

Traduction de spécifications en implémentations protocoles

David Cadé, Directeur de stage : Bruno Blanchet, CNRS, ENS, INRIA

23 août 2009

Le contexte général

Il s'agit de prouver la sécurité d'un protocole cryptographique. Montrer qu'un attaquant n'a pas de moyens de contourner un protocole pour obtenir des secrets ne peut pas être fait par tests, c'est pourquoi une preuve est nécessaire.

Il y a essentiellement deux modèles, le modèle de Dolev-Yao, qui est un modèle formel qui raisonne sur une algèbre de termes, et le modèle calculatoire, plus bas niveau, qui raisonne sur les chaînes de bits. Il y a beaucoup de résultats liant ces deux modèles, prouvant qu'une preuve dans le modèle formel avec des hypothèses restrictives est équivalente à une preuve dans le modèle calculatoire.

Il existe aussi actuellement, ProVerif, prouveur automatique dans le modèle formel, et CryptoVerif, prouveur automatique dans le modèle calculatoire, tous deux écrits par Bruno Blanchet. Il existe aussi d'autres vérificateurs comme AVISPA, développé par de nombreux laboratoires, Scyther, développé par Cas Cremers, et encore bien d'autres.

Beaucoup de travaux du domaine montrent la correction de protocoles existants.

Le problème étudié

Même lorsqu'on a une spécification d'un protocole prouvée correcte, l'implémentation peut renfermer des erreurs. C'est pourquoi on veut obtenir une implémentation prouvée dans un langage de programmation traditionnel. On peut répondre à cette question de plusieurs façons, soit en prouvant que l'implémentation actuelle est correcte, soit en en construisant une. Nous nous sommes penchés sur la deuxième possibilité.

Ce n'est pas un nouveau problème, des solutions ont été proposées pour ces deux approches. L'originalité de mon approche est l'obtention d'une implémentation prouvée dans le modèle calculatoire par traduction de la spécification.

La contribution proposée

Ma solution est d'écrire un compilateur qui prend en entrée une spécification CryptoVerif, et retourne une implémentation en OCaml correspondant à la spécification, et ensuite de prouver la validité du compilateur.

Les oracles sont des fonctions que l'attaquant peut appeler pour faire faire une partie du protocole à un agent, qui utilise ses secrets. L'attaquant n'a pas accès à ceux-ci. Pour pouvoir obtenir une implémentation, j'ai dû ajouter quelques structures au langage d'entrée. On obtient alors des modules qui implémentent les oracles, qu'il faut coupler avec le code des primitives cryptographiques et le code qui envoie les résultats des oracles sur le réseau. Le code réseau n'a pas besoin d'être vérifié car il peut être considéré comme faisant partie de l'attaquant, alors qu'on a besoin de vérifier que les primitives vérifient bien toutes les suppositions faites dans la spécification.

Les arguments en faveur de sa validité

Notre approche a des avantages par rapport à l'approche qui consiste à vérifier une implémentation : on part d'un langage d'entrée plus contraint, il y a donc moins de détails à prendre en compte. De plus, en pratique, on écrit d'abord une spécification informelle, puis on l'implémente, et ensuite on les corrige petit à petit. Notre approche permet d'éviter ces corrections.

J'ai testé le compilateur sur des exemples de protocoles avec succès. Pour que la solution soit valide, il faut faire plusieurs hypothèses, comme le fait de pouvoir s'échanger les données secrètes entre les modules sans que l'attaquant s'en rende compte.

Le bilan et les perspectives

Cette approche permet d'obtenir une implémentation prouvée de toute spécification de protocole que l'on peut prouver avec CryptoVerif, modulo la correction de l'implémentation des primitives cryptographiques.

Il faut terminer la preuve de correction du compilateur. Ensuite, on peut faire des études de cas plus ambitieuses, comme essayer d'obtenir une implémentation pour un protocole utilisé en pratique comme TLS ou Kerberos, puis essayer d'interagir avec une des implémentations déjà écrites.

Traduction de spécifications en implémentations protocoles

David Cadé, Directeur de stage : Bruno Blanchet, CNRS, ENS, INRIA

23 août 2009

Table des matières

1	Description du langage d'entrée de CryptoVerif	7
1.1	Syntaxe du calcul de processus	8
1.1.1	Termes	9
1.1.2	Définitions d'oracles	9
1.1.3	Corps d'oracles	10
1.2	Exemple	11
2	Annotations	12
2.1	Annotations sur le processus	12
2.1.1	Découpage en programmes	12
2.1.2	Fichiers de données externes	13
2.1.3	Hypothèses supplémentaires	14
2.2	Autres annotations	14
2.2.1	Taille des types	14
2.2.2	Fonctions	15
3	Génération de code	15
3.1	Interface d'un programme	15
3.1.1	Le module <code>Crypto</code>	16
3.1.2	La fonction d'initialisation	16
3.1.3	Les oracles	16
3.2	Implémentation	18
3.2.1	L'état du module	18
3.2.2	Implémentation d'un oracle	19
A	La fonction de traduction d'un corps d'oracle	21

Introduction

Notre but est d'obtenir une implémentation sûre dans le modèle calculatoire d'un protocole cryptographique. Pour cela, nous avons développé un compilateur prenant en entrée une spécification d'un protocole cryptographique écrite dans le langage d'entrée de CryptoVerif et retournant une implémentation en OCaml correspondant à cette spécification.

Contexte Pour être sûr qu'un protocole se comporte bien, il ne suffit pas de le tester et de vérifier que les messages soient bien envoyés. Il faut aussi s'assurer qu'un attaquant n'a pas de moyen de contourner le protocole pour obtenir les secrets des utilisateurs. Ce genre de résultats ne peut pas être obtenu par tests, c'est pourquoi il faut obtenir une preuve de la correction du protocole.

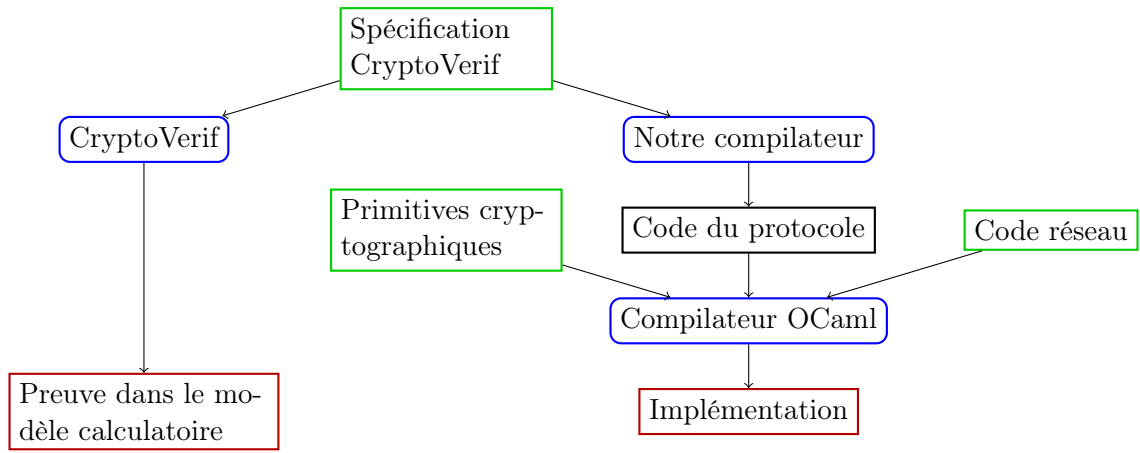
Il y a essentiellement deux modèles :

- le modèle de Dolev-Yao [DY83], qui est un modèle formel simple qui raisonne sur une algèbre de termes. Les messages sont des termes de cette algèbre, et les primitives cryptographiques sont des boîtes noires, des symboles de l'algèbre. L'attaquant peut uniquement calculer des termes dans l'algèbre, et donc il ne peut utiliser que les primitives cryptographiques définies dans celle-ci.
- Le modèle calculatoire, plus bas niveau, qui raisonne sur les chaînes de bits. Chaque message est une chaîne de bits, et les primitives cryptographiques sont des fonctions polynomiales sur ces chaînes de bits. L'attaquant est une machine de Turing polynomiale probabiliste. Ce modèle est plus réaliste que le modèle formel.

Il y a beaucoup de résultats qui essaient de relier ces deux modèles, comme :

- le résultat d'Abadi et Rogaway [AR00] qui montre que, dans le cas du chiffrement symétrique avec quelques hypothèses supplémentaires, lorsqu'on a une preuve dans le modèle formel, on obtient une preuve dans le modèle calculatoire avec un adversaire passif.
- le résultat de Cortier et Warinschi [CW05] qui montre que la correspondance entre les traces dans les deux modèles dans le cas des protocoles à clé publique, et en présence d'un attaquant actif.
- le résultat de Comon-Lundh et Cortier [CLC08] sur l'équivalence observationnelle.

Il existe aussi actuellement, ProVerif [Bla09], prouveur automatique dans le modèle de Dolev-Yao, et CryptoVerif [Bla08], prouveur automatique dans le modèle calculatoire, tous deux écrits par Bruno Blanchet. Il existe aussi



Légende : Outil Entrée Sortie

FIGURE 1 – Les différents outils utilisés et leurs entrées et sorties

d'autres vérificateurs comme AVISPA, développé par de nombreux laboratoires, et qui contient un module prouvant les protocoles dans le modèle calculatoire [CHW06], Scyther [Cre06], développé par Cas Cremers, et encore bien d'autres.

Vision d'ensemble La figure 1 décrit l'ensemble des outils et des entrées requises pour obtenir une implémentation prouvée avec l'outil que l'on développe. La spécification CryptoVerif est un fichier écrit dans le langage d'entrée de CryptoVerif. On utilise ensuite CryptoVerif qui nous donne une preuve de correction dans le modèle calculatoire de la spécification en entrée. On peut ensuite utiliser notre compilateur après avoir annoté la spécification pour obtenir le code du protocole que l'on couple avec le code pour les primitives cryptographiques et le code réseau pour obtenir une implémentation.

La preuve dans le modèle calculatoire CryptoVerif [Bla08] est un vérificateur de protocoles dans le modèle calculatoire qui peut prouver automatiquement des propriétés de protocoles, tels que le secret et l'authentification. Il génère des preuves par suites de jeux, telles que celles écrites manuellement par des cryptographes. Le premier jeu décrit la spécification du protocole à prouver ; les jeux suivants sont déduits par les hypothèses sur les primitives ou bien par transformation syntaxique du jeu. Ces transformations sont

telles que deux jeux consécutifs soient indistinguables, et dans le dernier jeu, la propriété de sécurité est évidente. On a donc que le premier jeu satisfait lui aussi cette propriété. Ces jeux sont formalisés dans un calcul de processus probabiliste en temps polynomial.

On obtient alors une preuve dans le modèle calculatoire que le protocole est correct.

L'implémentation Pour obtenir une implémentation sûre, on commence par annoter la spécification en ajoutant des indications à propos des détails d'implémentation. L'utilisateur indique comment la spécification est coupée en modules (par exemple, la partie gérant la génération de clés, celle gérant le client et celle gérant le serveur), les fichiers qui vont stocker les données partagées entre plusieurs modules, le nom des fonctions OCaml qui correspondent aux primitives cryptographiques, et la taille des nombres aléatoires. Le compilateur génère le code pour chaque module.

Ensuite on couple ces modules avec le code implémentant les primitives cryptographiques contenant toutes les fonctions OCaml déclarées, et le code réseau, qui envoie les messages obtenus par les fonctions des modules aux différents participants du protocole. Le code réseau n'a pas besoin d'être prouvé correct car on peut considérer qu'il fait partie de l'attaquant, mais on doit prouver que les primitives cryptographiques, elles, respectent les hypothèses indiquées dans la spécification.

Travaux en relation Il y a eu beaucoup de travaux qui sont reliés à ce travail. On peut citer, tout d'abord, le travail de Bruno Blanchet sur CryptoVerif [Bla08] qui m'a permis de développer cet outil.

Notre approche d'obtenir une implémentation correcte en créant un compilateur prenant une spécification d'un protocole cryptographique en entrée et qui renvoie une implémentation n'est pas nouvelle. Cette approche a aussi été utilisée dans des outils comme ceux de [SBP01, Mil02, PSD04]. Notre avantage par rapport à ces travaux est que l'on travaille dans le modèle calculatoire.

L'approche inverse, qui consiste à prendre une implémentation d'un protocole cryptographique et d'en déduire une spécification formelle, et de prouver ensuite la correction de celle-ci a eu beaucoup de succès. Par exemple, le travail de Goubault-Larrecq et Parrennes [GLP05] qui obtient d'un protocole écrit en C une abstraction de celui-ci en clauses de Horn, utilisées ensuite pour prouver des propriétés sur le protocole. Le travail de Bhargavan et al. [BFGT06] transforme du code écrit en F# en une spécification

ProVerif [Bla09], prouveur dans le modèle formel. Ils ont ensuite adapté leur outil [BCF07, BCFZ09] pour qu'il renvoie une spécification CryptoVerif. Son application au protocole TLS [BCFZ08] prouve une bonne partie du protocole avec ProVerif et une partie plus réduite avec CryptoVerif. Bengtson et al. [BBF⁺08] ont proposé un système de types pour prouver des propriétés de sécurité.

Dans le domaine de la vérification dans le modèle calculatoire, on peut citer les travaux de Tšahhrov et Laud [TL07], qui analyse statiquement la propriété de confidentialité dans des protocoles cryptographiques.

Plan Tout d'abord, dans la première partie, nous allons aborder dans quelle syntaxe est écrite une spécification de protocole cryptographique dans le langage d'entrée de CryptoVerif, puis dans une deuxième partie, quelles sont les annotations requises pour pouvoir utiliser notre compilateur et pourquoi ces annotations sont requises et enfin, dans la dernière partie, nous allons expliquer plus en détail comment le code est généré. En annexe, nous expliquons la génération du corps d'un oracle en détail et les deux dernières pages du rapport sont le résumé de ma présentation au workshop FCC'09.

1 Description du langage d'entrée de CryptoVerif

CryptoVerif possède deux langages d'entrée différents qu'il transforme dans une représentation interne appropriée, qui est la syntaxe des jeux dans un calcul de processus.

Canaux. Ce langage d'entrée est très proche de la représentation interne. C'est une variante du pi calcul, proche du langage traditionnellement utilisé pour la vérification formelle de protocoles.

Oracles. Ce langage d'entrée est presque le même que celui utilisant les canaux, sauf que lire une variable sur un canal est représenté comme la définition d'un oracle prenant cette variable en paramètre, et écrire sur un canal une variable comme le retour de cette variable à l'appelant. Ce langage est plus proche de celui utilisé par les cryptographes dans leurs preuves manuelles.

Pour développer l'outil, nous avons décidé d'utiliser celui utilisant les oracles. Le résultat obtenu par l'outil est un ensemble de fonctions qui, chacune, décrivent un oracle. C'eût été moins naturel d'utiliser le langage des canaux, où on aurait eu une fonction par canal de lecture, et cette fonction renvoyant le résultat écrit sur le premier canal rencontré.

$M, N ::=$	termes
i	indice de réplication
$x[M_1, \dots, M_m]$	accès à une variable
$f(M_1, \dots, M_m)$	application d'une fonction
$Q ::=$	définitions d'oracles
0	processus nul
$Q \mid Q'$	composition parallèle
foreach $i \leq n$ do Q	réplication n fois
$O[M_1, \dots, M_l](x_1[\tilde{i}] : T_1, \dots, x_k[\tilde{i}] : T_k) := P$	définition d'un oracle
$P ::=$	corps d'un oracle
return(N_1, \dots, N_k); Q	retour
end	fin
$x[i_1, \dots, i_m] \stackrel{R}{\leftarrow} T; P$	nombre aléatoire
$x[i_1, \dots, i_m] : T \leftarrow M; P$	affectation
let $x[i_1, \dots, i_m] : T = M$ in P	affectation
if defined(M_1, \dots, M_l) \wedge M then P else P'	condition
find ($\bigoplus_{j=1}^m u_{j1}[\tilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\tilde{i}] \leq n_{jm_j}$ suchthat defined($M_{j_1}, \dots, M_{j_l_j}$) \wedge M_j then P_j else P)	recherche dans un tableau

FIGURE 2 – Syntaxe simplifiée du langage d'entrée

1.1 Syntaxe du calcul de processus

On note \tilde{i} une liste d'indices de réplication i_1, \dots, i_n .

Dans le fichier d'entrée, il nous faut indiquer le processus décrivant quels sont les oracles et dans quel ordre l'attaquant a le droit de les exécuter. Mais ce n'est pas la seule chose qu'il faut mettre dans ce fichier d'entrée. On doit aussi y définir :

- les types utilisés. Un type définit un ensemble de valeurs que peuvent prendre les variables de ce type. Il y a des types prédéfinis, comme le type *bitstring* qui est le type contenant toutes les chaînes de bits possibles, ou le type *bitstring*_⊥ qui est le type contenant toutes les chaînes de bits possibles ou bien ⊥, valeur souvent utilisée pour décrire l'échec d'un déchiffrement. On a que pour tout type t , les valeurs de

ce type sont des valeurs du type $bitstring_{\perp}$. Le type booléen $bool$ est un type prédéfini qui peut prendre comme valeur $true$ ou bien $false$. Si l'on définit un nouveau type, il faut indiquer quelles propriétés ce type a, dans les propriétés suivantes :

- **bounded** : les valeurs du type sont bornées.
- **large** : l'ensemble des valeurs du type peuvent être suffisamment grandes, c'est-à-dire que la probabilité de collision lorsqu'on choisit un élément de ce type au hasard avec un autre élément de ce type est négligeable.
- **fixed** : les valeurs de ce type peuvent être choisies au hasard. On suppose que l'on peut choisir un bit au hasard, et donc on peut choisir au hasard un élément dans tout ensemble de la forme $[0, 2^n - 1]$. Pour l'implémentation, on suppose qu'un type **fixed** a ses valeurs dans un intervalle de cette forme.
- les primitives cryptographiques et les différentes hypothèses sur celles-ci : les hypothèses fonctionnelles (par exemple, déchiffrer un message chiffré donne le clair), et les hypothèses de sécurité (par exemple, le chiffrement est IND-CPA).

Décrivons maintenant la syntaxe du processus, résumée dans la figure 2.

1.1.1 Termes

Les termes représentent les variables utilisées dans le processus.

- À un point donné du processus, les indices de réplication sont les indices des boucles `foreach` qui sont utilisées pour arriver à ce point.
- Une variable est en fait un tableau qui prend les indices de réplication courants lors de sa définition comme paramètre. Ceci permet d'avoir des variables locales à un oracle différentes pour chaque appel de cet oracle. Donc, pour accéder à une variable, il faut indiquer de laquelle il s'agit en indiquant les indices de réplication sous lesquels elle a été définie. Si on est dans la portée de la variable, on peut ne pas indiquer ces indices, et les indices courants seront utilisés.
- L'appel à une fonction est utilisé pour appeler les primitives cryptographiques. Ces symboles de fonctions désignent des fonctions déterministes et polynomiales.

1.1.2 Définitions d'oracles

Les définitions d'oracles contrôlent dans quel ordre et comment l'attaquant peut appeler les oracles.

- Le processus nul, 0, est le processus « deadlock ». Dans cet état, l’attaquant ne peut appeler aucun oracle.
- La composition de deux processus, $Q \mid Q'$, indique que l’attaquant peut appeler tous les processus qu’il peut appeler dans Q et tous ceux qu’il peut appeler dans Q' .
- La réplication d’un processus n fois, `foreach $i \leq n$ do Q` , permet à l’attaquant d’appeler tous les processus qu’il peut appeler dans Q n fois, pour l’indice de réplication i allant de 1 à n .
- L’entrée, qui est en fait la définition d’un oracle, permet à l’attaquant d’appeler le corps de l’oracle en argument.

1.1.3 Corps d’oracles

Les corps d’oracles définissent ce que font les oracles.

- La construction de retour retourne à l’attaquant le contenu des variables retournées, et ensuite le processus continue sur la définition d’oracles qui suit.
- La construction `end` rend le contrôle à l’attaquant.
- La construction de génération d’un nombre aléatoire génère un nombre aléatoire, puis l’affecte à la variable et ensuite continue en exécutant le processus qui suit.
- Les constructions d’affectation affectent à la variable la valeur du terme, puis continue en exécutant le processus qui suit. La construction utilisant `let` a été introduite dans le langage pour permettre le pattern-matching.
- La construction `if defined(M_1, \dots, M_l) \wedge M then P else P'` regarde si les variables M_1 à M_l sont définies et si M vaut true, et dans ce cas là, exécute le processus P , sinon exécute P' . M doit donc être de type *bool*.
- La construction

$$\begin{aligned} & \text{find } (\bigoplus_{j=1}^m u_{j1}[\tilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\tilde{i}] \leq n_{jm_j} \\ & \quad \text{suchthat defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j \text{ then } P_j) \\ & \quad \text{else } P \end{aligned}$$

permet de chercher dans les variables. On teste sur chaque branche j du `find` en regardant s’il existe un m_j -uplet $(u_{j1}[\tilde{i}], \dots, u_{jm_j}[\tilde{i}])$ tel que les variables M_{j1} à M_{jl_j} sont définies et la variable M_j vaut true. S’il n’existe aucun de ces m_j -uplets, alors on exécute P , sinon on choisit au hasard un m_j -uplet dans les m_j -uplets qui vérifient les conditions,

$$A \longrightarrow B : \underbrace{\{k\}_{Kab}}_e, \text{mac}(e, mKab)$$

```

let processA = OA() :=
  k  $\xleftarrow{R}$  key; s  $\xleftarrow{R}$  seed;
  e  $\leftarrow$  enc(keyToBitstring(k), Kab, s);
  return (e, mac(e, mKab)).
let processB = OB(e : bitstring, m : macs) :=
  if check(e, mKab, m) then
  let injbot(keyToBitstring(k : key)) = dec(e, Kab) in
  return ().
process Ostart () :=
  rKab  $\xleftarrow{R}$  keyseed; Kab  $\leftarrow$  kgen(rKab);
  rmKab  $\xleftarrow{R}$  mkeyseed; mKab  $\leftarrow$  mkgen(rmKab);
  return ();
  ( foreach i1  $\leq$  N do processA |
    foreach i2  $\leq$  N do processB)

```

FIGURE 3 – Spécification d’un protocole cryptographique dans la syntaxe du backend ‘oracle’ de CryptoVerif

et on exécute P_j . Cette construction peut être utilisée pour chercher une clé dans une table de clés.

1.2 Exemple

La figure 3 montre un exemple de processus qui décrit le protocole où Alice (A) envoie à Bob (B) un message contenant le chiffré d’une clé par une clé symétrique entre Alice et Bob et le MAC de ce chiffré par une autre clé symétrique entre Alice et Bob.

On peut donner un nom à un processus en utilisant le mot clé `let`, et le processus commence par le mot clé `process`.

L’oracle `Ostart` est le seul oracle que l’on peut exécuter au début ; après l’avoir exécuté, on peut exécuter N fois les oracles `OA` et `OB`.

L’oracle `Ostart` sert à générer les clés des deux participants au protocole. Il commence par obtenir la clé `Kab` en générant aléatoirement une graine de

clé `rKab`, puis en utilisant la fonction de génération de clé `kgen` sur cette graine. Puis il obtient de la même manière la clé `mKab`. La clé `Kab` sert à chiffrer la clé que Alice envoie à Bob dans le protocole, et la clé `mKab` sert à obtenir le MAC de ce chiffré. Ces clés générées ici ne peuvent pas être connues de l'attaquant.

L'oracle `OA` représente le rôle d'Alice. Il génère aléatoirement une nouvelle clé `k` et un sel `s`, puis calcule le chiffré de cette clé `k` avec la clé `Kab` et le sel `s`. Il renvoie ensuite `e` et son MAC avec la clé `mKab`.

L'oracle `OB` représente le rôle de Bob. Son rôle ici ne sert qu'à vérifier que le message reçu est bien formé. Il commence par vérifier que la deuxième partie du message est bien le MAC sous la clé `mKab` de la première partie du message, puis il déchiffre la première partie du message avec la clé `Kab`.

Un détail que l'on peut observer est que dans l'oracle `OB`, on ne définit pas ce qui se passe dans la branche `else` du `if`. Dans les cas où on ne définit pas le processus sur lequel on continue dans une des branches d'un `if` ou d'un `find`, ce processus est `end` par défaut. Ceci est fait pour que le protocole ne continue pas si un test de vérification échoue.

On peut aussi remarquer les fonctions `injbot` et `keyToBitstring` utilisées dans la partie gauche du `let`. Ces fonctions sont injectives, et leur inverse est polynomial. Les utiliser à gauche est équivalent à utiliser à droite la fonction inverse. On appelle ces fonctions des fonctions décomposables.

2 Annotations

Dans cette partie, je vais présenter les modifications de la syntaxe d'entrée faites pour pouvoir obtenir une implémentation des oracles.

2.1 Annotations sur le processus

Dans la figure 4, on peut voir les annotations nécessaires pour obtenir une implémentation de l'exemple présenté dans la figure 3.

2.1.1 Découpage en programmes

On découpe essentiellement le processus en utilisant des accolades (`{,}`) en plusieurs programmes, chacun implémentant les oracles présents entre les accolades.

On a découpé l'exemple en trois programmes :

keygen : Ce programme ne contient que l'oracle `Ostart` qui génère les clés symétriques.

```

let processA = pA { OA () :=
    k  $\stackrel{R}{\leftarrow}$  key; s  $\stackrel{R}{\leftarrow}$  seed;
    e  $\leftarrow$  enc(keyToBitstring(k), Kab, s);
    return (e, mac(e, mKab))}.
let processB = pB { OB (e : bitstring, m : macs) :=
    if check(e, mKab, m) then
    let injbot(keyToBitstring(k : key)) = dec(e, Kab) in
    return ()}.
process keygen [Kab > Kab, mKab > mKab] { Ostart () :=
    rKab  $\stackrel{R}{\leftarrow}$  keyseed; Kab  $\leftarrow$  kgen(rKab);
    rmKab  $\stackrel{R}{\leftarrow}$  mkeyseed; mKab  $\leftarrow$  mkgen(rmKab);
    return ()};
( foreach i1  $\leq$  N do processA |
  foreach i2  $\leq$  N do processB)

```

FIGURE 4 – Annotations

pA : Ce programme contient tous les oracles utilisés par le rôle d’Alice, c’est-à-dire l’oracle OA.

pB : Ce programme contient tous les oracles utilisés par le rôle de Bob, c’est-à-dire l’oracle OB.

On a couramment un découpage des protocoles en programmes de cette manière, avec trois programmes séparés pour le générateur de clés, le client et le serveur.

Une accolade ouvrante ne peut être placée qu’avant le début d’une définition d’oracles, et une accolade fermante qu’après un `return(...)`. Il y a une accolade fermante implicite après un `end` et après un `return(...)` suivi par le processus 0, le processus étant terminé après cela. Pour une accolade ouvrante donnée il peut y avoir plusieurs accolades fermantes : en effet, après un `if`, on a deux branches du processus qui peuvent tous les deux avoir une accolade.

2.1.2 Fichiers de données externes

Toujours sur l’exemple 4, dans le programme `keygen`, on crée les clés de chiffrement et de MAC symétriques qui sont utilisés dans les programmes `pA` et `pB`. On doit donc pouvoir communiquer entre programmes, et pour ce faire, on utilise des fichiers.

Les programmes `pA` et `pB` ont besoin des clés `Kab` et `mKab` pour pouvoir être exécutés, et ces clés sont produites par le programme `Ostart`.

Les options entre crochets après le nom du programme indiquent dans quel fichier sauvegarder une variable définie dans celui-ci.

Ensuite tous les programmes utilisant une variable sauvegardée dans un fichier par un autre programme liront ce fichier et obtiendront le contenu de la variable. On peut aussi donner en option un fichier duquel lire une variable (`[k < file]`). Cela peut servir si le code générant les variables n'était pas dans un programme.

On peut, de la même manière, sauvegarder une table de clés dans un fichier avec la syntaxe `[n,k > t file]`, avec `n` et `k` deux variables définies dans le même oracle, sous réplification.

2.1.3 Hypothèses supplémentaires

Découper le processus en plusieurs programmes séparés est une chose que l'on veut avoir, ne serait-ce que pour pouvoir avoir la séparation génération de clés/client/serveur qui est courante.

Mais on doit supposer que l'attaquant ne peut pas lire les fichiers que les programmes écrivent. En effet, les variables écrites sont souvent internes au processus et pas directement accessibles à l'attaquant. Lorsqu'on a généré une clé symétrique comme dans l'exemple utilisé auparavant, le transport du fichier de clés aux deux participants ne fait pas partie du protocole.

En général, il faut aussi supposer que l'attaquant appelle les programmes dans le bon ordre, et ne ré-exécute pas un programme qu'il ne doit pas ré-exécuter. Ceci peut être problématique, mais on a tout de même que la sécurité d'un processus entraîne la sécurité du processus sous réplification, c'est-à-dire que l'on peut avoir plusieurs instances du protocole en parallèle. Donc, dans cet exemple, l'attaquant peut exécuter dans n'importe quel ordre les programmes après avoir exécuté une fois le générateur de clés.

2.2 Autres annotations

Dans cette partie, nous allons décrire les autres annotations requises pour pouvoir obtenir une implémentation.

2.2.1 Taille des types

Dans un oracle, on peut générer un nombre aléatoire d'un type t en utilisant la construction $x \stackrel{R}{\leftarrow} t$ lorsque t est de type `fixed`. Pour que dans l'im-

```
open Crypto
val init : unit -> unit
val oracle_OA : unit -> bitstring
```

FIGURE 5 – Interface du module PA

plémentation, on puisse générer ce nombre aléatoire, il nous faut connaître la taille du type.

On l'indique dans la spécification en écrivant après la définition du type et avant toute utilisation de celui-ci une ligne de cette forme :

```
implementation typesize keyseed 256.
```

Ceci signifie que les éléments du type `keyseed` sont des chaînes de bits de taille 256 bits.

2.2.2 Fonctions

Il nous faut aussi indiquer quelles fonctions OCaml représentent les primitives cryptographiques utilisées. Cela se fait de cette manière :

```
implementation function check "check".
```

Ceci signifie que la primitive `check` est implémentée par une fonction OCaml de nom `check`. Lorsque l'on a une fonction décomposable on indique les fonctions de composition et décomposition.

Pour obtenir une implémentation sûre, il faut prouver que les implémentations des fonctions respectent toutes les hypothèses que l'on a faites sur elles dans la spécification.

3 Génération de code

Dans cette partie, nous allons voir quel est le code et comment ce code est généré par notre compilateur.

3.1 Interface d'un programme

Pour chaque programme de la spécification, on crée un module OCaml qui implémente tous les oracles contenus dans le programme. On expliquera en détail le contenu de ce module dans la suite.

L'interface du module généré pour le programme `pA` de l'exemple est présenté dans la figure 5.

3.1.1 Le module `Crypto`

Chacun de ces modules ouvre le module `Crypto` qui contient la définition du type `bitstring`. Ce type correspond au type `bitstring⊥` qui englobe toutes les chaînes de bit possibles de `CryptoVerif`. Ce type est un alias du type `(string * int) option`. Le symbole \perp est encodé par `None`, et une chaîne de bits x est encodée par `Some (v,n)` avec v la chaîne de caractères qui correspond à cette chaîne de bits en y ajoutant si nécessaire des bits 0 de padding à la fin pour terminer sur les limites d'un octet. n est le nombre de bits. Toutes les variables dans le code sont de ce type. Comme `CryptoVerif` s'occupe du typage et de la correction du protocole, on n'a pas besoin de créer un type OCaml par type `CryptoVerif`. On trouve aussi dans ce module les fonctions qui écrivent le contenu d'une variable dans les fichiers, les fonctions utilitaires qui servent à l'intérieur du code des oracles, comme celles servant à grouper plusieurs `bitstring` en une seule ou l'opération inverse.

3.1.2 La fonction d'initialisation

On trouve dans chacun des modules générés une fonction `init` qui initialise l'état du module. Cette fonction ne prend pas d'arguments. On parlera plus en détail de l'état d'un module dans les parties suivantes.

3.1.3 Les oracles

Le prototype de la fonction implémentant un oracle donné dépend d'où il se trouve dans le processus par rapport aux réplifications le précédant. Si un oracle est sous réplification, alors cet oracle doit prendre en argument les indices de réplification en plus des variables de l'oracle.

Le résultat de l'oracle rendu par un `return(v1, ..., vn)` est une `bitstring` qui représente le n -uplet (v_1, \dots, v_n) . Dans le module `Crypto`, on trouve des fonctions permettant de décomposer un n -uplet de cette forme. Dans le futur, on fera en sorte que l'oracle lui-même rende le résultat décomposé.

La figure 6 montre un exemple de programme complexe et l'interface du module qui lui correspond. On n'a pas besoin d'indiquer les indices de réplification pour les oracles lorsque on écrit ce processus dans la spécification, mais les indiquer ici permet de mieux comprendre comment le processus est censé fonctionner. Présentons tout d'abord l'exemple. Au départ, on ne peut exécuter que l'oracle `Oa`. Ensuite après avoir exécuté `Oa`, on peut exécuter les processus `Ob[1]` à `Ob[N]`, puis après avoir exécuté `Ob[i]`, on peut exécuter `Oc[i]` et les processus `Of[1, i]` à `Of[N', i]`, et enfin, après avoir exécuté `Of[j, i]`, on peut exécuter `Og[j, i]`.

<pre> pA { (Oa() := return()); foreach i ≤ N do Ob[i]() := return(); ((Oc[i]() := return()) ((foreach j ≤ N' do Of[j, i]() := return(); Og[j, i]() := return())))) </pre>	<pre> open Crypto val init : unit → unit val oracle_Oa : unit → bitstring val oracle_Ob : unit → (bitstring*int) val oracle_Oc : unit → int → bitstring val oracle_Of : unit → int → (bitstring*int) val oracle_Og : unit → int → int → bitstring </pre>
---	--

FIGURE 6 – Prototypes des oracles

On a besoin, dans le module, de stocker quels oracles l’attaquant a le droit d’exécuter ensuite, et donc dans le cas de l’exemple, on aurait une consommation mémoire linéaire dans le nombre d’appels aux oracles. Pour réduire cette consommation, nous avons fait en sorte que le premier oracle après une réplication génère l’indice de réplication lui correspondant et ensuite que les oracles le suivant prennent en argument cet indice. C’est pour cela que la fonction `oracle_Ob` renvoie le résultat de type `bitstring` et l’indice de réplication i de la réplication le précédant de type `int`. Ensuite `oracle_Oc` et `oracle_Of` prennent en argument cet indice pour pouvoir être exécutés. L’oracle `oracle_Of` retourne l’indice j correspondant à la réplication le précédant, et l’oracle `oracle_Og` prend j et i en argument.

On garde dans l’état du module, pour un oracle juste après une réplication, uniquement l’indice courant de la réplication le précédant. Et pour un oracle le suivant, on garde l’ensemble des indices avec lesquels on peut exécuter cet oracle. Ceci permet de réduire la consommation mémoire, même si dans le pire des cas la consommation reste linéaire en fonction du nombre d’appels.

Un des problèmes de ce point de vue est qu’on ne peut plus avoir deux réplifications qui se suivent. On peut tout de même les émuler avec un oracle qui ne fait rien entre les deux et qui sert uniquement à changer l’indice de la première réplication. Mais la plupart des protocoles cryptographiques n’ont pas besoin d’avoir deux réplifications qui se suivent, et ne sont même pas aussi alambiqués que celui de la figure 6. Un protocole d’échange de clés consiste en un certain nombre d’échange de messages bien définis entre les participants pour établir une clé commune. Ensuite cette clé est utilisée pour communiquer un nombre indéfini de messages. Le rôle d’un des participants peut s’écrire par l’enchaînement des oracles qui exécutent le protocole pour établir la clé commune entre les participants, puis d’un oracle sous réplication

qui va envoyer les messages chiffrés sous cette clé. Dans la plupart des cas, il n'y a qu'une réplication dans les parties du protocole que l'on veut voir implémentées.

3.2 Implémentation

Dans cette section je vais expliquer plus en détail comment les oracles sont traduits en code OCaml.

3.2.1 L'état du module

L'état du module est un ensemble de variables d'environnement qui permettent de garder les informations requises pour lancer les oracles. La fonction `init` sert à initialiser cet état.

La variable d'environnement `envx`. Comme dit précédemment, on a besoin de savoir quels oracles l'attaquant a le droit d'exécuter. Cet variable est une structure (`record`) qui garde en elle une valeur pour chaque oracle. Cet valeur est d'un type qui dépend où se trouve l'oracle par rapport aux réplifications.

Soit n_O le nombre de réplifications sous O , et soit P_O la propriété que O se trouve juste après une réplication. La valeur correspondant à l'oracle O est présentée dans la figure 7.

La fonction `init` initialise cette variable avec les oracles exécutables au début. Les oracles, avant de retourner, modifient ces variables pour dire que l'instance actuelle ne peut plus être exécutée en modifiant la variable :

- Si $n_O = 0$, alors on la met à `false` ;
- si $n_O \geq 1$ et P_O vraie, alors on ajoute 1 à l'entier, de sorte que l'indice de réplication généré est incrémenté à chaque appel ;
- si $n_O \geq 1$ et P_O fausse, alors on enlève l'indice courant de l'ensemble.

Et aussi, ils modifient les variables des oracles qui le suivent en indiquant que l'on peut les exécuter.

L'oracle vérifie aussi au début qu'il a bien le droit de s'exécuter et lance une exception `Error` dans le cas contraire.

La variable d'environnement `env`. Il y a trois niveaux de variables, premièrement, les variables partagées entre plusieurs programmes qui sont enregistrées dans des fichiers, deuxièmement, les variables globales à un programme, contenant les variables requises d'un oracle sur l'autre et les va-

n_O	P_O	Type	Valeur
0	faux	booléen	Vrai si exécutable, faux sinon.
1	vrai	<code>int option</code>	Some i si exécutable, et i est l'indice de réplication renvoyé par l'oracle. None si non exécutable.
1	faux	ensemble d'entiers E représenté par une liste d'intervalles (couples d'entiers)	Exécutable si l'indice i en argument de l'oracle est dans E .
≥ 2	vrai	table de hachage des $n_O - 1$ indices de réplication les plus extérieurs vers un entier	Exécutable si les indices en argument de l'oracle sont associés à un entier i . i est renvoyé par l'oracle.
≥ 2	faux	table de hachage des $n_O - 1$ indices de réplication les plus extérieurs vers un ensemble d'entiers	Exécutable si les derniers indices en argument sont associés à un ensemble E et le premier indice i est dans E .

FIGURE 7 – Contenu de la variable `envx` pour l'oracle O

riables partagées entre plusieurs programmes, et troisièmement, les variables locales à un oracle.

La variable d'environnement `env` s'occupe du deuxième niveau de variables. On a besoin de connaître la valeur des variables qui sont définies dans un oracle et utilisées dans un des oracles suivants, et donc on a besoin de les garder globalement. S'il n'y a pas de réplifications sous la définition de la variable, alors on garde une `bitstring`, sinon on garde une table de hachage associant les indices de réplication à une `bitstring`. On stocke aussi dans cette variable la valeur des variables lues d'un fichier. La fonction `init` charge les fichiers requis dans la variable d'environnement, et l'oracle, au départ, charge de l'environnement l'ensemble des variables qui lui sont requises.

3.2.2 Implémentation d'un oracle

La fonction implémentant un oracle fait plusieurs choses avant d'exécuter le corps d'oracle correspondant.

1. Il vérifie, en regardant dans la variable d'environnement `envx` s'il peut être exécuté. S'il ne peut pas être exécuté, il lance l'exception `Error`.

2. Il charge les variables requises depuis la variable d'environnement `env` dans des variables locales.
3. Il enregistre les arguments de l'oracle dans des variables locales.

Le corps d'oracle est ensuite traduit par induction sur la syntaxe, cette fonction f de traduction est présentée en détail dans l'annexe A. La plupart des constructions ont un équivalent en OCaml, et leur traduction n'est pas très difficile.

Conclusion

Nous avons présenté le compilateur que nous développons permettant d'obtenir une implémentation à partir d'une spécification CryptoVerif. Pour pouvoir obtenir une implémentation prouvée, il faut prouver que le compilateur est correct. Nous avons commencé la preuve, mais elle n'est pas encore finie. La preuve consiste en essayer de faire correspondre les traces de la réduction dans la sémantique d'un terme écrit dans la syntaxe vue dans la figure 2 et de celle de la réduction du terme OCaml que l'on obtient par la fonction f présentée dans la figure 8.

Nous avons testé notre compilateur sur plusieurs petits protocoles :

- le protocole de l'exemple 3 ;
- un protocole légèrement plus compliqué que celui présenté dans la figure 6 ;
- un protocole simple avec une table de clés, utilisant la construction `find`.

Ces tests nous ont permis de trouver de nombreuses erreurs lors de l'implémentation de notre compilateur.

Ensuite, on peut faire des études de cas plus ambitieuses, comme par exemple :

- essayer d'obtenir une implémentation pour un protocole utilisé en pratique comme TLS ou Kerberos, puis essayer d'interagir avec une des implémentations déjà écrites.
- se pencher sur la preuve de l'implémentation des primitives cryptographiques, qui est un élément requis pour la preuve de l'implémentation.

Remerciements

Nous voudrions remercier Bruno Blanchet pour son aide dans ce travail. Ce travail a été partiellement pris en charge par le projet ANR FormaCrypt et la DGA.

A La fonction de traduction d'un corps d'oracle

On note $f_M(x)$ avec x un terme CryptoVerif le terme OCaml qui lui correspond. Lorsque cette fonction est utilisée sur une variable $v[\tilde{a}]$, le résultat va dépendre de si ses indices de réplication \tilde{a} sont les mêmes que les derniers indices de réplication courants \tilde{i} . Si c'est le cas, alors on traduit cela tout simplement par v , sinon on cherche la valeur dans la variable d'environnement **env**. Lorsqu'elle est appliquée à une fonction, on applique inductivement sa traduction, que l'on a dû indiquer dans les annotations, aux traductions de ses arguments.

La fonction $f_V(v[\tilde{i}])$ ne sert que lors d'affectations. Elle doit mettre à jour la variable d'environnement **env** après avoir affecté à v sa valeur.

On note $f_d(Q)$ avec Q une définition d'oracles l'ensemble des oracles de Q . Le résultat de cette fonction n'est utilisé que comme argument à la fonction `updateenvx(Q, (Q0, b))`, qui prend l'oracle actuel Q_0 et les oracles qui suivent Q en paramètre, supprime de la variable d'environnement **envx** la possibilité d'appeler Q_0 et y ajoute les oracles dans Q .

Pour traduire le corps d'oracle correspondant à un oracle, on a besoin de la définition de cet oracle Q_0 , et si oui ou non cet oracle vérifie la propriété P_{Q_0} , c'est-à-dire si Q_0 est juste après une réplication. La variable b utilisée dans la figure 8 vaut \top dans ce cas, et \perp dans le cas contraire.

Les règles de la figure 8 définissent pour chaque élément de la syntaxe d'un corps d'oracles (figure 2) comment il est traduit, en fonction de la définition de l'oracle Q_0 et de si l'oracle est juste après une réplication b .

Décrivons maintenant les règles :

- La règle (New) traduit la génération de nombre aléatoire par une affectation de la variable par le résultat de la fonction de génération aléatoire `random` qui prend le nombre de bits à générer qui est la taille du type et renvoie une valeur de type `bitstring`.
- La règle (Let) traduit l'affectation par une affectation.
- La règle (If) traduit la condition par une condition. La fonction OCaml `get_bool` sert à transformer une `bitstring` de type CryptoVerif `bool` en un booléen en OCaml.
- Les règles (Output1) et (Output2) traduisent le renvoi du résultat par un « renvoi de résultat » en OCaml. On a besoin de différencier les deux règles parce qu'il faut renvoyer l'indice de réplication si $b = \top$. Dans les deux cas, on met à jour la variable d'environnement **envx** avant de retourner le résultat.
- La règle (End) traduit la fin du processus par la mise à jour d'**envx** puis le lancement d'une exception. Il nous fallait un moyen de différencier le

$f((Q_0, b), x[\tilde{i}] \stackrel{R}{\leftarrow} T; P) = \text{let } f_V(x[\tilde{i}]) = \text{random } s \text{ in } f((Q_0, b), P)$ (New)
avec T type **fixed** de taille s .

$f((Q_0, b), x[\tilde{i}] \leftarrow M; P) = \text{let } f_V(x[\tilde{i}]) = f_M(M) \text{ in } f((Q_0, b), P)$ (Let)

$f((Q_0, b), \text{if } M \text{ then } P \text{ else } P') =$
 $\text{if } (\text{get_bool } f_M(M)) \text{ then } f((Q_0, b), P) \text{ else } f((Q_0, b), P')$ (If)

$f((Q_0, \perp), \text{return}(N_1, \dots, N_k); Q) =$
 $(\text{updateenvx}(f_d(Q), (Q_0, \perp)); (f_M(N_1), \dots, f_M(N_k)))$ (Output1)

$f((Q_0, \top), \text{return}(N_1, \dots, N_k); Q) =$
 $(\text{updateenvx}(f_d(Q), (Q_0, \top)); (f_M(i), (f_M(N_1), \dots, f_M(N_k))))$
avec i l'indice de réplication le plus intérieur.

(Output2)

$f((Q_0, b), \text{end}) = (\text{updateenvx}(\emptyset, (Q_0, b)); \text{raise Match_fail})$ (End)

$\forall j, l, M_{jl} = a_{jl}[\tilde{i}_{jl}] \wedge \tilde{i}_{jl} = (u_{jl_1}, \dots, u_{jl_{m_{jl}}})$

$\forall j, l, m, \text{test}_{jlm} = u_{jl_m} = \perp \parallel u_{jl_m} = ((\text{fun } (_, \dots, _, x, _, \dots, _) \rightarrow x)v_l)$
avec x à la $m^{\text{ème}}$ place dans le tuple

$\forall j, 1 \leq j \leq m, \text{collect}(j) = \text{let}(u_{j_1}, \dots, u_{j_{m_j}}) = (\perp, \dots, \perp) \text{ in}$
 $(\text{Hashtbl.fold } (\text{fun } v_1 \text{ value } d_1 \rightarrow \text{if } (\text{test}_{j11}) \&\& \dots \&\& (\text{test}_{j1m_{j1}})) \text{ then}$
 $\text{let } (u_{j1_1}, \dots, u_{j1_{m_{j1}}}) = v_1 \text{ in}$

⋮

$\text{Hashtbl.fold } (\text{fun } v_{l_j} \text{ value } d_{l_j} \rightarrow \text{if } (\text{test}_{jl_1}) \&\& \dots \&\& (\text{test}_{jl_j m_{jl_j}})) \text{ then}$

$\text{let } (u_{jl_{j1}}, \dots, u_{jl_{j m_{jl_j}}}) = v_{l_j} \text{ in}$

$\text{if } \text{get_bool } f(M_j) \text{ then } (j, [u_{j1}; \dots; u_{j m_j}]) :: d_{l_j} \text{ else } d_{l_j}$

$f_M(a_{jl_j}[\tilde{i}_{jl_j}]) \parallel \dots \parallel d_{jl_{j-1}} \text{ else } d_{jl_{j-1}} \dots) f_M(a_{j2}[\tilde{i}_{j2}]) \parallel \dots \parallel d_{j1}$

$\text{else } d_{j1} f_M(a_{j1}[\tilde{i}_{j1}]) (\text{collect}(j+1))$

et $\text{collect}(m+1) = []$

$f((Q_0, b), \text{find } (\bigoplus_{j=1}^m u_{j1}[\tilde{i}] \leq n_{j1}, \dots, u_{j m_j}[\tilde{i}] \leq n_{j m_j})$
 $\text{suchthat defined}(M_{j1}, \dots, M_{j l_j}) \wedge M_j \text{ then } P_j \text{ else } P)$
 $\text{let } \text{liste} = \text{collect}(1) \text{ in if } \text{liste} = [] \text{ then } f((Q_0, b), P)$
 else

$\text{let } (j, u) = \text{random}_l \text{ liste in}$

$(\text{match } j \text{ with}$

$| 1 \rightarrow \text{let } [u_{11}; \dots; u_{1 m_1}] = u \text{ in } f((Q_0, b), P_1)$

⋮

$| m \rightarrow \text{let } [u_{m1}; \dots; u_{m m_m}] = u \text{ in } f((Q_0, b), P_m))$

(Find)

fait que l'oracle renvoie normalement un résultat et le fait de terminer sur `end`, ce qu'on a fait en utilisant le système d'exceptions d'OCaml. De plus, la plupart du temps, on retrouve la construction `end` lorsque un test vérifiant la bonne forme d'un message échoue, et une exception dans ce cas semble être la chose la plus raisonnable à faire.

- La règle (Find) est la seule règle vraiment compliquée du fait qu'il n'existe pas un équivalent simple en OCaml. Le terme OCaml `collect(j)` calcule la liste des indices $(u_{j1}[\tilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\tilde{i}] \leq n_{jm_j})$ qui satisfont le terme `defined(M_{j1}, \dots, M_{jl_j}) \wedge M_j` , et on les rajoute dans la liste déjà calculée pour les $k, k > j$, après les avoir annotées avec j , la branche à partir d'où le résultat provient. Pour cela, on itère sur chaque variable que l'on doit vérifier comme définie M_{j1}, \dots, M_{jl_j} , qui sont de la forme d'une variable avec comme premiers indices de réplication des éléments de u_{j1}, \dots, u_{jm_j} . Cette hypothèse est vérifiée pour le premier jeu dans `CryptoVerif`, mais dans les jeux suivants on peut avoir n'importe quel type de terme ici. On suppose, de plus, que les tous les indices de réplication sont présents dans les termes du `defined`. C'est une hypothèse raisonnable parce qu'on ne peut pas utiliser de variables non définies dans le langage d'entrée, et seules les variables dans le bloc `defined` sont définies pour la suite du programme. On vérifie que les indices de réplication de la variable collent avec les indices des variables précédentes. C'est ce que font les tests `testjlm`. La valeur \perp signifie que la valeur de la variable n'a pas encore été décidée. Puis lorsqu'on a trouvé un groupe de variables vérifiant la condition `defined`, on teste si M_j est vrai, et on ajoute le couple contenant j et la liste des valeurs des u_{j1}, \dots, u_{jm_j} à la liste des couples déjà calculés. `collect(1)` contient alors tous les groupes d'indices vérifiant une branche. Une fois qu'on a obtenu cette liste, si la liste est vide, c'est qu'aucune branche du `find` n'a ses conditions remplies, et donc on exécute la traduction du processus P . Sinon on prend un élément au hasard dans cette liste en utilisant la fonction `randoml`, puis on affecte les indices correspondants à la branche j que l'on obtient grâce à l'annotation faite dans `collect`, et on continue sur la traduction du processus P_j .

Références

- [AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *First IFIP International Conference on Theoretical Computer Science*, volume 1872 of *Lecture Notes on Computer Science*, pages 3–22. Springer, August 2000.
- [BBF⁺08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andy Gordon, and Sergio Maffei. Refinement types for secure implementations. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 17–32, Pittsburgh, PA, June 2008. IEEE Computer Society.
- [BCF07] Karthikeyan Bhargavan, Ricardo Corin, and Cédric Fournet. Crypto-verifying protocol implementations in ML. In *Workshop on Formal and Computational Cryptography (FCC'07)*, Venice, Italy, July 2007.
- [BCFZ08] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 459–468. ACM, October 2008.
- [BCFZ09] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Automated computational verification for cryptographic protocol implementations. Unpublished draft available at <http://www.msr-inria.inria.fr/projects/sec/fs2cv/>, 2009.
- [BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, Venice, Italy, July 2006. IEEE Computer Society.
- [Bla08] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4) :193–207, October–December 2008.
- [Bla09] Bruno Blanchet. Proverif 1.82 (automatic protocol verifier). Available at <http://www.proverif.ens.fr/>, 2009.
- [CHW06] Véronique Cortier, Heinrich Hördegen, and Bogdan Warinschi. Explicit randomness is not necessary when modeling probabilistic

- encryption. In *Workshop on Information and Computer Security (ICS 2006)*, Timisoara, Romania, September 2006. Proceedings to appear.
- [CLC08] Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)*, pages 109–118, Alexandria, Virginia, USA, October 2008. ACM.
- [Cre06] Cas J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, November 2006.
- [CW05] Véronique Cortier and Bogdan Warinschi. Computationally sound, automated proofs for security protocols. In Mooly Sagiv, editor, *Proc. 14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes on Computer Science*, pages 157–171, Edimbourg, U.K., April 2005. Springer.
- [DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12) :198–208, March 1983.
- [GLP05] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes on Computer Science*, pages 363–379, Paris, France, January 2005. Springer.
- [Mil02] Giuseppe Milicia. χ -spaces : Programming security protocols. In *Proceedings of the 14th Nordic Workshop on Programming Theory (NWPT'02)*, Tallinn, Estonia, November 2002.
- [PSD04] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2Java : Automatic cryptographic protocol Java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA'04)*, volume 1, pages 400–405, Fukuoka, Japan, March 2004. IEEE Computer Society.
- [SBP01] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena : a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2) :47–74, 2001.
- [TL07] Ilja Tšahhirov and Peeter Laud. Application of dependency graphs to security protocol analysis. In Gilles Barthe and Cédric

Fournet, editors, *3rd Symposium on Trustworthy Global Computing (TGC'07)*, volume 4912 of *Lecture Notes on Computer Science*, Sophia-Antipolis, France, November 2007. Springer.

From CryptoVerif Specifications to Computationally Secure Implementations of Protocols

(Work in Progress)

David Cadé

École Normale Supérieure, CNRS, INRIA

CryptoVerif [Bla08] is a protocol verifier in the computational model that can automatically prove properties of protocols, such as secrecy and authentication. It generates proofs by sequences of games, like those written manually by cryptographers. The first game describes the specification of the protocol to prove; the next games are deduced by relying on security assumptions on primitives or by syntactic transformations. These transformations are such that consecutive games are indistinguishable and, in the last game, the desired security property is obvious. So the first game also satisfies the property. The games are formalized in a probabilistic polynomial-time process calculus.

We are implementing a compiler that takes a CryptoVerif specification and generates an implementation of the protocol in OCaml. The goal of this work is to obtain implementations of security protocols proved secure in the computational model. We will prove that our compiler is correct, that is, the semantics of the generated code corresponds to the semantics of the specification. Therefore, if CryptoVerif can prove a property on the protocol, then the implementation will also satisfy this property.

The CryptoVerif specification is annotated by the user with hints about the implementation details: the user indicates how the specification is split into modules (*e.g.*, the key generator part, the client part, and the server part), the files that will store data shared between several modules, the name of the OCaml functions corresponding to the cryptographic primitives, and the size of the random numbers. The compiler generates code for each module. For each step of the protocol, inputting a message and outputting the response, the compiler generates a function that takes the received message as argument and returns the response. These functions can then be called by a manually implemented network layer, which can be considered as part of the adversary, so we need not make any security assumptions on it. On the other hand, the CryptoVerif specification makes security assumptions on the cryptographic primitives. We do not verify that the OCaml implementation of these primitives satisfies these assumptions: they still need to be proved manually, but the CryptoVerif speci-

fication states precisely what has to be proved.

A related approach is the work of Bhargavan et al. about the verification of protocols written in ML [BCF07, BCFZ09] and its application to TLS [BCFZ08]. They generate a CryptoVerif specification from a ML implementation and then verify it with CryptoVerif. In contrast to their work, we generate a ML implementation from a CryptoVerif specification. Their use of ML as input language, which is more flexible than the CryptoVerif specification language and also better known, is an advantage of their approach. However they are still limited by the expressive power of CryptoVerif: one sometimes has to tweak the specification in order to prove the desired security properties. Our approach makes it easier to write a specification well-suited for CryptoVerif. Our approach also favors the methodology of first writing the formal specification, proving it and second implementing it. Our compiler facilitates this second step. Finally, we separate clearly what part of the implementation to prove manually (the primitives), what is proved automatically by CryptoVerif (the protocol), and what does not need any proof (the network code). This manual separation simplifies the verification process.

Acknowledgments

We would like to thank Bruno Blanchet for his help on this work. This work was partly supported by the ANR project FormaCrypt and by DGA.

References

- [BCF07] Karthikeyan Bhargavan, Ricardo Corin, and Cédric Fournet. Cryptoverifying protocol implementations in ML. In *Workshop on Formal and Computational Cryptography (FCC'07)*, Venice, Italy, July 2007.
- [BCFZ08] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 459–468. ACM, October 2008.
- [BCFZ09] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Automated computational verification for cryptographic protocol implementations. Unpublished draft available at <http://www.msr-inria.inria.fr/projects/sec/fs2cv/>, 2009.
- [Bla08] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, October–December 2008.