier la manière dont nous inférons les communications à partir de ProVerif. D'autre part, cela permet de contrôler la partie compilation de notre projet dans le sens où, si l'utilisateur met en entrée un fichier écrit dans notre formalisme Alice&Bob, puis qu'il le compile en ProVerif, il pourra alors observer qu'au moins l'une des trace Alice&Bob reconstruite à partir du fichier ProVerif correspond bien à celle qu'il a placée en entrée. Pour faciliter la comparaison, nous garderons le formalisme de notre langage Alice&Bob pour les traces que nous extrairons de ProVerif, et nous appliquerons aussi la simplification de termes présentée à la sous-section 2.3.1. Nous affichons aussi les buts de secrets de messages récupérés grâce à l'étape d'analyse syntaxique initiale.

La difficulté principale de cette fonctionnalité est qu'une spécification ProVerif peut recouvrir une infinité de traces Alice&Bob. Pour limiter le nombre de choix nous avons choisi de supprimer l'intrus pour la génération de spécification Alice&Bob, ce qui est cohérent avec le fait que nous voulons une spécification Alice&Bob pour le protocole d'entrée, qui ne prend pas l'intrus en compte comme un rôle. Cet outil nous permet donc actuellement de générer toutes les spécifications Alice&Bob exécutables entre les agents valides du protocole. Plus précisément, une spécification est dite exécutable si, lorsqu'il n'y a plus de communication possible, il ne reste aucun agent valide de l'exécution dont le rôle n'est pas fini. Pour ce qui est de la construction de la trace en elle-même, nous réutilisons la construction de communication vue à la sous-section 2.2.2, et nous déroulons les choix possibles sous la forme d'un arbre d'exécution, en supprimant les branches correspondant à une spécification qui n'est pas exécutable. L'exécution se termine nécessairement, d'une part car nous ne parcourons qu'une et une seule fois chaque branche, et d'autre part car, en l'absence de l'intrus, le nombre de communications possibles est toujours borné par le rôle des agents honnêtes, déterminés par la spécification ProVerif (et donc limité par le nombre de lignes du fichier ProVerif).

```
Agents : network, B, A
Protocol :
network -> A (skB): pk(B)
A -> B (Na): {(Na,pk(A))}_pk(B)
B -> A (Nb): {(Na,Nb)}_pk(A)
A -> B: {Nb}_pk(B)
Goals :
secretBNb secret
secretBNa secret
secretANb secret
secretANa secret
```

Figure 11. Exemple de trace Alice&Bob générée pour le protocole de Needham-Schroeder rédigé en ProVerif.

3.2 Retrouver et reproduire des attaques

Nous nous sommes également intéressés à la conservation de la « puissance » de vérification de ProVerif par notre outil. Plus exactement nous avons vérifié sur un échantillon de protocoles que notre traitement sur la sortie de ProVerif correspondait bien à la sortie ProVerif d'origine, puis que l'attaque obtenue (si attaque il y a) était bien reproductible dans notre outil de simulation de spécification de ProVerif.

Cela nous permet donc de vérifier la validité des trois aspects suivants de notre travail.

- La partie compilation, dans le sens où si le protocole Alice&Bob a bien été traduit en ProVerif, alors la trace finale obtenue doit effectivement correspondre à une attaque sur le protocole.
- La partie interaction, puisque nous vérifions que la trace d'attaque est incluse dans les traces rejouables par la simulation.
- La partie sortie, pour s'assurer que les opérations de simplification que nous avons effectuées sur la trace n'ont pas changé sa sémantique.

Nous n'avons pas eu le temps de faire des tests complets pour beaucoup de protocoles, mais nous avons notamment testés entièrement (génération de traces Alice&Bob, simulation, simplification de la trace de

sortie, reproduction de l'attaque dans l'outil de simulation) nos outils sur les protocoles de Needham-Schroeder et de Diffie-Hellman disponibles dans la bibliothèque ProVerif. Pour ce qui est de la chaîne complète, comprenant l'étape de compilation, nous l'avons testé sur le protocole de Needham-Schroeder à partir de sa spécification disponible sur le site SPORE de l'ENS Cachan [13].

3.3 Exemple complet de notre outil

Dans cette section, nous présentons un exemple complet de l'utilisation combinée de nos outils dans la version textuelle de notre outil, par souci de place et de visibilité.

Compilation. Nous commençons d'abord par rédiger le protocole souhaité en Alice-Bob. Pour cet exemple, nous voulons simplement illustrer le fonctionnement général de notre outil, et nous avons donc choisi un exemple de protocole assez simple :

```
Agents : A, B
Import : pk
Protocol :
A -> B (Na): {Na}_pk(B)
B -> A (Nb): ({Na}_pk(A),{Nb}_sk(B))
Goals :
Na secret
Nb secret
```

Le protocole consiste simplement en deux échanges de nonces entre agents, avec utilisation de chiffrement par clé privée et par clé publique. Le secret du second nonce, `Nb`, devrait être compromis (car `Nb` peut être trouvé en déchiffrant `Nb_sk(B)`), alors que le nonce `Na`, lui, devrait effectivement être secret.

Génération de spécifications Alice&Bob. Nous lançons ensuite notre outil de simulation, qui, dans le cas où le fichier entré est, comme c'est le cas ici, en Alice&Bob, commence par appeler le compilateur afin de traduire la spécification en ProVerif (pour le lecteur intéressé, la spécification ProVerif produite est disponible en annexe B). L'étape suivante est la génération de trace(s) Alice&Bob à partir du code compilé, ce qui donne le résultat suivant :

```
-----------------Alice-Bob specs:
Agents:network,process_B,process_A

Protocol 1 :
network->process_A (skB):pk(B)
process_A->process_B (Na):{Na}_pk(B)
process_B->process_A (Nb):({Na}_pk(A
    ),{Nb}_skB)

Protocol 2 :
network->process_A:pk(B)
process_A->process_B (Na):{Na}_pk(B)
process_B->process_A (Nb):({Na}_pk(A
    ),{Nb}_skB)

Goals :
secretNa secret, secretNb secret
```

Comme escompté, les spécifications Alice&Bob renvoyées par le programme correspondent bien au protocole d'origine. Nous obtenons de plus les agents du protocole, ainsi que les buts de secret de messages (champs `Agents` et `Goals`) grâce à l'étape de parsing du fichier ProVerif (c.f. section 2.2.1). Notons que nous n'indiquons pas les buts d'authentification car ils sont plus durs à retrouver. Enfin, nous pouvons aussi observer que la simplification de termes évoquée à la sous-section 2.3.1 a été appliquée ici : par exemple le terme `paenc(Na,pk(skB))` dans le formalisme ProVerif a été transformé en `Na_pk(B)`.

Notons cependant que nous obtenons deux spécifications Alice&Bob, en apparence identiques. En réalité les deux échanges initiaux sont différents. En effet, dans le fichier ProVerif compilé, à l'initialisation, l'agent `network` contient deux instructions `out(c,pkB)` : l'une sert à donner la clé publique de `B` à l'intrus, alors que l'autre sert à donner à `A` l'identité de son interlocuteur. A cause de ces deux échanges syntaxiquement distincts, mais sémantiquement équivalents, notre outil retourne deux traces similaires, selon l'ordre dans lequel ces deux messages sont envoyés. Cette source d'imprécision peut être rapidement corrigée par l'utilisateur s'il le souhaite après la génération du code, en factorisant les deux envois `out(c,pk(B))` en un seul.

Simulation. Ensuite commence la partie simulation. Nous nous sommes rapidement aperçus que le rôle d'initialisation, `network` était assez fastidieux à simuler, car le plus souvent il s'agit d'envoyer des clés publiques à l'intrus. Pour simplifier l'utilisation de notre outil, nous avons donc automatisé les choix de l'« agent » `network` (quitte à ce que l'utilisateur revienne sur ses pas dans l'outil de simulation s'il n'est pas satisfait de ces choix). Pour cela, nous choisissons arbitrairement de récupérer les actions de `network` dans la première trace Alice&Bob retrouvée à l'étape

précédente de génération (s'il en existe) et de les forcer dans l'outil de simulation. Sur notre exemple, ceci donne le résultat suivant :

```
--Default chosen action for Network:
network -> process_A (skB) : pk(skB)
network -> I (skA) : pk(skA)
network -> I : pk(skB)
```

Une fois ces actions effectuées, la vraie simulation commence, et nous proposons des choix de communications à l'utilisateur sous la forme suivante :

```
--------Choose communication:
0 : process_A -> process_B (Na) :
    paenc(Na,pk(skB))
1 : process_A -> I (Na) : paenc(Na,
    pk(skB))
2 : I -> process_B : ?(user choice)
3 : I -> process_A : ?(user choice)
```

Une fois que l'utilisateur a choisi une communication, nous affichons les connaissances actuelles de l'intrus, puis les choix de communication suivants (notons que dans le cas d'une communication où l'intrus `I` est l'agent émetteur, c'est à l'utilisateur de rentrer le message à envoyer par `I`).

Sortie. Enfin, la dernière étape est la génération d'une trace de sortie simplifiée. Dans notre exemple, ProVerif trouve bien une attaque pour chacune des propriétés. Pour le secret du nonce `Nb`, l'attaque est triviale car l'intrus peut simplement l'obtenir en observant les échanges du protocole et en déchiffrant le terme `Nb_sk(B)` avec la clé publique `pk(B)` qui est dans sa connaissance. Pour le nonce `Na`, l'attaque est un peu plus complexe et repose sur le fait que l'intrus `I` se fasse passer pour `B` auprès de `A`. La trace correspondante est la suivante :

```
-----Corresponding Alice&Bob trace :
Step 1: network -> I (skA) : pk(A)
Step 2: network -> I (skB) : pk(B)
Step 3: network -> I : pk(B)
Step 4: I -> process_A : pk(a_2)
Step 5: process_A -> I (Na) : {Na}
    _pk(a_2)
Step 6: I -> process_A : (a,a_1)
Step 7: process_A -> I :
    secretNa_enc(secretNa,Na)
```

Nous affichons de plus comme informations complémentaires les connaissances initiales et finales de l'intrus. Sa connaissance initiale permet notamment d'observer les termes qu'il génère ; ici la clé secrète `a_2` et les nonces `a` et `a_1`.

Conclusion

Nous avons détaillé la conception de plusieurs outils visant à améliorer le confort des utilisateurs cherchant à développer des protocoles cryptographiques en utilisant ProVerif. Bien qu'incomplet sur certaines fonctionnalités un peu plus avancées (indéterminisme par exemple), nous avons testé avec réussite leur fonctionnement sur plusieurs protocoles. La diversité des outils proposés leur confère un intérêt non seulement pour le développeur expérimenté, mais également pour des utilisateurs plus novices pour qui l'utilisation du langage Alice&Bob peut permettre d'appréhender plus aisément les concepts fondamentaux de la vérification de protocoles. Notre outil de compilation permet en effet de faire le lien entre les deux langages de manière automatique. À celui-ci se rajoute notre outil de simulation qui manquait vraiment à ProVerif et permet une meilleur compréhension de la spécification du protocole. De plus, la création d'une interface graphique nous permet de représenter les communications de manière intuitive par des diagrammes de séquences.

Références

1. Véronique Cortier. Vérifier les protocoles cryptographiques. *TSI. Technique et science informatiques*, 24(1) :115–140, 2005.
2. Gavin Lowe. A hierarchy of authentication specifications. pages 31–43. IEEE Computer Society Press, 1997.
3. Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56 :131–133, 1995.
4. Proverif. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.
5. Scyther tool. http://users.ox.ac.uk/~coml0529/scyther/.
6. Avantssar : Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures. http://www.avantssar.eu/.
7. 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.
8. Olivier Heen, Thomas Genet, Stéphane Geller, and Nicolas Prigent. An industrial and academic joint experiment on automated verification of a security protocol. In *Mobile and Wireless Networks Security*, pages 39–53, 2008.
9. Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels. 5789 :337–354, 2009.

10. Ueli M Maurer and Pierre E Schmid. A calculus for security bootstrapping in distributed systems. *Journal of Computer Security*, 4 :55–80, 1996.
11. Jérôme Vouillon and Vincent Balat. From bytecode to javascript : the js_of_ocaml compiler. *Software : Practice and Experience*, 2013.
12. Kineticjs : an html5 canvas javascript framework. http://kineticjs.com/.
13. Spore : Security protocols open repository. http://www.lsv.ens-cachan.fr/Software/spore/.

Annexe A : Pseudo-codes

Inférence des communications entre agents

Algorithme 3.1 : Fonction `generate` (calcul des communications possibles)

```
1 tant que il reste des agents actifs dans le protocole faire
2     Soit communications = ∅ ;
3     pour chaque agent actif a faire
4         next_interesting_step(a) ;
5         Ajouter compute_communications(a) à communications ;
6     Afficher communications ;
7     c = choix de l'utilisateur ;
8     Apply_communication(c) ;
```

Algorithme 3.2 : Fonction `next_interesting_step` (traitement d'un rôle pas à pas)

```
  Entrées : Un agent a de rôle r
1 pour chaque instruction i ∈ r faire
2     cas où i = Vide
3         Marquer l'agent a comme inactif ;
4         break ;
5     cas où i = In(c,m) ou i = P1|P2
6         break ;
7     cas où i = Out(c,m)
8         Ajouter m au canal c
9     cas où i = Event(e)
10        Émettre signal : événement e par agent a
```

Algorithme 3.3 : Fonction `compute_communications(a)` (calcul des communications possibles dont l'agent a est récepteur)

Entrées : Un agent `a`, dont la première instruction dans le rôle est une réception de message

Soit $In(c, pat)$ première ligne du rôle de `a` ;
pour chaque *message m présent sur le canal c* **faire**
  si *m correspond au motif pat* **alors**
    Soit `X` l'agent émetteur de m ;
    Construire la communication `a` $\xrightarrow{c}$ `X : m` ;

Algorithme 3.4 : Fonction `apply_communication(c)` (actions réalisées lorsque l'utilisateur choisit la communication `c` dans la simulation)

Entrées : Une communication `A` $\xrightarrow{c}$ `B : m`

Ajouter m à la connaissance de `B` ; **si** `c` *canal public* **alors**
  Ajouter m à la connaissance de l'intrus ;
Retirer m du canal `c` ;
Décrémenter le rôle de `B` d'une instruction ;

Retrouver la trace Alice&Bob à partir d'une trace ProVerif

Algorithme 3.5 : Retrouver la trace ProVerif sous forme Alice&Bob

Entrées : Une trace ProVerif, T
Sorties : `résultat`, la trace sous forme Alice&Bob. (type `animation list`)

`résultat` = [] ;
pour chaque *instruction* $i \in T$ **faire**
  soit `X` l'agent associé à i ;
  si i = *`ROutput(c, m)`* **alors**
    soit `comm` = Communication(`X` $\xrightarrow{c}$ `I : m`) ;
    ajouter `comm` à `résultat`
  sinon si i = *`RInput(c, m)`* **alors**
    si $\exists$`comm` $\in$ `resultat` *non marquée correspondant à* i **alors**
      Remplacer `I` par `X` dans comm ;
      Marquer comm ;
    sinon
      soit `comm` = Communication(`I` $\xrightarrow{c}$ `X : m`) ;
      ajouter `comm` à `résultat`
  sinon
    ajouter i à `résultat`

Annexe B : Sortie ProVerif de la section 3.3

```
free c : channel.

(* Import of pk library. *)
(* Secret/Public keys definition *)

type pkey. (* public key *)
type skey. (* secret key *)

fun pk(skey) : pkey. (* Public key
   corresponding to a secret key*)
fun saenc(bitstring, skey) :
   bitstring. (* Encoding messages
   with a secret key *)
reduc forall m : bitstring, k : skey
   ; sadec(saenc(m, k), pk(k)) = m.
   (* Decoding messages with a
   public key *)
fun skey_to_bitstring(skey) :
   bitstring [data, typeConverter].

fun paenc(bitstring, pkey) :
   bitstring. (* Encoding messages
   with a public key *)
reduc forall m : bitstring, k : skey
   ; padec(paenc(m, pk(k)), k) = m.
   (* Decoding messages with a
   secret key *)
fun pkey_to_bitstring(pkey) :
   bitstring [data, typeConverter].


fun secretNa_enc(bitstring,
   bitstring) : bitstring.
reduc forall m : bitstring, m' :
   bitstring; secretNa_dec(
   secretNa_enc(m, m'), m') = m.
free secretNa : bitstring[private].
query attacker (secretNa).

fun secretNb_enc(bitstring,
   bitstring) : bitstring.
reduc forall m : bitstring, m' :
   bitstring; secretNb_dec(
   secretNb_enc(m, m'), m') = m.
free secretNb : bitstring[private].
query attacker (secretNb).

(* Start of Protocol *)
```

```
let process_A(pkA : pkey, pkB : pkey
    , skA : skey) =
in(c, pk0 : pkey);
new Na : bitstring;
out(c, paenc(Na, pk0));
in(c, (
    EncryptedMessageWithPublicKeyOfA1_known_by_A
     : bitstring,
    EncryptedMessageWithSecretKeyOfB2_known_by_A
     : bitstring));
out(c, secretNa_enc(secretNa, Na)).

let process_B(pkA : pkey, pkB : pkey
    , skB : skey) =
in(c, m3 : bitstring);
let (Na_known_by_B : bitstring) =
    padec(m3, skB) in
new Nb : bitstring;
out(c, (paenc(Na_known_by_B, pkA),
    saenc(Nb, skB)));
out(c, secretNb_enc(secretNb, Nb)).

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