Vérification semi-automatique de primitives cryptographiques

Sylvain Heraud

To cite this version:

Sylvain Heraud. Vérification semi-automatique de primitives cryptographiques. Cryptographie et sécurité [cs.CR]. Université Nice Sophia Antipolis, 2012. Français. NNT : . tel-00766757

HAL Id: tel-00766757
https://theses.hal.science/tel-00766757

Submitted on 18 Dec 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour l' obtention du grade de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention : Informatique

présentée et soutenue par

Sylvain HERAUD

Vérification semi-automatique de primitives cryptographiques

Thèse dirigée par Benjamin Grégoire

soutenue le 12 Mars 2012

Jury :

M. Benjamin GREGOIRE	Chargé de recherche	Directeur de thèse
M. Pierre-Alain FOUQUE	Maître de Conférence	Rapporteur
Mme. Christine PAULIN-MOHRING	Professeur	Rapporteur
M. Gilles BARTHE	Directeur de recherche	
M. Bruno MARTIN	Professeur	

Résumé

CertiCrypt est une bibliothèque qui permet de vérifier la sécurité exacte de primitives cryptographiques dans l'assistant à la preuve Coq. CertiCrypt instrumente l'approche des preuves par jeux, et repose sur de nombreux domaines comme les probabilités, la complexité, l'algèbre, la sémantique des langages de programmation, et les optimisations de programmes.

Dans cette thèse, nous présentons deux exemples d'utilisation d'EasyCrypt : le schéma d'encryption Hashed ElGamal, et les protocoles à connaissance nulle. Ces exemples, ainsi que les travaux antérieurs sur CertiCrypt, démontrent qu'il est possible de formaliser des preuves complexes ; toutefois, l'utilisation de CertiCrypt demande une bonne expertise en Coq, et demeure laborieuse. Afin de faciliter l'adoption des preuves formelles par la communauté cryptographique, nous avons développé EasyCrypt, un outil semi-automatique capable de reconstruire des preuves formelles de sécurité à partir d'une ébauche formelle de preuve. EasyCrypt utilise des outils de preuves automatiques pour vérifier les ébauches de preuves, puis les compiles vers des preuves vérifiables avec CertiCrypt. Nous validons EasyCrypt en prouvant à nouveau Hashed ElGamal, et comparons cette nouvelle preuve avec celle en CertiCrypt. Nous prouvons également le schéma d'encryption Cramer-Shoup. Enfin, nous expliquerons comment étendre le langage de CertiCrypt à des classes de complexité implicite, permettant de modéliser la notion de fonctions en temps polynomial.

Abstract

CertiCrypt is a framework that enables the machine-checked construction and verification of cryptographic proofs in the Coq proof assistant. CertiCrypt instruments the code-based game-based approach to cryptographic proofs, and builds upon many areas, including probability and complexity theory, algebra, semantics of programming languages, and program optimizations.

In this thesis, we illustrate the application of CertiCrypt on two examples: the Hashed ElGamal encryption scheme and zero-knowledge protocols. Like previous case studies in CertiCrypt, these examples demonstrate the feasibility of formalizing complex cryptographic proofs. However, using CertiCrypt requires a high level of expertise in Coq, and is time consuming. In order to ease the adoption of formal proofs by the cryptographic community, we develop a semi-automated tool, called EasyCrypt, for elaborating security proofs of cryptographic systems from proof sketches. Proof sketches are checked automatically using SMT solvers and automated theorem provers, and then compiled into verifiable proofs in the CertiCrypt framework. We illustrate the application of EasyCrypt with two examples: the Hashed ElGamal encryption system, and the Cramer-Shoup encryption system.

Finally, we extend the language of CertiCrypt with a formalization of polytime functions.

Disclaimer

Cette these est basée sur plusieurs publications dont j'ai été co-auteur. Par ordre chronologique :

- *Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt.*
  avec Gilles Barthe, Benjamin Grégoire et Santiago Zanella Béguelin.
  Publié dans *5th International workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *Lecture Notes in Computer Science*, pages 1–19, Malaga, Spain, 2008. Springer-Verlag.

- *A machine-checked formalization of Sigma-protocols.*
  avec Gilles Barthe, Benjamin Grégoire, Daniel Hedin et Santiago Zanella Béguelin.
  Publié dans *23rd IEEE Computer Security Foundations symposium, CSF 2010*, pages 246–260, Edinburgh, Scotland, 2010. IEEE Computer Society.

- *Computer-Aided Security Proofs for the Working Cryptographer.*
  avec Gilles Barthe, Benjamin Grégoire et Santiago Zanella Béguelin.
  Publié dans *Advances in Cryptology, CRYPTO 2011*, volume 6841 of Lecture Notes in Computer Science, pages 71-90, Santa Barbara, United States, 2011. Springer-Verlag.
  Best Paper Award.

- *A Formalization of Polytime Functions.*
  avec David Nowak.
  Publié dans *2nd International Conference on Interactive Theorem Proving, ITP 2011*, volume 6898, pages 119–134, Berg en Dal, The Netherlands, 2011, Springer-Verlag

D'autres travaux effectués dans la meme periode ne seront pas expliqués dans la thèse.

- *Implementing a Direct Method for Certificate Translation.*
  avec Gilles Barthe, Benjamin Grégoire, Anne Pacalet et Cesar Knuz.
  Publié dans *11th International Conference on Formal Engineering Methods, ICFEM 2009*, volume 5885, pages 541–560, Rio de Janeiro, Brazil

- *Verifiable Indifferentiable Hashing into Elliptic Curves.*
  avec Gilles Barthe, Benjamin Grégoire, Federico Olmedo et Santiago Zanella Béguelin.
  Publié dans *1st Conference on Principles of Security and Trust, POST 2012*, Tallinn, Estonia, *to appear*

Remerciements

Voici les lignes qui sont lues en premier mais qui sont écrites en dernier. Il s'est passé tellement de choses entre le premier jour à l'INRIA et aujourd'hui que je ne sais pas par où commencer. Je me revois encore rentrer la première fois dans le bureau de Gilles pour "l'entretien d'embauche". Je ne pensais pas rencontrer autant de monde, apprendre et voir autant de choses.

Merci à l'équipe Marelle qui est une vraie famille. J'ai adoré partager les repas avec vous, à vous écouter, à rigoler. Vous allez tous me manquer. A tous les gens encore présents : Yves, Laurent, Laurence, Tamara, Guillaume (et son fabuleux chocolat), Maxime, l'affreux, et José. Et à tous les ex membres : Anne, Nicolas, Benoit, Sidi, Ioana, Julien et Clément. Et bien-sûr à Nathalie qui est au top !

A Benjamin, qui m'a accompagné tout au long de la thèse dans la joie et la "douleur", qui m'a beaucoup appris sur tous les plans.... Merci pour les nombreux fous rires et tous les bons moments ! Muchos Gracias à Gilles, "LE" Gilles, de m'avoir montré tout ça avec autant d'humour ! A Santiago pour son aide et les moments passés ensemble. Merci à vous trois de m'avoir integré dans cette super équipe.

Domo arigato à David pour m'avoir accueilli à Tokyo. Et m'avoir montré les bons restos de Akiba. Tous nos repas partagés et nos discussions interminables restent un super souvenir !

A mes profs, qui m'ont aidé à trouver cette voie qui m'a permis de faire cette thèse, M. Bertrand pour les maths, M. Kounalis pour l'algo, M. Roy pour le fonctionnel, et Loïc et Yves pour le soupçon final de Coq. A Prove&Run, mon nouveau boulot sur Paris qui reste dans le même esprit. Merci aussi pour la patience de mes deux rapporteurs...

Merci à mes amis, aux bons vieux de la Tocard Team que j'avais plaisir à retrouver après le boulot. Les pauses avec ce bon vieux RoLuP et Clément. A mes potes de fac qui ont pris la fâcheuse habitude de devenir propriétaire, se marier et faire des bébés... A Philippe, Karine, Sophie, Valérie, Ho, Hélène, Aline, Kevin ! Et aussi à tous ceux que j'ai rencontré sur la route , que j'ai oublié de mentionner.

A mes partenaires de vadrouille : Aurélien (alias P......), Yann (Oh Jack !!!), Benoit, Damien, Lou, Virgile. On part quand vous voulez.... A mes appareils photo, qui m'ont accompagnés sans broncher dans 15 pays durant ces 4 dernières années.

A ma famille de coeur, ma grande sœur spirituelle Hélène, ma petite sœur Alicia, Sonia, Jerome, Claire, Emilie. Et à mes parents qui ont toujours été là et heureusement...

Table des matières

1 Introduction 1
- 1.1 De la cryptographie vers la sécurité prouvée sur machine 1
  - 1.1.1 Introduction à la cryptographie 1
  - 1.1.2 Introduction à la sécurité prouvée 4
  - 1.1.3 Introduction aux assistants de preuve 6
  - 1.1.4 Certifier les preuves de sécurité 7
- 1.2 Contributions 7
  - 1.2.1 Formalisation de preuve 7
  - 1.2.2 Automatiser les preuves de sécurité 8
  - 1.2.3 Vérifier les hypothèses de complexité 9
- 1.3 Plan 10

2 CertiCrypt 11
- 2.1 Sécurité prouvée en Coq 11
- 2.2 Introduction à CertiCrypt 14
  - 2.2.1 Syntaxe et Sémantique des jeux 14
  - 2.2.2 Raisonnement sur les jeux 16
- 2.3 Sécurité Sémantique de la primitive Hashed ElGamal 19
  - 2.3.1 Sécurité dans le modèle standard 19
  - 2.3.2 Sécurité dans le *Random Oracle Model* 24
- 2.4 Conclusion 30

3 Protocoles Zero-Knowledge 31
- 3.1 Introduction 32
- 3.2 Σ Protocoles 34
  - 3.2.1 Relation entre sHVZK et HVZK 36
- 3.3 Σ Protocoles pour homomorphismes spéciaux 38
  - 3.3.1 Instances concrètes de Σ^ϕ protocoles 43
  - 3.3.2 Composition de Σ^ϕ protocoles 44
- 3.4 Σ Protocoles basés sur des permutations *claw-free* 46
  - 3.4.1 Instance de Σ protocole basé sur les permutations *claw-free* 48
- 3.5 Composition de Σ protocoles 49
  - 3.5.1 Composition ET 49
  - 3.5.2 Composition OU 51
- 3.6 Conclusion 55

4 EasyCrypt . . . 57
4.1 EasyCrypt . . . 58
4.2 Description d'EasyCrypt . . . 59
4.2.1 Entête des preuve . . . 59
4.2.2 Déclaration des jeux . . . 63
4.2.3 Jugements Relationnelles pRHL . . . 66
4.2.4 Preuve automatique de Jugements pRHL . . . 66
4.2.5 Preuve manuelle de jugement pRHL . . . 69
4.2.6 Raisonner sur les probabilités . . . 71
4.2.7 Raisonnement sur les *Failure Events* . . . 72
4.2.8 Équivalences algébriques . . . 74
4.2.9 Calcul de Probabilités . . . 76
4.3 Application : Cramer-Shoup . . . 77
4.4 Limitations et Extensions . . . 84
4.4.1 Comparaison avec CertiCrypt . . . 84
4.5 Conclusion . . . 85

5 Représentation et Génération des preuves . . . 87
5.1 Traduction de la sémantique . . . 88
5.1.1 Pré-requis . . . 88
5.1.2 Sémantique dans CertiCrypt . . . 89
5.1.3 Construction de la sémantique dans CertiCrypt à partir de EasyCrypt . . . 91
5.2 Traduction des Jeux . . . 97
5.2.1 Déclaration Globale . . . 98
5.2.2 Traduction des jeux . . . 99
5.3 Preuve d'équivalence . . . 101
5.3.1 Jugements pRHL . . . 101
5.3.2 Arbre de preuve . . . 104
5.3.3 Vérification dans CertiCrypt . . . 105
5.3.4 Règles de vérification de l'arbre de preuve . . . 106
5.4 Règles spécifiques aux adversaires . . . 113
5.5 Conclusion . . . 114

6 Formalisation de fonctions polynomiales . . . 115
6.1 Notions . . . 118
6.1.1 Polynômes à plusieurs variables . . . 118
6.1.2 Notations . . . 119
6.2 Caractérisation des fonctions *polytimes* . . . 119
6.2.1 Classe de Cobham . . . 119
6.2.2 Classe de Bellantoni-Cook . . . 121
6.2.3 Version avec inférence d'arité . . . 125
6.3 Compiler Bellantoni-Cook vers Cobham . . . 125
6.4 Compiler Cobham vers Bellantoni-Cook . . . 127
6.5 Temps d'exécution polynomial dans $\mathcal{B}$ . . . 130

6.6 Application à CertiCrypt 131
6.7 Conclusion 132

7 Conclusion 135
7.1 Travaux reliés 135
7.2 Conclusion 136
7.3 Perspectives 137

1
Introduction

Nous présenterons dans cette introduction les différentes notions sur lesquelles s'appuie cette thèse, de la cryptographie vers la sécurité prouvée et l'outil CertiCrypt. Nous présenterons ensuite les contributions de cette thèse et pour finir le plan de la thèse.

1.1 De la cryptographie vers la sécurité prouvée sur machine

Dans cette section nous présenterons les différentes notions qui ont amené la cryptographie utilisée depuis l'antiquité à être prouvée sur un ordinateur plus de 2000 ans plus tard.

1.1.1 Introduction à la cryptographie

La cryptographie est utilisée depuis l'antiquité pour envoyer des informations secrètes entre deux personnes, sans qu'une tierce personne ne puisse obtenir d'information sur le contenu du message en l'interceptant.

Il y a plus de 2000 ans, Jules César utilisait la cryptographie pour envoyer des messages à l'armée romaine. Les lettres étaient remplacées en utilisant une permutation secrète. Pour un alphabet $\Sigma = \{A, B, C, D, \ldots\}$, la permutation π de type $\Sigma \to \Sigma$ était définie par :

σ	A	B	C	D	$\ldots$
$\pi(\sigma)$	E	A	Z	U	$\ldots$

Le problème est que ce système n'est pas sûr. Connaissant la langue et les statistiques de fréquence des lettres, la permutation peut se retrouver facilement. La langue peut égalament se retrouver en utilisant l'indice de coïncidence.

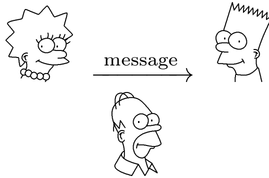
Un autre exemple est la machine Enigma utilisée par les allemands pendant la seconde guerre mondiale pour chiffrer des messages. Les messages ont été déchiffrés

par les alliés, travaux dirigés par Alan Turing. D'après Churchill, ce déchiffrement a été un facteur clé de la victoire.

La cryptographie est aujourd'hui utilisée dans la vie de tous les jours, quand nous achetons un livre sur internet au moment du paiement en ligne, quand nous lisons nos *mails* ou bien quand nous consultons notre compte bancaire sur internet. Plus généralement toutes les applications qui ont besoin de garder un secret comme un mot de passe ou un numéro de carte bleue utilisent la cryptographie,

Dans un monde idéal, le canal de communication ne peut pas être lu par une tierce personne. Nous pouvons imaginer un tuyau blindé entre chaque maison et personne ne peut regarder à l'intérieur. Si Alice chuchote un message, il sera parfaitement entendu par Bob. Et Bob sera sûr que c'est bien Alice qui envoie le message.

Ce monde n'est pas réalisable. Dans le monde réel nous communiquons à travers de grands réseaux publics comme internet. Pour sécuriser les communications, nous devons prendre en compte le fait que la communication peut être interceptée.



Le but de la cryptographie est de s'assurer que la communication respecte certaines règles de sécurité et d'éviter qu'une tierce personne, potentiellement malhonnête, ne puisse utiliser les informations contre les deux protagonistes.

Pour modéliser cette menace, nous introduisons une troisième personne, l'adversaire, qui intercepte les messages à travers le réseau public. Cet adversaire est intelligent et possède un ordinateur puissant.

Nous devons nous protéger de plusieurs attaques possibles. En particulier, les propriétés de confidentialité, d'authenticité, et d'intégrité sont importantes :

Confidentialité

Nous nous assurons que le message n'est accessible qu'à ceux dont l'accès est autorisé. Si le client Alice envoie son numéro de carte bleue au serveur Bob, nous voulons être sûrs que l'adversaire ne va pas pouvoir le lire et l'utiliser.

Authentication

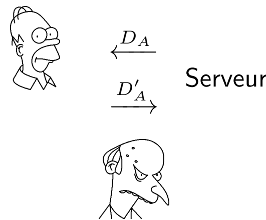
Nous permettons à la réception d'un message de vérifier s'il provient bien de l'expéditeur. Si le centre des impôts nous demande de payer un supplément, nous voulons être sûrs que ce n'est pas un organisme qui se fait passer pour les impôts (voir les récentes attaques par mail comme le hameçonnage *"phishing"*).

Intégrité

Nous voulons garantir que le message n'est pas modifié au cours de la transmission. Quand nous téléchargeons un programme, nous ne voulons pas qu'une tierce personne ajoute des informations pour modifier le comportement du programme en le rendant malveillant.

Par exemple, si Alice veut payer 100 € à Bob et l'adversaire Eve intercepte le message. Eve change le destinataire Bob par le destinataire Eve. De plus elle change le montant de 100€ en 1000€. L'intégrité permet d'éviter de telles modifications et l'authenticité permet d'assurer à la banque que Alice est bien Alice.

Prenons un autre exemple, si un docteur veut mettre à jour les données du patient A. Il doit se procurer les données D_A stockées dans une base de données puis les modifier en D'_A et les renvoyer à la même base par le réseau.



La confidentialité permet de garantir que les données D_A et D'_A ne peuvent pas être lues et qu'elles restent confidentielles. L'intégrité permet de garantir que les données D_A et D'_A ne sont pas modifiées, ce qui évite de stocker des informations médicales fausses dans la base de données. L'authenticité permet de vérifier que le docteur est bien autorisé à lire et mettre à jour le fichier de A et que c'est bien lui qui le met à jour.

En combinant confidentialité, authentification et intégrité, nous obtenons des méthodes qui s'approchent des tuyaux blindés et impénétrables entre chaque maison.

Un système cryptographique est composé d'une fonction de chiffrement et d'une fonction de déchiffrement. Pour bien fonctionner, les deux fonctions se composent. En déchiffrant un message chiffré, nous obtenons le même message.

Pour assurer la sécurité, ces deux fonctions sont paramétrées par des clés. En effet, si la fonction chiffre toujours de la même manière, ce serait facile d'obtenir des informations.

Dans la pratique, seule la clé est secrète. Un système où la clé et la fonction de chiffrement sont secrètes, sera plus sûr. Mais il est préférable d'arriver à obtenir un niveau de sécurité élevé sans forcement cacher quel système a été utilisé. Ce principe est énoncé par Kerckhoffs, qui exprime que la sécurité ne doit reposer que sur le secret de la clé. En pratique, si une clé est découverte, il est facile de la changer. En revanche, changer la fonction de chiffrement demande beaucoup plus de travail. De plus il y a moins de fonctions de chiffrement que de clés. Bruce Schneier dit "tout secret est en fait un point de cassure possible.".

Il y a deux manières de concevoir un système cryptographique à partir de clé : symétrique et asymétrique.

Dans le chiffrement symétrique, nous utilisons la même clé pour le chiffrement et le déchiffrement. Par exemple, dans le chiffrement *One-Time-Pad* de Vernam, la clé est un bitstring tiré aléatoirement dans $\{0,1\}^k$. Les fonctions de chiffrement et de déchiffrement sont définies pour tous messages $m \in \{0,1\}^k$ par

- $\mathsf{Enc}_K(m) = k \oplus m$
- $\mathsf{Dec}_K(c) = k \oplus c$

où $\oplus$ est la fonction OU-exclusif sur les bitstrings. Ce schéma est parfaitement sûr, tant que la clé n'est utilisée qu'une seule fois. Nous ne gagnons pas beaucoup à utiliser ce système, étant donné que pour échanger la clé nous devons également garantir que l'adversaire ne puisse pas l'intercepter.

Le chiffrement asymétrique permet de résoudre le problème d'échange de clés. Nous rajoutons une troisième fonction, l'algorithme de génération de clés, qui calcule un couple de clés, contenant la clé publique et la clé privée. La clé publique est diffusée alors que la clé privée est gardée secrète.

Par exemple, Bob utilise l'algorithme de génération de clé et publie sa clé publique. Il est le seul à posséder la clé secrète. Si Alice veut lui envoyer un message, elle utilise la fonction de chiffrement avec la clé publique de Bob. Pour lire le message, Bob utilise la fonction de déchiffrement avec sa clé privée. La sécurité du schéma nous assure que les messages ont une très faible probabilité d'être déchiffrés sans la clé privée.

1.1.2 Introduction à la sécurité prouvée

L'approche heuristique pour concevoir une nouvelle primitive cryptographique sûre est de proposer un couple de fonctions chiffrement/déchiffrement et d'attendre que quelqu'un trouve une attaque. Si une attaque est découverte, les fonctions sont corrigées et il faut attendre ensuite que quelqu'un trouve une autre attaque. Si aucune attaque n'est trouvée au bout d'un certain temps, la primitive est déclarée sûre, et nous pouvons l'utiliser. Le problème dans cette approche est de savoir combien de temps il faut attendre. Il a fallu cinq ans pour casser la primitive de Merkle-Hellman [79] et dix ans pour casser celle de Chor-Rivest [87]. Cette méthode ne permet pas de montrer la sécurité des primitives, car elle ne peut pas prendre en compte toutes les attaques possibles.

La *sécurité prouvée* préconisée dans [50, 81] est une bonne approche pour assurer la sécurité des primitives. Dans ce modèle, les relations entre le système cryptographique et l'adversaire sont spécifiées précisément et les preuves de sécurité sont établies avec rigueur, en rendant explicites toutes les hypothèses utilisées.

La difficulté est de pouvoir raisonner sur tous les comportements de l'adversaire, et montrer qu'aucun d'entre eux ne peut casser le système. Tester un certain nombre de comportements ne suffit pas à garantir que le système est sûr (sauf pour *One-Time-Pad*) car l'ensemble des comportements de l'adversaire est infini. De même, raisonner sur tous les adversaires possibles n'est pas possible. Un adversaire avec une puissance de calcul infini pourra toujours essayer toutes les combinaisons. Nous

devons raisonner sur des adversaires réalistes ou plausibles. Nous limitons leur puissance de calcul en restreignant la classe de complexité des adversaires aux fonctions polynomiales. Les adversaires ne sont pas non plus omniscients, c'est-à-dire qu'ils ne connaissent pas tout (par exemple ils ne connaissent pas la clé privée).

La plupart des primitives asymétriques sont basées sur des problèmes mathématiques difficiles (difficiles à calculer). Prenons la factorisation des grands nombres, et imaginons une primitive se servant du fait que la factorisation des grands nombres est difficile. Pour montrer que cette primitive est sûre, nous prouvons que si un adversaire arrive à casser la primitive, alors il peut facilement factoriser un grand nombre. Et la contraposé nous donne, si il est difficile de factoriser des grands nombres alors la primitive est sûre. Ces preuves s'appellent preuve par réduction, le problème "casser la primitive" se réduit au problème "factoriser un nombre".

Nous n'obtenons pas une preuve "il est impossible de casser la primitive", mais plutôt "aucune attaque nécessitant 2^{100} heures ne peut casser la primitive, avec une probabilité d'au plus 2^{-30}".

Il existe d'autres problèmes calculatoires difficiles, comme le problème du logarithme discret ou les problèmes NP-complet. Pour tous ces problèmes aucun algorithme polynomial n'est connu pout toutes les instances.

Si la primitive n'est pas sûre, ce n'est pas le problème sous-jacent qui est mis en cause, mais plutôt l'utilisation du problème difficile pour la primitive.

D'autres problèmes peuvent apparaître lors de l'implémentation, il faut également vérifier que l'implementation de primitives respecte bien ses spécifications. Ce point ne sera pas abordé dans cette thèse.

Pour effectuer ces preuves par réduction, nous utiliserons des preuves par jeu [26, 54, 80]. Nous allons mettre en situation un adversaire et lui demander d'obtenir une information sur un message chiffré. S'il réussit cela veut dire que le problème cryptographique calculatoire peut se résoudre avec un algorithme polynomial qui utilise l'adversaire. La preuve consiste à faire le lien entre l'adversaire du premier jeu et un autre adversaire qui casse le problème cryptographique.

$$\begin{array}{ccc} \mathsf{Jeu}_0 & \Rightarrow \quad \mathsf{Jeu}_1 \Rightarrow \cdots \Rightarrow & \mathsf{Jeu}_n \\ \mathcal{A} \text{ casse la primitive} & & \mathcal{B} \text{ résout le problème} \end{array}$$

$\mathcal{B}$ utilise $\mathcal{A}$ comme sous procédure.

La preuve par jeu décompose la réduction en petites étapes, ce qui permet de réduire la complexité de la preuve et d'améliorer sa lisibilité. De plus, ces petites étapes sont souvent les mêmes entre chaque preuve. Nous détaillerons les techniques de preuve par jeu dans le prochain chapitre.

La communauté en cryptographie produit énormément de nouvelles primitives, ou de nouvelles bornes. Chaque année de nouvelles preuves papier apparaissent, et souvent personne ne prend le temps de les lire et surtout de les vérifier. En effet, construire des preuves de sécurité peut être une tâche difficile. Certaines preuves papier de système cryptographique sont connues pour avoir gardées des failles pendant des années [26, 54]. Pour régler ce problème, Halevi [54] suggère de construire et d'utiliser un outil dédié pour faire ces preuves et les vérifier en expliquant les différentes caractéristiques et fonctionnalités de l'outil.

1.1.3 Introduction aux assistants de preuve

Nous allons utiliser des assistants de preuves pour vérifier ces preuves. Les preuves sont effectuées sur machine construite à l'aide de tactiques qui font modifier le but de preuve en theoremes déjà prouvés, hypotheses ou axiomes. La machine assure que la preuve est correcte. Ces preuves peuvent être distribuées, sous une forme que nous appelons certificats, qui sont généralement des fichiers, et une tierce personne peut facilement se convaincre que la preuve est correcte.

L'utilisation d'assistants de preuve casse la symétrie construction/vérification du papier. Sur papier, l'écriture des preuves est difficile. Mais la vérification est également aussi difficile car il faut à nouveau comprendre la preuve et tous les principes sous-jacents, le lecteur doit vérifier que les théorèmes utilisés sont bien appliqués et aussi que ces théorèmes sont vrais. Il faut également vérifier que tous les cas sont bien traités.

Les preuves machines cassent cette symétrie, la partie construction reste difficile, voire plus difficile. Par contre la partie vérification est bien plus facile, étant donné que le raisonnement est vérifié par la machine.

La construction sur machine est plus laborieuse. Par rapport à une preuve papier, toutes les étapes doivent être explicites. Des arguments papier tels que "intuitivement" ne sont pas compris par la machine. Une preuve sur machine est similaire à une loupe. Elle détaille tous les points de la preuve rendant chaque étape explicite. Refaire une preuve papier sur machine demande de comprendre la preuve papier, lire entre les lignes, boucher les trous, tout en maîtrisant l'assistant de preuve (dans notre cas Coq). Le simple fait de formaliser les différentes notions des preuves papier est une tâche difficile.

Au final, le jeu en vaut la chandelle, la vérification des preuves sur machine est très facile, et ne demande pas de comprendre le raisonnement de la preuve, ni de connaître Coq. Cette vérification se fait en deux étapes, il faut vérifier les différentes définitions, hypothèses et axiomes, et vérifier l'énoncé du théorème final (toutes les étapes intermédiaires n'ont pas besoin d'être vérifiées [1]). Lire ces définitions ne demande pas beaucoup d'expertise. Les énoncés des lemmes sont souvent très proches des preuves papier. La deuxième étape consiste à vérifier automatiquement la preuve en l'exécutant, si l'exécution se termine sans erreur, cela veut dire que le script de preuve est correct.

De plus en plus de travaux sont formalisés dans les assistants de preuve et obtiennent un grande sûreté :

- Xavier Leroy et ses collaborateurs formalisent en Coq un compilateur C [65] optimisant. La preuve garantit que la sémantique du programme source est identique à la sémantique du programme compilé et optimisé.
- George Gontier et ses collaborateurs vérifient la preuve de Feit Thomson, preuve mathématique de deux livres. Ils formalisent en Coq la bibliothèque SSReffect qui a également servi à formaliser la preuve du théorème des quatre couleurs.

1. Il faut aussi s'assurer que les fichiers ne contiennent pas la tactique `admit`.

– Le projet de G. Klein et al [63] permet de vérifier que le code C d'un noyau de système d'exploitation seL4 respecte les spécifications. Les preuves sont faites dans l'assistant de preuve Isabelle.
– ...

1.1.4 Certifier les preuves de sécurité

CertiCrypt [21] est un outil proposé par Gilles Barthe, Benjamin Grégoire et Santiago Zanella Béguelin. Cette librairie est construite au dessus de l'assistant de preuve Coq [83], qui permet à la fois d'obtenir un niveau de confiance élevé et aussi de générer un certificat vérifiable qui montre que les preuves de sécurité sont correctes. Une des ambitions de CertiCrypt, est d'augmenter la confiance des preuves en cryptographie en les prouvant en Coq. Le principe de CertiCrypt est de représenter les jeux comme des programmes en utilisant un langage probabiliste, appelé pWhile. Les transitions entre les jeux sont représentées par des transformations de programmes, et se prouvent en utilisant une logique relationnelle probabiliste, appelée pRHL (*probabilistic Relational Hoare Logic*).

La principale difficulté dans la construction d'un tel outil est que cela nécessite un large ensemble de concepts et méthodes de raisonnement, comme les tirages aléatoires, la théorie des probabilités, la théorie des groupes ou de la complexité. Dans le cas des preuves par jeux, les preuves sont également reliées à la sémantique des langages de programmation, transformation et vérification de programmes. Coq permet de mélanger tous ses concepts, tout en garantissant les preuves.

De nombreuses procédures ont été formalisées comme ElGamal, la preuve originale de *Full Domain Hash digital signature scheme*, et la borne optimale de Coron [89], ou plus récemment IBE [23], la primitive *OAEP padding scheme* dans IND-CPA et IND-CCA2 [20], mais également des résultats utilisés dans de nombreuses preuves telles que *PRP/PRF Switching Lemma* et *Fundamental Lemma of game-playing*[21].

1.2 Contributions

Cette thèse consiste à promouvoir l'outil CertiCrypt à la fois en l'utilisant et en ajoutant des exemples de preuves et en simplifiant son utilisation ou en ajoutant de nouvelles manières de raisonner.

1.2.1 Formalisation de preuve

Nous expliquerons deux travaux de formalisation à la fois pour introduire l'outil et montrer que l'outil peut être utilisé pour faire des preuves autres que celles des primitives de chiffrement.

Pour introduire l'outil nous expliquerons deux preuves de la primitive de chiffrement Hashed ElGamal en expliquant les différentes notions qui sont utilisées pour

faire des preuves par jeu sur machine tout en donnant un exemple qui sera repris tout au long de cette thèse.

Nous décrions la formalisation des protocoles Zero-Knowledge en utilisant les Σ protocoles introduits dans la thèse de Ronald Cramer [38].

D'autre travaux de formalisation sont en cours comme les récents travaux de Thomas Icart [60, 59, 32] montrant qu'il est possible de transformer un oracle aléatoire sur un corps en un autre oracle aléatoire sur une courbe elliptique sur ce même corps. La formalisation de la preuve, demande des preuves de mathématiques non triviales, sur la théories des groupes ou les courbes elliptiques. Nous utiliserons la libraire SSReflect pour les groupes et la représentation des courbes elliptiques de Laurent Théry[84]. Cette formalisation ne sera pas détaillée dans cette thèse.

Tous ces travaux montrent qu'il est possible d'obtenir une très grande garantie dans les preuves de sécurité, en utilisant l'idée de Halevi, mais CertiCrypt est encore difficile à utiliser, voire impossible pour un utilisateur ne connaissant pas Coq. Le programme de Halevi n'est pas complément atteint : ces travaux actuels sont loin des objectifs proposés qui suggèrent un outil automatique et expressif, doté d'une interface pour faire les preuves.

1.2.2 Automatiser les preuves de sécurité

L'expérience acquise en développant et utilisant CertiCrypt a permis de s'apercevoir que la plupart des étapes dans les preuves par jeu peuvent être automatisées. Nous utilisons un calcul de plus faible pré-condition adapté aux preuves d'équivalence dans un contexte probabiliste et nous déléguons à des prouveurs SMT les obligations de preuve.

Pour simplifier la vérification des preuves formelles de primitives cryptographiques dans CertiCrypt, nous avons développé EasyCrypt prenant une description de haut niveau de la preuve en restant le plus proche possible de la preuve papier. L'outil essaie différentes stratégies et les valide en utilisant les prouveurs automatiques. De plus EasyCrypt permet de raisonner sur les calculs probabilistes.

L'outil EasyCrypt est présenté dans cette thèse, nous expliquons à nouveau la preuve de Hashed ElGamal pour introduire les nouveaux concepts de l'outil et expliquer comment les preuves sont effectuées. De plus, nous prouverons le schéma Cramer-Shoup dans le modèle IND-CCA2.

EasyCrypt est écrit en OCaml et fait appel aux prouveurs SMT et n'offre pas la même garantie qu'un outil écrit en Coq. Mais il permet d'obtenir plus de flexibilité dans le développement. Pour certifier EasyCrypt, deux méthodes peuvent être utilisées :

- soit le programme est certifié : une preuve garantit que les preuves faites par le programme sont correctes.
- soit le programme est certifiant : une procédure certifiée garantit que le résultat du programme est correct.

Par exemple, pour vérifier la correction d'un compilateur, c'est-à-dire vérifier que le programme source et le programme compilé ont la même sémantique, deux solutions existent :

– soit nous prouvons que le code du compilateur en montrant qu'il préserve la sémantique [65].
– soit nous prouvons indépendemment du code du compilateur que les programmes source et compilé sont équivalents en utilisant un outil externe qui génère la preuve d'équivalence [86].

Nous adopterons la seconde méthode, nous vérifions les preuves construites par EasyCrypt en Coq en utilisant CertiCrypt. EasyCrypt va générer des fichiers de preuves compatibles avec CertiCrypt. Nous obtenons ainsi la même garantie que CertiCrypt. L'utilisateur n'aura qu'à compléter les trous apparaissant dans le processus de certification. Étant donné qu'à l'heure actuelle le lien entre les prouveurs SMT et Coq, est un travail de recherche en cours [7, 6], mais ces travaux ne sont pas encore utilisables.

EasyCrypt peut alors être vu comme une extension de CertiCrypt, qui génère des fichiers Coq, avec toutes les définitions et la trame de la preuve, où la plupart des transitions sont déjà effectuées.

Nous pensons que EasyCrypt est un bon candidat pour le programme d'Halevi et espérons que l'outil sera adopté par les cryptographes ou du moins qu'il en sera la première étape.

1.2.3 Vérifier les hypothèses de complexité

Nous avons vu que les adversaires doivent être restreints à une certaine classe de complexité en particulier la classe des fonctions calculables en temps polynomial. CertiCrypt permet alors de raisonner sur la complexité des programmes, en particulier sur la classe PPT (*probabilistic polynomial time*).

Une des conditions pour montrer qu'un programme est PPT est que le programme n'utilise que des expressions PPT, c'est-à-dire qu'il existe deux polynômes, un pour borner le temps d'exécution et un pour borner la taille des résultats. L'interprétation des expressions dans CertiCrypt se fait par une fonction Coq. Nous pouvons prouver que la taille du résultat d'une fonction Coq est bornée par un polynôme. Par contre il n'existe pas de moyen de raisonner sur le temps d'exécution des fonctions Coq.

Le chapitre 6 propose une méthode pour pouvoir raisonner sur la complexité en espace et en temps, indépendamment du modèle d'exécution. La complexité calculatoire implicite ICC permet de raisonner sur des langages, restreints à une certaine classe de complexité.

Nous formalisons deux classes de programmes où toutes les fonctions de cette classe se calculent en temps polynomial. De plus Cobham [33] montre que l'ensemble des fonctions polynomiales peuvent s'écrire dans sa classe.

Nous prouvons en Coq les résultats du papier de Bellantoni et Cook [24] en les adaptant au contexte des preuves de sécurité. Ensuite nous ferons le lien avec CertiCrypt, en ajoutant au langage les expressions de la classe de Bellantoni et Cook en prouvant pour toutes les expressions l'existence de deux polynômes pour borner la taille des résultats et le temps d'exécution.

1.3 Plan

Le but de cette thèse est de montrer des exemples d'utilisation et des extensions de de CertiCrypt ainsi que le développement de EasyCrypt. Le reste de la thèse est organisée de la manière suivante :

- Le Chapitre 2 décrit l'outil CertiCrypt en donnant deux preuves de sécurité de la primitive Hashed ElGamal dans le modèle standard et le modèle est oracle aléatoire. Nous donnerons aussi un état de l'art des autres formalisations de preuve de sécurité.
- Le Chapitre 3 présente la formalisation des Σ protocoles. Nous montrons comment utiliser notre formalisation pour obtenir facilement les preuves de protocole de la littérature,
- Le Chapitre 4 introduit l'outil EasyCrypt en expliquant les différentes étapes de la preuve de la primitive Hashed ElGamal. Nous expliquons également la preuve de la primitive Cramer-Shoup IND-CCA2.
- Le Chapitre 5 décrit la génération de preuve. Nous montrons que l'outil est bien plus facile à utiliser que CertiCrypt, nous gagnons à la fois en flexibilité et en automatisation, sans perdre la garantie offerte par les assistants de preuve.
- Le Chapitre 6 explique la formalisation de deux classes de complexité implicite ; de Cobham et de Bellantoni et Cook. Nous expliquons comment utiliser ces deux classes dans les preuves de sécurité en ajoutant la classe de Bellantoni et Cook à CertiCrypt.
- Nous conclurons dans le chapitre 7.

2

CertiCrypt

Dans ce chapitre, nous introduirons CertiCrypt, en donnant une présentation détaillée des preuves de sécurité en s'appuyant sur la primitive Hashed ElGamal. Nous suivrons les preuves du papier de Shoup introduisant les preuves par jeux [80], nous prouverons la primitive Hashed ElGamal dans deux cadres différents : le modèle standard, en faisant l'hypothèse que la fonction de hachage est *entropy smoothing*, et dans le *random oracle model*, qui fait l'hypothèse que la fonction de hachage est indistingable d'une vraie fonction aléatoire. Ces preuves généralisent la preuve de ElGamal, présentée dans [21].

2.1 Sécurité prouvée en Coq

Les notions de sécurité sont exprimées à travers des programmes probabilistes qui permettent de mettre en relation les systèmes cryptographiques et les adversaires. Pour effectuer ce lien, Nous utilisons la technique de preuve par jeux.

Un schéma de chiffrement est composé de trois fonctions (KE, Enc, Dec), soient les fonctions de génération de clé, de chiffrement et de déchiffrement.

IND-CPA est un exemple de notion de sécurité : l'attaquant choisit deux messages, un des deux est chiffré puis il doit ensuite deviner lequel a été chiffré. Plus clairement, un schéma de chiffrement est IND-CPA, si tout adversaire plausible qui ne connaît que la clé publique et qui choisit une paire de message (m_0, m_1), ne peut pas distinguer le chiffrement du message m_0 du chiffrement du message m_1. Pour éviter que l'adversaire ne chiffre m_0 et m_1 et compare le chiffrement de m_0 et m_1, avec le message chiffré qu'il a reçu, le chiffrement est probabiliste et ne renverra pas les mêmes messages chiffrés. Dans le contexte des preuves par jeux, la sécurité sémantique est spécifiée à l'aide du programme probabiliste suivant :

Game IND-CPA :	
$(sk, pk) \leftarrow \mathcal{KG}(\,)$;	(génération des clés)
$(m_0, m_1) \leftarrow \mathcal{A}(pk)$;	(génération des messages)
$b \stackrel{\$}{\leftarrow} \{0, 1\}$;	(tirage aléatoire du bit b)
$\zeta \leftarrow \mathsf{Enc}(pk, m_b)$;	(chiffrement du message m_b)
$b' \leftarrow \mathcal{A}'(pk, \zeta)$;	(l'adversaire devine quel message a été chiffré)
$d \leftarrow b = b'$	(le bit d est vrai si l'adversaire a réussi)

$\mathcal{A}$ et $\mathcal{A}'$ sont des procédures qui représentent l'adversaire. La syntaxe est donnée plus loin.

En plus des procédures qui apparaissent dans le programme, le jeu peut faire appel à des oracles pouvant aussi être appelés par l'adversaire ; dans Hashed ElGamal l'adversaire a accès à un oracle de hachage. Les spécifications du jeu IND-CPA sont complétées en déclarant que les adversaires sont dans la classe de complexité des programmes qui s'exécutent en temps polynomial probabiliste. L'adversaire n'a accès qu'à une variable globale Gadv qui permet d'unifier les procédures A et A' en leur permettant de partager des informations. L'adversaire a également accès à la clé publique pk en lecture seule et n'a évidemment ni accès à la clé secrète sk ni au bit b.

CertiCrypt fournit deux manières de définir les variables, soit de manière locale (la variable sera accessible uniquement dans la portée de la procédure) soit globale (la variable est accessible dans tous les programmes). Concernant l'adversaire, il est possible de limiter l'accès aux variables globales en lecture ou en écriture.

Deux autres notions de sécurité plus fortes que IND-CPA sont IND-CCA1 ou IND-CCA2. L'adversaire est plus puissant et peut donc faire plus de choses.

- La première étape de l'adversaire dans IND-CCA1 et IND-CCA2 est identique : l'adversaire peut appeler la fonction déchiffrement avec tous les messages chiffrés possibles (même celui du challenge)
- La deuxième étape de l'adversaire diffère entre les deux notions :
  - dans IND-CCA1 l'adversaire ne peut plus appeler la fonction de déchiffrement
  - dans IND-CCA2 l'adversaire peut appeler la fonction de déchiffrement, mais ne peut pas demander de déchiffrer le message chiffré.

Dans CertiCrypt, les définitions de ces notions sont très proches. Le jeu IND-CPA est le même pour toutes les notions, mais la définition de l'adversaire diffère en étendant ou pas la liste des oracles qu'il peut appeler.

Dans toutes ces notions le but des preuves est de montrer que la probabilité qu'un adversaire a de deviner quel message a été chiffré est proche de 1/2 : quoique fasse l'adversaire son comportement est identique à choisir au hasard quel message a été chiffré.

L'ensemble du schéma dépend de η appelé paramètre de sécurité. Le but de la preuve est de montrer que la probabilité de $d = 1$, noté $\mathsf{Pr}[d = 1]$ est proche de $1/2$. Plus formellement, les propriétés montrent que la différence entre $\mathsf{Pr}[d = 1]$ et $1/2$ est négligeable. Cette différence est une fonction paramétrée par η. Une fonction $\nu : \mathbb{N} \rightarrow \mathbb{R}$ est négligeable si

$$\mathsf{negligible}(\nu) \stackrel{\text{def}}{=} \forall c.\ \exists n_c.\ \forall n.\ n \geq n_c \Rightarrow |\nu(n)| \leq n^{-c}$$

Une fonction ν est proche d'une constante k de façon négligeable quand la fonction $\lambda\eta.|\nu(\eta) - k|$ est négligeable.

Le but des preuves par jeux est de montrer une propriété de sécurité, telle IND-CPA pour un schéma de chiffrement, à travers une série de transformations appliquées au jeu d'attaque initial G_0. Plus précisément, les preuves par jeux sont organisées comme une suite de transitions de la forme $G, A \to G', A'$ où G et G' sont des jeux, et A et A' sont des évènements. Le but est d'établir pour chaque transition $\Pr[G : A] \leq f(\Pr[G' : A'])$, pour une fonction monotone f. En combinant l'ensemble des inégalités pour chaque transition, l'inégalité $\Pr[G_0 : A_0] \leq f(\Pr[G_n : A_n])$ est obtenue. Si G_0, A_0 correspond au jeu initial et un évènement, il est possible d'obtenir une borne de $\Pr[G_0 : A_0]$ à partir d'une borne de $\Pr[G_n : A_n]$.

Dans beaucoup de cas, les transitions $G, A \to G', A'$ montrent que $\Pr[G : A] = \Pr[G' : A']$. Ce genre de transition appelée *bridging step*, inclut les transformations de programme qui préservent la sémantique. La préservation de la sémantique est définie par la non-interférence probabiliste [75], vu que les preuves ne portent que sur le comportement observable des programmes. Cependant, il y a beaucoup de cas où la préservation de la sémantique dépend du contexte. Dans ce cas, une logique relationnelle qui généralise la non-interférence probabiliste est utilisée. Elle permet de raisonner en fonction de la pré et post-condition, à la manière de la logique de Hoare mais dans le cadre relationnel.

Les preuves par jeux reposent également sur des *failure events*, qui permettent de borner la probabilité qui est "perdue" au fil des transitions dès qu'un certain évènement se produit. Un outil essentiel pour raisonner sur les évènements d'échec est appelé le lemme fondamental : étant donné deux jeux G_1 et G_2 dont le code diffère uniquement après qu'un évènement bad (variable booléenne) soit vrai. L'évènement bad devient vrai si il est initialisé à false au début du jeu et il reste vrai une fois qu'il prend la valeur true, jusqu'à la fin du jeu. Alors pour tout évènement A, $\Pr[G_1 : A \land \neg\mathsf{bad}] = \Pr[G_2 : A \land \neg\mathsf{bad}]$. Cela implique que

$$|\Pr[G_1 : A] - \Pr[G_2 : A]| \leq \Pr[G_1 : \mathsf{bad}] = \Pr[G_2 : \mathsf{bad}]$$

à condition que les deux jeux terminent avec la même probabilité.

Pour finir, certaines transitions sont justifiées par des hypothèses de sécurité. Par exemple, la preuve de la Section 2.3.2 est basée sur l'hypothèse *Decisional Diffie-Hellman* (DDH). Pour une famille de groupe cyclique $\mathbb{Z}_q$, où q est l'ordre premier du groupe et g est un générateur. Cette hypothèse exprime qu'il n'existe pas d'algorithme efficace qui peut distinguer un triplet de la forme (g^x, g^y, g^{xy}) d'un triplet de la forme (g^x, g^y, g^z), où x, y, z sont tirés aléatoirement dans $\mathbb{Z}_q$. L'une des particularités des preuves par jeux est de formuler ce genre d'hypothèse en utilisant des jeux. Pour tout adversaires $\mathcal{B}$, nous avons deux jeux definit par :

Définition 2.1 (DDH assumption). *Considérant les jeux*

Game DDH_0 :	**Game** DDH_1 :
$x, y \xleftarrow{\$} \mathbb{Z}_q$;	$x, y, z \xleftarrow{\$} \mathbb{Z}_q$;
$d \leftarrow \mathcal{B}(g^x, g^y, g^{xy})$	$d \leftarrow \mathcal{B}(g^x, g^y, g^z)$

et avec

$$\epsilon_{DDH}(\eta) \stackrel{def}{=} |\Pr[DDH_0^\eta : d = 1] - \Pr[DDH_1^\eta : d = 1]|$$

Alors ϵ_{DDH} est une fonction négligeable. La sémantique des deux jeux et l'ordre du groupe q dépendent du paramètre de sécurité η.

2.2 Introduction à CertiCrypt

Le but de cette section est d'expliquer comment fonctionne CertiCrypt. Nous expliquons la syntaxe et la sémantique du langage utilisé pour décrire les jeux et les outils pour raisonner entre eux. Plus d'informations sur CertiCrypt sont disponibles dans la thèse de Santiago Zanella [88].

2.2.1 Syntaxe et Sémantique des jeux

La sous-couche de CertiCrypt est représentée par un langage de programmation probabiliste avec appel de procédure. Étant donné un ensemble de variables $\mathcal{V}$ et un ensemble de noms de procédures $\mathcal{P}$, les commandes du langage sont définies inductivement par les clauses suivantes :

$\text{i} ::= \mathcal{V} \leftarrow \mathcal{E}$	affectation déterministe
$\mid\ \mathcal{V} \xleftarrow{\$} \mathcal{D}$	affectation aléatoire
$\mid\ \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C}$	instruction conditionelle
$\mid\ \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C}$	boucle while
$\mid\ \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \dots, \mathcal{E})$	appel de procédure
$\mathcal{C} ::= \mathsf{nil}$	nop
$\mid\ \text{i};\ \mathcal{C}$	séquence

où $\mathcal{E}$ est l'ensemble des expressions et $\mathcal{D}$ est l'ensemble des distributions où des valeurs aléatoires peuvent être tirées lors des affectations aléatoires. CertiCrypt contient des types et opérateurs de base mais la syntaxe peut être étendue par l'utilisateur, en ajoutant à la sémantique de nouveaux types et opérations avec leurs interprétations en Coq.

Les jeux sont définis par un programme principal et un environnement qui relie un identifiant de procédure à sa déclaration. Ces environnements sont représentés par des listes de paramètres formels, le corps de la fonction et l'expression de retour qui sera définie par l'instruction return

$$\mathsf{declaration} \stackrel{\text{def}}{=} \{\mathsf{params} : \mathcal{V}^\star;\ \mathsf{body} : \mathcal{C};\ \mathsf{re} : \mathcal{E}\}$$

Les programmes sont interprétés comme des fonctions s'exécutant à partir d'une mémoire initiale et renvoyant une distribution de sous-probabilité sur les mémoires

finales. Les sous-distributions à partir d'un support fini A pourraient être définies de la manière suivante :

$$\mu : A \to [0,1] \qquad \text{telle que} \sum_{x \in A} \mu(x) \leq 1$$

Mais la sémantique des jeux utilise une construction plus générale et utilise des distributions sur des mesures de probabilité. Le type des distributions sur un ensemble A est

$$\mathcal{D}(A) \stackrel{\text{def}}{=} (A \to [0,1]) \to [0,1]$$

Les monades sont définies à partir des constructeurs unit et bind [8] :

$$\begin{array}{rcl} \mathsf{unit} & : & X \to \mathcal{D}(X) \stackrel{\text{def}}{=} \lambda x.\ \lambda f.\ f\ x \\ \mathsf{bind} & : & \mathcal{D}(X) \to (X \to \mathcal{D}(Y)) \to \mathcal{D}(Y) \\ & \stackrel{\text{def}}{=} & \lambda \mu.\ \lambda M.\ \lambda f.\ \mu(\lambda\ x.\ M\ x\ f) \end{array}$$

Cette monade peut être vue comme une spécialisation des monades de continuation et permet de représenter la sémantique des jeux. Un élément de $\mathcal{D}(X)$ peut être interprété comme une (sous) probabilité de X. La dénotation d'un jeu établit un rapport entre la mémoire initiale et une continuation sur les probabilités des mémoires finales après avoir exécuté le programme. Pour comprendre cette thèse il n'est pas nécessaire de comprendre comment sont définies ces monades. Plus d'information est disponible dans [21] et [88].

La sémantique dénotationnelle des jeux est présentée dans la figure 2.1 ; une mémoire m est représentée par une paire $(m.\mathsf{loc}, m.\mathsf{glob})$, rendant explicite les variables locales et globales. Les expressions sont déterministes et leur sémantique est donnée par la fonction $[\![\cdot]\!]_{\mathcal{E}}$ qui évalue une expression étant donné une mémoire et retournent une valeur. La sémantique des distributions dans $\mathcal{D}$ est donnée par une autre fonction $[\![\cdot]\!]_{\mathcal{D}}$; nous donnons comme exemple la sémantique de la distribution uniforme sur $\mathbb{B}$ ($\{0,1\}$) et sur l'intervalle $[0..n]$. Dans la figure nous omettons les environnements de procédure E pour faciliter la lisibilité. Dans la suite nous ne ferons plus de distinction entre un jeu $G = (c, E)$ et sa commande principale c quand il n'y a pas d'ambiguïté. Les boucles sont definies par $[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]\ m = [\![\mathsf{if}\ e\ \mathsf{then}\ c;\ \mathsf{while}\ e\ \mathsf{do}\ c]\!]\ m$

La sémantique $[\![\cdot]\!]$ associe des distributions sur les mémoires $\mathcal{D}(\mathcal{M})$ à des mémoires $\mathcal{M}$. Il est facile de définir une autre fonction $[\![\cdot]\!]'$ de $\mathcal{D}(\mathcal{M})$ vers $\mathcal{D}(\mathcal{M})$ en utilisant l'opérateur bind de la monade : $[\![G]\!]'\ \mu \stackrel{\text{def}}{=} \mathsf{bind}\ \mu\ [\![G]\!]$. Un des avantages principaux pour utiliser la monade sur $\mathcal{D}(\mathcal{M})$ est que les probabilités d'un évènement A, représenté comme un prédicat sur les mémoires qui renvoie un booléen, peut être facilement défini en utilisant la fonction caractéristique $\mathbb{I}_A$ de A :

$$\Pr[G, m : A] \stackrel{\text{def}}{=} [\![G]\!]\ m\ \mathbb{I}_A \tag{2.1}$$

Nous omettrons quelques fois de mentionner la mémoire m ; et dans ce cas la mémoire initiale sera un *mapping* des variables par leur valeur par défaut.

$$\begin{array}{ll}
[\![\mathsf{nil}]\!]\ m & = \mathsf{unit}\ m \\
[\![i;\ c]\!]\ m & = \mathsf{bind}\ ([\![i]\!]\ m)\ [\![c]\!] \\
[\![x \leftarrow e]\!]\ m & = \mathsf{unit}\ m\{[\![e]\!]_{\mathcal{E}}\ m/x\} \\
[\![x \xleftarrow{\$} d]\!]\ m & = \mathsf{bind}\ ([\![d]\!]_{\mathcal{D}}\ m)\ (\lambda v.\ \mathsf{unit}\ m\{v/x\}) \\
[\![x \leftarrow f(\mathbf{e})]\!]\ m & = \begin{array}{l}\mathsf{bind}\ ([\![E(f).\mathsf{body}]\!]\ (\emptyset\{[\![\mathbf{e}]\!]_{\mathcal{D}}\ m/E(f).\mathsf{params}\}, m.\mathsf{glob})) \\ \quad (\lambda m'.\ (m.\mathsf{loc}, m'.\mathsf{glob})\{[\![E(f).\mathsf{re}]\!]_{\mathcal{E}}\ m'/x\})\end{array} \\
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\ m & = \left\{\begin{array}{l}[\![c_1]\!]\ m\ \mathsf{if}\ [\![e]\!]_{\mathcal{E}}\ m = \mathsf{true} \\ [\![c_2]\!]\ m\ \mathsf{if}\ [\![e]\!]_{\mathcal{E}}\ m = \mathsf{false}\end{array}\right. \\
[\![\{0,1\}]\!]_{\mathcal{D}}\ m & = \lambda f.\ \frac{1}{2}\ f(\mathsf{true}) + \frac{1}{2}\ f(\mathsf{false}) \\
[\![[0..e]]\!]_{\mathcal{D}}\ m & = \lambda f.\ \sum_{i=0}^{n} \frac{1}{n+1}\ f(i) \quad \text{où } n = [\![e]\!]_{\mathcal{E}}\ m
\end{array}$$

Fig. 2.1. Sémantique dénotationnelle pour les jeux

2.2.2 Raisonnement sur les jeux

Dans les preuves par jeux, certaines étapes (*bringing step*) correspondent à des preuves que la sémantique est préservée suite à une transformation de programme. Si une étape passe du jeu G au jeu G' pour montrer que pour un évènement A du jeu G et un évènement A' du jeu G' $\Pr[G, m : A] = \Pr[G', m : A']$. La définition (2.1), montre qu'il suffit de prouver que $[\![G]\!]\ m\ \mathbb{I}_A = [\![G']\!]\ m\ \mathbb{I}_{A'}$, et en généralisant à une paire de mémoires initiales m_1, m_2, et deux mesures arbitraires $f, g : \mathcal{M} \to [0,1]$, que $[\![G]\!]\ m_1\ f = [\![G']\!]\ m_2\ g$.

Équivalence Observationelle

CertiCrypt permet de prouver de telles égalités en utilisant la logique relationnelle pRHL, qui généralise la logique de Hoare relationnelle [27] aux probabilités. Les jugements dans pRHL sont de la forme $G_1 \sim G_2 : \Psi \Rightarrow \Phi$, où G_1 et G_2 sont des jeux, et Ψ et Φ sont des relations sur des états déterministes. Le jugement $G_1 \sim G_2 : \Psi \Rightarrow \Phi$ n'est valide que si pour toute paire de mémoires initiales m_1, m_2 telle que $m_1\ \Psi\ m_2$, alors $[\![G_1]\!]\ m_1 \sim_{\Phi} [\![G_2]\!]\ m_2$ est vraie.

Le prédicat range permet de montrer qu'un prédicat P est valide dans tous les états qui ont une probabilité non nulle dans la distribution $\mathcal{D}$

$$\mathsf{range}\ P\ \mu \stackrel{\text{def}}{=} \forall f.\ (\forall a.\ P\ a \Rightarrow f\ a = 0) \Rightarrow \mu\ f = 0$$

La définition de $\sim_{\Phi}$ pour une relation arbitraire est la suivante :

$$\mu_1 \sim_{\Phi} \mu_2 \stackrel{\text{def}}{=} \exists \mu.\ \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \mathsf{range}\ \Phi\ \mu$$

où les projections de la distribution μ sont définies par :

$$\pi_1(\mu) \stackrel{\text{def}}{=} \mathsf{bind}\ \mu\ (\lambda p.\mathsf{unit}\ (\mathsf{fst}\ p)) \quad \pi_2(\mu) \stackrel{\text{def}}{=} \mathsf{bind}\ \mu\ (\lambda p.\mathsf{unit}\ (\mathsf{snd}\ p))$$

Pour raisonner sur les jugements pRHL, CertiCrypt fournit un ensemble de règles de dérivation et un calcul de plus faibles pré-conditions. Les règles se trouvent

dans [21]. Une importante implication d'un jugement $G_1 \sim G_2 : \Psi \Rightarrow \Phi$ est que si deux fonctions f et g ne peuvent pas distinguer deux mémoires pour une relation Φ , i.e.

$$\forall m_1\ m_2.\ m_1\ \Phi\ m_2 \Rightarrow f\ m_1 = g\ m_2$$

alors

$$\forall m_1\ m_2.\ m_1\ \Psi\ m_2 \Rightarrow [\![G_1]\!]\ m_1\ f = [\![G_2]\!]\ m_2\ g. \qquad (=_{[\![]\!]})$$

En particulier, si Φ est une égalité sur des variables libres d'un prédicat booléen A, on obtient $\Pr[G_1, m_1 : A] = \Pr[G_2, m_2 : A]$.

En spécialisant les jugements pRHL à des prédicats sur l'égalité d'ensemble de variables, nous obtenons la notion de non-interférence probabiliste : étant donné un ensemble X de variables, le prédicat $=_X$ est défini par

$$m_1 =_X m_2 \stackrel{\text{def}}{=} \forall x \in X, m_1\ x = m_2\ x.$$

La non-interférence probabiliste : pour deux programmes c_1 et c_2 et étant donné un ensemble I de variables d'entrée et un ensemble O de variables de sortie, la notation $c_1 \sim_O^I c_1$ signifie $c_1 \sim c_1 : =_I \Rightarrow =_O$.

Tactique

CertiCrypt fournit plusieurs outils pour raisonner sur la non-interférence. En particulier, un ensemble de tactiques permet de prouver des jugements pRHL, en les simplifiant ou les transformant.

La tactique `eqobs_in` représente une procédure de semi-décision pour les jugements de la forme $c, E \sim_O^I c, E'$. D'autres tactiques comme `eqobs_hd`, `eqobs_tl`, `eqobs_ctxt`, `deadcode`, et `swap` simplifient le but en utilisant des fonctions qui prennent deux jeux c_1, E_1 et c_2, E_2 et un ensemble de variables I,O et calculent c_1', c_2' et I', O' tels que

$$c_1', E_1 \sim_{O'}^{I'} c_2', E_2 \quad \Rightarrow \quad c_1, E_1 \sim_O^I c_2, E_2.$$

Les tactiques se différencient par leur manière de calculer c_1', c_2' et I', O'. La tactique `eqobs_tl` recherche le plus grand préfixe commun c tel que $c_1 = c; c_1'$ et $c_2 = c; c_2'$, `eqobs_hd` fait une recherche similaire mais pour un plus grand suffixe, et `eqobs_ctxt` combine les deux tactiques. La tactique `swap` réordonne les instructions pour générer le plus grand suffixe tout en préservant l'équivalence observationelle, c'est à dire $c_1' = \hat{c}_1; c$ et $c_2' = \hat{c}_2; c$ sont des permutations de c_1 et c_2 (et $I' = I$ et $O' = O$). La tactique `deadcode` produit un programme équivalent qui supprime les instructions qui n'influent pas O.

De plus, CertiCrypt permet d'automatiser d'autres transformations de programmes courantes : propagation d'expression (`ep`), allocation de variable (`alloc`), et dépliage de fonction (`inline`). La correction et la préservation de la non-interferance de toutes ces tactiques sont prouvées de manière reflexive en Coq. La tactique `sinline` combine `inline`, `alloc`, `ep`, et `deadcode` en une seule tactique.

2.3 Sécurité Sémantique de la primitive Hashed ElGamal

Prenons G un groupe cyclique d'ordre premier q et g un générateur, et $(H_k)_{k \in K}$ une famille de fonctions de hachage à clés, reliant les éléments de G à des bitstrings d'une longueur fixée ℓ. Hashed ElGamal est un schéma de chiffrement à clé publique dont la sécurité est reliée au problème du logarithme discret dans G. Les algorithmes de génération de clé, de chiffrement et déchiffrement sont définis de la manière suivante :

$$\begin{array}{ll} \mathcal{KG}(\,) & \stackrel{\text{def}}{=} k \stackrel{\$}{\leftarrow} K;\ x \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ \mathsf{return}\ ((k,x),(k,g^x)) \\ \mathsf{Enc}((k,\alpha),m) & \stackrel{\text{def}}{=} y \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ h \leftarrow H_k(\alpha^y);\ \mathsf{return}\ (g^y, h \oplus m) \\ \mathsf{Dec}((k,x),(\beta,\zeta)) & \stackrel{\text{def}}{=} h \leftarrow H_k(\beta^x);\ \mathsf{return}\ h \oplus \zeta \end{array}$$

L'ensemble des textes clairs m de Hashed ElGamal est $\{0,1\}^\ell$, contrairement au protocole original de ElGamal où l'espace des textes clairs est simplement G.

Dans la suite de cette section nous présenterons la preuve par jeux de la sécurité sémantique du chiffrement Hashed ElGamal dans deux contextes différents. La première preuve est faite dans le modèle standard de la cryptographie ; en faisant l'hypothèse que la famille de fonction de hachage $(H_k)_{k \in K}$ est *entropy smoothing*. La sécurité est ensuite réduite au problème DDH. La deuxième preuve est faite dans le *Random Oracle Model* (ROM) : les fonctions de hashage se comportent comme des fonctions aléatoires parfaites et la sécurité est réduite au problème CDH pour les listes.

La première étape des preuves dans CertiCrypt, est d'étendre la syntaxe et la sémantique des jeux, pour y ajouter les nouveaux types et opérateurs utilisés dans la description de la primitive, qui ne sont pas déjà inclus dans les types ou opérateurs de base de CertiCrypt. Cette extension est faite de manière modulaire. Nous déclarons une famille de groupes cycliques $(G_\eta)_{\eta \in \mathbb{N}}$ indexée par le paramètre de sécurité et étendons les types de CertiCrypt avec le type groupe, pour déclarer les éléments de G_η et les bitstrings de taille ℓ. Les distributions sont également étendues avec le tirage aléatoire dans les bitstrings de longueur ℓ. les nouveaux opérateurs ajoutés sont q et g représentant l'ordre et un générateur de G_η. Des opérateurs binaires comme la multiplication, ou la fonction puissance dans les groupes sont ajoutées, ainsi que le OU exclusif pour les bitstrings de longueur ℓ. Dans la preuve dans le modèle standard, la fonction de hachage est représentée par un opérateur binaire prenant une clé dans K et un élément de G_η et renvoie un bitstring de longueur ℓ, tandis que la preuve dans le *random oracle model* les fonctions de hachage sont des procédures définies dans CertiCrypt.

2.3.1 Sécurité dans le modèle standard

La preuve est basée sur deux hypothèses : la famille de fonctions de hachage $(H_k)_{k \in K}$ est *entropy smoothing*, et la complixité du problème DDH pour G_η. La deuxième est déjà expliquée dans la Définition 2.1. La première est expliquée ci dessous :

Définition 2.2 (Hypothèse *Entropy Smoothing* (ES)). *Considérant les jeux*

Game ES_0 :	**Game** ES_1 :
$\boldsymbol{k} \xleftarrow{\$} K;\ h \xleftarrow{\$} \{0,1\}^\ell;$ $d \leftarrow \mathcal{D}(h)$	$\boldsymbol{k} \xleftarrow{\$} K;\ z \xleftarrow{\$} \mathbb{Z}_q;$ $d \leftarrow \mathcal{D}(H_{\boldsymbol{k}}(g^z))$

et définissant

$$\epsilon_{\mathsf{ES}}(\eta) \stackrel{def}{=} |\Pr[\mathsf{ES}_0 : d = 1] - \Pr[\mathsf{ES}_1 : d = 1]|$$

alors pour tout adversaire PPT $\mathcal{D}$, ϵ_{ES} est une fonction négligeable.

L'hypothèse ES consiste à dire que pour une famille de fonction de hachage, en tirant aléatoirement une fonction de hashage dans cette famille, le résultat est indistinguable d'un tirage aléatoire dans $\{0,1\}^\ell$. Un adversaire donnera le même résultat dans le jeu ES_0 et le jeu ES_1.

Shoup explique [80] qu'il est possible de construire des familles de fonctions de hachage qui respectent cette hypothèse en utilisant le *Leftover Hash Lemma* de [**?**].

Pour clarifier les définitions des jeux, nous choisirons de mettre l'indice de la fonction de hachage dans la variable globale $\boldsymbol{k}$; au lieu de passer dans les paramètres de chaque fonction.

Théorème 2.3 (Sécurité de Hashed ElGamal dans le modèle standard). *Pour tout adversaire $(\mathcal{A}, \mathcal{A}')$ bien formé et PPT,*

$$\left|\Pr[\mathit{IND\text{-}CPA} : d] - \frac{1}{2}\right| \leq \epsilon_{\mathit{DDH}}(\eta) + \epsilon_{\mathsf{ES}}(\eta)$$

donc sous les hypothèses DDH (2.1) et ES (2.2), $|\Pr[\mathit{IND\text{-}CPA} : d] - \frac{1}{2}|$ est négligeable, étant donné que $\epsilon_{\mathit{DDH}}(\eta)$ et $\epsilon_{\mathsf{ES}}(\eta)$ sont négligeables.

Nous rappelons que $\mathcal{A}$ et $\mathcal{A}'$ sont des paramètres du jeux IND-CPA. Les hypothèses faites sur les adversaires portent sur la complexité et sur le fait que les adversaires sont bien formés.

CertiCrypt contient une procédure d'analyse pour vérifier si les adversaires respectent les règles données dans [21]. Les règles garantissent qu'à chaque fois qu'une variable est écrite l'adversaire a le droit de le faire, c'est à dire si c'est une variable globale auquel il a accès ou bien une variable locale, précédemment initialisée. De même pour les oracles, les règles vérifient si les oracles qu'appelle l'adversaire sont bien dans la liste.

La figure 2.2 donne une vue d'ensemble de la preuve ; les scripts de preuve apparaissent dans les boites grises. L'adversaire est défini comme deux procédures partageant un état à travers un ensemble de variables globales appelé $\mathcal{G}_{\mathcal{A}}$. La condition de bonne fondaison déclare que les adversaires n'ont qu'un accès en lecture seule à $\boldsymbol{k}$ et ils ne peuvent pas appeler les procédures nommées $\mathcal{B}$ ou $\mathcal{D}$ étant donné que ces noms sont réservés pour les adversaires dans la réduction. L'adversaire est libre de définir et d'appeler d'autres procédures en privé tant qu'elles sont bien formées.

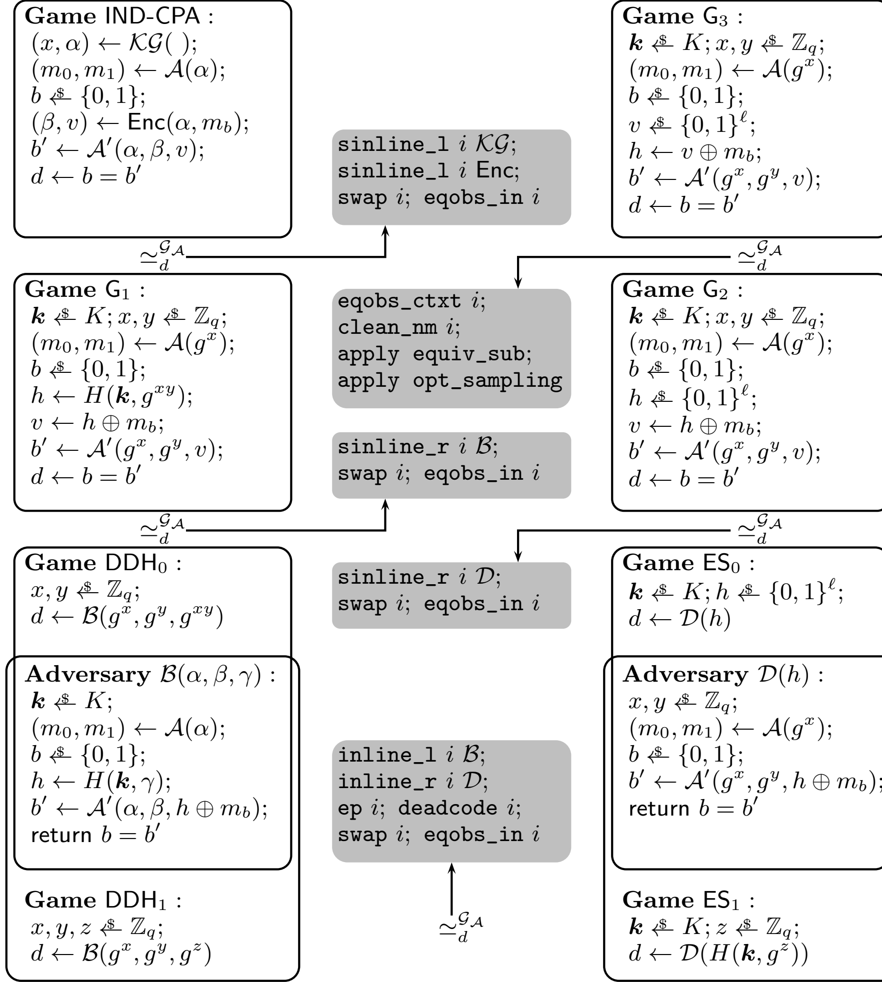


Fig. 2.2. Preuve par jeux de la sécurité du chiffrement Hashed ElGamal dans le modèle standard

La première étape de la preuve consiste à prouver

$$\Pr[\mathsf{IND\text{-}CPA} : d] = \Pr[\mathsf{DDH}_0 : d] \tag{2.2}$$

Pour clarifier la preuve, nous introduisons le jeu intermédiaire G_1 et montrons que

$$\mathsf{IND\text{-}CPA} \sim_d^{\mathcal{G}_\mathcal{A}} \mathsf{G}_1 \quad \text{et} \quad \mathsf{G}_1 \sim_d^{\mathcal{G}_\mathcal{A}} \mathsf{DDH}_0$$

La relation $\sim$ étant transitive,

$$\frac{c \sim_O^I c' \quad c' \sim_O^I c''}{c \sim_O^I c''}\ [\mathsf{R\text{-}Trans}]$$

nous obtenons $\mathsf{IND\text{-}CPA} \sim_d^{\mathcal{G}_\mathcal{A}} \mathsf{DDH}_0$ et l'équation s'obtient avec (2.2) et ($=_{[\![]\!]}$). Nous montrons ensuite que

$$\Pr[\mathsf{DDH}_1 : d] = \Pr[\mathsf{ES}_1 : d] \tag{2.3}$$

Nous commençons par prouver $\mathsf{DDH}_1 \sim_d^{\mathcal{G}_\mathcal{A}} \mathsf{ES}_1$. Nous expliquerons cette transition en détail en montrant les buts intermédiaires obtenus après avoir appliqué chaque tactique du script de preuve.

$x, y, z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ d \leftarrow \mathcal{B}(g^x, g^y, g^z)$	$\simeq_d^{\mathcal{G}_\mathcal{A}}$	$\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ d \leftarrow \mathcal{D}(H(\boldsymbol{k}, g^z))$
	`inline_l` $i\ \mathcal{B}$; `inline_r` $i\ \mathcal{D}$	
$x, y, z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ \alpha \leftarrow g^x; \beta \leftarrow g^y; \gamma \leftarrow g^z;$ $\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ (m_0, m_1) \leftarrow \mathcal{A}(\alpha);$ $b \stackrel{\$}{\leftarrow} \{0,1\};\ h \leftarrow H(\boldsymbol{k}, \gamma);$ $b' \leftarrow \mathcal{A}'(\alpha, \beta, h \oplus m_b);\ d \leftarrow b = b'$	$\simeq_d^{\mathcal{G}_\mathcal{A}}$	$\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ h \leftarrow H(\boldsymbol{k}, g^z);$ $x, y \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ (m_0, m_1) \leftarrow \mathcal{A}(g^x);$ $b \stackrel{\$}{\leftarrow} \{0,1\};$ $b' \leftarrow \mathcal{A}'(g^x, g^y, h \oplus m_b);\ d \leftarrow b = b'$
	`ep` i	
$x, y, z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ \alpha \leftarrow g^x; \beta \leftarrow g^y; \gamma \leftarrow g^z;$ $\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ (m_0, m_1) \leftarrow \mathcal{A}(g^x);$ $b \stackrel{\$}{\leftarrow} \{0,1\};\ h \leftarrow H(\boldsymbol{k}, g^z);$ $b' \leftarrow \mathcal{A}'(g^x, g^y, H(\boldsymbol{k}, g^z) \oplus m_b);$ $d \leftarrow b = b'$	$\simeq_d^{\mathcal{G}_\mathcal{A}}$	$\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ h \leftarrow H(\boldsymbol{k}, g^z);$ $x, y \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ (m_0, m_1) \leftarrow \mathcal{A}(g^x);$ $b \stackrel{\$}{\leftarrow} \{0,1\};$ $b' \leftarrow \mathcal{A}'(g^x, g^y, H(\boldsymbol{k}, g^z) \oplus m_b);$ $d \leftarrow b = b'$
	`deadcode` i	
$x, y, z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;$ $\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ (m_0, m_1) \leftarrow \mathcal{A}(g^x);$ $b \stackrel{\$}{\leftarrow} \{0,1\};$ $b' \leftarrow \mathcal{A}'(g^x, g^y, H(\boldsymbol{k}, g^z) \oplus m_b);$ $d \leftarrow b = b'$	$\simeq_d^{\mathcal{G}_\mathcal{A}}$	$\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;$ $x, y \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ (m_0, m_1) \leftarrow \mathcal{A}(g^x);$ $b \stackrel{\$}{\leftarrow} \{0,1\};$ $b' \leftarrow \mathcal{A}'(g^x, g^y, H(\boldsymbol{k}, g^z) \oplus m_b);$ $d \leftarrow b = b'$
	`swap` i	
$x, y, z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;$ $\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ (m_0, m_1) \leftarrow \mathcal{A}(g^x);$ $b \stackrel{\$}{\leftarrow} \{0,1\};$ $b' \leftarrow \mathcal{A}'(g^x, g^y, H(\boldsymbol{k}, g^z) \oplus m_b);$ $d \leftarrow b = b'$	$\simeq_d^{\mathcal{G}_\mathcal{A}}$	$x, y, z \stackrel{\$}{\leftarrow} \mathbb{Z}_q;$ $\boldsymbol{k} \stackrel{\$}{\leftarrow} K;\ (m_0, m_1) \leftarrow \mathcal{A}(g^x);$ $b \stackrel{\$}{\leftarrow} \{0,1\};$ $b' \leftarrow \mathcal{A}'(g^x, g^y, H(\boldsymbol{k}, g^z) \oplus m_b);$ $d \leftarrow b = b'$
	`eqobs_in` i	

Nous commençons par déplier les appels aux procédures $\mathcal{B}$ et $\mathcal{D}$ dans chaque jeu. Lors de ce dépliage, les expressions qui apparaissent dans la liste des paramètres actuels sont assignées aux paramètres formels correspondant qui apparaissent dans la déclaration. De la même manière l'expression *return* est assignée à la variable de retour. Nous utilisons ensuite la tactique `ep` pour propager les assignements dans tout le jeu et la tactique `deadcode` élimine les instructions qui n'affectent pas —directement ou indirectement— la valeur de d. La tactique `sinline` produit le même résultat que la combinaison des tactiques `inline; ep; deadcode`. À ce point de la preuve, les deux programmes sont identiques, modulo l'ordre des instructions ; nous utilisons la tactique `swap` pour réordonner les instructions du programme du coté droit dans le même ordre que celui de gauche. La tactique `eqobs_in` termine la preuve, en effectuant une analyse de dépendance pour montrer que la valeur de d dépend uniquement de la valeur initiale des variables dans $\mathcal{G}_\mathcal{A}$.

Finalement, nous montrons que

$$\Pr[\mathsf{ES}_0 : d] = \Pr[\mathsf{G}_3 : d]. \tag{2.4}$$

Comme ci-dessus, nous introduisons un jeu intermédiaire, G_2 et prouvons $\mathsf{ES}_0 \sim_d^{\mathcal{G}_\mathcal{A}} \mathsf{G}_2$, et $\mathsf{G}_2 \sim_d^{\mathcal{G}_\mathcal{A}} \mathsf{G}_3$. Par [R-Trans] nous obtenons $\mathsf{ES}_0 \sim_d^{\mathcal{G}_\mathcal{A}} \mathsf{G}_3$ qui par ($=_{[\![]\!]}$) donne (2.4). La transition du jeu ES_0 au jeu G_2 est similaire à celle détaillée ci-dessus. Cependant, la transition du jeu G_2 au jeu G_3 est plus intéressante car elle utilise une propriété algébrique du $\oplus$ appelée *optimistic sampling* :

$$x \xleftarrow{\$} \{0,1\}^\ell; y \leftarrow x \oplus z \sim_{\{x,y,z\}}^{\{z\}} y \xleftarrow{\$} \{0,1\}^\ell; x \leftarrow y \oplus z \tag{2.5}$$

Nous expliquons à nouveau comment se déroule l'interaction pas à pas avec CertiCrypt :

$\boldsymbol{k} \xleftarrow{\$} K;\ x, y \xleftarrow{\$} \mathbb{Z}_q;$ $(m_0, m_1) \leftarrow \mathcal{A}(g^x);\ b \xleftarrow{\$} \{0,1\};$ $h \xleftarrow{\$} \{0,1\}^\ell;\ v \leftarrow h \oplus m_b;$ $b' \leftarrow \mathcal{A}'(g^x, g^y, v);$ $d \leftarrow b = b'$	$\simeq_d^{\mathcal{G}_\mathcal{A}}$	$\boldsymbol{k} \xleftarrow{\$} K;\ x, y \xleftarrow{\$} \mathbb{Z}_q;$ $(m_0, m_1) \leftarrow \mathcal{A}(g^x);\ b \xleftarrow{\$} \{0,1\};$ $v \xleftarrow{\$} \{0,1\}^\ell;\ h \leftarrow v \oplus m_b;$ $b' \leftarrow \mathcal{A}'(g^x, g^y, v);$ $d \leftarrow b = b'$
	`eqobs_ctxt` i	
$h \xleftarrow{\$} \{0,1\}^\ell;\ v \leftarrow h \oplus m_b$	$\simeq_{\{k,x,y,b,v\} \cup \mathcal{G}_\mathcal{A}}^{\{k,x,y,m_0,m_1,b\} \cup \mathcal{G}_\mathcal{A}}$	$v \xleftarrow{\$} \{0,1\}^\ell;\ h \leftarrow v \oplus m_b$
	`clean_nm`	
$h \xleftarrow{\$} \{0,1\}^\ell; v \leftarrow h \oplus m_b$	$\simeq_{\{v\}}^{\{k,x,y,m_0,m_1,b\} \cup \mathcal{G}_\mathcal{A}}$	$v \xleftarrow{\$} \{0,1\}^\ell;\ h \leftarrow v \oplus m_b$
	`apply equiv_sub`	
$h \xleftarrow{\$} \{0,1\}^\ell;\ v \leftarrow h \oplus m_b$	$\simeq_{\{h,v,m_b\}}^{\{m_b\}}$	$v \xleftarrow{\$} \{0,1\}^\ell;\ h \leftarrow v \oplus m_b$
	`apply opt_sampling`	

Pour commencer, la tactique `eqobs_ctxt` est utilisée pour supprimer les préfixes et suffixes communs dans les programmes (`eqobs_ctxt` est la combinaison des tactiques `eqobs_hd` et `eqobs_tl`). Ensuite, la tactique `clean_nm` enlève de l'ensemble de sortie, les variables apparaissant dans l'ensemble d'entrée qui ne sont pas modifiées dans les programmes. Cette tactique se justifie par la règle suivante :

$$\frac{X \cap \mathsf{modifies}(c_1) = \emptyset \quad X \cap \mathsf{modifies}(c_2) = \emptyset \qquad X \subseteq I \quad c_1 \sim_O^I c_2}{c_1 \sim_{O \cup X}^I c_2}$$

Pour finir, nous appliquons la règle suivante (equiv_sub)

$$\frac{I \subseteq I' \qquad c_1 \sim_{O'}^{I'} c_2 \quad O' \subseteq O}{c_1 \sim_O^I c_2} \text{[R-Sub]}$$

qui permet d'affaiblir la pré-condition et de renforcer la post-condition. Ces changements nous permettent d'obtenir un but compatible avec le lemme *optimistic sampling* (2.5), qui nous permet de finir la preuve.

La dernière transition supprime la dépendance entre v et b, et donc la dépendance entre b' et b. Ce qui nous permet de prouver que

$$\Pr\left[\mathsf{G}_3 : d\right] = \frac{1}{2} \tag{2.6}$$

Nous utilisons les tactiques `swap` et `deadcode` pour placer le tirage de b après l'appel à $\mathcal{A}'$. Nous pouvons ensuite utiliser la propriété suivante :

$$\forall G\ e\ d,\ \Pr\left[G; d \leftarrow e : d\right] = \Pr\left[G : e\right]$$

et nous avons alors $\Pr\left[\mathsf{G}_3 : d\right] = \Pr\left[\mathsf{G}_3 : b = b'\right]$.

Pour conclure, nous utilisons le fait que pour tout jeu G et pour toutes variables b, b', $\Pr\left[G; b \stackrel{\$}{\leftarrow} \{0,1\} : b = b'\right] = \frac{1}{2}$. Cette propriété requiert que G termine.

CertiCrypt fournit une procédure de semi-décision pour prouver la terminaison d'un programme. Dans notre cas, G_3 termine si $\mathcal{A}$ et $\mathcal{A}'$ terminent, que nous avons avec les hypothèses que $\mathcal{A}$ et $\mathcal{A}'$ sont PPT.

Pour résumer, à partir des équations (2.2), (2.3), (2.4), et (2.6) nous obtenons

$$\begin{array}{rcl}
|\Pr\left[\textsf{IND-CPA} : d\right] - \frac{1}{2}| & = & |\Pr\left[\textsf{DDH}_0 : d\right] - \frac{1}{2}| \\
& = & |\Pr\left[\textsf{DDH}_0 : d\right] - \Pr\left[\textsf{DDH}_1 : d\right] + \Pr\left[\textsf{DDH}_1 : d\right] - \frac{1}{2}| \\
& \leq & |\Pr\left[\textsf{DDH}_0 : d\right] - \Pr\left[\textsf{DDH}_1 : d\right]| + |\Pr\left[\textsf{DDH}_1 : d\right] - \frac{1}{2}| \\
& = & |\Pr\left[\textsf{DDH}_0 : d\right] - \Pr\left[\textsf{DDH}_1 : d\right]| + |\Pr\left[\textsf{ES}_1 : d\right] - \frac{1}{2}| \\
& = & |\Pr\left[\textsf{DDH}_0 : d\right] - \Pr\left[\textsf{DDH}_1 : d\right]| + |\Pr\left[\textsf{ES}_1 : d\right] - \Pr\left[\textsf{G}_3 : d\right]| \\
& = & |\Pr\left[\textsf{DDH}_0 : d\right] - \Pr\left[\textsf{DDH}_1 : d\right]| + |\Pr\left[\textsf{ES}_1 : d\right] - \Pr\left[\textsf{ES}_0 : d\right]| \\
& = & \epsilon_{\textsf{DDH}}(\eta) + \epsilon_{\textsf{ES}}(\eta)
\end{array}$$

Nous prouvons donc que $|\Pr\left[\textsf{IND-CPA} : d\right] - \frac{1}{2}$ est négligeable, sous les hypothèses DDH et ES. Pour finir la preuve, nous devons vérifier que les adversaires $\mathcal{B}$ et $\mathcal{D}$ utilisés dans la réduction sont des procédures PPT. La tactique `PPT_proc` prouve automatiquement que ces procédures sont PPT à partir des hypothèses que $\mathcal{A}$ et $\mathcal{A}'$ sont PPT

2.3.2 Sécurité dans le *Random Oracle Model*

Le chiffrement Hashed ElGamal est sûr, dans le *random oracle model* sous l'hypothèse *Computational Diffie-Hellman* CDH sur une famille de groupes $(G_\eta)_{\eta \in \mathbb{N}}$. Cette hypothèse affirme qu'il est difficile de calculer g^{xy} étant donnés g^x et g^y avec x et y tirés uniformément dans $\mathbb{Z}_q$.

Hypothèses autour du logarithme discret

Nous allons clarifier les différentes hypothèses faites autour du logarithme discret. En particulier les concepts de réductibilité, d'hypothèses plus fortes ou plus faibles... ou les relations entre les problèmes DDH, CDH et le logarithme discret DL. Dans la suite nous utilisons les définitions suivantes :

– DL : étant donné g, g^x il est difficile de trouver x

– CDH : étant donné g^x et g^y il est difficile de trouver g^{xy}

– DDH : les triplets (g^x, g^y, g^{xy}) et (g^x, g^y, g^z) sont difficiles à distinguer avec x, y, z tirés uniformément.

Nous avons les relations suivantes :

$$\mathsf{DDH} \Rightarrow \mathsf{CDH} \Rightarrow \mathsf{DL}$$

Ce qui veut dire que l'hypothèse DDH est plus forte que l'hypothèse CDH et CDH est plus forte que DL . Un protocole sera plus sûr s'il fait une hypothèse plus faible

En terme de réductibilité, un problème A est réductible au problème B, si à partir des solutions de B on peut trouver les solutions de A.

Le problème DDH est réductible au problème CDH. Avec les paires de triplet (g^x, g^y, g^{xy}) et (g^x, g^y, g^z), les solutions de CDH avec g^x et g^y permettent de tester la troisième composante du triplet (sauf dans le cas où $g^{xy} = g^z$).

De plus, CDH est réductible à DL, il suffit d'appliquer le logarithme discret sur g^x ou g^y puis de calculer g^{xy}. Une autre manière de comprendre la réductibilité, est que si le logarithme discret devient facile alors, alors l'hypothèse DL est fausse.

La contraposée n'est pas nécessairement vraie, il existe des groupes où l'hypothèse DL semble vraie (cela reste une hypothèse), mais par contre l'hypothèse DDH est fausse, dans les autres cas c'est un problème ouvert.

La relation entre CDH et DL est moins claire, pour certains groupes, il est possible de prouver que les deux problèmes sont équivalents, mais plus généralement la réductibilité de DL vers CDH est un problème ouvert [68].

Nous considérons une version un peu différente de CDH, mais qui est équivalente :

Définition 2.4 (Hypothèse LCDH (list CDH)). *Étant donné le jeu*

Game LCDH :
$x, y \stackrel{\$}{\leftarrow} \mathbb{Z}_q$;
$L \leftarrow \mathcal{C}(g^x, g^y)$

et définissant

$$\epsilon_{\mathsf{LCDH}}(\eta) \stackrel{def}{=} \Pr[\mathsf{LCDH} : g^{xy} \in L]$$

Alors, pour tout adversaire PPT, ϵ_{LCDH} est une fonction négligeable.

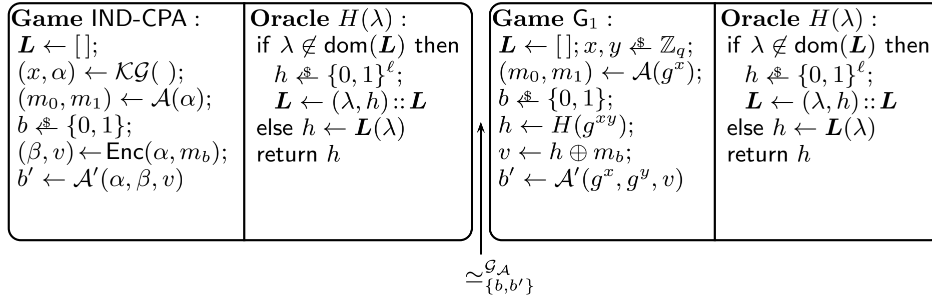
Les hypothèses CDH et LCDH sont équivalentes, à partir d'un adversaire contre LCDH qui a un avantage non négligeable, il est possible de construire un adversaire contre CDH, en tirant un élément dans la liste L ; la taille de la liste L étant de taille polynomiale, l'avantage de casser CDH est encore négligeable (il y a une perte de sécurité proportiennelle à la taille de la liste L). Dans l'autre sens, un adversaire cassant CDH, peut facilement construire cette liste L.

Théorème 2.5 (Sécurité de Hashed ElGamal dans ROM). *Pour tous adversaires* $(\mathcal{A}, \mathcal{A}')$ *bien formés et PPT*

$$\left|\Pr\left[\mathit{IND\text{-}CPA} : d\right] - \frac{1}{2}\right| \leq \epsilon_{\mathsf{LCDH}}(\eta)$$

alors, sous l'hypothèse LCDH, $|\Pr\left[\mathit{IND\text{-}CPA} : d\right] - \frac{1}{2}|$ *est négligeable.*

Par rapport à la preuve dans le modèle standard, l'hypothèse sur la famille de groupes est probablement plus faible, par contre l'hypothèse que nous faisons sur la famille de fonction de hachage est plus forte. Dans le modèle de l'oracle aléatoire, les fonctions de hachage sont de vraies fonctions aléatoires représentées comme des procédures à états. Les requêtes renvoient un résultat consistant : si une valeur est demandée deux fois, la fonction retourne le même résultat. Dans ce modèle, les fonctions de hachage n'ont plus besoin d'être indexées par une clé, contrairement à la première preuve.

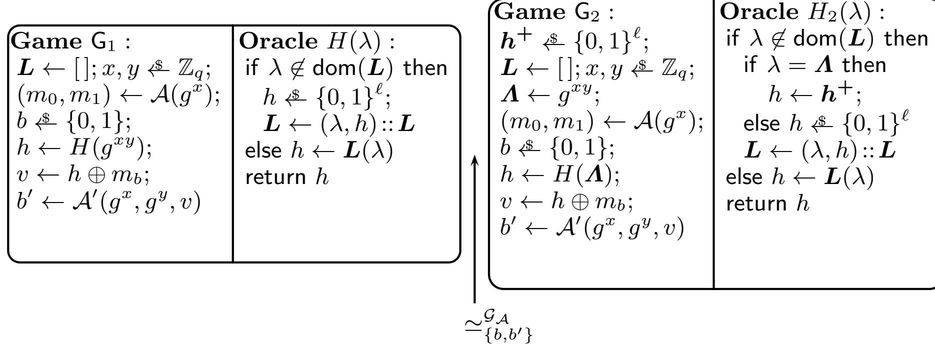


Comme pour la preuve dans le modèle standard, nous commençons par déplier les appels aux procédures $\mathcal{KG}$ et Enc dans le jeu IND-CPA pour obtenir un jeu équivalent au jeu G_1 et tel que

$$\Pr\left[\mathsf{IND\text{-}CPA} : b = b'\right] = \Pr\left[\mathsf{G}_1 : b = b'\right]. \tag{2.7}$$

Le but des prochaines transformations est de supprimer la dépendance entre b et v, ce qui permettra de placer le tirage dans b à la fin, et montrer que la probabilité est proche de $\frac{1}{2}$. Nous allons utiliser la même technique que pour la preuve dans le modèle standard.

Pour supprimer la dépendance entre b et v, la preuve va transformer l'appel $h \leftarrow H(g^{xy})$ en $h \overset{\$}{\leftarrow} \{0,1\}^\ell$. Le problème qui va se poser, est que si l'appel $H(g^{xy})$ est supprimé, cela signifie que la mémoire de l'oracle va changer et si l'adversaire appelle H avec l'instruction g^{xy}, le jeu aura un comportement différent.



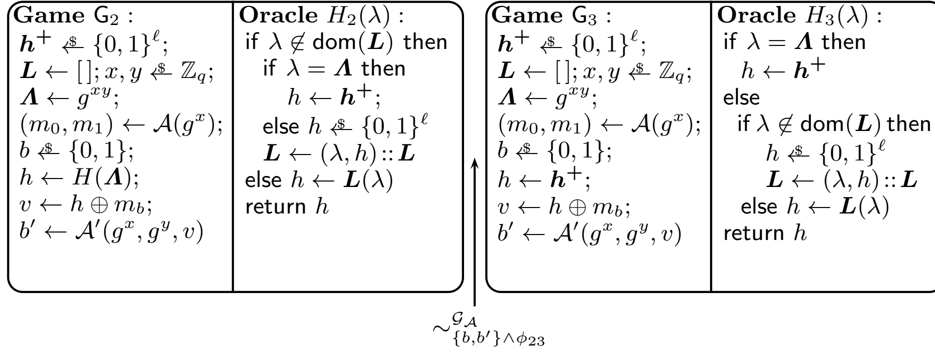
Pour raisonner sur $H(g^{xy})$ nous allons faire le tirage aléatoire dans le jeu et non pas dans l'oracle. Dans le jeu G_2, nous fixons ensuite la valeur h^+ qui représente la valeur de l'appel à $H(g^{xy})$. La variable globale Λ contient g^{xy}. Pour que les jeux aient la même sémantique, l'oracle est aussi modifié, si la requête est Λ la réponse sera h^+. Cette étape est une instance de *lazy sampling* expliqué dans [21]. Nous obtenons

$$\Pr\left[\mathsf{G}_1 : b = b'\right] = \Pr\left[\mathsf{G}_2 : b = b'\right]. \tag{2.8}$$

Nous allons maintenant, supprimer g^{xy} de $\boldsymbol{L}$; ce qui nous permettra de supprimer $\boldsymbol{h}^+$ du code de H. Nous prouvons que les jeux G_2 et G_3 sont équivalents sous l'invariant suivant :

$$\begin{aligned}\phi_{23} \stackrel{\text{def}}{=} & (\boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L}) \Longrightarrow \boldsymbol{L}[\boldsymbol{\Lambda}] = \boldsymbol{h}^+)\langle 1\rangle \wedge \\ & \forall \lambda, \lambda \neq \boldsymbol{\Lambda}\langle 1\rangle \Longrightarrow \boldsymbol{L}[\lambda]\langle 1\rangle = \boldsymbol{L}[\lambda]\langle 2\rangle\end{aligned}$$

où $e\langle 1\rangle$ (resp., $e\langle 2\rangle$) signifie la valeur de l'expression e dans le programme de gauche (resp. droit). L'invariant signifie que la liste d'association $\boldsymbol{L}$ de l'oracle H coïncide sauf pour Λ où le jeu G_2 va stocker la requête dans la liste.



Nous prouvons l'invariant sur le code des oracles H_2 et H_3. Nous décomposons la preuve en plusieurs étapes : nous avons la pré-condition ϕ_{23}, et nous prouvons qu'elle

est vraie après l'appel à $\mathcal{A}$, étant donné que H_2 et H_3 respectent ϕ_{23}. Nous pouvons ensuite déplier et utiliser l'invariant pour prouver que $\mathsf{G}_2 \sim \mathsf{G}_3 : \mathcal{G}_\mathcal{A} \Rightarrow \{b, b'\} \wedge \phi_{23}$, et conclure par

$$\Pr[\mathsf{G}_2 : b = b'] = \Pr[\mathsf{G}_3 : b = b'] . \tag{2.9}$$

Nous annulons les modifications faites sur l'oracle et prouvons que les jeux G_3 et G_4 sont équivalents, i.e. $\mathsf{G}_3 \sim^{\mathcal{G}_\mathcal{A}}_{\{b,b'\}} \mathsf{G}_4$

Game G_3 :	**Oracle** $H_3(\lambda)$:	**Game** G_4	**Oracle** $H_4(\lambda)$:
$\boldsymbol{h}^+ \stackrel{\$}{\leftarrow} \{0,1\}^\ell$;	if $\lambda = \boldsymbol{\Lambda}$ then	**bad** $\leftarrow$ false;	if $\lambda \notin \mathsf{dom}(\boldsymbol{L})$ then
$\boldsymbol{L} \leftarrow [\,]; x, y \stackrel{\$}{\leftarrow} \mathbb{Z}_q$;	$h \leftarrow \boldsymbol{h}^+$	$\boldsymbol{h}^+ \stackrel{\$}{\leftarrow} \{0,1\}^\ell$;	if $\lambda = \boldsymbol{\Lambda}$ then
$\boldsymbol{\Lambda} \leftarrow g^{xy}$;	else	$\boldsymbol{L} \leftarrow [\,]; x, y \stackrel{\$}{\leftarrow} \mathbb{Z}_q$;	**bad** $\leftarrow$ true;
$(m_0, m_1) \leftarrow \mathcal{A}(g^x)$;	if $\lambda \notin \mathsf{dom}(\boldsymbol{L})$ then	$\boldsymbol{\Lambda} \leftarrow g^{xy}$;	$h \leftarrow \boldsymbol{h}^+$
$b \stackrel{\$}{\leftarrow} \{0,1\}$;	$h \stackrel{\$}{\leftarrow} \{0,1\}^\ell$	$(m_0, m_1) \leftarrow \mathcal{A}(g^x)$;	else $h \stackrel{\$}{\leftarrow} \{0,1\}^\ell$
$h \leftarrow \boldsymbol{h}^+$;	$\boldsymbol{L} \leftarrow (\lambda, h) :: \boldsymbol{L}$	$b \stackrel{\$}{\leftarrow} \{0,1\}$;	$\boldsymbol{L} \leftarrow (\lambda, h) :: \boldsymbol{L}$
$v \leftarrow h \oplus m_b$;	else $h \leftarrow \boldsymbol{L}(\lambda)$	$v \leftarrow \boldsymbol{h}^+ \oplus m_b$;	else $h \leftarrow \boldsymbol{L}(\lambda)$
$b' \leftarrow \mathcal{A}'(g^x, g^y, v)$	return h	$b' \leftarrow \mathcal{A}'(g^x, g^y, v)$	return h

$\sim^{\mathcal{G}_\mathcal{A}}_{\{b,b'\}}$

nous obtenons :

$$\Pr[\mathsf{G}_3 : b = b'] = \Pr[\mathsf{G}_4 : b = b'] \tag{2.10}$$

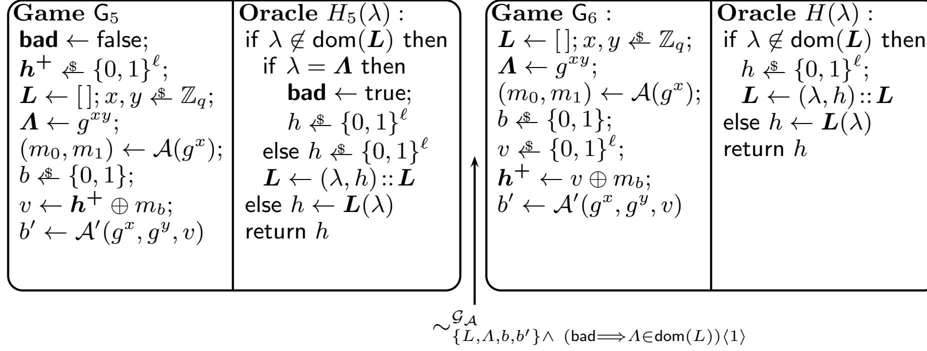
Pour pouvoir appeler optimistic sampling, deux solutions sont possibles : soit il faut déplacer l'instruction $v \leftarrow h \oplus m_b$ en dessous de $\boldsymbol{h}^+ \stackrel{\$}{\leftarrow} \{0,1\}^\ell$ mais ce n'est pas possible car m_b dépend de $b \stackrel{\$}{\leftarrow} \{0,1\}$, soit déplacer $\boldsymbol{h}^+ \stackrel{\$}{\leftarrow} \{0,1\}^\ell$ au-dessus de $\boldsymbol{h}^+ \stackrel{\$}{\leftarrow} \{0,1\}^\ell$, mais $\mathcal{A}$ peut appeler H et $\boldsymbol{h}^+$ apparaît dans H. Nous allons supprimer $\boldsymbol{h}^+$ de H.

Game G_4	**Oracle** $H_4(\lambda)$:	**Game** G_5	**Oracle** $H_5(\lambda)$:
bad $\leftarrow$ false;	if $\lambda \notin \mathsf{dom}(\boldsymbol{L})$ then	**bad** $\leftarrow$ false;	if $\lambda \notin \mathsf{dom}(\boldsymbol{L})$ then
$\boldsymbol{h}^+ \stackrel{\$}{\leftarrow} \{0,1\}^\ell$;	if $\lambda = \boldsymbol{\Lambda}$ then	$\boldsymbol{h}^+ \stackrel{\$}{\leftarrow} \{0,1\}^\ell$;	if $\lambda = \boldsymbol{\Lambda}$ then
$\boldsymbol{L} \leftarrow [\,]; x, y \stackrel{\$}{\leftarrow} \mathbb{Z}_q$;	**bad** $\leftarrow$ true;	$\boldsymbol{L} \leftarrow [\,]; x, y \stackrel{\$}{\leftarrow} \mathbb{Z}_q$;	**bad** $\leftarrow$ true;
$\boldsymbol{\Lambda} \leftarrow g^{xy}$;	$h \leftarrow \boldsymbol{h}^+$	$\boldsymbol{\Lambda} \leftarrow g^{xy}$;	$h \stackrel{\$}{\leftarrow} \{0,1\}^\ell$
$(m_0, m_1) \leftarrow \mathcal{A}(g^x)$;	else $h \stackrel{\$}{\leftarrow} \{0,1\}^\ell$	$(m_0, m_1) \leftarrow \mathcal{A}(g^x)$;	else $h \stackrel{\$}{\leftarrow} \{0,1\}^\ell$
$b \stackrel{\$}{\leftarrow} \{0,1\}$;	$\boldsymbol{L} \leftarrow (\lambda, h) :: \boldsymbol{L}$	$b \stackrel{\$}{\leftarrow} \{0,1\}$;	$\boldsymbol{L} \leftarrow (\lambda, h) :: \boldsymbol{L}$
$v \leftarrow \boldsymbol{h}^+ \oplus m_b$;	else $h \leftarrow \boldsymbol{L}(\lambda)$	$v \leftarrow \boldsymbol{h}^+ \oplus m_b$;	else $h \leftarrow \boldsymbol{L}(\lambda)$
$b' \leftarrow \mathcal{A}'(g^x, g^y, v)$	return h	$b' \leftarrow \mathcal{A}'(g^x, g^y, v)$	return h

$\sim^{\mathcal{G}_\mathcal{A}}_{\{b,b'\}}$

Les jeux G_4 et G_5 sont syntaxiquement équivalents jusqu'au moment où le "drapeau" booléen **bad** est levé. Le lemme fondamental décrit dans la Section 2.1, montrant que la différence des probabilités de tout évènement est bornée par la probabilité de **bad** dans le jeu G_5 nous permet de déduire :

$$|\Pr[\mathsf{G}_4 : b = b'] - \Pr[\mathsf{G}_5 : b = b']| \leq \Pr[\mathsf{G}_5 : \textbf{bad}] . \tag{2.11}$$



Nous prouvons ensuite que

$$\mathsf{G}_5 \sim \mathsf{G}_6 : =_{\mathcal{G_A}} \Rightarrow =_{\{L,\Lambda,b,b'\}} \wedge (\textbf{bad} \Longrightarrow \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L}))\langle 1 \rangle.$$

En utilisant la tactique `ep`, nous fusionnons les deux branches de la condition dans H_5, nous retrouvons l'oracle initial. Nous pouvons maintenant déplacer le tirage de $\boldsymbol{h}^+$ dans G_5 juste au dessus de l'instruction où v est calculée en utilisant la tactique `swap` et nous remplaçons

$$\boldsymbol{h}^+ \xleftarrow{\$} \{0,1\}^\ell;\ v \leftarrow \boldsymbol{h}^+ \oplus m_b \qquad \text{par} \qquad v \xleftarrow{\$} \{0,1\}^\ell;\ \boldsymbol{h}^+ \leftarrow v \oplus m_b$$

en utilisant l'équivalence (2.5) présentée dans la Section 2.3.1. Alors,

$$\Pr\left[\mathsf{G}_5 : b = b'\right] = \Pr\left[\mathsf{G}_6 : b = b'\right] \tag{2.12}$$

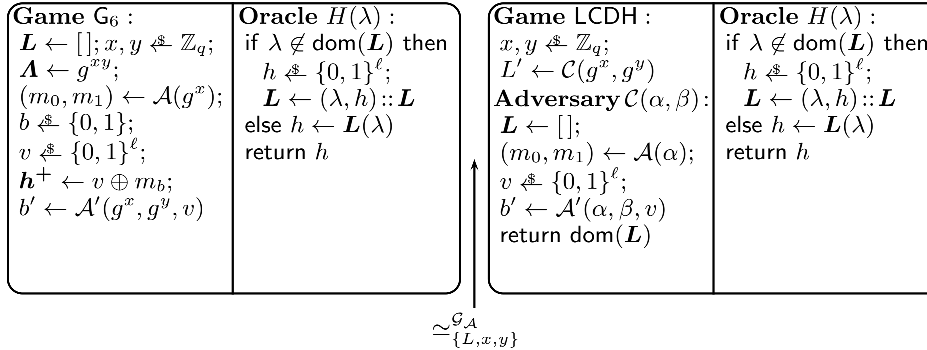
et par ($\leq_{[\![]\!]}$),

$$\Pr\left[\mathsf{G}_5 : \textbf{bad}\right] \leq \Pr\left[\mathsf{G}_6 : \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L})\right]. \tag{2.13}$$

Dans le jeu G_6, b' ne dépend plus de b, le tirage à b peut être placé à la fin du jeu, et nous obtenons

$$\Pr\left[\mathsf{G}_6 : b = b'\right] = \frac{1}{2}. \tag{2.14}$$

Pou finir, nous construisons l'adversaire $\mathcal{C}$ contre LCDH qui utilise les adversaires $(\mathcal{A}, \mathcal{A}')$ comme sous procédures jouant le rôle de challengeur IND-CPA.



$\mathcal{C}$ renvoit la liste des requêtes que l'adversaire $(\mathcal{A}, \mathcal{A}')$ a fait à l'oracle. $\mathcal{C}$ n'a pas besoin de x ou y car il prend en paramètre g^x et g^y. La probabilité de succès de $\mathcal{C}$ est la même que la probabilité que $\boldsymbol{\Lambda} = g^{xy}$ soit dans le domaine $\boldsymbol{L}$ dans G_6. Au final nous obtenons

$$\Pr[\mathsf{G}_6 : \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L})] = \Pr[\mathsf{G}_6 : g^{xy} \in \mathsf{dom}(\boldsymbol{L})] = \Pr[\mathsf{LCDH} : g^{xy} \in L']. \quad (2.15)$$

Pour résumer, à partir des équations (2.7)—(2.15) nous obtenons

$$\begin{aligned} |\Pr[\mathsf{IND\text{-}CPA} : b = b'] - \tfrac{1}{2}| &= |\Pr[\mathsf{G}_4 : b = b'] - \tfrac{1}{2}| \\ &= |\Pr[\mathsf{G}_4 : b = b'] - \Pr[\mathsf{G}_6 : b = b']| \\ &= |\Pr[\mathsf{G}_4 : b = b'] - \Pr[\mathsf{G}_5 : b = b']| \\ &\leq \Pr[\mathsf{G}_5 : \textbf{bad}] \\ &\leq \Pr[\mathsf{G}_6 : \boldsymbol{\Lambda} \in \mathsf{dom}(\boldsymbol{L})] \\ &= \Pr[\mathsf{LCDH} : g^{xy} \in L'] \\ &= \epsilon_{\mathsf{LCDH}}(\eta) \end{aligned}$$

À partir des équations ci dessus, et sous l'hypothèse LCDH (2.4), l'avantage IND-CPA de l'adversaire $(\mathcal{A}, \mathcal{A}')$ est négligeable par rapport à $\frac{1}{2}$. Comme pour la preuve dans le standard modèle, il faut vérifier que l'adversaire $\mathcal{C}$ s'exécute en temps polynomial. C'est le cas, étant donné que l'adversaire $(\mathcal{A}, \mathcal{A}')$ s'exécute en temps polynomial, et $\mathcal{C}$ n'effectue aucun calcul coûteux.

La thèse de Santiago Zanella [88], propose une version alternative de la preuve en faisant directement appel à CDH sans passer par LCDH. Les deux preuves sont equivalentes.

2.4 Conclusion

CertiCrypt est un librairie complètement formalisée en Coq, qui permet de faire des preuves de sécurité par jeux. Les preuves dans CertiCrypt reposent sur une base minimale, et vérifier une preuve peut être facilement effectué par un tiers.

Dans ce chapitre, nous avons illustré certain aspect de CertiCrypt, en expliquant les formalisations des preuves de sécurité sémantique du chiffrement à clé publique Hashed ElGamal, dans le modèle standard et le *random oracle model*, nous montrons les différences entre notre preuve et celle apparaissant dans la littérature. Pour obtenir plus d'information sur CertiCrypt nous invitons le lecteur à lire [21] ou [88].

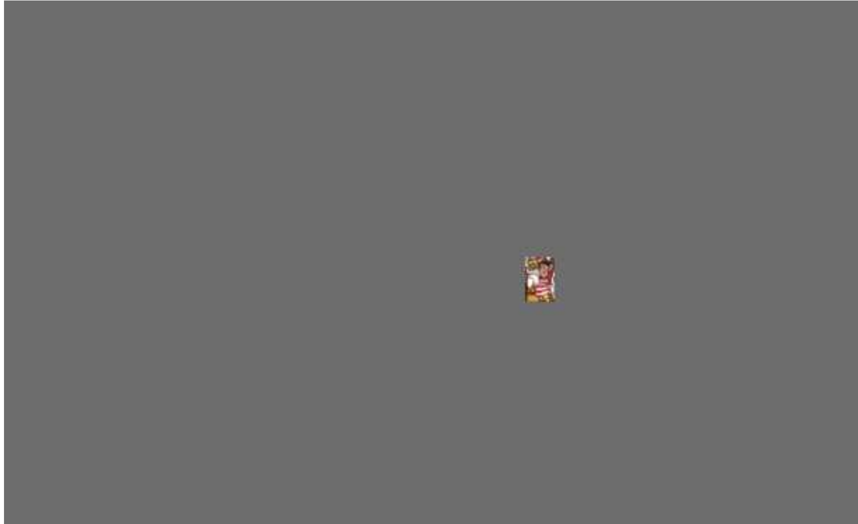
Ces deux preuves sont assez simples, mais ont servi à prendre en main l'outil CertiCrypt. La preuve dans le modèle standard a pris moins deux semaines, en s'aidant de la preuve existante de ElGamal. La seconde preuve a pris un mois, en s'aidant de la preuve de FDH.

3
Protocoles Zero-Knowledge

Tout le monde connaît le jeu “ Où est Charlie ?” ou “Waldo”, qui consiste à retrouver Charlie, toujours habillé de la même manière, dans une image, comme celle ci :

Le docteur K. veut montrer à ses disciples qu’il possède un grand pouvoir magique. Il prétend être capable de trouver Charlie dans n’importe quelle image, de n’importe quelle taille. Un matin, un de ses disciples amène une grande image, tellement grande que personne n’arrive à trouver Charlie. Il est demandé au docteur K. de montrer où est Charlie. Ce qui permettrait de vérifier que le docteur a bien des pouvoirs magiques. Le docteur leur répond “Si j’utilise mon pouvoir et que je vous montre où est Charlie, n’importe qui d’entre vous pourra prétendre posséder mon pouvoir, en montrant à son tour où est Charlie sur cette image à d’autres. Je veux vous prouver que je possède le pouvoir, sans vous dire où est Charlie”.

Le docteur K demanda à ses disciples de sortir tout le monde de la salle et il prit une grande toile grise, bien plus grande que l'image, et très épaisse, de telle sorte que personne ne puisse rien voir à travers. Il fit un trou dans la toile, de la taille de Charlie. Et mit la toile devant l'image en faisant en sorte que le trou montre Charlie. La toile étant bien plus grande, le docteur K fit en sorte que personne ne puisse deviner comment l'image était placée tout garentissant que c'est bien la même image.



Ainsi les disciples du docteur K. découvrirent que leur maître avait un grand pouvoir, et que personne ne pourrait faire croire que le maître le leur avait transmis.

3.1 Introduction

Ces discussions entre deux parties s'appellent des preuves de connaissances[51, 48], le docteur K s'appelle le prouveur et le disciple est le vérifieur. Le prouveur convainc le vérifieur qu'il connaît *quelque chose*. Les deux parties partagent une information commune x et le *quelque chose* correspond à un témoin w qui montre que l'entrée x est dans un langage $\mathcal{NP}$. Un langage est dit $\mathcal{NP}$ si il existe un programme deterministe $W(x, w)$ polynomial en la taille de x. tel que $x \in L$ si il existe un w tel que la relation $W(x, w)$ est acceptée.

Ces preuves de connaissance servent à rendre une partie honnête, même si elle est peut être malhonnête [10] : le témoin représente un moyen de montrer que le *prouveur* est valable pour le *vérifieur*, il montre que les messages envoyés par le *prouveur* respectent bien les règles du protocole. Dans notre cas l'entrée publique commune est l'image de “Où est Charlie ?”, le témoin est la position de Charlie dans l'image.

Les preuves de connaissance doivent être : *complètes* (un *prouveur* qui connaît le témoin peut toujours convaincre un *vérifieur* honnête) et *cohérentes* ou *significatives* (un *prouveur* malhonnête a peu de chance d'être convaincant).

Les logiciels ont souvent besoin de garder un secret ou de préserver l'anonymat, les discussions entre parties ne doivent pas dévoiler d'information sur le témoin. Les preuves *Zero-knowledge* (à divulgation nulle de connaissance), sont des preuves de connaissance qui atteignent cet objectif. Elles sont convaincantes et le *vérifieur* n'apprend rien du témoin dans sa discussion avec le *prouveur*, mais sait que le *prouveur* connaît un témoin pour l'entrée considérée.

Cette propriété est formulée de la manière suivante : un protocole est *Zero-knowledge* quand l'ensemble des messages échangés entre un *prouveur* P et un *vérifieur* V (qui est peut-être malhonnête) peut être simulé sans même interagir avec le *prouveur*, en respectant la stratégie de V.

Cela rend la preuve non transférable, la conversation ne peut pas être réutilisée pour convaincre une autre partie. Si une troisième partie analyse les échanges entre P et V, elle n'apprendra rien, car elle ne peut pas faire la différence entre une vraie conversation et une conversation avec un simulateur qui ne connaît pas le témoin.

Les Σ protocoles ont été introduit par Ronald Cramer dans sa thèse [38]. C'est une classe de protocoles à trois étapes, qui sont utiles pour construire des outils cryptographiques efficaces et sécurisés. Cramer décrit ces protocoles de manière abstraite et montre qu'ils peuvent être utilisés en les appliquant avec des hypothèses cryptographiques.

Cramer donne également une méthode efficace pour combiner ces Σ protocoles, qui permet d'obtenir des preuves *Zero-Knowledge* à partir d'un ensemble de Σ protocoles composés à l'aide de formule booléenne ET et OU. Les applications concrètes des Σ protocoles incluent les protocoles d'identification, vote électronique [55], authentification biométrique [62], e-cash [31], calculs multi-parties [66].

Ce chapitre décrit la formalisation de Σ protocole dans CertiCrypt. Ces travaux représentent 20000 lignes de code Coq. Nous présenterons les définitions, les relations entre les différentes notions de sécurité, les diverses constructions qui permettent de créer des instances de protocoles connus, comme Schnorr, Guillou-Quisquater, Okamoto et Feige-Fiat-Shamir. L'idée est de faire des preuves de sécurité pour un Σ^ϕ protocole générique, qui prouve la connaissance d'un antécédent pour un homomorphisme de groupe. Nous nous servirons des modules de Coq pour définir les Σ et Σ^ϕ protocoles. Notre formalisation de Σ^ϕ protocole est un foncteur paramétré par un module prenant les définitions des deux groupes et d'un homomorphisme entre ces deux groupes. Pour montrer que le Σ^ϕ protocole peut être construit comme un Σ protocole, ce module contient un ensemble de propriétés (dites spéciales) sur cet homomorphisme. Pour prouver qu'une instance de Σ^ϕ protocole et bien un Σ protocole, il suffit de créer un nouveau module dans Coq, contenant la définition de l'homomorphisme et les preuves que cet homomorphisme est spécial. Nous montrerons également que le produit de deux homomorphismes spéciaux est un homomorphisme spécial. Cette composition est une sorte de composition ET restreinte au Σ^ϕ protocole.

3.2 Σ Protocoles

Les Σ protocoles sont des conversations à trois étapes, où un prouveur P interagit avec un vérifieur V. Les deux interlocuteurs ont accès à une entrée commune x, et le but du prouveur est de convaincre le vérifieur qu'il connaît une valeur w associée à x, sans rien révéler du lien entre x et w. Le protocole se déroule ainsi, le prouveur commence par envoyer un *commitment* au vérifieur, en réponse le prouveur envoie un challenge c choisit aléatoirement et uniformément dans un ensemble C. Le prouveur renvoie une réponse s calculée à partir du challenge c, et le vérifieur finit par accepter ou rejeter la conversation.

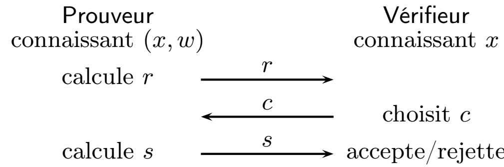


Fig. 3.1. Description des interactions d'un Σ protocole

Formellement, un Σ protocole est défini par rapport à une relation R. Une preuve de connaissance revient alors à prouver l'appartenance à un langage $\mathcal{NP}$ L. Un langage est dit $\mathcal{NP}$, si il existe un algorithme déterministe polynomial qui vérifie si un mot du langage lui appartient selon une relation R_L :

$$L = \{x \mid \exists w, (x, w) \in R_L\}$$

Prouver que x appartient au langage revient à prouver qu'il existe un témoin w relié à x via R_L. Dans CertiCrypt, la classe des Σ protocoles est paramétrée par une relation (appelée "relation de connaissance"), et par les procédures qui représentent les différentes phases du prouveur et du vérifieur ; la signature du module spécifie toutes les bonnes propriétés, permettant de vérifier qu'une instance est bien un Σ protocole. Dans le reste de cette section, nous montrerons en détails notre formalisation des Σ protocoles et d'une version alternative équivalente de propriété de "non divulgation de connaissance".

Définition 3.1 (Σ Protocole). *Un Σ protocole pour une relation de connaissance R est un protocole à trois étapes entre un prouveur P et un vérifieur V, dont l'interaction est décrite par le programme suivant :*

Protocole(x, w) :
$(r, state) \leftarrow \mathsf{P}_1(x, w)$;
$c \leftarrow \mathsf{V}_1(x, r)$;
$s \leftarrow \mathsf{P}_2(x, w, state, c)$;
$b \leftarrow \mathsf{V}_2(x, r, c, s)$

Dans le programme ci dessus, les deux phases du prouveur P sont décrites par les procédures P_1 est P_2, tandis que celles du vérifieur V sont décrites par les

procédures V_1 et V_2. On note que le protocole passe explicitement les états entre les différentes phases des participants ; des variables globales auraient pu être utilisées pour faire communiquer P_1 et P_2 d'un côté et de l'autre entre V_1 et V_2, mais cette représentation aurait compliqué les preuves car il aurait fallu spécifier le fait que les procédures d'une partie n'aient pas accès aux variables globales de l'autre. Tous les protocoles que nous considérons dans la suite sont *public coin*, ce qui veut dire que le vérifieur (honnête) choisit un élément uniformément dans un certain ensemble.

Un Σ protocole doit satisfaire les trois propriétés suivantes :

1. **Complétude** : Étant donné une entrée publique x et un témoin w tel que $(x, w) \in R$, le prouveur est toujours capable de convaincre le vérifieur, i,e, quand le protocole est exécuté dans une mémoire μ où $R(\mu(x), \mu(w))$, la valeur finale de la variable b est toujours true :

$$\forall \mu, R(\mu(x), \mu(w)) \implies \Pr_\mu[\mathsf{Protocole} : b = \mathsf{true}] = 1$$

2. sHVZK : **Special Honest Verifier Zero-Knowledge** : Il existe un algorithme probabiliste et polynomial appelé simulateur S prenant un paramètre x qui est dans $\mathsf{dom}(R)$ et un challenge c, et qui calcule le triplet (r, c, s) avec la même distribution qu'une conversation valide. Cette propriété représente une version du protocole où le challenge c est fixé.

$$\boxed{\begin{array}{l}\textbf{Protocole}'(x, w, c) : \\ (r, state) \leftarrow \mathsf{P}_1(x, w); \\ s \leftarrow \mathsf{P}_2(x, w, state, c); \\ b \leftarrow \mathsf{V}_2(x, r, c, s)\end{array}} \sim^{\{x,c\} \wedge R(x,w)}_{\{r,c,s\}} \quad (r, s) \leftarrow \mathsf{S}(x, c)$$

Cette propriété montre qu'on ne peut obtenir aucune information en regardant une communication, cette communication pouvant être simulée.

3. **Correction Spéciale (Special Soundness)** : Étant donné, deux conversations acceptantes (r, c_1, s_1), (r, c_2, s_2) pour une entrée x, avec des challenges différents $c_1 \neq c_2$ mais avec la même *commitment* r, il existe une fonction PPT d'extraction de connaissance, appelée KE qui calcule un témoin w tel que $(x, w) \in R$. Formellement, pour toutes mémoires μ,

$$\left.\begin{array}{l}\mu(c_1) \neq \mu(c_2) \\ \Pr_\mu[b \leftarrow \mathsf{V}_2(x, r, c_1, s_1) \ : \ b = \mathsf{true}] = 1 \\ \Pr_\mu[b \leftarrow \mathsf{V}_2(x, r, c_2, s_2) \ : \ b = \mathsf{true}] = 1\end{array}\right\} \implies$$
$$\Pr_\mu[w \leftarrow \mathsf{KE}(x, r, c_1, c_2, s_1, s_2) \ : \ (x, w) \in R] = 1$$

Cette propriété vérifie que si une personne arrive à répondre correctement à deux questions pour un même *commitment*, c'est qu'elle possède la clé. Cette preuve ne s'intéresse pas à la manière dont les deux communications ont été générées. Un prouveur qui n'a pas la clé secrète, ne pourra pas répondre à plus d'une question.

Formalisation

En Coq, les Σ protocoles sont formalisés par un module dont la signature regroupe l'ensemble des propriétés suivantes :

Module Type	Sigma Protocole
xT wT rT stateT cT sT :	Type
R :	$[\![\mathsf{xT}]\!] \to [\![\mathsf{wT}]\!] \to \mathsf{Prop}$
R_dec :	$\forall (x : [\![\mathsf{xT}]\!])(w : [\![\mathsf{wT}]\!]), \{\mathsf{R}\ x\ w\} + \{\neg \mathsf{R}\ x\ w\}$
P_1 :	Proc (xT :: wT) (rT, stateT)
P_2 :	Proc (xT :: wT :: stateT :: cT) sT
V_1 :	Proc (xT :: rT) cT
V_2 :	Proc (xT :: rT :: cT :: sT)bool
S :	Proc (xT :: cT) (rT, sT)
KE :	Proc (xT :: rT :: cT :: cT :: sT :: sT) wT
completeness :	$\forall m\ x\ w, \mathsf{R}\ x\ w \to \mathsf{Pr}_{\mathsf{Protocole}(x,w)}[b] = 1$
SHVZK :	$\mathsf{Protocole}'(x,w,c) \sim^{\{x,c\} \wedge \mathsf{R}(x,w)}_{\{r,c,s\}} (r,s) \leftarrow S(x,c)$
KE_correct :	$c_1 \neq c_2 \to$
	$\mathsf{Pr}_{b \leftarrow V_2(x,r,c_1,s_1)}[b] = 1 \to$
	$\mathsf{Pr}_{b \leftarrow V_2(x,r,c_2,s_2)}[b] = 1 \to$
	$\mathsf{Pr}_{w \leftarrow \mathsf{KE}(x,r,c_1,c_2,s_1,s_2)}[x = w] = 1$

Proc est le type des procédures de CertiCrypt prenant en argument une liste de type CertiCrypt représentant les types des arguments et le type de retour.

La procédure Protocole fait appel aux procédures P_1, P_2, V_1 et V_2, ces procédures sont ajoutées lors de la construction de l'environnement. De même que pour KE.

Les types xT,..., sont des types de CertiCrypt, lors de la définition de R, nous interprétons les types Coq, avec la fonction $[\![.]\!]$.

3.2.1 Relation entre sHVZK et HVZK

Certains auteurs ont besoin que les Σ protocoles satisfassent une propriété plus faible que la version *spéciale* expliquée ci dessus. Pour sHVZK le challenge est fixé et est donné en paramètre, alors que pour HVZK, le simulateur peut choisir le challenge. Le simulateur d'un protocole satisfaisant HVZK reçoit juste un $x \in \mathsf{dom}(R)$ et calcule un triplet (r, c, s) avec la même distribution qu'une conversation normale avec un vérifieur. La relation entre les deux notions a été étudiée par Cramer [38]. Nous prouverons formellement que les deux notions sont équivalentes.

Théorème 3.2 (sHVZK implique HVZK). *Un Σ protocole satisfaisant sHVZK, satisfait aussi HVZK.*

Preuve. Un simulateur HVZK S′ peut être construit à partir d'un simulateur sHVZK S :

```
Simulateur S'(x) :
  c <-$ {0,1}^k;
  (r,s) <- S(x,c);
  return (r,c,s)
```

Nous prouvons que S' simule parfaitement les conversations du protocole, en utilisant la séquence de jeux suivante :

$$\begin{aligned}\mathsf{Protocole}(x,w) &\sim^{\{x\}\wedge R(x,w)}_{\{r,c,s\}} c \stackrel{\$}{\leftarrow} \{0,1\}^k; \mathsf{Protocole}'(x,w,c)\\ &\sim^{\{x\}\wedge R(x,w)}_{\{r,c,s\}} c \stackrel{\$}{\leftarrow} \{0,1\}^k; (r,s) \leftarrow \mathsf{S}(x,c)\\ &\sim^{\{x\}\wedge R(x,w)}_{\{r,c,s\}} (r,c,s) \leftarrow \mathsf{S}'(x)\end{aligned}$$

La première et la dernière équivalence sont facilement prouvables après avoir déplié les appels de procédure avec la tactique `inline`, et réordonné les instructions avec la tactique `swap`. Pour prouver la seconde équivalence, la tactique `eqobs_hd` est utilisée pour enlever l'instruction $c \stackrel{\$}{\leftarrow} \{0,1\}^k$ qui est commune dans les deux jeux ; le but obtenu est exactement la définition de sHVZK pour S, soit :

$$\mathsf{Protocole}'(x,w,c) \sim^{\{x\}\wedge R(x,w)}_{\{r,c,s\}} (r,s) \leftarrow \mathsf{S}(x,c)$$

La propriété sHVZK est plus forte que HVZK, car un protocole satisfaisant sHVZK satisfait HVZK. Et pour tout protocole (P,V) satisfaisant HVZK, il est possible de construire un protocole $(\mathsf{P}',\mathsf{V}')$ qui satisfait sHVZK :

$$\begin{aligned}\mathsf{P}'_1(x,w) &\stackrel{\text{def}}{=} (r, state) \leftarrow \mathsf{P}_1(x,w); c' \stackrel{\$}{\leftarrow} \{0,1\}^k;\\ &\quad \mathsf{return}\ ((r,c'),(state,c')))\\ \mathsf{P}'_2(x,w,(state,c'),c) &\stackrel{\text{def}}{=} s \leftarrow \mathsf{P}_2(x,w,state,c\oplus c'); \mathsf{return}\ s\\ \mathsf{V}'_1(x,(r,c')) &\stackrel{\text{def}}{=} c \leftarrow \mathsf{V}_1(x,r);\ \mathsf{return}\ (c\oplus c')\\ \mathsf{V}'_2(x,(r,c'),c,s) &\stackrel{\text{def}}{=} b \leftarrow \mathsf{V}_2(x,r,c\oplus c',s);\ \mathsf{return}\ b\end{aligned}$$

Cette construction crée un nouveau protocole où HVZK et sHVZK coïncident. La différence est que dans le nouveau protocole, on applique la fonction XOR au challenge que le vérifieur choisit et à un *bitstring* tiré aléatoirement par le prouveur au début du protocole.

Théorème 3.3 (sHVZK à partir de HVZK). *Si un protocole (P,V) est un Σ protocole comme dans la Définition 3.1 mais satisfait HVZK au lieu de sHVZK, alors le protocole $(\mathsf{P}',\mathsf{V}')$ définit au dessus est un Σ protocole .*

Preuve. Complétude

Cette propriété se montre facilement à partir de la propriété de Complétude du protocole (P,V) et de la propriété du OU exclusif suivante :

$$(c\oplus c')\oplus c' = c$$

sHVZK

Le programme suivant est un simulateur sHVZK pour le protocole

$$\mathsf{S}'(x,c) \stackrel{\text{def}}{=} (\hat{r},\hat{c},\hat{s}) \leftarrow \mathsf{S}(x);\ \mathsf{return}\ ((\hat{r}, c\oplus\hat{c}),\hat{s})$$

(Les variables du protocole original ont un accent circonflexe). Nous prouvons ceci par la séquence d'équivalences suivante :

$$\begin{aligned}
\mathsf{Protocole}'(x,w,c) &\sim^{\{x,c\}\wedge R(x,w)}_{\{r,c,s\}} \mathsf{Protocole}(x,w);\\
&\qquad r \leftarrow (\hat{r}, c \oplus \hat{c}); s \leftarrow \hat{s}\\
&\sim^{\{x,c\}\wedge R(x,w)}_{\{r,c,s\}} (\hat{r},\hat{c},s) \leftarrow \mathsf{S}(x);\\
&\qquad r \leftarrow (\hat{r}, c \oplus \hat{c})\\
&\sim^{\{x,c\}\wedge R(x,w)}_{\{r,c,s\}} (r,s) \leftarrow \mathsf{S}'(x,c)
\end{aligned}$$

La première et la dernière équivalences sont prouvées sans difficulté en utilisant les tactiques, décrites dans le Chapitre 2. La deuxième équivalence peut être réduite à la propriété HVZK de S en utilisant la tactique `alloc` et `eqobs_tl` pour simplifier le but.

Correction

À partir d'une conversation $((r,c'),(c \oplus c'),s)$ de $(\mathsf{P}',\mathsf{V}')$ une conversation (r,c,s) du protocole original peut être facilement retrouvée. L'extracteur de connaissance suivant prouve la propriété de Correction spéciale de $(\mathsf{P}',\mathsf{V}')$:

$$\begin{aligned}
&\mathsf{KE}'(x,(r,c'),c_1,c_2,s_1,s_2):\\
&\quad w \leftarrow \mathsf{KE}(x,r,c' \oplus c_1, c' \oplus c_2, s_1, s_2);\ \mathsf{return}\ w
\end{aligned}$$

3.3 Σ Protocoles pour homomorphismes spéciaux

Les Σ^ϕ protocoles, sont des preuves de connaissance d'un antécédent pour homomorphisme de groupe. Le protocole de Schnorr [76], est l'exemple type de preuve de non divulgation de connaissance. Schnorr est une instance de Σ^ϕ protocole qui prouve la connaissance d'un logarithme discret pour un groupe cyclique. Dans ce cas l'homomorphisme est la fonction exponentiation $\phi(x) = g^x$, où g est un générateur du groupe.

Notre formalisation des Σ^ϕ protocoles est constructive. Nous fournissons un foncteur qui, à partir d'un homomorphisme ϕ accompagné d'un ensemble de propriétés sur ϕ, construit un Σ protocole concret, qui montre que l'ont connaît un antécédent pour ϕ. Ce protocole est un Σ protocole car il contient les preuves de complétude, correction et de sHVZK. Au final, tout ce qu'il faut pour construire une instance de Σ^ϕ protocole est de spécifier un homomorphisme et de prouver les bonnes propriétés. Nous nous servons des signatures des modules de Coq pour décrire ces modules. Le vérifieur de types de Coq permet de vérifier qu'aucune des propriétés requises n'est oubliée. Nous donnons quelques exemples de Σ^ϕ protocole comprenant les protocoles de Schnorr, Guillou-Quisquater et Feige-Fiat-Shamir. Notre construction de Σ^ϕ protocole nous évite de prouver à chaque fois les propriétés de Def. 3.1. Ces instances ne sont pas pour autant faciles, car nous devons

à chaque fois, formaliser les homomorphismes, les deux groupes où s'applique l'homomorphisme et prouver les propriétés.

Notons $(\mathcal{G}, \oplus)$ et $(\mathcal{H}, \otimes)$ nos deux groupes, où $\oplus$ (resp. $\otimes$) est la loi de composition interne de $\mathcal{G}$ (resp. $\mathcal{H}$)

Définition 3.4 (Σ^ϕ Protocole). *Prenons l'homomorphisme $\phi : \mathcal{G} \to \mathcal{H}$ et une relation $R \stackrel{def}{=} \{(x, w) \mid x = \phi(w)\}$. Le Σ^ϕ protocole pour une relation R avec un ensemble de challenges C est un protocole Σ (P, V) définit comme ceci :*

$$\begin{array}{ll} \mathsf{P}_1(x, w) & \stackrel{def}{=} y \stackrel{\$}{\leftarrow} \mathcal{G};\ \textit{return}\ (\phi(y), y) \\ \mathsf{P}_2(x, w, y, c) & \stackrel{def}{=} \textit{return}\ (y \oplus cw) \\ \mathsf{V}_1(x, r) & \stackrel{def}{=} c \stackrel{\$}{\leftarrow} C;\ \textit{return}\ c \\ \mathsf{V}_2(x, r, c, s) & \stackrel{def}{=} \textit{return}\ (\phi(s) = r \otimes x^c) \end{array}$$

Pour le groupe G nous utilisons la notation additive avec l'élement neutre 0 et pour H la notation multiplication avec l'élement neutre 1.

Nous pouvons montrer que le protocole précédemment défini satisfait les propriétés d'un Σ protocole où $C = \{0, 1\}$. Cependant un prouveur qui triche, peut convaincre un vérifieur avec un probabilité de $1/2$; cette probabilité peut être réduite à $1/2^n$ en répétant le protocole n fois. Nous verrons que certaines classes d'homomorphisme, admettent un ensemble de challenges plus grand, et permettent d'obtenir une erreur de correction plus petite pour une seule exécution du protocole.

Définition 3.5 (Homomorphisme Spécial). *Un homomorphisme $\phi : \mathcal{G} \to \mathcal{H}$ est spécial si il existe une valeur $v \in \mathbb{Z} \setminus \{0\}$ (appelée* exposant spécial*) et un algorithme PPT qui étant donné $x \in \mathsf{Im}(\phi)$ calcule $u \in \mathcal{G}$ tel que $\phi(u) = x^v$.*

Pour formaliser les Σ^ϕ protocoles , nous étendons les types de CertiCrypt avec les groupes $\mathcal{G}, \mathcal{H}$ et les opérateurs pour calculer les opérations de groupe, exponentiation, produit et inverse ; nous ajoutons aussi l'opérateur $\phi(\cdot)$, $u(\cdot)$, et l'expression constante v qui représente l'exposant spécial de l'homomorphisme (Def. 3.5).

Un Σ^ϕ protocole construit à partir d'un homomorphisme spécial, est basé sur un ensemble de challenge de la forme $[0..c^+]$, où c^+ est plus petit que le plus petit nombre premier qui divise l'exposant spécial v. Nous verrons plus loin comment cette hypothèse est utilisée dans la preuve.

Prenons p, le plus petit premier qui divise v (nous supposons que $|v| \geq 2$), alors le plus grand c^+ qui peut être choisi est $p - 1$. Nous ajoutons à notre construction un foncteur qui construit le plus grand ensemble de challenges pour chaque homomorphisme spécial ϕ ; et permet alors de minimiser l'erreur de correction.

Formalisation

Nous expliquons les différents signature des modules pour les groupes, homomorphismes et *challenges set*.

Module Type	Group
t :	Type
elems :	list t
elems_full :	$\forall (x : t),\ x \in$ elems
elems_nodup :	NoDup elems
eqb :	$t \to t \to$ bool
eqb_spec :	$\forall (x\ y : t)$, if eqb $x\ y$ then $x = y$ else $x \neq y$
e :	t
mul :	$t \to t \to t$
inv :	$t \to t \to t$
mul_assoc :	$\forall (x\ y\ z : t)$, mul x (mul $y\ z$) $=$ mul (mul $x\ y$) z
mul_e_l :	$\forall (x : t)$, mul $e\ x = x$
mul_inv_l :	$\forall (x : t)$, mul (inv x) $x = e$
mul_comm :	$\forall (x\ y : t)$, mul $x\ y =$ mul $y\ x$

t représente le type des éléments du groupe en Coq, la liste elems représente les éléments du groupe. Cette liste ne contient pas de doublons. Tous les éléments de la liste doivent être dans cette liste. La fonction eqb représente l'égalité de deux éléments, et cette fonction assure que la relation $x = y$ est décidable. e représente l'élément neutre. La loi de groupe et l'inverse doivent être définies, avec leur preuve de correction.

Module Type	Homomorphisme($G\ H$: Groupe)
ϕ :	$t_G \to t_H$
ϕ_homo :	$\forall (x\ y : t_G), \phi(\text{mul}_G\ x\ y) = \text{mul}_H\ \phi(x)\ \phi(y)$
special$_v$:	$\mathbb{Z}$
special$_v$_spec :	special$_v \neq 0$
special$_u$:	$t_H \to t_G$
ϕ_special :	$\forall (x : t_H), x \in \text{Im}(\phi) \to \phi(\text{special}_u\ x) = x^{\text{special}_v}$

ϕ est la définition de l'homomorphisme entre les groupes G et H. avec la preuve que c'est bien un homomorphisme. Le module contient également les définitions de l'exposant spécial v, de la fonction u et la preuve qui les relie.

Module Type	Challenge Set($G\ H$: Groupe)(HM : Homomorphisme $G\ H$)
c^+ :	$\mathbb{N}$
c^+_spec :	$\forall p$, prime $p \to (p \mid HM.\text{special}_v) \to c^+ < p$

Le module Challenge Set définit les propriétés sur la borne c^+. Nous créons un module de type Challenge Set qui cherche le plus petit premier qui divise v et nous prenons son prédécesseur.

Théorème 3.6 (Σ^ϕ protocole pour les homomorphismes spéciaux). *Si un homomorphisme ϕ est spécial et c^+ est plus petit que tous les diviseurs premiers de l'exposant spécial v, alors le protocole de la Définition 3.4 est un Σ protocole avec l'ensemble de challenges $C = [0..c^+]$.*

Preuve.

Complétude

Nous montrons qu'un prouveur honnête arrive toujours à convaincre le vérifieur, i.e.

$$\forall \mu, R(\mu(x), \mu(w)) \implies \Pr_\mu[\mathsf{Protocole} : b = \mathsf{true}] = 1$$

Nous pouvons reformuler ce but, en terme d'équivalence de programme, comme ceci :

$$\mathsf{Protocole}(x, w) \sim^{R(x,w)}_{\{b\}} b \leftarrow \mathsf{true}$$

Nous utilisons le script suivant pour prouver cette équivalence. On commence par déplier tous les appels de procédure et simplifier le résultat en appliquant les optimisations comme propagation de constante et élimination de code mort.

```
inline P1; inline P2; inline V1; inline V2;
ep; deadcode.
```

Le but obtenu est le suivant :

$$\begin{array}{l} y \xleftarrow{\$} \mathcal{G};\ c \xleftarrow{\$} [0..c^+]; \\ b \leftarrow \phi(y) \otimes \phi(w)^c = \phi(y) \otimes x^c \end{array} \sim^{\phi(w)=x}_{\{b\}} b \leftarrow \mathsf{true}$$

Nous utilisons la tactique `ep_eq` x $\phi(w)$ pour remplacer la dernière instruction du jeu de gauche par $b \leftarrow \mathsf{true}$, ceci rajoute un but où nous devrons prouver que $\phi(y) \otimes \phi(w)^c = \phi(y) \otimes x^c = \mathsf{true}$, qui est vraie sous l'hypothèse $\phi(w) = x$. Ensuite la tactique `deadcode` supprime les deux premières instructions $y \xleftarrow{\$} \mathcal{G};\ c \xleftarrow{\$} [0..c^+]$ qui ne servent plus dans le calcul de b, Nous terminons la preuve par l'application de la tactique `eqobs_in`.

sHVZK

Le programme S est un simulateur sHVZK pour le protocole :

$$\boxed{\begin{array}{l} \textbf{Protocole}(x, w, c) : \\ (r, state) \leftarrow \mathsf{P}_1(x, w); \\ s \leftarrow \mathsf{P}_2(x, w, state, c); \\ b \leftarrow \mathsf{V}_2(x, r, c, s) \end{array}} \sim^{\{x,w,c\} \wedge R(x,w)}_{\{r,c,s\}} \boxed{\begin{array}{l} \textbf{Simulateur } \mathsf{S}(x, c) : \\ s \xleftarrow{\$} \mathcal{G}; \\ r \leftarrow \phi(s) \otimes x^{-c}; \\ \mathsf{return}\ (r, s) \end{array}}$$

Nous allons montrer les étapes de la preuve que S simule parfaitement les conversations du protocole.

1. Nous commençons la preuve en dépliant les appels de procédure P_1 et P_2 avec le script `inline_l` P_1; `inline_l` P_2. Nous simplifions le but en appelant `ep`; `deadcode` l'appel à V_2 disparaît car les variables $\{r, c, s\}$ ne dépendent pas de b.
2. Nous introduisons le jeu B en utilisant la propriété de transitivité de l'équivalence observationelle. L'équivalence avec le jeu précédent se prouve en réordonnant les instructions en utilisant la tactique `swap`. Les deux jeux ont le même suffixe ($r \leftarrow \phi(y); s \leftarrow y \oplus cw$), que nous enlevons en appelant la tactique `eqobs_tl`, pour obtenir le but suivant :

$$y \xleftarrow{\$} \mathcal{G} \sim^{\{x,w,c\} \wedge R(x,w)}_{\{y,w,c\}} s' \xleftarrow{\$} \mathcal{G};\ y \leftarrow s' \oplus -cw$$

Nous pouvons enlever les variables non modifiées de la post condition (w et c) en utilisant la tactique `clean_nm`. Ensuite la tactique `alloc` $y\ s'$ remplace l'instruction $y \xleftarrow{\$}$ par $s' \xleftarrow{\$} \mathcal{G};\ y \leftarrow s'$. En affaiblissant la pré-condition par `true`, nous obtenons le but suivant :

$$s' \xleftarrow{\$} \mathcal{G};\ y \leftarrow s' \sim_{\{y\}} s' \xleftarrow{\$} \mathcal{G};\ y \leftarrow s' \oplus -cw$$

qui se prouve en utilisant le fait que la fonction `fun` $x \Rightarrow\ x - cw$ agit comme une fonction "one-pad"

3. Nous propageons la constante y à l'aide de la tactique `ep`, et simplifions le but avec `deadcode`. La tactique `ep_eq` est utilisée pour remplacer $(s' \oplus -cw) \oplus cw$ par s', et $\phi(s' \oplus -cw)$ par $\phi(s') \otimes \phi(w)^{-c}$ (propriété d'homomorphisme de ϕ).
4. Nous introduisons le jeu D ; et prouvons l'équivalence en allouant la variable s' dans s ; le jeu résultant est identique au jeu C.
5. La variable s est substituée par s' dans le jeu D, la precondition $R(x, w)$ permet de substituer x par $\phi(w)$. Les jeux D et E sont équivalents une fois les instructions réordonnées.
6. Le simulateur est dépilé.

Protocole(x, w, c) :
$(r, state) \leftarrow \mathsf{P}_1(x, w)$;
$s \leftarrow \mathsf{P}_2(x, w, state, c)$;
$b \leftarrow \mathsf{V}_2(x, r, c, s)$

(1) ↓

A
$y \xleftarrow{\$} \mathcal{G}$;
$r \leftarrow \phi(y)$;
$s \leftarrow y \oplus cw$

(2) $\sim^{\{x,w,c\} \wedge R(x,w)}_{\{r,c,s\}}$

B
$s' \xleftarrow{\$} \mathcal{G}$;
$y \leftarrow s' \oplus -cw$;
$s \leftarrow y \oplus cw$;
$r \leftarrow \phi(y)$

(3) ↓

C
$s' \xleftarrow{\$} \mathcal{G}$;
$s \leftarrow s'$;
$r \leftarrow \phi(s') \otimes \phi(w)^{-c}$

(4) ↓

D
$s \xleftarrow{\$} \mathcal{G}$;
$r \leftarrow \phi(s) \otimes \phi(w)^{-c}$

(5) $\sim^{\{x,w,c\} \wedge R(x,w)}_{\{r,c,s\}}$

E
$s' \xleftarrow{\$} \mathcal{G}$;
$r \leftarrow \phi(s') \otimes x^{-c}$;
$s \leftarrow s'$

(6) ↑

Simulateur(x, c) :
$(r, s) \leftarrow \mathsf{S}(x, c)$

Correction

Pour prouver la correction des Σ^{ϕ} protocoles, il est montré qu'il existe un algorithme PPT, KE prenant deux conversations acceptantes (x, r, c_1, s_1), (x, r, c_2, s_2),

avec $c_1 \neq c_2$ mais avec le même *commitement* r, qui calcule w tel que $x = \phi(w)$. Nous proposons l'extracteur de connaissance suivant,

```
KE(x, c1, c2, s1, s2) :
 (a, b, d) ← pgcd_etendu(c1 − c2, v);
 w ← a(s1 ⊕ −s2) ⊕ b u(x);
 return w
```

où pgcd_etendu implémente l'algorithme d'Euclide étendu. Pour deux entiers a, b, pgcd_etendu(a, b) calcule un triplet d'entiers (x, y, d) tel que d est le Plus Grand Commun Diviseur de a et b, et x, y satisfont l'identité de Bézout

$$ax + by = \mathsf{PGCD}(a, b) = d.$$

Vu que tous les calculs faits par l'extracteur de connaissance peuvent être implémentés efficacement, KE est un algorithme PPT ; nous le prouvons en Coq en appelant la tactique automatique `PPT_proc` qui prouve qu'un programme sans boucle ou appel récursif est PPT en calculant un polynôme qui borne le temps et l'espace, du moment que les expressions sont PPT. Nous vérifions que KE calcule un antécédent de l'entrée publique x. Pour deux conversations acceptantes (x, r, c_1, s_1) et (x, r, c_2, s_2), nous avons

$$\phi(s_1) = r \otimes x^{c_1} \;\wedge\; \phi(s_2) = r \otimes x^{c_2}$$

et donc

$$x^{c_1 - c_2} = \phi(s_1 \oplus -s_2) \tag{3.1}$$

De plus, ϕ est spécial, nous pouvons alors exécuter u qui vérifie $x^v = \phi(u)$. Le triplet (a, b, d) donné par l'algorithme d'Euclide étendu satisfait l'identité de Bézout

$$a(c_1 - c_2) + bv = \mathsf{PGCD}(c_1 - c_2, v) = d \tag{3.2}$$

Les challenges c_1 et c_2 sont plus petits que c^+, qui est plus petit que le plus petit diviseur premier de v. Nous déduisons qu'aucun diviseur de $|c_1 - c_2|$ ne peut diviser v et donc $d = \mathsf{PGCD}(|c_1 - c_2|, v) = 1$. ϕ est un homomorphisme, ajouté à (3.1) et (3.2) nous avons

$$\phi(w) = \phi(a(s_1 \oplus -s_2) \oplus bu) = x^{a(c_1 - c_2)} \otimes x^{bv} = x^d = x$$

3.3.1 Instances concrètes de Σ^ϕ protocoles

Nous avons formalisé plusieurs Σ^ϕ protocoles en utilisant le foncteur décrit dans la section précédente. Pour chaque protocole, nous avons spécifié les groupes $\mathcal{G}, \mathcal{H}$ et l'homomorphisme spécial $\phi : \mathcal{G} \to \mathcal{H}$, ainsi que l'interprétation de l'opérateur $u(\cdot)$ et de l'exposant spécial v. Le tableau 3.1 récapitule tous les protocoles formalisés.

Les protocoles de Schnorr [76] et de Okamoto [72] sont basés sur le problème du logarithme discret. Pour deux nombres premiers p et q tel que q divise $p - 1$,

Protocole	$\mathcal{G} \to \mathcal{H}$	$\phi(x)$	$u(x)$	v
Schnorr	$\mathbb{Z}_q^+ \to \mathbb{Z}_p^*$	g^x	0	q
Okamoto	$(\mathbb{Z}_q^+, \mathbb{Z}_q^+) \to \mathbb{Z}_p^*$	$g_1^{x_1} \otimes g_2^{x_2}$	$(0,0)$	q
Diffie-Hellman	$\mathbb{Z}_q^+ \to \mathbb{Z}_p^* \times \mathbb{Z}_p^*$	(g^x, g^{bx})	0	q
Guillou-Quisquater	$\mathbb{Z}_N^* \to \mathbb{Z}_N^*$	x^e	x	e
Feige-Fiat-Shamir	$\{-1,1\} \times \mathbb{Z}_N^* \to \mathbb{Z}_N^*$	$s.x^2$	$\lvert x.2 \rvert$	2

Tableau 3.1. Homomorphismes spéciaux pour les Σ^ϕ protocoles choisis

un groupe de Schnorr est un sous-groupe multiplicatif de $\mathbb{Z}_p^*$ d'ordre q avec un générateur g. Le Σ protocole pour prouver la connaissance d'un logarithme discret dans un groupe de Schnorr est obtenu en paramétrant le foncteur de Def. 3.4 avec l'homomorphisme $\phi(x) = g^x$. Pour respecter la condition de correction spéciale, nous définissons le groupe $\mathcal{H}$ comment étant un sous groupe de $\mathbb{Z}_p^*$ d'ordre q. Nous définissons alors le groupe H de telle sorte que tout elements x respectent la propriété suivante $x^q \bmod p = 1$. Nous prenons ensuite l'ordre q comme exposant spécial et $u(x) = 0$ pour tout $x \in \mathbb{Z}_p^*$.

Le protocole d'Okamoto est similaire à celui de Schnorr, mais il est basé sur deux sous groupes de Schnorr avec les générateurs g_1 and g_2. Dans ce cas ϕ prend une paire (x_1, x_2) et calcule $g_1^{x_1} \otimes g_2^{x_2}$.

Prenons un "module de chiffrement" (RSA modulus) N avec comme facteur premier p and q, et prenons e l'exposant public ; e doit être premier avec l'indicatrice d'Euler $\varphi = (p-1)(q-1)$ (i.e. $\gcd(e, \varphi(N)) = 1$). Les protocoles de Guillou-Quisquater [53], et Feige-Fiat-Shamir [44] sont basés sur la difficulté de résoudre le problème RSA : étant donné N, e et $y \equiv x^e \mod N$, essayer de calculer x, la $e^{\text{ième}}$-racine de y modulo N. Le protocole de Guillou-Quisquater est obtenu en prenant $\phi : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$, $\phi(x) = x^e$. Le protocole de Feige-Fiat-Shamir est obtenu en prenant $\phi : \{-1,1\} \times \mathbb{Z}_N^* \to \mathbb{Z}_N^*$, $\phi(s,x) = s.x^2$. La notation $|x.2|$ represente la partie droite de la paire de type $\{-1,1\} \times \mathbb{Z}_N^*$.

Remarque. Nos résultats sont vrais indépendamment des hypothèses calculatoires. La difficulté d'inverser l'homomorphisme sur lequel est basé le protocole rendra Σ^ϕ protocole intéressant, mais ce n'est pas essentiel pour montrer les propriétés que nous prouvons sur ce protocole.

3.3.2 Composition de Σ^ϕ protocoles

Prenons deux homomorphismes spéciaux $\phi_1 : \mathcal{G}_1 \to \mathcal{H}_1$ et $\phi_2 : \mathcal{G}_2 \to \mathcal{H}_2$, avec comme exposant spécial v_1, v_2 et les algorithmes associés u_1, u_2, respectivement. Nous expliquerons dans cette partie comment combiner deux Σ^ϕ protocoles définis par ces homomorphismes.

Théorème 3.7 (Produit d'homomorphismes spéciaux). *L'homomorphisme suivant construit à partir du produit de $\mathcal{G}_1$ et $\mathcal{G}_2$ au produit de $\mathcal{H}_1$ et $\mathcal{H}_2$ est un homomorphisme spécial. Cette construction permet de combiner par l'opérateur logique*

ET deux Σ^{ϕ} protocoles .

$$\begin{array}{ll} \phi & : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{H}_1 \times \mathcal{H}_2 \\ \phi(x_1, x_2) & \overset{def}{=} (\phi_1(x_1), \phi_2(x_2)) \end{array}$$

Preuve. Il suffit de prendre

$$\begin{array}{ll} v & \overset{\text{def}}{=} \text{ppcm}(v_1, v_2) \\ u(x_1, x_2) & \overset{\text{def}}{=} (u_1(x_1)^{v/v_1}, u_2(x_2)^{v/v_2}) \end{array}$$

Ainsi,

$$\begin{aligned} \phi(u(x_1, x_2)) &= (\phi_1(u_1(x_1)^{v/v_1}), \phi_2(u_2(x_2)^{v/v_2})) \\ &= (x_1^{v_1 v/v_1}, x_2^{v_2 v/v_2}) \\ &= (x_1, x_2)^v \end{aligned}$$

Ce résultat est plus général que le protocole de Maurer [67, Théoreme 6.2] car nous n'avons pas besoin que les exposants spéciaux, ni que les algorithmes u_1 et u_2 soient identiques.

Théorème 3.8 (Égalité des pré-images). *Prenons deux homomorphismes partant du même domaine $\mathcal{G}_1 = \mathcal{G}_2 = \mathcal{G}$, $v_1 = v_2$, et u_1, u_2 tel que :*

$$\forall x_1, x_2 \in \mathsf{Im}(\phi), u_1(x_1) = u_2(x_2)$$

Alors, l'homomorphisme suivant allant de $\mathcal{G}$ au produit de $\mathcal{H}_1$ et $\mathcal{H}_2$ est un homomorphisme spécial :

$$\begin{array}{ll} \phi & : \mathcal{G} \to \mathcal{H}_1 \times \mathcal{H}_2 \\ \phi(x) & \overset{def}{=} (\phi_1(x), \phi_2(x)) \end{array}$$

Preuve. Prenons $v \overset{\text{def}}{=} v_1$ et $u(x_1, x_2) \overset{\text{def}}{=} u_1(x_1) = u_2(x_2)$,

$$\begin{aligned} \phi(u(x_1, x_2)) &= (\phi_1(u_1(x_1)), \phi_2(u_2(x_2))) \\ &= (x_1^{v_1}, x_2^{v_2}) \\ &= (x_1, x_2)^v \end{aligned}$$

Nous pouvons la construction précédente pour construire un Σ protocole qui prouve la correction de l'échange de clés de Diffie-Hellman. Étant donné un groupe avec un ordre premier q et un générateur g, nous voulons prouver qu'un triplet d'éléments du groupe de la forme (α, β, γ) est un triplet de Diffie-Hellman, c'est-à-dire que si $\alpha = g^a$ et $\beta = g^b$, alors $\gamma = g^{ab}$. Nous construisons les homomorphismes $\phi_1(x) = g^x$, et $\phi_2(x) = \beta^x$. La connaissance d'une pré-image a de (α, γ) implique que (α, β, γ) est un triplet de Diffie-Hellman (et que γ est une clé partagée de Diffie-Hellman valide).

3.4 Σ Protocoles basés sur des permutations *claw-free*

Cette section décrit une autre construction générale de Σ protocoles, en reprenant l'idée de foncteur générique vue dans la section précédente. Nous nous intéressons ici aux paires de permutations *claw-free* (sans pince) au lieu des homomorphismes spéciaux.

Définition 3.9 (*Trapdoor permutations* (Permutation avec trappe)). *Une famille de* Trapdoor permutations *est un triplet* $(\mathcal{KG}, f, f^{-1})$*, où* $\mathcal{KG}$ *est un algorithme aléatoire de génération de clé, qui génère des paires de clés de la forme* (pk, sk)*, tel que* $f(pk, \cdot)$ *est une permutation sur un domaine* D*, et* $f^{-1}(sk, \cdot)$ *est son inverse. Ces permutations sont dites* Trapdoor *(avec trappe), car les fonctions* f *et* f^{-1} *doivent être munies d'une information supplémentaire pour être calculées efficacement en temps polynomial.*

Définition 3.10 (Paire de permutations *Claw-Free* [51]). *Une paire de permutations avec trappe* (f_0, f_1) *sur le même domaine* D *est* claw-free *si il est difficile de calculer* $x, y \in D$ *tel que* $f_0(pk, x) = f_1(pk, y)$*.*

Étant donné une paire de permutation *claw-free* f, et un *bitstring* $a \in \{0,1\}^k$, nous définissons

$$f_{[a]}(b) = f_{a_1}(f_{a_2}(\dots(f_{a_k}(b))\dots))$$

où a_i représente le $i^{\text{ème}}$ bit de a.

Théorème 3.11 (Σ Protocole basé sur des permutations *Claw-Free*). *Prenons* (f_0, f_1) *une paire de permutations* Claw-Free *sur* D *et prenons une relation* R *tel que*

$$\begin{aligned} &R(pk, sk) \iff \\ &\quad \forall x, f_0(pk, f_0^{-1}(sk, x)) = f_0^{-1}(sk, f_0(pk, x)) = x \wedge \\ &\qquad f_1(pk, f_1^{-1}(sk, x)) = f_1^{-1}(sk, f_1(pk, x)) = x \end{aligned}$$

Le protocole suivant est un Σ protocole pour une relation R :

$$\begin{array}{lll} \mathsf{P}_1(pk, sk) & \stackrel{def}{=} & y \stackrel{\$}{\leftarrow} D;\ \textsf{return}\ (y, y) \\ \mathsf{P}_2(pk, sk, y, c) & \stackrel{def}{=} & \textsf{return}\ f_{[c]}^{-1}(sk, y) \\ \mathsf{V}_1(pk, r) & \stackrel{def}{=} & c \stackrel{\$}{\leftarrow} \{0,1\}^k;\ \textsf{return}\ c \\ \mathsf{V}_2(pk, r, c, s) & \stackrel{def}{=} & \textsf{return}\ \left(f_{[c]}(pk, s) = r\right) \end{array}$$

excepté qu'il ne satisfait pas la propriété de correction spéciale.

Preuve.

Complétude

La preuve suit le même raisonnement que la preuve de Complétude pour les Σ^ϕ protocoles. Après avoir déplié les appels de fonction, nous obtenons le but suivant.

$$b \leftarrow f_{[c]}(pk, f_{[c]}^{-1}(sk, y)) = y \sim_{\{b\}}^{R(pk,sk)} b \leftarrow \mathsf{true}$$

Nous utilisons le fait que la paire (pk, sk) est dans R pour prouver que $f_{[c]}(pk, f_{[c]}^{-1}(sk, y)) = y$ par récurrence sur c.

Special Honest Verifier Zero-Knowledge

Le programme suivant est un simulateur sHVZK simulateur pour le protocole,

```
Simulator S(pk, c) :
 s <-$ D;
 r <- f_[c](pk, s);
 return (r, s)
```

Pour prouver que

$$\mathsf{Protocole}(pk, sk, c) \sim^{\{pk,c\}\wedge R(pk,sk)}_{\{r,c,s\}} (r, s) \leftarrow \mathsf{S}(pk, c)$$

Nous déplions tous les appels de fonction dans les deux programmes et appliquons les optimisations de propagation de constante et d'élimination de code mort, nous obtenons le but suivant :

$$\begin{array}{l} r \stackrel{\$}{\leftarrow} D; \\ s \leftarrow f^{-1}_{[c]}(sk, r) \end{array} \sim^{\{pk,sk,c\}}_{\{r,s\}} \begin{array}{l} s \stackrel{\$}{\leftarrow} D; \\ r \leftarrow f_{[c]}(pk, s) \end{array}$$

qui est prouvable en utilisant le fait que f est une paire de permutation.

Nous observons que le protocole ci-dessus ne satisfait pas forcément la propriété de correction spéciale. À la place, il satisfait la propriété de *collision intractability* : aucun algorithme efficace ne peut trouver deux conversations acceptantes avec des challenges différents en ayant le même *commitment*, c'est à dire une collision, avec une probabilité non négligeable. Ce type de protocole est utilisé dans les protocoles de signatures.

Théorème 3.12. *Il est difficile de calculer efficacement une collision pour le protocole du Théorème 3.11.*

Preuve. Par l'absurde, car à partir de deux conversations acceptantes (r, c_1, s_1), (r, c_2, s_2) pour une entrée publique pk avec $c_1 \neq c_2$, il est possible de trouver efficacement une *claw* (b, b') tel que $f_0(b) = f_1(b')$. Les deux conversations sont acceptantes, nous avons alors :

$$f_{[c_1]}(pk, s_1) = f_{[c_2]}(pk, s_2) = r$$

L'algorithme suivant calcule une *claw* (griffe)

```
find_claw(s1, c1, s2, c2) :
   if head(c1) = head(c2)
   then find_claw(s1, tail(c1), s2.tail(c2))
   else if head(c1) = 0
          then(f_[tail(c1)](pk, s1), f_[tail(c2)](pk, s2))
          else (f_[tail(c2)](pk, s2), f_[tail(c1)](pk, s1))
```

L'algorithme s'exécute en temps polynomial car les permutations f_0 et f_1 peuvent être évaluées en temps polynomial, et c_1, c_2 sont bornés par un polynôme. Pour un ensemble de *challenges* borné par un polynôme, ce qui contredit le fait que la paire de permutations (f_0, f_1) est *claw-free*.

3.4.1 Instance de Σ protocole basé sur les permutations *claw-free*

Goldwasser, Micali et Rivest ont prouvé que les paires de permutations *claw-free* existent si la factorisation d'entier est un problème difficile [51] ; Cramer [38] utilise cette construction pour définir un Σ protocole. Dans la suite nous montrerons comment formaliser en Coq cette famille de permutations *trapdoor claw-free* qui satisfait le Théorème 3.11.

Nous rappelons que le symbole de Jacobi pour un nombre a et un nombre premier impair p est défini de la manière suivante [1]

$$\begin{pmatrix} a \\ p \end{pmatrix} = \begin{cases} 0 & \text{si } a \equiv 0 \pmod p \\ +1 & \text{si } a \not\equiv 0 \pmod p \text{ et } a \text{ est un carré parfait modulo } p \\ -1 & \text{sinon} \end{cases}$$

Le symbole de Jacobi pour un nombre a et un nombre composé N est défini comme étant le produit des symboles de Jacobi pour les facteurs premiers de N. Si $N = pq$, alors $\binom{a}{N}$ est défini de la manière suivante :

$$\begin{pmatrix} a \\ N \end{pmatrix} = \begin{pmatrix} a \\ p \end{pmatrix}\begin{pmatrix} a \\ q \end{pmatrix}$$

Nous prenons deux nombres premiers différents p et q, tel que $p \equiv 3 \pmod 8$, et $q \equiv 7 \pmod 8$, et définissons le nombre n par le produit de p et q ($n = pq$ un tel n est souvent appelé entier de Blum). Le domaine D_n est défini par :

$$D_n \stackrel{\text{def}}{=} \left\{ x : \mathbb{Z}_p^* \;\middle|\; 0 < x < \tfrac{n}{2} \wedge \begin{pmatrix} x \\ n \end{pmatrix} = 1 \right\}$$

La paire de permutations suivante sur D_n est *claw-free* :

$$f_0(x) \stackrel{\text{def}}{=} \begin{cases} x^2 \bmod n & \text{si } 0 < x^2 \bmod n < n/2 \\ -x^2 \bmod n & \text{si } n/2 < x^2 \bmod n < n \end{cases}$$

$$f_1(x) \stackrel{\text{def}}{=} \begin{cases} 4x^2 \bmod n & \text{si } 0 < 4x^2 \bmod n < n/2 \\ -4x^2 \bmod n & \text{si } n/2 < 4x^2 \bmod n < n \end{cases}$$

Si les facteurs premiers de n sont connus, ces permutations peuvent être facilement inversées en calculant les racines carrées dans $\mathbb{Z}_n^*$ et en appliquant le théorème des restes Chinois ; de plus, nous pouvons montrer qu'à partir d'une *claw* (x, y), les facteurs premiers p et q peuvent être retrouvés efficacement [38]. Nous utilisons la construction du Théorème 3.11 pour obtenir un Σ protocole avec la relation $R(n, (p, q)) \stackrel{\text{def}}{=} n = pq$.

Nous formalisons D_n comme un groupe. Nous utilisons la formalisation des groupes $\mathbb{Z}_p^*$ définie par David Nowak, qui contient les définitions et propriétés de $\mathbb{Z}_p^*$ et les définitions du symbole de Legendre et de Jacobi. La multiplication dans D_n est définie de la manière suivante :

1. Le symbole de Jacobi équivaut au symbole de Legendre dans ce cas.

$$x.y \stackrel{\text{def}}{=} \begin{cases} x.y \bmod n & \text{if } 0 < x.y \bmod n < n/2 \\ -x.y \bmod n & \text{if } n/2 < x.y \bmod n < n \end{cases}$$

L'utilisation des modules types de Coq permet de garantir que le domaine des fonctions est bien l'intervalle $[0, n/2]$ et que $\binom{x.y}{n} = 1$ à partir de $\binom{x}{n} = 1$ et $\binom{y}{n} = 1$. Nous définissons également les deux opérateurs opposé et inverse et les propriétés de groupe. Une fois le groupe défini, les fonction f_0 et f_1 sont définies en Coq par $f_0(x) \stackrel{\text{def}}{=} x^2$ et $f_1(x) \stackrel{\text{def}}{=} 4x^2$.

3.5 Composition de Σ protocoles

Il y a deux moyens immédiats de composer deux Σ protocoles (P^1, V^1) et (P^2, V^2) avec les relations de connaissance R_1 et R_2 respectivement : Composition-ET, et Composition-OU. Le premier prouve que le *prouveur* connaît un témoin w tel que $(x_1, w) \in R_1$ et $(x_2, w) \in R_2$ soient vraies. Le deuxième permet de prouver qu'il connaît un témoin prouvant que soit $(x_1, w) \in R_1$ est vrai soit $(x_2, w) \in R_2$ est vrai, mais sans révéler lequel est vrai. Ce résultat peut être étendu aux preuves de composition de n'importe quelle formule booléenne. Même si les deux compositions que nous allons expliquer sont simples, elles sont très puissantes, et sont utilisées par exemple dans le vote électronique[38].

3.5.1 Composition ET

Deux Σ protocoles peuvent être combinés pour en faire un seul qui prouve simultanément la connaissance des deux relations, c'est à dire que la relation R est définie à partir de R_1 et R_2 de la manière suivante :

$$R \stackrel{\text{def}}{=} \{((x_1, x_2), (w_1, w_2)) \mid (x_1, w_1) \in R_1 \wedge (x_2, w_2) \in R_2\}$$

Nous formalisons les compositions ET par un foncteur, qui prend deux Σ protocoles (P^1, V^1) et (P^2, V^2) et qui a le type d'un module Σ. Sans perte de généralité, nous demandons que les *vérifieurs* des deux protocoles fassent leur tirages aléatoires à partir d'un bitstring d'une même longueur k. Cette construction est la composition parallèle de deux sous protocoles qui utilisent le même *challenge* choisi uniformément.

$$\begin{array}{l}
\mathsf{P}_1((x_1, x_2), (w_1, w_2)) \stackrel{\text{def}}{=} \\
\quad (r_1, state_1) \leftarrow \mathsf{P}_1^1(x_1, w_1); \\
\quad (r_2, state_2) \leftarrow \mathsf{P}_1^2(x_2, w_2); \\
\quad \mathsf{return}\ ((r_1, r_2), (state_1, state_2)) \\
\mathsf{P}_2((x_1, x_2), (w_1, w_2), state_1, state_2, c) \stackrel{\text{def}}{=} \\
\quad s_1 \leftarrow \mathsf{P}_2^1(x_1, w_1, state_1, c); \\
\quad s_2 \leftarrow \mathsf{P}_2^2(x_2, w_2, state_1, c); \\
\quad \mathsf{return}\ (s_1, s_2) \\
\mathsf{V}_1((x_1, x_2), (r_1, r_2)) \stackrel{\text{def}}{=} c \stackrel{\$}{\leftarrow} \{0, 1\}^k;\ \mathsf{return}\ c \\
\mathsf{V}_2((x_1, x_2), (r_1, r_2), c, (s_1, s_2)) \stackrel{\text{def}}{=} \\
\quad b_1 \leftarrow \mathsf{V}_2^1(x_1, r_1, c, s_1) \\
\quad b_2 \leftarrow \mathsf{V}_2^2(x_2, r_2, c, s_2) \\
\quad \mathsf{return}\ (b_1 = \mathsf{true} \wedge b_2 = \mathsf{true})
\end{array}$$

Nous observons que V_1 n'est pas construit à partir de V_1^1 et V_1^2, ceci à cause de la preuve de la propriété de correction, deux exécutions pour la même entrée publique x avec le même *commitment* r, mais avec différents challenges $c \neq c'$ va donner deux exécutions de chaque sous-protocoles avec des challenges différents. Si le challenge du protocole principal avait été construit à partir des challenges calculés par V_1^1 and V_1^2, par exemple en les concaténant, nous n'aurions pas pu conclure que les deux challenges dans la pair de conversation extraite des sous protocoles sont différents - mais nous aurions pu seulement montrer que c'est le cas pour une seul sous protocole. Pour résoudre ce problème, un nouveau challenge commun est tiré dans V_1 et sera utilisé dans les deux sous protocoles. Cette méthode résout le problème, mais requiert que les deux sous protocoles satisfassent la propriété sHVZK, car nous avons besoin de simuler les sous protocoles pour un challenge fixé.

La combinaison ET représente une paire des deux sous protocoles tout en respectant la structure d'un Σ protocole. Toutes les preuves ont la même structure, les appels de procédure sont dépliés, les buts sont transformés pour pouvoir appliquer les propriétés des sous-protocoles et conclure. Nous donnons une idée de la preuve pour sHVZK et la complétude spéciale ; le détail de la preuve des ces deux propriétés et de la complétude se trouve dans [22].

sHVZK

Le simulateur sHVZK pour la composition peut simplement appeler les simulateurs des deux sous protocoles pour obtenir une conversation pour chaque sous-protocoles avec le même challenge c, les conversations sont ensuite combinées pour obtenir une conversation du protocole ET :

Simulator $\mathsf{S}((x_1, x_2), c)$:
$(r_1, s_1) \leftarrow \mathsf{S}_1(x_1, c)$;
$(r_2, s_2) \leftarrow \mathsf{S}_2(x_2, c)$;
$\mathsf{return}((r_1, r_2), (s_1, s_2))$

Correction

La Correction requiert l'existence d'un extracteur de connaissance PPT qui calcule un témoin pour la relation R à partir de deux exécutions acceptantes du protocole avec différents challenges, mais avec le même *commitment*. Un témoin est calculé à partir des témoins des deux sous protocoles, de la manière suivante :

$\mathsf{KE}((x_1, x_2), (r_1, r_2), c, c', (s_1, s_2), (s_1', s_2'))$:
$w_1 \leftarrow \mathsf{KE}^1(x_1, r_1, c, c', s_1, s_1')$;
$w_2 \leftarrow \mathsf{KE}^2(x_2, r_2, c, c', s_2, s_2')$;
$\mathsf{return}\ (w_1, w_2)$

Nous utilisons le challenge du protocole comme le challenge pour les sous protocoles, nous pouvons extraire deux exécutions acceptantes avec deux différents challenges vu que $c \neq c'$. Concrètement, à partir de

$$\Pr_{b \leftarrow \mathsf{V}_2((x_1,x_2),(r_1,r_2),c\ ,(s_1,s_2))}[b = \mathsf{true}\ :\ \mu] = 1$$
$$\Pr_{b \leftarrow \mathsf{V}_2((x_1,x_2),(r_1,r_2),c',(s_1',s_2'))}[b = \mathsf{true}\ :\ \mu] = 1$$

nous avons pour $i = 1, 2$,

$$\Pr_{w_i \leftarrow \mathsf{KE}^i(x_i,r_i,c,c',s_i,s_i')}[(x_i, w_i) \in R_i\ :\ \mu] = 1$$

à partir de la complétude des deux sous protocoles et du fait que

$$\Pr_{b_i \leftarrow \mathsf{V}_2^i(x_i,r_i,c\ ,s_i)}[b_i = \mathsf{true}\ :\ \mu] = 1$$
$$\Pr_{b_i \leftarrow \mathsf{V}_2^i(x_i,r_i,c',s_i')}[b_i = \mathsf{true}\ :\ \mu] = 1$$

3.5.2 Composition OU

Deux Σ protocoles peuvent aussi être composés pour obtenir un protocole qui montre que le prouveur connaît un seul témoin d'un des deux sous protocoles, mais sans dire lequel. La construction est reliée à la capacité de simuler des exécutions acceptantes ; l'idée est que le *prouveur* exécute le protocole pour lequel il connaît le témoin, et utilise le simulateur pour l'autre. La relation de connaissance est suggérée dans [42],

$$\hat{R} \stackrel{\text{def}}{=} \{((x_1, x_2), w) \mid (x_1, w) \in R_1\ \vee (x_2, w) \in R_2\}$$

Cette relation pose problème, car le *prouveur* peut faire appel à des demandes irréalisables au simulateur. Comme cela est expliqué dans [38], le simulateur peut échouer si l'entrée $x \notin \mathsf{dom}(R)$. Cependant, pour prouver la complétude pour la relation ci dessous, le simulateur peut fonctionner en dehors du domaine de la relation de connaissance. À la place, nous prouvons la complétude (et sHVZK) de la composition avec une relation de connaissance dont le domaine est restreint au produit cartésien des domaines des relations de connaissance des sous protocoles, i.e.

$$R \stackrel{\text{def}}{=} \left\{ ((x_1, x_2), w) \;\middle|\; \begin{array}{l} ((x_1, w) \in R_1 \wedge x_2 \in \mathsf{dom}(R_2)) \vee \\ ((x_2, w) \in R_2 \wedge x_1 \in \mathsf{dom}(R_1)) \end{array} \right\}$$

Nous ne pouvons pas prouver la propriété de complétude pour R, nous pouvons seulement la prouver pour $\hat{R}$. La raison est qu'une exécution acceptante de combinaison de protocole garantit seulement l'existence d'un témoin pour l'entrée publique d'un des protocoles, la simulation de l'autre protocole peut réussir si l'entrée n'est pas dans la relation de connaissance. À partir de deux exécutions acceptantes de la combinaison des protocoles avec des challenges différents, nous ne pouvons pas extraire deux exécutions acceptantes avec des challenges différents pour chaque sous protocoles ; nous pouvons seulement garantir que nous pouvons le faire pour un d'entre eux. Nous ne perdons rien en prouvant la complétude avec une relation R plus faible. Si nous admettons que la paire (x_1, x_2) est une entrée publique où une composante n'appartient pas au domaine de la relation de connaissance correspondante, nous ne pourrons rien dire à propos de la réussite du simulateur. Le simulateur doit pouvoir échouer, en révélant que le *prouveur* n'aurait pas pu connaître un témoin pour l'entrée correspondante, et rendrait le protocole inutile pour une telle entrée.

Comparé à la composition ET, montrer que la composition OU est un Σ protocole est plus difficile. La première phase du prouveur a besoin d'utiliser le simulateur

pour un des protocoles, ce qui oblige à passer une conversation acceptante dans la second phase du prouveur. Étant donné $(x_1, w) \in R_1$, le *prouveur* doit exécuter le *prouveur* du premier protocole et le simulateur du second, et retourner les *commitments* des deux sous protocoles. Le résultat de la simulation et du challenge sont passés dans *state*.

$$
\begin{array}{l}
\mathsf{P}_1((x_1, x_2), w) \stackrel{\text{def}}{=} \\
\quad \textsf{if } (x_1, w) \in R_1 \textsf{ then} \\
\qquad (r_1, \mathit{state}_1) \leftarrow \mathsf{P}_1^1(x_1, w); \\
\qquad c_2 \xleftarrow{\$} \{0,1\}^k; \\
\qquad (r_2, s_2) \leftarrow \mathsf{S}_2(x_2, c_2); \\
\qquad \mathit{state} \leftarrow (\mathit{state}_1, c_2, s_2) \\
\quad \textsf{else} \\
\qquad (r_2, \mathit{state}_2) \leftarrow \mathsf{P}_1^2(x_2, w); \\
\qquad c_1 \xleftarrow{\$} \{0,1\}^k; \\
\qquad (r_1, s_1) \leftarrow \mathsf{S}_1(x_1, c_1); \\
\qquad \mathit{state} \leftarrow (\mathit{state}_2, c_1, s_1) \\
\quad \textsf{return } ((r_1, r_2), \mathit{state})
\end{array}
$$

Ci-dessus, le test $(x_1, w) \in R_1$ permet au prouveur de savoir quelle relation correspond au témoin w, cela permet de savoir quel sous protocole doit être exécuté et lequel doit être simulé. Le *commitment* (r_1, r_2) est passé au *vérifieur* qui renvoie un bitstring choisi uniformément au prouveur, la composition est un protocole *public-coin*,

$$\mathsf{V}_1((x_1, x_2), (r_1, r_2)) \stackrel{\text{def}}{=} c \xleftarrow{\$} \{0,1\}^k;\ \textsf{return } c$$

Nous pouvons dire sans perte de généralité que $(x_1, w) \in R_1$. Dans la seconde phase, le *prouveur* construit un challenge pour le premier protocole en faisant un xor entre le challenge c du protocole OU avec le challenge utilisé dans la simulation du deuxième protocole dans la première phase. La seconde phase du prouveur est ensuite exécutée du premier protocole pour calculer la réponse. Le résultat de la seconde phase est construit à partir des challenges des deux protocoles et le *prouveur* répond (le challenge venant du protocole simulé vient de l'*état*) :

$$
\begin{array}{l}
\mathsf{P}_2((x_1, x_2), w, (\mathit{state}, c', s), c) \stackrel{\text{def}}{=} \\
\quad \textsf{if } (x_1, w) \in R_1 \textsf{ then} \\
\qquad \mathit{state}_1 \leftarrow \mathit{state};\ c_2 \leftarrow c';\ s_2 \leftarrow s; \\
\qquad c_1 \leftarrow c_2 \oplus c; \\
\qquad s_1 \leftarrow \mathsf{P}_2^1(x_1, w, \mathit{state}_1, c_1) \\
\quad \textsf{else} \\
\qquad \mathit{state}_2 \leftarrow \mathit{state};\ c_1 \leftarrow c';\ s_1 \leftarrow s; \\
\qquad c_2 \leftarrow c_1 \oplus c; \\
\qquad s_2 \leftarrow \mathsf{P}_2^2(x_2, w, \mathit{state}_2, c_2) \\
\quad \textsf{return } ((c_1, s_1), (c_2, s_2))
\end{array}
$$

Le *vérifieur* accepte la conversation quand l'exécution des deux protocoles est acceptée et que le challenge est bien le OU-exclusif des challenges utilisés dans les deux protocoles,

$$
\begin{array}{l}
\mathsf{V}_2((x_1, x_2), (r_1, r_2), c, ((c_1, s_1), (c_2, s_2))) \stackrel{\text{def}}{=} \\
\quad b_1 \leftarrow \mathsf{V}_2^1(x_1, r_1, c_1, s_1); \\
\quad b_2 \leftarrow \mathsf{V}_2^2(x_2, r_2, c_2, s_2); \\
\quad \textsf{return } (c = c_1 \oplus c_2 \wedge b_1 = \textsf{true} \wedge b_2 = \textsf{true})
\end{array}
$$

Completeness

La preuve est un peu plus compliquée que celle de la composition ET, car seulement un des deux protocoles est exécuté réellement, pendant que l'autre est simulé, et dépend de la connaissance du *Prouveur*. La preuve est alors séparée en deux cas :

– **cas** $(x_1, w) \in R_1$: les grandes lignes de la preuve sont décrites ci dessous :

$$\begin{aligned}\mathsf{Protocole}((x_1,x_2),w) \simeq\ & \mathsf{Protocole}_1(x_1,w);\\ & c_2 \stackrel{\$}{\leftarrow} \{0,1\}^k; (r_2,s_2) \leftarrow \mathsf{S}_2(x_2,c_2)\\ \simeq\ & \mathsf{Protocole}_1(x_1,w); \mathsf{Protocole}_2(x_2,w')\end{aligned}$$

La première équivalence est immédiate en dépliant les procédures et en simplifiant le but. La deuxième se prouvant en s'aidant du fait que $\exists w', R_2(x_2, w')$ et de la propriété de sHVZK du deuxième protocole. La preuve se finit en appliquant les propriétés de complétude des sous protocoles.

– **cas** $(x_2, w) \in R_2$: Idem.

sHVZK

Le simulateur de la composition OU se construit facilement à partir des simulateurs des sous protocoles.

Simulator $\mathsf{S}((x_1, x_2), c)$:
$c_2 \stackrel{\$}{\leftarrow} \{0,1\}^k$;
$c_1 \leftarrow c \oplus c_2$;
$(r_1, s_1) \leftarrow \mathsf{S}_1(x_1, c_1)$;
$(r_2, s_2) \leftarrow \mathsf{S}_2(x_2, c_2)$;
return $((r_1, r_2), ((c_1, s_1), (c_2, s_2)))$

Comme plus haut, la preuve est divisée en deux cas.

– **cas** $(x_1, w) \in R_1$:

$$\begin{aligned}\mathsf{Protocol}((x_1,x_2),w) \simeq\ & \mathsf{Protocole}_1(x_1,w);\ \mathsf{S}_2(x_2)\\ \simeq\ & \mathsf{S}_1(x_1);\ \mathsf{S}_2(x_2)\\ \simeq\ & \mathsf{S}((x_1,x_2),c)\end{aligned}$$

Où les première et dernière étape sont simplement un dépliage de procédure et des simplifications de buts. La deuxième étape applique la propriété de HVZK de S_1 (qui s'obtient à partir de la propriété de sHVZK en appliquant le Théorème 3.2).

– **cas** $(x_2, w) \in R_2$: Idem.

Correction

Contrairement à la composition ET, la combinaison OU ne satisfait pas la propriété que des exécutions avec des challenges différents garantissent que les challenges des sous protocoles sont différents également. Ce n'est pas aussi problématique que pour la composition ET, car il suffit de calculer un w tel que $(x_1, w) \in R_1$ ou $(x_2, w) \in R_2$ soit vrai. De plus, à partir de

$$c = c_1 \oplus c_2 \neq c' = c'_1 \oplus c'_2$$

Soit $c_1 \neq c'_1$ soit $c_1 = c'_1$ et nous avons nécessairement $c_2 \neq c'_2$. L'extracteur de connaissance a juste à faire l'analyse de cas :

```
KE((x1, x2), (r1, r2), c, c',
   ((c1, s1), (c2, s2)), ((c'1, s'1), (c'2, s'2))) :
if c1 ≠ c'1 then
   w ← KE^1(x1, r1, c1, c'1, s1, s'1)
else
   w ← KE^2(x2, r2, c2, c'2, s2, s'2)
return w
```

Prenons deux exécutions acceptantes de la composition des protocoles avec le même *commitment* et $c \neq c'$:

$$((x_1, x_2), (r_1, r_2), c, ((c_1, s_1), (c_2, s_2)))$$
$$((x_1, x_2), (r_1, r_2), c', ((c'_1, s'_1), (c'_2, s'_2)))$$

– **cas** $c_1 \neq c'_1$: À partir de la correction spéciale de $\mathsf{Protocole}_1$,

$$\Pr\left[w_1 \leftarrow \mathsf{KE}^1(x_1, r_1, c_1, c'_1, s_1, s'_1) : (x_1, w_1) \in R_1\right] = 1$$

– **cas** $c_1 = c'_1$ (et donc $c_2 \neq c'_2$) : À partir de la correction spéciale de $\mathsf{Protocole}_2$,

$$\Pr\left[w_2 \leftarrow \mathsf{KE}^2(x_2, r_2, c_2, c'_2, s_2, s'_2) : (x_2, w_2) \in R_2\right] = 1$$

Travaux reliés

Notre travail participe à l'intérêt général actuel porté aux Σ protocoles, et partage une certaine motivation avec les travaux récents. Notre description des Σ^ϕ protocoles coïncide avec l'unification de Maurer [67] des preuves de connaissance de pre-image d'homomorphisme de groupe. En pratique, Maurer donne un protocole principal qui utilise un homomorphisme de groupe, qui dans notre formalisation correspond au module des Σ^ϕ protocoles de la Section. 3.3, Maurer montre que (dans le Théorème 3) avec les bonnes hypothèses ce protocole est un Σ protocole. Il donne plusieurs instances de protocoles en prenant des homomorphismes de groupe qui satisfont les bonnes hypothèses.

Notre travail est lié au travaux récents de Bangerter et Al. [15, 14] qui conçoivent et implémentent des preuves de connaissances *Zero-Knowledge* efficaces. Ils fournissent un ensemble d'hypothèses sur les homomorphismes ϕ sous lequel le Σ^ϕ protocole correspondant, est un Σ protocole [15, Theoreme 1], ils généralisent ensuite leur résultat ce qui permet de considérer un ensemble de relations linéaires pour les pre-images d'homomorphisme de groupe [15, Theoreme 2]. Le dernier résultat est utilisé pour justifier la sûreté du compilateur qui génère un code efficace à partir d'une description haut niveau du protocole. Dans [14], les auteurs ont certifié le compilateur en Isabelle/HOL et génèrent automatiquement la preuve de correction.

Les schémas de cryptographie ne doivent pas seulement être sûrs ; elles doivent aussi être utilisées correctement. Dans une série de papier, Backes et Al. [12, 11] développent une méthode sûre pour les protocoles qui utilisent des preuves *zero-knowledge*, ils appliquent leurs analyses pour vérifier des propriétés d'authentification et de secret du *Direct Anonymous Attestation Protocol*. Un objectif ambitieux serait d'utiliser leurs résultats, pour compléter les nôtres, pour certifier la sécurité des protocoles dans le modèle calculatoire. Pour cela, d'important travaux de formalisation sur les preuves *computational soundess* [3, 36] doivent être effectués.

3.6 Conclusion

Dans ce chapitre nous avons montré la formalisation des Σ protocoles dans CertiCrypt. Les points importants de la formalisation sont la construction générique de la classe des Σ^{ϕ} protocoles et les détails que nous avons apportés aux preuves des compositions ET/OU. Ce travail complète les travaux récents dans le domaine, et permet de faire un pas pour formaliser d'autres résultats du monde des preuves *Zero-Knowledge*. Nous pensons que devant le grand nombre de petites variations dans les définitions trouvées dans la littérature, ce travail doit être poursuivi pour améliorer la clarté et la cohérence du domaine.

Comparée aux autres applications de CertiCrypt, comme la vérification des preuve de sécurité des primitives de chiffrement ou bien des fonctions de signature [21, 89, 19], la formalisation présentée dans ce chapitre, donne d'autres objectifs à l'utilisateur. Dans les autres travaux, il était important de développer un ensemble de techniques et de raisonnements qui apparaissent dans les preuves, la formalisation des protocoles n'a pas besoin de raisonnement probabiliste complexe, mais est plus exigeante à propos de la composition de preuve. Ce travail montre que certains choix qui ont était fait lors de la conception de CertiCrypt doivent être révisés et nous ont donné des idées pour améliorer l'outil, rendre les résultats plus réutilisables et qu'ils puissent être composés plus facilement. Par exemple un des problèmes rencontré est quand l'on compose des preuves d'équivalence observationelle, l'utilisateur doit renommer les variables pour les rendre compatibles avec le contexte où la preuve doit être utilisée ; l'utilisateur doit faire appel à la tactique `alloc` pour faire cela, mais une simple heuristique devrait suffire dans la plupart des cas.

Nous pouvons nous servir de la formalisation actuelle pour vérifier d'autres résultats importants à propos des preuves *Zero-Knowledge*. Par exemple d'autres manières de composer des protocoles : composition par séquence [49] ou concurrente [47, 41] ; transformer des preuves *Zero-Knowledge public-coin* en preuves *Zero-Knowledge* plus générales [48], ou d'autres formulations comme les preuves *Zero-Knowledge* non interactives [30] ou d'autre propriétés comme les preuves *Zero-Knowledge* statistiques ou les *Zero-Knowledge* calculatoires au lieu des *perfect Zero-Knowledge*. De plus les Σ protocoles sont à la base de beaucoup de protocoles de vote électronique, identité [38], et *commitment schemes* [38, 40]. Beaucoup de pistes pour de futures formalisations.

Remerciement. Dominique Unruh pour avoir remarqué que notre définition originale de correction était plus faible que la définition standard, et que la définition standard de la composition OU peut être seulement prouvée avec une relation plus forte.

4
EasyCrypt

Nous avons vu qu'il est possible de construire des preuves vérifiables sur machine, à l'aide de CertiCrypt. L'outil permet d'écrire des preuves complexes, rendant le développement de preuves sur machine convaincant. Pourtant CertiCrypt n'a pas atteint une grande popularité auprès des cryptographes.

La preuve de OAEP IND-CCA2 [20] fait plus de 10.000 lignes de script Coq et est décomposée en plus de 30 jeux. Ceci montre qu'il existe un grand contraste entre une preuve papier et une preuve vérifiable sur machine. L'effort requis pour construire une preuve CertiCrypt est trop important et l'utilisation de CertiCrypt demande une trop forte expertise en preuve formelle pour être utilisé par les cryptographes.

L'écriture des preuves dans CertiCrypt est difficile, la construction de la sémantique ou des environnements est répétitive, souvent effectuée par copier-coller. La construction des informations utilisées par les tactiques est laborieuse car si elle échoue, il est difficile de savoir pourquoi. Les sémantiques, les environnements et les informations pourraient être générées par un outil externe ce qui simplifierait l'écriture des preuves.

Les preuves d'invariants génèrent des obligations de preuves assez grosses, mais elles sont souvent dans un fragment décidable de la logique des prédicats et peuvent être généralement prouvées par des prouveurs SMT. Pour le moment Coq permet d'appeler des outils externes, mais ne permet pas de récupérer les traces des prouveurs SMT et reconstruire un certificat en Coq. Là encore écrire un autre outil pour appeler les outils externes serait d'une grande aide.

Sous cette forme il est peu probable que CertiCrypt soit adopté un jour par les cryptographes. Le souhait de Halevi concernant la construction d'un outil pour faire des preuves [54], n'est pas complètement atteint. L'idée est de proposer un nouvel outil, qui simplifierait l'écriture des preuves (sémantique, information, ...), et qui déchargerait les preuves d'invariant à des prouveurs externes.

4.1 EasyCrypt

EasyCrypt est un outil écrit en ML, prenant en entrée une description de preuve qui contient la définition des jeux, des opérateurs, et des différentes étapes qui composent la preuve par jeux. Le principe de construction de preuves est le même que dans CertiCrypt : l'utilisateur prouve des propriétés probabilistes sur des évènements, à partir de preuve d'équivalence opérationnelle, utilisant une logique de Hoare probabiliste (pRHL). Ses équivalences sont appelées jugement pRHL.

Le principe des preuves d'équivalence dans CertiCrypt est d'effectuer des petites transformations sur les jeux avec les tactiques et d'appeler la tactique eqobs_in pour finir la preuve. L'utilisation du calcul de plus faible pré-condiction (wp) se faisait dans certain cas, en particulier pour prouver des invariants. Mais l'expérience a montré que beaucoup de ces transformations de programme étaient déjà incluses dans le calcul de wp.

EasyCrypt propose une approche plus directe, qui consiste à utiliser intensivement le wp et envoyer les obligations de preuve vers des prouveurs SMT. La plus part du temps ce procédé rend la preuve des jugements pRHL complètement automatique.

Les prouveurs SMT sont appelés en utilisant l'outil Why [46] qui permet de prouver des programmes écrits en C, ML ou Java en génerant des obligations de preuves et en les envoyant à des prouveurs SMT. Pour EasyCrypt nous nous servirons uniquement de la partie appel aux SMT de l'outil Why.

Pour chaque étape, EasyCrypt dispose de procédures automatiques qui génèrent pour tout jugement pRHL, un ensemble d'obligations de preuve (ou condition de vérification) écrites dans la logique du premier ordre. La validité des obligations de preuve est suffisante pour assurer la validité des jugements pRHL. De plus l'outil propose des mécanismes d'inférence, pour trouver par exemple les spécifications des adversaires.

La clé de l'efficacité de EasyCrypt est de pouvoir décharger les conditions de vérification dans des prouveurs externes comme les prouveurs SMT, mais permet également d'utiliser des assistants de preuve dans le cas où certaines obligations de preuve ne seraient pas supportées par les outils automatiques.

EasyCrypt permet de raisonner sur les probabilités d'évènement entre deux programmes, mais aussi de calculer les probabilités d'un évènement pour un programme. L'outil combine des règles élémentaires de calcul de probabilité, par exemple la probabilité qu'un élément tiré aléatoirement appartienne à une liste.

Le lecteur pourrait arguer que EasyCrypt offre moins de confiance que CertiCrypt, le wp est écrit en ML et n'est pas prouvé, ainsi que les prouveurs SMT. Pour certifier nos preuves, EasyCrypt permet de générer un fichier Coq compatible avec CertiCrypt. Cette étape permet de valider la preuve effectuée par EasyCrypt, en gardant la même garantie que CertiCrypt. A l'heure actuelle, la génération de preuve est partielle, étant donné que le lien entre les prouveurs SMT et Coq ne permet pas de récupérer les traces des prouveurs et créer un certificat en Coq, ce point est un sujet de recherche [7, 6]. Pour le moment pour finir la preuve, nous demandons à

l'utilisateur de reprouver en Coq, les obligations prouvées par les prouveurs SMT. Ce point sera expliqué en détail dans le chapitre suivant.

Au final, EasyCrypt est un outil automatique, qui permet avec un minium d'effort d'effectuer des preuves tout en gardant la même garantie que CertiCrypt, ce qui nous rapproche de la vision de Halevi et fait de EasyCrypt un candidat pour l'adoption d'un outil pour les cryptographes. Nous soutenons que EasyCrypt est significativement plus simple à utiliser que les outils précédents et permettrait au cryptographes de faire leurs preuves sur machine. Nous appuierons la présentation de EasyCrypt avec deux preuves : Hashed ElGamal qui permettra au lecteur de comparer avec la preuve CertiCrypt du Chapitre 2 et un exemple plus complexe avec le schéma de chiffrement Cramer-Shoup.

4.2 Description d'EasyCrypt

Cette section explique les diverses caractéristiques de l'outil, tout en justifiant les choix de conception effectués.

4.2.1 Entête des preuves

Les fichiers EasyCrypt débutent en déclarant tous les ingrédients qui vont servir dans la preuve. L'entête comprend la déclaration des nouveaux types qui n'existent pas dans les types de base de EasyCrypt, ou bien les constantes, les opérateurs et les adversaires qui serviront dans la suite.

Type

EasyCrypt contient un ensemble de types de base comprenant unit, bool, int, list, option, map et $\mathsf{bitstring}_k$. Les types list, option et map sont polymorphes.

CertiCrypt a montré l'importance du type $\mathsf{bitstring}_k$, dans les preuves de cryptographie, mais l'architecture ne permettait pas de les ajouter dans les types de base. Nous les avons ajoutés dans EasyCrypt. Ce type dépend de la taille des messages k, qui doit être une constante.

Le type map est aussi très important dans les preuves, en particulier pour modéliser les oracles du *Random Oracle Model*. Les map sont des listes d'associations de type $(A \times B)$ list, EasyCrypt fournit un ensemble de fonctions permettant de manipuler ces listes.

Pour avoir la même généricité que dans CertiCrypt, les types de base peuvent être étendus. L'utilisateur peut créer des types abstraits comme les groupes, il peut aussi créer de nouveaux types à partir des types existants, pour ajouter une sorte de notation et clarifier les preuves.

Le type des bitstrings est traité différemment car c'est un type dépendant et ce genre de type n'est pas supporté par les prouveurs SMT. EasyCrypt ne possède pas d'axiome ou d'opérateur sur les bitstrings, nous expliquerons pourquoi.

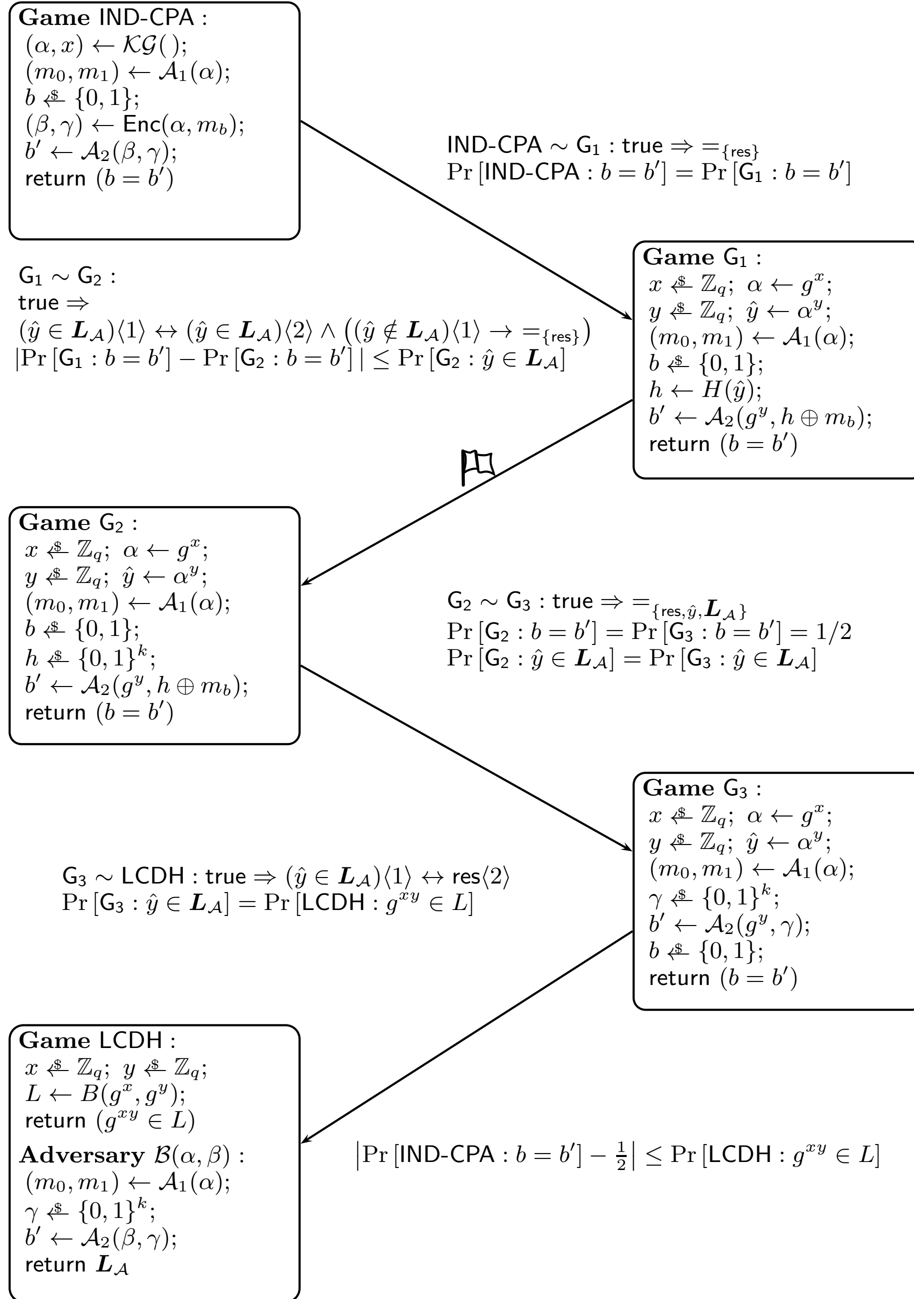


Fig. 4.1. Preuve par jeu de la sécurité IND-CPA de Hashed ElGamal

```
type group

cnst q    : int
cnst g    : group
cnst k    : int
cnst zero : bitstring{k}

type skey    = int
type pkey    = group
type key     = skey * pkey
type message = bitstring{k}
type cipher  = group * bitstring{k}

op (*) : group, group → group = mul
op (^) : group, int → group = pow
op (^^) : bitstring{k}, bitstring{k} → bitstring{k} = xor

axiom pow_mult : ∀(x:int,y:int). { (g^x)^y = g^(x*y) }
axiom xor_comm : ∀ (x:bitstring{k}, y:bitstring{k}).
  { (x ^^ y) = (y ^^ x) }
axiom xor_assoc : ∀ (x:bitstring{k},y:bitstring{k},z:bitstring{k}).
  { ((x ^^ y) ^^ z) = (x ^^ (y ^^ z)) }
axiom xor_zero : ∀ (x:bitstring{k}). { (x ^^ zero) = x }
axiom xor_cancel : ∀ (x:bitstring{k}). { (x ^^ x) = zero }

adversary A1(pk:pkey) : message * message    { group → message}
adversary A2(pk:pkey,c:cipher) : bool        { group → message}
```

Fig. 4.2. Entete de la preuve de Hashed ElGamal

Dans la preuve de Hashed-ElGamal, la figure 4.2 montre la création du type abstrait group, sans donner d'interprétation. En plus des types abstraits, l'utilisateur définit les types skey et pkey qui représentent le type des clés privées et publiques, key représente le type des couples clé privé/clé publique.

Constantes

L'outil contient des constantes de base comme true, false, les entiers, la liste vide (nil) et l'élément none du type option. L'utilisateur peut aussi ajouter de nouvelles constantes. Comme pour les types, ces constantes peuvent être abstraites ou être définies à partir des autres constantes et des opérateurs.

Dans la figure 4.2, les constantes déclarées sont l'ordre du groupe q, un générateur g, et un entier k qui représentera la tailles des bitstrings. zero représente un bitstring de taille k, qui sera interprété comme le bitstring nul. Les tailles de bitstrings sont forcément des constantes (ici k).

Opérateurs

L'outil contient de nombreuses opérations sur les types de base comme les opérateurs logiques, les fonctions standards (arithmétiques, liste,...). EasyCrypt comprend cinq opérateurs sur les map :

– update : permet de mettre à jour une valeur de la liste d'association, la syntaxe est la suivante : $L[x] = y$ si x n'est pas associée à une valeur, le couple (x, y) est ajouté, sinon la valeur est mise à jour.

- get : permet d'obtenir la valeur d'un élément, l'instruction $y = L[x]$ affecte la valeur associée à x dans y
- in_dom : permet de tester si une valeur est dans le domaine de la liste d'association
- in_rng : permet de tester si une valeur est dans l'image de la liste d'association
- empty_map : initialise par une map vide, $L = \text{empty_map}$

De nouveaux opérateurs peuvent être déclarés par l'utilisateur. L'utilisateur donne la notation qui servira dans l'écriture des jeux. Cette notation peut être préfixe ou infixe (avec parenthèse). Dans la preuve trois opérateurs infixes sont déclarés. L'utilisateur indique le type des arguments et du résultat et le nom de l'opérateur qui servira dans la génération des fichiers Why ou des fichiers Coq.

Dans la figure 4.2, les nouveaux opérateurs définis sont la loi de groupe, la fonction puissance et le OU exclusif sur les bitstrings. Le OU exclusif n'appartient pas aux opérateurs de base, car le type bitstring_k est dépendent et n'est pas compatible avec les prouveurs SMT. C'est pourquoi l'utilisateur devra écrire une fonction sur les bitstrings par longueur de bitstring.

Les opérateurs sur des bitstrings, pourraient être inclus directement dans EasyCrypt. Le problème est que les SMT ne sont pas compatibles avec les types dépendants. Deux solutions pourraient être utilisées, la première serait de parcourir l'ensemble des définitions du fichier et de répertorier toutes les tailles utilisées et générer autant d'opérateurs que nécessaire. La deuxième est de supprimer la dépendance entre le bitstring et le type et de créer un prédicat vérifiant que le bitstring a bien la bonne taille.

Axiomes

EasyCrypt contient un ensemble d'axiomes servant de spécification des opérateurs de base. Ces axiomes sont envoyés au prouveur SMT. L'utilisateur peut également rajouter de nouveaux axiomes.

Dans la preuve, les axiomes définissent les propriétés des lois de groupe et de la fonction puissance, et l'opérateur XOR, comme le fait que la multiplication des groupes est distributive ou que le XOR est commutatif et associatif. Les deux derniers axiomes sont des propriétés du bitstring nul.

Les axiomes sur les bitstrings posent le même problème que les opérateurs sur les bitstrings, et les axiomes de base de EasyCrypt ne contiennent aucun de ces axiomes pour les mêmes raisons.

Adversaires

Les adversaires sont définis comme des fonctions abstraites qui ont accès à un ensemble de procédures. Contrairement à CertiCrypt les adversaires n'ont pas accès aux variables globales. Si l'utilisateur veut définir des adversaires actifs, il doit faire passer un état entre les adversaires. Si les adversaires doivent accéder à une variable globale utilisée par le jeu, ils peuvent toujours le faire par l'intermédiaire d'un oracle.

Ce principe simplifie la déclaration des adversaires par rapport aux politiques de sécurité de CertiCrypt, où l'utilisateur devait spécifier pour chaque variable, si elle pouvait être lue ou bien écrite par l'adversaire.

Dans la preuve nous déclarons deux adversaires A_1 et A_2, leurs signatures et la liste des signatures des oracles que l'adversaire peut appeler. Ensuite pour chaque jeu, nous devons relier les adversaires abstraits à la définition des oracles que nous définirons au fur et à mesure dans les jeux.

Comparaison avec CertiCrypt

L'exemple de la figure 4.2 montre que l'écriture des définitions est plus simple. Dans la preuve Coq de Hashed-ElGamal, le fichier contenait plus de 500 lignes, rien que pour générer la sémantique.

Pour l'instant, certaines étapes de création de la sémantique disparaissent. L'utilisateur ne peut plus créer de nouveau support pour les tirages aléatoires. Mais il est souvent possible de le faire indirectement. Par exemple, si l'utilisateur veut écrire un support pour un groupe d'ordre q, il commencera par tirer un entier n entre $[0..q-1]$ et calculera ensuite g^n (où g est un générateur). Le support des bitstrings est inclus dans EasyCrypt.

Pour le moment, EasyCrypt ne contient pas la notion d'opérateur PPT. Dans les futures versions, nous pourrions ajouter un mot clé PPT lors de la déclaration des nouveaux opérateurs. Et lors de la génération de preuve vers CertiCrypt (voir chapitre suivant), l'utilisateur pourrait prouver que le temps d'exécution et la taille des arguments sont bornés par des polynômes.

4.2.2 Déclaration des jeux

Les jeux comprennent les déclarations des variables globales et des procédures. Les procédures peuvent être de deux types : soit une procédure connue et la déclaration est accompagnée du code, soit un adversaire. Dans le cas des adversaires, nous utiliserons les adversaires définis de manière globale (voir section 4.2.1) en leur donnant le nom des oracles qui ont été définis.

Dans les preuves, les différences entre les jeux sont souvent petites, EasyCrypt propose un mécanisme qui permet de définir un nouveau jeu à partir d'un autre en ne redéfinissant que les fonctions modifiées.

Les fonctions sont représentées dans un langage typé, probabiliste, procédurale et impératif. Par simplicité, le langage ne contient pas de boucles (pour le moment). La syntaxe des programmes est donnée par la grammaire suivante :

$\mathcal{I} ::= \mathcal{T} = \mathcal{E}$	affectation
$\mid$ $\mathsf{if}(\mathcal{E})\ \{\mathcal{C}\}\ \mathsf{else}\ \{\mathcal{C}\}$	test conditionnel
$\mid$ $\mathcal{T} = \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E})$	appel de procédure
$\mathcal{C} ::= \mathsf{nil}$	nop
$\mid$ $\mathcal{I};\ \mathcal{C}$	séquence
$\mathcal{T} ::= \mathcal{V}$	affectation dans une variable
$\mid$ $(\mathcal{V}, \mathcal{V})$	affectation dans une paire variable
$\mid$ $\mathcal{V}[\mathcal{E}]$	mise à jour de map
$\mathcal{E} ::= \mathsf{cnst}$	constante
$\mid$ $\mathcal{V}$	variable
$\mid$ $\mathcal{D}$	distribution aléatoire
$\mid$ $\mathsf{op}\ (\mathsf{list}\ \mathcal{E})$	application de fonction préfixe
$\mid$ $\mathcal{E}\ \mathsf{op}\ \mathcal{E}$	application de fonction infixe
$\mid$ $(\mathcal{E}, \mathcal{E})$	paire
$\mid$ $\mathcal{E}\ ?\ \mathcal{E} : \mathcal{E}$	condition

où $\mathcal{V}$ est l'ensemble des variables, $\mathcal{P}$ est l'ensemble des procédures, et $\mathcal{D}$ est l'ensemble des distributions. $\mathcal{T}$ représente la partie gauche d'une affectation, qui peut être une variable, une paire de variables ou un index de map.

Dans la preuve de Hashed-ElGamal, le premier jeu représente la sécurité IND-CPA, avec les définitions des fonctions de chiffrement et de génération de clé. La fonction H est définie comme un oracle aléatoire, l'adversaire a accès à un autre oracle $H_{\mathcal{A}}$ qui appelle H en sauvegardant les valeurs dans une autre liste. Les deux oracles utilisent les variables globales $\boldsymbol{L}$ et $\boldsymbol{L}_{\mathcal{A}}$. L'utilisation des deux oracles permet de distinguer les appels fait par l'oracle des appels de l'adversaire, nous verrons plus loin comment cette distinction simplifie la preuve. Les adversaires sont ensuite associés aux oracles. La fonction Main représente le jeu principal, contenant toutes les interactions avec l'adversaire (le nom Main n'est pas réservé).

$$H(x) \stackrel{\text{def}}{=} \mathsf{if}\ x \notin \mathsf{dom}(\boldsymbol{L})\ \mathsf{then}\ h \stackrel{\$}{\leftarrow} \{0,1\}^k; \boldsymbol{L}[x] \leftarrow h\ \mathsf{end\ if}; \mathsf{return}\ \boldsymbol{L}[x]$$
$$H_{\mathcal{A}}(x) \stackrel{\text{def}}{=} \boldsymbol{L}_{\mathcal{A}} \leftarrow x :: \boldsymbol{L}_{\mathcal{A}};\ m \leftarrow H(x);\ \mathsf{return}\ m$$

Le jeu IND-CPA est définit de la manière suivante :

```
game INDCPA = {

var L : (group, bitstring{k}) map
var LA : group list

fun Hash(lam:group) : message = {
  var h : message ={0,1}^k;
  if ( ¬in_dom(lam, L) ) { L[lam] = h; };
  return L[lam];
}

fun Hash_adv(lam:group) : message = {
  var m : message;
  LA = lam :: LA;
  m = Hash(lam);
  return m;
}
```

```
abs A1 = A1 {Hash_adv}
abs A2 = A2 {Hash_adv}

fun KG() : key = {
  var x : int = [0..q-1];
  return (x, g^x);
}

fun Enc(pk:pkey, m:message): cipher = {
  var y : int = [0..q-1];
  var hy : message;
  hy = Hash(pk^y);
  return (g^y, hy ^^ m);
}

fun Main() : bool = {
  var sk : skey;
  var pk : pkey;
  var m0, m1 : message;
  var c: cipher;
  var b, b' : bool;

  L = empty_map ();
  LA = [];
  (sk,pk) = KG();
  (m0,m1) = A1(pk);
  b = {0,1};
  c = Enc(pk, b ? m0 : m1);
  b' = A2(pk, c);
  return (b = b');
}
```

Les type de $A1$ et $A2$ sont donnés par la figure 4.2. Dans la première étape de la preuve, la procédure Enc est dépliée à l'interieur de la procédure Main, l'utilisateur n'a pas besoin de redéfinir toutes les fonctions, si une seule est modifiée. Pour cette étape, seule la fonction Main est redéfinie en utilisant la syntaxe suivante :

```
game G1 = INDCPA
  var y' : group
  where  Main = {
    var sk : skey;
    var pk : pkey;
    var m0, m1 : message;
    var c: cipher;
    var b, b' : bool;
    var y:int;
    var hy : message;

    L = empty_map ();
    LA = [];
    (sk,pk) = KG();
    y = [0..q-1];
    y' = pk^y;
    (m0,m1) = A1(pk);
    b = {0,1};
    hy = Hash(y');
    b' = A2(pk, (g^y, hy ^^ (b ? m0 : m1)));
    return (b = b');
  } ;;
```

4.2.3 Jugements Relationnels pRHL

Les jugements pRHL sont de la forme $\mathsf{G}_1 \sim \mathsf{G}_2 : \Psi \Rightarrow \Phi$, où G_1 et G_2 sont des jeux, la pre-condition Ψ et la post-condition Φ sont des relations sur les mémoires des programmes. Les relations sont construites à partir d'expressions relationnelles. Ces expressions sont semblables à celles utilisées dans les programmes mais nous avons besoin de distinguer plusieurs types de variable :

- les variables de programme, et dans ce cas, nous devons encore distinguer les variables du programme de gauche des variables du programme de droite.
- les variables logiques créées par les quantificateurs universelles. Ce choix permet d'éviter d'utiliser les symboles de De Bruijn. EasyCrypt génère des noms de variables uniques, et garantit ainsi qu'il n'y a pas de capture de variable.

Ces variables sont définies de la manière suivante :

$$\mathsf{Var} ::= \mathsf{Left}\ \mathcal{V} \mid \mathsf{Right}\ \mathcal{V} \mid \mathsf{Logic}\ \mathcal{V}$$

Nous notons $x\langle 1\rangle$ la variable ($\mathsf{Left}\ x$) et $x\langle 2\rangle$ la variable ($\mathsf{Right}\ x$). Les relations sont souvent des équivalences sur un ensemble de variables X. Nous utilisons la notation $=_X$ pour la relation $\forall x \in X.\ x\langle 1\rangle = x\langle 2\rangle$.

Les formules logiques sont alors définies par la grammaire suivante :

$$\Psi ::= \mathsf{Exp}_{\mathsf{bool}} \mid \neg\Phi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \Psi \rightarrow \Psi \mid \Psi \leftrightarrow \Psi \mid (\Psi) \mid \forall x.\ \Psi \mid \exists x.\ \Psi$$

avec $\mathsf{Exp}_{\mathsf{bool}}$, des expressions relationnelles, de type bool (égalité, comparaison, ...).

Les formules logiques sont interprétées comme des relations entre deux mémoires de programmes. Par exemple la formule $x\langle 1\rangle + y\langle 2\rangle \leq z\langle 1\rangle$ est interprétée comme la relation

$$R = \{(m_1, m_2) \mid m_1(x) + m_2(y) \leq m_1(z)\}$$

4.2.4 Preuve automatique de Jugements pRHL

L'outil fonctionne de la manière suivante : les calculs de plus faible pré condition (wp) génèrent un ensemble de conditions suffisantes à partir des programmes et de ses spécifications (appelées aussi conditions de vérification), de sorte que si ces conditions sont vraies alors le programme respecte les spécifications. Les conditions sont ensuite envoyées à des outils de preuve automatiques, permettant ainsi de prouver automatiquement des programmes.

L'application de ce principe à la logique pRHL est un challenge significatif pour deux raisons : premièrement la génération de conditions de vérification pour une logique relationnelle est un sujet de recherche ouvert, deuxièmement, il n'y a pas d'application de cette méthodologie pour les programmes probabilistes.

Deux stratégies s'offrent à nous :

- effectuer le wp sur les deux programmes en même temps
- effectuer le wp sur chacun des programmes et les composer en utilisant la *self-composition* [17] : le wp est appliqué sur le premier programme et la pré-condition calculée sert de post-condition pour calculer le wp sur le second programme.

Aucune des deux méthodes ne peut être appliquée complètement à la logique pRHL. Travailler sur les deux programmes en même temps pose problèmes si les programmes n'ont pas la même structure, tandis que la *self-composition* pose des problème avec les tirages aléatoires ou les appels d'adversaire. Pour contourner ces limitations, EasyCrypt propose une alternative qui combine les deux stratégies et se décompose en cinq étapes.

1 : Déplier les fonctions

Les appels de fonction, qui ne sont pas des adversaires, sont éliminés des jeux en dépliant leurs définitions. L'absence d'appel récursif permet de faire en sorte que cette transformation se termine. À la fin de cette étape, nous savons que les seuls appels restants sont les appels aux adversaires.

2 : Transformer en programme déterministe

Les instructions probabilistes sont placées au début. Les programmes se décomposent alors en une séquence d'instructions probabilistes suivie d'un programme déterministe avec appels d'adversaires. Par exemple, le programme suivant :

$$\begin{array}{l} y \leftarrow \mathcal{B}(x); \\ \mathsf{if}\ y = x\ \mathsf{then}\ b \xleftarrow{\$} \{0,1\}\ \mathsf{else}\ b \leftarrow \mathsf{true} \end{array}$$

est transformé en

$$\begin{array}{l} b' \xleftarrow{\$} \{0,1\}; \\ y \leftarrow \mathcal{B}(x); \\ \mathsf{if}\ y = x\ \mathsf{then}\ b \leftarrow b'\ \mathsf{else}\ b \leftarrow \mathsf{true} \end{array}$$

Cette transformation peut échouer quand les distributions sont dépendantes. Par exemple si une distribution $[0..n]$ dépend de la variable n qui est dans la partie déterministe du programme.

Ce point peut être vu comme une version très simplifiée de la transformation *eager sampling* que nous avons vue dans le chapitre 2. A la fin de cette étape, les deux programmes sont de la forme suivante : une séquence de tirages aléatoires, suivie d'un programme déterministe contenant des adversaires probabilistes (nous expliquerons dans la suite comment gérer les adversaires).

3 : Calcul de Wp

Le calcul de wp est alors appliqué sur la partie déterministe des deux programmes. Il y a deux situations, soit les programmes ne contiennent pas d'appel aux adversaires dans ce cas nous utilisons directement la *self-composition*, soit ils contiennent des appels aux adversaires et les deux programmes ont la même structure, et dans ce cas nous utilisons une approche relationnelle.

Pour chaque adversaire, EasyCrypt essaie de générer une spécification relationnelle à partir de la spécification que nous essayons de prouver. Cette spécification est d'abord prouvée sur les oracles à l'aide du wp et de la *self-composition*. Si les

obligations sont vérifiées par des prouveurs SMT, EasyCrypt prouve les spécifications des adversaires à partir des spécifications sur les oracles. Nous expliquerons en détail ce point dans le chapitre suivant.

Si EasyCrypt ne parvient pas à inférer l'invariant, l'utilisateur devra écrire l'invariant sur l'adversaire lui même.

Une fois le calcul de wp effectué sur la partie déterministe les deux programmes obtenus ont la forme suivante :

$$x_1 \stackrel{\$}{\leftarrow} T_1; \ldots x_l \stackrel{\$}{\leftarrow} T_l \sim y_1 \stackrel{\$}{\leftarrow} U_1; \ldots y_n \stackrel{\$}{\leftarrow} U_n : \Psi \Rightarrow \Phi$$

Ils ne contiennent plus que les tirages aléatoires.

4 : Effectuer une bijection entre tirages aléatoires

Le but est de trouver une fonction $f : T_1 \times \cdots \times T_l \to U_1 \times \cdots \times U_n$ qui fait un *mapping* entre les deux ensembles de tirages aléatoires. Cette fonction est ensuite utilisée pour générer la condition $\Psi \Rightarrow_f \Phi$, définie comme

$$\forall m_1\ m_2\ t_1 \ldots t_l\ .\ m_1\ \Psi\ m_2 \implies m_1\{\mathbf{t}/\mathbf{x}\}\ \Phi\ m_2\{f(t_1, \ldots, t_l)/\mathbf{y}\}$$

La validité de $\Psi \Rightarrow_f \Phi$ implique que le jugement pRHL correspondant est correct. En pratique f est un *mapping* 1-1 et la plupart du temps f est simplement l'identité. Cependant dans certains cas d'autres *mapping* doivent être utilisés.

Pour expliquer cette étape, nous illustrerons cette règle, en montrant la transition *optimistic-sampling* :

$$(x \stackrel{\$}{\leftarrow} \{0,1\}^k;\ y \leftarrow x \oplus z) \sim (y \stackrel{\$}{\leftarrow} \{0,1\}^k;\ x \leftarrow y \oplus z) : =_{\{z\}} \Rightarrow =_{\{x,y\}}$$

Une fois que le wp a substitué les deux instructions déterministes, nous obtenons le jugement suivant :

$$x \stackrel{\$}{\leftarrow} \{0,1\}^k \sim y \stackrel{\$}{\leftarrow} \{0,1\}^k : =_{\{z\}} \Rightarrow (x\langle 1\rangle = y\langle 2\rangle \oplus z\langle 2\rangle)$$

Dans ce cas le *mapping* 1-1 utilisé est de la forme $f(x) \Rightarrow\ x \oplus z\langle 1\rangle$, dans ce cas on prouve que f est bijective en passant pas le fait que f est involutive. Nous obtenons la condition suivante :

$$\forall x\ z, \langle 1\rangle = z\langle 2\rangle \to x = f(x) \oplus z\langle 2\rangle$$

Nous obtenons l'obligation de preuve suivante ce qui donne

$$\forall x\ y, z\langle 1\rangle = z\langle 2\rangle \to x = x \oplus z\langle 1\rangle \oplus z\langle 2\rangle$$

qui est ensuite envoyé aux prouveurs SMT.

EasyCrypt vérifie également que f est involutive en vérifiant l'obligation suivante $\forall x, f(x) \circ f(x) = x$. Nous expliquerons ce point d'avantage dans le chapitre suivant.

5 : Prouver automatiquement les conditions

La formule $\Phi \Rightarrow_f \Psi$ est exprimée dans la logique du premier ordre, et les prouveurs SMT permettent de prouver automatiquement leur validité. Au lieu d'appeler les différents outils directement, EasyCrypt génère les conditions dans un format intermédiaire à l'aide de l'outil Why [46]. Les prouveurs SMT sont ensuite appelés par Why, qui supporte de nombreux prouveurs comme Simplify [43] ou alt-ergo [34]. Si les prouveurs SMT échouent, nous pouvons rajouter des axiomes, qui seront ensuite prouvés dans Coq.

La vérification de condition est incomplète (sens logique) et échoue si le jugement pRHL où les jeux appellent les adversaires dans un ordre différent. Mais cette stratégie fonctionne dans les preuves que nous effectuons.

La tactique auto regroupe l'ensemble de ces cinq étapes, elle est utilisée pour prouver l'équivalence entre les jeux IND-CPA et G_1, cette transition est prouvée par les deux lignes suivantes :

```
equiv auto Fact1 : INDCPA.Main ~ G1.Main : {true} ==> ={res}
 inv ={L,LA}
```

La tactique auto est construite à partir des tactiques élémentaires (inline, derandomize, wp, rnd, ...).

4.2.5 Preuve manuelle de jugement pRHL

Dans le cas où la stratégie automatique, ne fonctionne pas, EasyCrypt fournit à l'utilisateur un jeu de tactiques permettant de prouver manuellement les jugements pRHL. Ces tactiques ressemblent à celles de CertiCrypt et permettent d'appliquer les règles de la logique ou des transformation de programmes (inline, derandomize, swap).

Les tactiques peuvent s'appliquer sur les deux programmes en même temps, ou bien sur un seul des programmes. Les tactiques pour la bijection entre les deux tirages aléatoires ou bien le wp pour l'adversaire, ne peuvent s'appliquer que sur les deux programmes simultanément. Le wp pour l'adversaire s'applique sur des programmes qui ont la même structure (même appels aux mêmes adversaires).

Quand les programmes n'ont pas la même structure, EasyCrypt permet d'effectuer certaines transformations locales. Par exemple, l'équivalence ne peut pas être établie directement entre les deux programmes :

$$x \xleftarrow{\$} X; y \leftarrow \mathcal{A}(z) \quad \text{et} \quad y \leftarrow \mathcal{A}(z); x \xleftarrow{\$} X$$

Pour pouvoir exécuter le wp relationnel pour les tirages aléatoires ou bien celui des appels aux adversaires, la tactique swap permet de réordonner les instructions indépendantes.

Liste des tactiques de EasyCrypt

Nous allons expliquer toutes les tactiques présentes dans EasyCrypt. La représentation des preuves sera expliquée en même temps que la génération de preuve en Coq dans le chapitre suivant.

Pour savoir sur quel programme sont exécutées les tactiques, nous ajoutons l'information side qui décrit les trois possibilités Left | Right | Both.

- asgn : Effectue un calcul de wp sur les parties déterministes situées à la fin des deux programmes.
- rnd info : Les deux programmes se terminent par un tirage aléatoire, la tactique rnd effectue le calcul de plus faible pré condition. info contient les informations sur la bijection, nous détaillerons se point dans le chapitre suivant.
- call info : Effectue un calcul de plus faible pré condition si les deux dernières instructions sont des appels de fonction. info contient une preuve d'équivalence sur le corps des fonctions.
- inline side info : par défaut, déplie les définitions de toutes les fonctions. Plusieurs options sont disponibles selon la valeur de info :
  - last : déplie le dernier appel de fonction
  - at (list int) : donne la liste d'indices des fonctions à déplier
  - f : déplie toute les occurrences de la fonction f
- swap info : permet de déplacer des portions de programme selon les indications données dans info
  - swap s l d déplace le bloc de taille l commençant par la ligne s et d indique le décalage. si d est négatif, le bloc est déplacé vers le début du programme.
  - pop n remonte la dernière instruction de n ligne(s), se traduit en swap $|c|$ $1-n$ où c est le programme sur lequel on applique swap
  - push n déplace la première instruction de n ligne(s) se traduit en swap 0 1 n où c est le programme sur lequel on applique swap

  Les tactiques pop et push sont écrites à partir de la tactique swap
- trivial applique asgn et appelle les prouveurs SMT.
- wp : effectue un wp similaire à asgn mais traite les instructions conditionnelles.
- derandomize : remonte les tirages aléatoires en haut du programme, séparant le programme en deux partie, une qui ne contient que des tirage aléatoire et une autre qui est déterministe.
- case e : transforme un but de la forme $c_1 \sim c_2 : P \Rightarrow Q$ en deux sous buts $c_1 \sim c_2 : P \wedge e \Rightarrow Q$ et $c_1 \sim c_2 : P \wedge \neg e \Rightarrow Q$.
- app n_1 n_2 R : transforme un but de la forme

  $$c_{h1}\texttt{++}c_{t1} \sim c_{h2}\texttt{++}c_{t2} : P \Rightarrow Q$$

  en deux sous buts

  $$c_{h1} \sim c_{h2} : P \Rightarrow R \quad \text{et} \quad c_{t1} \sim c_{t2} : R \Rightarrow Q$$

  avec $|c_{h1}| = n_1$ et $|c_{h2}| = n_2$.
- if : transforme un but de la forme

  $$\mathsf{if}\ e\ \mathsf{then}\ c_t\ \mathsf{else}\ c_f :: c_1 \sim c_2 : P \Rightarrow Q$$

  en deux sous buts

  $$c_t :: c_1 \sim c_2 : P \wedge e \Rightarrow Q \quad \text{et} \quad c_f :: c_1 \sim c_2 : P \wedge \neg e \Rightarrow Q$$

D'autres variantes de cette tactique sont expliquées en détails dans le chapitre suivant.
- auto inv : effectue la stratégie expliquée dans la section 4.2.4, en prenant un invariant.

A la manière de Coq, EasyCrypt permet de composer des tactiques,
- try tactic1 : essaie d'appliquer la tactique tactic1
- tactic1 ;tactic2 ; applique la tactique tactic1 et applique ensuite la tactique tactic2 aux sous buts
- repeat tactic1 : répète la tactique, et termine quand elle échoue
- do n t : répète la tactique t n fois
- [tactic1 | tactic2 | ...] : applique tactic1 au premier sous but et tactic1 au second ...
- tactic1 || tactic2 : essaie la tactique tactic1, si elle echoue, essaie la tactique tactic2

Pour illustrer le jeu de tactiques, nous reprouverons la transition entre le jeu IND-CPA et G_1 fait par la tactique auto en décomposant le script de preuve :

```
equiv Fact1 : INDCPA.Main ~ G1.Main : {true} ==> ={res}
 inline KG, Enc; derandomize;
 auto inv ={L,LA};
 pop⟨2⟩ 1; repeat rnd; trivial;;
save;;
```

4.2.6 Raisonner sur les probabilités

EasyCrypt permet de prouver des propriétés sur les probabilités, ce qui est essentiel dans les preuves de sécurité. En effet, l'objectif final des preuves est de montrer une borne entre la probabilité qu'un adversaire casse le jeu initial et la probabilité qu'un ou plusieurs adversaires cassent les hypothèses de sécurité.

Dans EasyCrypt, les preuves de probabilité peuvent êtres déduites des preuves d'équivalence, en utilisant les règle ci dessous :

$$\frac{m_1 \; \Psi \; m_2 \qquad \mathsf{G}_1 \sim \mathsf{G}_2 : \Psi \Rightarrow \Phi \qquad \Phi \rightarrow (A\langle 1\rangle \rightarrow B\langle 2\rangle)}{\Pr\left[\mathsf{G}_1, m_1 : A\right] \leq \Pr\left[\mathsf{G}_2, m_2 : B\right]} \; [\text{PrLe}]$$

$$\frac{m_1 \; \Psi \; m_2 \qquad \mathsf{G}_1 \sim \mathsf{G}_2 : \Psi \Rightarrow \Phi \qquad \Phi \rightarrow (A\langle 1\rangle \leftrightarrow B\langle 2\rangle)}{\Pr\left[\mathsf{G}_1, m_1 : A\right] = \Pr\left[\mathsf{G}_2, m_2 : B\right]} \; [\text{PrEq}]$$

La seconde règle se déduit de la première. Ces règle peuvent s'appliquer en donnant le nom du lemme qui prouve l'équivalence, les conditions $m_1 \; \Psi \; m_2$ et $\Phi \rightarrow (A\langle 1\rangle \leftrightarrow B\langle 2\rangle)$ sont envoyées au prouveur SMT.

La première transition de Hashed ElGamal se termine en prouvant que la probabilité d'avoir res est la même dans les deux jeux. Cette propriété se prouve à partir de Fact1 montrant que res est indistinguable dans les deux jeux.

```
proba Pr1 : INDCPA.Main[res] = G1.Main[res]
 using Fact1
```

4.2.7 Raisonnement sur les *Failure Events*

Les preuves par jeux comportent souvent des transitions qui portent sur deux jeux G_1 et G_2 qui ont un comportement identique jusqu'au moment où un évènement F (appelé *failure event* et qui est souvent désigné par la variable bad), se produit. Cette transition est justifiée par le *Fundamental Lemma* [80, 26], qui permet de borner la différence entre la probabilité d'un évènement A dans le jeu G_1 et la probabilité d'un évènement B dans le jeu G_2 par la probabilité de F. La plupart du temps, nous utilisons une version syntaxique de cette propriété, l'évènement F est représenté par une expression booléenne dans le code des jeux. EasyCrypt propose une version plus générale de ce lemme qui utilise la logique relationnelle. Cette méthode est une alternative à la version du Chapitre 2.

Lemme 4.1 (Fundamental Lemma). *Prenons G_1, G_2 deux jeux et A, B, et F des évènements tels que*

$$\mathsf{G}_1 \sim \mathsf{G}_2 : \Psi \Rightarrow (F\langle 1\rangle \leftrightarrow F\langle 2\rangle) \wedge (\neg F\langle 1\rangle \rightarrow (A\langle 1\rangle \leftrightarrow B\langle 2\rangle))$$

Alors, si $m_1\ \Psi\ m_2$,

1. $\Pr[\mathsf{G}_1 : A \wedge \neg F] = \Pr[G_2 : B \wedge \neg F]$,
2. $|\Pr[\mathsf{G}_1 : A] - \Pr[\mathsf{G}_2 : B]| \leq \Pr[\mathsf{G}_1 : F] = \Pr[\mathsf{G}_2 : F]$

Preuve.

$$\begin{aligned}
&|\Pr[G_1 : A] - \Pr[G_2 : B]| \\
&\quad = |\Pr[G_1 : A \wedge F] + \Pr[G_1 : A \wedge \neg F] - \Pr[G_2 : B \wedge F] - \Pr[G_2 : B \wedge \neg F]| \\
&\quad = |\Pr[G_1 : A \wedge F] - \Pr[G_2 : B \wedge F]| \\
&\quad \leq \Pr[G_1 : A \wedge F] \\
&\quad \leq \Pr[G_1 : F] = \Pr[G_2 : F]
\end{aligned}$$

Nous avons $\Pr[G_1 : F] = \Pr[G_2 : F]$ par le jugement pRHL.

Les hypothèses du lemme se prouvent comme un jugement pRHL classique, c'est à dire à l'aide de la stratégie ci dessus et des prouveurs SMT. La clé de la preuve est de trouver la bonne spécification pour l'adversaire. Pour chaque appel d'adversaire $x \leftarrow \mathcal{A}(\mathbf{e})$, EasyCrypt infère une relation Θ et vérifie le jugement

$$\begin{aligned}\models \mathcal{A} \sim \mathcal{A} : (F\langle 1\rangle \leftrightarrow F\langle 2\rangle) \wedge (\neg F\langle 1\rangle \rightarrow =_{\mathsf{params}(A)} \wedge \Theta) \Rightarrow \\ (F\langle 1\rangle \leftrightarrow F\langle 2\rangle) \wedge \left(\neg F\langle 1\rangle \rightarrow =_{\{\mathsf{res}\}} \wedge \Theta\right)\end{aligned}$$

où $\mathsf{params}(\mathcal{A})$ représente l'ensemble des paramètres formels de $\mathcal{A}$. EasyCrypt infère également des spécifications pour tous les oracles que l'adversaire peut appeler. Pour tout oracle O nous prouvons

$$\begin{aligned}\models O \sim O : (\neg F\langle 1\rangle \wedge \neg F\langle 2\rangle \wedge =_{\mathsf{params}(O)} \wedge \Theta) \Rightarrow \\ (F\langle 1\rangle \leftrightarrow F\langle 2\rangle) \wedge \left(\neg F\langle 1\rangle \rightarrow =_{\{\mathsf{res}\}} \wedge \Theta\right)\end{aligned}$$

Nous avons besoin d'une condition supplémentaire sur les oracles, nous devons vérifier que si F est vrai alors il reste vrai. Nous prouvons cette propriété en utilisant

une logique de Hoare, ce coup ci non relationnel, et la vérifions par un calcul de plus faible pré-condition, et pour tous les oracles O appelés par l'adversaire que

$$\{F\}O\{F\}$$

Nous prouvons en Coq que si tous les oracles que peut appeller l'adversaire respectent cette condition alors l'adversaire respecte aussi cette condition.

$$\{F\}\mathcal{A}\{F\}$$

Ces heuristiques suffisent dans la plupart des cas, l'utilisateur peut choisir de prouver lui même les spécifications pour un ou plusieurs adversaires ou oracles, si besoin, en laissant si il le souhaite l'outil inférer le reste.

Dans la suite de la preuve, nous remplaçons l'appel à l'oracle $h_y \leftarrow \mathsf{Hash}(y')$ par un tirage aléatoire $h \xleftarrow{\$} \{0,1\}^k$.

```
game G2 = G1
  where Main = {
    var sk : secret_key;
    var pk : public_key;
    var m0, m1 : message;
    var c: cipher;
    var b, b' : bool;
    var y:int;
    var h : message;

    L = empty_map ();
    LA = [];
    (sk,pk) = KG();
    y = [0..q-1];
    y' = pk^y;
    (m0,m1) = A1(pk);
    b = {0,1};
    h ={0,1}^k;
    b' = A2(pk, (g ^ y, h ^^ (b ? m0 : m1)));
    return (b = b');
  }
```

Les jeux G_1 et G_2 ne sont pas équivalents. Si A_1 demande la valeur y' à l'oracle Hash, la valeur ne sera pas la même dans les deux jeux. Cette étape est identique à celle du chapitre 2 mais contient 100 fois moins de ligne. Nous utilisons les *Failure Event* pour effectuer cette transition.

La différence entre ces deux probabilités est bornée par la probabilité que $\hat{y}$ soit dans la liste $\boldsymbol{L}_{\mathcal{A}}$ dans le jeu G_2. Pour utiliser le *Fondamental Lemma*, l'utilisateur montre que les deux jeux sont équivalents jusqu'au moment où l'évènement $\hat{y} \in \boldsymbol{L}_{\mathcal{A}}$ est vrai avec l'invariant suivant :

$$(\hat{y} \in \boldsymbol{L}_{\mathcal{A}})\langle 1\rangle \leftrightarrow (\hat{y} \in \boldsymbol{L}_{\mathcal{A}})\langle 2\rangle \wedge \big((\hat{y} \notin \boldsymbol{L}_{\mathcal{A}})\langle 1\rangle \rightarrow =_{\{\mathsf{res}\}}\big)$$

Dans la suite nous expliquons la transition entre le jeu G_1 et G_2

```
equiv auto G1_G2_A1 : G1.A1 ~ G2.A1 :
 ={y', LA, L} ∧ ({(in_dom(y',L))⟨1⟩} ⇒ {(in(y',LA))⟨1⟩});;

equiv auto Fact2 : G1.Main ~ G2.Main :
 {true} ⟹
```

```
{ (in(y',LA))⟨1⟩ = (in(y',LA))⟨2⟩ } ∧
({(¬in(y',LA))⟨1⟩} ⇒ ={res})
auto inv upto {in(y',LA)}
with ={y',LA} ∧
 ∀ (w:group).
   {¬(w = y'⟨1⟩)} ⇒ {L⟨1⟩[w] = L⟨2⟩[w]} ∧ {in_dom(w,L⟨1⟩) = in_dom(w,L⟨2⟩)};;
rnd; wp; rnd; call G1_G2_A1;
wp; rnd; wp; rnd; trivial;;
```

Les preuves d'équivalence pour les adversaires A_1 et A_2 ne sont pas faites de la même manière. Pour A_1 les deux jeux sont équivalents, entre A_1 et A_2 nous modifions le jeu, et dans la preuve pour A_2 nous prouvons l'équivalence en nous servant du *Fundamental Lemma* 4.1.

Étant donné que les tactiques s'appliquent sur la fin du programme, nous commençons par prouver l'équivalence sur les adversaires A_2. Les deux programmes sont identiques jusqu'au moment où y', est appelé par l'adversaire. L'évènement *bad* est modélisé par l'expression booléenne $\hat{y} \in \boldsymbol{L}_{\mathcal{A}}$. L'invariant sur l'adversaire, signifie que les deux listes sont équivalentes pour toute valeur différente de $\hat{y}$. Entre les deux adversaires, nous utilisons les différentes tactiques pour appeler les wp sur les tirages aléatoires. Pour finir nous prouvons que les deux appels des adversaires sont équivalents sous l'invariant que les deux listes globales et $\hat{y}$ sont équivalents et que si y est dans le domaine de L cela implique que y est dans $\boldsymbol{L}_{\mathcal{A}}$.

Une fois le jugement pRHL prouvé, nous dérivons automatiquement les propriétés du *Fundamental Lemma* :

```
proba Pr2 : | G1.Main[res] − G2.Main[res] | ≤ G2.Main[in(y',LA)]
 using Fact2;;
```

Cette étape est une variante du *Fundamental Lemma*; la logique permet de supprimer les transformations de code qu'effectuerait CertiCrypt, et d'appliquer le lemme directement. En effet CertiCrypt utilise un critère syntaxique du lemme, c'est à dire que les deux jeux doivent être identiques jusqu'au moment où l'événement bad est mis à vrai. Pour être identique nous devions faire des transformations de programme. Alors que dans EasyCrypt le lemme peut être appelé directement.

4.2.8 Équivalences algébriques

EasyCrypt permet de raisonner sur des équivalences entre deux tirages aléatoires où nous devons appliquer une bijection entre deux domaines. Par exemple dans la preuve de ElGamal nous utilisons le fait que dans un groupe cyclique, la multiplication par un élément aléatoire agit comme une fonction *one-time pad* (expliqué dans le chapitre 2) :

$$x \xleftarrow{\$} \mathbb{Z}_q; \alpha \leftarrow g^x \times \beta \sim y \xleftarrow{\$} \mathbb{Z}_q; \alpha \leftarrow g^y : \mathsf{true} \Rightarrow =_\alpha$$

Pour prouver cette équivalence, nous montrons que les images des deux distributions sont égales en appliquant une bijection, nous expliquerons ce point en détail dans le chapitre suivant.

Dans la preuve de Hashed ElGamal, nous utilisons le fait que la fonction $\oplus$ agit comme *one-time pad* :

```
game G3 = G2
  where Main = {
    var sk : skey;
    var pk : pkey;
    var m0, m1 : message;
    var b, b' : bool;
    var y:int;
    var h : message;

    L = empty_map ();
    LA = [];
    (sk,pk) = KG();
    y = [0..q-1];
    y' = pk^y;
    (m0,m1) = A1(pk);
    h ={0,1}^k;
    b' = A2(pk, (g ^ y, h));
    b = {0,1};
    return (b = b');
  }

equiv Fact3 : G2.Main ~ G3.Main : {true} ==> ={res,y',LA};;
 pop ⟨2⟩ 2;; auto;;
 rnd (h^^(b ? m0 : m1));;
 rnd; auto; rnd;;
 auto inv ={LA,L};;
save;;
```

La transition du jeu G_2 au jeu G_3 effectue la transformation *optimistic sampling.* Elle transforme les instructions

$$h \xleftarrow{\$} \{0,1\}^k; \gamma \leftarrow h \oplus m_b \quad \text{en} \quad \gamma \xleftarrow{\$} \{0,1\}^k; h \leftarrow \gamma \oplus m_b.$$

Dans EasyCrypt, la première étape consiste à déplacer l'instruction $b \xleftarrow{\$} \{0,1\}$ dans le jeu G_2, en la remontant de deux lignes. Cette transformation est effectuée par la tactique *pop* $\langle 2 \rangle$ 2.

Une fois le calcul de wp effectué en utilisant la tactique auto, nous prouvons que les distributions sont équivalentes, car l'instruction $\oplus$ agit comme un *one-time pad*. Les deux distributions sont équivalentes, car il existe une bijection entre les deux distributions. EasyCrypt propose un mécanisme, qui permet de donner cette bijection et l'outil vérifie ensuite à l'aide des prouveurs SMT, que la transition est correcte.

Dans notre cas, la transition est *optimistic sampling*, la bijection est définie par fun $h \ \Rightarrow h \oplus (b \ ? \ m_0 : m_1)$. Nous expliquerons comment EasyCrypt vérifie la bijection dans le chapitre suivant.

Nous terminons la preuve en appelant deux fois le wp sur les deux tirages aléatoires restant, et en appelant auto avec l'invariant sur les adversaires.

Le but de cette étape est de supprimer l'instruction $h \leftarrow \gamma \oplus m_b$ qui n'est plus utilisée pour calculer res, il n'y a plus de dépendance entre b et b'. Et la logique permet de déduire ensuite que la probabilité de res est $1/2$.

La preuve se poursuit en dérivant des propriétés sur les probabilités à partir des équivalences précédemment prouvées. $\mathsf{Pr3}_1$ et $\mathsf{Pr3}_2$ utilisent l'équivalence prouvée dans Fact3.

```
proba Pr3_1 : G2.Main[res] = G3.Main[res]
 using Fact3

proba Pr3_2 : G2.Main[in(y',LA)] = G3.Main[in(y',LA)]
 using Fact3
```

4.2.9 Calcul de Probabilités

EasyCrypt permet de calculer des bornes à l'aide d'un mécanisme simple. Par exemple, la probabilité qu'une expression quelconque soit égale à une valeur tirée uniformément dans un ensemble T est égale à $1/|T|$, ou bien la probabilité d'appartenir à une liste de longueur n, est au plus $n/|T|$. Il faut faire attention, si le domaine est l'intervalle $[0..k]$ nous devons ajouter des propriétés supplémentaires sur le fait que pour toutes valeurs x tirées uniformement dans $[0..k]$ respectent la condition $(0 < x < l)$.

Le lemme $\mathsf{Pr3}_3$ montre que la probabilité de res dans le programme terminant par les instructions suivantes : $b \xleftarrow{\$} \{0,1\}$; return $b = b'$ et de $\frac{1}{2}$.

```
proba Pr3_3 : G3.Main[res] = 1%r / 2%r
 compute;;
```

La preuve se conclut en construisant l'adversaire $\mathcal{B}$ qui utilise $\mathcal{A}$ comme sous fonction. Les jeux LCDH et G_3 sont prouvés équivalents. La conclusion utilise tout les lemmes précédemment prouvés.

```
game LCDH = {
  var L : (group, bitstring{k}) map
  var LA : group list

  fun Hash(lam:group) : message = {
    var h : message ={0,1}^k;
    if (¬ in_dom(lam, L) ) { L[lam] = h; };
    return L[lam];
  }

  fun Hash_adv(lam:group) : message = {
    var m : message;
    LA = lam :: LA;
    m = Hash(lam);
    return m;
  }

  abs A1 = A1 {Hash_adv}
  abs A2 = A2 {Hash_adv}

  fun B(gx:group,gy:group) : group list = {
    var pk : public_key;
    var m0, m1 : message;
    var h : message;
    var b' : bool;

    L = empty_map ();
    LA = [];
    pk = gx;
    (m0,m1) = A1(pk);
    h ={0,1}^k;
    b' = A2(pk, (gy, h));
```

```
    return LA;
  }

  fun Main() : bool = {
   var x, y : int;
   var l : group list;
   x = [0..q-1];
   y = [0..q-1];
   l = B(g^x, g^y);
   return (in(g^(x*y), l));
  }
}

equiv auto Fact4 : G3.Main ~ LCDH.Main :
 {true} ==> {(in(y',LA))<1> = res<2> }
 inv ={L,LA}

proba Pr4 : G3.Main[in(y',LA)] = LCDH.Main[res] using Fact4

proba Conclusion : | INDCPA.Main[res] - (1%r/2%r) | <= LCDH.Main[res]
```

La conclusion est prouvée automatiquement en utilisant les prouveurs SMT.

Au final, considérant toutes les étapes, l'avantage de l'adversaire $\mathcal{A}$ peut être borné par la probabilité que $\mathcal{B}$ résolve LCDH. Le script de preuve est de 250 lignes, soit 5 fois plus court que la preuve du Chapitre 1, et est également plus simple et plus proche de la preuve papier.

4.3 Application : Cramer-Shoup

Le système Cramer-Shoup est une primitive de chiffrement à clé publique, basée sur le chiffrement ElGamal. Ce système est le premier chiffrement asymétrique dont la sécurité a été prouvée IND-CCA2 dans le modèle standard. La taille des messages est juste doublée par rapport à ceux de ElGamal.

Étant donnée une famille de groupe cyclique $\mathcal{G}$ d'ordre q, et une fonction de hashage $\{H_k : \mathcal{G}^3 \to \mathbb{Z}_q\}_{k \in K}$ paramétrée par un indice $k \in K$, faisant le lien entre un triplet d'éléments du groupe et un entier dans $\mathbb{Z}_q$. Les fonctions de génération de clé, de chiffrement et de déchiffrement sont définies par :

$$
\begin{array}{rcl}
\mathcal{KG}() & \stackrel{\text{def}}{=} & g, \hat{g} \stackrel{\$}{\leftarrow} \mathcal{G} \setminus \{1\}; \\
& & x_1, x_2, y_1, y_2, z_1, z_2 \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ k \stackrel{\$}{\leftarrow} K; \\
& & e \leftarrow g^{x_1} \hat{g}^{x_2}; \\
& & f \leftarrow g^{y_1} \hat{g}^{y_2}; \\
& & h \leftarrow g^{z_1} \hat{g}^{z_2}; \\
& & pk \leftarrow (k, g, \hat{g}, e, f, h); \\
& & sk \leftarrow (k, g, \hat{g}, x_1, x_2, y_1, y_2, z_1, z_2); \\
& & \mathsf{return}\ (pk, sk)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Enc}((k, g, \hat{g}, e, f, h), m) & \stackrel{\text{def}}{=} & u \stackrel{\$}{\leftarrow} \mathbb{Z}_q;\ a \leftarrow g^u \\
& & \hat{a} \leftarrow \hat{g}^u;\ c \leftarrow h^u \cdot m; \\
& & v \leftarrow H_k(a, \hat{a}, c);\ d \leftarrow e^u \cdot f^{uv}; \\
& & \mathsf{return}\ (a, \hat{a}, c, d)
\end{array}
$$

$$\begin{array}{l}\mathsf{Dec}((k,g,\hat{g},x_1,x_2,y_1,y_2,z_1,z_2),(a,\hat{a},c,d)) \stackrel{\mathrm{def}}{=} \\ v \leftarrow H_k(a,\hat{a},c); \\ \text{if } d = a^{x_1+vy_1}\cdot \hat{a}^{x_2+vy_2} \text{ then} \\ \quad \text{return } c/(a^{z_1}\cdot\hat{a}^{z_2}) \\ \text{else return } \bot\end{array}$$

Nous prouvons que le chiffrement de Cramer-Shoup est sûr sous la notion de sécurité IND-CCA2 dans le modèle standard, avec l'hypothèse que le problème DDH est difficile pour la famille de groupes considérée et l'hypothèse que la fonction de hachage H est *target collision-resistant* (i.e., *universal one-way*).

Définition 4.2 (Target Collision-Resistance). *Pour une famille de fonctions de hashage $\{H_k : A \to B\}_{k\in K}$ indicée par $k \in K$. L'avantage de l'adversaire $\mathcal{C}$ de casser la propriété de* collision-resistance *de H est définie par*

$$\mathbf{Adv}^{\mathcal{C}}_{\mathit{TCR}} \stackrel{def}{=} \Pr[\mathit{TCR} : H_k(x) = H_k(y) \land x \neq y]$$

où l'expérience TCR est définie par le jeu suivant :

$$\textbf{Game } \mathsf{TCR} : x \leftarrow \mathcal{C}_1(\,);\; k \stackrel{\$}{\leftarrow} K;\; y \leftarrow \mathcal{C}_2(k)$$

L'adversaire choisit une valeur x, nous tirons aléatoirement la clé k qui est l'indice de la fonction de hachage H_k. Ensuite nous demandons à l'adversaire de trouver une autre valeur y telle que $H_k(x) = H_k(y)$ avec $x \neq y$.

Définition 4.3 (Avantage CCA). *Pour un algorithme de chiffrement asymétrique $(\mathcal{KG}, \mathsf{Enc}, \mathsf{Dec})$, l'avantage* CCA *d'un adversaire $\mathcal{A}$ limité à q_{Dec} appels à l'oracle de déchiffrement sous IND-CCA2 est défini par*

$$\mathbf{Adv}^{\mathcal{A}}_{\mathit{CCA}}(q_{\mathsf{Dec}}) \stackrel{def}{=} \left|\Pr[\mathit{IND\text{-}CCA2} : b = b'] - \frac{1}{2}\right|$$

où le jeu IND-CCA2 est défini par :

Game *IND-CCA2* :	**Oracle** $\mathsf{Dec}_{\mathcal{A}}(\gamma)$:
$(pk, \boldsymbol{sk}) \leftarrow \mathcal{KG}(\,)$;	if $\lvert\boldsymbol{L}_{\mathsf{Dec}}\rvert < q_{\mathsf{Dec}} \land \neg(\boldsymbol{\gamma}^*_{\mathsf{def}} \land \gamma = \boldsymbol{\gamma}^*)$ then
$(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;	$\quad\boldsymbol{L}_{\mathsf{Dec}} \leftarrow \gamma :: \boldsymbol{L}_{\mathsf{Dec}}$;
$b \stackrel{\$}{\leftarrow} \{0,1\}$;	$\quad$ *return* $\mathsf{Dec}(\boldsymbol{sk}, \gamma)$
$\boldsymbol{\gamma}^* \leftarrow \mathsf{Enc}(pk, m_b)$; $\boldsymbol{\gamma}^*_{\mathsf{def}} \leftarrow \mathsf{true}$;	else *return* $\bot$
$b' \leftarrow \mathcal{A}_2(\boldsymbol{\gamma}^*)$;	
return $(b = b')$	

Le jeu IND-CCA2 est le même que le jeu IND-CPA, mais nous autorisons l'adversaire à appeler l'oracle de déchiffrement. Dans la première phase $(\mathcal{A}_1)$ l'adversaire peut appeler l'oracle de déchiffrement avec n'importe quel message, même γ^*. Par contre dans la seconde phase $(\mathcal{A}_2)$ l'adversaire peut toujours appeler l'oracle de déchiffrement mais il ne peut pas déchiffrer γ^*.

Théorème 4.4 (Sécurité de Cramer-Shoup). *Prenons $\mathcal{A}$ un adversaire contre la sécurité IND-CCA2 de Cramer-Shoup limité à q_{Dec} appels à l'oracle de déchiffrement. Alors il existe un algorithme $\mathcal{B}$ contre le problème DDH et un adversaire $\mathcal{C}$ contre l'hypothèse* target collision-resistance *de la fonction de hashage H tel que,*

$$\mathbf{Adv}^{\mathcal{A}}_{CCA}(q_{\mathsf{Dec}}) \leq \mathbf{Adv}^{\mathcal{B}}_{DDH} + \mathbf{Adv}^{\mathcal{C}}_{TCR} + \frac{q^4_{\mathsf{Dec}}}{q^4} + \frac{q_{\mathsf{Dec}} + 2}{q}$$

avec

$$\mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} = |\Pr[\mathsf{DDH}_0 : b = b'] - \Pr[\mathsf{DDH}_1 : b = b']| \quad (4.1)$$

Notre preuve est très proche de celle présentée dans [54] ; nous donnerons une description de haut niveau dans ce chapitre.

Le jeu G_1 est obtenu à partir du jeu IND-CCA2 défini pour Cramer-Shoup, en dépliant les définitions de génération de clé et d'algorithme de chiffrement, en propageant les assignements de variables et en remplaçant les expressions par des expressions équivalentes. Nous observons que toutes les conditions de vérification générées par les différentes transformations sont correctes et peuvent être vérifiées automatiquement par des prouveurs SMT. Ce point dépasse le souhait de Halevi [54] qui suggère que cette transformation soit décomposée en trois étapes alors que dans EasyCrypt cette transition est prouvée automatiquement en une seule étape.

Game G_1 :	**Oracle** $\mathsf{Dec}(a, \hat{a}, c, d)$:
$g, \hat{g} \xleftarrow{\$} \mathcal{G} \setminus \{1\}; x_1, x_2, y_1, y_2, z_1, z_2 \xleftarrow{\$} \mathbb{Z}_q;$	if $\lvert \boldsymbol{L}_{\mathsf{Dec}} \rvert < q_{\mathsf{Dec}} \wedge \neg(\boldsymbol{\gamma}^*_{\mathsf{def}} \wedge (a, \hat{a}, c, d) = \boldsymbol{\gamma}^*)$
$k \xleftarrow{\$} K;$	then
$e \leftarrow g^{x_1}\hat{g}^{x_2};\ f \leftarrow g^{y_1}\hat{g}^{y_2};\ h \leftarrow g^{z_1}\hat{g}^{z_2};$	$\boldsymbol{L}_{\mathsf{Dec}} \leftarrow \gamma :: \boldsymbol{L}_{\mathsf{Dec}};$
$(m_0, m_1) \leftarrow \mathcal{A}_1(k, g, \hat{g}, e, f, h);\ b \xleftarrow{\$} \{0, 1\};$	$v \leftarrow H_k(a, \hat{a}, c);$
$u \xleftarrow{\$} \mathbb{Z}_q;\ a \leftarrow g^u;\ \hat{a} \leftarrow \hat{g}^u;$	if $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$ then
$c \leftarrow a^{z_1} \cdot \hat{a}^{z_2} \cdot m_b;$	return $c/(a^{z_1} \cdot \hat{a}^{z_2})$
$v \leftarrow H_k(a, \hat{a}, c);\ d \leftarrow a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2};$	else return $\perp$
$\boldsymbol{\gamma}^* \leftarrow (a, \hat{a}, c, d);\ \boldsymbol{\gamma}^*_{\mathsf{def}} \leftarrow \mathsf{true};$	else return $\perp$
$b' \leftarrow \mathcal{A}_2(\boldsymbol{\gamma}^*);$ return $(b = b')$	

$\mathsf{IND\text{-}CCA2} \sim \mathsf{G}_1 : \mathsf{true} \Rightarrow =_{\{\mathsf{res}\}}$

$$\Pr[\mathsf{IND\text{-}CCA2} : b = b'] = \Pr[\mathsf{G}_1 : b = b'] \quad (4.2)$$

Nous construisons l'adversaire $\mathcal{B}$ pour DDH tel que les distributions soient identiques sur la valeur de $(b = b')$ pour les jeux DDH_0 et G_1.

Game $\fbox{DDH}_0$ $\boxed{\mathsf{DDH}_1}$:	**Oracle** $\mathsf{Dec}(a, \hat{a}, c, d)$:

$g \xleftarrow{\$} \mathcal{G} \setminus \{1\}; x \xleftarrow{\$} \mathbb{Z}_q^*;\ y \xleftarrow{\$} \mathbb{Z}_q;$
$\fbox{$z \leftarrow xy$}$ $\boxed{z \xleftarrow{\$} \mathbb{Z}_q}$;
return $\mathcal{B}(g, g^x, g^y, g^z)$
Adversary $\mathcal{B}(g, \hat{g}, a, \hat{a})$:
$x_1, x_2, y_1, y_2, z_1, z_2 \xleftarrow{\$} \mathbb{Z}_q;\ k \xleftarrow{\$} K;$
$e \leftarrow g^{x_1}\hat{g}^{x_2};\ f \leftarrow g^{y_1}\hat{g}^{y_2};\ h \leftarrow g^{z_1}\hat{g}^{z_2};$
$(m_0, m_1) \leftarrow \mathcal{A}_1(k, g, \hat{g}, e, f, h);\ b \xleftarrow{\$} \{0, 1\};$
$c \leftarrow a^{z_1} \cdot \hat{a}^{z_2} \cdot m_b;$
$v \leftarrow H_k(a, \hat{a}, c);\ d \leftarrow a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2};$
$\boldsymbol{\gamma}^* \leftarrow (a, \hat{a}, c, d);\ \boldsymbol{\gamma}^*_{\mathsf{def}} \leftarrow \mathsf{true};$
$b' \leftarrow \mathcal{A}_2(\boldsymbol{\gamma}^*);$ return $(b = b')$
	if $\lvert \boldsymbol{L}_{\mathsf{Dec}} \rvert < q_{\mathsf{Dec}} \wedge \neg(\boldsymbol{\gamma}^*_{\mathsf{def}} \wedge (a, \hat{a}, c, d) = \boldsymbol{\gamma}^*)$
	then
	  $\boldsymbol{L}_{\mathsf{Dec}} \leftarrow \gamma :: \boldsymbol{L}_{\mathsf{Dec}};$
	  $v \leftarrow H_k(a, \hat{a}, c);$
	  if $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$ then
	    return $c/(a^{z_1} \cdot \hat{a}^{z_2})$
	  else return $\perp$
	else return $\perp$

$\mathsf{G}_1 \sim \mathsf{DDH}_0 : \mathsf{true} \Rightarrow =_{\{\mathsf{res}\}}$

En dépliant les procédures, nous obtenons :

$$\Pr[\mathsf{G}_1 : b = b'] = \Pr[\mathsf{DDH}_0 : b = b'] \tag{4.3}$$

Game G_2 :	**Oracle** $\mathsf{Dec}(a, \hat{a}, c, d)$:

$g \xleftarrow{\$} \mathcal{G} \setminus \{1\};\ w \xleftarrow{\$} \mathbb{Z}_q^*;\ \hat{g} \leftarrow g^w;$
$u, u' \xleftarrow{\$} \mathbb{Z}_q;\ a \leftarrow g^u;\ \hat{a} \leftarrow \hat{g}^{u'};$
$x_1, x_2, y_1, y_2, z_1, z_2 \xleftarrow{\$} \mathbb{Z}_q;\ k \xleftarrow{\$} K;$
$e \leftarrow g^{x_1}\hat{g}^{x_2};\ f \leftarrow g^{y_1}\hat{g}^{y_2};\ h \leftarrow g^{z_1}\hat{g}^{z_2};$
$(m_0, m_1) \leftarrow \mathcal{A}_1(k, h, \hat{g}, e, f, h);\ b \xleftarrow{\$} \{0, 1\};$
$c \leftarrow a^{z_1} \cdot \hat{a}^{z_2} \cdot m_b;$
$v \leftarrow H_k(a, \hat{a}, c);\ d \leftarrow a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2};$
$\boldsymbol{\gamma}^* \leftarrow (a, \hat{a}, c, d);\ \boldsymbol{\gamma}^*_{\mathsf{def}} \leftarrow \mathsf{true};$
$b' \leftarrow \mathcal{A}_2(\boldsymbol{\gamma}^*);$
return $(b = b')$
	if $\lvert \boldsymbol{L}_{\mathsf{Dec}} \rvert < q_{\mathsf{Dec}} \wedge \neg(\boldsymbol{\gamma}^*_{\mathsf{def}} \wedge (a, \hat{a}, c, d) = \boldsymbol{\gamma}^*)$
	then
	  $\boldsymbol{L}_{\mathsf{Dec}} \leftarrow \gamma :: \boldsymbol{L}_{\mathsf{Dec}};\ v \leftarrow H_k(a, \hat{a}, c);$
	  if $\hat{a} = a^w$ then ;
	    if $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$ then
	      return $c/(a^{z_1} \cdot \hat{a}^{z_2})$
	    else return $\perp$
	  elsif $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$ then
	    bad $\leftarrow$ true; return $c/(a^{z_1} \cdot \hat{a}^{z_2})$
	  else return $\perp$
	else return $\perp$

$\mathsf{DDH}_1 \sim \mathsf{G}_2 : \mathsf{true} \Rightarrow =_{\{\mathsf{res}\}}$

De la même manière, nous prouvons la transition entre le jeu DDH_1 et G_2, avec le même adversaire $\mathcal{B}$.

$$\Pr[\mathsf{DDH}_1 : b = b'] = \Pr[\mathsf{G}_2 : b = b'] \tag{4.4}$$

Nous modifions l'oracle de déchiffrement dans le jeu G_2 pour changer la valeur de bad à vrai quand $\mathcal{A}$ appelle l'oracle de déchiffrement avec un message valide c'est-à-dire avec $\log_a \hat{a} \neq \log_g \hat{g}$.

Game G_3 :	**Oracle** $\mathsf{Dec}(a, \hat{a}, c, d)$:
$g \xleftarrow{\$} \mathcal{G} \setminus \{1\};\ w \xleftarrow{\$} \mathbb{Z}_q^*;\ \hat{g} \leftarrow g^w;\ k \xleftarrow{\$} K;$ $x, x_2 \xleftarrow{\$} \mathbb{Z}_q;\ x_1 \leftarrow x - wx_2;\ e \leftarrow g^x;$ $y, y_2 \xleftarrow{\$} \mathbb{Z}_q;\ y_1 \leftarrow y - wy_2;\ f \leftarrow g^y;$ $z, z_2 \xleftarrow{\$} \mathbb{Z}_q;\ z_1 \leftarrow z - wz_2;\ h \leftarrow g^z;$ $(m_0, m_1) \leftarrow \mathcal{A}_1(k, h, \hat{g}, e, f, h);\ b \xleftarrow{\$} \{0,1\};$ $u, u' \xleftarrow{\$} \mathbb{Z}_q;\ a \leftarrow g^u;\ \hat{a} \leftarrow \hat{g}^{u'};$ $c \leftarrow a^{z_1} \cdot \hat{a}^{z_2} \cdot m_b;$ $v \leftarrow H_k(a, \hat{a}, c);\ d \leftarrow a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2};$ $\boldsymbol{\gamma}^* \leftarrow (a, \hat{a}, c, d);\ \boldsymbol{\gamma}^*_{\mathsf{def}} \leftarrow \mathsf{true};$ $b' \leftarrow \mathcal{A}_2(\boldsymbol{\gamma}^*);\ \mathsf{return}\ (b = b')$	if $\lvert \boldsymbol{L}_{\mathsf{Dec}} \rvert < q_{\mathsf{Dec}} \wedge \neg(\boldsymbol{\gamma}^*_{\mathsf{def}} \wedge (a, \hat{a}, c, d) = \boldsymbol{\gamma}^*)$ then $\boldsymbol{L}_{\mathsf{Dec}} \leftarrow \gamma :: \boldsymbol{L}_{\mathsf{Dec}};\ v \leftarrow H_k(a, \hat{a}, c);$ if $\hat{a} = a^w$ then if $d = a^{x+vy}$ then return c/a^z else return $\perp$ elsif $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$ then bad $\leftarrow$ true; return $\perp$ else return $\perp$ else return $\perp$

Nous montrons en utilisant la version sémantique du Lemme Fondamental 4.1 que la différence des probabilités de $(b = b')$ entre le jeu G_2 et le jeu G_3, où Dec rejette de tels messages, sont bornée par la probabilité de bad dans le jeu G_3.

$$\lvert \Pr[\mathsf{G}_2 : b = b'] - \Pr[\mathsf{G}_3 : b = b'] \rvert \leq \Pr[\mathsf{G}_3 : \mathsf{bad}] \tag{4.5}$$

Nous simplifions ensuite la manière de calculer e, f et h tout en préservant la sémantique.

Game G_4 :	**Oracle** $\mathsf{Dec}(a, \hat{a}, c, d)$:
$g \xleftarrow{\$} \mathcal{G} \setminus \{1\};\ w \xleftarrow{\$} \mathbb{Z}_q^*;\ \hat{g} \leftarrow g^w;\ k \xleftarrow{\$} K;$ $x, x_2 \xleftarrow{\$} \mathbb{Z}_q;\ x_1 \leftarrow x - wx_2;\ e \leftarrow g^x;$ $y, y_2 \xleftarrow{\$} \mathbb{Z}_q;\ y_1 \leftarrow y - wy_2;\ f \leftarrow g^y;$ $z \xleftarrow{\$} \mathbb{Z}_q;\ h \leftarrow g^z;$ $u, u' \xleftarrow{\$} \mathbb{Z}_q;\ a \leftarrow g^u;\ \hat{a} \leftarrow \hat{g}^{u'};$ $r \xleftarrow{\$} \mathbb{Z}_q;\ c \leftarrow g^r;$ $v \leftarrow H_k(a, \hat{a}, c);\ d \leftarrow a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2};$ $(m_0, m_1) \leftarrow \mathcal{A}_1(k, h, \hat{g}, e, f, h);\ b \xleftarrow{\$} \{0,1\};$ $\boldsymbol{\gamma}^* \leftarrow (a, \hat{a}, c, d);\ \boldsymbol{\gamma}^*_{\mathsf{def}} \leftarrow \mathsf{true};$ $b' \leftarrow \mathcal{A}_2(\boldsymbol{\gamma}^*);\ \mathsf{return}\ (b = b')$	if $\lvert \boldsymbol{L}_{\mathsf{Dec}} \rvert < q_{\mathsf{Dec}} \wedge \neg(\boldsymbol{\gamma}^*_{\mathsf{def}} \wedge (a, \hat{a}, c, d) = \boldsymbol{\gamma}^*)$ then $\boldsymbol{L}_{\mathsf{Dec}} \leftarrow \gamma :: \boldsymbol{L}_{\mathsf{Dec}};\ v \leftarrow H_k(a, \hat{a}, c);$ if $\hat{a} = a^w$ then if $d = a^{x+vy}$ then return c/a^z else return $\perp$ elsif $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$ then bad $\leftarrow$ true; return $\perp$ else return $\perp$ else return $\perp$

$\mathsf{G}_3 \sim \mathsf{G}_4 : \mathsf{true} \Rightarrow (u = u')\langle 1 \rangle \leftrightarrow (u = u')\langle 2 \rangle \wedge \big((u \neq u')\langle 1 \rangle \rightarrow =_{\{\mathsf{res}, \mathsf{bad}\}}\big)$

Dans G_4, nous supprimons la dépendance par rapport à b en choisissant uniformément r et en définissant $c = g^r$. Cette étape demande de pouvoir calculer z_2 à partir de $\log_g(c) = uz + (u - u')wz_2 + \log_g(m_b)$, ce qui n'est pas possible si $u = u'$, et cela se produit avec une probabilité $1/q$.

$$\Pr[\mathsf{G}_4 : b = b'] = 1/2. \tag{4.6}$$

Nous utilisons à nouveau la formulation sémantique du *Fundamental Lemma* pour borner la différence entre la probabilité de $(b = b')$ entre les jeux G_3 et G_4 par $1/q$.

$$\lvert \Pr[\mathsf{G}_3 : b = b'] - \Pr[\mathsf{G}_4 : b = b'] \rvert \leq \Pr[\mathsf{G}_3 : u = u'] = 1/q \tag{4.7}$$

De plus, l'invariant nous donne

$$\Pr[\mathsf{G}_3 : \mathsf{bad}] = \Pr[\mathsf{G}_4 : \mathsf{bad}] \tag{4.8}$$

Game $\boxed{\mathsf{G}_4'}$ G_5 :	**Oracle** $\mathsf{Dec}(a, \hat{a}, c, d)$:

$g \xleftarrow{\$} \mathcal{G} \setminus \{1\};\ w \xleftarrow{\$} \mathbb{Z}_q^*;\ \hat{g} \leftarrow g^w;\ k \xleftarrow{\$} K;$
$u, u' \xleftarrow{\$} \mathbb{Z}_q;\ a \leftarrow g^u;\ \hat{a} \leftarrow \hat{g}^{u'};$
$y, y_2 \xleftarrow{\$} \mathbb{Z}_q;\ y_1 \leftarrow y - wy_2;\ f \leftarrow g^y;$
$x \xleftarrow{\$} \mathbb{Z}_q;\ e \leftarrow g^x;\ r' \xleftarrow{\$} \mathbb{Z}_q;\ d \leftarrow g^{r'};$
$x_2 \leftarrow (r' - u(x + vy))/(w(u' - u)) - vy_2;$
$x_1 \leftarrow x - wx_2;\ z \xleftarrow{\$} \mathbb{Z}_q; h \leftarrow g^z;$
$r \xleftarrow{\$} \mathbb{Z}_q;\ c \leftarrow g^r;$
$v \leftarrow H_k(a, \hat{a}, c);\ \boldsymbol{\gamma}^* \leftarrow (a, \hat{a}, c, d);$
$(m_0, m_1) \leftarrow \mathcal{A}_1(k, h, \hat{g}, e, f, h);$
$\boldsymbol{\gamma}^*_{\mathsf{def}} \leftarrow \mathsf{true};\ b' \leftarrow \mathcal{A}_2(\boldsymbol{\gamma}^*);\ \mathsf{return}\ (b = b')$
	if $\lvert \boldsymbol{L}_{\mathsf{Dec}} \rvert < q_{\mathsf{Dec}} \wedge \neg\boldsymbol{\gamma}^*_{\mathsf{def}} \wedge (a, \hat{a}, c, d) = \boldsymbol{\gamma}^*$
	then $\mathsf{bad}_1 \leftarrow \mathsf{true}$;
	if $\lvert \boldsymbol{L}_{\mathsf{Dec}} \rvert < q_{\mathsf{Dec}} \wedge (\boxed{\neg\boldsymbol{\gamma}^*_{\mathsf{def}} \vee} (a, \hat{a}, c, d) \neq \boldsymbol{\gamma}^*)$
	then $\boldsymbol{L}_{\mathsf{Dec}} \leftarrow \gamma :: \boldsymbol{L}_{\mathsf{Dec}};\ v \leftarrow H_k(a, \hat{a}, c);$
	  if $\hat{a} = a^w$ then
	    if $d = a^{x+vy}$ then return c/a^z
	    else return $\perp$
	  elsif $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$ then
	    $\mathsf{bad} \leftarrow \mathsf{true}$;
	    if $v = H_k(g^u, \hat{g}^{u'}, g^r)$ then
	      $\mathsf{bad}_2 \leftarrow \mathsf{true}$
	  else return $\perp$
	else return $\perp$

$\mathsf{G}_4 \sim \mathsf{G}_4' : \mathsf{true} \Rightarrow (u = u')\langle 1 \rangle \leftrightarrow (u = u')\langle 2 \rangle \wedge \left((u \neq u')\langle 1 \rangle \to =_{\{\mathsf{bad}\}}\right)$
$\mathsf{G}_4' \sim \mathsf{G}_5 : \mathsf{true} \Rightarrow =_{\{\mathsf{bad}_1\}} \wedge \left(\neg\mathsf{bad}_1\langle 1 \rangle \to =_{\{\mathsf{bad}, u, u'\}}\right)$

Nous pouvons maintenant déplacer la plupart des instructions du jeu avant l'appel à $\mathcal{A}_1$. Nous pouvons alors rendre d aléatoire, en choisissant uniformément $r' = \log_g(d)$ et nous définissons x_2 en fonction de r'. Le jeu calcule le *ciphertext* en avance, nous modifions Dec pour rendre bad_1 vrai si le *ciphertext* $\boldsymbol{\gamma}^*$ est demandé pendant la première phase du jeu. Notons que dans cette étape, le *ciphertext* est un 4-uplet d'éléments tirés uniformément, la probabilité de bad_1 est bornée par $(q_{\mathsf{Dec}}/q)^4$. Nous obtenons

$$\Pr[\mathsf{G}_4 : \mathsf{bad} \wedge u \neq u'] \leq \Pr[\mathsf{G}_5 : \mathsf{bad} \wedge u \neq u'] + (q_{\mathsf{Dec}}/q)^4. \tag{4.9}$$

Game TCR :	**Oracle** $\mathsf{Dec}(a, \hat{a}, c, d)$:

$m_0 \leftarrow \mathcal{C}_1(\,);\ k \xleftarrow{\$} K;\ m_1 \leftarrow \mathcal{C}_2(k);$
return $(H_k(m_0) = H_k(m_1) \wedge m_0 \neq m_1)$
Adversary $\mathcal{C}_1()$:
$g \xleftarrow{\$} \mathcal{G} \setminus \{1\};\ w \xleftarrow{\$} \mathbb{Z}_q^*;\ \hat{g} \leftarrow g^w;$
$u, u' \xleftarrow{\$} \mathbb{Z}_q;\ a \leftarrow g^u;\ \hat{a} \leftarrow \hat{g}^{u'};$
$r \xleftarrow{\$} \mathbb{Z}_q;\ c \leftarrow g^r;$ return $(a, \hat{a}, c)$
Adversary $\mathcal{C}_2(k)$:
$r', x, y, z \xleftarrow{\$} \mathbb{Z}_q;$
$d \leftarrow g^{r'}; e \leftarrow g^x; f \leftarrow g^y;\ h \leftarrow g^z;$
$y_2 \xleftarrow{\$} \mathbb{Z}_q;\ y_1 \leftarrow y - wy_2;\ \hat{k} \leftarrow k;$
$v \leftarrow H_k(a, \hat{a}, c);$
$x_2 \leftarrow (r' - u(x + vy))/(w(u' - u)) - vy_2;$
$x_1 \leftarrow x - wx_2;$
$(m_0, m_1) \leftarrow \mathcal{A}_1(h, \hat{g}, e, f, h);$
$\gamma \leftarrow (a, \hat{a}, c, d);\ b' \leftarrow \mathcal{A}_2(\boldsymbol{\gamma}^*);$ return $\hat{m}$
	if $\lvert \boldsymbol{L}_{\mathsf{Dec}} \rvert < q_{\mathsf{Dec}} \wedge (a, \hat{a}, c, d) \neq \boldsymbol{\gamma}^*$ then
	  $\boldsymbol{L}_{\mathsf{Dec}} \leftarrow \gamma :: \boldsymbol{L}_{\mathsf{Dec}};$
	  $v \leftarrow H_{\hat{k}}(a, \hat{a}, c);$
	  if $\hat{a} = a^w$ then
	    if $d = a^{x+vy}$ then return c/a^z
	    else return $\perp$
	  elsif $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$ then
	    if $v = H_{\hat{k}}(g^u, \hat{g}^{u'}, g^r)$ then
	      $\hat{m} \leftarrow (a, \hat{a}, c);$
	      return $\perp$
	  else return $\perp$
	else return $\perp$

$\mathsf{G}_5 \sim \mathsf{TCR} : \mathsf{true} \Rightarrow \mathsf{bad}_2\langle 1 \rangle \to \mathsf{res}\langle 2 \rangle$
$\Pr[\mathsf{G}_5 : \mathsf{bad} \wedge u \neq u'] \leq$
$\Pr[\mathsf{TCR} : H_k(m_0) = H_k(m_1) \wedge m_0 \neq m_1] + \Pr[\mathsf{G}_5 : \mathsf{bad} \wedge u \neq u' \wedge \neg\mathsf{bad}_2]$

L'oracle de déchiffrement dans le jeu G_5 rend l'évènement bad_2 vrai, si un *ciphertext*

valide avec $H_k(a,\hat{a},c) = H_k(g^u,\hat{g}^{u'},g^r)$ est demandé. Cette requête créerait une collision, nous pouvons ensuite construire un adversaire $\mathcal{C}$ contre TCR pour H tel que le succès de $\mathcal{C}$ soit plus petit que la probabilité de bad_2 dans le jeux G_5. Nous avons

$$\Pr\left[\mathsf{G}_5 : \mathsf{bad} \wedge u \neq u'\right] \leq \mathbf{Adv}^{\mathcal{C}}_{\mathsf{TCR}} + \Pr\left[\mathsf{G}_5 : \mathsf{bad} \wedge u \neq u' \wedge \neg\mathsf{bad}_2\right]. \quad (4.10)$$

Nous concluons la preuve en montrant que la probabilité de bad et de $\neg\mathsf{bad}_2$ est bornée par q_{Dec}/q. Nous reformulons le code de l'oracle pour que le bad_2 ne dépende pas de x_1,x_2,y_1,y_2. La probabilité que ce test soit vrai dans l'oracle de déchiffrement, (sous la condition que $u \neq u'$) est égale à la probabilité que l'adversaire tire une certaine valeur dans un groupe. Étant donné que l'adversaire fait q_{Dec}, cette probabilité est q_{Dec}/q.

$$\Pr\left[\mathsf{G}_5 : \mathsf{bad} \wedge u \neq u' \wedge \neg\mathsf{bad}_2\right] = q_{\mathsf{Dec}}/q. \quad (4.11)$$

En résumant la preuve nous obtenons :

$$\begin{array}{ll}
\mathbf{Adv}^{\mathcal{A}}_{\mathsf{CCA}}(q_{\mathsf{Dec}}) = \left|\Pr\left[\mathsf{IND\text{-}CCA2} : b = b'\right] - \frac{1}{2}\right| & \\
= \left|\Pr\left[\mathsf{G}_1 : b = b'\right] - \frac{1}{2}\right| & (4.2)\\
= \left|\Pr\left[\mathsf{DDH}_0 : b = b'\right] - \frac{1}{2}\right| & (4.3)\\
\leq \left|\Pr\left[\mathsf{DDH}_0 : b = b'\right] - \Pr\left[\mathsf{DDH}_1 : b = b'\right]\right| + \left|\Pr\left[\mathsf{DDH}_1 : b = b'\right] - \frac{1}{2}\right| & (\mathsf{triang})\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \left|\Pr\left[\mathsf{DDH}_1 : b = b'\right] - \frac{1}{2}\right| & (4.1)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \left|\Pr\left[\mathsf{G}_2 : b = b'\right] - \frac{1}{2}\right| & (4.4)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \left|\Pr\left[\mathsf{G}_2 : b = b'\right] - \Pr\left[\mathsf{G}_4 : b = b'\right]\right| & (4.6)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \left|\Pr\left[\mathsf{G}_2 : b = b'\right] - \Pr\left[\mathsf{G}_3 : b = b'\right]\right| + \left|\Pr\left[\mathsf{G}_3 : b = b'\right] - \Pr\left[\mathsf{G}_4 : b = b'\right]\right| & (\mathsf{triang})\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \Pr\left[\mathsf{G}_3 : \mathsf{bad}\right] + \left|\Pr\left[\mathsf{G}_3 : b = b'\right] - \Pr\left[\mathsf{G}_4 : b = b'\right]\right| & (4.5)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \Pr\left[\mathsf{G}_3 : \mathsf{bad}\right] + 1/q & (4.7)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \Pr\left[\mathsf{G}_3 : \mathsf{bad} \wedge u = u'\right] + \Pr\left[\mathsf{G}_3 : \mathsf{bad}, u \neq u'\right] + 1/q & (\mathsf{pr}_1)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \Pr\left[\mathsf{G}_3 : u = u'\right] + \Pr\left[\mathsf{G}_4 : \mathsf{bad} \wedge u \neq u'\right] + 1/q & (\mathsf{pr}_2, 4.8)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + 2/q + \Pr\left[\mathsf{G}_4 : \mathsf{bad} \wedge u \neq u'\right] & (4.7)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + 2/q + \Pr\left[\mathsf{G}_5 : \mathsf{bad} \wedge u \neq u'\right] + (q_{\mathsf{Dec}}/q)^4 & (4.9)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \mathbf{Adv}^{\mathcal{C}}_{\mathsf{TCR}} + 2/q + (q_{\mathsf{Dec}}/q)^4 + \Pr\left[\mathsf{G}_5 : \mathsf{bad} \wedge u \neq u' \wedge \neg\mathsf{bad}_2\right] & (4.10)\\
\leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \mathbf{Adv}^{\mathcal{C}}_{\mathsf{TCR}} + \frac{q_{\mathsf{Dec}}^4}{q^4} + \frac{q_{\mathsf{Dec}}+2}{q}. & (4.11)
\end{array}$$

Avec pour tous évènements A, B et C :

$$\begin{array}{ll}
\mathsf{pr}_1 & : \Pr[A] = \Pr[A \wedge B] + \Pr[A \wedge \neg B]\\
\mathsf{pr}_2 & : \Pr[A \wedge B] \leq \Pr[A]\\
\mathsf{triang} & : |\Pr[A] - \Pr[C]| \leq |\Pr[A] - \Pr[B]| + |\Pr[B] - \Pr[C]|
\end{array}$$

4.4 Limitations et Extensions

EasyCrypt est encore un outil en développement ; nous décrivons dans cette section certaine limitations et les futures extensions possibles :

- En comparaison avec CertiCrypt, le langage d'EasyCrypt ne contient pas de boucle, d'appel récursif et de tirage aléatoire non uniforme. Nous n'avons pas besoin d'étendre le langage avec des appels récursifs. Par contre, nous pensons que rajouter les boucles et d'avantage de types de tirage aléatoire est utile (la génération de condition pour les boucles demande d'ajouter des invariants pour chaque boucle).
- La génération de preuves : EasyCrypt ne génère qu'une partie des preuves en Coq (voir Chapitre suivant). L'absence de prouveur SMT qui génèrent des preuves Coq, nous oblige à admettre les obligations de preuves. La génération de preuve pour les SMT est un sujet de recherche active [82] et des avancés dans ce domaine serait bénéfique pour EasyCrypt.
- Calcul de probabilité : EasyCrypt ne génère pas complètement les preuves pour les propriétés probabilistes. Il faut pouvoir calculer symboliquement la probabilité d'un évènement dans une distribution.

Nous donnerons deux pistes pour améliorer l'automatisation des preuves dans EasyCrypt :

- améliorer le mécanisme d'inférence pour les spécification relationnelles de l'adversaire
- utiliser la sythèse de programme (*program synthesis*) dont le but est de construire un programme satisfaisant une certaine spécification. Nous pourrions utiliser cette méthode pour trouver une partie de la séquence de jeu qui permettrait de finir la preuve, et de construire les adversaires pour justifier les réductions utilisant les hypothèses cryptographiques.

L'inférence de spécification et la synthèse de programmes sont reliés à la génération de condition de vérification et aux prouveurs automatiques, les blocs de base pour étendre EasyCrypt sont déjà en place.

Finalement, Halevi [54] souligne que "the usefulness of (a) tool will depend crucially on the willingness of the customers (in this case the cryptographic community) to use it". Il suggère également que l'outil doit fournir une interface utilisateur. Nous adhérons à son point de vue, et le développement d'une telle interface et un de nos objectifs pour les travaux futurs.

4.4.1 Comparaison avec CertiCrypt

Le tableau 4.1 compare CertiCrypt et EasyCrypt sur différentes preuves de sécurité formalisées dans les deux outils. Le temps d'exécution est mesuré sur un processeur 2.8GHz Intel Core 2 Duo avec 4GB de RAM sous Mac OS X 10.6.7. Pour comparer les outils, nous montrons la taille et le temps d'exécution des preuves CertiCrypt extraites à partir de EasyCrypt. La comparaison n'est pas juste, car les preuves extraites admettent les obligations de preuve vérifiées par les prouveurs automatiques.

Nous avons complété la preuve de ElGamal généré en Coq depuis EasyCrypt. Nous obtenons une preuve de 1130 lignes et seulement 43 lignes de Coq sont nécessaires pour prouver les obligations de preuve et 25s sont nécessaires pour vérifier la preuve.

Tableau 4.1. Comparaison de la tailles des preuves entre CertiCrypt et EasyCrypt.

	CertiCrypt		EasyCrypt		Extracted	
	Lignes	Temps	Lignes	Temps	Lignes	Temps
ElGamal (IND-CPA)	565	45s	190	12s	1130	23s
Hashed ElGamal (IND-CPA)	1255	1m05s	243	33s	1772	41s
Full-Domain Hash (EF-CMA)	2035	5m46s	509	1m26s	2724	1m11s
Cramer-Shoup (IND-CCA2)	n/a	n/a	1637	5m12s	5504	3m14s
OAEP (IND-CPA)	2451	3m27s	n/a	n/a	n/a	n/a
OAEP (IND-CCA2)	11162	37m32s	n/a	n/a	n/a	n/a

4.5 Conclusion

La vérification sur machine de protocoles dans le modèle symbolique est un sujet de recherche concluant : des outils robustes existent et ont été utilisée avec succès pour analyser des protocoles existants (e.g. [73, 39, 28, 13]).

Par contre, il y a peu de travaux équivalents dans le modèle calculatoire. Les deux principaux outils pour faire de telles preuves sont : CertiCrypt permettant de faire des preuves générales et d'obtenir des preuves vérifiables sur machines et CryptoVerif qui permet de faire des preuves automatiquement.

Dans ce chapitre nous avons présenté EasyCrypt, un nouvel outil qui permet à la fois d'être un outil flexible et automatique et qui génère des preuves vérifiables sur machine. Nous avons illustré l'outil en présentant deux preuves : Hashed ElGamal dans le modèle des oracles aléatoires et Cramer-Shoup dans le modèle standard.

Ces exemples démontrent que les preuves dans EasyCrypt sont bien plus simples et rapides à écrire que dans les outils précédents, et dans le Chapitre suivant nous expliquerons comment les preuves EasyCrypt offrent la même garantie que celles de CertiCrypt.

Nous pensons que EasyCrypt est une étape importante vers l'adoption pour les cryptographes, d'outils pour faire des preuves sur machine.

5
Représentation et Génération des preuves

En lisant le Chapitre 4, le lecteur peut penser que EasyCrypt est bien plus pratique que CertiCrypt, mais n'est pas certifié. Le choix de développer EasyCrypt indépendamment de Coq est volontaire. Ceci nous permet d'avoir une plus grande flexibilité dans le développement en particulier cela permet d'appeler facilement les prouveurs SMT.

Pour obtenir une garantie sur les preuves effectuées dans EasyCrypt, nous allons les certifier dans Coq. Nous générons pour chaque preuve un fichier compatible avec la bibliothèque CertiCrypt, contenant la justification de la plupart des étapes de la preuve.

À l'heure actuelle, la preuve générée est partielle, plusieurs parties ne sont pas certifiées. En particulier les appels des prouveurs SMT effectués par EasyCrypt sont admis en Coq. Pour le moment, l'utilisateur peut toujours combler les trous en prouvant en Coq les bouts restants. À plus long terme, il sera possible d'appeler des prouveurs SMT directement dans Coq, comme le montrent les travaux récents de Armand et al. et Stump [7, 6, 82].

Pour vérifier l'ensemble de la preuve dans CertiCrypt, la première étape consiste à exporter l'ensemble des types, constantes, axiomes, adversaires et jeux vers un fichier Coq.

La représentation des jeux est différente entre EasyCrypt et CertiCrypt. CertiCrypt ne contient pas la notion de jeux de manière intrinsèque. Les jeux sont représentés par des procédures qui peuvent s'appeler à travers un environnement. Ces différences sont un point important lors de la traduction.

Nous avons vu dans le chapitre précèdent, que la majorité des transitions sont des propriétés qui se déduisent à partir des jugements pRHL. Nous commencerons par expliquer comment vérifier les preuves des jugements pRHL, qui sont représentées sous la forme d'arbre.

Les arbres de preuve contiennent les informations nécessaires pour vérifier la preuve. Chaque noeud représente l'application d'une règle logique. Les feuilles représentent des formules logiques vérifiées par les prouveurs SMT.

Ensuite, nous prouvons les jugements pRHL en utilisant l'arbre de preuve. Nous générons une preuve de l'existence d'une dérivation. Une fois les équivalences prouvées nous pouvons dériver les propriétés probabilistes.

Dans ce chapitre nous expliquerons comment construire la sémantique des langages probabilistes dans CertiCrypt, puis comment la générer automatiquement à partir d'un fichier EasyCrypt. Nous expliquerons ensuite la traduction des preuves d'équivalence. Pour finir nous montrerons les règles relatives aux adversaires et aux probabilités.

5.1 Traduction de la sémantique

CertiCrypt est construit de manière modulaire et permet de rajouter de nouveaux types ou opérateurs à la sémantique prédéfinie dans CertiCrypt. Dans cette partie, nous commencerons par expliquer comment cette sémantique peut être étendue dans CertiCrypt, puis nous montrerons comment générer automatiquement cette sémantique à partir des fichiers EasyCrypt. Nous utilisons à nouveau la preuve de Hashed ElGamal comme exemple.

5.1.1 Pré-requis

L'ensemble des tactiques fournies dans CertiCrypt utilise intensivement le test d'égalité sur les variables. Les variables sont définies de la manière suivante :

$$\begin{array}{l} \textsf{Inductive Var} : \textsf{T.type} \to \textsf{Type} := \\ \mid \textsf{Gvar} : \forall (t : \textsf{T.type}) \to \mathbb{N} \to \textsf{Var}\ t \\ \mid \textsf{Lvar} : \forall (t : \textsf{T.type}) \to \mathbb{N} \to \textsf{Var}\ t \end{array}$$

Le type Gvar t n, représente la variable globale de type t identifiée par l'entier n. Deux variables sont égales si les types sont égaux et les identifients sont égaux. Nous noterons $\mathcal{V}_t$ le type inductif Var t de CertiCrypt.

Pour définir l'égalité sur les variables, nous avons besoin de définir l'égalité sur les types et sur les entiers. La définition de l'égalité ne pose pas de problème, par contre l'évaluation de l'égalité pose problème si les entiers ou les types ne sont pas définis.

Nous devons nous placer dans une situation où tous les types et les entiers sont définis. Si un nom de variables est représenté par une variable Coq (sans valeur), la fonction d'égalité ne se réduira pas. De même, les types doivent être clos et ne pas dépendre de variable Coq. Ceci afin de permettre aux tactiques d'évaluer le test d'égalité sur les variables.

Nous avons le même problème avec l'égalité des noms des procédures qui sont définis de la manière suivante :

$$\begin{array}{l} \textsf{Inductive Proc} : \textsf{T.type} \to \textsf{Type} := \\ \mid \textsf{mkP} : \mathbb{N} \to \textsf{list T.type} \to \forall (t : \textsf{T.type}) \to \textsf{Proc}\ t \end{array}$$

5.1.2 Sémantique dans CertiCrypt

La sémantique des programmes probabilistes est construite de manière modulaire dans CertiCrypt. Elle est paramétrée par trois modules Coq qui contiennent les déclarations des types, des opérateurs et des tirages aléatoires définis pas l'utilisateur.

Les modules contiennent les différentes interprétations en Coq des objets manipulés ainsi qu'une fonction d'égalité sur les types et les opérateurs.

Types

Nous décrivons la signature du module UTYPE pour ajouter de nouveaux types à la sémantique.

```
Module Type UTYPE :=
t : Type
eqb : t → t → bool
eqb_spec : ∀(x y : t), if eqb x y then x = y else x ≠ y.
interp : nat → t → Type
default : ∀k t, interp k t
i_eqb : ∀ k t, interp k t → interp k t → bool
i_eqb_spec : ∀k (x y : interp k t), if i_eqb k x y then x = y else x ≠ y.
End UTYPE
```

Le type t représente une version syntaxifiée des types de l'utilisateur. eqb_spec est une preuve que eqb x y reflète l'égalité de x et y. La fonction interp fait le lien entre le type syntaxifié CertiCrypt et les types de Coq. La fonction default représente l'élément par défaut permettant d'obtenir une valeur par défaut quand une variable n'est pas définie, l'interprétation d'un type ne peut donc pas être vide.

Les fonctions interp et default dépendent d'un entier représentant le paramètre de sécurité. La taille des types peut dépendre de ce paramètre et dans la preuve, nous nous servons du fait que les fonctions sont négligeables en fonction du paramètre de sécurité. Par exemple nous pouvons construire un type bitstring dont la taille est le paramètre de sécurité.

La sémantique pour les types prédéfinis fournit une fonction d'égalité polymorphe. Pour un type syntaxique t, la fonction i_eqb teste l'égalité entre deux éléments dont le type est l'interprétation du type t, la fonction i_eqb permet de définir la fonction d'égalité polymorphe. Comme pour eqb, nous prouvons la spécification de l'égalité par le lemme i_eqb_spec.

Le module type TYPE représente les types prédéfinis dans CertiCrypt, étendu par les types de l'utilisateur. Le foncteur

```
Module MakeType (UT : UTYPE) <: TYPE UT.
```

construit la sémantique des types. Ce foncteur contient un élément type contenant l'ensemble des types sous une forme syntaxifiée. Cet élément contient un type User qui contient les types définis par l'utilisateur.

Opérateurs

Nous pouvons ensuite déclarer de nouveaux opérateurs à partir des types de CertiCrypt. Nous décrivons la signature du module type UOP paramétré par un module de type TYPE UT lui même paramétré par un module UTYPE :

```
Module Type UOP (UT : UTYPE) (T : TYPE UT) :=
t : Type
eqb : t → t → bool
eqb_spec : ∀(x y : t), if eqb x y then x = y else x ≠ y.
targs : t → list T.type
tres : t → T.type
interp_op : ∀op, T.type_op (targs op) (tres op)
End UOP
```

t représente l'ensemble des opérateurs sous forme syntaxifiée. eqb représente la fonction d'égalité sur les opérateurs et eqb_spec sa spécification. La fonction targs associe un opérateur à la liste des types syntaxifiés de ses arguments et tres rend le type de son résultat. Enfin interp_op est la fonction d'interprétation des opérateurs. La fonction T.type_op permet d'interpréter une liste de types syntaxifiés représentant les types des arguments de l'opérateur et le type de retour en un type coq. Pour une liste de types syntaxifiés $(t_1 :: t_2 :: \ldots :: t_n)$ et un type de retour t, la fonction T.type_op renvoie le type Coq suivant :

$$\mathsf{T.interp}\ k\ t_1 \to \mathsf{T.interp}\ k\ t_2 \to \cdots \to \mathsf{T.interp}\ k\ t_n \to \mathsf{T.interp}\ k\ t$$

L'interprétation d'un opérateur est donc une fonction Coq dont le type est compatible avec le type déclaré de l'opérateur.

Tirages aléatoires

Le module type USUPPORT permet d'ajouter de nouveaux tirages aléatoires (par exemple sur les nouveaux types crées). Comme pour UOP, le module est paramétré par un module TYPE

```
Module Type USUPPORT (UT : UTYPE) (T : TYPE UT) :=
usupport : T.type → Type
eval : ∀k t, usupport t → list (T.interp k t)
eval_support_nil : ∀k t (s : usupport t), eval k s ≠ nil
eqb : ∀ t1 t2, usupport t1 → usupport t2 → bool
eqb_spec_dep : ∀ t1 t2 (e1 : usupport t1) (e2 : usupport t2),
if eqb e1 e2 then eq_dep t1 e1 t2 e2 else ¬eq_dep t1 e1 t2 e2
eqb_spec : ∀t (e1 e2 : usupport t), if eqb e1 e2 then e1 = e2 else e1 ≠ e2.
End USUPPORT
```

usupport prend en argument le type sur lequel nous voulons définir le tirage aléatoire, et renvoie le type syntaxifié Coq correspondant au support. La fonction eval donne

l'interprétation du support, qui est une liste de valeurs pour le type associé. Nous devons prouver que l'interprétation du support n'est pas vide (eval_support_nil). eqb donne l'égalité sur les supports.

Une fois les trois modules définis, nous pouvons construire la sémantique et les tactiques à partir des modules suivants :

```
Module Sem := MakeSem.Make Ut T Uop US.
Module SemT := BaseProp.Make Sem.
```

Un dernier module permet de définir les différents polynômes qui bornent la taille et le temps d'exécution des supports et des expressions. Ce module sert à prouver que les programmes sont PPT. Pour le moment nous n'avons par la notion de PPT dans EasyCrypt, nous ne traiterons donc pas ce module dans la traduction. Nous pourrions ajouter dans EasyCrypt un mot clé qui permettrait de distinguer les opérateurs polynomiaux des autres et demander à l'utilisateur de prouver l'existence des polynômes dans CertiCrypt. Les propriétés vérifiées dans ce module sont expliquées dans le chapitre 6.

5.1.3 Construction de la sémantique dans CertiCrypt à partir de EasyCrypt

Un programme CertiCrypt est une famille de programmes paramétrés par un paramètre de sécurité. L'interprétation des types et des opérateurs dépend donc de ce paramètre appelé k dans la suite de ce chapitre.

Traduction des types

Les types de EasyCrypt sont abstraits ou définis à partir d'autres types. Dans Coq, nous définissons un nouveau type dans le premier cas et une nouvelle notation dans le second. Les types prédéfinis de EasyCrypt sont déjà inclus dans CertiCrypt à l'exception des bitstrings que nous devons rajouter.

Nous avons vu dans les pré-requis que les tactiques ont besoin de pouvoir évaluer l'égalité des types. C'est pourquoi, le type bitstring k représentant les bitstrings de longueur k (avec k abstrait), ne peut pas être ajouté dans les types de base de CertiCrypt. En effet, la fonction d'égalité pose problème, si nous avons deux longueurs de bitstring bitstring k_1 et bitstring k_2, et que k_1 et k_2 sont des constantes abstraites, la fonction d'égalité eqb ne se réduira pas.

La solution adoptée est la suivante : nous parcourons l'ensemble des définitions de EasyCrypt, pour prendre en compte toutes les tailles de bitstring possibles. Pour chaque longueur l, nous créons le type bitstring_l. Et nous choisissons d'écrire la fonction d'égalité des types de telle sorte que pour deux longueurs k_1 et k_2 les types bitstring_k_1 et bitstring_k_2 soient différents.

Nous aurions pu aussi avoir un type bitstring ne dépendant pas de la taille des éléments, et un prédicat size b l déclarant que le bitstring b est de taille l.

Dans la preuve de Hashed ElGamal, nous déclarons un type abstrait group et nous utilisons des bitstrings de longueur k où k est une constante dépendant du paramètre de sécurité.

```
Inductive ut : Type :=
 | group
 | BitString_k.
```

Pour définir la sémantique, nous avons besoin d'interpréter les types, et de prouver certaines propriétés sur ces types. Dans la preuve EasyCrypt nous déclarons le type group :

```
type group
```

Dans Coq, nous devons ajouter des axiomes pour définir l'interprétation des types Coq :

```
Parameter group_def : nat → Type
Parameter group_default : ∀(k : nat), group_def k.
Parameter group_eqb : ∀k(x y : group_def k), bool.
Parameter group_eqb_spec : ∀k(x y : group_def k),
  if group_eqb k x y then x = y else x ≠ y.
```

group_def représente l'interprétation du type en Coq, group_default l'élément par défaut. Nous ajoutons la fonction d'égalité décidable group_eqb pour les éléments des groupes et sa spécification group_eqb_spec. L'ensemble de ces paramètres dépend du paramètre de sécurité k

Nous déclarons ensuite un module de type UTYPE, qui comprend le type group et également un type bitstring par longueur de bitstrings utilisés :

```
Module Ut <: UTYPE.
Definition t := ut.

Definition interp (k : nat) (t0 : t) :=
match t0 with
 | group => group_def k
 | BitString_k => Bvector (Zabs_nat (k_ k))
end.

Definition default (k : nat)(t0 : t) : interp k t0 :=
match t0 with
 | group => group_default k
 | BitString_k => Bvect_false (Zabs_nat (k_ k))
end.

Definition i_eqb(k : nat)(t0 : t)(x y : interp k t0) : bool :=
match t0 return interp k t0 → interp k t0 → bool with
 | group => group_eqb k
 | BitString_k => fun x y => Veqb x y
end x y.

End Ut.

Module T := MakeType Ut.
```

La longueur des bitstring k est expliquée plus loin dans le chapitre, $k_$ représente l'interprétation en Coq de la constante EasyCrypt k (ne pas confondre avec le paramètre de sécurité).

Le module comprend la définition du type. L'égalité sur ces types et sa spécification sont générés automatiquement par la commande Scheme Equality. L'interprétation des types fait le lien entre les types CertiCrypt et les types Coq en utilisant les paramètres définis ci-dessus. La fonction default renvoie l'élément par défaut en utilisant également ces paramètres. La fonction i_eqb est définie à partir des paramètres que nous avons définis plus haut et de l'égalité sur des bitstrings.

Nous construisons ensuite l'ensemble des types, en utilisant le foncteur MakeType

Notations

Dans EasyCrypt nous avons vu que certains types sont définis à partir d'autres types :

```
type skey    = int
type pkey    = group
type key     = skey * pkey
type message = bitstring{k}
type cipher  = group * bitstring{k}
```

Nous créons simplement une notation de la manière suivante :

```
Notation skey := T.Zt.
Notation pkey := (T.User group).
Notation key := (T.Pair skey pkey).
Notation message := (T.User BitString_k).
Notation cipher := (T.Pair (T.User group) (T.User BitString_k)).
```

Les notations nous permettent d'obtenir la même lisibilité entre EasyCrypt et CertiCrypt.

Nous créons également des notations pour chaque variable, de la manière suivante :

Notation x_{24} := (Var.Lvar (T.User group) 128).

ceci nous permet de tester l'égalité des variables tout en restant lisible.

Dans CertiCrypt les variable sont définit par un identifiant représenté par un entier et de son type. Pour chaque variable nous déclarons également une notation, par exemple :

```
Notation INDCPA_Enc := (Proc.mkP 31 (pkey :: message :: nil) cipher).
```

Nous notons que le nom des variables ou des procédures (ici 128 ou 31) est généré automatiquement et était souvent une source d'erreur dans l'écriture de la sémantique dans CertiCrypt.

Traduction des opérateurs

Nous créons ensuite le module pour les nouveaux opérateurs. Les constantes EasyCrypt sont définies comme des opérateurs d'arité 0. Dans EasyCrypt, nous avons les déclarations suivantes :

```
cnst q    : int
cnst g    : group
cnst k    : int
cnst zero : bitstring{k}
op (*) : group, group → group = mul
op (^) : group, int → group = pow
op (^^) : bitstring{k}, bitstring{k} → bitstring{k} = xor
```

que nous traduisons en Coq, par le type uop :

```
Inductive uop : Type :=
 | Omul
 | Opow
 | Oxor
 | Oq
 | Og
 | Ok
 | Ozero.
```

Et pour chaque nouvelle constante ou opérateur, nous déclarons également son interprétation en Coq comme nous l'avons fait pour les types. Par exemple pour la preuve de Hashed ElGamal :

```
Parameter k_ : ∀ (k : nat), Z.
Parameter zero : ∀ (k : nat), Bvector (Zabs_nat (k_ k)).
Parameter mul : ∀ (k : nat), (group_def k) → (group_def k) → (group_def k).
...
```

Nous définissions un module de type UOP, paramétré par les modules Ut et T précédemment définis.

```
Module Uop <: UOP Ut T.
Definition t := uop.
Definition targs (op : t) : list T.type :=
match op with
 | Omul ⇒ (T.User group) :: (T.User group) :: nil
 | Ozero ⇒ nil
...
end.

Definition tres (op : t) : T.type :=
match op with
 | Omul ⇒ (T.User group)
 | Ozero ⇒ (T.UserBitString_k)
...
end.


Definition interp_op (k : nat)(op : t) : T.type_op k (targs op) (tres op) :=
match op as op0 return T.type_op k (targs op0)(tres op0) with
 | Ok_ ⇒ k_ k
 | Omul ⇒ mul k
 | Ozero ⇒ zero k
...
end.
End Uop.
```

la fonction d'égalité et sa spécification sont définies par les commandes Scheme Equality. Nous donnons ensuite pour tous les opérateurs le type des paramètres et le type de retour. Nous définissons l'interprétation des opérateurs, à partir des axiomes que nous avons posés ci-dessus. Nous ajoutons automatiquement un tiret après chaque constante, pour éviter de confondre le k du paramètre de sécurité et une variable EasyCrypt ayant le nom k.

Nous ajoutons ensuite le tirage aléatoire sur les bitstrings. Les autres tirages de EasyCrypt étant déjà inclus dans CertiCrypt. Nous commençons par définir le type des supports pour les tirages

```
Inductive usupport_ (Ttype : Type)(Tuser : ut → Ttype) : Ttype → Type :=
 | Ubs_k : usupport_ Tuser (Tuser BitString_k).
```

Comme pour les types et opérateurs, nous construisons un module, pour définir le nouveau support de type USUPPORT.

```
Module US <: USUPPORT Ut T.
Definition usupport := usupport_ T.User.

Definition eval k t (s : usupport t) : list (T.interp k t) :=
match s in usupport_ t0 return list (T.interp kt0) with
 | Ubs_k => bs_support (Zabs_nat (k_ k))
end.
```

nous définissons la fonction d'interprétation des supports où bs_support l contient l'ensemble des bitstrings de longueur l. Nous prouvons également que les supports sur les bitstrings ne sont pas vides et définissons la fonction d'égalité avec ses spécifications.

Axiomes

Nous finissons de déclarer la sémantique en ajoutant les axiomes déclarés dans EasyCrypt :

```
axiom pow_mult : ∀(x:int,y:int). { (g^x)^y = g^(x*y) }
axiom xor_comm : ∀ (x:bitstring{k}, y:bitstring{k}).
  { (x ^^ y) = (y ^^ x) }
axiom xor_assoc : ∀ (x:bitstring{k},y:bitstring{k},z:bitstring{k}).
  { ((x ^^ y) ^^ z) = (x ^^ (y ^^ z)) }
axiom xor_zero : ∀ (x:bitstring{k}). { (x ^^ zero) = x }
axiom xor_cancel : ∀ (x:bitstring{k}). { (x ^^ x) = zero }
```

Nous exporterons une version interprétée des axiomes, car l'utilisateur doit prouver une version interprétée des obligations de preuves.

```
Axiom pow_mul : ∀k (x y : Z),
  pow (pow (g_k) x) y = pow (g_k) (Zmult x y).
Axiom xor_comm : ∀k (x y : Bvector (Zabs_nat (k_ k))),
  xor x y = xor y x.
Axiom xor_assoc : ∀k (x y z : Bvector (Zabs_nat (k_ k))),
  xor (xor x y) z = xor x (xor y z).
Axiom xor_zero : ∀k (x : Bvector (Zabs_nat (k_ k))),
  xor x (zero_ k) = x.
Axiom xor_cancel : ∀ k (x : Bvector (Zabs_nat (k_ k))),
  xor x x = zero_ k.
```

Les axiomes et les paramètres peuvent être instanciés et prouvés par l'utilisateur.

5.2 Traduction des Jeux

Un jeu est un ensemble de procédures, pouvant s'appeler entre elles. La notion de jeu est directement définie dans EasyCrypt. Dans CertiCrypt, les appels de procédures se font à travers un environnement qui fait le lien entre le nom d'une procédure et sa définition. Définir un jeu dans CertiCrypt consiste à bien construire l'environnement pour chaque jeu. Nous expliquerons comment générer l'environnement à partir de la description de la preuve EasyCrypt.

Déclarations Globales :

```
adversary A(p1:t1) : type_retour_A   { type_o1 -> type_o2}
```

$\mathcal{A}$ | $\mathcal{A}$.params := (p1, t1) :: nil
| $\mathcal{A}$.body : cmd
| $\mathcal{A}$.res : type_retour_$\mathcal{A}$

$\mathcal{O}_{\mathcal{A}}$ | $\mathcal{O}_{\mathcal{A}}$.params : type_o1 :: nil
| $\mathcal{O}_{\mathcal{A}}$.res : type_o2

Environnement Initial :

init_env := add_env ($\mathcal{A}$, $\mathcal{A}$.params, $\mathcal{A}$.body, $\mathcal{A}$.res) env

Déclarations du jeu G :

```
fun H(p2:t2) : type_retour_H = { ... ; return r }
```

$G.H$ | $G.H$.params := (p2, t2)
| $G.H$.body := ...
| $G.H$.res := (r, type_retour_H)

$\mathcal{O}_{\mathcal{A}}$ | $\mathcal{O}_{\mathcal{A}}$.body := $\mathcal{O}_{\mathcal{A}}$.res $\leftarrow G.H(\mathcal{O}_{\mathcal{A}}$.params)

```
abs B = A {H}
```

$G.\mathcal{B}$ | $G.\mathcal{B}$.params
| $G.\mathcal{B}$.body := $G.\mathcal{B}$.res $\leftarrow \mathcal{A}(G.\mathcal{B}$.params)
| $G.\mathcal{B}$.res

Environnement du jeu G :

G_env := add_env ($G.\mathcal{B}$, $G.\mathcal{B}$.params, $G.\mathcal{B}$.body, $G.\mathcal{B}$.res)
(add_env ($G.H$, $G.H$.params, $G.H$.body, $G.H$.res)
(add_env ($\mathcal{O}_{\mathcal{A}}$, $\mathcal{O}_{\mathcal{A}}$.params, $\mathcal{O}_{\mathcal{A}}$.body, $\mathcal{O}_{\mathcal{A}}$.res))) init_env

Fig. 5.1. Correspondance entre les jeux EasyCrypt et CertiCrypt

Les preuves de EasyCrypt demandent des propriétés assez fortes sur les environnements en particulier sur le nom des variables et des procédures. La figure 5.1 donne une idée générale de la traduction des jeux de EasyCrypt vers CertiCrypt. Nous détaillerons toutes les étapes en nous servant de la preuve de Hashed ElGamal.

5.2.1 Déclaration Globale

Nous commençons par déclarer l'adversaire et l'environnement contenant l'adversaire. Dans EasyCrypt, nous déclarons le nom de l'adversaire, sa signature, ainsi que la signature des oracles qu'il peut appeler.

Dans la preuve de Hashed ElGamal nous avons :

```
adversary A1(pk:pkey) : message * message    { group → message}
adversary A2(pk:pkey,c:cipher) : bool        { group → message}
```

Nous commençons par déclarer les adversaires en indiquant le nom, les paramètres et la variable de retour. Dans la figure, nous notons ces déclarations par

$$\mathcal{A}\left|\begin{array}{l}\mathcal{A}.\mathsf{params} := (\mathsf{p1}, \mathsf{t1}) :: \mathsf{nil} \\ \mathcal{A}.\mathsf{body} : \mathsf{cmd} \\ \mathcal{A}.\mathsf{res} : \mathsf{type_retour_}\mathcal{A}\end{array}\right.$$

Pour l'adversaire A_1, nous gérèrerons le script Coq suivant :

```
Notation A1 := (Proc.mkP 36 (pkey :: nil) ((T.Pair message message))).
Notation pk_20 := (Var.Lvar pkey 2).
Definition A1_param : (var_decl (Proc.targs A1)) := dcons pk_20 dnil.
Variable A1_body : cmd.
Notation A_res_3 := (Var.Lvar (T.Pair message message) 3).
Definition A1_res : E.expr (T.Pair message message) := A1_res_3.
```

Pour les oracles que peut appeler l'adversaire, nous aller créer de nouveaux noms pour chaque oracle de chaque adversaire. Ce nom nous permettra lors des preuves d'équivalence entre chaque jeu d'avoir le même nom d'oracle pour deux adversaires. En effet la règle pour prouver l'équivalence entre deux adversaires a besoin des deux propriétés suivantes :

- Eq_adv_decl pour toutes les fonctions en dehors des oracles et des fonctions privées, les paramètres et résultats des oracles sont les mêmes dans les deux environnements.
- Eq_orcl_params pour tout oracle que l'adversaire peut appeler les paramètres dans les deux environnements sont égaux.

Dans la suite, nous générerons les jeux en Coq de manière à respecter les deux propriétés ci-dessus.

La déclaration des noms se fait de la manière suivante :

```
Notation param_1_for_orcl_1_of_A1 := (Var.Lvar(T.Usergroup) 4).
Notation oracle_1_for_A1 := (Proc.mkP 37 ((T.Usergroup) :: nil) message).
Definition params_oracle_1_for_A1 : var_decl (Proc.targs oracle_1_for_A1) :=
  dcons param_1_for_orcl_1_of_A1 dnil.
Definition res_oracle_1_for_A1 := (Var.Lvar message 5).
```

Nous déclarons ensuite l'environnement initial à partir d'un environnement env qui est donné en argument.

```
Variable env_adv0 : env.
Definition init_env :=
  (add_decl A1 A1_param A1_body A1_res
  (add_decl A2 A2_param A2_body A2_res env)).
```

cet environnement sera ensuite étendu dans chaque jeu.

5.2.2 Traduction des jeux

Pour traduire les procédures dans les jeux nous distinguons deux cas.

- Procédure définie : dans ce cas nous écrirons le corps de la fonction, en prenant garde à certains changements dus à la représentation des programmes entre EasyCrypt et CertiCrypt
- Procédure abstraite : dans ce cas nous instancions l'adversaire global avec le nom de l'oracle précédemment défini.

Fonctions définies

La traduction du corps des procédures ne présente pas de difficulté, hormis le fait que l'assignation est différente entre EasyCrypt et CertiCrypt, et que CertiCrypt ne peut pas mettre à jour directement les listes d'associations.

Par exemple dans la preuve de Hashed ElGamal les fonctions Hash et Main, sont définies dans le jeu IND-CPA par

```
fun H(x:group) : message = {
  var h : message = {0,1}^k;
  if ( ¬in_dom(x, L) ) { L[x] = h; };
  return L[x];
}
```

Dans CertiCrypt nous les traduisons par :

```
Notation x_24 := (Var.Lvar(T.User group)128).
Definition INDCPA_H_param : (var_decl (Proc.targs INDCPA_H)) :=
  dcons x_24 dnil.
Notation h_25 := (Var.Lvar message 130).
Definition INDCPA_H_body : cmd :=
  [h_25 <$- {0,1}^k;
  If(!(x_24 in_dom L_22)) then [L_22 : [{x_24}] <<- h_25] else nil].
Definition INDCPA_H_res : E.expr message := (L_22[x_24]).
```

Pour traduire l'affectation dans des paires, nous créons une variable intermédiaire de type T.pair, l'instruction $(m_1, m_2) \leftarrow A(pk)$ est convertie en

$$m_1m_2 \leftarrow A(pk);\ m_1 \leftarrow \mathsf{fst}\ m_1m_2;\ m_2 \leftarrow \mathsf{snd}\ m_1m_2.$$

Dans la suite, nous devrons toujours prendre en compte ce décalage, en particulier pour la tactique swap où nous donnons les indices des instructions à déplacer.

La mise à jour des listes d'associations n'existe pas dans CertiCrypt. Nous avons une notation nous permettant d'obtenir la même syntaxe, définie par :

$$\text{Notation } l : [\{x\}]\texttt{<<-}v := (l \leftarrow l.[\{x \texttt{«-} v\}])(\text{at level } 65).$$

Procédures Abstraites

Dans EasyCrypt les adversaires sont instantiés de manière concrète de la manière suivante :

```
abs A1 = A1 {H}
```

Ici le nom est le même, mais dans la figure 5.1 nous voyons que le nom peut être modifié ; ce qui pose problème car nous avons besoin de la propriété Eq_adv_decl. Nous allons alors encapsuler l'adversaire dans un *wrapper*, ce qui permettra d'avoir toujours le même nom entre tous les jeux.

Dans la figure 5.1 nous voyons que dans les jeux, les adversaires sont redéfinis de la manière suivante :

$$G.\mathcal{B}.\text{res} \leftarrow \mathcal{A}(G.\mathcal{B}.\text{params})$$

Nous obtenons en Coq le script correspondant :

```
Notation pk37 := (Var.Lvar pkey 141).
Definition INDCPA_A1_param : (var_decl (Proc.targs INDCPA_A1)) := dcons pk_37 dnil.
Notation res38 := (Var.Lvar (T.Pair message message) 142).
Definition INDCPA_oracle_1_for_A1_body : cmd :=
  [res_oracle_1_for_A1 ← INDCPA_H_A(param_1_for_orcl_1_of_A1)].
Definition INDCPA_A1_body : cmd := [res38 ← A1(pk37)].
Definition INDCPA_A1_res := E.Evar res38.
```

Nous faisons ensuite le lien entre les adversaires que nous avons définis de manière globale, et les oracles que nous définissions pour chaque jeu. Nous utilisons à nouveau les *wrappers*.

Nous obtenons alors le même nom pour tous les oracles, en redéfinissant pour chaque jeu le corps de l'oracle de la manière suivante :

$$\mathcal{O}_{\mathcal{A}}.\text{res} \leftarrow G.H(\mathcal{O}_{\mathcal{A}}.\text{params})$$

Et nous obtenons le script correspondant :

```
Notation pk37 := (Var.Lvar pkey 141).
Definition INDCPA_A1_param : (var_decl (Proc.targs INDCPA_A1)) := dcons pk_37 dnil.
Notation res38 := (Var.Lvar (T.Pair message message) 142).
Definition INDCPA_oracle_1_for_A1_body : cmd :=
  [res_oracle_1_for_A1 ← INDCPA_H_A(param_1_for_orcl_1_of_A1)].
Definition INDCPA_A1_body : cmd := [res38 ← A1(pk37)].
Definition INDCPA_A1_res := E.Evar res38.
```

Environnement

Une fois toutes les procédures et *wrapper* créés, nous construisons l'environnement

```
(* Environment for game INDCPA *)
Definition E_INDCPA :=
add_decls
(Build_decl INDCPA_Main INDCPA_Main_param
       INDCPA_Main_body INDCPA_Main_res ::
...
Build_decl INDCPA_A1 INDCPA_A1_param INDCPA_A1_body INDCPA_A1_res ::
Build_decl oracle_1_for_A1 params_oracle_1_for_A1
       INDCPA_oracle_1_for_A1_body res_oracle_1_for_A1 ::
Build_decl INDCPA_Enc INDCPA_Enc_param INDCPA_Enc_body INDCPA_Enc_res
...
:: nil) env_adv.
```

Propriétés et Hypothèses

Nous pouvons prouver automatiquement les propriétés Eq_adv_decl et Eq_orcl_params en parcourant l'ensemble des listes et en vérifiant la propriété pour tous les éléments.

Une fois l'environnement défini, nous pouvons poser les différentes hypothèses sur l'adversaire :

- l'adversaire est bien formé : l'adversaire appelle un adversaire bien formé ou bien une procédure qu'il a le droit d'appeler.
- l'adversaire est *lossless*, c'est-à-dire qu'il termine avec une probabilité 1. Nous avons besoin d'une hypothèse plus forte *strong lossless* dans la suite, nous devons garantir que toutes les sous-parties du programme sont *strong lossless*.

5.3 Preuve d'équivalence

Nous allons expliquer dans cette section, la sur-couche CertiCrypt permettant de manipuler les objets EasyCrypt. Nous commençons par redéfinir l'équivalence de CertiCrypt, en prenant des relations syntaxifiées. Cette syntaxification des relations va permettre de vérifier l'arbre de preuve de manière réflexive.

Dans la suite du chapitre, nous faisons abstraction du paramètre de sécurité pour clarifier les explications.

5.3.1 Jugements pRHL

Les jugements pRHL de EasyCrypt diffèrent par rapport à ceux de CertiCrypt, ils représentent une sous-classe de relations étant donné qu'ils sont syntaxifiés et ne permettent pas d'utiliser l'ordre supérieur. Ils permettent d'écrire moins de relations, mais la syntaxification nous permet d'écrire d'avantage de fonctions pour travailler sur les prédicats. Dans CertiCrypt le type des relations a le type suivant

$$\mathsf{mem_rel} : \mathsf{Mem} \to \mathsf{Mem} \to \mathsf{Prop}$$

Par exemple, la post-condition montrant que la variable x dans le programme de gauche est équivalente à la variable y dans le programme de droite, s'écrit en Coq

$$\mathsf{fun}\ m_1\ m_2\ \Rightarrow m_1\ x = m_2\ y.$$

Étant de type Prop, cette relation ne facilite pas les calculs, en particulier les calculs de wp que nous utilisons intensivement dans EasyCrypt. En effet les *binding* de Coq ne facilitent pas l'écriture de fonction comme la substitution. Dans EasyCrypt nous écrivons cette relation de la manière suivante

$$x\langle 1\rangle\ \ ==\ \ y\langle 2\rangle.$$

Une partie du travail de la vérification de preuve a été de formaliser en Coq les représentations des types syntaxifiés de EasyCrypt et les différentes opérations en ajoutant une couche à CertiCrypt.

Le type des expressions change par rapport à CertiCrypt, les expressions sont relationnelles, c'est à dire que nous pouvons designer directement les variables des programmes en utilisant le type suivant :

$$\begin{array}{l}
\mathsf{Inductive}\ \mathsf{Var}(t : \mathcal{T}) : \mathsf{Type} := \\
\mid \mathsf{Left}\ \ : \mathcal{V}_t \to \mathsf{Var}\ t \\
\mid \mathsf{Right} : \mathcal{V}_t \to \mathsf{Var}\ t \\
\mid \mathsf{Logic} : \mathcal{V}_t \to \mathsf{Var}\ t
\end{array}$$

Nous notons les variables de programme $x\langle 1\rangle$ la variable (Left x) et $x\langle 2\rangle$ la variable (Right x). Les variables logiques ne sont pas des variables de programme et permettent de désigner les variables introduites par les quantificateurs logiques.

Le type des expressions utilisées dans les prédicats est le même que dans CertiCrypt, mais les variables sont remplacées par les variables relationnelles Var définies ci-dessus.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{Inductive}\ \mathsf{Exp} : \mathcal{T} \to \mathsf{Type} :=} \\
\mid \mathsf{Ecnst} & :> & \forall t, \mathcal{C}_t \to \mathsf{Exp}_t \\
\mid \mathsf{Evar} & :> & \forall t, \mathsf{Var}_t \to \mathsf{Exp}_t \\
\mid \mathsf{Eapp} & : & \forall op, \mathsf{Exp}^{\star}_{(\mathsf{targs}\ op)} \to \mathsf{Exp}_{(\mathsf{tres}\ op)} \\
\mid \mathsf{Eexists} & : & \forall t, \mathcal{V}_t \to \mathsf{Exp}_{\mathsf{Bool}} \to \mathsf{Exp}_{(\mathsf{List}\ t)} \to \mathsf{Exp}_{\mathsf{Bool}} \\
\mid \mathsf{Eforall} & : & \forall t, \mathcal{V}_t \to \mathsf{Exp}_{\mathsf{Bool}} \to \mathsf{Exp}_{(\mathsf{List}\ t)} \to \mathsf{Exp}_{\mathsf{Bool}} \\
\mid \mathsf{Efind} & : & \forall t, \mathcal{V}_t \to \mathsf{Exp}_{\mathsf{Bool}} \to \mathsf{Exp}_{(\mathsf{List}\ t)} \to \mathsf{Exp}_t.
\end{array}$$

où $\mathcal{C}_t$ est le type des constantes de type t. Les constructeurs Eexists, Eforall et Efind n'apparaissent pas dans EasyCrypt, mais nous avons besoin dans la suite d'effectuer des transformations entre les expressions relationnelles Exp et les expressions utilisées dans les programmes.

Les relations sont définies en utilisant des expressions relationnelles de type bool :

```
Inductive Pred : Type :=
| Ptrue    : Pred
| Pfalse   : Pred
| Pnot     : Pred → Pred
| Pand     : Pred → Pred → Pred
| Por      : Pred → Pred → Pred
| Pimplies : Pred → Pred → Pred
| Piff     : Pred → Pred → Pred
| Pif      : Exp_Bool → Pred → Pred → Pred
| Pforall  : ∀t, V_t → Pred → Pred
| Pexists  : ∀t, V_t → Pred → Pred
| Plet     : ∀t, V_t → Exp_t → Pred → Pred
| Pexp     : Exp_Bool → Pred
| Peq      : ∀t, Exp_t → Exp_t → Pred
| Ple      : Exp_Z → Exp_Z → Pred
| Plt      : Exp_Z → Exp_Z → Pred
```

L'évaluation des expressions se fait dans les deux mémoires et une liste d'association utilisée pour évaluer les variables logiques. La liste a le type suivant :

$$\mathsf{lmem} : \mathsf{list}\ (t : \mathsf{Type}, \mathsf{var} : \mathcal{V}_t, \mathsf{val} : \mathsf{interp}\ k\ t)$$

L'évaluation des variables se décompose en deux cas :

- les variables de programme sont évaluées dans leurs mémoires respectives
- les variables logiques sont évaluées en utilisant la liste d'association.

La fonction d'évaluation des expressions eeval a le type suivant :

$$\mathsf{eeval}\ (m_1\ m_2 : \mathsf{Mem})\ (l : \mathsf{lmem})\ (e : \mathsf{Exp}_t) : \mathsf{interp}\ k\ t$$

Nous utiliserons la notation suivant $[\![e]\!]_{m_1,m_2,l}$.

La fonction est totale, et si une variable logique n'apparaît pas dans la liste, la fonction renverra une valeur par défaut. Dans la preuve de correction ce cas peut poser un problème, mais les hypothèses nous garantissent que ce cas n'apparaît pas.

La fonction d'évaluation des prédicats peval a le type suivant :

$$\mathsf{peval}\ (m_1\ m_2 : \mathsf{Mem})\ (l : \mathsf{lmem})\ (p : \mathsf{Pred}) : \mathsf{Prop}$$

Les prédicats Ptrue, Pfalse s'évaluent simplement en True, False. Les Pnot, Pand, Por, Pimplies, Piff s'évaluent en utilisant les constructeurs logiques $\sim, \wedge, \vee, \rightarrow, \leftrightarrow$ de Coq. Le prédicat conditionnel utilise le if then else de Coq :

$$\mathsf{peval}\ m_1\ m_2\ l\ (\mathsf{Pif}\ e\ p_t\ p_f)$$

s'évalue en

$$\mathsf{if}\ \mathsf{eeval}\ m_1\ m_2\ l\ e\ \mathsf{then}\ \mathsf{peval}\ m_1\ m_2\ l\ p_t\ \mathsf{else}\ \mathsf{peval}\ m_1\ m_2\ l\ p_f$$

Les prédicats Pforall $t\ X\ P$, Pexists $t\ X\ P$ s'évaluent en utilisant les $\forall, \exists$ de Coq tout en étendant, la liste d'associations :

$$\forall\ x : \mathsf{interp}\ t, \mathsf{peval}\ m_1\ m_2\ ((X,x) :: l)\ P$$
$$\exists\ x : \mathsf{interp}\ t, \mathsf{peval}\ m_1\ m_2\ ((X,x) :: l)\ P$$

où interp est la fonction d'interprétation des types de CertiCrypt dans Coq. P est interprété dans les mémoires m_1 et m_2, et dans la liste l qui est étendue par le couple (X, x).

De la même manière Plet $t\ X\ e\ P$ s'évalue en

$$\mathsf{peval}\ m_1\ m_2\ ((X, [\![e]\!]_{m_1,m_2,l}) :: l)\ P$$

sauf que la valeur ajoutée dans la liste correspond à l'évaluation de e. Pour finir Pexp e s'évalue en utilisant l'égalité de Coq $[\![e]\!]_{m_1,m_2,l} = \mathsf{true}$.

Etant donné que les prédicats manipulés sont clos, nous définissons la fonction ipred qui appelle la fonction peval avec une liste d'associations vide. Toutes les preuves sont faites sur peval pour toutes les listes puis sont instanciées pour une liste vide.

Nous définissons la fonction substitution, pour les expressions puis pour les prédicats. La signature de la fonction de substitution pour les expressions a le type suivant :

$$\mathsf{esubst} : \mathsf{Var}_{t'} \to \mathsf{Exp}_{t'} \to \mathsf{Exp}_t \to \mathsf{Exp}_t$$

où esubst $v\ e'\ e$ se note $e\{x \leftarrow e'\}$.

Le fait que EasyCrypt génère les noms, nous place dans une situation où il n'y a pas de capture de variables. Nous vérifions cette propriété par le calcul. Par exemple le lemme de correction de la fonction de substitution pour les expressions s'énonce ainsi :

$$\forall k\ t\ t'\ m_1\ m_2\ (v : \mathsf{Var}'_t)\ (e' : \mathsf{Exp}'_t)\ (e : \mathsf{Exp}_t)\ l,$$
$$\mathsf{fv}\ e' \cap \mathsf{bind}\ e = \varnothing \to$$
$$[\![e\{v \leftarrow e'\}]\!]_{m_1,m_2,l} = [\![e]\!]_{m_1\{v \leftarrow [\![e']\!]_{m_1,m_2,l}\}, m_2\{v \leftarrow [\![e']\!]_{m_1,m_2,l}\}, l\{v \leftarrow [\![e']\!]_{m_1,m_2,l}\}}$$

La fonction fv calcule l'ensemble des variables logiques libres d'une expression et bind calcule les variables liées par les quantificateurs (let, $\forall$ et $\exists$).

La condition fv $e' \cap$ bind $e = \varnothing$ permet de vérifier que les variables logiques libres de e' ne sont pas liées dans e ce qui assure qu'il n'y a pas de capture de variables. Nous évitons ainsi l'utilisation des indices de Brujin, qui aurait compliqué les preuves.

La contre-partie est de toujours avoir à vérifier les conditions à chaque fois que nous utilisons la substitution. Pour un utilisateur qui utilise directement CertiCrypt, cela peut poser problème, mais dans notre cas les conditions sont vérifiées automatiquement par le calcul et EasyCrypt assure que ces conditions sont toujours vraies.

Nous écrivons et prouvons de la même manière la substitution sur les prédicats.

5.3.2 Arbre de preuve

Nous allons expliquer dans cette section comment l'arbre de preuve est formalisé en Coq. Nous introduirons les différentes notions qui permettront de retrouver la preuve effectuée dans EasyCrypt et de la certifier en Coq.

L'arbre de preuve peut être vu comme un témoin, ou la description d'une preuve pour le jugement pRHL donné. L'arbre de preuve contient toutes les informations nécessaires pour rejouer la preuve. Dans CertiCrypt nous allons rejouer la preuve en certifiant presque toutes les étapes. Nous montrons que si la vérification est correcte, alors le jugement pRHL est valide.

Nous allons effectuer une preuve par réflection pour vérifier l'arbre, ce qui implique que tous les objets de la preuve et le jugement pRHL, se calculent. Tous nos objets sont alors syntaxifiés. L'arbre de preuve a le type suivant :

```
Inductive RhlRule : Type :=
| Rnil          : RhlRule
| Rsub_pre      : Pred → RhlRule → RhlRule
| Rsub_post     : Pred → RhlRule → RhlRule
| Rcase         : Pred → RhlRule → RhlRule → RhlRule
| RcondLeft     : RhlRule → RhlRule → RhlRule
| RcondRight    : RhlRule → RhlRule → RhlRule
| RcondBoth     : RhlRule → RhlRule → RhlRule
| Rapp          : ℕ → ℕ → Pred → RhlRule → RhlRule → RhlRule
| Rpre_false    : RhlRule
| Rpost_true    : RhlRule
| Rnot_modify   : Pred → Pred → RhlRule → RhlRule
| Rwp_asgn      : RhlRule → RhlRule
| Rwp_simpl     : RhlRule → RhlRule
| Rwp_call      : list 𝒱 → list 𝒱 → EquivFun_info → RhlRule → RhlRule
| Rwp_bij       : 𝒱 → TypeRand → RhlRule → RhlRule
| Rwp_disj      : 𝒱 → Bool → RhlRule → RhlRule
| RinlineLeft   : cmd → RhlRule → RhlRule
| RinlineRight  : cmd → RhlRule → RhlRule
| Rderandomize  : cmd → cmd → RhlRule → RhlRule
| Rswap         : Bool → ℕ → ℕ → Bool → ℕ → RhlRule → RhlRule
```

Ce type est le même dans EasyCrypt et dans Coq, ce qui nous permet d'avoir une tradution directe. Chaque noeud de l'arbre représente l'application d'une règle. Nous commencerons par expliquer comment l'arbre de preuve est vérifié en Coq et nous détaillerons ensuite les règles une par une.

5.3.3 Vérification dans CertiCrypt

Chaque règle est une étape de la preuve, nous nous servons de la logique pRHL pour prouver chaque étape en Coq. Nous utilisons un vérificateur prouvé de manière réflexive qui permettra d'exécuter la vérification de l'arbre de preuve et d'obtenir automatiquement un certificat.

Pour formaliser la vérification de l'arbre, nous définissons la fonction check_proof qui a le type suivant :

```
Definition check_proof (P Q : pred) (c1 c2 : cmd) (d : RhlRule) : deriv_status
```

Le type deriv_status est défini de la manière suivante :

```
Inductive deriv_status : Type :=
| DS_cond : list Pred → deriv_status
| DS_error : deriv_status
```

et permet de distinguer les deux cas de la vérification, soit l'arbre est correct et nous stockons la liste des obligations à vérifier, soit il y a une erreur et dans ce cas, il y a soit un *bug* dans EasyCrypt ou dans la traduction.

En Coq, l'énoncé du lemme pour prouver les jugements pRHL étant donné un arbre de preuve est le suivant :

```
Lemma check_proof_correct :
∀(d : RhlRule)(P : Pred)(c1 c2 : cmd)(Q : pred)(l : list pred),
check_proof P c1 c2 Q d = DS_cond l →
interp_cond l →
c1 ∼ c2 : P ⇒ Q.
```

où interp_cond évalue chacun des prédicats dans Prop.

La vérification produit un ensemble d'obligations de preuves que nous ne pouvons pas vérifier automatiquement en Coq pour le moment. Nous prouvons l'équivalence en vérifiant que l'arbre de preuve est correct sous l'hypothèse que l'ensemble des obligations de preuves générées sont vraies.

Pour appliquer le lemme dans les preuves, si nous voulons prouver que $c_1 \sim c_2 : P \Rightarrow Q$ avec l'arbre de preuve d, nous commençons par exécuter la fonction check_proof P c_1 c_2 Q d, et nous appliquons le lemme ci dessus avec la liste l qui vient d'être calculée. Nous laissons à l'utilisateur la preuve de interp_cond l qui est laissée admise.

Pour le moment, nous ne sommes pas sûrs que la condition admise en Coq est bien la condition qui est vérifiée par les prouveurs SMT dans EasyCrypt. Pour augmenter le niveau de sécurité nous pouvons poser un axiome pour chaque formule vérifiées par les prouveurs SMT dans EasyCrypt, et prouver ensuite les obligations de preuves générées dans CertiCrypt à partir de ces axiomes. Le problème est que nous ne sommes pas certains que les simplifications des expressions se font de la même manière dans EasyCrypt soit dans CertiCrypt.

5.3.4 Règles de vérification de l'arbre de preuve

Preuve des règles de base

L'ensemble des règles de la figure 5.3.4 sont des conséquences logiques des règles prouvées dans CertiCrypt. Nous détaillerons la preuve de la règle Rnot_modify.

La règle Rnot_modify permet de supprimer des sous parties de la post condition qui ne sont pas modifiées par c_1 et c_2. Le calcul de M est fait par EasyCrypt et correspond à la condition qui n'est pas modifiée par les programmes. La règle

$$\frac{P \to Q}{[] \sim [] : P \Rightarrow Q}[\mathsf{Rnil}]$$

$$\frac{c_1 \sim c_2 : P' \Rightarrow Q \quad P \to P'}{c_1 \sim c_2 : P \Rightarrow Q}[\mathsf{Rsub_pre}\ P']$$

$$\frac{c_1 \sim c_2 : P \Rightarrow Q' \quad Q' \to Q}{c_1 \sim c_2 : P \Rightarrow Q}[\mathsf{Rsub_post}\ P']$$

$$\frac{\mathsf{is_dec}\ R \quad c_1 \sim c_2 : P \wedge R \Rightarrow Q \quad c_1 \sim c_2 : P \wedge \neg R \Rightarrow Q}{c_1 \sim c_2 : P \Rightarrow Q}[\mathsf{Rcase}\ R]$$

$$\frac{c_t{+\!+}c_1' \sim c_2 : P \wedge e \Rightarrow Q \quad c_f{+\!+}c_1' \sim c_2 : P \wedge \neg e \Rightarrow Q}{\mathsf{if}\ e\ \mathsf{then}\ c_t\ \mathsf{else}\ c_f :: c_1' \sim c_2 : P \Rightarrow Q}[\mathsf{RcondLeft}]$$

$$\frac{c_1 \sim c_t{+\!+}c_2' : P \wedge e \Rightarrow Q \quad c_1 \sim c_f{+\!+}c_2' : P \wedge \neg e \Rightarrow Q}{c_1 \sim \mathsf{if}\ e\ \mathsf{then}\ c_t\ \mathsf{else}\ c_f :: c_2' : P \Rightarrow Q}[\mathsf{RcondRight}]$$

$$\frac{\begin{array}{c} c_{t1}{+\!+}c_1' \sim c_{t2}{+\!+}c_2' : P \wedge e_1 \wedge e_2 \Rightarrow Q \\ c_{f1}{+\!+}c_1' \sim c_{f2}{+\!+}c_2' : P \wedge \neg e_1 \wedge \neg e_2 \Rightarrow Q \quad P \to (e_1 = e_2) \end{array}}{\mathsf{if}\ e_1\ \mathsf{then}\ c_{t1}\ \mathsf{else}\ c_{f1} :: c_1' \sim \mathsf{if}\ e_2\ \mathsf{then}\ c_{t2}\ \mathsf{else}\ c_{f2} :: c_2' : P \Rightarrow Q}[\mathsf{RcondBoth}]$$

$$\frac{c_{h1} \sim c_{h2} : P \Rightarrow R \quad c_{t1} \sim c_{t2} : R \Rightarrow Q \quad |c_{h1}| = n_1 \quad |c_{h2}| = n_2}{\vdash c_{h1}{+\!+}c_{t1} \sim c_{h2}{+\!+}c_{t2} : P \Rightarrow Q}[\mathsf{Rapp}\ n_1\ n_2\ R]$$

$$\frac{P \Rightarrow \mathsf{False}}{c_1 \sim c_2 : P \Rightarrow Q}[\mathsf{Rpre_false}]$$

$$\frac{\mathsf{is_lossless}\ c_1 \quad \mathsf{is_lossless}\ c_2}{c_1 \sim c_2 : P \Rightarrow \mathsf{True}}[\mathsf{Rpost_true}]$$

$$\frac{\begin{array}{c} d \vdash c_1 \sim c_2 : P \Rightarrow Q' \quad (X_1, X_2) := \mathsf{pdepend}\ P \\ X_1 \cap \mathsf{modify}\ c_1 = \varnothing \quad X_2 \cap \mathsf{modify}\ c_2 = \varnothing \\ P \to M \quad M \to (Q' \to Q) \end{array}}{c_1 \sim c_2 : P \Rightarrow Q}[\mathsf{Rnot_modify}\ M\ Q']$$

Fig. 5.2. Règles de base

NotModify permet de raisonner sur un invariant M qui n'est pas modifié par c_1 et c_2 alors si nous ajoutons M en pré-condition il sera vrai en post-condition :

$$\frac{\begin{array}{c} c_1 \sim c_2 : P \Rightarrow Q \quad X_1 \cap \mathsf{modify}\ c_1 = \varnothing \quad X_2 \cap \mathsf{modify}\ c_2 = \varnothing \\ \mathsf{depend_only_rel}\ M\ X_1\ X_2 \end{array}}{c_1 \sim c_2 : P \wedge M \Rightarrow Q \wedge M}[\mathsf{NotModify}]$$

où modify calcule les variables modifiées par un programme, pdepend calcule une sur-approximation des variables dont la relation dépend. Le prédicat depend_only_rel est défini par

$$\begin{array}{l} \forall P\ X_1\ X_2\ m_1\ m_2\ m_1'\ m_2', \\ m_1 =_{X_1} m_1' \to m_2 =_{X_2} m_2' \to \\ P\ m_1\ m_2 \to P\ m_1'\ m_2' \end{array}$$

il exprime que le prédicat P ne dépend que des variables X_1 dans la mémoire m_1 et des variables X_2 dans la mémoire m_2. Nous prouvons la propriété sur depend_only_rel et pdepend

$$\forall P, \mathsf{depend_only_rel}\ P\ (\mathsf{fst}\ (\mathsf{pdepend}\ p))\ (\mathsf{snd}\ (\mathsf{pdepend}\ p))$$

Nous donnons une idée de la preuve de la règle Rnot_Modify en Coq :

$$\dfrac{\dfrac{\vdots}{c_1 \sim c_2 : P \wedge M \Rightarrow Q' \wedge M}\,[\mathsf{Modify}] \quad \dfrac{P \to M}{P \to P \wedge M}\,[\mathsf{tauto}] \quad \dfrac{M \to (Q' \to Q)}{Q' \wedge M \to Q}\,[\mathsf{tauto}]}{c_1 \sim c_2 : P \Rightarrow Q}\,[\mathsf{Sub}]$$

Les hypothèses de Modify sont vérifiées par les conditions de Rnot_modify.

Par rapport à CertiCrypt, certaines propriétés sont prouvées par le calcul grâce à la syntaxification. Nous pouvons prouver par le calcul les propriétés qui utilisent pdepend ou bien tester si un prédicat est décidable (is_dec).

WP pour les instructions déterministes : Rwp_simpl et Rwp_asgn

La règle Rwp_simpl calcule le wp sur la partie déterministe du programme. Elle utilise la *self-composition* c'est à dire que nous utilisons le wp sur chacun des programmes et nous composons ensuite les deux conditions générées.

Pour montrer des propriétés sur un programme probabiliste, nous utilisons le prédicat range défini dans le chapitre 2 de type

$$\mathsf{range}\ A\ (P : A \to \mathsf{Prop})(d : \mathsf{Distr}\ A) : \mathsf{Prop}.$$

Nous appliquons le prédicat range aux mémoires. range permet de montrer que le prédicat P est valide dans tous les états qui ont une probabilité non nulle dans la distribution d.

Nous prouvons le wp sur chaque programme c'est à dire utilisant la logique de Hoare dans le cas non relationnel. Les triplets de Hoare sont définis en utilisant le prédicat range. Étant donné que les relations s'exécutent sur deux mémoires nous sommes obligés de distinguer deux définitions des triplés de Hoare pour chaque programme :

$$\forall\ m_1\ m_2, \mathsf{ipred}\ P\ m_1\ m_2 \to \mathsf{range}\ (\mathsf{fun}\ m \Rightarrow \mathsf{ipred}\ Q\ m\ m_2)\ [\![c_1]\!]_{m_1}$$

pour le programme de gauche, noté $\{P\}c_1\{Q\}_l$ et

$$\forall\ m_1\ m_2, \mathsf{ipred}\ P\ m_1\ m_2 \to \mathsf{range}\ (\mathsf{fun}\ m \Rightarrow \mathsf{ipred}\ Q\ m_1\ m)\ [\![c_2]\!]_{m_2}$$

pour le programme de droite, noté $\{P\}c_2\{Q\}_r$.

Le wp prend en paramètre la programme c et une post-condition Q et génère une condition Q' et un programme c' :

$$\mathsf{wp}_{side} i\ c\ Q = (c', Q')$$

c' est un préfixe de c, c'est à dire qu'il existe une commande c'' telle que $c = c'{+}{+}c''$. Le wp se calcule sur la partie déterministe c'' de c. Nous prouvons que

$$\begin{array}{l}\mathsf{wp}_{side}\ c\ Q = (c', Q') \rightarrow \\ \exists c'', c = c''{+}{+}c' \wedge \{Q'\}c'\{Q\}_{side} \wedge \mathsf{lossless}\ c'\end{array}$$

Le wp est paramétré par *side* qui indique si le programme est du côté gauche ou du côté droit.

Pour faire le lien entre ces triplets de Hoare sur un seul programme et une équivalence, nous prouvons les deux lemmes suivants

$$\frac{\{P\}c_1\{Q\}_l \quad \mathsf{lossless}\ c_1}{c_1 \sim \mathsf{nil} : P \Rightarrow Q}[\mathsf{Hoare_l}] \quad \text{et} \quad \frac{\{P\}c_2\{Q\}_r \quad \mathsf{lossless}\ c_2}{\mathsf{nil} \sim c_2 : P \Rightarrow Q}[\mathsf{Hoare_r}]$$

Pour appliquer ces deux règles nous avons besoin de vérifier que les programmes c_1 ou c_2 sont lossless, c'est-à-dire qu'ils terminent avec une probabilité 1.

Pour prouver le wp relationnel, nous appliquons la *self-composition* en composant les deux wp. Nous écrivons les programmes c_1 et c_2 sous la forme $c_1'{+}{+}c_1''$ et $c_2'{+}{+}c_2''$ où c_1'' et c_2'' sont les parties déterministes de c_1 et c_2.

$$\frac{\dfrac{\vdots}{c_1' \sim c_2' : P \Rightarrow \mathsf{wp}\ c_1\ (\mathsf{wp}\ c_2\ Q)} \qquad \dfrac{\dfrac{\{\mathsf{wp}\ c_1\ (\mathsf{wp}\ c_2\ Q)\}c_1''\{\mathsf{wp}\ c_2\ Q\}_l \quad \mathsf{lossless}\ c_1''}{c_1'' \sim \mathsf{nil} : \mathsf{wp}\ c_1\ (\mathsf{wp}\ c_2\ Q) \Rightarrow \mathsf{wp}\ c_2\ Q}[\mathsf{Hoare_l}] \quad \dfrac{\{\mathsf{wp}\ c_2\ Q\}c_2''\{Q\}_r \quad \mathsf{lossless}\ c_2''}{\mathsf{nil} \sim c_2'' : \mathsf{wp}\ c_1\ c_2\ Q \Rightarrow Q}[\mathsf{Hoare_r}]}{c_1'' \sim c_2'' : \mathsf{wp}\ c_1\ (\mathsf{wp}\ c_2\ Q) \Rightarrow Q}[\mathsf{Rapp}]}{c_1'{+}{+}c_1'' \sim c_2'{+}{+}c_2'' : P \Rightarrow Q}[\mathsf{Rapp}]$$

L'arbre de dérivation inclus dans le certificat du wp sert à prouver l'équivalence $c_1' \sim c_2' : P \Rightarrow \mathsf{wp}\ c_1\ (\mathsf{wp}\ c_2\ Q)$. Les conditions lossless c_1'' et lossless c_2'' se prouvent automatiquement étant donné que ces programmes ne contiennent que des affectations et des instructions conditionnelles.

La règle Rwp_asgn est une variante de Rwp_simpl qui ne traite pas les instructions conditionnelles.

WP pour les appels de procédure : Rwp_call

La règle Rwp_call permet de calculer le wp, quand les deux programmes se terminent par un appel de procédure. La fonctions wp_call prend une paire de programmes (c_1, c_2) de la forme

$$(c_1'{+}{+}(x_1 \leftarrow f_1(\overline{a_1})), c_2'{+}{+}(x_2 \leftarrow f_2(\overline{a_2}))$$

Le calcul de wp a besoin d'une spécification sur le corps des fonctions f_1 et f_2. De plus, nous avons besoin des deux propriétés suivantes :

- only_params_or_global les variables locales de la pré-condition sont incluses dans les paramètres des fonctions
- only_global_or_res les variables locales de la post-condition sont soit l'ensemble vide, soit la variable de retour.

Nous encapsulons toutes ces informations dans un Record qui a le type suivant :

```
Record EquivFun t P (f1 f2 : Proc.proc t) Q : Type := {
 res1  : V_t;
 res2  : V_t;
 equiv : f1.body ~ f2.body : P ⇒ wp_return f1 f2 Q res1 res2;
 pre   : only_params_or_global f1 f2 P;
 post  : only_global_or_res f1 f2 Q res1 res2}.
```

Les $\mathsf{res_i}$ sont des variables qui représentent la valeur de retour de la fonction f_i dans la post-condition. L'objet contient une preuve d'équivalence sur le corps des deux fonctions, pour P et Q sauf que la variable $\mathsf{res_i}$ est substituée par l'expression de retour $f_i.\mathsf{res}$ (wp_return).

Pour chaque équivalence nous créons un objet de type EquivFun_info qui fait partie des arguments de wp_call :

```
Inductive EquivFun_info : Type :=
  Mk_EquivFun_info : ∀t (P Q : Pred)(f1 f2 : Proc.proc t),
  EquivFun P f1 f2 Q → EquivFun_info
```

Le wp est défini de la manière suivante :

$\mathsf{wp_call}\ l_1\ l_2\ (\mathsf{info} : \mathsf{EquivFun_info})\ Q :=$

$$
\begin{array}{l}
P_f\{\overrightarrow{f_1.\mathsf{params}} \leftarrow \overrightarrow{a_1}\}\{\overrightarrow{f_2.\mathsf{params}} \leftarrow \overrightarrow{a_2}\} \wedge \\
\forall\ l_1\ l_2, (Q_f \rightarrow Q\{x_1 \leftarrow f_1.\mathsf{res}_1\langle 1\rangle\}\{x_2 \leftarrow f_2.\mathsf{res}_2\langle 2\rangle\}) \\
\quad \{f_1.\mathsf{res}_1\langle 1\rangle \leftarrow \mathsf{hd}\ l_1\}\{f_2.\mathsf{res}_2\langle 2\rangle \leftarrow \mathsf{hd}\ l_2\}\{\mathsf{modifiy}\ f_1\langle 1\rangle \leftarrow \mathsf{tl}\ l_1\}\{\mathsf{modifiy}\ f_2\langle 2\rangle \leftarrow \mathsf{tl}\ l_2\}
\end{array}
$$

P_f et Q_f proviennent de info. Les deux listes de variables fraîches l_1 et l_2 créées par EasyCrypt permettent de donner de nouveaux noms aux variables modifiées par les fonctions et éviter que ces variables ne soient ensuite substituées.

Cette règle n'existe pas dans CertiCrypt. La spécification des fonctions dans le EquivFun est assez faible. La condition que nous obtenons est bien plus forte car nous enrichissons la spécification de la fonction avec la post condition.

Nous prouvons le wp à partir de la règle equiv_call de CertiCrypt :

$$
\frac{f_1.\mathsf{body} \sim f_2.\mathsf{body} : P_f \Rightarrow Q_f \quad P \rightarrow P_f\{\overrightarrow{f_1.\mathsf{params}} \leftarrow \overrightarrow{a_1}\}\{\overrightarrow{f_2.\mathsf{params}} \leftarrow \overrightarrow{a_2}\} \quad P \rightarrow Q_f\{x_1 \leftarrow f_1.\mathsf{res}_1\}\{x_2 \leftarrow f_2.\mathsf{res}_2\} \rightarrow Q}{x_1 \leftarrow f_1(a_1) \sim x_2 \leftarrow f_2(a_2) : P \Rightarrow Q}
$$

WP relationnel pour les tirages aléatoires : Rwp_bij

Les deux programmes (c_1, c_2) finissent par deux instructions aléatoires de la forme

$$(c_1' {+}{+} (y_1 \overset{\$}{\leftarrow} d_1), c_2' {+}{+} (y_2 \overset{\$}{\leftarrow} d_2))$$

Les variables y_1 et y_2 doivent avoir le même type.

Dans CertiCrypt la règle générale pour montrer l'équivalence entre deux tirages aléatoires est donnée par la règle equiv_random_permut :

$$x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : (d_1 \simeq_f d_2 \wedge \forall v \in d_1, Q\{x_1 \leftarrow f\ v\}\{x_2 \leftarrow v\}) \Rightarrow Q$$

où $d_1 \simeq_f d_2$ signifie que d_1 est une permutation de d_2 en utilisant la fonction f. La difficulté de cette étape est de trouver la fonction f et prouver que c'est une permutation.

La plupart du temps, la permutation est l'identité, et nous pouvons utiliser la règle equiv_random qui est un cas particulier de la règle ci dessus :

$$x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : (d_1 = d_2 \wedge \forall v \in d_1, Q\{x_1 \leftarrow v\}\{x_2 \leftarrow v\}) \Rightarrow Q$$

Nous donnerons deux exemples d'application de ces règles. Dans le premier exemple nous prouvons l'équivalence entre les deux programmes suivants :

$$x \xleftarrow{\$} \mathbb{Z}_q \sim y \xleftarrow{\$} \mathbb{Z}_q : \mathsf{Ptrue} \Rightarrow g^x = g^y$$

Après application de la règle et simplification, l'équivalence se prouve en vérifiant la condition suivante :

$$\forall v \in \mathbb{Z}_g, g^v = g^v$$

Dans le deuxième exemple, nous multiplions la variable g^x par une constante β.

$$x \xleftarrow{\$} \mathbb{Z}_g \sim y \xleftarrow{\$} \mathbb{Z}_g : \mathsf{Ptrue} \Rightarrow g^x \times \beta = g^y$$

Nous appliquons la règle equiv_random_permut avec la permutation suivante :

$$\mathsf{fun}\ x \Rightarrow (x - \mathsf{log}\ \beta)\ \mathsf{mod}\ q$$

après simplification nous obtenons la condition suivante :

$$\forall v \in \mathbb{Z}_g, (g^{v - \mathsf{log}\ \beta} \times \beta)\langle 1 \rangle = g^v \langle 2 \rangle$$

La règle equiv_random ne pose pas de problème dans EasyCrypt. Par contre pour equiv_random_permut, nous devons montrer que d_1 est une permutation de d_2. Nous utiliserons le fait que la fonction f est une bijection entre les deux domaines.

Le certificat de EasyCrypt contient le nom de la variable, que nous utiliserons dans le quantificateur universel. L'objet TypeRand contient les informations nécessaires pour vérifier la permutation.

```
Inductive TypeRand : Type :=
| RIid          : TypeRand
| RIidempotent : ∀t, Exp_t → TypeRand
| RIbij         : ∀t, Exp_t → Exp_t → TypeRand
```

- RIid est le cas où les deux distributions sont identiques et nous appelons la règle equiv_random.
- RIidempotant f signifie que la fonction f est une involution (e.g. $f \circ f = \mathsf{id}$). Cette propriété suffit pour montrer que les deux supports sont des permutations et permet d'appeler la règle equiv_random_permut.

– $\mathsf{RIbij}\ f\ f^{-1}$, les fonctions f et f^{-1} sont des bijections et s'annulent (e.g. $f \circ f^{-1} = \mathsf{id}$). Par exemple, pour la transition utilisant la multiplication dans un groupe cyclique que nous avons vue au dessus, nous utilisons les fonctions suivantes :

$$f(z) = z + \mathsf{log}\ \beta \quad \text{et} \quad f^{-1}(z) = z - \mathsf{log}\ \beta$$

Nous prouvons que les deux fonctions sont des bijections en prouvant que $\forall z, z + \mathsf{log}\ \beta - \mathsf{log}\ \beta = z$.

Nous distinguons les trois cas, mais seule la dernière règle suffit. Les règles Rid et RIidempotant sont des cas particuliers de RIbij mais produisent de plus petits certificats.

Nous devons vérifier d'autres propriétés pour prouver la permutation. Dans le cas où les distributions sont des intervalles uniformes $[l_1..r_1]$ et $[l_2..r_2]$ nous générons la condition suivante pour le cas RIbij :

$$\begin{array}{l} l_1 = l_2 \wedge r_1 = r_2 \wedge \\ l_1 \leq r_1 \wedge \\ \forall\ v, l_1 \leq v \leq r_1 \rightarrow \\ \quad (l_1 \leq f(v) \leq r_1 \wedge l_1 \leq f^{-1}(v) \leq r_1 \wedge \\ \quad f^{-1}(f(v)) = v \wedge f(f^{-1}(v)) = v \wedge \\ \quad Q\{y_1\langle 1\rangle \leftarrow v\}\{y_2\langle 2\rangle \leftarrow f(v)\}) \end{array}$$

Pour les autres cas, la condition est plus simple

$$\begin{array}{l} \forall\ v, \quad f^{-1}(f(v)) = v \wedge f(f^{-1}(v)) = v \wedge \\ \quad Q\{y_1\langle 1\rangle \leftarrow v\}\{y_2\langle 2\rangle \leftarrow f(v)\}) \end{array}$$

Pour RIidempotent $f^{-1}(f(v)) = v \wedge f(f^{-1}(v)) = v$ est remplacée par $f(f(v)) = v$.

La condition est plus simple car pour les autres distributions comme sur bool ou sur les bitstring, tous les éléments se trouvent dans le domaine.

Dans la preuve de Hashed ElGamal nous utilisons la transition *optimistic sampling*, et nous devons prouver l'équivalence suivante :

$$h \stackrel{\$}{\leftarrow} \{0,1\}^k;\ \gamma \leftarrow h \oplus m_b \sim \gamma \stackrel{\$}{\leftarrow} \{0,1\}^k;\ h \leftarrow \gamma \oplus m_b : =_{\{m_b\}} \Rightarrow =_{\{h\}}$$

Une fois que le wp a substitué les deux dernières instructions déterministes, nous obtenons

$$h \stackrel{\$}{\leftarrow} \{0,1\}^k \sim \gamma \stackrel{\$}{\leftarrow} \{0,1\}^k : =_{\{m_b\}} \Rightarrow (h\langle 1\rangle = \gamma\langle 2\rangle \oplus m_b\langle 2\rangle).$$

Dans ce cas la bijection pour passer d'une distribution à l'autre est $f(x) = x \oplus m_b$, et nous prouvons qu'elle est involutive en utilisant la condition suivante :

$$x \oplus m_b \oplus m_b = x.$$

WP pour un tirage aléatoire : Rwp_disj

Cette règle permet d'effectuer le calcul de wp sur un seul des deux programmes. Ce wp est similaire à Rwp_bij. Pour une instruction $y \xleftarrow{\$} [l..r]$ nous obtenons la règle :

$$l \leq r \wedge \forall v, (l \leq v \leq r \wedge Q\{y\langle i\rangle \leftarrow v\})$$

et dans les autres cas, nous obtenons :

$$\forall\ v, Q\{y\langle i\rangle \leftarrow v\}.$$

Transformation de programme : RinlineLeft, RinlineRight et Rderandomize

Pour le moment nous effectuons ces transformations dans CertiCrypt, mais le résultat n'est pas prouvé par manque de temps.

Déplacement de bloc d'instruction : Rswap

$$\frac{\begin{array}{c} M_1 := \mathsf{modify}\ c_1 \quad M_2 := \mathsf{modify}\ c_2 \\ I_1 := \mathsf{eqobs_in}\ c_1\ M_1 \quad I_2 := \mathsf{eqobs_in}\ c_2\ M_2 \\ M_1 \cap M_2 = \varnothing \quad I_1 \cap M_2 = \varnothing \quad I_2 \cap M_2 = \varnothing \end{array}}{c \sim \mathsf{swap}(c, \mathsf{side}, \mathsf{start}, \mathsf{length}, \mathsf{dir}, \mathsf{delta}) : \mathsf{Meq} \Rightarrow \mathsf{Meq}} \ [\mathsf{Rswap\ side\ start\ length\ dir\ delta}]$$

La fonction swap prend le bloc d'instructions commençant de la ligne start et de taille length et le déplace de delta instructions. dir représente le sens du déplacement et side donne sur quel programme nous effectuons le déplacement. Pour prouver cette règle nous utilisons la règle suivante que nous appliquons après avoir utilisé Rapp sur les parties qui ne sont pas déplacées :

Lemma swapable_correct : $\forall (c_1\ c_2 : \mathsf{cmd})$,
$M_1 := \mathsf{modify}\ c_1 \to M_2 := \mathsf{modify}\ c_2 \to$
$I_1 := \mathsf{eqobs_in}\ c_1\ M_1 \to I_2 := \mathsf{eqobs_in}\ c_2\ M_2 \to$
$M_1 \cap M_2 = \varnothing \to I_1 \cap M_2 = \varnothing \to I_2 \cap M_2 = \varnothing \to$
$c1{+}{+}c2 \sim c2{+}{+}c1 : \mathsf{Meq} \Rightarrow \mathsf{Meq}$.

La fonction eqobs_in fait une analyse de dépendance. Elle prend une commande c et un ensemble O et calcule un ensemble I tel que $c \sim c : =_I \Rightarrow =_O$. Une fois que nous avons l'équivalence sur Meq nous remplaçons le programme c_1 (ou c_2) par c'.

5.4 Règles spécifiques aux adversaires

Dans la section précédente nous avons expliqué le wp pour les appels de procédure avec la règle Rwp_call. Cette règle peut être utilisée pour n'importe quel appel de procédure en particulier les adversaires. Dans cette section nous expliquerons comment prouver les spécifications sur les adversaires. Nous aurons besoin de manipuler les *wrappers* et de prouver certaines propriétés dessus. Nous pouvons

prouver deux types de spécification : l'adversaire respecte un invariant ou bien la règle des *Failure Events*.

Nous avons prouvé en Coq les règles sur l'adversaire vues dans le Chapitre 4. Pour les appliquer nous devons prouver des propriétés sur les *wrappers*. Nous devons être capable de les manipuler en faisant le lien entre une spécification sur les corps des fonctions et un *wrapper* sur ces fonctions.

Étant donné une équivalence incluse dans un objet W de type

$$W : \mathsf{EquivFun}\ P_f\ E_1\ f_1\ E_2\ f_2\ Q_f$$

nous voulons obtenir automatiquement une preuve d'équivalence sur les *wrappers* de la forme

$$r_1 \leftarrow f_1(a_1) \sim r_2 \leftarrow f_2(a_2) : P \Rightarrow Q.$$

Les relations P et Q sont définies en substituant les arguments

$$\begin{array}{l} P = P_f\{f_1.\mathsf{params} \leftarrow a_1\}\{f_2.\mathsf{params} \leftarrow a_2\} \\ Q = Q_f\{W.\mathsf{eqf_res1}\langle 1\rangle \leftarrow r_1\langle 1\rangle\}\{W.\mathsf{eqf_res2}\langle 2\rangle \leftarrow r_2\langle 2\rangle\} \end{array}$$

Nous pouvons maintenant obtenir une spécification sur les *wrappers* en utilisant cette construction.

Pour vérifier les triplets de Hoare utilisés dans les hypothèses de la règles pour l'adversaire dans le cas des *Failure Events*, nous devons prouver que l'oracle préserve l'évènement F

$$\{F\}O.\mathsf{body}\{F\}$$

nous utilisons un calcul de plus faible prédiction non relationnel. Comme pour les équivalences EasyCrypt génère un arbre de preuve permettant de donner des indications pour effectuer le calcul de wp. Enfin nous avons aussi un système de *wrapper* étant donnés que les oracles de l'adversaire sont encapsulés.

5.5 Conclusion

Dans ce chapitre, nous avons expliqué comment certifier les preuves faites dans EasyCrypt. Nous avons dû étendre CertiCrypt pour pouvoir contenir les mêmes types que EasyCrypt. Nous avons également prouvé la plupart des règles de EasyCrypt.

EasyCrypt est un outil en développement, de nouveaux types et nouveaux raisonnements sont ajoutés ce qui oblige la traduction a être étendue au fur et à mesure.

6
Formalisation de fonctions polynomiales

Indépendamment de la cryptographie, les preuves de programme portent sur la correction ou la terminaison, mais rarement sur la complexité. Les hypothèses cryptographiques se font généralement sur des programmes polynomiaux. L'adversaire est restreint à la classe des fonctions polynomiales, un adversaire avec une puissance de calcul illimitée casserait n'importe quel protocole, par exemple par force brute où un adversaire testerait l'ensemble des clés possibles.

Une manière de raisonner sur la complexité dans les preuves formelles, serait de formaliser un modèle d'exécution précis (e.g., machines de Turing) et de compter explicitement le nombre de pas nécessaire pour exécuter l'algorithme. Une telle approche est une tâche fastidieuse et le résultat dépend d'un modèle d'exécution particulier, alors que raisonner sur la complexité de manière générale indépendamment du modèle d'exécution est plus intéressant.

Une approche plus appropriée est d'utiliser la complexité implicite qui permet de raisonner sur des classes de complexité sans pour autant dépendre d'un modèle d'exécution particulier, ni de compter explicitement le nombre de pas nécessaire.

La principale motivation de ce chapitre est de présenter un moyen de raisonner sur la complexité dans le cadre des preuves de sécurité en cryptographie. En particulier pour prouver que la puissance de calcul de l'adversaire est réalisable. En 1964, Alan Cobham, montre qu'être plausible ou réalisable est équivalent à être calculable en temps polynomial [33]. Les cryptographes suivent cette thèse dans leurs preuves de sécurité en déclarant que les adversaires sont calculables en temps polynomial probabiliste PPT, c'est à dire que les adversaires peuvent s'exécuter sur une machine de Turing étendue d'une bande aléatoire, lisible en lecture seule, contenant des bits aléatoires, et qui fonctionne dans le pire des cas en temps polynomial.

De plus la classe des fonctions calculables en temps polynomial (*polytime*) a des propriétés de fermeture utiles en programmation, comme la composition et le nombre limité d'appels récursifs. Cobham utilise ces propriétés pour caractériser ces fonctions *polytime* indépendamment du modèle d'exécution. Pourtant sa preuve utilise une machine de Turing particulière, mais il explique que le choix n'a pas de

conséquence sur la preuve. Par exemple le nombre de bandes ou d'instructions n'a pas d'importance. De même, qu'ajouter une instruction pour effacer la bande, ou remettre la tête de lecture dans sa position initial ne casse pas la preuve [33].

Malheureusement, la caractérisation de Cobham n'est pas entièrement syntaxique : chaque appel récursif doit être accompagné d'une preuve qu'il existe un autre programme qui borne l'appel. Ce point rend l'appartenance à la classe non décidable, étant donné qu'il faut faire la preuve.

Trente ans plus tard, Bellantoni et Cook ont proposé un mécanisme syntaxique, pour contrôler l'augmentation de la taille des fonctions et éliminer alors cette borne explicite [24]. Le fait d'être une caractérisation syntaxique complète, permet de vérifier automatiquement l'appartenance à leur classe. Ils montrent que pour tout algorithme dans la classe de Cobham, il existe un algorithme dans la classe de Bellantoni-Cook qui calcule la même fonction et inversement. Cette classe est alors une caractérisation complète et correcte des fonctions *polytime* : toute fonction définissable dans la classe de Bellantoni-Cook (ou Cobham) est calculable en temps polynomial, et toute fonction calculable en temps polynomial est définissable dans la classe de Bellantoni-Cook (et Cobham).

Travaux reliés

Les outils pour faire des preuves de sécurité sur machine ignorent la complexité des fonctions ou mettent les hypothèses en axiomes. CertiCrypt permet de raisonner sur la complexité en temps et en espace, des programmes probabilistes. Les opérations de base sont accompagnées de deux polynômes, un pour la taille et un pour le temps. Celui pour l'espace est prouvé, mais celui pour le temps est posé en axiome étant donné qu'il n'est pas possible de raisonner sur le temps d'exécution des fonctions en Coq.

Zhang [90] propose un langage de programmation probabiliste avec un système de type pour s'assurer que les calculs se font en temps polynomial, et une logique pour raisonner sur ses programmes. Dans [71], ce langage a été appliqué aux preuves de sécurité. Zhang étend le langage SLR de Hofmann [58] et son extension probabiliste par Mitchell et al. [69].

Les approches de Cobham et Bellantoni-Cook ne permettent que de raisonner sur des entiers positifs. Comme dans ce chapitre, Zhang raisonne sur les bitstrings au lieu des entiers. Ce changement permet de se rapprocher du contexte des preuves de sécurité où par exemple les bitstrings 0 et 00 sont différentes, alors qu'elles représentent le même entier.

Un autre domaine où ce travail pourrait s'appliquer sont les preuves de réduction de problème NP complet. Par exemple, pour prouver que le problème 3-SAT est NP complet, il faut montrer qu'il existe une fonction polynomiale qui transforme les instances des solutions de SAT en solutions de 3-SAT. Dans [77], les auteurs suggèrent l'utilisation d'un "*polytime checker*", et qui pourrait être fondé sur les travaux de Bellantoni et Cook, et pourrait être utilisé pour prouver automatiquement que la réduction entre deux problèmes NP complets s'effectue en temps polynomial. Ce

chapitre propose un tel *polytime checker* et pourrait être utilisé pour formaliser des réductions polynomiales de problèmes NP complets.

Il y a d'autres critères pour vérifier qu'une fonction, définie selon différents paradigmes de programmation, est dans une classe de complexité particulière. Nous citerons quelques caractérisations logiques de fonction *polytime* [64, 78] ou caractérisation en terme de système de réécriture [9]. D'autres raisonnent sur d'autres classes de complexité [5]. À notre connaissance aucun de ces travaux n'a été appliqué à la cryptographie.

Contributions

Ce chapitre explique la formalisation de Cobham et de Bellantoni-Cook sur les bitstrings. Nous montrerons les relations entre les deux classes. Initialement, ces classes portent sur des entiers positifs, mais dans le contexte de la cryptographie, nous les adaptons pour manipuler des bitstrings. La formalisation de la classe de Cobham avec des bitstrings et la preuve que cette classe contient exactement les fonctions calculables en temps polynomial a été faite dans [85]. De la même manière, nous reformulerons les définitions de la classe Bellantoni-Cook.

Nous formalisons la preuve de Bellantoni et Cook que leur classe est équivalente à celle de Cobham de manière constructive, i.e., nous écrivons deux compilateurs pour passer d'une classe à une autre. Nous améliorons aussi les bornes des polynômes pour obtenir de meilleures transformations, tandis que Bellantoni et Cook donnent des sur-approximations étant donné qu'ils sont intéressés par l'existence des transformations et non leur optimisation.

Nous appliquons ce travail aux preuves de sécurité, principalement en étendant les expressions de CertiCrypt, en ajoutant la classe de Bellantoni-Cook. Notre librairie nous permet de calculer des polynômes pour borner la taille des résultats et le temps d'exécution automatiquement, pour tout expression de Bellantoni-Cook.

Pour raisonner sur le temps d'exécution, nous avons formalisé une sémantique qui calcule le nombre de pas d'exécution, et le polynôme qui borne ce nombre de pas.

Dans [56], nous expliquons également comment intégrer nos travaux avec la librairie de Nowak [70].

Plan

Nous commencerons par définir les différentes notions qui serviront dans le reste du chapitre, Dans la Section 6.2, nous formaliserons les classes de Cobham et de Bellantoni-Cook qui caractérisent les fonctions *polytime*. Dans les Sections 6.3 et 6.4 nous formaliserons les compilateurs de la classe de Bellantoni-Cook à la classe de Cobham et vice versa. Pour conclure, la Section 6.6 montre comment notre formalisation peut être utilisée dans les preuves de sécurité.

6.1 Notions

Dans cette section, nous introduirons notre formalisation des polynômes à plusieurs variables et aux différentes notations qui seront utilisées dans le reste de ce chapitre.

6.1.1 Polynômes à plusieurs variables

Nous implémentons une bibliothèque de polynômes positifs à plusieurs variables. Un *shallow embedding* de cette bibliothèque consiste à représenter ces polynômes directement comme des fonctions Coq sur des entiers. Dans la Section 6.3, nous avons besoin de transformer ces polynômes en expression de la classe de Cobham, nous utiliserons donc un *deep embedding*, pour obtenir un niveau d'abstraction supplémentaire, pouvoir effectuer des induction .... De plus dans la section sur CertiCrypt nous avons besoin de transformer les polynômes à plusieurs variables en polynôme à une variable.

Les polynômes sont représentés par le type suivant :

$$\mathbb{N} \times \mathsf{list}\ (\mathbb{N} \times \mathsf{list}\ (\mathbb{N} \times \mathbb{N}))$$

qui est interprété comme

$$\text{nombre de variables} \times \mathsf{list}\ \text{monôme}$$

où un monôme a le type

$$\text{constante multiplicative} \times \mathsf{list}\ (\text{variable} \times \text{puissance}))$$

Par exemple le polynôme $3y^3 + 5x^2y + 16$ est représenté par

$$(2, [(3, [(1, 3)]); (5, [(0, 2); (1, 1)]); (16, [])])$$

le 2 à droite représente le nombre de variables, les variables x et y sont respectivement représentées par 0 et 1.

Nous choisissons de mettre le nombre de variables dans le type, car nous pouvons plus facilement transformer un polynôme utilisant m variables en polynôme utilisant n variables si $n > m$. Dans la bibliothèque, nous fournissons un ensemble de fonctions pour raisonner sur des polynômes : création de polynômes et opérations d'addition, multiplication et composition. Toutes les fonctions sont paramétrées par le nombre de variables, mais nous omettrons ce paramètre dans le reste du chapitre. Nous écrivons $x_0, \ldots, x_{n-1}$ pour les variables d'un polynôme à n variables. Si P est un polynôme avec m variables et $\overline{Q} = \langle Q_0, \ldots, Q_{m-1} \rangle$ est un vecteur de polynômes à n variables, nous écrivons $P(\overline{Q})$, le polynôme à n variables où les x_i de P sont remplacées par Q_i.

Dans [52], les polynômes à plusieurs variables sont représentés en forme de Horner, ce qui permet d'obtenir une évaluation plus efficace. Dans notre formalisation, nous n'avons pas besoin d'évaluer les polynômes, nous choisirons une approche plus directe. De plus notre représentation facilite la connexion avec les polynômes à une variable de CertiCrypt (cf. Section 6.6).

6.1.2 Notations

Nous expliquerons les notations utilisées pour présenter les résultats et leurs preuves. Nous écrirons :

- xb pour la concaténation du bitstring x avec le bit b placé en bit de poids faible.
- $\overline{x}$ pour un vecteur $\langle x_0, \ldots, x_{n-1} \rangle$ (pour un n donné).
- $\overline{x}, \overline{y}$ pour la concaténation des vecteurs $\overline{x}$ et $\overline{y}$.
- $|\overline{x}|$ pour la taille du vecteur $\overline{x}$;
- $|x|$ pour la taille du bitstring x ;
- $\overline{|x|}$ représente le vecteur des tailles des polynômes dans $\overline{x}$, i.e., si $\overline{x} = \langle x_0, \ldots, x_{n-1} \rangle$ alors $\overline{|x|} = \langle |x_0|, \ldots, |x_{n-1}| \rangle$;
- $\overline{f(x)}$ applique la fonction f à chaque élément de $\overline{x}$, i.e., si $\overline{x} = \langle x_0, \ldots, x_{n-1} \rangle$ alors $\overline{f(x)} = \langle f(x_0), \ldots, f(x_{n-1}) \rangle$.
- $\overline{f}(x)$ applique chaque fonction de $\overline{f}$ à x, i.e., si $\overline{f} = \langle f_0, \ldots, f_{n-1} \rangle$ alors $\overline{f}(x) = \langle f_0(x), \ldots, f_{n-1}(x) \rangle$.

6.2 Caractérisation des fonctions *polytimes*

Dans cette section, nous expliquerons la version bitstring des classes de Cobham et de Bellantoni-Cook et quelques propriétés sur les bornes des résultats.

6.2.1 Classe de Cobham

Dans [33], Cobham caractérise les fonctions *polytimes* comme un ensemble de fonctions minimales, clos par composition et muni d'un principe de récursion. Cette caractérisation n'est pas complètement syntaxique : chaque fois que la récursion est utilisée, un autre programme de la classe de Cobham doit être fourni avec une preuve que la taille du résultat de ce programme, est plus grande que la taille de la fonction recursive.

Nous utilisons la représentation de la classe de Cobham de [85] qui permet d'utiliser directement des bitstrings plutôt que des entiers positifs codées par des bitstrings comme dans [33].

La syntaxe de la classe de Cobham $\mathcal{C}$ est donnée par :

$$
\begin{array}{rcll}
\mathcal{C} & ::= & O & \text{constante zero} \\
& | & \Pi_i^n & \text{projection} \quad (i < n) \\
& | & S_b & \text{successeur} \\
& | & \# & \text{smash} \\
& | & \mathsf{Comp}^n\ h\ \overline{g} & \text{composition} \\
& | & \mathsf{Rec}\ g\ h_0\ h_1\ j & \text{récursion}
\end{array}
$$

où i et n représentent des entiers positifs, b un bit, g, h, h_0, h_1 et j sont des expressions de $\mathcal{C}$, et $\overline{g}$ est un vecteur d'expressions de $\mathcal{C}$.

Pour raisonner sur une expression e de $\mathcal{C}$ nous avons besoin que les expressions soient bien formées. Nous définissons un prédicat $\mathcal{A}(e)$ défini par les règles suivantes

$$\mathcal{A}(O) = 0 \qquad \mathcal{A}(\Pi_i^n) = n \qquad \mathcal{A}(S_b) = 1 \qquad \mathcal{A}(\#) = 2$$

$$\frac{\mathcal{A}(h) = a_h \quad |\overline{g}| = a_h \quad \forall g \in \overline{g}, \mathcal{A}(g) = n}{\mathcal{A}(\mathsf{Comp}^n\ h\ \overline{g}) = n}$$

$$\frac{\mathcal{A}(g) = a_g \quad \mathcal{A}(h_0) = \mathcal{A}(h_1) = a_h \quad \mathcal{A}(j) = a_j \quad a_h = a_g + 2 = a_j + 1}{\mathcal{A}(\mathsf{Rec}\ g\ h_0\ h_1\ j) = a_j}$$

Dans notre implantation, $\mathcal{A}$ est une fonction qui calcule l'arité d'une expression de Cobham si celle ci est bien formée ou renvoie une erreur dans le cas contraire. Ces erreurs sont d'une grande aide pour programmer et débuguer les fonctions.

La sémantique est donnée par :

- O représente la fonction constante qui renvoie le bitstring vide ϵ.
- $\Pi_i^n(x_0, \ldots, x_{n-1})$ est égale à x_i.
- $S_b(x)$ est égale à xb.
- $\#(x, y)$ représente la fonction $1\underbrace{0\ldots\ldots0}_{|x|.|y| \text{ times}}$.
- $\mathsf{Comp}^n\ h\ \overline{g}$ est égale à la fonction telle que :

$$f(\overline{x}) = h(\overline{g}(\overline{x}))$$

- $\mathsf{Rec}\ g\ h_0\ h_1\ j$ est égale à la fonction f telle que :

$$\begin{array}{ll} f(\epsilon, \overline{x}) & = g(\overline{x}) \\ f(yi, \overline{x}) & = h_i(y, f(y, \overline{x}), \overline{x}) \\ |f(y, \overline{x})| & \leq |j(y, \overline{x})| \qquad\qquad (\mathsf{RecBounded}) \end{array}$$

Le prédicat RecBounded de type ($\mathcal{C} \rightarrow$ Prop) parcourt l'ensemble de l'expression et vérifie que pour chaque appel récursif Rec $g\ h_0\ h_1\ j$ que $|f(y, \overline{x})| \leq |j(y, \overline{x})|$ (pour tout x et y).

Si la fonction # est appelée récursivement, il serait possible d'écrire la fonction 2^n. Mais la condition RecBounded ne le permet pas, car il faudrait trouver une autre fonction, plus grande, qui respecte également la condition RecBounded.

Exemple

Pour illustrer la classe de Cobham, nous montrerons comment définir la fonction successeur en binaire :

$$\begin{array}{llll} \mathsf{Succ}(\epsilon) & = 1 & \mathsf{Rec} & (\mathsf{Comp}_0\ S_1\ O) \\ \mathsf{Succ}(x0) & = x1 & & (\mathsf{Comp}_0\ S_1\ \Pi_0^2) \\ \mathsf{Succ}(x1) & = Succ(x)0 & & (\mathsf{Comp}_0\ S_0\ \Pi_1^2) \\ \mathsf{Succ}(x) & \leq x1 & & (\mathsf{Comp}_0\ S_1\ \Pi_0^1) \end{array}$$

Propriété

Nous prouvons que la taille des résultats des fonctions de Cobham est bornés par un polynôme prenant en paramètre la taille des entrées. Nous définissons la fonction $\mathsf{Pol}_{\mathcal{C}}(f)$ par

$$\begin{array}{lcl}
\mathsf{Pol}_{\mathcal{C}}(O) & = & 0 \\
\mathsf{Pol}_{\mathcal{C}}(\Pi_i^n) & = & x_i \\
\mathsf{Pol}_{\mathcal{C}}(S_b) & = & x_0 + 1 \\
\mathsf{Pol}_{\mathcal{C}}(\#) & = & x_0.x_1 + 1 \\
\mathsf{Pol}_{\mathcal{C}}(\mathsf{Comp}^n\ h\ \overline{g}) & = & (\mathsf{Pol}_{\mathcal{C}}(h))(\overline{\mathsf{Pol}_{\mathcal{C}}(g)}) \\
\mathsf{Pol}_{\mathcal{C}}(\mathsf{Rec}\ g\ h_0\ h_1\ j) & = & \mathsf{Pol}_{\mathcal{C}}(j)
\end{array}$$

Proposition 6.1. *Pour tout f dans $\mathcal{C}$ bien formé $\mathcal{A}(f)$ et satisfaisant* RecBounded*, f est bornée par le polynôme* $\mathsf{Pol}_{\mathcal{C}}(f)$

$$|f(\overline{x})| \leq (\mathsf{Pol}_{\mathcal{C}}(f))(\overline{|x|})$$

Preuve. Par induction sur f.

Pour vérifier que les expressions dans $\mathcal{C}$, s'exécutent en temps polynomial, il faut considérer un modèle d'exécution précis. Cette partie est expliquée dans la section 6.5.

6.2.2 Classe de Bellantoni-Cook

Bellantoni et Cook ont donné une caractérisation syntaxique complète des fonctions *polytimes* qui n'utilisent pas de mécanisme explicite pour compter le nombre de pas d'exécution [24]. Le contrôle de la taille de l'exécution se fait en distinguant deux types de variable : *normal* et *safe* écrites respectivement à droite et à gauche du point virgule :

$$f(\underbrace{x_0, \ldots, x_{n-1}}_{\text{normal}}; \underbrace{x_n, \ldots, x_{n+s-1}}_{\text{safe}})$$

L'idée est que les appels récursifs se font seulement sur un argument normal, et l'appel récursif est passé du côté *safe*. Il n'est pas possible de faire un appel récursif sur le résultat d'un appel récursif. Cette limitation empêche de pouvoir écrire des fonctions exponentielles.

La syntaxe de la classe de Bellantoni-Cook $\mathcal{B}$ est donnée par :

$$\begin{array}{rcll}
\mathcal{B} & ::= & 0 & \text{constante zero} \\
& | & \pi_i^{n,s} & \text{projection} \quad (i < n+s) \\
& | & s_b & \text{successeur} \\
& | & \mathsf{pred} & \text{prédécesseur} \\
& | & \mathsf{cond} & \text{conditionnelle} \\
& | & \mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S} & \text{composition} \\
& | & \mathsf{rec}\ g\ h_0\ h_1 & \text{récursion}
\end{array}$$

où i, n et s sont des entiers positifs, b est un bit, g, h, h_0 et h_1 sont des expressions de $\mathcal{B}$, et $\overline{g_N}$ et $\overline{g_S}$ sont des vecteurs d'expression de $\mathcal{B}$. Contrairement à la classe de Cobham, la fonction j qui borne la récursion n'est pas requise. Comme pour la classe de Cobham, nous définissons une fonction qui vérifie que les expressions sont bien formées, en donnant l'arité des expressions, qui est ici une paire de deux entiers pour les variables normales et *safes*, donnée par

$$\mathcal{A}(0) = (0,0) \qquad \mathcal{A}(\pi_i^{n,s}) = (n,s) \qquad \mathcal{A}(s_b) = (0,1)$$
$$\mathcal{A}(\mathsf{pred}) = (0,1) \qquad \mathcal{A}(\mathsf{cond}) = (0,4)$$

$$\frac{\mathcal{A}(h) = (n_h, s_h) \quad |\overline{g_N}| = n_h \quad |\overline{g_S}| = s_h \qquad \forall g \in \overline{g_N}, \mathcal{A}(g) = (n,0) \quad \forall g \in \overline{g_S}, \mathcal{A}(g) = (n,s)}{\mathcal{A}(\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}) = (n,s)}$$

$$\frac{\mathcal{A}(g) = (n_g, s_g) \quad \mathcal{A}(h_0) = \mathcal{A}(h_1) = (n_h, s_h) \quad n_h = n_g + 1 \quad s_h = s_g + 1}{\mathcal{A}(\mathsf{rec}\ g\ h_0\ h_1) = (n_h, s_g)}$$

La sémantique est donnée par :
- 0 renvoie le bitstring vide ϵ.
- $\pi_i^{n,s}(x_0, \ldots, x_{n-1}; x_n, \ldots, x_{n+s-1})$ est égale à x_i.
- $s_b(; x)$ est égale à xb.
- $\mathsf{pred}(; \epsilon) = \epsilon$ et $\mathsf{pred}(; xi) = x$.
- $\mathsf{cond}(; \epsilon, x, y, z) = x$, $\mathsf{cond}(; w0, x, y, z) = y$ et $\mathsf{cond}(; w1, x, y, z) = z$.
- $\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}$ est égale à la fonction f telle que :

  $$f(\overline{x}; \overline{y}) = h(\overline{g_N}(\overline{x};); \overline{g_S}(\overline{x}; \overline{y}))$$

  Notons que les fonctions dans $\overline{g_N}$ n'ont accès qu'aux variables normales. Sinon on pourrait utiliser la composition pour faire un appel récursif sur un argument *safe*.
- $\mathsf{rec}\ g\ h_0\ h_1$ est égale à la fonction f telle que :

  $$\begin{aligned} f(\epsilon, \overline{x}; \overline{y}) &= g(\overline{x}; \overline{y}) \\ f(zi, \overline{x}; \overline{y}) &= h_i(z, \overline{x}; f(z, \overline{x}; \overline{y}), \overline{y}) \end{aligned}$$

  Notons que le résultat de l'appel récursif $f(z, \overline{x}; \overline{y})$ est passé du côté *safe*. Cela empêche de faire une récursion sur un appel récursif.

Notre définition de la classe de Bellantoni-Cook est légèrement différente de celle de [24]. Premièrement l'instruction conditionnelle distingue trois cas (vide, pair et impair), alors que dans [24] le bitstrings vide représente le nombre 0 et est traité comme le cas pair. Deuxièmement, le cas de base pour la récursion est le bitstring vide, alors que dans [24] le cas de base était un bitstring dont l'interprétation est l'entier 0, i.e., le bitstring vide ou bien le bitstring composé que de bits 0.

Nous faisons ce choix car dans la cryptographie, nous aimerions distinguer les bitstrings 0 et 00, même si ces deux bitstrings ont la même interprétation dans les entiers.

Exemple

Les deux exemples suivants montrent la version unaire de l'addition et de la multiplication dans la classe de Bellantoni-Cook et leur arité

$$\begin{array}{ll}
plus := \mathsf{rec} & mult := \mathsf{rec} \\
\quad (\pi_0^{0,1}) & \quad (\mathsf{comp}^{1,0}\ O\ \langle\rangle\ \langle\rangle) \\
\quad (\mathsf{comp}^{1,2}\ s_1\ \langle\rangle\ \langle\pi_1^{1,2}\rangle) & \quad (\mathsf{comp}^{1,2}\ plus\ \langle\pi_1^{2,0}\rangle\ \langle\pi_2^{2,1}\rangle) \\
\quad (\mathsf{comp}^{1,2}\ s_1\ \langle\rangle\ \langle\pi_1^{1,2}\rangle) & \quad (\mathsf{comp}^{1,2}\ plus\ \langle\pi_1^{2,0}\rangle\ \langle\pi_2^{2,1}\rangle) \\
 & \\
\mathcal{A}(plus) = (1,1) & \mathcal{A}(mult) = (2,0)
\end{array}$$

Dans le cas unaire, les cas pairs et les cas impairs sont identiques.

La fonction *plus* fait une récursion sur le premier argument. La fonction s_1 ajoute un bit à l'argument récursif qui est du coté *safe*. Cette expression a l'arité $(1,1)$ et peut être utilisée dans un appel récursif.

La fonction aurait pu s'écrire avec l'arité $(2,0)$, mais il est préférable d'écrire les fonctions avec le maximum d'argument *safe*, pour pouvoir utiliser les fonctions avec les appels récursifs. De plus, une fonction e d'arité $(1,1)$ peut se transformer en fonction $(2,0)$ en utilisant comp de la manière suivante :

$$\mathsf{comp}^{2,0}\ e\ \pi_0^{2,0}\ \pi_1^{2,0}$$

La fonction *mult* ne peut pas être écrite en (1,1), comme pour l'addition. Deux contraintes obligent à l'écrire en $(2,0)$. Premièrement, la récursion oblige à mettre l'argument x en normal. De plus, quand la fonction *plus* (1,1) est appelée dans le cas récursif, l'argument récursif est passé du côté *safe*, alors le y est forcément passé du côté normal, donc le deuxième argument du *mult* est aussi normal.

La fonction factorielle ne peut pas être écrite dans cette classe, car la multiplication ne peut pas être appliquée au résultat d'un appel récursif.

Propriété

Nous prouvons que la taille des résultats des fonctions de Bellantoni-Cook est bornée par un polynôme $\mathsf{Pol}_{\mathcal{B}}$ paramétré par les arguments normaux. Ce polynôme ne prend pas de variables *safe* étant donné que les variables ne servent qu'en lecture. Le résultat dépend aussi de la taille des variables *safe*, nous ajoutons alors la plus grande variable *safe*,

Nous définissons la fonction $\mathsf{Pol}_{\mathcal{B}}$ qui prend une fonction f d'arité (n,s), et renvoie un polynôme à n variables, $x_0, \ldots, x_{n-1}$. $\mathsf{Pol}_{\mathcal{B}}(f)$ est définie par :

$$\begin{array}{lcl}
\mathsf{Pol}_{\mathcal{B}}(0) & = & 0 \\
\mathsf{Pol}_{\mathcal{B}}(\pi_i^{n,s}) & = & x_i \text{ si } i < n \\
 & & 0 \ \text{ sinon} \\
\mathsf{Pol}_{\mathcal{B}}(s_b) & = & 1 \\
\mathsf{Pol}_{\mathcal{B}}(\mathsf{pred}) & = & 0 \\
\mathsf{Pol}_{\mathcal{B}}(\mathsf{cond}) & = & 0 \\
\mathsf{Pol}_{\mathcal{B}}(\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}) & = & \mathsf{Pol}_{\mathcal{B}}(h)(\overline{\mathsf{Pol}_{\mathcal{B}}(g_N)}) + \sum(\overline{\mathsf{Pol}_{\mathcal{B}}(g_S)}) \\
\mathsf{Pol}_{\mathcal{B}}(\mathsf{rec}\ g\ h_0\ h_1) & = & \mathsf{shift}(\mathsf{Pol}_{\mathcal{B}}(g)) + x_0.(\mathsf{Pol}_{\mathcal{B}}(h_0) + \mathsf{Pol}_{\mathcal{B}}(h_1))
\end{array}$$

Pour le comp, le polynôme qui borne la taille de l'exécution de h est composé avec la taille des polynômes qui bornent les expressions de g_N. Pour les polynômes du côté *safe*, la somme des polynômes qui bornent les expressions de g_S est ajoutée, pour éviter d'avoir à calculer le plus grand polynôme de $\overline{\mathsf{Pol}_{\mathcal{B}}(g_S)}$.

Pour rec, les variables du polynôme pour g sont décalées, étant donné que g prend un argument de moins. Pour décaler les variables, nous utilisons la fonction $\mathsf{shift}(P)$ qui représente le polynôme P avec les variables x_i remplacées par x_{i+1}. Les polynômes qui bornent h_0 et h_1 sont multipliés par le nombre de récursions x_0.

Proposition 6.2 (Polymax Bounding). *Pour toute fonction f dans $\mathcal{B}$ d'arité (n, s), et pour tout argument $\overline{x}$ et $\overline{y}$, le polynôme $\mathsf{Pol}_{\mathcal{B}}(f)$ borne la taille du résultat de f de la manière suivante :*

$$|f(\overline{x}; \overline{y})| \leq (\mathsf{Pol}_{\mathcal{B}}(f))(\overline{|x|}) + \mathsf{max}_i |y_i|$$

Preuve. Par induction sur f.

Principe de récurrence

Tous les lemmes que nous écrivons font l'hypothèse que l'expression est bien formée. C'est à dire que pour une fonction e, la fonction $\mathcal{A}(e) = (n, s)$. La fonction $\mathcal{A}$ effectue des tests pour vérifier si l'expression est bien formée, et ces tests se calculent dans Coq. Sans ce principe de récurrence tout les debuts de preuves consistent à déplier la fonction $\mathcal{A}$. Pour éviter de faire cette étape à chaque preuve nous écrivons un principe d'induction, qui prend en compte la fonction d'arité. Le lemme d'induction prend des formules de la forme suivante :

$$\forall\ e\ n\ s, \mathcal{A}(e) = (n, s) \rightarrow P\ n\ s\ e$$

avec P de type $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathcal{B} \rightarrow \mathsf{Prop}$ pour être le plus général possible. Les hypothèses à vérifier sont les suivantes :

- $P\ 0\ 0\ 0$ (où le troisième 0 est la notation de l'expression zero dans $\mathcal{B}$)
- $\forall\ n\ s\ i, i < n + s \rightarrow P\ n\ s\ \pi_i^{n,s}$
- $P\ 0\ 1\ \mathsf{pred}$
- $P\ 0\ 4\ \mathsf{cond}$
- $\begin{cases} \forall\ n\ s\ g\ h_0\ h_1, \mathcal{A}(g) = (n, s) \rightarrow \\ \mathcal{A}(h_0) = (S\ n, S\ s) \rightarrow \mathcal{A}(h_1) = (S\ n, S\ s) \rightarrow \\ P\ n\ s\ g \rightarrow P\ (S\ n)\ (S\ s)\ h_0 \rightarrow P\ (S\ n)\ (S\ s)\ h_1 \rightarrow \\ P\ (S\ n)\ s\ (\mathsf{rec}\ g\ h_0\ h_1) \end{cases}$
- $\begin{cases} \forall\ n\ s\ h\ l_n\ l_s, A(h) = (|l_n|, |l_s|) \rightarrow P\ |l_n|\ |l_s|\ h \rightarrow \\ (\forall\ e\ \in\ l_n, \mathcal{A}(e) = (n, 0) \wedge P\ n\ 0\ e) \rightarrow \\ (\forall\ e\ \in\ l_s, \mathcal{A}(e) = (n, s) \wedge P\ n\ s\ e) \rightarrow \\ P\ n\ s\ (\mathsf{comp}^{n,s}\ h\ l_n\ l_s) \end{cases}$

6.2.3 Version avec inférence d'arité

Pour faire les preuves en Coq, nous avons besoin de vérifier que les tailles des listes d'arguments correspondent bien à l'arité de la fonction. Nous avons rajouté des informations supplémentaires dans les classes pour calculer l'arité.

Ces informations supplémentaires rendent l'écriture des fonctions difficiles, car il faut savoir à chaque fois dans quel contexte nous nous trouvons. C'est pourquoi nous proposons une autre définition de la classe $\mathcal{B}$ que nous appelons $\mathcal{B}_{\mathsf{inf}}$. Cette nouvelle classe ne prend pas d'information et l'arité calculée est juste le nombre de variables utilisées.

Par exemple une fonction dans $\mathcal{B}$ d'arité $(5, 0)$, qui n'utilise que x_2 et x_4, sera d'arité $(4, 0)$. Cette nouvelle classe permet surtout de simplifier l'écriture des fonctions, car les informations pour calculer l'arité sont inférées.

Pour valider cette nouvelle classe, nous écrivons deux compilateurs certifiés de $\mathcal{B}$ vers $\mathcal{B}_{\mathsf{inf}}$ et de $\mathcal{B}_{\mathsf{inf}}$ vers $\mathcal{B}$. Nous prouvons que les expressions générées sont bien formeés et que le resulat est le meme.

Cette nouvelle classe est définie par :

$$
\begin{array}{lcllr}
\mathcal{B}_{\mathsf{inf}} & ::= & 0 & \text{constante zero} & \\
 & | & \pi_i^{\mathsf{normal}} & \text{projection (normal)} & (i < n) \\
 & | & \pi_i^{\mathsf{safe}} & \text{projection (safe)} & (i < s) \\
 & | & s_b & \text{successeur} & \\
 & | & \mathsf{pred} & \text{prédécesseur} & \\
 & | & \mathsf{cond} & \text{conditionnelle} & \\
 & | & \mathsf{comp}\ h\ \overline{g_N}\ \overline{g_S} & \text{composition} & \\
 & | & \mathsf{rec}\ g\ h_0\ h_1 & \text{récursion} &
\end{array}
$$

La fonction comp ne prend plus d'information, car elle calcule les arités des fonctions des listes g_N et g_S. Par exemple, si les arités des listes, g_N sont $(3, 0), (0, 0)$ et que celles de g_S sont $(1, 2)$ et $(2, 0)$, l'arité calculée prend le maximum de toutes les valeurs soit $(3, 2)$.

Le calcul d'arité échoue dans moins de cas que pour la classe $\mathcal{B}$, mais prend soin de vérifier que les fonctions de g_N n'ont pas accès aux variables *safe*.

Pour la projection, demander la n^{ieme} ne permet pas de distinguer si la variable est normale ou *safe*. C'est pourquoi nous avons deux projections une pour les variables normales et une pour les *safes*.

6.3 Compiler Bellantoni-Cook vers Cobham

Dans cette section nous expliquerons comment compiler des expressions de Bellantoni-Cook vers les expressions de Cobham. Nous définissons le compilateur prenant une expression f de $\mathcal{B}$ et renvoyant une expression $[\![f]\!]_{\mathcal{B}}$ qui calcule la même fonction dans $\mathcal{C}$. Nous prouvons le théorème suivant :

Théorème 6.3. *Pour tout f dans $\mathcal{B}$ bien définie avec $\mathcal{A}(f) = (n, s)$, et pour tous les arguments $\overline{x}$ et $\overline{y}$,*

$$f(\overline{x}; \overline{y}) = [\![f]\!]_{\mathcal{B}}(\overline{x}, \overline{y})$$

Nous expliquerons comment est défini le compilateur tout en donnant une idée de la preuve.

Preuve. Nous devons prouver plusieurs propriétés sur la compilation, la sémantique est bien préservée, la fonction générée est bien définie et elle respecte la condition *RecBounded*.

Pour garantir la propriété *RecBounded*, nous devrons pour chaque récursion fournir une expression j qui borne l'appel récursif.

– les premiers cas sont triviaux :

$$\begin{array}{lcl} [\![0]\!]_{\mathcal{B}} & = & O \\ [\![\pi_i^{n,s}]\!]_{\mathcal{B}} & = & \Pi_i^{n+s} \\ [\![s_b]\!]_{\mathcal{B}} & = & S_b \end{array}$$

– pred et cond sont transformées en utilisant Rec :

$$\begin{array}{lcl} [\![\mathsf{pred}]\!]_{\mathcal{B}} & = & \mathsf{Rec}\ O\ \Pi_0^2\ \Pi_0^2\ \Pi_0^1 \\ [\![\mathsf{cond}]\!]_{\mathcal{B}} & = & \mathsf{Rec}\ \Pi_0^3\ \Pi_4^5\ \Pi_3^5 \\ & & \quad \mathsf{Comp}^4\ \#\ \langle \\ & & \quad\quad \mathsf{Comp}^4\ S_1\ \langle \Pi_1^4 \rangle; \\ & & \quad\quad \mathsf{Comp}^4\ \#\ \langle\ \mathsf{Comp}^4\ S_1\ \langle \Pi_2^4 \rangle; \mathsf{Comp}^4\ S_1\ \langle \Pi_3^4 \rangle\ \rangle\rangle \end{array}$$

Pour le pred, nous effectuons une récursion qui s'arrête au bout du premier tour de boucle. Pour le cond, nous nous servons du Rec pour distinguer les cas nil, pairs, impairs. Si cond est appelée avec $\mathsf{cond}(; c, x, y, z)$, nous bornons la récursion par $\#(x, \#(y, z))$

– Pour $\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}$ nous avons besoin de rajouter des variables inutiles, car les fonctions de $\overline{g_N}$ ne prennent pas d'argument *safe*, Nous transformons les fonctions de $\overline{g_N}$ en fonctions d'arité $n + s$. La fonction $\mathsf{dummies}_s$ (écrit dans $\mathcal{C}$) ajoute s variables inutiles qui seront ignorées :

$$[\![\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}]\!]_{\mathcal{B}} \quad = \quad \mathsf{Comp}^{n+s} \quad [\![h]\!]_{\mathcal{B}} \quad \left(\overline{\mathsf{dummies}_s\ ([\![g_N]\!]_{\mathcal{B}})}, \overline{[\![g_S]\!]_{\mathcal{B}}}\right)$$

– Pour rec $g\ h_0\ h_1$ nous avons besoin de changer l'ordre des arguments pour h_0 et h_1. Dans $\mathcal{B}$, l'argument récursif est passé dans le premier argument *safe* juste après les arguments normaux. Dans $\mathcal{C}$ il est passé en second argument. Nous écrivons une fonction dans $\mathcal{C}$ qui permet de changer la position d'un argument, on écrit $\mathsf{move_arg}_{2,n}$.

De plus, nous avons besoin de donner le quatrième argument, pour Rec. Nous commencerons par calculer un polynôme qui borne la taille du résultat, à l'aide de $\mathsf{Pol}_{\mathcal{B}}$, puis nous transformerons ce polynôme en expression de $\mathcal{B}$. La proposition 6.2, nous donne :

$$|\mathsf{rec}\ g\ h_0\ h_1| \leq (\mathsf{Pol}_{\mathcal{B}}(\mathsf{rec}\ g\ h_0\ h_1)) + \mathsf{max}_i|y_i|.$$

On peut affaiblir cette borne en prenant la somme des arguments *safe* plutôt que le maximum, car

$$\mathsf{max}_i|y_i| \leq |y_n| + \cdots + |y_{n+s-1}|$$

La récursion est donc bornée par le polynôme :

$$\mathsf{Pol}_{\mathcal{B}}(\mathsf{rec}\ g\ h_0\ h_1) + |y_n| + \cdots + |y_{n+s-1}|$$

Nous définissons ensuite la fonction $\mathsf{Pol}_{\mathcal{C}}^{-1}$ qui transforme un polynôme en expression de Cobham. Pour tout polynôme P, $\mathsf{Pol}_{\mathcal{C}}^{-1}(P)$ encode la taille de P dans la classe de Cobham, c'est-à-dire,

$$\left|\mathsf{Pol}_{\mathcal{C}}^{-1}(P)(\overline{x})\right| = P(\overline{|x|}).$$

Nous pouvons utiliser cette fonction pour obtenir le quatrième argument de l'argument récursif, la preuve que le programme est donné par la spécification de $\mathsf{Pol}_{\mathcal{B}}$. Nous obtenons :

$$\begin{array}{rcl} [\![\mathsf{rec}\ g\ h_0\ h_1]\!]_{\mathcal{B}} & = & \mathsf{Rec}\quad [\![g]\!]_{\mathcal{B}} \\ & & \quad \mathsf{move_arg}_{2,n}\ [\![h_0]\!]_{\mathcal{B}} \\ & & \quad \mathsf{move_arg}_{2,n}\ [\![h_1]\!]_{\mathcal{B}} \\ & & \quad \mathsf{Pol}_{\mathcal{C}}^{-1}\begin{pmatrix} \mathsf{Pol}_{\mathcal{B}}(\mathsf{rec}\ g\ h_0\ h_1) + \\ |y_n|\ +\ \cdots\ +\ |y_{n+s-1}| \end{pmatrix} \end{array}$$

6.4 Compiler Cobham vers Bellantoni-Cook

Contrairement à la classe de Bellantoni-Cook $\mathcal{B}$, la classe de Cobham $\mathcal{C}$ ne fait aucune distinction entre les arguments normaux et les arguments *safe*. Dans $\mathcal{C}$ il est possible de faire des récurrences sur n'importe quel argument tandis que $\mathcal{B}$ permet uniquement d'en faire sur les arguments normaux. Lors de la transformation de $\mathcal{C}$ vers $\mathcal{B}$, nous devons introduire cette distinction et gérer correctement la récursion. Nous suivons la transformation suggérée par Bellantoni et Cook en mettant tous les arguments en positon *safe*, et en ajoutant un argument artificiel w dont la taille doit être suffisamment grande pour simuler tous les pas de la récursion. Nous prouverons le lemme suivant qui est une étape intermédiaire pour prouver la Compilation de Cobham vers Bellantoni-Cook.

Lemme 6.4 (Récursion Simulation). *Pour toute fonction f dans $\mathcal{C}$ avec une arité $\mathcal{A}(f) = n$ et respectant la condition RecBounded, alors la fonction $[\![f]\!]_{\mathcal{C}}$ dans $\mathcal{B}$ et le polynôme $\mathcal{P}(f)$ respectent la propriété suivante : pour tout argument $\overline{x}$ et bitstring w tel que $\mathcal{P}(f)(\overline{|x|}) \leq |w|$,*

$$f(\overline{x}) = [\![f]\!]_{\mathcal{C}}(w; \overline{x})$$

Nous expliquerons dans la suite comment la fonction $[\![.]\!]_{\mathcal{C}}$ et le polynôme $\mathcal{P}$ sont définis en donnant une idée de la preuve.

Preuve. Notre traduction de $\mathcal{C}$ vers $\mathcal{B}$ prend une fonction f d'arité n et renvoie une fonction d'arité $(1, n)$. Les arguments de f sont passés du côté *safe*. L'argument normal correspond à l'argument artificiel w, qui doit être suffisamment grand pour simuler la récursion. Nous avons en hypothèse que w est plus grand que $\mathcal{P}(f)(\overline{|x|})$

Les premiers cas sont immédiats et doivent simplement prendre en compte l'arité.

$$\begin{array}{lcl} [\![O]\!]_{\mathcal{C}} & = & \mathsf{comp}^{1,n}\ 0\ \langle\rangle\ \langle\rangle \\ [\![\Pi_i^n]\!]_{\mathcal{C}} & = & \pi_{i+1}^{1,n} \\ [\![S_b]\!]_{\mathcal{C}} & = & \mathsf{comp}^{1,n}\ s_b\ \langle\rangle\ \langle\pi_1^{1,n}\rangle \\ [\![\mathsf{Comp}^n\ h\ \overline{g}]\!]_{\mathcal{C}} & = & \mathsf{comp}^{1,n}\ [\![h]\!]_{\mathcal{C}}\ \langle\pi_1^{1,0}\rangle\ \overline{[\![g]\!]_{\mathcal{C}}} \end{array}$$

le polynôme $\mathcal{P}$ pour ces quatre instructions est défini par :

$$\begin{array}{lcl} \mathcal{P}(O) & = & 0 \\ \mathcal{P}(\Pi_i^n) & = & 0 \\ \mathcal{P}(S_b) & = & 0 \\ \mathcal{P}(\mathsf{Comp}^n\ h\ \overline{g}) & = & \mathcal{P}(h)(\overline{\mathsf{Pol}_{\mathcal{C}}(g)}) + \sum_{g \in \overline{g}} \mathcal{P}(g) \end{array}$$

La preuve pour le Comp applique les hypothèses de récurrence, et la Proposition 6.1.

Pour le $\mathsf{Rec}\ g\ h_0\ h_1\ j$, nous suivons la preuve de [24] qui utilise un ensemble de fonctions dans $\mathcal{B}$, mais notre construction est plus simple.

Nous utilisons la fonction P telle que $P(a; b)$ supprime les $|a|$ bits de poids faible de b. La fonction est définie par $P(\epsilon; b) = b$ et $P(ai; b) = \mathsf{pred}(; P(a; b))$.

La fonction Y est définie tel que $Y(z, w; y)$ supprime les $|w| - |z|$ bits y : $Y(z, w; y) = P(P'(z, w); y)$ où $P'(a, b;) = P(a; b)$. Nous définissons la fonction intermédiaire $\hat{f}$ qui utilise P and Y :

$$\begin{array}{l} \hat{f}(\epsilon, w; y, \overline{x}) = g(w; \overline{x}) \\ \hat{f}(zj, w; y, \overline{x}) = \begin{cases} g'(w; \overline{x}) & \text{if } Y(S_1 z, w; y) \text{ is } \epsilon \\ h'_0(w, Y(z, w; y), \hat{f}(z, w; y, \overline{x}), \overline{x}) & \text{if } Y(S_1 z, w; y) \text{ is even} \\ h'_1(w; Y(z, w; y), \hat{f}(z, w; y, \overline{x}), \overline{x}) & \text{if } Y(S_1 z, w; y) \text{ is odd} \end{cases} \end{array}$$

paramétrée par les fonctions g' (d'arité $(1, s-1)$) et h'_0, h'_1 (d'arité $(1, s+1)$). Notre définition de $\hat{f}$ est plus simple que celle de [24], car pour le cas de base, nous n'avons pas besoin de vérifier que y correspond à l'entier naturel 0. Notre cas de base est simplement le bitstring vide ϵ, étant donné que le cond peut vérifier que le premier argument *safe* est vide. Nous définissons la fonction f' par $\hat{f}(w, w; y, \overline{x})$, et pour finir :

$$[\![\mathsf{Rec}\ g\ h_0\ h_1\ j]\!]_{\mathcal{C}} = f'\ [\![g]\!]_{\mathcal{C}}\ [\![h_0]\!]_{\mathcal{C}}\ [\![h_1]\!]_{\mathcal{C}}$$

Le polynôme correspondant est défini par :

$$\begin{array}{r} \mathcal{P}(\mathsf{Rec}\ g\ h_0\ h_1\ j)(|y|, \overline{|x|}) = (\mathcal{P}(h_0) + \mathcal{P}(h_1))(|y|, \mathsf{Pol}_{\mathcal{C}}(f), \overline{|x|}) + \\ \mathsf{shift}(\mathcal{P}(g)(\overline{|x|})) + |y| + 2 \end{array}$$

La fonction *smash* est définie par une double récursion de la manière suivante :

$$\begin{array}{lcll} \#'(\epsilon, y) & = & y & \\ \#'(xi, y) & = & \#'(x, y)\,0 & \text{(concatenation avec un bit 0)} \\ \#(\epsilon, y) & = & 1 & \\ \#(xi, y) & = & \#'(y, \#(x, y)) & \end{array}$$

Nous construirons la fonction *smash* en utilisant deux fois la fonction f', étant donné que nous avons besoin de deux récursions. La fonction f' est paramétrée par trois fonctions. Dans le cas *Rec*, ces trois fonctions sont les appels récursifs. Pour le *smash*, nous donnerons nous mêmes ces fonctions. Les cas h_0 et h_1 étant équivalents pour le #, nous écrirons $f\ g\ h$ au lieu de $f\ g\ h\ h$. Nous définissons $\#'$ dans $\mathcal{B}$ par

$$[\![\#']\!]_{\mathcal{C}} = f'\ (\pi_1^{1,1})\ (\mathsf{comp}^{1,3}\ S_0\ \langle\rangle\ \langle\pi_2^{1,3}\rangle).$$

et nous concluons

$$[\![\#]\!]_{\mathcal{C}} = f'\ (\mathsf{one}^{1,1})\ (\mathsf{dummies}_{0,1}(\mathsf{comp}^{1,2}\ \#'\ \langle\pi_0^{1,0}\rangle\ \langle\pi_2^{1,2};\pi_1^{1,2}\rangle))$$

Pour le polynôme, après simplification, nous obtenons :

$$\mathcal{P}(\#) = x_0 + 2x_1 + 18.$$

Pour finir la preuve, nous devons calculer automatiquement le premier argument w, pour qu'il soit suffisamment grand. Nous définissons la fonction b_f par

$$\mathsf{Pol}_{\mathcal{B}}^{-1}(\mathcal{P}(f))$$

avec $\mathsf{Pol}_{\mathcal{B}}^{-1}$ une fonction transformant un polynôme en expression de Bellantoni-Cook tel que pour tout polynôme P, $\mathsf{Pol}_{\mathcal{B}}^{-1}(P)$ est un encodage unaire du polynôme dans $\mathcal{B}$, c'est à dire

$$|\mathsf{Pol}_{\mathcal{B}}^{-1}(P)(\overline{x})| = P(\overline{|x|}).$$

Nous construisons le compilateur $\wr . \int_{\mathcal{C}}$ par

$$\wr f \int_{\mathcal{C}}(\overline{x};) = [\![f]\!]_{\mathcal{C}}(b_f(\overline{x};);\overline{x}).$$

Le résultat final est énoncé dans le théorème suivant :

Théorème 6.5. *Pour tout f dans $\mathcal{C}$ avec une arité $\mathcal{A}(f) = n$ et respectant la condition RecBounded, alors*

$$f(\overline{x}) = \wr f \int_{\mathcal{C}}(\overline{x};)$$

Preuve. Nous prouvons ce théorème en appelant le Lemme 6.4, l'hypothèse sur w est vérifiée car nous savons que

$$|b_f(\overline{x};)| = \mathcal{P}(f)(\overline{|x|})$$

Notre traduction est plus efficace que celle de [24] car notre définition de b_f est plus précise : nous ne faisons pas plus d'appel récursif que nécessaire. Dans [24] les auteurs utilisent une propriété sur les polynômes à plusieurs variables pour montrer qu'il existe deux entiers a et c tel que

$$\mathcal{P}(f)(\overline{|x|}) \leq (\sum_j |x_j|)^a + c$$

et utilisent a et c pour construire b_f qui satisfait la condition du Lemme 6.4, i.e., $\mathcal{P}(f)(\overline{|x|}) \leq |b(\overline{x};)|$. Leur b_f est une sur-approximation , alors que le nôtre donne le nombre nécessaire de pas.

6.5 Temps d'exécution polynomial dans $\mathcal{B}$

Pour le moment, les résultats présentés ne montrent que la partie concernant la complexité en espace. La section 3.4.2 de la thèse de Bellantoni propose de borner le temps d'exécution par la taille de la dérivation.

Nous modifions notre sémantique pour prendre en compte le nombre de pas d'exécution. Nous prouvons que si il existe des polynômes pour borner les opérations de base (tous les opérateurs sauf rec et comp), alors nous calculons un polynôme qui borne le temps d'exécution de la fonction.

Nous avons les fonctions de coût pour les opérateurs de base, cost_0, cost_{S_i}, $\mathsf{cost}_{\mathsf{pred}}$, $\mathsf{cost}_{\mathsf{cond}}$. Le coût pour le proj est paramétré, nous pouvons imaginer que le temps d'exécution de la projection dépend de l'indice, nous avons $\mathsf{cost}_{\mathsf{proj\ n\ s\ j}}$. Pour chaque opérateur de base, il existe un polynôme qui borne le coût, soit Pcost_0, Pcost_{S_i}, $\mathsf{Pcost}_{\mathsf{pred}}$, $\mathsf{Pcost}_{\mathsf{cond}}$ et $\mathsf{Pcost}_{\mathsf{proj\ n\ s\ j}}$.

Le polynôme à plusieurs variables $\mathsf{Pol}_{\mathsf{time}}$ suivant est une borne supérieure du temps d'exécution :

$$\begin{array}{lcl}
\mathsf{Pol}_{\mathsf{time}}(0) & = & \mathsf{Pcost}_0 \\
\mathsf{Pol}_{\mathsf{time}}(\pi_i^{n,s}) & = & \mathsf{Pcost}_{\mathsf{proj\ n\ s\ j}} \\
\mathsf{Pol}_{\mathsf{time}}(s_b) & = & \mathsf{Pcost}_{S_i} \\
\mathsf{Pol}_{\mathsf{time}}(\mathsf{pred}) & = & \mathsf{Pcost}_{\mathsf{pred}} \\
\mathsf{Pol}_{\mathsf{time}}(\mathsf{cond}) & = & \mathsf{Pcost}_{\mathsf{cond}} \\
\mathsf{Pol}_{\mathsf{time}}(\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}) & = & \mathsf{Pol}_{\mathsf{time}}(h)(\overline{\mathsf{Pol}_{\mathcal{B}}(g_N)})+ \\
 & & \sum(\overline{\mathsf{Pol}_{\mathsf{time}}(g_N)}) + \sum(\overline{\mathsf{Pol}_{\mathsf{time}}(g_S)}) \\
\mathsf{Pol}_{\mathsf{time}}(\mathsf{rec}\ g\ h_0\ h_1) & = & \mathsf{shift}(\mathsf{Pol}_{\mathsf{time}}(g))+ \\
 & & x_0.(\mathsf{Pol}_{\mathsf{time}}(h_0) + \mathsf{Pol}_{\mathsf{time}}(h_1))
\end{array}$$

où $\mathsf{shift}(P)$ est le polynôme P où toutes les variables x_i sont remplacées par x_{i+1}.

Dans le cas du $\mathsf{comp}^{n,s}\ h\ \overline{g_N}\ \overline{g_S}$ le h prend en argument les polynômes bornant la taille des fonctions dans $\overline{g_N}$.

Le temps d'exécution ne dépend pas de la taille des arguments *safe*. La restriction syntaxique assure qu'il n'est pas possible de faire une récursion sur les variables *safes*.

6.6 Application à CertiCrypt

Les preuves de sécurité des chapitres précédents, font appel à la notion de programmes polynomiaux. Les adversaires sont aussi polynomiaux.

Dans CertiCrypt, la sémantique des programmes permet de mesurer le temps d'exécution. La sémantique prend en paramètre une mémoire initiale et un programme, et renvoie une distribution sur les mémoires pour chaque état atteignable avec une probabilité non nulle.

Un programme est PPT si il se termine et si il existe un polynôme qui borne la taille de toute variable de la mémoire initiale, alors on peut calculer deux polynômes qui bornent le temps et l'espace des distributions.

Plus formellement un programme est PPT, si il se termine et qu'il existe deux fonctions sur des polynômes P et G telles que pour tout polynôme (p,q) qui bornent les distributions sur la mémoire initiale, la distribution sur les mémoires finales qui ont une probabilité non nulle d'être atteinte, est bornée par $F(p)$ et $q + G(p)$.

CertiCrypt prouve l'existence de ces transformateurs de polynômes pour chacune des instructions. Pour l'assignement $v \leftarrow e$, l'expression e a également besoin d'être PPT. En particulier si elle utilise un opérateur.

La définition de PPT pour une expression e est que si toute variable libre de e de la mémoire initiale est bornée par un polynôme p, alors il existe F et G tel que la taille du résultat et le temps d'exécution sont bornés par les polynômes $F(p)$ et $G(p)$.

Tous les opérateurs de base de CertiCrypt contiennent ces preuves, et si l'utilisateur ajoute de nouvelles expressions, il doit également faire ces preuves. Pour le moment seules les preuves pour borner la taille sont faites, les preuves pour le temps sont laissées en axiomes, étant donné qu'il n'y a pas de moyen de raisonner sur le temps d'exécution des fonctions Coq.

Dans cette section, nous proposons d'ajouter à CertiCrypt les expressions de la classe de Bellantoni-Cook. L'ensemble du travail sur la classe $\mathcal{B}$ permet d'ajouter un nouvel opérateur dans CertiCrypt, tout en prouvant l'existence des transformateurs de polynômes F et G.

Conversion des Polynômes

Les polynômes de CertiCrypt sont des polynômes à une variable, tandis que notre formalisation utilise des polynômes à plusieurs variables.

Pour borner la taille et le temps d'exécution des expressions dans $\mathcal{B}$, nous utilisons des polynômes à plusieurs variables, pour réutiliser ces polynômes dans CertiCrypt nous devons les convertir en polynôme à une variable.

Pour un polynôme à plusieurs variables P, la fonction $\lceil P \rceil$ remplace toutes les variables $x_0, \ldots, x_{n-1}$ par une seule variable x. Ensuite le nouveau polynôme à une variable est appliqué avec le maximum des variables $x_0, \ldots, x_{n-1}$, nous obtenons :

$$\lceil P \rceil \; =_{def} \; P[x_0 \mapsto x; \ldots ; x_{n-1} \mapsto x]$$

Et nous prouvons que

$$P(x_0, \ldots, x_{n-1}) \leq \lceil P \rceil(\mathsf{max}(x_0, \ldots, x_{n-1}))$$

Polynôme pour l'espace

Pour un programme c définit dans la classe de Bellantoni-Cook, le polynôme $F(p)$ est défini par

$$1 \; + \; 2 \lceil \mathsf{Pol}_{\mathcal{B}}(c) \rceil (p)$$

justifié par la Proposition 6.2. La multiplication par 2 vient du fait que la taille d'un booléen dans CertiCrypt est 2.

Polynôme pour le temps

Pour un programme c, nous définissions la fonction de coût dans CertiCrypt par la fonction cost et le polynôme $G(p)$ par

$$\lceil \mathsf{Pol}_{\mathsf{time}}(c) \rceil (p).$$

Le lecteur peut être surpris car dans cette section nous utilisons une implémention particulière de la classe de Bellantoni-Cook alors que dans l'introduction nous avons dit que nous nous intéressions à la complexité indépendemment du modèle d'exécution. La raison est que Cobham ou Bellantoni et Cook garantissent l'existence d'un polynôme qui borne le temps d'exécution, sans donner aucune indication sur sa valeur. CertiCrypt a besoin d'une définition exacte du polynôme, c'est pourquoi nous considérons ici un modèle d'exécution particulier pour calculer ce polynôme.

6.7 Conclusion

Nous avons formalisé les classes de Cobham et de Bellantoni-Cook, et leurs relations dans l'assistant de preuve Coq. L'intérêt d'un tel développement en Coq est de pouvoir formaliser des parties qui étaient décrites de manière informelle dans le papier de Bellantoni et Cook. Notre formalisation permet d'utiliser ces deux classes comme des langages de programmation, pour y définir des fonctions calculables en temps polynomial. Nous montrons que la formalisation de fonction polynomiale peut être utilisée dans les preuves de sécurité, pour la définition des adversaires.

Travaux Futurs

Pour faciliter l'utilisation de notre formalisation, il est important d'accompagner notre travail d'une librairie de fonctions polynomiales, sur les bitstrings, permettant de construire des fonctions plus complexes.

Nous avons implémenté des fonctions simples, comme les opérations bit à bit, (XOR, NOT, AND, etc). Mais des fonctions plus complexes comme l'addition binaire sont nettement plus difficiles à écrire, et montrent que ce genre de langage n'est pas forcement adapté pour la programmation. À l'heure actuelle, nous sommes

encore loin de pouvoir formaliser des algorithmes utilisés dans les preuves de Cryptographie comme l'algorithme d'Euclide.

Une solution serait d'utiliser la classe de Cobham pour programmer, et de la convertir ensuite dans celle de Bellantoni-Cook en utilisant notre compilateur $\mathcal{C} \to \mathcal{B}$ (défini dans la Section 6.4). L'addition et la multiplication binaires sont définies dans [74], et nous avons commencé à les implémenter. Mais encore une fois ce travail est laborieux : chaque étape demande une preuve de *RecBounded* suivie d'une preuve de correction.

Nous pensons que l'idéal serait d'étendre notre formalisation à des langages de programmation plus puissants, comme CSLR [71]. Notre travail peut être vu comme une première étape, et servirait de base pour la formalisation d'autres classes polynomiales.

7
Conclusion

7.1 Travaux reliés

Une des formalisations en Coq la plus proche de la nôtre, en utilisant les preuves par jeux, et celle de Nowak [70], qui prouve la sécurité de ElGamal. Nous utilisons un *deep embedding*, alors que Nowak utilise un *shallow embedding* et les adversaires sont modélisés comme des fonctions Coq. Cette différence implique que cette bibliothèque offre un support limité d'automatisation des preuves. Pour les mêmes raisons, la formalisation de Nowak ne peut pas utiliser de *random oracle*, et ne présente alors que la preuve de Hashed ElGamal dans le modèle standard. ElGamal est un exemple standard de preuve par jeux, qui permet de fournir des comparatifs par rapport aux autres travaux.

Des travaux plus anciens, effectués par Barthe, Cederquist et Tarento [16] donnent les bases des preuves formelles en Coq, en prouvant la sécurité du chiffrement ElGamal signé. À la différence de notre travail, la preuve est effectuée dans le modèle générique, qui est idéal et très spécialisé, et se débarrasse de beaucoup de points pertinents en cryptographie.

Corin et den Hartog [35] ont développé une logique de Hoare (non relationnelle) pour raisonner sur les programmes probabilistes. Ils utilisent cette logique pour prouver la sécurité du chiffrement ElGamal. Leur formalisme n'est pas assez puissant pour exprimer précisément les notions de sécurité : le fait que l'adversaire soit bien formé ou plausible, ne peut pas être modélisé.

Blanchet et Pointcheval ont développé le premier outil de vérification qui utilisent les preuves par jeux. CryptoVerif a permis de prouver la sécurité de FDH et leur outil est utilisé pour de nombreux protocoles comme Kerberos [29]. Une perspective intéressante serait d'utiliser CryptoVerif pour générer la séquence de jeux et ensuite, CertiCrypt ou EasyCrypt pour vérifier les preuves.

Impagliazzo et Kapron [61] ont développé une logique pour raisonner sur l'indistingabilité. Leur logique est construite sur une logique très générale, basée sur

l'arithmétique non standard. Ils montrent la correction de générateur pseudo aléatoire et que *next-bit unpredictability* implique *pseudo-randomness*.

Plus récemment, Zhang [90] développe une approche similaire au dessus de la classe SLR de Hoffman [57], appelée CSLR et refait les exemples de Impagliazzo et Kapron. Ensuite Nowak et Zhang [71] étendent la logique aux preuves par jeux. Leur système de type garantit que les calculs se font en temps polynomial probabiliste, mais leur bibliothèque n'est pas formalisée. Les travaux de formalisation de classe de complexité polynomiale, effectués dans le chapitre 6, peuvent être étendus à la classe SLR, puis à la classe CSLR.

Barthe et al. [18] développent une logique appelée *Computational Indistinguishability Logic* (CIL), permettant de capturer les différents raisonnements des preuves de sécurité, comme la simulation, la réduction, la gestion des oracles, et les adversaires adaptatifs. Ils utilisent CIL pour prouver le schéma PS (*Probabilistic Signature Scheme*) un système de signature très utilisé faisant partie du standard PKCS [25]. CIL utilise la logique de Courant et al. [37], qui développe un calcul de plus forte post-condition pour établir automatiquement la sécurité asymptotique (IND-CPA et IND-CCA2) des algorithmes de chiffrement, qui utilise les fonctions *one-way* et les fonctions de hachage modélisées comme des oracles aléatoires. Ils prouvent la correction de leur logique, et écrivent un prototype qui couvre de nombreux exemples de la littérature. La formalisation de CIL en Coq est en cours.

7.2 Conclusion

CertiCrypt est une bibliothèque entièrement formalisée en Coq, permettant de faire des preuves de primitives cryptographiques. Dans cette thèse nous avons présenté deux travaux de formalisation. Les preuves introductives du schéma de chiffrement Hashed ElGamal dans le modèle standard et le modèle des oracles aléatoires. Nous avons également présenté la formalisation des Σ protocoles, permettant de convaincre une personne que nous connaissons un secret sans dévoiler ce secret.

Nous avons également formalisé deux classes de programmes permettant de définir des fonctions polynomiales. Nous avons prouvé les relations entre ces classes et nous avons étendu le langage de CertiCrypt pour utiliser la notion de fonctions polynomiales dans les preuves.

Les développements dans CertiCrypt ont montré que faire des preuves de schémas cryptographiques sur machine est possible. Mais l'outil est inutilisable pour des utilisateurs qui ne sont pas experts en Coq et difficile à prendre en main.

La principale contribution de cette thèse à été de développer un autre outil appelé EasyCrypt qui est plus simple à utiliser et plus proche des preuves papier. EasyCrypt permet d'automatiser la vérification de certaines étapes de preuve en utilisant des prouveurs automatiques. EasyCrypt peut être vu comme une sur couche à CertiCrypt, faisant la plupart du travail puis en générant des fichiers de preuves Coq compatibles avec la bibliothèque CertiCrypt.

Nous pensons que EasyCrypt est proche du programme d'Halevi et pourrait être adopté un jour par les cryptographes.

7.3 Perspectives

De nombreuses directions sont encore à explorer. EasyCrypt est aujourd'hui utilisable, mais peut être grandement amélioré.

Durant la rédaction de cette thèse beaucoup de travaux ont été entrepris :

- Guido Genzone travaille sur une interface graphique utilisant ProofGeneral [2] ce qui permet d'utiliser EasyCrypt à travers Emacs, comme le fait l'assistant de preuve Coq.
- César Kunz travaille sur l'ajout de boucle annotée par un invariant dans EasyCrypt. Mais la partie génération de preuve, expliquée dans le Chapitre 5, n'a pas encore été étendue.
- Gille Barthe, Boris Köpf, Federico Olmedo et Santiago Zanella Béguelin ont travaillé sur une version de CertiCrypt appelée CertiPriv. La sémantique est modifiée pour raisonner sur la notion de Differential Privacy qui permet de garantir la confidentialité des données à travers différents calculs.
- La formalisation des preuves de sécurité des candidats de la fonction de hashage SHA3 [1] est également en cours dans EasyCrypt.
- Manuel Barbosa et al. travaillent sur une nouvelle version du compilateur présenté dans [4] qui est vérifié formellement dans CertiCrypt.

D'autres preuves de schémas doivent être ajoutées dans EasyCrypt, pour tester l'outil. Ces preuves serviront d'exemple et aideront les utilisateurs à prendre en main l'outil.

Pour le moment EasyCrypt n'a servit qu'a prouver des schémas, mais nous pensons qu'il est possible d'étendre EasyCrypt aux preuves de protocoles sans trop de difficulté.

Nous pouvons aussi étendre le langage de EasyCrypt à un langage plus riche, par exemple une version de C minimaliste, sans *cast*, ni *alias*, ni pointeur. Cette extension permettrait d'utiliser EasyCrypt sur des implémentations concrètes.

Littérature

[1] NIST cryptographic hash algorithm competition. `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[2] Proof general. Online – `http://proofgeneral.inf.ed.ac.uk`.

[3] Abadi, M., and Rogaway, P. Reconciling two views of cryptography (The computational soundness of formal encryption). *J. Cryptology 15*, 2 (2002), 103–127.

[4] Almeida, J., Bangerter, E., Barbosa, M., Krenn, S., Sadeghi, A.-R., and Schneider, T. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. In *Computer Security – ESORICS 2010, 15th European Symposium on Research In Computer Security* (Berlin, 2010), vol. 6345 of *Lecture Notes in Computer Science*, Springer, pp. 151–167.

[5] Arai, T., and Eguchi, N. A new function algebra of exptime functions by safe nested recursion. *ACM Trans. Comput. Log. 10*, 4 (2009).

[6] Armand, M., Faure, G., Grégoire, B., Keller, C., Spiwack, A., Théry, L., and Werner, B. A modular integration of sat/smt solvers to coq through proof witnesses. In *1st International Conference on Certified Programs and Proofs – CPP 2011* (2011). To appear.

[7] Armand, M., Grégoire, B., Spiwack, A., and Théry, L. Extending coq with imperative features and its application to sat verification. In *ITP* (2010), pp. 83–98.

[8] Audebaud, P., and Paulin-Mohring, C. Proofs of randomized algorithms in Coq. *Sci. Comput. Program. 74*, 8 (2009), 568–589.

[9] Avanzini, M., and Moser, G. Complexity analysis by rewriting. In *FLOPS* (2008), pp. 130–146.

[10] Backes, M., Grochulla, M. P., Hritcu, C., and Maffei, M. Achieving security despite compromise using zero-knowledge. In *22nd IEEE Computer Security Foundations symposium, CSF 2009* (2009), IEEE Computer Society, pp. 308–323.

[11] Backes, M., Hritcu, C., and Maffei, M. Type-checking zero-knowledge. In *15th ACM conference on Computer and Communications Security, CCS 2008* (2008), ACM, pp. 357–370.

[12] Backes, M., Maffei, M., and Unruh, D. Zero-knowledge in the applied pi-calculus and automated verification of the Direct Anonymous Attestation protocol. In *29th IEEE symposium on Security and Privacy, S&P 2008* (2008), IEEE Computer Society, pp. 202 –215.

[13] Backes, M., Maffei, M., and Unruh, D. Computationally sound verification of source code. In *17th ACM conference on Computer and Communications Security, CCS 2010* (New York, 2010), ACM, pp. 387–398.

[14] Bangerter, E., Camenisch, J., and Krenn, S. Efficiency limitations for Sigma-protocols for group homomorphisms. In *7th Theory of Cryptography conference, TCC 2010* (2010), vol. 5978 of *Lecture Notes in Computer Science*, Springer, pp. 553–571.

[15] Bangerter, E., Camenisch, J., Krenn, S., Sadeghi, A.-R., and Schneider, T. Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, 2008.

[16] Barthe, G., Cederquist, J., and Tarento, S. A machine-checked formalization of the generic model and the random oracle model. In *Automated Reasoning, 2nd International Joint conference, IJCAR 2004* (Berlin, 2004), vol. 3097 of *Lecture Notes in Computer Science*, Springer, pp. 385–399.

[17] Barthe, G., D'Argenio, P., and Rezk, T. Secure information flow by self-composition. In *17th IEEE workshop on Computer Security Foundations, CSFW 2004* (Washington, 2004), IEEE Computer Society, pp. 100–114.

[18] Barthe, G., Daubignard, M., Kapron, B., and Lakhnech, Y. Computational indistinguishability logic. In *17th ACM conference on Computer and Communications Security, CCS 2010* (New York, 2010), ACM.

[19] Barthe, G., Grégoire, B., Heraud, S., and Zanella Béguelin, S. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *5th International workshop on Formal Aspects in Security and Trust, FAST 2008* (Berlin, 2009), vol. 5491 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

[20] Barthe, G., Grégoire, B., Lakhnech, Y., and Zanella Béguelin, S. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Topics in Cryptology — CT-RSA 2011* (Berlin, 2011), vol. 6558 of *Lecture Notes in Computer Science*, Springer, pp. 180–196.

[21] Barthe, G., Grégoire, B., and Zanella Béguelin, S. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2009* (New York, 2009), ACM, pp. 90–101.

[22] Barthe, G., Hedin, D., Zanella Béguelin, S., Grégoire, B., and Heraud, S. A machine-checked formalization of Sigma-protocols. In *23rd IEEE Computer Security Foundations symposium, CSF 2010* (Los Alamitos, Calif., 2010), IEEE Computer Society, pp. 246–260.

[23] Barthe, G., Olmedo, F., and Zanella Béguelin, S. Verifiable security of boneh-franklin identity-based encryption. In *5th International Conference on Provable Security – ProvSec 2011* (2011), Lecture Notes in Computer Science, Springer. To appear.

[24] Bellantoni, S., and Cook, S. A. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity 2* (1992), 97–110.

[25] Bellare, M., and Rogaway, P. The exact security of digital signatures – How to sign with RSA and Rabin. In *Advances in Cryptology – EUROCRYPT*

1996 (Berlin, 1996), vol. 1070 of *Lecture Notes in Computer Science*, Springer, pp. 399–416.

[26] Bellare, M., and Rogaway, P. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006* (Berlin, 2006), vol. 4004 of *Lecture Notes in Computer Science*, Springer, pp. 409–426.

[27] Benton, N. Simple relational correctness proofs for static analyses and program transformations. In *31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2004* (New York, 2004), ACM, pp. 14–25.

[28] Bhargavan, K., Fournet, C., and Gordon, A. D. Modular verification of security protocol code by typing. In *37th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL 2010* (2010), ACM, pp. 445–456.

[29] Blanchet, B., Jaggard, A. D., Scedrov, A., and Tsay, J.-K. Computationally sound mechanized proofs for basic and public-key Kerberos. In *15th ACM conference on Computer and Communications Security, CCS 2008* (New York, 2008), ACM, pp. 87–99.

[30] Blum, M., Feldman, P., and Micali, S. Non-interactive zero-knowledge and its applications. In *20th Annual ACM symposium on Theory of computing, STOC 1988* (1988), ACM, pp. 103–112.

[31] Brands, S. Untraceable off-line cash in wallet with observers. In *CRYPTO 93* (1994), D. R. Stinson, Ed., vol. 773 of *LNCS*, Springer, pp. 302–318.

[32] Brier, E., Coron, J.-S., Icart, T., Madore, D., Randriam, H., and Tibouchi, M. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology – CRYPTO 2010* (2010), vol. 6223 of *Lecture Notes in Computer Science*, Springer, pp. 237–254.

[33] Cobham, A. The intrinsic computational difficulty of functions. In *Congress for Logic, Mathematics and Philosophy of science* (1964), pp. 24–30.

[34] Conchon, S., Contejean, E., Kanig, J., and Lescuyer, S. CC(X) : Semantic combination of congruence closure with solvable theories. *Electronic Notes in Theoretical Computer Science 198*, 2 (2008), 51–69.

[35] Corin, R., and den Hartog, J. A probabilistic Hoare-style logic for game-based cryptographic proofs. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006* (2006), vol. 4052 of *Lecture Notes in Computer Science*, Springer, pp. 252–263.

[36] Cortier, V., and Warinschi, B. Computationally sound, automated proofs for security protocols. In *Programming Languages and Systems, 14th European symposium on Programming, ESOP 2005* (2005), vol. 3444 of *Lecture Notes in Computer Science*, Springer, pp. 157–171.

[37] Courant, J., Daubignard, M., Ene, C., Lafourcade, P., and Lakhnech, Y. Towards automated proofs for asymmetric encryption schemes

in the random oracle model. In *15th ACM conference on Computer and Communications Security, CCS 2008* (New York, 2008), ACM, pp. 371–380.

[38] Cramer, R. *Modular Design of Secure yet Practical Cryptographic Protocols.* PhD thesis, CWI and Uni. of Amsterdam, 1996.

[39] Cremers, C. The Scyther Tool : Verification, falsification, and analysis of security protocols. In *20th International Conference on Computer Aided Verification, CAV 2008* (Heidelberg, 2008), vol. 5123 of *Lecture Notes in Computer Science*, Springer, pp. 414–418.

[40] Damgård, I. On the existence of bit commitment schemes and zero-knowledge proofs. In *Advances in Cryptology – CRYPTO 1989* (1990), vol. 435 of *Lecture Notes in Computer Science*, Springer, pp. 17–27.

[41] Damgård, I. Efficient concurrent zero-knowledge in the auxiliary string model. In *Advances in Cryptology – EUROCRYPT 2000* (2000), vol. 1807 of *Lecture Notes in Computer Science*, Springer, pp. 418–430.

[42] Damgård, I. On sigma-protocols. Lecture Notes on Cryptologic Protocol Theory, 2010.

[43] David Detlefs, Greg Nelson, J. B. S. Simplify : A theorem prover for program checking. Tech. Rep. HPL-2003-148, HP Laboratories Palo Alto, 2003.

[44] Feige, U., Fiat, A., and Shamir, A. Zero-knowledge proofs of identity. *J. Cryptology 1*, 2 (1988), 77–94.

[45] Fiat, A., and Shamir, A. How to prove yourself : practical solutions to identification and signature problems. In *Advances in Cryptology – CRYPTO 1986* (1987), Springer, pp. 186–194.

[46] Filliâtre, J.-C. The WHY verification tool : Tutorial and Reference Manual Version 2.28. Online – `http://why.lri.fr`, 2010.

[47] Garay, J. A., MacKenzie, P., and Yang, K. Strengthening zero-knowledge protocols using signatures. *J. Cryptology 19* (2006), 169–209.

[48] Goldreich, O. Zero-knowledge twenty years after its invention. Tech. Rep. TR02-063, Electronic Colloquium on Computational Complexity, 2002.

[49] Goldreich, O., and Oren, Y. Definitions and properties of zero-knowledge proof systems. *J. Cryptology 7*, 1 (1994), 1–32.

[50] Goldwasser, S., and Micali, S. Probabilistic encryption. *J. Comput. Syst. Sci. 28*, 2 (1984), 270–299.

[51] Goldwasser, S., Micali, S., and Rackoff, C. The knowledge complexity of interactive proof systems. *SIAM J. Comput. 18*, 1 (1989), 186–208.

[52] Grégoire, B., and Mahboubi, A. Proving equalities in a commutative ring done right in coq. In *TPHOLs* (2005), pp. 98–113.

[53] Guillou, L., and Quisquater, J.-J. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *Advances in Cryptology – EUROCRYPT 1988* (1988), vol. 330 of *Lecture Notes in Computer Science*, Springer, pp. 123–128.

[54] Halevi, S. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

[55] Han, W., Chen, K., and Zheng, D. Receipt-freeness for Groth e-voting schemes. *Journal of Information Science and Engineering 25*, 2 (2009), 517–530.

[56] Heraud, S., and Nowak, D. A formalization of polytime functions. In *ITP* (2011), pp. 119–134.

[57] Hofmann, M. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *11th International workshop on Computer Science Logic, CSL 1997* (Berlin, 1998), vol. 1414 of *Lecture Notes in Computer Science*, Springer, pp. 275–294.

[58] Hofmann, M. Safe recursion with higher types and bck-algebra. *Ann. Pure Appl. Logic 104*, 1-3 (2000), 113–166.

[59] Icart, T. How to hash into elliptic curves. In *Advances in Cryptology – CRYPTO 2009* (2009), vol. 5677 of *Lecture Notes in Computer Science*, Springer, pp. 303–316.

[60] Icart, T. *Algorithms Mapping into Elliptic Curves and Applications.* PhD thesis, Université du Luxembourg, 2010.

[61] Impagliazzo, R., and Kapron, B. M. Logics for reasoning about cryptographic constructions. *J. Comput. Syst. Sci. 72*, 2 (2006), 286–320.

[62] Kikuchi, H., Nagai, K., Ogata, W., and Nishigaki, M. Privacy-preserving similarity evaluation and application to remote biometrics authentication. *Soft Computing 14*, 5 (2010), 529–536.

[63] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4 : formal verification of an OS kernel. In *22nd ACM symposium on Operating Systems Principles, SOSP 2009* (2009), ACM Press, pp. 207–220.

[64] Leivant, D. A foundational delineation of computational feasibility. In *Logic in Computer Science, 1991. LICS '91., Proceedings of Sixth Annual IEEE Symposium on* (july 1991), pp. 2 –11.

[65] Leroy, X. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL 2006* (New York, 2006), ACM, pp. 42–54.

[66] Lindell, Y., Pinkas, B., and Smart, N. P. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN 08* (2008), R. Ostrovsky, R. D. Prisco, and I. Visconti, Eds., vol. 5229 of *LNCS*, Springer, pp. 2–20.

[67] Maurer, U. Unifying zero-knowledge proofs of knowledge. In *Progress in Cryptology – AFRICACRYPT 2009* (2009), vol. 5580 of *Lecture Notes in Computer Science*, Springer, pp. 272–286.

[68] Maurer, U. M. Towards the equivalence of breaking the diffie-hellman protocol and computing discrete algorithms. In *CRYPTO* (1994), pp. 271–281.

[69] Mitchell, J. C., Mitchell, M., and Scedrov, A. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *FOCS* (1998), pp. 725–733.

[70] Nowak, D. A framework for game-based security proofs. In *9th International conference on Information and Communications Security, ICICS 2007* (Berlin, 2007), vol. 4861 of *Lecture Notes in Computer Science*, Springer, pp. 319–333.

[71] Nowak, D., and Zhang, Y. A calculus for game-based security proofs. In *ProvSec* (2010), pp. 35–52.

[72] Okamoto, T. Provably secure and practical identification schemes and corresponding signature schemes. In *Advances in Cryptology – CRYPTO 1992* (1993), vol. 740 of *Lecture Notes in Computer Science*, Springer, pp. 31–53.

[73] Paulson, L. C. The inductive approach to verifying cryptographic protocols. *J. of Comput. Secur. 6*, 1-2 (1998), 85–128.

[74] Rose, H. E. Subrecursion : functions and hierarchies. In *Oxford Logic Guides 9, Clarendon Press, Oxford* (1984).

[75] Sabelfeld, A., and Sands, D. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation 14*, 1 (2001), 59–91.

[76] Schnorr, C.-P. Efficient signature generation by smart cards. *J. Cryptology 4*, 3 (1991), 161–174.

[77] Schürmann, C., and Shah, J. Representing reductions of np-complete problems in logical frameworks : a case study. In *MERLIN* (2003).

[78] Schürmann, C., and Shah, J. Identifying polynomial-time recursive functions. In *CSL* (2005), pp. 525–540.

[79] Shamir, A., and Diffie, N. A polynomial-time algorithm for breaking the basic merkle-hellman cryptosystem. In *In Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science* (1982), IEEE, pp. 145–152.

[80] Shoup, V. Sequences of games : a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.

[81] Stern, J. Why provable security matters ? In *Advances in Cryptology – EUROCRYPT 2003* (Berlin, 2003), vol. 2656 of *Lecture Notes in Computer Science*, Springer, pp. 644–644.

[82] Stump, A. Proof checking technology for satisfiability modulo theories. *Electr. Notes Theor. Comput. Sci. 228* (2009), 121–133.

[83] The Coq development team. The Coq Proof Assistant Reference Manual Version 8.3. Online – `http://coq.inria.fr`, 2010.

[84] Théry, L. Proving the group law for elliptic curves formally. Technical Report RT-0330, INRIA, 2007.

[85] Tourlakis, G. J. Computability. In *Reston* (1984).

[86] Tristan, J.-B., and Leroy, X. Formal verification of translation validators : A case study on instruction scheduling optimizations. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)* (Jan. 2008), ACM Press, pp. 17–27.

[87] Vaudenay, S. Cryptanalysis of the chor-rivest cryptosystem. In *CRYPTO '98* (1998), Springer-Verlag, pp. 243–256.

[88] Zanella Béguelin, S. *Formal Certification of Game-Based Cryptographic Proofs*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2010.

[89] Zanella Béguelin, S., Grégoire, B., Barthe, G., and Olmedo, F. Formally certifying the security of digital signature schemes. In *30th IEEE symposium on Security and Privacy, S&P 2009* (Los Alamitos, Calif., 2009), IEEE Computer Society, pp. 237–250.

[90] Zhang, Y. The computational SLR : A logic for reasoning about computational indistinguishability. In *8th International conference on Typed Lambda Calculi and Applications, TLCA 2008* (Berlin, 2009), vol. 5608 of *Lecture Notes in Computer Science*, Springer, pp. 401–415.