École normale supérieure
Département d'Informatique

Équipe CASCADE
INRIA

Université Paris 7
Denis Diderot

Étude de protocoles cryptographiques à base de mots de passe

Thèse

présentée et soutenue publiquement le 3 décembre 2009

pour l'obtention du

Doctorat de l'Université Paris 7 Denis Diderot
(spécialité Informatique)

Céline CHEVALIER

Composition du jury :

Directeur de thèse :	David Pointcheval	(CNRS/ENS/INRIA)
Rapporteurs :	Jean-Sébastien Coron	(Université de Luxembourg)
	Louis Goubin	(Université de Versailles)
Examinateurs :	Xavier Boyen	(Stanford University)
	Emmanuel Bresson	(ANSSI)
	Arnaud Durand	(Université Paris 7 Denis Diderot)
	Jacques Stern	(ENS)

Travaux effectués au Laboratoire d'Informatique de l'École normale supérieure
45 rue d'Ulm, 75230 Paris Cedex 05

École normale supérieure
Département d'Informatique

Équipe CASCADE
INRIA

Université Paris 7
Denis Diderot

Étude de protocoles cryptographiques à base de mots de passe

Thèse

présentée et soutenue publiquement le 3 décembre 2009

pour l'obtention du

Doctorat de l'Université Paris 7 Denis Diderot
(spécialité Informatique)

Céline CHEVALIER

Composition du jury :

Directeur de thèse :	David Pointcheval	(CNRS/ENS/INRIA)
Rapporteurs :	Jean-Sébastien Coron	(Université de Luxembourg)
	Louis Goubin	(Université de Versailles)
Examinateurs :	Xavier Boyen	(Stanford University)
	Emmanuel Bresson	(ANSSI)
	Arnaud Durand	(Université Paris 7 Denis Diderot)
	Jacques Stern	(ENS)

Travaux effectués au Laboratoire d'Informatique de l'École normale supérieure
45 rue d'Ulm, 75230 Paris Cedex 05

Remerciements

Cette page incontournable est sans doute la plus difficile à écrire dans un mémoire. D'une part, parce que la tradition veut que ce soit à peu près la seule lue par plus de 10 personnes, et d'autre part parce que la crainte d'oublier quelqu'un est omniprésente. Pour ma part, certains oublis sont volontaires, car trop personnels, trop anciens, et concernant des personnes qui n'auront jamais l'occasion de lire ces lignes, mais qui malgré tout ont toute ma reconnaissance depuis plusieurs années. Les autres seraient bien malgré moi et je prie leurs éventuelles victimes de bien vouloir m'en excuser par avance.

Mes premiers remerciements vont tout naturellement à Jacques Stern, puis David Pointcheval, qui m'ont accueillie dans le laboratoire pour mon stage de M2, et à Dario Catalano qui l'a encadré avec enthousiasme et gentillesse. Merci également à David Pointcheval, cette fois-ci en tant que directeur de thèse, pour avoir guidé mon travail au quotidien. J'ai beaucoup appris à ses côtés. Ce fut un plaisir de travailler en sa présence, sous ses encouragements, sa foi inébranlable dans l'existence de solutions, ses bonnes idées, sa disponibilité sans faille et son exemple. Les jours de deadline resteront en particulier de très bons souvenirs avec cette manie de toujours trouver au dernier moment les problèmes (et les solutions !) de nos protocoles. C'est l'occasion de remercier aussi mes coauteurs, Michel Abdalla, Xavier Boyen, Dario Catalano, Pierre-Alain Fouque, Georg Fuchsbauer et Sébastien Zimmer. Je tiens également à remercier Jean-Sébastien Coron et Louis Goubin pour avoir accepté d'être les rapporteurs de cette thèse, ainsi que Xavier Boyen, Emmanuel Bresson, Arnaud Durand et Jacques Stern de m'avoir fait l'amitié de participer à mon jury. De manière plus personnelle, merci à Thomas et à Seb pour le rôle qu'ils ont joué dans mon orientation.

En plus des personnes exceptionnelles avec qui l'occasion m'a été donnée de travailler, j'aimerais saluer les conditions excellentes du laboratoire, et remercier tous ceux qui participent à cette ambiance : les membres permanents, les thésards et post-docs et en particulier Aurore, Émeline, Malika et Aurélie avec qui j'ai passé de nombreux moments à discuter, ceux qui ont partagé mes bureaux successifs et en ont fait des endroits très agréables, Dario, Sébastien, Pierre-Alain, Gaëtan et Michel, le service informatique, et l'équipe administrative, en particulier Joëlle, Valérie et Michelle pour leur efficacité, leur gentillesse et leur patience, dans la préparation des missions en particulier. Et comme les conférences ne se réduisent pas qu'aux présentations, merci à ceux qui ont partagé ces voyages avec moi : les membres du laboratoire, mais aussi Nicolas, Tiphaine, Xavier, Sébastien (KJ), les thésards de Versailles et Ayoub.

Enfin, et parce que la vie ce n'est pas qu'une suite de 0 et de 1, je souhaiterais remercier Anne, pour nos discussions sur tout et rien autour de crêpes et sa façon de me remettre les pieds sur Terre avec « ses grecs », Manuel, pour nos discussions LaTeXiennes, la H&K Team, et en particulier Seb, JJ, Teteph, Alex, Walter et Vince, pour les nombreuses heures passées ensemble, à travailler ou non, de manière électronique ou réelle, parfois dans des lieux paradisiaques, mais toujours dans des environnements très sympathiques, et Michèle, pour les moments d'évasion et sa gentillesse.

Je conclurai cette page par les premiers dans mes pensées, ma famille et Olivier, qui m'offrent leur soutien permanent. Cette thèse leur doit beaucoup, qu'ils en soient ici remerciés.

Table des matières

Introduction

La cryptographie est souvent appelée « la science du secret ». Son nom a pour origine les mots grecs $\kappa\rho\upsilon\pi\tau\acute{o}\varsigma$ (*kruptos*, « caché ») et $\gamma\rho\acute{\alpha}\varphi\epsilon\iota\nu$ (*graphein*, « écrire »), ce qui révèle son objectif, souvent exprimé à travers trois fonctionnalités fondamentales. La première, la *confidentialité*, garantit que le caractère secret du message transmis est préservé ; la deuxième, l'*intégrité*, assure que le message n'a pas été modifié pendant la transmission ; et enfin la troisième, l'*authentification*, permet de vérifier l'identité de l'émetteur, et évite qu'un adversaire puisse se faire passer pour lui.

Un objectif premier de la cryptographie est donc de permettre l'établissement de *canaux de communication sécurisés*, qui garantissent ces trois propriétés. L'essentiel de cette thèse se concentre sur la troisième propriété.

Authentification

L'authentification est le procédé par lequel une personne dans un réseau se persuade de l'identité de son partenaire de communication. Dans la plupart des cas, cette étape n'est qu'un préalable à la communication réelle, et ce procédé est couplé avec la génération d'une *clef de session* secrète, que les partenaires peuvent ensuite utiliser pour s'échanger des messages en toute confidentialité et en étant certains de leur intégrité : on appelle cette action un *échange de clef*. Autrement dit, c'est l'étape qui permet de passer d'un réseau public de communication à un canal sûr (confidentiel, intègre et authentifié). Ceci est un problème central en pratique, et cette importance est reflétée par le nombre considérable d'articles qui s'y rapportent dans la littérature.

La génération de clefs de session a été étudiée initialement par Diffie et Hellman dans l'article fondateur [DH76], dans lequel ils considéraient un adversaire *passif* qui peut écouter la communication des participants (les joueurs), mais qui ne peut pas modifier les messages envoyés. Ils supposaient ainsi implicitement l'existence de canaux intègres et authentifiés pour assurer la communication entre les joueurs. Beaucoup de protocoles sûrs et efficaces sont connus dans ce cadre. On s'intéresse plus généralement à un adversaire beaucoup plus puissant (tel qu'un routeur sur Internet), qui peut modifier, effacer ou insérer des messages entre les joueurs. Un tel adversaire est dit *actif*.

L'authentification peut prendre diverses formes : il peut y avoir deux personnes en jeu, ou bien tout un groupe d'utilisateurs. Les mécanismes étant les mêmes dans les deux cas, on parlera essentiellement dans la suite de cette introduction du cas où les joueurs sont au nombre de deux. Les autres différences possibles sont les suivantes : l'authentification peut être unilatérale (une personne est convaincue de l'identité de l'autre) ou bien mutuelle (les deux personnes sont convaincues de l'identité de leur partenaire) ; les deux personnes peuvent partager au préalable un secret (tel qu'un mot de passe) ou non.

Ce dernier cas (pas de secret partagé au préalable) peut généralement se présenter sous deux aspects. Dans le premier cas, le protocole repose sur une PKI (*public-key infrastructure*, IGC en français, « infrastructure de gestion de clefs »), dans laquelle une autorité crée des couples clef privée/clef publique pour chaque utilisateur (cette méthode nécessite donc l'existence de canaux intègres entre l'autorité et ces derniers). Il consiste ensuite à vérifier qu'une personne connaît bien la clef privée associée à une clef publique : c'est par exemple la méthode employée dans les normes de sécurisation des échanges sur Internet (SSL, *Secure Socket Layer* puis TLS, *Transport Layer Security*), associée à un certificat.

Dans la seconde variante, chaque utilisateur génère un couple clef privée/clef publique pour lui-même, qu'il distribue aux autres (via des canaux supposés intègres). Ceci est utilisé dans le protocole de connexion sécurisée SSH (*Secure Shell*), qui impose un échange de clefs de chiffrement en début de connexion, en demandant un mot de passe à l'utilisateur. Le serveur peut alors vérifier que le couple (identité, mot de passe) est bien présent dans sa base de données. Cela suppose en particulier qu'un mauvais serveur peut apprendre le mot de passe du client en menant une attaque active dite *man-in-the-middle* (il s'agit d'intercaler un adversaire entre les deux interlocuteurs, qui peut modifier les messages envoyés) ; ainsi, il faut avoir confiance en la clef publique du serveur.

Ces protocoles ne sont donc pas des échanges de clefs à base de mots de passe, puisqu'ils dépendent d'une information supplémentaire. On se concentrera ici sur le cas où les joueurs partagent un secret commun, sous la forme d'une petite clef de faible entropie (contenant peu d'aléa, donc pas forcément difficile à deviner) : un mot de passe.

Échange de clefs à base de mots de passe

L'intérêt de ce type d'authentification est de permettre aux clients de s'authentifier à travers un procédé très léger, sans avoir besoin d'une quelconque infrastructure complexe pour la sécurité (de type PKI) ou de matériel complexe pour retenir une clef de forte entropie, mais simplement en utilisant un petit mot de passe qu'un utilisateur peut retenir facilement. Cette technique, l'*échange de clefs à base de mots de passe*, permet alors aux deux entités en présence (possédant le même mot de passe) de procéder à la création d'une clef de session secrète, cette fois de forte entropie, afin de créer un canal de communication sécurisé à travers lequel toute leur communication future va pouvoir transiter de manière intègre, confidentielle et authentifiée. Le point remarquable est que les deux entités n'ont besoin pour ce faire que de retenir de petits mots de passe (tels qu'un code pin à 4 chiffres), ce qui est particulièrement utile dans le cas des interactions humaines. La sécurité repose évidemment sur la fragilité du mot de passe, mais il est possible de prouver que c'est vraiment le seul maillon faible : autrement dit, quelqu'un qui n'a pas deviné le mot de passe n'apprend rien sur la clef. Le premier protocole proposé fut celui de Bellovin et Merritt [BM92] (voir page 37), bien que présenté sans preuve ; il fut à la base de beaucoup de travaux dans ce domaine.

On considère donc un scénario à plusieurs joueurs, dans lequel chaque paire de joueurs possède en commun un mot de passe choisi uniformément dans un petit *dictionnaire*. Les joueurs interagissent dans un réseau non sûr, dans lequel toutes les communications sont totalement contrôlées par un adversaire actif. Ceci signifie essentiellement que les joueurs ne communiquent pas directement les uns avec les autres, mais que tous les messages passent à travers l'adversaire. L'objectif des joueurs est de créer des clefs de session qu'ils peuvent ensuite utiliser pour communiquer entre eux de manière sûre et secrète. Dans ce schéma, le *succès* de l'adversaire (par exemple, sa chance d'apprendre la clef de session) n'est clairement pas négligeable : il peut en effet essayer de deviner le mot de passe d'un joueur et prendre sa place pour discuter avec l'autre. S'il a bien deviné, il obtient alors la clef de session. Comme le dictionnaire de mots de passe est petit (par exemple polynomial en le paramètre de sécurité), le succès de l'adversaire dans cette attaque naïve peut être très élevé. Ce type d'attaque est appelé une *attaque par dictionnaire en ligne*. Elle est inévitable dès lors que la sécurité dépend de mots de passe à faible entropie mais on peut limiter le nombre de tentatives en ligne. L'objectif des échanges de clefs authentifiés par des mots de passe est de restreindre l'adversaire à cette attaque exclusivement (on souhaite en particulier l'empêcher d'être capable de mener des attaques *hors ligne*). En outre, toute observation passive conduit à une clef de session impossible à deviner par un adversaire, malgré la petite taille du mot de passe : cela reviendrait sinon à une attaque par dictionnaire hors ligne réussie.

Cadre de sécurité UC

Pour parvenir à limiter l'adversaire à cette attaque, et plus généralement analyser la sécurité de ces protocoles, encore faut-il préciser qui est l'adversaire, quels pouvoirs on lui donne, et quels sont ses objectifs. C'est le principe des cadres de sécurité, qui ont donné lieu à des travaux menés parallèlement à la conception des protocoles. On peut définir deux grandes familles : d'une part, les cadres basés sur un jeu de sécurité (*Find-Then-Guess* ou FTG [BR94, BPR00, BR95] et *Real-or-Random* ou ROR [AFP05], voir pages 33 et 35), qui consistent à exécuter un jeu entre l'adversaire (qui essaye de réussir un certain défi) et son *challenger*, et d'autre part les cadres basés sur la simulation ([BMP00], voir page 35 et *Universal Composability* ou UC [Can01], voir page 58), qui essaient de prouver que la situation dans le monde réel est analogue à celle dans un monde idéalisé, sans considérer une action de l'adversaire en particulier, mais en regardant plutôt l'exécution dans son ensemble.

Tout ce mémoire est centré sur le cadre UC (« composabilité universelle » en français), qui est bien adapté à la cryptographie à base de mots de passe. Les intérêts sont multiples : ce cadre garantit la concurrence des protocoles, il modélise correctement les attaques par dictionnaire en ligne et les mots de passe sont choisis par un adversaire particulier dit *environnement*, ce qui n'impose rien sur leur nature : ils peuvent être choisis selon une distribution arbitraire, reliés entre eux, ou même réutilisés dans divers protocoles. Ce cadre fait apparaître deux principales difficultés : d'une part, il doit être possible d'extraire systématiquement le mot de passe d'un joueur dans la preuve, ce qui complique la conception des protocoles. D'autre part, dans les cadres traditionnels, la preuve ne continue que jusqu'à temps que l'adversaire gagne, la simulation s'arrête alors. Ici, elle doit continuer coûte que coûte de manière indistinguable dans tous les cas : ce sont cette fois-ci les preuves qui s'en trouvent complexifiées. Ce cadre permet toutefois de garantir une sécurité très forte sur le protocole, qui peut ensuite être exécuté dans un environnement arbitraire.

Cryptographie à clef publique à base de mots de passe

On voit donc l'intérêt des mots de passe pour la cryptographie à clef secrète : ils servent à élaborer la clef secrète en question, de façon très pratique et sûre. La question naturelle qui se pose alors est de savoir si l'on peut étendre cette utilisation au cadre à clef publique. Il semble naturel a priori de considérer que ceci est impossible à réaliser, puisque la clef publique va ici pouvoir servir de fonction test non-interactive : toute clef privée de faible entropie va alors succomber rapidement à une attaque par dictionnaire hors ligne. Cependant, nos travaux montrent que cela devient réalisable en se plaçant dans un cadre distribué.

Plan du mémoire

Ce mémoire s'intéresse donc à étudier l'authentification à base de mots de passe selon ces différents axes, regroupés en quatre parties :

État de l'art sur les échanges de clefs à base de mots de passe. Les protocoles d'échange de clefs ont suscité beaucoup d'intérêt dans la communauté cryptographique ces trente dernières années : la première partie de ce mémoire (page 17) consiste en une description des protocoles existant dans la littérature. Plus précisément, le chapitre 1 (page 17) est consacré à des rappels sur les primitives cryptographiques qui seront nécessaires dans la suite. Nous verrons ensuite les cadres de sécurité (chapitre 2 page 29), qui ont permis de formaliser le problème après une ère d'élaborations de protocoles informels (considérés comme sûrs tant qu'ils n'étaient pas cassés). Les principaux protocoles sont ensuite passés en revue (chapitre 3 page 37), puis leurs extensions dans le cas, non plus d'une situation client/serveur, mais plutôt d'un groupe d'utilisateurs souhaitant établir une communication (chapitre 4 page 47). Enfin, le cadre UC fera l'objet du chapitre 5 (page 58) en deux parties : d'abord une présentation générale, puis un état de l'art dans le cas particulier des échanges de clefs à base de mots de passe.

Échange de clefs pour deux joueurs (RSA 2008) et pour un groupe de joueurs (Africacrypt 2009). La deuxième partie de ce mémoire (page 89) étudie le cas des échanges dans le modèle de l'oracle aléatoire et du chiffrement idéal (éventuellement étiqueté). Nous prouvons dans ces modèles qu'une variante des protocoles [BCP03b] (chapitre 6 page 89) et [ABCP06] (chapitre 7 page 99) sont sûrs dans le cadre UC, avec certaines propriétés supplémentaires, telles que la sécurité adaptative (lorsque l'adversaire a le droit de corrompre les joueurs à tout moment au cours de l'exécution), et la *contributory*[1] dans le second cas, notion que nous définirons et qui consiste informellement à dire que l'adversaire ne peut pas biaiser la clef tant qu'il n'a pas corrompu un nombre suffisant de joueurs.

Mises en gage conditionnellement extractibles et équivocables (Crypto 2009). La troisième partie de ce mémoire (page 119) s'intéresse au modèle standard, sans oracle aléatoire ni chiffrement idéal. Dans un premier temps (chapitre 8 page 119), nous présentons une nouvelle primitive de mise en gage conditionnellement extractible et équivocable, ainsi que son utilisation conjointe avec les *smooth hash functions* telles que définies par Gennaro et Lindell dans [GL03]. Cette nouvelle notion va nous permettre de présenter dans le chapitre 9 (page 137) le premier protocole d'échange de clefs sûr dans le cadre UC contre les attaques adaptatives dans le modèle standard.

Extraction d'entropie d'un élément de groupe (Eurocrypt 2009). L'échange de clefs précédent aboutit à une clef de session uniformément distribuée dans un groupe cyclique, mais les clefs secrètes sont généralement utiles sous la forme d'une chaîne de bits aléatoire. Le chapitre 10 (page 149) de cette partie III va donc s'attacher à extraire à partir de là une chaîne de bits aléatoire, en montrant qu'il suffit pour cela de considérer environ la moitié des bits de poids faible de l'élément obtenu. Cet extracteur est appliqué à des groupes cycliques d'entiers et à des courbes elliptiques.

Cryptographie distribuée à base de mots de passe (PKC 2009). Cette dernière partie (page 165) est consacrée à l'étude de l'application des mots de passe dans le cadre de la cryptographie à clef publique. Nous définissons la nouvelle primitive dans le chapitre 11 (page 165), et donnons un exemple concret dans le chapitre 12 (page 175). Ce dernier se termine par une extension pour obtenir une autre instanciation de la primitive dans le cadre des groupes bilinéaires, plus utile en pratique, par exemple pour permettre de l'extraction de clef dans les schémas de chiffrement basés sur l'identité.

[1] Nous avons pris la convention dans ce mémoire de traduire les notions tant que c'était possible et clair. Pour les autres cas, plutôt que de chercher une paraphrase française douteuse et pas forcément compréhensible, nous avons préféré laisser la version anglaise (en italique), ce qui est aussi le cas des mots passés dans le « langage courant » (tel que *zero-knowledge*).

Première partie

État de l'art sur les protocoles d'échanges de clefs basés sur des mots de passe

Chapitre 1

Préliminaires

1.1 Introduction à la sécurité prouvée

1.1.1 Des lois de Kerckhoffs à la sécurité conditionnelle

En 1883, dans *La Cryptographie militaire*, Kerckhoffs énonçait les six lois suivantes :

1. « Le système doit être matériellement, sinon mathématiquement, indéchiffrable. »
2. « Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi. »
3. « La clef doit pouvoir en être communiquée et retenue sans le secours de notes écrites, et être changée ou modifiée au gré des correspondants. »
4. « Il faut qu'il soit applicable à la correspondance télégraphique. »
5. « Il faut qu'il soit portatif, et que son maniement ou son fonctionnement n'exige pas le concours de plusieurs personnes. »
6. « Enfin, il est nécessaire, vu les circonstances qui en commandent l'application, que le système soit d'un usage facile, ne demandant ni tension d'esprit, ni la connaissance d'une longue série de règles à observer. »

Les trois premières sont communément admises et peuvent être considérées comme la base de la cryptographie moderne. Claude Shannon, qui fonda dans les années 1940 la théorie de l'information, base de la cryptographie moderne, a plus tard réénoncé la deuxième sous la forme percutante : « l'adversaire connaît le système ».

On peut remarquer que les deuxième et troisième lois définissent le concept de cryptographie symétrique, c'est-à-dire de schémas tels que le chiffrement $\mathcal{E}$ et le déchiffrement $\mathcal{D}$ soient réciproques, et fonctionnent avec une clef identique k. En notant m le message et c le chiffré, on peut schématiser cette opération ainsi :

$$\begin{array}{ccccc} & & k & & k \\ & & \downarrow & & \downarrow \\ m \rightarrow & \boxed{\mathcal{E}} & \xrightarrow{\quad c \quad} & \boxed{\mathcal{D}} & \rightarrow m \end{array}$$

La sécurité d'un tel schéma est heuristique. Par sécurité, on entend ici « il est impossible de retrouver m à partir de c sans posséder la clef k. » On remarque la nécessité de s'accorder au préalable sur la clef : ce sera l'objet des « échanges de clefs » étudiés dans ce mémoire.

La cryptographie asymétrique, dont l'idée de base remonte à Diffie et Hellman dans *New Directions in Cryptography* en 1976 [DH76], revient à étendre la deuxième loi. Le principe est très simple. Tout utilisateur Bob doit posséder deux clefs : l'une, publique, est la clef de chiffrement, connue de tous. L'autre, privée, est la clef de déchiffrement, connue de lui seul. Lorsqu'Alice souhaite lui envoyer un message de façon privée, elle le chiffre avec sa clef publique (elle pose un cadenas sur le colis). Lorsque Bob le reçoit, il le déchiffre avec la clef privée (il ouvre le cadenas avec sa clef). La sécurité repose sur le fait que personne ne peut ouvrir le cadenas sans cette clef. La plupart du temps, ces algorithmes sont *probabilistes*, c'est-à-dire qu'Alice ajoute un aléa r :

$$\begin{array}{ccccc} & & k_e & & k_d \\ & & \downarrow & & \downarrow \\ m \rightarrow & \boxed{\mathcal{E}} & \xrightarrow{\quad c \quad} & \boxed{\mathcal{D}} & \rightarrow m \\ r \rightarrow & & & & \end{array}$$

Le chiffré mathématique se présente donc sous la forme $c = \mathcal{E}_{k_e}(m, r)$. La clef de chiffrement est publique et il existe un élément m unique satisfaisant la relation (éventuellement avec plusieurs aléas r). On ne peut donc pas garantir une sécurité parfaite dans laquelle la distribution de m serait indistinguable d'une distribution uniforme, puisque cette dernière est réduite à un point. En particulier, une recherche exhaustive sur m et r peut mener à m.

1.1.2 Paramètre de sécurité

Deux cadres sont principalement utilisés pour définir la sécurité d'un protocole. Ce dernier est dit *sûr au sens de la théorie de l'information* si l'adversaire n'obtient aucune information sur les messages échangés : il peut donc avoir une puissance de calcul infinie, cela ne change rien à son pouvoir. On dit au contraire qu'il est *calculatoirement sûr* si l'adversaire a accès à tous les messages échangés, mais qu'il ne peut rien en tirer à moins de savoir résoudre un problème calculatoire difficile.

D'après ce qu'on vient de voir, la sécurité parfaite (au sens de la théorie de l'information) est impossible pour le chiffrement asymétrique. Elle est donc conditionnelle, et repose sur une hypothèse algorithmique.

Le cadre d'évaluation des algorithmes est celui des machines de Turing à plusieurs rubans. Nous ne le précisons pas ici, puisqu'il sera vu en détail (avec quelques variantes) dans le chapitre 5 page 58. Le temps de calcul est le nombre de pas de calculs avant l'arrêt de la machine. Sauf exception, ce temps sera toujours polynomial, c'est-à-dire borné par un polynôme en fonction de la taille des données. Cette taille des paramètres prend souvent le nom de *paramètre de sécurité*, généralement noté k. Il s'agit par exemple de la taille des entiers considérés, ou de celles des messages envoyés, ou encore d'autres dimensions pertinentes suivant les cas. Ce paramètre permet d'avoir un nombre fini d'instances à considérer (il y a un nombre fini d'entiers ayant moins de k bits, ou de groupes à moins de 2^k éléments par exemple).

1.1.3 Problèmes classiques et hypothèses standards

Rappelons désormais certaines des hypothèses algorithmiques sur lesquelles reposent les primitives habituelles.

Notations. Si l'on appelle k le paramètre de sécurité, on dit qu'un événement est *négligeable* si sa probabilité est inférieure à l'inverse de tout polynôme en k. Si G est un ensemble fini, $x \xleftarrow{R} \mathrm{G}$ indique le processus de sélection de x uniformément et au hasard dans G (ce qui implique implicitement que G peut être énuméré efficacement). Enfin, on dit qu'une fonction est calculable efficacement si elle peut être calculée en un temps polynomial en k.

Factorisation. Soit n un module RSA, c'est-à-dire de la forme $n = pq$ avec p et q premiers. Le problème de la factorisation consiste à retrouver p et q à partir de n.

Problème RSA. Soient n un module RSA, $y \in \mathbb{Z}_n^*$ et $e \geqslant 3$ premier avec l'indicatrice d'Euler $\varphi(n)$. Le problème RSA consiste, à partir de y, en connaisant n et e, à trouver sa racine e-ième, c'est-à-dire $x \in \mathbb{Z}_n^*$ tel que $y \equiv x^e \bmod n$.

Dans toute la suite, $\mathbb{G} = \langle g \rangle$ est un groupe cyclique fini d'ordre premier q où q est grand (k bits), et g un générateur de ce groupe (noté multiplicativement).

Problème du logarithme discret. Soit $y \in \mathbb{G}$. Le problème du logarithme discret revient à trouver $a \in \mathbb{Z}_q$ tel que $g^a = y$.

Hypothèse CDH (Diffie-Hellman calculatoire). L'hypothèse CDH signifie qu'étant donné g^x et g^y, où x et y sont des éléments aléatoires de $\mathbb{Z}_q^*$, il est difficile de calculer g^{xy}. On modélise cela par l'expérience $\mathsf{Exp}^{\mathsf{cdh}}_{\mathbb{G}}(\mathcal{A})$, dans laquelle le challenger choisit aléatoirement x et y de façon uniforme, calcule $\mathrm{X} = g^x$, $\mathrm{Y} = g^y$ et donne X et Y à un adversaire $\mathcal{A}$. Soit Z la valeur retournée par $\mathcal{A}$. Alors le résultat de l'expérience est 1 si $\mathrm{Z} = g^{xy}$ et 0 sinon. La *probabilité de succès* de $\mathcal{A}$ contre le CDH est alors défini par $\mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(\mathcal{A}) = \Pr\left[\mathsf{Exp}^{\mathsf{cdh}}_{\mathbb{G}}(\mathcal{A}) = 1\right]$. L'hypothèse CDH consiste à supposer que cet avantage est négligeable pour tout adversaire polynomial.

Hypothèse DDH (Diffie-Hellman décisionnelle). L'hypothèse DDH signifie que quels que soient $x, y, z \in \mathbb{Z}_q^*$, il est calculatoirement difficile de distinguer les deux triplets (g^x, g^y, g^{xy}) et (g^x, g^y, g^z). On modélise cela par l'expérience $\mathsf{Exp}^{\mathsf{ddh}}_{\mathbb{G}}(\mathcal{A})$, dans laquelle le challenger choisit aléatoirement x, y et z de façon uniforme, ainsi qu'un bit b. Si $b = 1$, il envoie (g^x, g^y, g^{xy}) à l'adversaire, sinon il lui envoie (g^x, g^y, g^z). L'adversaire envoie alors un bit b' en réponse, et il a gagné si $b = b'$. L'*avantage* de $\mathcal{A}$ contre le DDH est alors défini par $\mathsf{Adv}^{\mathsf{ddh}}_{\mathbb{G}}(\mathcal{A}) = 2\left|\Pr\left[b = b'\right] - 1/2\right|$. L'hypothèse DDH consiste à supposer que cet avantage est négligeable pour tout adversaire polynomial.

Hypothèse DLin (linéaire décisionnelle [BBS04]). L'hypothèse DLin signifie qu'étant donné des éléments aléatoires $x, y, r, s \in \mathbb{Z}_q^*$ et $(g, f = g^x, h = g^y, f^r, h^s) \in \mathbb{G}^5$, il est calculatoirement difficile connaisant g^d de distinguer entre le cas où $d = r+s$ et celui où d est aléatoire. Un triplet de la forme (f^r, h^s, g^d) est appelé un *triplet linéaire* en base (f, g, h) si $d = r + s$.

Le problème du logarithme discret est plus fort que le CDH, lui-même plus fort que le DDH.

1.1.4 Preuves par réduction

Les hypothèses algorithmiques sont nécessaires : par exemple, si le problème RSA est facile, n'importe qui peut retrouver m à partir de son chiffré RSA c. Mais sont-elles suffisantes ? Ce sont les preuves de sécurité qui assurent qu'elles le sont bien : si un adversaire peut briser le secret d'un schéma de chiffrement, alors on peut l'utiliser pour casser l'hypothèse sous-jacente. Par contraposée, on en déduit alors qu'une hypothèse difficile assure la sécurité d'un schéma. Ce sont des preuves dites *par réduction*. Pour prouver la sécurité d'un schéma cryptographique, on doit alors préciser

- les hypothèses calculatoires (par exemple RSA) ;

- les notions de sécurité à garantir (en fonction du schéma, par exemple l'impossibilité de retrouver le clair à partir du seul chiffré et des données publiques) ;
- une réduction (comment un adversaire contre ce schéma aide à casser l'hypothèse).

On peut connaître ou non le coût exact de la réduction, ce qui la rend ou non réalisable en pratique. Bien entendu, plus une réduction est fine (c'est-à-dire que la complexité de l'attaquant de l'hypothèse n'est pas tellement plus élevée que celle de l'attaquant du schéma), meilleure elle est.

Plus précisément, si l'on considère un adversaire en temps t et si l'on obtient un attaquant contre le problème calculatoire en temps $\mathrm{T}(t)$, alors en théorie de la complexité T est polynomial. Si T est explicite, la sécurité est exacte, et s'il est petit (linéaire), elle est pratique.

1.1.5 Adversaire calculatoire et distingueur

Pour un problème calculatoire, l'espace des réponses grandit avec le paramètre de sécurité, et la probabilité de trouver la bonne réponse diminue d'autant. Le bon objet pour mesurer l'efficacité d'un attaquant est donc sa probabilité de succès Succ, comme pour le CDH.

Au contraire, dans un problème décisionnel, ou de distinction entre deux situations, l'attaquant a une probabilité $1/2$ de réussite, et l'espace des réponses est fixé, comme pour le DDH. La quantité pertinente dans ce cas est donc l'avantage du distingueur, qui vaut

$$\mathsf{Adv} = 2\,|\mathsf{Succ} - 1/2|$$

Dans les deux cas, la personne posant le problème à l'attaquant est appelée le *challenger*.

1.1.6 Simulateur

L'information recueillie lors d'une attaque est appelée la *vue* d'un adversaire. Pour garantir que des paramètres privés (telle qu'une clef de déchiffrement) restent à l'abri des attaques, on introduit la notion de *simulation*, sorte de version algorithmique de la sécurité de Shannon. Un simulateur est une machine capable de produire une vue indistinguable de celle d'un véritable attaquant sans jamais accéder aux secrets. On souhaite montrer que les vues sont essentiellement identiques, c'est-à-dire qu'on les considère comme des distributions de probabilité sur l'ensemble des vues possibles et qu'on parle de :

- simulation *parfaite* si les distributions sont identiques ;
- simulation *statistique* si les distributions ont une distance négligeable ;
- simulation *calculatoire* (ou *algorithmique*) si les distributions ne peuvent pas être distinguées de façon non négligeable par un test probabiliste.

1.1.7 Preuves par jeux

Dans les cas simples, une réduction peut être effectuée en indiquant en une seule fois comment utiliser l'adversaire pour aider à résoudre le problème difficile considéré. Mais, la plupart du temps, il faut distinguer des cas, et prendre en considération l'environnement de l'adversaire, souvent très différent de celui du problème difficile. Pour faire intervenir ces modifications, Shoup a introduit dans [Sho01b] et [Sho01a] la notion de *preuves par jeux*, dans lesquelles on modifie petit à petit l'environnement du protocole afin d'aboutir sur une situation similaire à celle du problème difficile, en quantifiant à chaque étape les différences. L'objectif est d'aboutir à un jeu final qui quantifie explicitement la probabilité de succès ou l'avantage de l'adversaire.

1.2 Fonctions usuelles

Nous présentons désormais les grandes familles de schémas cryptographiques qui seront utiles dans la suite de ce mémoire. Certaines de ces définitions seront informelles, ce qui sera suffisant pour la suite.

1.2.1 Fonctions à sens unique

Une fonction $f : x \mapsto f(x)$ est dite *à sens unique* (*one-way*) s'il est facile de calculer $y = f(x)$ à partir de x, mais s'il est difficile, à partir d'un élément y de l'ensemble d'arrivée, de trouver un élément x dans l'ensemble de départ tel que $y = f(x)$.

Une fonction à sens unique est en outre dite *à trappe* si, à l'aide d'une donnée secrète supplémentaire, appelée *la trappe*, cet antécédent est aisément calculable.

1.2.2 Fonctions de hachage

Informellement, une *fonction de hachage* est une fonction créant un condensé (le *haché*) de taille fixe d'un message de taille arbitraire. Sa principale application est de garantir l'*intégrité* du message. Une telle fonction doit généralement être *résistante aux collisions*, c'est-à-dire qu'il doit être difficile de savoir créer deux messages ayant le même haché. Elle peut aussi être *résistante à la première préimage*, ce qui signifie qu'il doit être difficile de savoir créer un message ayant un haché donné, ou *à la seconde préimage*, ce qui implique qu'il doit être difficile de trouver un deuxième message avec le même haché à partir d'un message donné et du haché correspondant.

1.2.3 Fonctions de hachage universelles

Une famille de fonctions de hachage universelles $\mathrm{UH} = \{(h_i)_{i\in\{0,1\}^d}\}$ avec $h_i : \{0,1\}^n \to \{0,1\}^k$, pour $i \in \{0,1\}^d$, est une famille de fonctions telles que, pour tout $x \neq y$ dans $\{0,1\}^n$, $\Pr_{i\in\{0,1\}^d}[h_i(x) = h_i(y)] \leqslant 1/2^k$.

On définit généralement pour chaque $i \in \{0,1\}^d$ une instance particulière $\mathrm{UH}_i : \{0,1\}^n \to \{0,1\}^k$ de la famille en fixant l'indice à utiliser dans une CRS (voir page 27).

Elles seront essentiellement utilisées pour faire de l'extraction d'entropie. En effet, l'une des propriétés intéressantes de ces familles de fonctions est que si un élément x est choisi uniformément au hasard dans $\{0,1\}^n$, alors la distribution de $\mathrm{UH}_i(x)$ est statistiquement proche de la distribution uniforme dans $\{0,1\}^k$. C'est le *Leftover Hash Lemma* [HILL99, IZ89]. Ceci sera détaillé dans le chapitre 10.

1.2.4 Schémas de chiffrement

Un schéma de chiffrement à clef publique est défini par les trois algorithmes suivants :

- un algorithme de génération de clefs $\mathcal{K}$: si k est le paramètre de sécurité, l'algorithme probabiliste $\mathcal{K}(1^k)$ renvoie une paire (clef publique, clef privée) notée $(\mathsf{pk}, \mathsf{sk})$.
- un algorithme de chiffrement $\mathcal{E}$: étant donné un message $m \in \mathcal{M}$ et une clef publique pk, $\mathcal{E}_{\mathsf{pk}}(m)$ produit un chiffré c de m. Cet algorithme peut éventuellement être probabiliste, auquel cas on utilise la notation $\mathcal{E}_{\mathsf{pk}}(m, r)$, où $r \in \Omega$ est l'aléa fourni à l'algorithme.
- un algorithme de déchiffrement $\mathcal{D}$: étant donné un chiffré c et la clef privée sk associée à pk, $\mathcal{D}_{\mathsf{sk}}(c)$ retourne le message clair m correspondant, ou $\perp$ pour un chiffré non valide. Cet algorithme est nécessairement déterministe, même si le chiffrement est probabiliste.

Notions de sécurité. L'objectif majeur d'un adversaire est bien entendu de retrouver l'intégralité du message clair à partir du chiffré et des informations publiques. La formalisation de cette notion consiste à dire que la fonction de chiffrement est *à sens unique*. C'est la propriété de *one-wayness* (OW) : pour tout adversaire $\mathcal{A}$, sa probabilité de succès d'inverser $\mathcal{E}_{\mathsf{pk}}$ sans connaître la clef privée sk est négligeable sur l'espace de probabilité $\mathcal{M}$ (ou $\mathcal{M} \times \Omega$) ainsi que sur le ruban aléatoire de $\mathcal{A}$:

$$\mathsf{Succ}^{\mathsf{OW}}(\mathcal{A}) = \Pr_{m,r}\left[(\mathsf{pk},\mathsf{sk}) \longleftarrow \mathcal{K}(1^k) : \mathcal{A}(\mathsf{pk}, \mathcal{E}_{\mathsf{pk}}(m,r)) = m\right]$$

Mais l'attaquant peut aussi se contenter d'une information partielle sur le message clair. On sait déjà que la sécurité parfaite est impossible, ce qui peut se formuler en disant qu'un

attaquant tout puissant ne peut pas préduire un seul bit du message clair. La *sécurité sémantique* (ou *indistinguabilité des chiffrés* IND) est une version calculatoire de la sécurité parfaite, et se contente de considérer les attaquants polynomiaux. L'avantage de l'attaquant contre la sécurité sémantique consiste en sa capacité à distinguer, entre deux messages de son choix, lequel correspond au chiffré donné par le challenger. Formellement, l'attaquant procède en deux temps dans le jeu de sécurité. Premièrement, à la vue de la clef publique, il choisit deux messages m_0 et m_1 de même taille. Le challenger choisit alors un bit b au hasard et lui donne le chiffré de m_b. L'attaquant renvoie alors un bit b' et son avantage est défini par

$$\mathsf{Adv}^{\mathsf{IND}}(\mathcal{A}) = \left| 2 \Pr_{b,r} \left[\begin{matrix} (\mathsf{pk},\mathsf{sk}) \longleftarrow \mathcal{K}(1^k), (m_0,m_1,s) \longleftarrow \mathcal{A}_1(\mathsf{pk}), \\ c = \mathcal{E}_{\mathsf{pk}}(m_b,r), b' = \mathcal{A}_2(m_0,m_1,s,c) : b' = b \end{matrix} \right] - 1 \right|$$

où s modélise simplement la transmission d'information entre les deux étapes successives de l'attaquant.

L'attaquant peut posséder différents moyens pour mener à bien cette attaque. D'abord, dans le contexte asymétrique, il peut obtenir le chiffré de tout message de son choix grâce à sa connaissance de la clef publique. Il peut donc mettre en œuvre une attaque dite *à clairs choisis* (*chosen-plaintext attack*, CPA) : on parle alors de $\mathsf{IND}-\mathsf{CPA}$. Mais on peut aussi lui donner accès à un oracle de déchiffrement, auquel cas l'attaque est *à chiffrés choisis* (*chosen-ciphertext attack*, CCA) : on parle alors de $\mathsf{IND}-\mathsf{CCA}$. Plus précisément, si cet accès n'est autorisé qu'avant la vue du challenge (dans la première étape de l'attaque), on parle d'attaque *non-adaptative* CCA_1. S'il est illimité (sauf bien sûr pour le challenge lui-même), l'attaque est *adaptative* CCA_2.

Une autre notion utile est la non-malléabilité NM. Informellement, elle consiste à empêcher un attaquant, étant donné un chiffré $c = \mathcal{E}_{\mathsf{pk}}(m)$, de produire un nouveau chiffré $c' = \mathcal{E}_{\mathsf{pk}}(m')$ tel que les messages m et m' satisfassent une relation particulière. La non-malléabilité est équivalente à la sécurité $\mathsf{IND}-\mathsf{CCA}$.

Chiffrement à clef publique étiqueté. Le chiffrement étiqueté (*labeled encryption* [Sho04]) est une variante de la notion standard de chiffrement qui prend en compte la présence d'étiquettes dans les algorithmes de chiffrement et de déchiffrement. Plus précisément, ces deux algorithmes acceptent un paramètre supplémentaire, auquel on fait référence en tant qu'*étiquette* (*label*), et l'algorithme de déchiffrement ne déchiffre correctement un chiffré que si son étiquette est la même que celle utilisée pour créer le chiffré.

La notion de sécurité pour le chiffrement étiqueté est similaire à celle des schémas de chiffrement standards. La principale différence est que, au lieu de soumettre une paire de messages (m_0, m_1), il doit fournir une paire $((m_0,\ell_0),(m_1,\ell_1))$ pour obtenir le chiffré challenge (c,ℓ). Quant à la sécurité à chiffrés choisis ($\mathsf{IND}-\mathsf{CCA}$), l'adversaire est aussi autorisé à faire une requête à l'oracle de déchiffrement sur n'importe quelle paire (c',ℓ') tant que $\ell' \neq \ell$ ou $c' \neq c$. Le lecteur est renvoyé à [CHK+05, AP06] pour les définitions formelles de sécurité.

Chiffrements ElGamal et Cramer-Shoup. Pour les besoins du chapitre 8, nous décrivons ici ces schémas en interprétant le chiffrement ElGamal comme un chiffrement extrait d'un chiffrement Cramer-Shoup. Une clef publique d'un schéma de Cramer-Shoup est définie par

$$\mathsf{pk} = (g_1, g_2, c = {g_1}^{x_1} {g_2}^{x_2}, d = {g_1}^{y_1} {g_2}^{y_2}, h = {g_1}^{z}, \mathrm{H})$$

où g_1 et g_2 sont des éléments du groupe $\mathbb{G} = \langle g \rangle$ aléatoires, x_1, x_2, y_1, y_2 et z des scalaires aléatoires de $\mathbb{Z}_q$, et H une fonction de hachage résistante aux collisions (en fait, la résistance à la seconde préimage suffit). La clef privée associée est alors $\mathsf{sk} = (x_1, x_2, y_1, y_2, z)$. On peut voir (g_1, h) comme la clef publique d'un chiffrement ElGamal, avec z la clef privée associée.

Si $\mathrm{M} \in \mathbb{G}$, le chiffrement ElGamal multiplicatif est défini par

$$\mathsf{EG}^{\times}_{\mathsf{pk}}(\mathrm{M}; r) = (u_1 = g_1^r, e = h^r \mathrm{M})$$

qui peut être déchiffré par $\mathrm{M} = e/u_1^z$. Si $\mathrm{M} \in \mathbb{Z}_q$, le chiffrement ElGamal additif est défini par

$$\mathsf{EG}^{+}_{\mathsf{pk}}(\mathrm{M}; r) = (u_1 = g_1^r, e = h^r g^{\mathrm{M}})$$

Remarquons que $\mathsf{EG}^{\times}_{\mathsf{pk}}(g^{\mathrm{M}};r) = \mathsf{EG}^{+}_{\mathsf{pk}}(\mathrm{M};r)$. Il peut être déchiffré après un calcul additionnel de logarithme discret : M doit être suffisamment petit.

De même, si $\mathrm{M} \in \mathbb{G}$, le chiffrement Cramer-Shoup étiqueté multiplicatif est défini par

$$\mathsf{CS}^{\times\,\ell}_{\mathsf{pk}}(\mathrm{M};r) = (u_1, u_2, e, v)$$

où $$u_1 = g_1^r,\ \ u_2 = g_2^r,\ \ e = h^r\mathrm{M},\ \ \theta = \mathrm{H}(\ell, u_1, u_2, e)\ \text{ et }\ v = (cd^{\theta})^r$$

Le déchiffrement fonctionne de la même manière, avec $\mathrm{M} = e/u_1^z$, mais seulement si le chiffré est valide : $v = u_1^{x_1+\theta y_1}u_2^{x_2+\theta y_2}$. Si $\mathrm{M} \in \mathbb{Z}_q$, son chiffrement additif $\mathsf{CS}^{+\,\ell}_{\mathsf{pk}}(\mathrm{M};r)$ est tel que $e = h^r g^{\mathrm{M}}$. La relation suivante est vérifiée : $\mathsf{CS}^{\times\,\ell}_{\mathsf{pk}}(g^{\mathrm{M}};r) = \mathsf{CS}^{+\,\ell}_{\mathsf{pk}}(\mathrm{M};r)$. Le déchiffrement fonctionne comme ci-dessus si M est suffisamment petit.

Comme on l'a déjà noté, on peut extraire depuis n'importe quel chiffré Cramer-Shoup (u_1, u_2, e, v) d'un message M avec l'aléa r, quelle que soit l'étiquette ℓ, un chiffré ElGamal (u_1, e) du même message M avec le même aléa r. Cette extraction s'applique aussi bien à la version additive qu'à la version multiplicative. Ceci vient du fait que le déchiffrement fonctionne de la même manière pour les chiffrés ElGamal ou Cramer-Shoup, excepté pour la vérification de validité qui procure le niveau de sécurité CCA au schéma de chiffrement Cramer-Shoup, tandis que le chiffrement ElGamal fournit une sécurité CPA seulement (les deux reposant sur l'hypothèse DDH).

Le chiffrement linéaire [BBS04] On se donne trois générateurs $g_1 = {g_3}^{1/x_1}$, $g_2 = {g_3}^{1/x_2}$ et g_3 de $\mathbb{G}$. La clef publique est $\mathsf{pk}_{\mathrm{lin}} = (g_1, g_2, g_3)$ et la clef privée $\mathsf{sk}_{\mathrm{lin}} = (x_1, x_2)$. On choisit alors r_1 et r_2 de façon aléatoire dans $\mathbb{Z}_p$ et on pose, pour $\mathrm{M} \in \mathbb{G}$,

$$\mathrm{C} = \mathcal{E}_{\mathsf{pk}_{\mathrm{lin}}}(\mathrm{M};r_1,r_2) = (u_1,u_2,u_3) = ({g_1}^{r_1}, {g_2}^{r_2}, \mathrm{M}{g_3}^{r_1+r_2})$$

Le déchiffrement est alors effectué de la façon suivante : $\mathrm{M} = \mathcal{D}_{\mathsf{pk}_{\mathrm{lin}}}(\mathrm{C}) = \dfrac{u_3}{{u_1}^{x_1}{u_2}^{x_2}}$. Ce schéma est $\mathsf{IND}-\mathsf{CPA}$ sous l'hypothèse DLin.

Le chiffrement RSA [RSA78]. Soient p et q deux nombres premiers distincts et secrets et $n = pq$ public. Soit e un entier public premier avec $\varphi(n)$ et d son inverse (secret) modulo $\varphi(n)$. Le couple $\mathsf{sk} = (n, e)$ est la clef publique, et $\mathsf{pk} = (n, d)$ la clef secrète. On pose alors, pour $\mathrm{M} \in \mathbb{Z}_n^*$, $\mathcal{E}_{\mathsf{pk}}(\mathrm{M}) = \mathrm{M}^e \bmod n$ et $\mathcal{D}_{\mathsf{sk}}(\mathrm{C}) = \mathrm{C}^d \bmod n$. Ce schéma est $\mathsf{OW}-\mathsf{CPA}$ sous l'hypothèse RSA.

1.2.5 Échanges de clefs

Cette primitive fera l'objet d'un développement dans les trois chapitres suivants ainsi qu'une bonne partie du mémoire, donc nous nous contentons ici d'une idée informelle et de l'exposé du plus célèbre protocole.

Un échange de clefs consiste en l'exécution conjointe d'un protocole par deux ou plusieurs utilisateurs afin d'aboutir à une clef commune alors que les messages circulent à travers un canal public non sécurisé. Il s'agit de l'étape préalable à un chiffrement à clef secrète, pour lequel les deux utilisateurs doivent se mettre d'accord sur la clef à utiliser au préalable.

Le premier et le plus connu est sans doute le protocole Diffie-Hellman [DH76], dans lequel Alice choisit au hasard $a \in \mathbb{Z}_q^*$ et envoie $\mathrm{A} = g^a$ à Bob. Bob choisit alors $b \in \mathbb{Z}_q^*$ aléatoirement et envoie $\mathrm{B} = g^b$ à Alice. Les deux sont ainsi capables de calculer leur clef commune g^{ab}. La sécurité de ce schéma repose sur les problèmes Diffie-Hellman exposés plus haut. Remarquons qu'il s'agit d'un échange *non-authentifié*, sensible à une attaque par le milieu (*man-in-the-middle*). Un adversaire peut en effet s'intercaler entre Alice et Bob afin de partager une clef avec chacun d'eux, sans que ces derniers ne s'en rendent compte :

Alice		Attaquant		Bob
$a \xleftarrow{\$} \mathbb{Z}_q$		$a', b' \xleftarrow{\$} \mathbb{Z}_q$		$b \xleftarrow{\$} \mathbb{Z}_q$
$\mathrm{A} = g^a$	$\xrightarrow{\mathrm{A}}$	$\mathrm{A}' = g^{a'}$	$\xrightarrow{\mathrm{A}'}$	
	$\xleftarrow{\mathrm{B}'}$	$\mathrm{B}' = g^{b'}$	$\xleftarrow{\mathrm{B}}$	$\mathrm{B} = g^b$
$\mathrm{K} = g^{ab'}$		K, K'		$\mathrm{K}' = g^{a'b}$

On peut facilement ajouter l'authentification à l'aide de signatures, et nous verrons dans ce mémoire d'autres types d'authentification à base de mots de passe.

1.2.6 Signatures

Un schéma de signature standard est composé de trois algorithmes $\mathsf{SIG} = (\mathsf{SKG}, \mathsf{Sign}, \mathsf{Ver})$. L'algorithme de génération de clefs SKG prend en entrée 1^k où k est le paramètre de sécurité et retourne une paire $(\mathsf{SK}, \mathsf{VK})$ contenant la clef secrète de signature et la clef publique de vérification. L'algorithme de signature Sign prend en entrée la clef secrète SK et un message m et retourne la signature σ de ce message. Enfin, l'algorithme de vérification Ver prend en entrée le triplet (VK, m, σ) et retourne 1 si et seulement si σ est une signature valide pour le message m par rapport à la clef de vérification VK. La notion de sécurité requise est $\mathsf{SUF} - \mathsf{CMA}$ (*strong existential unforgeability under chosen-message attacks*), c'est-à-dire l'impossibilité d'effectuer des contrefaçons existentielles à messages choisis. Cela signifie informellement qu'un adversaire, muni de l'oracle de signature et de la clef de vérification mais pas de la clef de signature, ne doit pas être capable de donner un couple (m, σ) tel que σ soit une signature valide de m (telle que m ouσ soit nouveau).

Signatures *one-time*. Elles sont définies exactement de la même manière sauf que l'accès à l'oracle de signature n'est permis qu'une seule fois.

1.2.7 Preuves *zero-knowledge*

Un *prédicat polynomial* est une relation $\mathrm{R}(x, y)$ qui peut être testée par une machine de Turing polynomiale (en fonction du paramètre de sécurité). On requiert de plus que les tailles de x et y restent polynomiales (toujours en fonction du paramètre de sécurité). Un problème de la classe NP consiste, sur une donnée x, en la recherche d'un élément y tel que $\mathrm{R}(x, y)$, où R est un prédicat polynomial.

On considère ici une situation dans laquelle un *prouveur* et un *vérifieur* connaissent une valeur publique x en commun (typiquement une clef publique). Le prouveur dispose en outre d'un élément y privé tel que $\mathrm{R}(x, y)$ soit vrai (la clef secrète associée). Cet élément est appelé un *témoin*. Le prouveur cherche à convaincre le vérifieur qu'il possède ce témoin, sans révéler quelqu'information que ce soit sur lui. À l'issue de leur interaction, le serveur accepte ou rejette (souvent de manière probabiliste).

Une telle preuve est alors dite *preuve de connaissance* pour le problème NP associé à R si elle satisfait les deux propriétés suivantes :

- Complétude : un prouveur connaissant effectivement le témoin est accepté par un vérifieur avec probabilité d'échec négligeable.
- Correction : un prouveur parvenant à se faire accepter par un vérifieur connaît un témoin. Formellement, on dit qu'on peut construire un simulateur, appelé *extracteur*, qui puisse, à partir d'un tel prouveur et en ayant accès à toutes les étapes de la preuve, lui extraire son témoin avec probabilité d'échec négligeable.

Elle est alors dite *zero-knowledge* (« à divulgation nulle de connaissance ») si elle satisfait en outre cette propriété supplémentaire :

- *zero-knowledge* : le secret de y est préservé lorsqu'un prouveur se fait accepter par un vérifieur (pas forcément honnête). Formellement, cela passe à nouveau par une simulation. La vue de l'adversaire est constituée de tous les messages échangés entre prouveur

et vérifieur, et l'on requiert que le simulateur soit capable de fournir une vue indistinguable, sans connaître le secret y. En fonction de la simulation, cette propriété peut être parfaite, statistique ou algorithmique.

1.2.8 Mises en gage

Les schémas de mise en gage (*commitments* en anglais) sont l'une des primitives cryptographiques fondamentales, utilisées autant pour des preuves *zero-knowledge* [GMW91] qu'en *multiparty computation* sûre [GMW87a]. Ils sont aussi utiles dans le *public-key registration setting* et les échanges de clefs [GL03], comme on le verra plus loin.

Avec un schéma de mise en gage, un joueur peut s'engager sur une valeur secrète x en publiant une valeur $\mathrm{C} = \mathsf{com}(x; r)$ avec l'aléa r, de telle sorte que C ne révèle rien à propos du secret x, ce qui s'appelle la propriété de *hiding*. Le joueur peut ensuite ouvrir C pour révéler x, en publiant x et un élément permettant l'ouverture de la mise en gage (*decommitment*), aussi appelé un *témoin*, de manière publiquement vérifiable. Le joueur ne doit pas pouvoir ouvrir C sur une valeur autre que x, ce qui est la propriété de *binding*. Dans la plupart des cas, l'ouverture consiste en l'aléa r lui-même ou une partie de sa valeur. On ne considère ici que des schémas de mise en gage dans le modèle de la CRS (décrit dans la section 1.4 page 27) dans lequel les paramètres communs (la CRS) sont générés honnêtement et accessibles à tous. Informellement, cela revient à dire que le joueur place sa valeur dans une enveloppe scellée, et que cette dernière sera ouverte plus tard pour révéler la valeur à l'intérieur.

Remarquons qu'un schéma de chiffrement à clef publique $\mathsf{IND}-\mathsf{CPA}$ procure une telle mise en gage : la propriété de *binding* est garantie par l'unicité du message clair (parfaitement *binding*), et la propriété de *hiding* est garantie par la sécurité $\mathsf{IND}-\mathsf{CPA}$ (calculatoirement *hiding*). Dans ce cas, la CRS consiste simplement en la clef publique du schéma de chiffrement. À l'inverse, le schéma de Pedersen $\mathrm{C} = \mathsf{com}(x; r) = g^r h^x$ (où g est un générateur de $\mathbb{G}$ et h un élément de $\mathbb{G}$) procure une mise en gage parfaitement *hiding* mais seulement calculatoirement *binding* sous l'hypothèse que le logarithme discret de h en base g soit difficile. Dans ce cas, la CRS consiste en l'élément h.

Présentons désormais des propriétés supplémentaires pouvant être satisfaites par la mise en gage. Elle est dite *extractible* s'il existe un algorithme efficace, appelé *extracteur*, capable de générer un nouvel ensemble de paramètres communs (c'est-à-dire une nouvelle CRS) de distribution équivalente à celle d'une CRS générée honnêtement, et telle qu'il puisse extraire la valeur engagée x depuis une mise en gage C. Ceci n'est évidemment possible que pour des mises en gage calculatoirement *hiding*, telles que des schémas de chiffrement : la clef de déchiffrement est la trappe d'extraction.

Ensuite, une mise en gage est dite *équivocable* s'il existe un algorithme efficace, appelé *équivocateur*, capable de générer une nouvelle CRS et une nouvelle mise en gage de distributions similaires à celles du schéma en question, et tels que la mise en gage puisse être ouverte de plusieurs manières. Encore une fois, ceci n'est possible que pour des mises en gage calculatoirement *binding*, telles que le schéma de Pedersen : la connaissance du logarithme discret de h en base g est une trappe qui permet l'ouverture de la mise en gage de plus d'une façon.

Enfin, une mise en gage *non-malléable* assure que si un adversaire qui reçoit une mise en gage C d'une certaine valeur inconnue x peut générer une mise en gage valide pour une valeur reliée y, alors un simulateur pourrait en faire de même sans avoir vu C. Un schéma de chiffrement à clef publique $\mathsf{IND}-\mathsf{CCA}$ procure une telle mise en gage non-malléable [GL03].

Pour les définitions formelles de sécurité des schémas de mise en gage, on recommande la lecture de [GL03, DKOS01, CF01].

1.2.9 *Smooth projective hash functions*

La notion de *smooth projective hash functions* (SPHF) a été proposée par Cramer et Shoup dans [CS02] et peut être vue comme un type spécifique de preuve *zero-knowledge* pour un langage. Elles ont originellement été utilisées comme un moyen de construire des schémas de

chiffrement à clef publique CCA efficaces. La nouvelle abstraction n'a pas seulement permis une description plus intuitive du schéma de chiffrement Cramer-Shoup original [CS98], elle a aussi donné lieu à une généralisation de ce dernier, et en particulier à de nouvelles instanciations basées sur différentes hypothèses de sécurité telles que la résiduosité quadratique ou la N-résiduosité [Pai99].

Des variations de ces fonctions ont aussi trouvé des applications dans plusieurs autres contextes, tels que les échanges de clefs authentifiés à base de mots de passe (PAKE, [GL03]) et l'*oblivious transfer* [Kal05]. Dans le contexte des PAKE, les travaux de Gennaro et Lindell [GL03], qui ont légèrement modifié cette notion de SPHF comme on le verra ici, ont abstrait et généralisé (sous différentes hypothèses d'indistinguabilité) le protocole original de Katz, Ostrovsky et Yung [KOY01] et sont devenus la base de plusieurs autres schémas [BCL$^+$05, AP06, BGS06]. Dans le contexte de l'*oblivious transfer*, le travail de Kalai [Kal05] a aussi généralisé d'anciens protocoles de Naor et Pinkas [NP01] et Aiello, Ishai, et Reingold [AIR01].

Principe des SPHF. On utilise ici les définitions de Gennaro et Lindell [GL03], tant pour des mises en gage non-malléables comme dans leur article que pour des chiffrements étiquetés, comme c'est fait par Canetti *et al.* [CHK$^+$05] et par Abdalla et Pointcheval [AP06]. Les définitions formelles sont données dans la section suivante ; nous nous contentons ici du principe directeur.

Soient X le domaine de ces fonctions et L un certain sous-ensemble de points de ce domaine (un langage). L doit être un langage NP, donc il doit être calculatoirement difficile de distinguer un élément aléatoire dans L d'un élément aléatoire dans $\mathrm{X} \setminus \mathrm{L}$. On dit à ce sujet que L est un *hard partitioned subset* de X (on parle aussi du problème de *hard partitioned subset membership*).

L'une des propriétés-clef qui rend les SPHF si utiles est que, pour un point $x \in \mathrm{L}$, la valeur hachée peut être calculée en utilisant soit une clef de hachage *secrète* hk, soit une clef projetée *publique* hp (dépendant de x [GL03] ou non [CS02]) associée à un témoin w du fait que $x \in \mathrm{L}$. Tandis que le calcul utilisant la clef de hachage *secrète* fonctionne pour tous les points du domaine X de la fonction de hachage, le calcul utilisant la clef projetée *publique* ne fonctionne que pour les points $x \in \mathrm{L}$ et requiert la connaissance de la valeur w témoin du fait que $x \in \mathrm{L}$.

Une famille de *projective hash functions* est dite *smooth* si la valeur de cette fonction sur des entrées en dehors du sous-ensemble L est indépendante de la clef projetée (étant donné cette clef, la valeur est statistiquement indistinguable d'une valeur aléatoire).

Pour les points $x \in \mathrm{L}$, la sortie est uniquement déterminée une fois donnée la clef projetée hp et un témoin w. En revanche, sans la connaisance de ce témoin, la sortie est *pseudo-aléatoire* (calculatoirement indistinguable d'une valeur aléatoire).

Dans le cas particulier du schéma Gennaro-Lindell [GL03], le sous-ensemble $\mathrm{L}_{\mathsf{pk},m}$ est défini comme l'ensemble $\{(c)\}$ des c qui sont des mises en gage de m utilisant pk en tant que paramètre public : il existe r tel que $c = \mathsf{com}_{\mathsf{pk}}(m; r)$ où com est l'algorithme de mise en gage du schéma. Dans le cas du schéma de Canetti *et al.* [CHK$^+$05], le sous-ensemble $\mathrm{L}_{\mathsf{pk},(\ell,m)}$ est défini comme l'ensemble $\{(c)\}$ des c qui sont des chiffrés de m avec l'étiquette ℓ, sous la clef publique pk : il existe r tel que $c = \mathcal{E}^{\ell}_{pk}(m; r)$ où $\mathcal{E}$ est l'algorithme de chiffrement du schéma de chiffrement étiqueté. Dans le cas d'un schéma standard de chiffrement, l'étiquette est simplement omise. La sécurité sémantique du schéma de chiffrement garantit l'indistinguabilité calculatoire entre les éléments de L et ceux de X. On renvoie le lecteur intéressé à [GL03, CHK$^+$05, AP06] pour plus de détails, ainsi qu'au chapitre 8 pour les définitions formelles de cette primitive, ainsi que son utilisation.

1.3 Modèles idéaux

L'oracle aléatoire, décrit par Bellare et Rogaway dans [BR93], est un modèle simplifiant les preuves de sécurité en donnant à tous les joueurs un accès en tant qu'oracle à une fonction aléatoire universelle de $\{0,1\}^*$ dans $\{0,1\}^n$. Autrement dit, l'utilisateur peut faire une requête à cet oracle, qui lui répond en choisissant une valeur aléatoire. Deux requêtes identiques doivent

obtenir la même réponse. En général, on interprète ce résultat en disant qu'un protocole sûr dans le modèle de l'oracle aléatoire continue à être vrai même si l'oracle aléatoire est remplacé par une fonction de hachage concrète et « raisonnable », explicitement connue de tous les joueurs. Cependant, les auteurs de [CGH98] ont montré qu'il était impossible de remplacer l'oracle aléatoire de façon générique par une fonction concrète, en exhibant un contre-exemple. Néanmoins ce contre-exemple était non-naturel. Ainsi, aucun schéma « raisonnable » prouvé sûr dans le modèle de l'oracle aléatoire n'a été montré faible avec une instanciation concrète de l'oracle aléatoire.

Un chiffrement idéal [BPR00, LRW02] est une famille de fonctions aléatoires d'un ensemble $\mathcal{G}$ vers un ensemble $\mathcal{C}$ tels que $|\mathcal{G}| = |\mathcal{C}|$ qui chiffrent parfaitement les éléments de $\mathcal{G}$. Autrement dit, si $\mathrm{K} \in \{0,1\}^*$, $\mathcal{E}_\mathrm{K} : \mathcal{G} \to \mathcal{C}$ est une fonction aléatoire bijective. On l'étend ensuite à $\{0,1\}^*$ en définissant $\mathcal{D}_\mathrm{K} : \{0,1\}^* \to \mathcal{G}$ de la façon suivante : si $y \in \mathcal{C}$, $\mathcal{D}_\mathrm{K}(y) = x$ avec x tel que $\mathcal{E}_\mathrm{K}(x) = y$, et $\mathcal{D}_\mathrm{K}(y) = \perp$ sinon. On a longtemps cru que le chiffrement idéal était un modèle plus fort que l'oracle aléatoire, mais Coron, Patarin et Seurin [CPS08] ont montré en 2008 que les deux modèles sont en fait équivalents (à une réduction coûteuse près).

Un chiffrement idéal étiqueté est le modèle idéal équivalent pour un chiffrement étiqueté, c'est-à-dire avec une étiquette ℓ en plus de la clef.

1.4 Modèle standard et CRS

Un protocole qui n'est pas dans l'un de ces modèles idéaux est dit *dans le modèle standard*. Il peut alors utiliser une *common reference string* : il s'agit d'un modèle dans lequel une chaîne de caractères est tirée avec une distribution donnée D, et est disponible à tous les utilisateurs. Il s'agit de s'assurer dans ce modèle que cette chaîne est générée correctement. C'est d'ailleurs dans ce sens qu'il simplifie les preuves étant donné qu'on autorise le simulateur à programmer cette chaîne et à y insérer une trappe le cas échéant.

Chapitre 2

Cadres traditionnels de sécurité

Nous rappelons dans ce chapitre le modèle de communication usuel des échanges de clefs pour deux joueurs en nous concentrant sur les protocoles basés sur des mots de passe, et décrivons ensuite les cadres de sécurité standards pour de tels schémas. Commençons par rappeler les circonstances du développement de ces cadres : nous verrons que l'élaboration des cadres a suivi de près celle des protocoles associés. Tous les termes « techniques » seront définis dans ce chapitre.

2.1 Principaux objectifs

Décrivons brièvement dans cette section les notions que l'on souhaite atteindre pour les protocoles d'échange de clefs.

2.1.1 Authentifications implicite ou explicite

Lorsque deux personnes échangent une clef, on souhaite au minimum que personne d'autre ne connaisse le secret commun : c'est l'authentification *implicite*. Elle ne garantit pas que des clefs ont effectivement été mises en commun, mais seulement qu'en cas d'accord, ces clefs sont aléatoire et inconnues par l'extérieur. Cette authentification sera garantie par les notions décrites dans ce chapitre de *Find-then-Guess* et *Real-or-Random*.

On peut souhaiter en outre que, lorsqu'un participant accepte une clef, il est sûr que son partenaire est également en mesure de calculer cette clef (il a effectivement reçu tout le matériel nécessaire) : c'est l'authentification *explicite*. Cette garantie peut être apportée aux deux joueurs (authentification mutuelle) ou uniquement à l'un des deux (authentification unilatérale). Elle leur assure non seulement que personne d'autre ne connaît ce secret commun, mais aussi qu'ils le partagent bien. L'authentification mutuelle a plus de sens dans le cadre des groupes, dans lesquels elle assure qu'il est impossible que deux joueurs aient accepté des clefs de session différentes. Notons cependant qu'un adversaire peut modifier les messages suivants

et finalement faire rejeter certains des joueurs pendant que d'autres acceptent. Mais cette authentification signifie surtout que si un joueur accepte une clef de session, alors les autres vont soit accepter la même, soit rejeter, mais pas en accepter une autre.

Pour cela, une phase de confirmation de clef est ajoutée à la fin du protocole. Une conversion générique de implicite vers explicite a été proposée, et prouvée dans le modèle de l'oracle aléatoire par Bellare, Pointcheval et Rogaway [BPR00].

2.1.2 Robustesse et *forward secrecy*

Un protocole est dit *robuste* si la perte d'une clef de session ne fragilise que cette session. En particulier, aucune information n'est révélée sur une autre clef, et aucune authentification ultérieure n'est rendue possible grâce à la possession de la clef perdue.

Il possède la propriété de *forward secrecy* si aucune clef à long terme (un mot de passe) révélée à l'adversaire ne lui donne d'information sur les clefs de session déjà générées. Ceci est important en raison de la faible entropie des mots de passe qui les rend vulnérables à une recherche exhaustive.

2.2 Historique des échanges de clefs à deux joueurs

Le premier protocole d'échange de clefs fut celui présenté par Diffie et Hellman [DH76], en 1976, que nous avons rappelé dans le chapitre précédent page 23. À cette époque, les protocoles cryptographiques étaient en général construits par essais/erreurs : après la proposition de schémas, ces derniers étaient considérés comme sûrs si aucun cryptanalyste ne parvenait à les casser. L'histoire a montré que cette méthode a ses limites et de nombreux protocoles présumés sûrs furent ainsi mis en défaut, souvent bien des années après leur spécification.

Au début des années 1980, Goldwasser et Micali [GM84], puis Blum-Micali [BM84] et Yao [Yao82b] ont suggéré que la sécurité des protocoles pouvait être prouvée sous des hypothèses calculatoires « standards » : c'est la naissance de la *sécurité prouvée*. Dès 1985, des preuves furent exhibées pour le chiffrement probabiliste, la génération de nombres pseudo-aléatoires et la signature numérique. La première idée que des mots de passe faibles pouvaient être utilisés en ligne (dans un protocole d'échange de clefs) a été faite par Bellovin et Merritt dans [BM92] qui proposèrent le protocole EKE, pour *encrypted key exchange*. Ils ont donné les premiers l'intuition que le succès d'un adversaire pour casser le protocole devait être proportionnel au nombre de fois où l'adversaire interagit avec le serveur, et relié de manière négligeable à ses capacités de calcul hors-ligne. En 1993, Bellare et Rogaway s'attaquèrent à ces problèmes d'authentification et de distribution de clefs que les outils connus pour les autres primitives (chiffrement, signature, preuve *zero-knowledge*, preuve de connaissance) ne permettaient pas de traiter : des notions telles que l'indistinguabilité et les preuves par simulation n'étant pas suffisantes, ils durent commencer par définir le cadre sous-jacent à la communication, désormais connu comme le cadre *Find-then-Guess* et présenté dans la section 2.4 page 33. Ce cadre est basé sur un jeu qui définit la sécurité.

Bellare, Pointcheval et Rogaway [BPR00] étendirent ensuite ce cadre au cas des protocoles avec authentification par mots de passe. Ils définissent dans cet article les idées de partenariat, de fraîcheur des clefs de session, et les utilisent pour mesurer la sécurité d'un protocole d'échange de clefs authentifié, d'authentification unilatérale, ainsi que d'authentification mutuelle. Ils apportent en particulier quelques ajouts à la version de Bellare et Rogaway, ainsi que certaines modifications formelles pour que le cadre puisse : spécifier des identifiants de session (sid) pour définir correctement les notions de partenariat, distinguer entre les sessions qui « terminent » et celles qui « acceptent », donner à l'adversaire la capacité bien distincte d'obtenir des exécutions du protocole (afin de mesurer la sécurité contre les attaques passives ou actives) et lui donner des capacités de corruption afin de mesurer la *forward secrecy*.

De manière contemporaine à l'article [BPR00], Boyko, MacKenzie et Patel [BMP00] présentèrent leur cadre de sécurité basé sur des preuves par simulation multi-joueurs, ainsi qu'une preuve dans ce cadre d'un protocole basé sur EKE.

Avant eux, Lucks [Luc97] avait déjà cherché à étendre les idées de [BR94] pour traiter les attaques par dictionnaire. Halevi et Krawczyk [HK98] avaient aussi donné des définitions et protocoles pour l'authentification unilatérale à base de mots de passe, dans le cas où le serveur possédait un couple (clef secrète, clef publique) donné par une autorité (PKI, *public-key infrastructure*), mais il s'agit d'un problème assez différent de celui considéré ici. Ce type d'authentification est dit *hybride* puisqu'il mélange PKI et mots de passe. Leur travail fut suivi (et critiqué) par celui de Boyarsky [Boy99] qui donna une notion de sécurité basée sur la simulation.

Le groupe de travail du standard IEEE P1363 [IEE01, IEE04] sur les méthodes d'échange de clefs authentifiés par des mots de passe montra son intérêt sur les protocoles dans lesquels les clients utilisaient des mots de passe courts au lieu de certificats pour s'identifier vis-à-vis des serveurs. Cet effort de standardisation s'appuyait sur [BPR00] et [BMP00] dans lesquels les cadres formels et les objectifs de sécurité pour ce type de protocoles étaient clairement formulés. Les auteurs de ces deux papiers analysèrent la sécurité de EKE [BM92], mais dans des modèles différents, les premiers dans les modèles de l'oracle aléatoire et du chiffrement idéal, et les seconds dans le modèle de l'oracle aléatoire uniquement.

Dans le même temps, EKE évolua pour devenir le protocole Auth_A [BR00], mais à nouveau sans preuve. Il fut modélisé formellement en OEKE [BCP03b] avec une preuve de sécurité dans le modèle de l'oracle aléatoire et du chiffrement idéal, puis en OMDHKE [BCP04] dans le modèle de l'oracle aléatoire uniquement. Dans les deux cas, un seul message est chiffré, ce qui est crucial pour enrichir TLS avec des échanges de clefs basés sur des mots de passe [BEWS00, ABC$^+$06], en raison du format imposé des échanges.

Goldreich et Lindell [GL01] proposèrent ensuite un protocole sûr dans le modèle standard, mais non-concurrent, et ce travail fut suivi du protocole KOY [KOY01, KOY02] de Katz, Ostrovsky et Yung, lui aussi dans le modèle standard, et généralisé par la suite par Gennaro et Lindell [GL03]. Les seules versions raisonnables reposent sur le DDH, mais demeurent ni efficaces, ni vraiment utilisables en pratique.

Parallèlement au développement de ces protocoles, Canetti [Can01] proposa un nouveau cadre de sécurité (la composabilité universelle, ou UC, dans l'esprit du cadre par simulation), qui fera l'objet du chapitre 5 page 58, pour gérer la composition arbitraire de protocoles. Avec Halevi, Katz, Lindell et MacKenzie, il proposa un protocole dans le modèle standard [CHK$^+$05], basé sur KOY/GL, et sûr dans le cadre UC contre les adversaires passifs. À l'heure actuelle, le seul protocole dans le modèle standard sûr dans le cadre UC contre les adversaires adaptatifs était celui de Barak, Canetti, Lindell, Pass et Rabin [BCL$^+$05]. Nous en proposons un nouveau [ACP09] dans le chapitre 9 page 137. Nous proposons aussi un protocole sûr contre les attaques adaptatives dans le chapitre 6 page 89, mais dans le modèle du chiffrement idéal et de l'oracle aléatoire [ACCP08].

2.3 Modèle de communication, objectifs et notations

2.3.1 Objectifs et notations

Dans un protocole d'échanges de clefs, le scénario est couramment le suivant : on est en présence de deux joueurs, Alice (le client) et Bob (le serveur), qui souhaitent établir une connexion sécurisée. Alice possède un mot de passe pw et Bob possède une clef reliée à pw. Il peut soit s'agir de pw lui-même (dans ce cas l'échange est dit « symétrique »), soit de l'image de pw par une fonction à sens unique (échange « asymétrique » souvent appelé *verifier-based*). L'avantage de ce dernier cas est que si le serveur est corrompu, le mot de passe du client n'est pas (immédiatement) en danger : une recherche exhaustive est bien sûr possible, en vérifiant l'image obtenue pour chaque essai, mais cette recherche peut être ralentie, et ne peut être massive, car l'attaquant doit lancer une recherche complète pour chaque utilisateur. Des conversions génériques ont été données dans [ACP05] puis par Gentry *et al.* [GMR06] (dans le cadre UC). L'objectif est qu'à l'issue de la conversation Alice et Bob partagent une clef de session sk connue d'eux seuls.

De manière formelle, on fixe un ensemble E d'utilisateurs. Cet ensemble est l'union disjointe d'un ensemble de *clients* et d'un ensemble de *serveurs*. Chaque joueur a une identité U, S, A, B, ... qui le caractérise. Pendant l'exécution d'un protocole, il peut y avoir plusieurs instances en parallèle d'un même joueur U : on appelle ces instances des oracles et on les note Π^i_{U}. On peut voir une instance comme un processus contrôlé par le joueur.

Chaque client A possède un mot de passe pw_{A} (c'est bien le même pour chacune des instances de ce client). Chaque serveur B possède une liste de mots de passe $\{\mathsf{pw}_{\mathrm{B}}[\mathrm{A}]\}_{\mathrm{A\ client}}$ contenant un mot de passe (*transformé*) par client. Dans le modèle dit *symétrique*, $\mathsf{pw}_{\mathrm{A}} = \mathsf{pw}_{\mathrm{B}}[\mathrm{A}]$. Dans le modèle *asymétrique*, $\mathsf{pw}_{\mathrm{B}}[\mathrm{A}]$ est choisi tel qu'il soit difficile de calculer pw_{A} à partir de A, B et $\mathsf{pw}_{\mathrm{B}}[\mathrm{A}]$. Les mots de passe sont aussi appelés les *clefs ou secrets à long-terme*.

Un protocole est alors un algorithme probabiliste où des messages sont échangés entre les deux joueurs. L'algorithme détermine comment les instances de joueurs réagissent à réception d'un message. C'est l'adversaire qui transmet tous les messages qui transitent entre les joueurs : il s'agit d'un algorithme probabiliste qui peut poser des « requêtes » particulières ; ces dernières seront décrites dans les différents cadres de sécurité. Pendant l'exécution d'un protocole, il peut y avoir une infinité d'instances de chaque joueur.

2.3.2 Modèle de communication

Pour refléter au maximum la réalité d'un réseau, on suppose que toute la communication est contrôlée par un adversaire, soit passif (« honnête mais curieux » : il se contente d'écouter toute la conversation), soit actif, c'est-à-dire qu'il peut non seulement lire les messages échangés, mais aussi modifier les messages entre les deux joueurs avant qu'ils arrivent à destination, retarder des messages, en créer de nouveaux, les renvoyer une seconde fois ou ne pas les transmettre (« déni de service ») : c'est par exemple le cas d'un routeur sur Internet. De manière formelle, on suppose que toute la communication passe par l'adversaire : dans le premier cas il agit comme un câble qui se contente de faire le relais ; dans le second, il contrôle la communication à sa guise. Autrement dit, les joueurs ne se parlent jamais directement mais uniquement à travers l'adversaire. Dans le cas de protocoles « concurrents », on l'autorise aussi à entamer de nouvelles conversations entre des « instances » de joueurs, ce qui modélise le fait que des joueurs peuvent être engagés dans plusieurs discussions à la fois en parallèle. Chaque joueur est ainsi modélisé par une infinité d'oracles que l'adversaire peut mettre en route. Ces oracles n'interagissent qu'à travers l'adversaire, et jamais directement.

2.3.3 Les différentes attaques possibles

Il est clair dans ces conditions qu'il est impossible d'empêcher l'adversaire d'essayer de deviner un mot de passe et de l'utiliser pour essayer de se faire passer pour un joueur. Si cette attaque échoue, il raye ce mot de passe de sa liste et en essaie un autre : ainsi, au bout de n essais (n étant la taille du dictionnaire des mots de passe, $n = 10^4$ pour un code pin), il aura essayé tous les mots de passe et il en aura déduit quel était le bon relativement facilement (n étant particulièrement petit pour que les mots de passe puissent être retenus par un être humain). Cette attaque, appelée « par dictionnaire en ligne », est inévitable : l'objectif de l'authentification par mots de passe est de restreindre l'adversaire à cette attaque uniquement. Il sera en outre facile de la bloquer en interdisant à un utilisateur de faire un trop grand nombre d'essais (comme dans les cartes bancaires, qui n'autorisent pas l'utilisateur à essayer plus de trois codes). En particulier, on ne veut pas qu'une écoute passive des conversations permette à l'adversaire de faire une recherche exhaustive hors ligne (« attaque par dictionnaire hors ligne ») car dans ce cas, la sécurité serait limitée par la puissance de calcul de l'attaquant et non par le nombre d'interactions entre lui et le système d'authentification.

L'adversaire dispose de plusieurs types d'attaque, dont : le *rejeu*, où il s'agit informellement de réenvoyer un message déjà utilisé dans une exécution de protocole, souvent pour en retirer de l'information ; l'attaque de type *man-in-the-middle*, déjà décrite page 23, dans laquelle l'adversaire s'intercale entre les deux joueurs honnêtes ; le *déni de service*, dans lequel

l'adversaire retient un message ; les *tentatives de prise de contrôle*, où l'adversaire tente de jouer à la place d'un utilisateur ; et les *corruptions*, où l'adversaire prend le contrôle du joueur en apprenant un certain nombre d'informations sur lui, dont au minimum les secrets à long terme. Nous donnons un peu plus de détail sur cette dernière possibilité de l'attaquant dans la section suivante.

2.3.4 Les corruptions

On modélise couramment la malhonnêteté par la présence d'un *adversaire* (ou *attaquant*), qui peut *corrompre* des joueurs, en récupérant toutes les données stockées dans leur mémoire, ce qui inclut ou non toutes les informations échangées jusque-là (la corruption est dite *forte* si c'est le cas, et la quantité précise d'information révélée est décrite dans le protocole lui-même). Cette définition standard de la corruption, a priori assez généreuse envers l'adversaire, est suffisamment forte pour englober tous les adversaires rencontrés en pratique dans les applications.

On peut distinguer deux types de corruption. La corruption *passive* revient à l'obtention de la totalité de l'information détenue par les joueurs corrompus, tout en les laissant exécuter le protocole correctement (l'adversaire n'intervient pas dans leur comportement). À l'inverse, la corruption *active* signifie que l'adversaire a tout le contrôle sur ces mêmes joueurs et leur comportement. Les adversaires peuvent en outre être *adaptatifs* et choisir les joueurs qu'ils souhaitent corrompre à tout moment, ou bien *non-adaptatifs* et les déterminer une fois pour toutes avant l'exécution du protocole.

Les joueurs honnêtes ne savent pas quels joueurs sont corrompus. En revanche, aucun protocole ne peut être sûr *quel que soit* le sous-ensemble de joueurs corrompus : il faut toujours imposer des conditions pour limiter leur nombre.

2.4 Le cadre *Find-then-Guess* ([BR94], [BPR00] et [BR95])

Ce cadre, basé sur un jeu de sécurité, a été présenté par Bellare et Rogaway en 1993, et modifié par Bellare, Pointcheval et Rogaway en 2000. Étant donné que toute la communication passe par l'adversaire, il reste à définir ce que signifie dans ce cadre d'être convaincu qu'un joueur a réellement engagé une conversation avec un tiers. L'idée est que l'objectif sera atteint si la seule manière qu'a l'adversaire de faire accepter des joueurs à la fin d'une exécution de protocole est de relayer tous les messages à la manière d'un câble. Ceci sera formalisé par la notion de « transcriptions identiques ».

Formellement, l'adversaire est une machine probabiliste possédant une infinité d'oracles de la forme Π^i_{U} avec $i \in \mathbb{N}$ (i-ième instance du joueur U). L'adversaire $\mathcal{A}$ communique avec les oracles via des requêtes qui sont décrites ci-dessous. En réponse à sa requête, l'adversaire apprend le message en sortie ainsi que si l'oracle a terminé, accepté (créé une clef de session) ou abandonné (aucune clef n'a été générée). Il n'apprend en revanche pas la sortie privée de l'oracle.

À l'origine, toutes les sessions sont dites *fraîches*. Elles cessent de l'être lorsque leurs partenaires ou elles-mêmes ont subi une requête `reveal` (voir la description des requêtes plus bas) de la part de l'attaquant. Une instance de client commence le protocole en envoyant le premier message ; une instance de serveur lui répond alors avec un deuxième message. Ce procédé continue pour un nombre fixé de messages (en général entre 2 et 5) jusqu'à ce que les deux instances *terminent*. À cet instant, les deux instances devraient avoir *accepté*, et posséder une *clef de session* sk, un *identifiant de session* sid et un identifiant de partenaire pid. La clef de session est l'objectif du protocole ; le sid est un identifiant unique pour nommer chaque session et définir les « partenaires » (dans [BPR00], il est défini – publiquement – à la fin de la session comme concaténation de tous les messages envoyés en excluant le dernier, tandis que dans [BR94], il s'agit d'une valeur définie dès le début) ; le pid est le nom du joueur avec lequel l'utilisateur pense avoir échangé une clef (attention, la notation pid aura un sens très différent dans le cadre UC). Notez que les deux dernières valeurs sont publiques, tandis que la première est privée. Ces instances peuvent accepter au maximum une seule fois.

Le modèle de communication place l'adversaire au centre de tout. Il possède un nombre infini d'oracles d'instances de joueurs et peut leur poser des requêtes de cinq différents types :

- `send`$(\mathrm{U}, i, \mathrm{M})$: cela modélise l'envoi, par l'attaquant, d'un message M à l'oracle Π^i_{U}. L'oracle calcule alors le résultat en suivant le protocole et retourne la réponse. Cette requête permet à l'attaquant de faire des attaques actives, en contrôlant tout : il peut initier une exécution, relayer des messages, en créer ou en dupliquer.

- `reveal`(U, i) : si l'oracle Π^i_{U} a accepté et possède une clef de session sk, alors on donne sk à l'adversaire. Cela modélise la *robustesse* du protocole, c'est-à-dire l'idée que la perte d'une clef de session ne fragilise que cette session. En particulier, aucune information n'est révélée sur une autre clef, et aucune authentification ultérieure n'est rendue possible grâce à la possession de la clef perdue.

- `corrupt`$(\mathrm{U}, \mathsf{pw})$: l'adversaire obtient pw_{U} et l'état de toutes les instances du joueur U. Cette attaque, très destructrice, modélise une attaque du type « cheval de Troie » permettant de prendre le contrôle d'une machine. Il s'agit de la corruption dite « forte ». En cas de corruption faible, l'adversaire apprend seulement le mot de passe du joueur, et pas son état interne.

  Cette requête permet de traiter correctement la *forward secrecy*. Cette dernière signifie qu'une clef à long terme (mot de passe) révélée à l'adversaire ne lui donne aucune information sur les clefs de session déjà générées. Le deuxième argument d'une requête dirigée contre un client permet de remplacer le mot de passe dérivé possédé par un serveur par la valeur imposée pw.

- `execute`$(\mathrm{A}, i, \mathrm{B}, j)$: cette requête permet de lancer une exécution honnête entre l'oracle de client Π^i_{A} et l'oracle de serveur Π^j_{B}. Cela permet de modéliser une attaque passive. Cette requête peut sembler inutile car elle peut être instanciée par des requêtes `send` (qui peut relayer les réponses des oracles), mais séparer les deux cas de figure permet de gérer finement les attaques par dictionnaire (car seules les attaques actives doivent être comptabilisées).

- `test`(U, i) : si l'oracle Π^i_{U} a accepté et possède la clef de session sk, alors on tire un bit b au hasard. Si $b = 0$, on donne la clef sk à l'adversaire. Sinon, on lui donne une clef aléatoire. Cette requête mesure simplement le succès de l'adversaire, elle ne lui donne aucun pouvoir supplémentaire. Cette requête servira à modéliser la capacité d'un attaquant à distinguer la véritable clef de session d'une clef aléatoire. Cette requête n'est accessible que si la session est fraîche. Une session est fraîche tant qu'aucune des instances impliquée n'a subi de requête `reveal`. Intuitivement, un joueur possède une clef de session fraîche si l'adversaire ne la connaît pas de façon triviale.

L'objectif d'un échange de clefs est que l'adversaire soit incapable d'obtenir de l'information sur une clef de session fraîche. La formalisation de ce but est une adaptation de l'indistinguabilité des chiffrements [GM84] : à la fin d'un protocole AKE sûr, l'adversaire doit être incapable de distinguer une clef de session fraîche d'un élément aléatoire.

Formellement, ce cadre est défini par le jeu de sécurité suivant : l'adversaire est autorisé à poser autant de requêtes `execute` et `send` qu'il le souhaite, puis une unique requête `test` à une session fraîche d'un joueur. La réponse à la requête consiste à choisir un bit b et à renvoyer la clef de session si $b = 0$ ou une chaîne aléatoire si $b = 1$. L'adversaire renvoie alors un bit b' et son but est d'avoir deviné b (avec b'). Son avantage est défini par la formule suivante et le protocole est sûr s'il est négligeable en le paramètre de sécurité :

$$\mathsf{Adv}^{\mathcal{A}}_{\mathsf{ake}} = 2\left|\Pr\left[b = b'\right] - \frac{1}{2}\right|$$

2.5 Le cadre *Real-or-Random* [AFP05]

Le cadre Real-or-Random (ROR) est basé sur la même idée que le cadre Find-then-Guess (FTG) : les clefs doivent « avoir l'air aléatoires ». La seule différence avec ce qui précède est que l'on autorise plusieurs requêtes `test` à l'attaquant : on ne cherche pas simplement à montrer qu'une clef d'une session choisie par l'attaquant est aléatoire, mais que toutes les clefs de session sont aléatoires et indépendantes les unes des autres. Ces deux cadres sont polynomialement équivalents [BDJR97], mais avec un facteur lié au nombre de sessions (il y a une perte égale au nombre de requêtes `test` entre le cadre FTG et le cadre ROR), ce qui remet en question la sécurité fasse aux attaques par dictionnaires. En effet, on souhaite que l'avantage de l'attaquant soit de l'ordre du nombre de sessions actives divisé par la taille du dictionnaire. Une perte en le nombre de requêtes `test` ne permet plus de vérifier cette condition. Le cadre ROR est donc finalement strictement plus fort que le cadre FTG, dans le cas de ces systèmes. Il est même possible de supprimer les requêtes `reveal`, en ne perdant qu'un facteur 2 dans la réduction.

2.6 Le cadre de Boyko, MacKenzie et Patel [BMP00]

L'idée du cadre (par simulation) décrit par Boyko, MacKenzie et Patel en 2000 est basée sur le cadre de Shoup [Sho99] (lui-même élaboré à partir de [BCK98]), et généralisée à l'aide de notions de sécurité d'authentification à base de mots de passe données dans [HK98]. Elle repose sur la tradition des preuves par simulation (ou *simulatability*) dans les protocoles à plusieurs joueurs. On retrouvera certaines de ces idées dans le cadre UC, décrit dans le chapitre 5. Il s'agit dans un premier temps de définir un système idéal qui modélise, à l'aide d'une unité centrale digne de confiance, le service à proposer (ici, des échanges de clefs authentifiés à l'aide de mots de passe) ; puis, ensuite, de prouver que le protocole dans le monde réel (dans lequel évoluent les joueurs et l'adversaire) est essentiellement équivalent à celui dans le système idéal. Intuitivement, une preuve de sécurité cherche dans ce cadre à montrer que tout ce qu'un adversaire peut faire dans le monde réel peut aussi être effectué dans le monde idéal. Cela implique ainsi que le protocole est sûr dans le monde réel.

Les joueurs et les instances de joueurs sont définis comme dans les cadres précédents. En revanche, au début d'un protocole, chaque joueur apprend son « rôle », c'est-à-dire s'il va être un client ou un serveur. L'adversaire est défini comme précédemment. Les auteurs considèrent aussi une autorité, qui conserve les clefs de session $\mathrm{K}_{i,j}$ partagées entre les instances de joueurs. C'est elle qui donne les mots de passe (suivant une distribution arbitraire) aux joueurs. Une différence dans ce cadre est que la clef d'une instance est définie dès que cette instance entame une session. L'autorité possède aussi l'*environnement*, c'est-à-dire une chaîne arbitraire R de taille fixée (et inconnue de l'adversaire), qui modélise les informations partagées entre les joueurs dans des protocoles de plus haut niveau.

Dans le monde idéal, l'adversaire peut réaliser un certain nombre d'opérations :

- **initialiser un utilisateur** : l'adversaire crée alors un nouvel utilisateur avec une identité de son choix, et l'autorité crée les mots de passe associés (partagés entre lui et les autres joueurs – on se place dans le cadre symétrique).
- **donner une valeur à un mot de passe** : l'adversaire peut choisir le mot de passe partagé par un utilisateur i et un nouvel utilisateur (dont l'identité n'appartient à aucun utilisateur existant). Cette identité ne peut alors plus être utilisée dans la précédente requête. Cette requête lui permet en particulier (via un compte factice) de partager un mot de passe avec un utilisateur.
- **initialiser une instance** : l'adversaire doit alors lui donner un rôle et un partenaire (qui doit avoir fait l'objet d'une requête du type de la précédente au préalable).
- **demander à une instance de terminer**.

- **tester le mot de passe d'une instance** : cette requête ne peut être posée qu'une seule fois par instance, qui doit avoir été initialisée et qui ne doit pas être engagée dans une session (*ie* soit elle n'a pas commencé, soit elle a terminé).
- **commencer une session** : l'adversaire demande à ce qu'une clef de session soit générée pour l'instance en question, soit en tant que client, soit en tant que serveur, soit à une instance qui n'a pas de partenaire ou dont le mot de passe a été correctement deviné. Dans ce dernier cas, la clef de session est retournée à l'adversaire.
- **obtenir des informations sur l'environnement** : l'adversaire peut obtenir toute l'information souhaitée sur l'environnement ou les clefs de session. Cela modélise une fuite d'information, par exemple à travers l'utilisation de la clef dans un chiffrement.

Dans le monde réel, une instance a accès à son identité, celle de son partenaire, et au mot de passe qu'ils ont en commun. Elle commence dans un certain état initial et change d'état uniquement à réception d'un message, tout en générant un message en réponse et en précisant son statut : *continuer* (l'instance est prête à recevoir un autre message), *accepter* (l'instance a terminé et généré une clef de session) ou *rejeter* (l'instance a terminé sans générer de clef de session). L'adversaire peut réaliser le même type d'opérations que dans le monde idéal (initialiser un utilisateur, une instance de joueur, donner une valeur à un mot de passe, obtenir des informations). Il peut aussi **livrer un message**, ou **poser une question à un oracle aléatoire**.

La définition de sécurité est alors la suivante :

- ***complétude*** : si l'adversaire dans le monde réel transmet correctement les messages entre les deux joueurs, alors les deux joueurs acceptent ;
- ***simulabilité*** : pour tout adversaire dans le monde réel, il existe un adversaire dans le monde idéal tel que les transcriptions dans les deux mondes soient indistinguables.

Le liens précis entre ce cadre et les deux précédents est encore un problème ouvert qu'on ne cherchera pas à résoudre ici. On remarque que les auteurs de [CHK+05] ont montré que le cadre UC était plus fort que les cadres usuels. Même si elle est inconnue, la relation entre les autres perd ainsi de son importance.

Chapitre 3

Principaux protocoles à deux joueurs

L'invention de nouveaux schémas d'authentification à base de mots de passe a suivi le développement des cadres de sécurité. Les protocoles pour deux joueurs possédant une preuve de sécurité peuvent être regroupés dans trois grandes familles : EKE [BM92], KOY/GL ([KOY01], [GL03]) et *multiparty computation* ([NV04], [BCL+05]). D'autres familles seront brièvement décrites, mais elles n'ont pas fait l'objet d'analyse rigoureuse de sécurité.

- Les protocoles du premier type s'inspirent du schéma de base EKE (pour *encrypted key exchange*, fondé sur un certain nombre d'échanges de messages chiffrés avec le mot de passe comme clef commune). Si les premiers d'entre eux ([BM92], [BR00]) reposent sur des heuristiques, les derniers (par exemple [BCP03b], [BCP04], [CPP04], [AP05]), plus élaborés, possèdent une preuve de sécurité dans le modèle de l'oracle aléatoire.
- La deuxième famille est dans le *modèle standard* (sans oracle aléatoire), éventuellement basée sur une CRS (*common reference string*). Ils s'agit du KOY [KOY01] et de sa généralisation KOY/GL [GL03]. L'efficacité est déjà un peu réduite.
- Enfin, la dernière famille est assez différente et utilise la *multiparty computation*. L'efficacité n'est plus du tout l'objectif de ces derniers protocoles.

3.1 Protocoles avec oracle aléatoire : la famille EKE [BM92]

3.1.1 Bellovin et Merritt [BM92]

Toutes familles confondues, le premier protocole proposé pour réaliser un échange de clefs avec authentification par mots de passe fut celui de Bellovin et Merritt [BM92], en 1992. Il repose sur une heuristique. Dans leur article, le schéma est d'abord décrit de manière formelle

et générique, et deux instanciations pratiques sont ensuite proposées. Elles sont respectivement basées sur les schémas de chiffrement RSA [RSA78] et ElGamal [ElG85] (voir page 21). Le protocole EKE générique est présenté figure 3.1, pour deux joueurs Alice et Bob partageant une clef pw à faible entropie.

Le protocole EKE inspira de nombreux cryptographes dans l'élaboration de nouveaux schémas d'échange de clefs. La première avancée la plus notable est l'œuvre de Bellare et Rogaway [BR00], qui proposèrent en mars 2000 un protocole simple appelé Auth_A au groupe de travail pour la standardisation IEEE P1363.

Un inconvénient d'EKE est que la clef et le chiffré doivent être dans le même espace, ce qui ne fonctionne qu'avec un chiffrement ElGamal. Pour contrer cette difficulté, Lucks proposa alors le protocole OKE [Luc97], dans lequel il ne chiffre que le deuxième message.

Fig. 3.1 – Le protocole EKE [BM92]

1. Alice génère une clef publique aléatoire $\mathrm{E_A}$, la chiffre à l'aide d'un chiffrement symétrique $\mathcal{E}$ et de la clef pw et envoie $(\mathrm{A}, \mathcal{E}_{\mathsf{pw}}(\mathrm{E_A}))$ à Bob, A étant son identité.
2. Bob déchiffre le message obtenu afin d'obtenir $\mathrm{E_A}$, choisit une clef secrète aléatoire R, la chiffre deux fois consécutivement à l'aide de $\mathrm{E_A}$ puis de pw et envoie $\mathcal{E}_{\mathsf{pw}}\left(\mathcal{E}_{\mathrm{E_A}}(\mathrm{R})\right)$ à Alice.
3. Alice déchiffre le message afin d'obtenir R, génère un challenge c_A, le chiffre avec R et envoie $\mathcal{E}_\mathrm{R}(c_\mathrm{A})$ à Bob.
4. Bob déchiffre le message afin d'obtenir c_A, génère un challenge c_B, chiffre les deux avec R et envoie le résultat $\mathcal{E}_\mathrm{R}(c_\mathrm{A}, c_\mathrm{B})$ à Alice.
5. Alice déchiffre le message afin d'obtenir c_A et c_B. Si le premier est conforme à ce qu'elle a envoyé, alors elle chiffre c_B avec R et envoie $\mathcal{E}_\mathrm{R}(c_\mathrm{B})$ à Bob.
6. Bob déchiffre le message afin d'obtenir c_B. S'il est conforme à ce qu'il a envoyé, alors le protocole a réussi, et la clef de session obtenue est R.

3.1.2 EKE_2 [BPR00]

Dans leur article, Bellare, Pointcheval et Rogaway ont reformulé cette construction avec le chiffrement ElGamal, en modélisant le chiffrement symétrique par un chiffrement idéal. Ils ont ensuite ajouté la phase de confirmation de clef, dans le modèle de l'oracle aléatoire. Il sera la base des améliorations présentées ci-dessous : on peut le voir comme un cas particulier de la famille Auth_A.

3.1.3 La famille de protocoles $\mathbf{Auth_A}$ [BR00]

Ces protocoles de Bellare et Rogaway se placent dans le modèle asymétrique. Les deux premiers messages du protocole consistent en un échange Diffie-Helman (dans un groupe dans lequel ce dernier est difficile) ; en outre, au moins l'un des deux messages est chiffré. Ensuite, un message d'authentification est envoyé du client vers le serveur. Il consiste simplement en un haché de valeurs facilement calculables par les deux joueurs. Le serveur vérifie alors cet authentifiant avant d'accepter la clef de session.

Les auteurs donnent des heuristiques, mais aucune preuve de sécurité : on pense ainsi que le protocole garantit la sécurité contre les adversaires actifs, les attaques par dictionnaire, ainsi que la *forward secrecy* et l'authentification du client (c'est-à-dire que le serveur est sûr d'avoir parlé au client) mais aucune preuve n'est donnée pour chacun de ces points. L'authentification du serveur peut être ajoutée au schéma à l'aide d'un dernier message, du serveur vers le client.

Les calculs s'effectuent dans un groupe cyclique $\mathbb{G} = \langle g \rangle$ noté multiplicativement et de cardinal q. Le groupe $\mathbb{G}$ doit être choisi tel que le problème Diffie-Hellman y soit difficile, par

exemple $\mathbb{G} = \mathbb{Z}_q^*$ où q est un grand nombre premier. Le protocole Auth_A a un mécanisme de dérivation du mot de passe, interdisant à un adversaire ayant corrompu la table de mots de passe d'un serveur de pouvoir se faire passer pour un client automatiquement. Pour cela, il tire profit de l'aspect asymétrique : à partir d'un mot de passe pw, le client possède un mot de passe $\mathsf{pw}_\mathrm{A} = \mathcal{H}'(\mathrm{A}\|\mathrm{B}\|\mathsf{pw})$ (de même entropie que pw, mais cette fois dans $\mathbb{Z}_q^*$) et le serveur possède un mot de passe pw_B dérivé par la formule $\mathsf{pw}_\mathrm{B} = g^{\mathsf{pw}_\mathrm{A}}$, qui a lui aussi la même entropie. Le protocole utilise aussi une seconde fonction de hachage $\mathcal{H}$ ainsi que deux schémas de chiffrement $\mathcal{E}^1$ et $\mathcal{E}^2$, l'un des deux pouvant être la fonction identité.

Les messages ne sont pas envoyés dans un ordre précis : autrement dit, le client comme le serveur peut commencer en premier. Par simplicité, le protocole de la figure 3.2 décrit le cas où c'est le client qui commence.

Fig. 3.2 – Le protocole Auth_A [BR00]

1. Le client Alice choisit une valeur aléatoire $x \in \{1, \ldots, q\}$, calcule $\mathrm{X} = g^x$, le chiffre en $\mathrm{X}^* = \mathcal{E}^1_{\mathsf{pw}_\mathrm{B}}(\mathrm{X})$ et envoie cette dernière valeur à Bob.
2. Le serveur Bob choisit une valeur aléatoire $y \in \{1, \ldots, q\}$, calcule $\mathrm{Y} = g^y$, le chiffre en $\mathrm{Y}^* = \mathcal{E}^2_{\mathsf{pw}_\mathrm{B}}(\mathrm{Y})$ et envoie Y^* à Alice.
3. Alice reçoit Y^*, calcule $\mathrm{Y} = \mathcal{D}^2_{\mathsf{pw}_\mathrm{B}}(\mathrm{Y}^*)$ ainsi que $\mathsf{DiffieHellmanKey}_\mathrm{A} = \mathrm{Y}^x$.
4. Bob reçoit X^*, calcule $\mathrm{X} = \mathcal{D}^1_{\mathsf{pw}_\mathrm{B}}(\mathrm{X}^*)$ ainsi que $\mathsf{DiffieHellmanKey}_\mathrm{B} = \mathrm{X}^y$.
5. Alice calcule les valeurs suivantes, envoie Auth_A à Bob et accepte directement la clef de session $\mathsf{SessionKey}_\mathrm{A}$:

$$\begin{aligned} \mathsf{MasterKey}_\mathrm{A} &= \mathcal{H}(\mathrm{A}\|\mathrm{B}\|\mathrm{X}\|\mathrm{Y}\|\mathsf{DiffieHellmanKey}_\mathrm{A}) \\ \mathsf{SessionKey}_\mathrm{A} &= \mathcal{H}(\mathsf{MasterKey}_\mathrm{A}\|0) \\ \mathsf{Auth}_\mathrm{A} &= \mathcal{H}(\mathsf{MasterKey}_\mathrm{A}\|\mathrm{Y}^{\mathsf{pw}_\mathrm{A}}) \end{aligned}$$

   De même, Bob calcule

$$\begin{aligned} \mathsf{MasterKey}_\mathrm{B} &= \mathcal{H}(\mathrm{A}\|\mathrm{B}\|\mathrm{X}\|\mathrm{Y}\|\mathsf{DiffieHellmanKey}_\mathrm{B}) \\ \mathsf{SessionKey}_\mathrm{B} &= \mathcal{H}(\mathsf{MasterKey}_\mathrm{B}\|0) \\ \mathsf{AuthCheck}_\mathrm{A} &= \mathcal{H}(\mathsf{MasterKey}_\mathrm{B}\|{\mathsf{pw}_\mathrm{B}}^y) \end{aligned}$$

   et il accepte la clef de session $\mathsf{SessionKey}_\mathrm{B}$ si et seulement si on a l'égalité $\mathsf{Auth}_\mathrm{A} = \mathsf{AuthCheck}_\mathrm{A}$.

La version proposant l'authentification mutuelle diffère très peu de ce schéma. Dans cette dernière, Alice n'accepte pas tout de suite. Bob accepte si et seulement si $\mathsf{Auth}_\mathrm{A} = \mathsf{AuthCheck}_\mathrm{A}$. Si cette égalité est vérifiée, il calcule ensuite $\mathsf{Auth}_\mathrm{B} = \mathcal{H}(\mathsf{MasterKey}_\mathrm{B}\|2)$ et l'envoie à Alice. Elle calcule alors la valeur $\mathsf{AuthCheck}_\mathrm{B} = \mathcal{H}(\mathsf{MasterKey}_\mathrm{A}\|2)$ et elle accepte enfin si et seulement si $\mathsf{Auth}_\mathrm{B} = \mathsf{AuthCheck}_\mathrm{B}$.

Les auteurs ne donnent pas de preuve de sécurité, mais ils suggèrent plusieurs heuristiques. Ils formulent ainsi l'hypothèse que le protocole est sûr si l'on instancie les schémas de chiffrement dans le modèle du chiffrement idéal, ou bien si l'on utilise un oracle aléatoire, par exemple si $\mathcal{E}_{\mathsf{pw}_\mathrm{B}}(x) = x \cdot \mathcal{H}_0(\mathsf{pw}_\mathrm{B})$. Ce sont ces idées qui seront réutilisées dans les protocoles qui vont suivre : les auteurs de [BCP03b] prouvent la sécurité dans le premier cas, puis dans le second cas dans l'article [BCP04].

Les auteurs de [BMP00] proposèrent eux aussi une instanciation de Auth_A dans le modèle de l'oracle aléatoire, nommée PAK, mais ils en donnent une preuve de sécurité dans le cadre de la simulation (et non dans le FTG, voir la section 2.6 page 35) et la preuve est assez complexe.

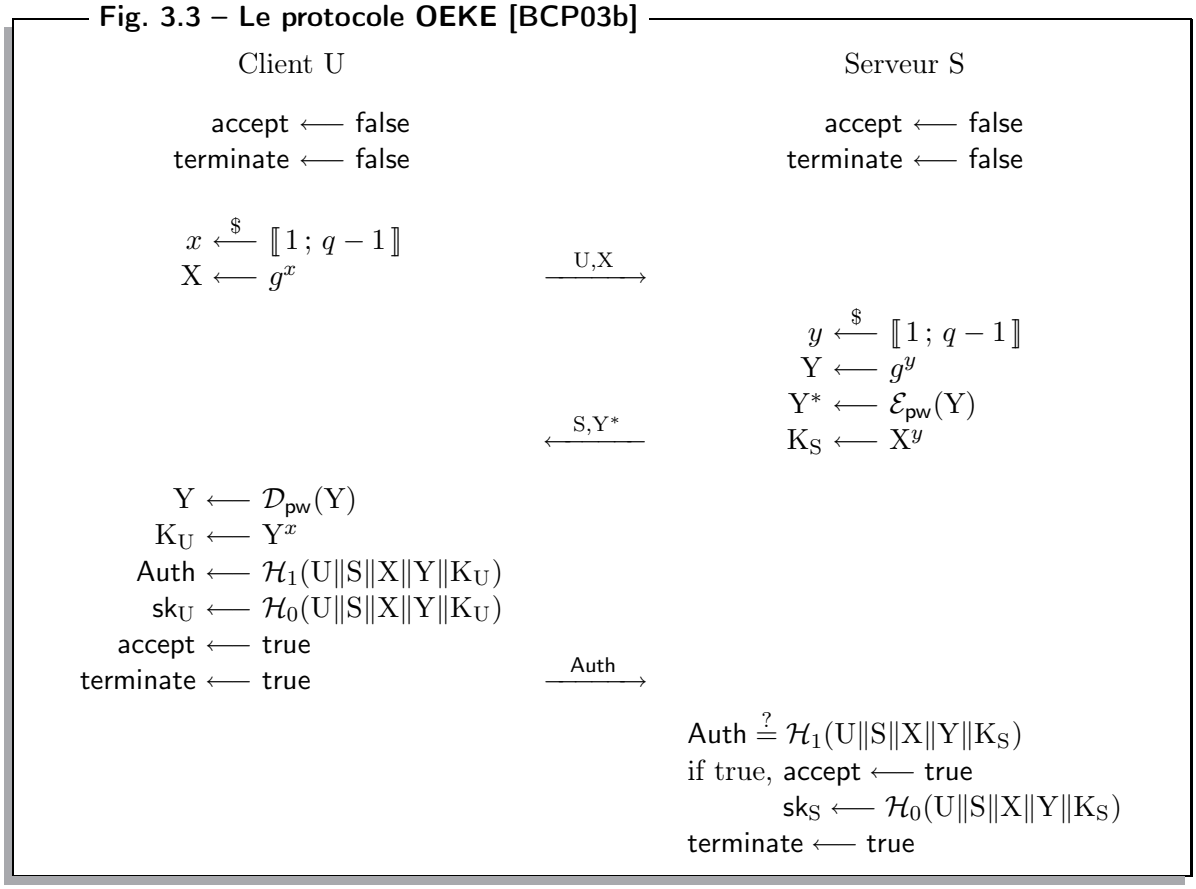
3.1.4 Instanciation du protocole Auth_A dans le modèle du chiffrement idéal et de l'oracle aléatoire [BCP03b]

Le protocole Auth_A est étudié en détail par Bresson, Chevassut et Pointcheval dans l'article [BCP03b]. Leur analyse montre que ce protocole ainsi que ses multiples modes opératoires possèdent une preuve de sécurité dans les modèles de l'oracle aléaoire et du chiffrement idéal, sous l'hypothèse CDH.

Dans leur article, Auth_A est formellement modélisé par le protocole OEKE (*One-Encryption Key-Exchange*) dans lequel un seul message est chiffré, soit par une primitive de chiffrement symétrique, soit par une fonction multiplicative telle que le produit d'une valeur Diffie-Hellman avec le haché du mot de passe. L'avantage d'un tel schéma sur EKE repose sur sa facilité d'intégration qui évite des problèmes de compatibilité : le format des échanges est plus conforme a ce qui est décrit dans la norme SSL. Ceci a un grand intérêt pratique pour TLS (*Transport Layer Security*) par exemple.

Une description du protocole OEKE entre un client U et un serveur S partageant un mot de passe pw peut être trouvé sur la figure 3.3. Les calculs sont effectués dans un groupe cyclique $\mathbb{G} = \langle g \rangle$ de cardinal q dans lequel le CDH est difficile. Le protocole utilise aussi deux oracles aléatoires $\mathcal{H}_0$ et $\mathcal{H}_1$.

Fig. 3.3 – Le protocole OEKE [BCP03b]



La différence avec le protocole EKE est qu'un seul message est chiffré ; il est alors clair qu'au moins un authentifiant doit être envoyé, et les auteurs de l'article prouvent que c'est suffisant. En outre, ce protocole procure l'authentification du client vers le serveur.

Enfin, la différence avec Auth_A est qu'ici, les deux joueurs possèdent le même mot de passe (version symétrique), mais il est aisé de revenir à l'ancien modèle (c'est-à-dire à l'utilisation de deux mots de passe $\mathsf{pw_U}$ et $\mathsf{pw_S}$ au lieu d'un seul mot de passe pw), simplement en posant $\mathsf{pw_S} = \mathsf{pw}$ et

$$\mathcal{H}_0(\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{Z}) \longleftarrow \mathcal{H}(\mathcal{H}(\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{Z})\|0)$$
$$\mathcal{H}_1(\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{Z}) \longleftarrow \mathcal{H}(\mathcal{H}(\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{Z})\|\mathrm{Y}^{\mathsf{pw_U}})$$

Cette variante possède exactement la même sécurité que la précédente. En ce qui concerne les autres modes (chiffrement des deux messages de l'échange Diffie-Hellman, authentification mutuelle, premier message envoyé par le serveur), les auteurs pensent qu'ils n'altèrent en rien la sécurité, mais des analyses de sécurité doivent être menées avant de pouvoir l'affirmer.

En ce qui concerne l'instanciation, un chiffrement par blocs ne peut pas remplacer le chiffrement idéal, ou bien des attaques par partition [BMN01] existent : pour chaque clef secrète, le chiffrement doit être une permutation. Une autre idée est d'instancier la primitive de chiffrement comme le produit d'une valeur Diffie-Hellman par le haché du mot de passe, comme suggéré dans [BR00]. Cela mène à un protocole d'échange de clefs à base de mots de passe sûr dans le modèle de l'oracle aléatoire uniquement, et c'est justement l'objet de l'article suivant, par les mêmes auteurs.

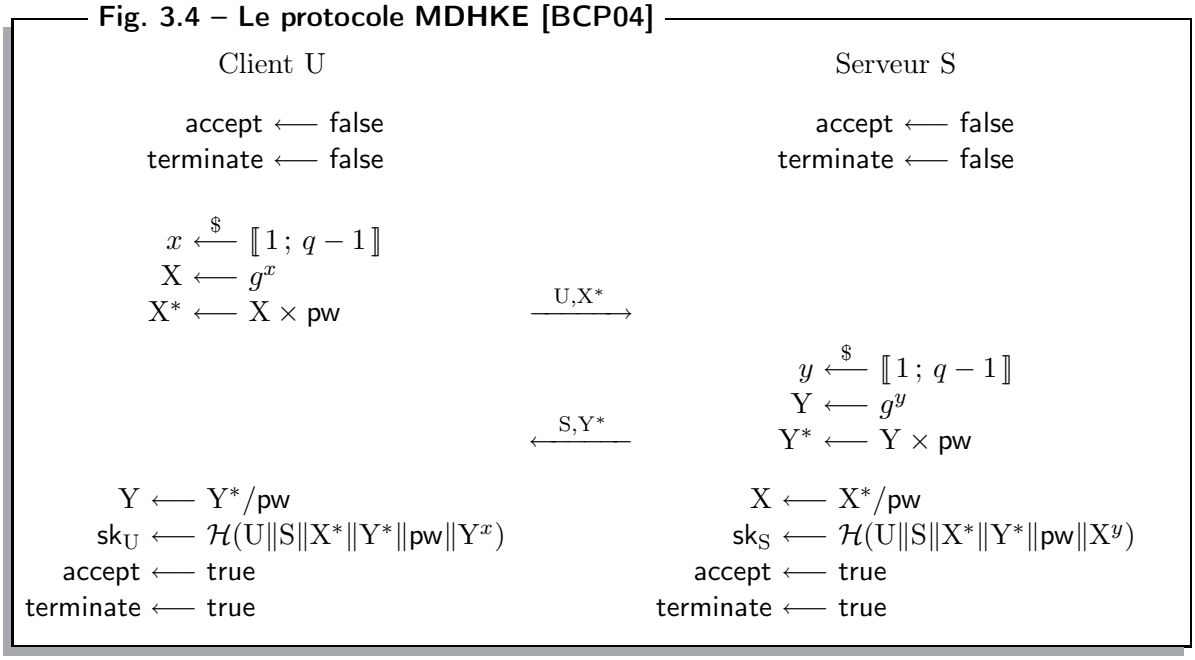
3.1.5 Instanciation du protocole Auth_A dans le modèle de l'oracle aléatoire uniquement [BCP04]

Ce deuxième article dans la lignée du protocole Auth_A analyse la sécurité de la deuxième idée de Bellare et Rogaway [BR00], à savoir l'instanciation du schéma de chiffrement par un oracle aléatoire. Les auteurs parviennent donc à se passer de l'hypothèse du chiffrement idéal. Ils proposent en outre un mécanisme supplémentaire pour lutter contre les attaques du type « déni de service ». L'hypothèse calculatoire sous-jacente est à nouveau le CDH et la preuve est exécutée dans le cadre FTG.

Le mécanisme pour éviter les attaques de type « déni de service » consiste à envoyer du serveur vers le client une sorte d'« énigme » à résoudre, qui va demander au client d'effectuer un certain nombre d'opérations cryptographiques, tandis que le serveur peut facilement et efficacement vérifier que la solution est correcte. Ainsi, les calculs ne sont effectués qu'après que le client a prouvé sa capacité à résoudre une énigme donnée, ce qui décharge le serveur en cas de tentative d'attaque.

Les auteurs proposent deux variantes de leur protocole. La première utilise un masque unique, c'est-à-dire qu'un seul message est chiffré : il s'agit de OMDHKE (pour *one-mask Diffie-Hellman key exchange*). La seconde, MDHKE (pour *masked Diffie-Hellman key exchange*) chiffre les deux messages. La preuve donnée ne montre pas la *forward secrecy* mais les auteurs affirment qu'il serait facile de l'ajouter, comme dans l'article précédent, en perdant simplement un facteur quadratique dans la réduction.

Fig. 3.4 – Le protocole MDHKE [BCP04]



Les calculs sont effectués dans un groupe $\mathbb{G} = \langle g \rangle$, d'ordre q, noté multiplicativement. $\mathcal{H}_0$ et $\mathcal{H}_1$ sont deux fonctions de hachage, de $\{0,1\}^*$ vers $\{0,1\}^{\ell_0}$ ou $\{0,1\}^{\ell_1}$, respectivement (elles sont modélisées par deux oracles aléatoires dans la preuve de sécurité). Enfin, $\mathcal{G}$ est une fonction de hachage de $\{0,1\}^*$ dans $\mathbb{G}$, modélisée elle aussi par un oracle aléatoire (qui renvoie, par définition, des réponses aléatoires dans $\mathbb{G}$). Le mot de passe pw commun au client U et au serveur S est choisi dans un dictionnaire $\mathcal{D}$. Le protocole est décrit sur la figure 3.5. Le protocole MDHKE repose quant à lui sur le problème *square Diffie Hellman* SDH, dans lequel $x = y$ (équivalent au problème CDH). Il est décrit sur la figure 3.4.

Fig. 3.5 – Le protocole OMDHKE [BCP04]

Client U		Serveur S
$\mathsf{accept} \longleftarrow \mathsf{false}$		$\mathsf{accept} \longleftarrow \mathsf{false}$
$\mathsf{terminate} \longleftarrow \mathsf{false}$		$\mathsf{terminate} \longleftarrow \mathsf{false}$
$x \xleftarrow{\$} [\![1 ; q-1]\!]$		
$\mathrm{X} \longleftarrow g^x$		
$\mathrm{X}^* \longleftarrow \mathrm{X} \times \mathsf{pw}$	$\xrightarrow{\mathrm{U},\mathrm{X}^*}$	
		$\mathrm{X} \longleftarrow \mathrm{X}^*/\mathsf{pw}$
		$y \xleftarrow{\$} [\![1 ; q-1]\!]$
		$\mathrm{Y} \longleftarrow g^y$
		$\mathrm{K_S} \longleftarrow \mathrm{X}^y$
		$\mathsf{Auth_S} \longleftarrow \mathcal{H}_1(\mathrm{U}\|\mathrm{S}\|\mathrm{X}^*\|\mathrm{Y}\|\mathsf{pw}\|\mathrm{K_S})$
		$\mathsf{accept} \longleftarrow \mathsf{true}$
		$\mathsf{sk_S} \longleftarrow \mathcal{H}_0(\mathrm{U}\|\mathrm{S}\|\mathrm{X}^*\|\mathrm{Y}\|\mathsf{pw}\|\mathrm{K_S})$
	$\xleftarrow{\mathrm{S},\mathrm{Y},\mathsf{Auth_S}}$	$\mathsf{terminate} \longleftarrow \mathsf{true}$
$\mathrm{K_U} \longleftarrow \mathrm{Y}^x$		
$\mathsf{Auth_S} \stackrel{?}{=} \mathcal{H}_1(\mathrm{U}\|\mathrm{S}\|\mathrm{X}^*\|\mathrm{Y}\|\mathsf{pw}\|\mathrm{K_U})$		
si vrai, $\mathsf{accept} \longleftarrow \mathsf{true}$		
$\mathsf{sk_U} \longleftarrow \mathcal{H}_0(\mathrm{U}\|\mathrm{S}\|\mathrm{X}^*\|\mathrm{Y}\|\mathsf{pw}\|\mathrm{K_U})$		
$\mathsf{terminate} \longleftarrow \mathsf{true}$		

3.1.6 Le protocole IPAKE [CPP04]

Les protocoles $\mathsf{Auth_A}$, OEKE, et OMDHKE ne sont que des variantes de OKE, proposé initialement par Lucks [Luc97]. Son objectif était de pouvoir faire fonctionner EKE avec un chiffrement RSA, et donc éviter le besoin que la clef publique et le chiffré asymétrique soient dans le même espace. Sa solution a donc été de ne pas chiffrer le premier envoi. Catalano, Pointcheval et Pornin ont formalisé les propriétés nécessaires à la fonction de chiffrement pour permettre une utilisation sûre dans OKE. IPAKE est cette généralisation, qui peut être instanciée avec ElGamal, pour conduire à OEKE, mais aussi avec RSA ou le chiffrement de Rabin [Rab79], proposant ainsi le premier protocole de mise en accord de clef avec authentification par mots de passe ayant une sécurité qui repose sur le problème de la factorisation. De plus, l'efficacité est une des meilleures, car les calculs se limitent à k multiplications modulaires, où k est le paramètre de sécurité (typiquement 80).

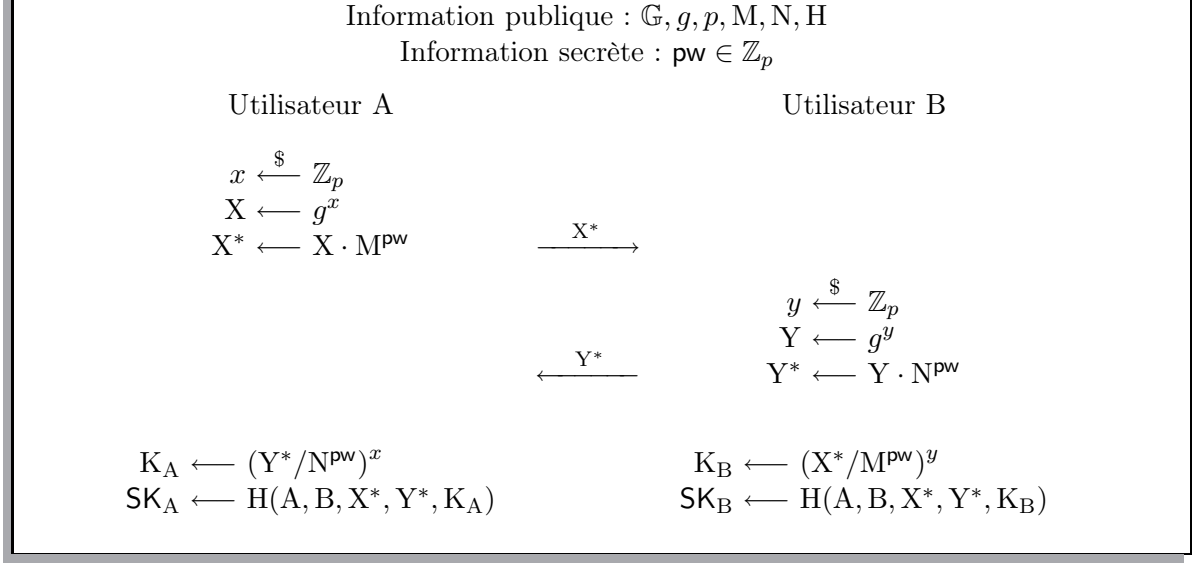
3.1.7 Le protocole SPAKE [AP05]

Abdalla et Pointcheval ont proposé dans [AP05] un protocole basé sur EKE, aussi efficace que ses prédécesseurs, mais qui ne requiert qu'une seule instance d'oracle aléatoire, pour la dérivation de la clef, et non plus pour instancier le chiffrement par le mot de passe. Sa preuve de

sécurité repose sur le CDH. En effet, l'objectif des protocoles qui ont suivi EKE a essentiellement été de dépendre le moins possible de l'oracle aléatoire, en parvenant idéalement à s'en passer complètement, comme dans KOY [KOY01] (voir la section suivante). Cependant, les protocoles sans oracle aléatoire ont tendance à être plus coûteux que les variantes de EKE. Dans cet article, Abdalla et Pointcheval ont essayé de garder le meilleur des deux mondes. Le problème principal des schémas sans oracle aléatoire est que l'adversaire peut généralement créer deux sessions différentes dont les clefs de session sont reliées. Il s'agit d'une attaque du type *man-in-the-middle*, dans laquelle l'adversaire modifie les messages qui sont envoyés d'un joueur à l'autre. Ainsi, si l'adversaire apprend ces deux clefs, il peut facilement en déduire le mot de passe à l'aide d'une attaque par dictionnaire hors-ligne. Ce sont des attaques dite « à clefs reliées » (*related-key attacks*).

Les attaques à clefs reliées ne peuvent pas facilement être contournées[1] à l'aide de MAC (*message authentication code*), lors de la phase de confirmation de clef, parce que cette notion forte de sécurité ne peut être atteinte que par un MAC construit à partir d'un oracle aléatoire. Dans cet article, les auteurs résolvent ce problème en utilisant une seule instance d'oracle aléatoire, au moment de la dérivation de la clef. Ils présentent en fait deux variantes du même protocole, qui diffèrent uniquement par la présence ou non du mot de passe dans la fonction de dérivation de clef. L'absence du mot de passe trouve son intérêt par exemple dans le cas distribué, dans lequel chaque serveur ne dispose que d'une partie du mot de passe. Le premier schéma n'est prouvé que dans le cas non-concurrent, tandis que pour le second (avec mot de passe dans la dérivation), une réduction très fine peut être obtenue.

Fig. 3.6 – Le protocole SPAKE1 [AP05]



Variantes du problème Diffie-Hellman. Soit $\mathbb{G}$ un groupe cyclique, d'ordre p premier, généré par un élément g. Le problème CDH *à bases choisies* (CCDH) est une variante du problème CDH. On considère un adversaire à qui l'on donne les trois éléments aléatoires M, N et X dans $\mathbb{G}$ et dont l'objectif est de trouver un triplet de valeurs (Y, u, v) telles que $u = \mathsf{CDH}(\mathrm{X}, \mathrm{Y})$ et $v = \mathsf{CDH}(\mathrm{X/M}, \mathrm{Y/N})$. L'idée est que l'adversaire est capable de calculer correctement soit u (avec $\mathrm{Y} = g$ et $u = \mathrm{X}$), soit v (avec $\mathrm{Y} = g\,\mathrm{N}$ et $v = \mathrm{X/M}$), mais pas les deux. Pour la définition formelle de ce problème, voir [AP05, page 6].

Le problème CDH à bases choisies basé sur un mot de passe (PCCDH) est une variante de ce problème. Dans ce cas, on considère un dictionnaire $\mathcal{D}$ contenant n valeurs équiprobables de mots de passe et $\mathcal{P}$ une application injective de $\mathcal{D}$ dans $\mathbb{Z}_p$. À la première étape, l'adversaire

[1] Sans contrainte, c'est-à-dire lorsque l'on ne sait pas bien restreindre les relations que l'attaquant va pouvoir mettre en place. Si on limite l'ensemble des relations, les contourner devient possible.

possède trois éléments aléatoires M, N et X dans $\mathbb{G}$ ainsi que l'application injective publique $\mathcal{P}$ et il retourne une valeur $\mathrm{Y} \in \mathbb{G}$. Ensuite, on choisit un mot de passe aléatoire $k \in \mathcal{D}$ et on l'envoie à l'adversaire. En posant $r = \mathcal{P}(k)$, l'objectif de l'adversaire dans cette seconde étape est de trouver une valeur K telle que $\mathrm{K} = \mathsf{CDH}(\mathrm{X}/\mathrm{M}^r, \mathrm{Y}/\mathrm{N}^r)$.

On peut remarquer qu'un adversaire qui devinerait k dans la première étape peut calculer $r = \mathcal{P}(k)$ et trouver facilement K en posant $\mathrm{Y} = g\mathrm{N}^r$ et $\mathrm{K} = \mathrm{X}/\mathrm{M}^r$. Comme k est choisi aléatoirement dans le dictionnaire, l'adversaire a alors une probabilité de succès égale à $1/n$. L'idée derrière cette hypothèse calculatoire est donc de garantir qu'un adversaire ne puisse pas faire mieux que celui ainsi décrit. Plus précisément, les auteurs montrent que ce résultat est vrai dès que le problème CDH est difficile dans le groupe $\mathbb{G}$, autrement dit, les problèmes PCCDH et CDH sont équivalents.

Le problème ensembliste équivalent ($\mathsf{S}-\mathsf{PCCDH}$) est une variation multi-dimensionnelle du problème précédent dans laquelle l'adversaire est autorisé à renvoyer, non pas une seule clef, mais plusieurs à la fin de la première étape. Dans ce cas, l'adversaire a gagné si la liste de ces clefs contient la bonne. Ce problème peut lui aussi être réduit au CDH.

Les protocoles. Après l'exposé de ces problèmes, les auteurs décrivent leur premier protocole, SPAKE1, et ils prouvent sa sécurité dans le cadre non-concurrent sous l'hypothèse $\mathsf{S}-\mathsf{PCCDH}$. Il s'agit d'une variante de EKE dans laquelle ils remplacent la fonction de chiffrement $\mathcal{E}$ par la fonction $\mathrm{X} \mapsto \mathrm{X}\mathrm{M}^{\mathsf{pw}}$, où M est un élément de $\mathbb{G}$ associé à A et en supposant que $\mathsf{pw} \in \mathbb{Z}_p$. L'identifiant de session est défini comme la transcription de la conversation entre A et B, et la clef de session est le haché (par un oracle aléatoire) de l'identifiant de session, des identités des utilisateurs, et des clefs Diffie-Helman (et pas du mot de passe). Le protocole SPAKE1 est décrit sur la figure 3.6. La preuve de ce protocole est une preuve par réduction : si l'on connaît un adversaire A contre le schéma, alors on peut construire des adversaires contre les primitives sous-jacentes de telle sorte que l'un au moins de ces adversaires casse la sécurité de l'une de ces primitives. Il s'agit d'une preuve par jeux.

Les auteurs décrivent ensuite leur second protocole, SPAKE2, dont ils prouvent la sécurité dans le cadre concurrent et sous l'hypothèse CDH. La seule différence avec SPAKE1 est que la fonction de dérivation de clef inclut cette fois-ci le mot de passe pw. Par souci de clarté, le protocole SPAKE2 est décrit en entier sur la figure 3.7.

Fig. 3.7 – Le protocole SPAKE2 [AP05]

Information publique : $\mathbb{G}, g, p, \mathrm{M}, \mathrm{N}, \mathrm{H}$
Information secrète : $\mathsf{pw} \in \mathbb{Z}_p$

Utilisateur A		Utilisateur B
$x \xleftarrow{\$} \mathbb{Z}_p$		
$\mathrm{X} \longleftarrow g^x$		
$\mathrm{X}^* \longleftarrow \mathrm{X} \cdot \mathrm{M}^{\mathsf{pw}}$	$\xrightarrow{\mathrm{X}^*}$	
		$y \xleftarrow{\$} \mathbb{Z}_p$
		$\mathrm{Y} \longleftarrow g^y$
	$\xleftarrow{\mathrm{Y}^*}$	$\mathrm{Y}^* \longleftarrow \mathrm{Y} \cdot \mathrm{N}^{\mathsf{pw}}$
$\mathrm{K}_\mathrm{A} \longleftarrow (\mathrm{Y}^*/\mathrm{N}^{\mathsf{pw}})^x$		$\mathrm{K}_\mathrm{B} \longleftarrow (\mathrm{X}^*/\mathrm{M}^{\mathsf{pw}})^y$
$\mathsf{SK}_\mathrm{A} \longleftarrow \mathrm{H}(\mathrm{A}, \mathrm{B}, \mathrm{X}^*, \mathrm{Y}^*, \mathsf{pw}, \mathrm{K}_\mathrm{A})$		$\mathsf{SK}_\mathrm{B} \longleftarrow \mathrm{H}(\mathrm{A}, \mathrm{B}, \mathrm{X}^*, \mathrm{Y}^*, \mathsf{pw}, \mathrm{K}_\mathrm{B})$

3.2 Protocoles dans le modèle standard : [KOY01, GL03]

Gennaro et Lindell ont proposé en 2003 [GL03] un cadre général pour l'authentification à base de mots de passe dans le modèle de la CRS. Leur protocole est basé sur le protocole KOY [KOY01], présenté par Katz, Ostrovsky et Yung en 2001, et peut même être vu comme une abstraction de ce dernier, ce qui présente trois avantages : il est modulaire et peut être décrit à l'aide de trois primitives cryptographiques de haut-niveau ; la preuve de sécurité est beaucoup plus simple et elle est aussi modulaire ; cela leur permet d'obtenir des analogues du KOY sous d'autres hypothèses cryptographiques. Le protocole, dans le modèle de la CRS, tire profit de la notion de *smooth projective hash function* de Cramer et Shoup [CS02], décrite dans la section 1.2.9 page 25. Si c est un chiffré de m sous la clef publique pk et avec l'aléa r, alors la clef de hachage est générée par $\mathsf{HashKG}(\mathsf{pk}) = \mathsf{hk}$, celle de projection par $\mathsf{ProjKG}(\mathsf{hk}, c) = \mathsf{hp}$. Les deux modes de calcul $\mathsf{Hash}(\mathsf{hk}, m, c)$ et $\mathsf{ProjHash}(\mathsf{hp}, m, c, r)$ donnent alors le même résultat (dans un groupe).

L'idée du protocole, pour un client $(\mathrm{P_I}, \mathsf{ssid})$ avec l'indice I et le mot de passe pw_I, et un serveur $(\mathrm{P_J}, \mathsf{ssid})$ avec l'indice J et le mot de passe pw_J, est présentée sur la figure 3.8.

Fig. 3.8 – Le protocole KOY/GL

Client U		Serveur S
(U1) $(\mathsf{VK_I}, \mathsf{SK_I}) \leftarrow \mathsf{SKG}$		
$c_\mathrm{I} = \mathcal{E}_{\mathsf{pk}}(\mathsf{pw_I}, \mathsf{VK_I}; \mathrm{R_I})$	$\xrightarrow{(\mathrm{U}, c_\mathrm{I}, \mathsf{VK_I})}$	
		(S2) $\mathsf{hk_J} = \mathsf{HashKG}(\mathsf{pk})$
		$\mathsf{hp_J} = \mathsf{ProjKG}(\mathsf{hk_J}; c_\mathrm{I})$
	$\xleftarrow{(\mathrm{S}, c_\mathrm{J}, \mathsf{hp_J})}$	$c_\mathrm{J} = \mathcal{E}_{\mathsf{pk}}(\mathsf{pw_J}, \mathsf{VK_J}; \mathrm{R_J})$
(U3) $\mathsf{hk_I} = \mathsf{HashKG}(\mathsf{pk})$		
$\mathsf{hp_I} = \mathsf{ProjKG}(\mathsf{hk_I}; c_\mathrm{J})$		
$\sigma_\mathrm{I} = \mathsf{Sign}(\mathsf{SK_I}, (c_\mathrm{J}, \mathsf{hp_I}, \mathsf{hp_J}))$		
$\mathsf{sk_I} = \mathsf{Hash}(\mathsf{hk_J}, \mathsf{pw_I}, c_\mathrm{I})$		
$\mathsf{sk_J} = \mathsf{ProjHash}(\mathsf{hp_I}, \mathsf{pw_J}, c_\mathrm{J}, \mathrm{R_I})$		
$\mathsf{sk} = \mathsf{sk_I} \circ \mathsf{sk_J}$	$\xrightarrow{(\sigma_\mathrm{I}, \mathsf{hp_I})}$	
		(S4) abandonne si
		$\mathsf{Ver}(\mathsf{VK_I}, (c_\mathrm{J}, \mathsf{hp_I}, \mathsf{hp_J}), \sigma_\mathrm{I}) = 0$
		$\mathsf{sk_J} = \mathsf{Hash}(\mathsf{hk_I}, \mathsf{pw_J}, c_\mathrm{J})$
		$\mathsf{sk_I} = \mathsf{ProjHash}(\mathsf{hp_J}, \mathsf{pw_I}, c_\mathrm{I}, \mathrm{R_J})$
		$\mathsf{sk} = \mathsf{sk_I} \circ \mathsf{sk_J}$

À un haut niveau de description, les joueurs du protocole KOY/GL échangent des chiffrés $\mathsf{IND} - \mathsf{CCA}$ du mot de passe, sous la clef publique incluse dans la CRS. C'est donc essentiellement une mise en gage du mot de passe. Ils calculent alors la clef de session en combinant par la loi de groupe des SPHF des deux paires (mot de passe, chiffré). Plus précisément, chaque joueur choisit une clef de hachage pour la SPHF, et envoie la clef de projection correspondante à l'autre joueur. Chaque joueur peut alors calculer le résultat de sa propre fonction de hachage à l'aide de la clef de hachage, et le résultat de l'autre fonction en utilisant la clef de projection et la connaissance de l'aléa qui a été utilisé pour générer le chiffré du mot de passe. Tous les messages générés par le client sont liés par une signature (*one-time*), calculée dans le dernier message, et la clef publique de vérification est incluse dans la mise en gage et donc implicitement revérifiée dans la clef de session. Nous utiliserons dans le chapitre 9 des chiffrements étiquetés, et cette clef de vérification sera incluse dans l'étiquette. Il n'y a en effet aucune raison de la masquer vu que sa valeur est publique (et sert à l'authentification).

Pour voir rapidement pourquoi ce protocole est sûr, on considère d'abord le cas dans lequel l'adversaire est passif. Dans ce cas, la propriété pseudo-aléatoire de la fonction de hachage

assure que la valeur de clef de session va être calculatoirement indistinguable d'une valeur uniforme puisque l'adversaire ne connaît pas l'aléa qui a été utilisé pour chiffrer le mot de passe. Maintenant, si l'adversaire fournit un chiffré d'un mauvais mot de passe à un utilisateur, la sécurité va reposer sur la *smoothness* de la fonction, qui va garantir que la clef de session va être aléatoire et indépendante de toute la communication passée. Ainsi, pour gagner, l'adversaire doit générer un chiffrement d'un mot de passe correct. Pour cela, il pourrait essayer de copier ou modifier des chiffrements existants. Comme le schéma de chiffrement est $\mathsf{IND}-\mathsf{CCA}$, et donc non-malléable, modifier un chiffré n'est pas possible. Copier ne l'est pas non plus étant donné que soit la clef de vérification utilisée dans la session ne va pas correspondre (et donc la clef de session aura l'air aléatoire) soit la signature va être invalide (si l'adversaire change les clefs de projection sans changer la clef de vérification). La seule stratégie gagnante est donc de deviner le mot de passe et d'effectuer une attaque par dictionnaire en ligne, ce qui était bien le résultat espéré.

3.3 Protocoles basés sur la *multiparty computation*

Parallèlement au premier protocole pratique dans le modèle standard, mais avec une CRS (le protocole KOY [KOY01]), Goldreich et Lindell [GL01, GL06] proposent le premier protocole sans aucune hypothèse sur sa mise en œuvre (bien sûr dans le modèle standard, mais également sans CRS). Malheureusement, le protocole est assez complexe, fait appel à de nombreuses preuves *zero-knowledge*, et ne résiste pas aux attaques concurrentes. Néanmoins, ils parviennent à prouver qu'une attaque active ne permet d'éliminer qu'un seul mot de passe.

Ainsi, le seul protocole d'échange de clefs à base de mots de passe sûr contre un adversaire adaptatif, permettant des exécutions concurrentes, est le protocole proposé par Barak, Canetti, Lindell, Pass et Rabin [BCL+05]. Sa sécurité est valide dans le cadre UC (voir chapitre 5 page 58). Il utilise des techniques générales de *multiparty computation* et mène donc à des schémas assez inefficaces.

3.4 Deux autres familles

3.4.1 La famille SPEKE

SPEKE a été proposé par Jablon en 1996 [Jab96]. Il s'agit d'un protocole Diffie-Hellman dans lequel la base de calcul dépend du mot de passe. L'efficacité est *a priori* similaire à celle de EKE, mais avec les mêmes contraintes : pour EKE, un chiffrement symétrique adéquat sur des éléments de groupe, utilisant le mot de passe comme clef de chiffrement, doit être trouvé ; pour SPEKE, un moyen efficace de dérivation d'un élément de groupe (dont le logarithme discret est inconnu) à partir du mot de passe doit être utilisé. Si une telle dérivation est possible dans le modèle de l'oracle aléatoire, et c'est l'analyse menée par MacKenzie [Mac01] qui en prouve la sécurité, elle est très coûteuse en pratique, à moins de travailler sur des sous-groupes de faible cofacteur. Récemment, Icart et Coron ont montré une méthode efficace pour construire un oracle aléatoire sur une courbe elliptique [Ica09, CI09].

3.4.2 La famille SRP

SRP, pour *Secure Remote Protocol*, a été proposé par Wu en 1998 [Wu98]. Il s'agit en fait d'une variante de OKE très efficace, où un dérivé du mot de passe est utilisé comme masque (chiffrement symétrique), mais en utilisant simplement l'addition dans $\mathbb{Z}_p^*$.

Cette construction, qui « casse » la structure de groupe de $\mathbb{Z}_p^*$, empêche toute analyse de sécurité. Néanmoins, des attaques ont montré la possibilité de tester 2 mots de passe pour une attaque active. Wu l'avait proposée comme RFC 2945 mais, en raison de l'attaque ci-dessus, une variante l'a remplacée [Wu02].

De son côté, Jablon a combiné SPEKE et $\mathsf{SRP}-3$ pour proposer $\mathsf{SRP}-4$ [Jab02]. Le coût de la dérivation d'un élément de groupe à partir du mot de passe demeure, et l'absence de preuve de sécurité de toutes les variantes de SRP également.

Chapitre 4

Extension aux groupes

Depuis le protocole Diffie-Hellman [DH76], plusieurs extensions de ce protocole au cadre de groupe ont été proposées dans la littérature [BD94, AST98] sans un cadre de sécurité formel. Le premier cadre de sécurité pour les échanges de clefs de groupe (fortement basé sur [BPR00]) a été proposé par Bresson *et al.* [BCPQ01], qui l'ont ensuite étendu au cadre dynamique et concurrent [BCP01, BCP02a], en utilisant les principes de Bellare *et al.* [BR94, BR95].

Dans le scénario à base de mots de passe, la plupart des travaux ont considéré le cas à deux joueurs. Pour les groupes, il y a eu quelques protocoles proposés, à partir du travail initial de Bresson *et al.* [BCP02b, BCP07] jusqu'aux propositions plus récentes de Dutta et Barua [DB06], Abdalla *et al.* [ABGS07, ABCP06, AP06], et Bohli *et al.* [BGS06].

4.1 Les cadres de sécurité

Le cadre des protocoles d'échange de clefs pour des groupes est très similaire à celui pour deux joueurs, excepté qu'on s'intéresse à un ensemble de joueurs $\mathrm{P}_1, \ldots, \mathrm{P}_n$, disposés en cycle (ie P_{n+1} est implicitement égal à P_1). On suppose que le cycle est ordonné d'un bout à l'autre de l'exécution pour faciliter l'échange de messages. L'idée de base reste la même : une clef de session doit être indistinguable d'une clef aléatoire aux yeux d'un adversaire. Les mêmes requêtes (`send`, `execute`, `reveal`, `test`) modélisent les capacités de l'adversaire et la sécurité sémantique d'un protocole est définie comme précédemment, dans le cadre de sécurité FTG comme dans le cadre de sécurité ROR. C'est l'authentification mutuelle qui a du sens ici, et elle signifie qu'il est impossible que plusieurs joueurs acceptent des clefs différentes.

Katz et Yung [KY03] définissent les partenaires de la façon suivante : l'ensemble des identités des partenaires (l'identifiant du groupe) est fixé à l'avance, les identifiants de session sont la concaténation de tous les messages d'une instance et deux oracles sont partenaires s'ils ont le même identifiant de session et de groupe. Dutta, Barua et Sarkar [DBS04] en ont proposé une variante dans laquelle l'identifiant de session est plus simple (une suite de nombres) mais plus difficile à mettre en pratique (unicité), ce cadre est donc peu utilisé. Katz et Shin [KS05] ont été les premiers à considérer les adversaires infiltrés dans le groupe (*insiders*), et se sont placés dans le cadre UC, nous en parlerons donc dans le chapitre 5. Un cadre de preuve par simulation a aussi été développé, sur le modèle de [BMP00].

4.2 Protocoles existants

Dans ces protocoles, l'espace des clefs de session est généralement $\{0,1\}^\ell$ muni de la distribution uniforme, et les calculs sont effectués dans un groupe $\mathbb{G}$, d'ordre q premier et de générateur g, tel que le DDH soit difficile.

4.2.1 Hypothèses calculatoires [BCP03a]

Le problème $\mathsf{G}-\mathsf{CDH}$. Soient $\mathrm{I}=\{1,\ldots,n\}$ et $x_1,\ldots,x_n$ n valeurs aléatoires. Alors étant donné des sous-ensembles stricts J_ℓ de I, et les valeurs $\{g^{\prod_{x\in \mathrm{J}_\ell} x}\}$, il est impossible de calculer $g^{x_1\cdots x_n}$. Le problème $\mathsf{G}-\mathsf{CDH}$ peut être réduit au problème CDH.

Le problème $\mathsf{G}-\mathsf{DDH}$. Soient $\mathrm{I}=\{1,\ldots,n\}$ et $x_1,\ldots,x_n$ n valeurs aléatoires. Alors étant donné des sous-ensembles stricts J_ℓ de I, et les valeurs $\{g^{\prod_{x\in \mathrm{J}_\ell} x}\}$ et g^r, il est impossible de décider si $r=x_1\ldots x_n$. Le problème $\mathsf{G}-\mathsf{DDH}$ peut être réduit au problème DDH.

4.2.2 Protocoles basés sur le $\mathsf{G}-\mathsf{DDH}$

Les auteurs de [BCP02b, BCP07] présentent un protocole avec un nombre linéaire de tours, sûr dans le modèle de l'oracle aléatoire et du chiffrement idéal sous l'hypothèse $\mathsf{G}-\mathsf{DDH}$. Le protocole consiste en deux étapes (voir la figure 4.1 pour le cas général et la figure 4.2 pour un exemple). Dans la première, chaque joueur reçoit un chiffré de son prédécesseur et le déchiffre pour obtenir X_{i-1}. Il calcule à partir de là $\mathrm{X}_i=\Phi(\mathrm{X}_{i-1},x_i,r_i)$ en masquant X_{i-1} à l'aide de deux valeurs aléatoires, puis il le chiffre et l'envoie à son successeur. Dans la seconde phase, quand U_n reçoit la valeur X_{n-1}, il le masque cette fois-ci à l'aide de l'opérateur Φ' et envoie sa valeur chiffrée (les fonctions Φ et Φ' sont décrites sur la figure). Chaque joueur peut alors déchiffrer et calculer la clef de session à partir de X'_n.

4.2.3 Protocoles basés sur celui de Burmester et Desmedt [BD94, BD05]

L'échange de clefs de Burmester et Desmedt est un protocole à nombre de tours constant basé sur un échange Diffie-Hellman. Il est résistant aux attaques passives et repose sur le CDH. Katz et Yung [KY03] en ont donné une preuve basée sur le DDH et ont construit un compilateur générique pour passer d'un échange de clefs à un échange de clefs authentifié. Ensuite, Kim, Lee et Lee [KLL04] ont proposé une variante de ce protocole utilisant des oracles aléatoires ainsi que des XOR. Le protocole de Burmester et Desmedt est présenté sur la figure 4.3 pour les joueurs P_{i-1}, P_i et P_{i+1} et celui de Kim, Lee et Lee sur la figure 4.4.

On peut schématiquement voir ce schéma en trois étapes génériques (figure 4.5). Tout d'abord, P_i choisit un exposant privé x_i, et calcule $\mathrm{X}_i=g^{x_i}$. Ensuite, P_i et P_{i+1} calculent une clef Diffie-Hellman $\mathrm{K}_i=g^{x_ix_{i+1}}={\mathrm{X}_{i+1}}^{x_i}={\mathrm{X}_i}^{x_{i+1}}$. Enfin, ils transmettent $\mathrm{Z}_i=\mathrm{K}_i/\mathrm{K}_{i-1}$ et $\mathrm{Z}_{i+1}=\mathrm{K}_{i+1}/\mathrm{K}_i$ pour permettre à chaque joueur de calculer la clef de session de proche en proche, puisque $\mathrm{K}_j=\mathrm{K}_i\mathrm{Z}_{i+1}\mathrm{Z}_{i+2}\ldots\mathrm{Z}_j$ pour tout j. Les trois mêmes étapes s'appliquent au protocole de Kim, Lee et Lee.

Ce protocole a été étendu au cas authentifié d'une part par Lee, Hwang et Lee [LHL04] dans le modèle de l'oracle aléatoire, et d'autre part par Dutta et Barua [DB06] dans le modèle de l'oracle aléatoire et du chiffrement idéal. L'idée naturelle est d'utiliser l'approche EKE, c'est-à-dire de chiffrer tous les messages :

$$\mathrm{X}_i^*=\mathcal{E}_{\mathsf{pw}}(\mathrm{X}_i)\quad\text{et}\quad\mathrm{Z}_i^*=\mathcal{E}_{\mathsf{pw}}(\mathrm{Z}_i)$$

Mais le problème qui apparaît est que dans le protocole de Burmester et Desmedt, on a la relation suivante : $\mathrm{Z}_1\times\cdots\times\mathrm{Z}_n=1$, ce qui permet une attaque par dictionnaire. En effet, il suffit de choisir un mot de passe pw, de calculer pour tout i, $\mathrm{Z}_i=\mathcal{D}_{\mathsf{pw}}(\mathrm{Z}_i^*)$, et enfin de vérifier si l'égalité est vraie.

Pour surmonter cette difficulté, Dutta et Barua ont proposé dans [DB06] d'utiliser trois schémas de chiffrement différents pour supprimer la redondance (figure 4.6) : si $i=n$, la valeur

Fig. 4.1 – Le protocole [BCP02b]

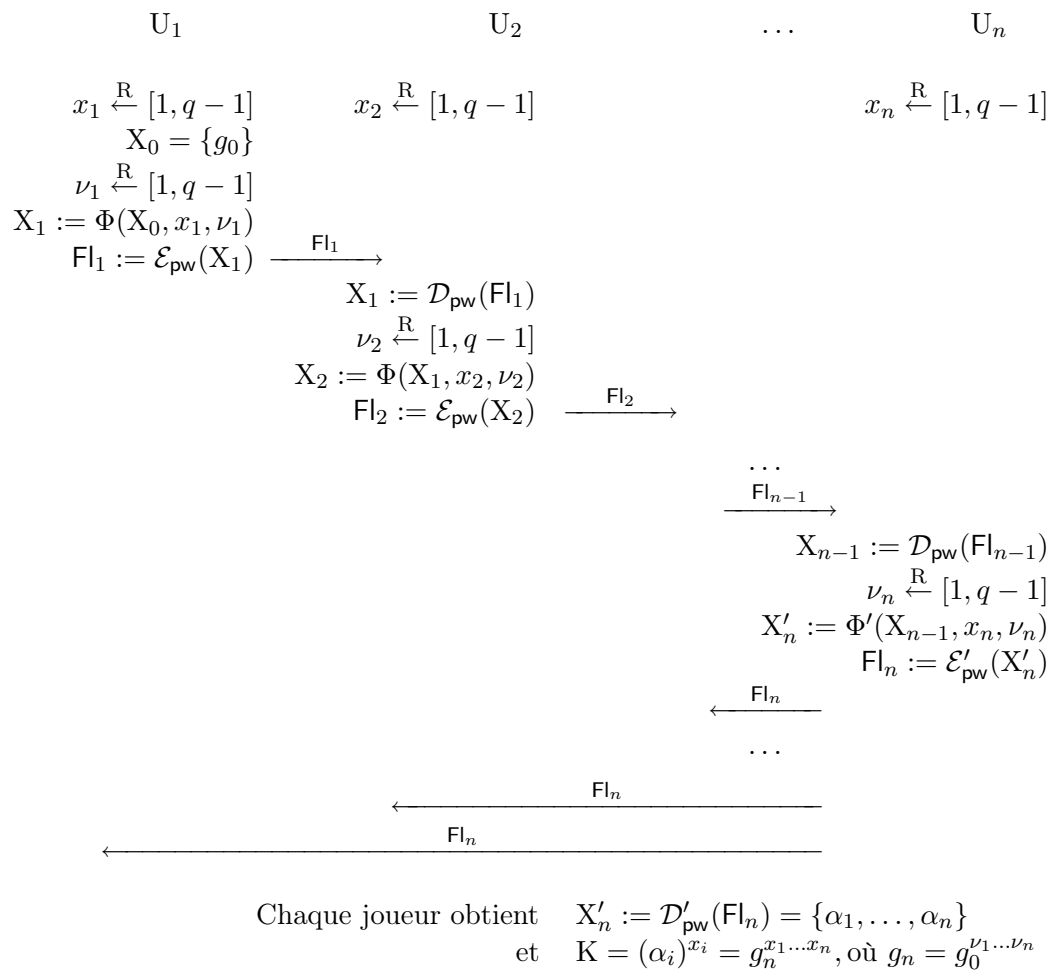


Fig. 4.2 – Le protocole [BCP02b] ($n = 4$)

g_0					$= \mathrm{X}_0$
g_1	$g_1^{x_1}$				$= \mathrm{X}_1 = \Phi(\mathrm{X}_0, x_1, \nu_1)$
$g_2^{x_2}$	$g_2^{x_1}$	$g_2^{x_1x_2}$			$= \mathrm{X}_2 = \Phi(\mathrm{X}_1, x_2, \nu_2)$
$g_3^{x_2x_3}$	$g_3^{x_1x_3}$	$g_3^{x_1x_2}$	$g_3^{x_1x_2x_3}$		$= \mathrm{X}_3 = \Phi(\mathrm{X}_2, x_3, \nu_3)$
$g_4^{x_2x_3x_4}$	$g_4^{x_1x_3x_4}$	$g_4^{x_1x_2x_4}$	$g_4^{x_1x_2x_3}$	$g_4^{x_1x_2x_3x_4}$	$= \mathrm{X}_4 = \Phi(\mathrm{X}_3, x_4, \nu_4)$
$(g_4^{x_2x_3x_4})^{x_1}$	$(g_4^{x_1x_3x_4})^{x_2}$	$(g_4^{x_1x_2x_4})^{x_3}$	$(g_4^{x_1x_2x_3})^{x_4}$	$g_4^{x_1x_2x_3x_4}$	$= \mathsf{sk}$

envoyée à la fin est $\mathcal{E}''_{\mathsf{pw}}(\mathrm{T}_n)$. Mais le problème reste identique et une attaque par dictionnaire a été exhibée dans [ABCP06]. Supposons pour simplifier que $n = 3$, que P_1 et P_2 sont honnêtes et que P_3 est contrôlé par l'attaquant. Ce dernier attend alors que les deux premiers joueurs aient envoyé leurs valeurs $\mathrm{X}_1^* = \mathcal{E}_{\mathsf{pw}}(\mathrm{X}_1)$ et $\mathrm{X}_2^* = \mathcal{E}_{\mathsf{pw}}(\mathrm{X}_2)$ et fixe $\mathrm{X}_3^* = \mathrm{X}_3$ (et donc implicitement $x_3 = x_1$). Ceci impose $\mathrm{K}_1 = \mathrm{K}_2$ et $\mathrm{Z}_2 = 0$. La valeur finale envoyée par P_2 est donc $\mathcal{E}_{\mathsf{pw}}(0\|s_2)$, ce qui permet une attaque par dictionnaire.

Fig. 4.3 – Le protocole [BD94, BD05]

$$
\begin{array}{lll}
x_{i-1} \leftarrow \mathbb{Z}_p & x_i \leftarrow \mathbb{Z}_p & x_{i+1} \leftarrow \mathbb{Z}_p \\
\mathrm{X}_{i-1} \leftarrow g^{x_{i-1}} & \mathrm{X}_i \leftarrow g^{x_i} & \mathrm{X}_{i+1} \leftarrow g^{x_{i+1}} \\
\qquad \xleftarrow{\mathrm{X}_{i-1}} & \qquad \xleftarrow{\mathrm{X}_i} & \qquad \xleftarrow{\mathrm{X}_{i+1}} \\
\mathrm{K}_{i-1} \leftarrow \mathrm{X}_i{}^{x_{i-1}} & \mathrm{K}_i \leftarrow \mathrm{X}_{i+1}{}^{x_i} & \mathrm{K}_{i+1} \leftarrow \mathrm{X}_{i+2}{}^{x_{i+1}} \\
\mathrm{K}_{i-2} \leftarrow \mathrm{X}_{i-2}{}^{x_{i-1}} & \mathrm{K}_{i-1} \leftarrow \mathrm{X}_{i-1}{}^{x_i} & \mathrm{K}_i \leftarrow \mathrm{X}_i{}^{x_{i+1}} \\
\mathrm{Z}_{i-1} \leftarrow \mathrm{K}_{i-1}/\mathrm{K}_{i-2} & \mathrm{Z}_i \leftarrow \mathrm{K}_i/\mathrm{K}_{i-1} & \mathrm{Z}_{i+1} \leftarrow \mathrm{K}_{i+1}/\mathrm{K}_i \\
\qquad \xleftarrow{\mathrm{Z}_{i-1}} & \qquad \xleftarrow{\mathrm{Z}_i} & \qquad \xleftarrow{\mathrm{Z}_{i+1}}
\end{array}
$$

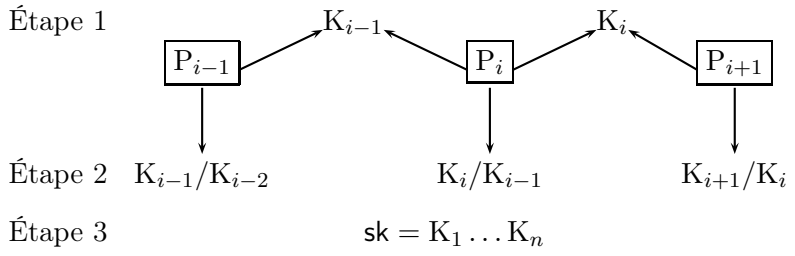
$$\mathsf{sk} = \mathrm{K}_1 \circ \mathrm{K}_2 \circ \cdots \circ \mathrm{K}_n$$

Fig. 4.4 – Le protocole [KLL04]

$$
\begin{array}{lll}
s_{i-1} \leftarrow \$ & s_i \leftarrow \$ & s_{i+1} \leftarrow \$ \\
x_{i-1} \leftarrow \mathbb{Z}_p & x_i \leftarrow \mathbb{Z}_p & x_{i+1} \leftarrow \mathbb{Z}_p \\
\mathrm{X}_{i-1} \leftarrow g^{x_{i-1}} & \mathrm{X}_i \leftarrow g^{x_i} & \mathrm{X}_{i+1} \leftarrow g^{x_{i+1}} \\
\qquad \xleftarrow{\mathrm{X}_{i-1}} & \qquad \xleftarrow{\mathrm{X}_i} & \qquad \xleftarrow{\mathrm{X}_{i+1}} \\
\mathrm{K}_{i-1} \leftarrow \mathrm{H}(\mathrm{X}_i{}^{x_{i-1}}) & \mathrm{K}_i \leftarrow \mathrm{H}(\mathrm{X}_{i+1}{}^{x_i}) & \mathrm{K}_{i+1} \leftarrow \mathrm{H}(\mathrm{X}_{i+2}{}^{x_{i+1}}) \\
\mathrm{K}_{i-2} \leftarrow \mathrm{H}(\mathrm{X}_{i-2}{}^{x_{i-1}}) & \mathrm{K}_{i-1} \leftarrow \mathrm{H}(\mathrm{X}_{i-1}{}^{x_i}) & \mathrm{K}_i \leftarrow \mathrm{H}(\mathrm{X}_i{}^{x_{i+1}}) \\
\mathrm{Z}_{i-1} \leftarrow \mathrm{K}_{i-1} \oplus \mathrm{K}_{i-2} & \mathrm{Z}_i \leftarrow \mathrm{K}_i \oplus \mathrm{K}_{i-1} & \mathrm{Z}_{i+1} \leftarrow \mathrm{K}_{i+1} \oplus \mathrm{K}_i \\
\mathrm{T}_{i-1} \leftarrow s_{i-1} & \mathrm{T}_i \leftarrow s_i & \mathrm{T}_{i+1} \leftarrow s_{i+1} \\
\qquad \xleftarrow{\mathrm{Z}_{i-1}\|\mathrm{T}_{i-1}} & \qquad \xleftarrow{\mathrm{Z}_i\|\mathrm{T}_i} & \qquad \xleftarrow{\mathrm{Z}_{i+1}\|\mathrm{T}_{i+1}}
\end{array}
$$

$$\mathsf{sk} = \mathrm{H}'(s_1 \circ s_2 \circ \cdots \circ s_n)$$

Fig. 4.5 – Une version générique du protocole [BD94]



Lee, Hwang et Lee [LHL04] ont ensuite essayé de supprimer le dernier chiffrement (figure 4.7), mais les auteurs de [ABCP06] ont alors construit une attaque contre la sécurité sémantique de ce protocole. Pour simplifier, posons $n = 4$. Supposons que l'attaquant exécute deux protocoles en parallèle (voir les schémas de la figure 4.8) tels que P_1 est honnête et $\mathcal{A}$ joue le rôle de P_2, P_3 et P_4. Dans chacune de ces deux exécutions, il rejoue $\mathrm{E}_{\mathsf{pw}}(\mathrm{X}_1)$ et $\mathrm{E}_{\mathsf{pw}}(\mathrm{X}'_1)$ (donc implicitement x_1, X_1, x'_1 et X'_1). Ceci impose que tous les K_i sont alors égaux et la clef de session est la même dans les deux exécutions.

Ainsi, si $\mathcal{A}$ demande une requête `test` dans la première exécution, et une requête `reveal` dans la deuxième exécution, il peut comparer les deux réponses obtenues et donc distinguer

Fig. 4.6 – Le protocole [DB06]

$$
\begin{array}{lll}
s_{i-1} \leftarrow \$ & s_i \leftarrow \$ & s_{i+1} \leftarrow \$ \\
x_{i-1} \leftarrow \mathbb{Z}_p & x_i \leftarrow \mathbb{Z}_p & x_{i+1} \leftarrow \mathbb{Z}_p \\
\mathrm{X}_{i-1} \leftarrow g^{x_{i-1}} & \mathrm{X}_i \leftarrow g^{x_i} & \mathrm{X}_{i+1} \leftarrow g^{x_{i+1}} \\
\xleftarrow{\mathcal{E}_{\mathsf{pw}}(\mathrm{X}_{i-1})} & \xleftarrow{\mathcal{E}_{\mathsf{pw}}(\mathrm{X}_i)} & \xleftarrow{\mathcal{E}_{\mathsf{pw}}(\mathrm{X}_{i+1})} \\
\mathrm{K}_{i-1} \leftarrow \mathrm{H}({\mathrm{X}_i}^{x_{i-1}}) & \mathrm{K}_i \leftarrow \mathrm{H}({\mathrm{X}_{i+1}}^{x_i}) & \mathrm{K}_{i+1} \leftarrow \mathrm{H}({\mathrm{X}_{i+2}}^{x_{i+1}}) \\
\mathrm{K}_{i-2} \leftarrow \mathrm{H}({\mathrm{X}_{i-2}}^{x_{i-1}}) & \mathrm{K}_{i-1} \leftarrow \mathrm{H}({\mathrm{X}_{i-1}}^{x_i}) & \mathrm{K}_i \leftarrow \mathrm{H}({\mathrm{X}_i}^{x_{i+1}}) \\
\mathrm{Z}_{i-1} \leftarrow \mathrm{K}_{i-1} \oplus \mathrm{K}_{i-2} & \mathrm{Z}_i \leftarrow \mathrm{K}_i \oplus \mathrm{K}_{i-1} & \mathrm{Z}_{i+1} \leftarrow \mathrm{K}_{i+1} \oplus \mathrm{K}_i \\
\mathrm{T}_{i-1} \leftarrow s_{i-1} & \mathrm{T}_i \leftarrow s_i & \mathrm{T}_{i+1} \leftarrow s_{i+1} \\
\xleftarrow{\mathcal{E}'_{\mathsf{pw}}(\mathrm{Z}_{i-1}\|\mathrm{T}_{i-1})} & \xleftarrow{\mathcal{E}'_{\mathsf{pw}}(\mathrm{Z}_i\|\mathrm{T}_i)} & \xleftarrow{\mathcal{E}'_{\mathsf{pw}}(\mathrm{Z}_{i+1}\|\mathrm{T}_{i+1})}
\end{array}
$$

$$\mathsf{sk} = \mathrm{H}'(s_1 \circ s_2 \circ \cdots \circ s_n)$$

Fig. 4.7 – Le protocole [LHL04]

$$
\begin{array}{lll}
x_{i-1} \leftarrow \mathbb{Z}_p & x_i \leftarrow \mathbb{Z}_p & x_{i+1} \leftarrow \mathbb{Z}_p \\
\mathrm{X}_{i-1} \leftarrow g^{x_{i-1}} & \mathrm{X}_i \leftarrow g^{x_i} & \mathrm{X}_{i+1} \leftarrow g^{x_{i+1}} \\
\xleftarrow{\mathcal{E}_{\mathsf{pw}}(\mathrm{X}_{i-1})} & \xleftarrow{\mathcal{E}_{\mathsf{pw}}(\mathrm{X}_i)} & \xleftarrow{\mathcal{E}_{\mathsf{pw}}(\mathrm{X}_{i+1})} \\
\mathrm{K}_{i-1} \leftarrow \mathrm{H}({\mathrm{X}_i}^{x_{i-1}}) & \mathrm{K}_i \leftarrow \mathrm{H}({\mathrm{X}_{i+1}}^{x_i}) & \mathrm{K}_{i+1} \leftarrow \mathrm{H}({\mathrm{X}_{i+2}}^{x_{i+1}}) \\
\mathrm{K}_{i-2} \leftarrow \mathrm{H}({\mathrm{X}_{i-2}}^{x_{i-1}}) & \mathrm{K}_{i-1} \leftarrow \mathrm{H}({\mathrm{X}_{i-1}}^{x_i}) & \mathrm{K}_i \leftarrow \mathrm{H}({\mathrm{X}_i}^{x_{i+1}}) \\
\mathrm{Z}_{i-1} \leftarrow \mathrm{K}_{i-1} \oplus \mathrm{K}_{i-2} & \mathrm{Z}_i \leftarrow \mathrm{K}_i \oplus \mathrm{K}_{i-1} & \mathrm{Z}_{i+1} \leftarrow \mathrm{K}_{i+1} \oplus \mathrm{K}_i \\
\xleftarrow{\mathrm{Z}_{i-1}} & \xleftarrow{\mathrm{Z}_i} & \xleftarrow{\mathrm{Z}_{i+1}}
\end{array}
$$

$$\mathsf{sk} = \mathrm{H}'(\mathrm{K}_1 \circ \mathrm{K}_2 \circ \cdots \circ \mathrm{K}_n)$$

entre la vraie clef de la première exécution (c'est-à-dire la clef de la deuxième exécution) et une clef aléatoire. $\mathcal{A}$ est donc un attaquant contre la sécurité sémantique du protocole.

Abdalla, Bresson, Chevassut et Pointcheval [ABCP06] ont alors proposé un schéma résistant à ces attaques, prouvé sûr dans le modèle de l'oracle aléatoire et du chiffrement idéal. Leur protocole, décrit sur la figure 4.9, commence par un nouveau tour de nonces aléatoires r_i et la session est définie par $\mathrm{S} = \mathrm{P}_1\|r_1\|\ldots\|\mathrm{P}_n\|r_n$. Ceci permet d'utiliser une clef de chiffrement différente pour chaque utilisateur et chaque session et cela évite le rejeu : $\mathsf{pw}_i = \mathrm{H}(\mathsf{pw}\|\mathrm{S}\|i)$.

Mais cette modification n'est pas suffisante car les messages ne sont pas authentifiés et cela permet une attaque contre la non-malléabilité. Par exemple, si $n = 4$, l'attaquant envoie $\mathrm{Z}_1 \times g$ et Z_2/g à P_3 et P_4 au lieu de Z_1 et Z_2. Ainsi, P_3 et P_4 voient la même transcription mais calculent des clefs différentes. Dans le cas réel, les clefs sont différentes, alors que dans le cas aléatoire, les clefs sont (aléatoires mais) égales. Les deux situations sont donc distinguables. Pour pallier ce problème, les auteurs ont aussi ajouté un tour d'authentification et la clef de session est alors un haché de sk et de la transcription : $\mathsf{Auth}_i = \mathrm{H}'(\mathrm{S}\|\mathcal{E}_{\mathsf{pw}_1}(\mathrm{X}_1)\|\mathrm{Z}_1\|\ldots\|\mathcal{E}_{\mathsf{pw}_n}(\mathrm{X}_n)\|\mathrm{Z}_n\|\mathsf{sk}\|i)$.

4.2.4 Protocoles dans le modèle standard

Afin de construire un protocole dans le modèle standard, Abdalla et Pointcheval [AP06] sont partis de la version générique du protocole de Burmester et Desmedt (figure 4.5) en utilisant un protocole à deux joueurs sûr dans le modèle standard (il s'agit de celui de Gennaro et Lindell [GL03], décrit dans la section 3.2). L'idée est d'exécuter des $2 - \mathsf{PAKE}$ entre deux

Fig. 4.8 – L'attaque contre la sécurité sémantique de [LHL04]

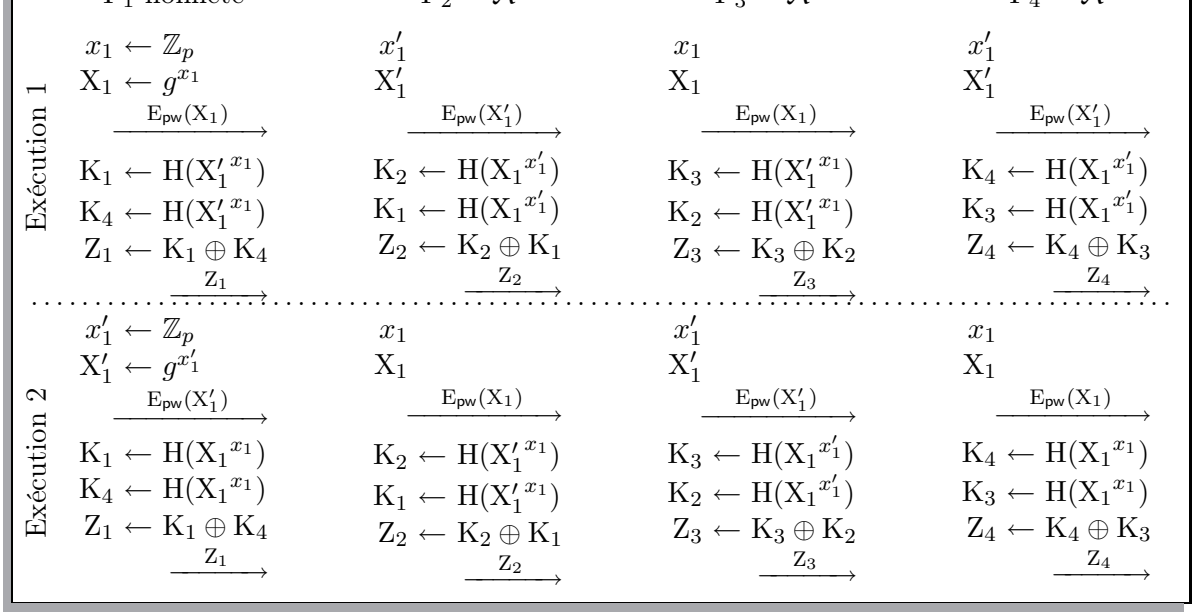
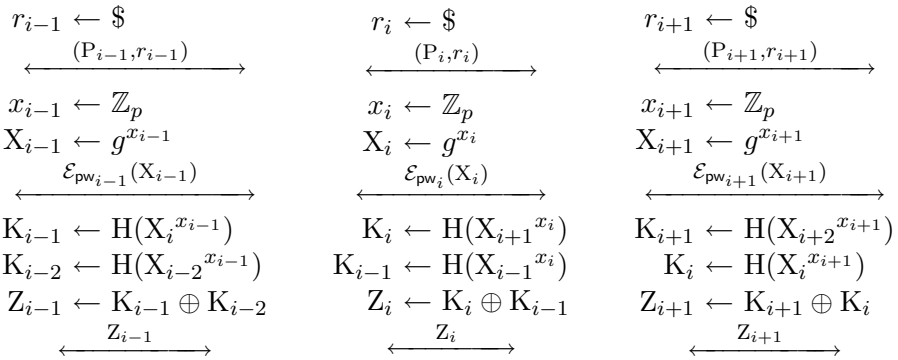


Fig. 4.9 – Le protocole [ABCP06]



joueurs consécutifs de façon à générer deux fois deux clefs pour chacun des joueurs, partagées entre lui et ses plus proches voisins. Chaque joueur peut alors authentifier son successeur et son prédécesseur en utilisant l'une de ces clefs, et utiliser l'autre clef pour générer la clef du groupe, à la Burmester et Desmedt. Des signatures authentifient alors la transcription de tous les messages qui ont été envoyés précédemment et qui doivent être tous liés. Le protocole est décrit schématiquement sur la figure 4.10. Les notations sont expliquées dans la section 4.2.5.

Le protocole utilise des familles de fonctions de hachage universelles (voir page 21). Le schéma de signature utilisé dans le protocole est celui introduit par Goldwasser, Micali et Rivest [GMR88]. Le protocole étant basé sur [GL03], il utilise aussi des schémas de chiffrements étiquetés (*labeled encryption*) et des SPHF– toutes ces notions sont décrites dans le chapitre 1.

Bohli, González Vasco et Steinwandt [BGS06] en ont alors proposé une version plus efficace à 3 tours, basée à nouveau sur [GL03]. Mais leur protocole n'utilise pas de schéma de signature *one-time* et ils utilisent la définition originale de Cramer et Shoup [CS02] pour les SPHF. Il est décrit sur la figure 4.11.

Fig. 4.10 – Le protocole [AP06]

Utilisateur U_i

$\mathsf{pid}_i = \{\mathrm{U}_1, \ldots, \mathrm{U}_n\}$
$(\mathsf{SK}_i, \mathsf{VK}_i) \xleftarrow{\$} \mathsf{SKG}(1^k)$
$\ell_i = \mathsf{VK}_i \mid \mathrm{U}_1 \mid \ldots \mid \mathrm{U}_n$
$c_i^{\mathtt{R}} = \mathsf{Enc}(\mathsf{pk}, \ell_i, \mathsf{pw}; r_i^{\mathtt{R}})$

$\xrightarrow{\ell_i,\ c_i^{\mathtt{R}}}$

$\mathsf{hk}_i^{\mathrm{L}} \xleftarrow{\$} \mathsf{HashKG}(\mathsf{pk})$
$\mathsf{hp}_i^{\mathrm{L}} \xleftarrow{\$} \mathsf{ProjKG}(\mathsf{hk}_i^{\mathrm{L}}, \ell_{i-1}, c_{i-1}^{\mathtt{R}})$
$c_i^{\mathtt{L}} = \mathsf{Enc}(\mathsf{pk}, \ell_i, \mathsf{pw}; r_i^{\mathtt{L}})$

$\xrightarrow{\mathsf{hp}_i^{\mathrm{L}},\ c_i^{\mathtt{L}}}$

$\mathsf{hk}_i,\ \mathsf{hk}_i^{\mathrm{R}} \xleftarrow{\$} \mathsf{HashKG}(\mathsf{pk})$
$\mathsf{hp}_i \xleftarrow{\$} \mathsf{ProjKG}(\mathsf{hk}_i, \ell_{i+1}, c_{i+1}^{\mathtt{L}})$
$\mathsf{hp}_i^{\mathrm{R}} \xleftarrow{\$} \mathsf{ProjKG}(\mathsf{hk}_i^{\mathrm{R}}, \ell_{i+1}, c_{i+1}^{\mathtt{L}})$
$K_{i+1}^{\mathtt{L}} = \mathsf{ProjHash}(\mathsf{hp}_{i+1}^{\mathrm{L}}, c_i^{\mathtt{R}}, \ell_i, \mathsf{pw}, r_i^{\mathtt{R}})$
$K_i^{\mathtt{R}} = \mathsf{Hash}(\mathsf{hk}_i^{\mathrm{R}}, c_{i+1}^{\mathtt{L}}, \ell_{i+1}, \mathsf{pw})$
$X_i^{\mathtt{R}} = K_{i+1}^{\mathtt{L}} \cdot K_i^{\mathtt{R}}$
$\mathtt{test}_i^{\mathrm{R}} = \mathsf{UH}_1(X_i^{\mathtt{R}})$
$\sigma_i^{\mathtt{R}} = \mathsf{Sign}(\mathsf{SK}_i, T_i^{\mathtt{R}})$

$\xrightarrow{\mathsf{hp}_i,\ \mathsf{hp}_i^{\mathrm{R}},\ \mathtt{test}_i^{\mathrm{R}},\ \sigma_i^{\mathtt{R}}}$

si $\mathsf{Ver}(\mathsf{VK}_{i-1}, T_{i-1}^{\mathtt{R}}, \sigma_{i-1}^{\mathtt{R}}) = 0$ alors $\mathsf{acc}_i = \mathsf{false}$
$K_i^{\mathtt{L}} = \mathsf{Hash}(\mathsf{hk}_i^{\mathrm{L}}, c_{i-1}^{\mathtt{R}}, \ell_{i-1}, \mathsf{pw})$
$K_{i-1}^{\mathtt{R}} = \mathsf{ProjHash}(\mathsf{hp}_{i-1}^{\mathrm{R}}, c_i^{\mathtt{L}}, \ell_i, \mathsf{pw}, r_i^{\mathtt{L}})$
$X_i^{\mathtt{L}} = K_i^{\mathtt{L}} \cdot K_{i-1}^{\mathtt{R}}$
si $\mathtt{test}_{i-1}^{\mathrm{R}} \neq \mathsf{UH}_1(X_i^{\mathtt{L}})$ alors $\mathsf{acc}_i = \mathsf{false}$
$\mathtt{test}_i^{\mathrm{L}} = \mathsf{UH}_2(X_i^{\mathtt{L}})$
$K_i = \mathsf{Hash}(\mathsf{hk}_i, c_{i+1}^{\mathtt{L}}, \ell_{i+1}, \mathsf{pw})$
$K_{i-1} = \mathsf{ProjHash}(\mathsf{hp}_{i-1}, c_i^{\mathtt{L}}, \ell_i, \mathsf{pw}, r_i^{\mathtt{L}})$
$X_i = K_i / K_{i-1}$

$\xrightarrow{X_i,\ \mathtt{test}_i^{\mathrm{L}}}$

si $\mathtt{test}_{i+1}^{\mathrm{L}} \neq \mathsf{UH}_2(X_i^{\mathtt{R}})$ alors $\mathsf{acc}_i = \mathsf{false}$
si $\prod_{l=1}^{n} X_l \neq 1$ alors $\mathsf{acc}_i = \mathsf{false}$
$T = T_1 \mid \ldots \mid T_n$
$\sigma_i = \mathsf{Sign}(\mathsf{SK}_i, T)$

$\xrightarrow{\sigma_i}$

pour $j = 1, \ldots, i-1, i+1, \ldots, n$
  si $\mathsf{Ver}(\mathsf{VK}_j, T, \sigma_j) = 0$ alors $\mathsf{acc}_i = \mathsf{false}$
$\mathsf{msk} = K_i^n \cdot \prod_{j=1}^{n-1} X_{i+j}^{n-j}$
$\mathsf{sk} = \mathsf{UH}'(\mathsf{msk})$
$\mathsf{acc}_i = \mathsf{true}$
$\mathsf{sid}_i = T$

Fig. 4.11 – Le protocole [BGS06]

1. Chaque joueur U_i choisit uniformément une valeur aléatoire $k_i \in \mathrm{K}$ et un nonce aléatoire r_i. Il construit alors $\mathrm{C}_i = \mathrm{C}_\rho(\mathsf{pw}, r_i)$, $\mathsf{hp}_i = \mathsf{ProjKG}(\mathsf{hk}_i)$ et il envoie $\mathrm{M}_i^1 = (\mathrm{U}_i, \mathsf{hp}_i, \mathrm{C}_i)$. Quand il a reçu tous les messages provenant des joueurs P_j, il vérifie que toutes les valeurs c_j sont dans le bon ensemble.
2. Chaque joueur U_i calcule $\mathrm{Z}_{i,i+1} = \mathsf{Hash}(\mathsf{hk}_i, \mathsf{pw}, \mathrm{C}_{i+1}) \times \mathsf{Hash}(\mathsf{hk}_{i+1}, \mathsf{pw}, \mathrm{C}_i)$ et $\mathrm{Z}_{i,i-1} = \mathsf{Hash}(\mathsf{hk}_i, \mathsf{pw}, \mathrm{C}_{i-1}) \times \mathsf{Hash}(\mathsf{hk}_{i-1}, \mathsf{pw}, \mathrm{C}_i)$. Il fixe ensuite $\mathrm{X}_i = \mathrm{Z}_{i,i+1}/\mathrm{Z}_{i,i-1}$ et choisit une valeur aléatoire r'_i pour calculer une mise en gage $\mathrm{C}_\rho(\mathrm{U}_i, \mathrm{X}_i, r'_i)$. Il envoie alors $\mathrm{M}_i^2 = (\mathrm{U}_i, \mathrm{C}_\rho(\mathrm{U}_i, \mathrm{X}_i, r'_i))$.
3. Chaque joueur envoie $\mathrm{M}_i^3 = (\mathrm{U}_i, \mathrm{X}_i, r'_i)$ et peut alors vérifier après avoir reçu toutes ces valeurs que $\mathrm{X}_1 \dots \mathrm{X}_n = 0$ et que les mises en gage sont correctes. Si l'un de ces tests échoue, le joueur refuse et termine l'exécution du protocole. Sinon, il peut calculer les valeurs

$$\begin{aligned} \mathrm{Z}_{i-1,i-2} &= \mathrm{Z}_{i,i-1}/\mathrm{X}_{i-1} \\ \mathrm{Z}_{i-2,i-3} &= \mathrm{Z}_{i-1,i-2}/\mathrm{X}_{i-2} \\ \mathrm{Z}_{i,i+1} &= \mathrm{Z}_{i+1,i+2}/\mathrm{X}_{i+2} \end{aligned}$$

   La clef maître est alors définie par $\mathrm{K} = (\mathrm{Z}_{1,2}, \mathrm{Z}_{2,3}, \dots, \mathrm{Z}_{n-1,n}, \mathrm{Z}_{n,1})$. La clef de session vaut alors $\mathsf{sk}_i = \mathrm{F}_{\mathrm{UH(K)}}(v_1)$ et l'identifiant de session $\mathsf{sid}_i = \mathrm{F}_{\mathrm{UH(K)}}(v_0)$. Le joueur accepte et termine le protocole.

4.2.5 Compilateur générique

Abdalla, Bohli, González Vasco et Steinwandt ont exhibé dans [ABGS07] un compilateur générique dans le modèle standard (avec CRS) pour construire un protocole de groupe à partir d'un échange de clefs à deux joueurs authentifié (noté $2 - \mathsf{PAKE}$) sûr dans le sens des cadres de sécurité décrits dans le chapitre 2. Il peut être vu comme une généralisation de [BGS06].

Soit $\mathcal{C}$ un schéma de mise en gage non interactif, non-malléable et parfaitement *binding* (voir page 25). Avec une CRS ρ, un tel schéma peut être construit à partir de n'importe quel schéma de chiffrement non-malléable et sûr pour des chiffrements multiples (en particulier à partir de n'importe quel schéma $\mathsf{IND} - \mathsf{CCA}_2$).

Fig. 4.12 – Le compilateur générique [ABGS07]

1. Pour $i = 1, \dots, n$, exécuter $2 - \mathsf{PAKE}(\mathrm{U}_i, \mathrm{U}_{i+1})$, à l'issue duquel chaque utilisateur possède deux clefs $\mathrm{K}_{i,i+1}$ et $\mathrm{K}_{i,i-1}$ partagées avec chacun de ses voisins directs.
2. Chaque joueur U_i calcule $\mathrm{X}_i = \mathrm{K}_{i,i+1} + \mathrm{K}_{i,i-1}$ et choisit une valeur aléatoire r_i pour calculer la mise en gage $\mathrm{C}_i = \mathrm{C}_\rho(i, \mathrm{X}_i; r_i)$. Il envoie alors $(\mathrm{U}_i, \mathrm{C}_i)$.
3. Chaque joueur envoie $(\mathrm{U}_i, \mathrm{X}_i, r_i)$ et peut alors vérifier après avoir reçu toutes ces valeurs que $\mathrm{X}_1 \oplus \dots \oplus \mathrm{X}_n = 0$ et que les mises en gage sont correctes. Si l'un de ces tests échoue, le joueur refuse et termine l'exécution du protocole. Sinon, il pose alors $\mathrm{K}_i = \mathrm{K}_{i,i+1}$ et calcule les $n-1$ valeurs $\mathrm{K}_{i-j} = \mathrm{K}_i \oplus \mathrm{X}_{i-1} \oplus \dots \oplus \mathrm{X}_{i-j}$. La clef maître est alors définie par $\mathrm{K} = (\mathrm{K}_1, \dots, \mathrm{K}_n, \mathsf{pid}_i)$, pid_i étant l'ensemble des identités du groupe tel que vu par U_i, lui-même inclus. La clef de session vaut alors $\mathsf{sk}_i = \mathrm{F}_{\mathrm{UH(K)}}(v_1)$ et l'identifiant de session $\mathsf{sid}_i = \mathrm{F}_{\mathrm{UH(K)}}(v_0)$. Le joueur accepte et termine le protocole.

Soit $\mathcal{F} = \{\mathrm{F}^\ell\}_{\ell \in \mathbb{N}}$ une famille de fonctions pseudo-aléatoires résistantes aux collisions, telles qu'utilisées par Katz et Shin [KS05]. On suppose que chaque fonction F^ℓ s'écrit $\{\mathrm{F}^\ell_\eta\}_{\eta \in \{0,1\}^{\mathrm{L}}}$ où L est de taille superpolynomiale, et on appelle $v_0(\ell)$ une valeur publique telle qu'aucun

adversaire ne puisse trouver deux indices différents λ et μ dans $\{0,1\}^{\mathrm{L}}$ vérifiant $\mathrm{F}_{\lambda}^{\ell}(v_0(\ell)) = \mathrm{F}_{\mu}^{\ell}(v_0(\ell))$. On note v_1 une deuxième telle valeur, qui sera utilisée dans la dérivation de clef, et on encode v_0 et v_1 dans la CRS.

Soit $\mathcal{UH}$ une famille de fonctions de hachage universelles qui fait correspondre la concaténation de chaînes de bits de $\{0,1\}^{kn}$ et d'une identité $\mathsf{pid}_i^{s_i}$ à $\{0,1\}^{\mathrm{L}}$. La CRS choisit une fonction UH à l'intérieur de cette famille, et cette dernière sera utilisée pour choisir un indice pour la famille de fonctions pseudo-aléatoires décrite précédemment.

L'idée derrière le compilateur est d'utiliser la construction de Burmester et Desmedt [BD94] dans laquelle une clef de groupe est construite à partir de clefs construites deux-à-deux. Dès que ces clefs sont établies, chaque joueur doit mettre en gage le XOR des deux valeurs qu'il partage avec ses voisins. Ces valeurs sont révélées dans un tour suivant, ce qui permet à chacun des joueurs de déterminer toutes ces clefs. Intuitivement, si un adversaire n'a pas pu casser la sécurité de l'un des échanges à deux joueurs, il ne sera pas capable d'obtenir d'information sur la clef de groupe résultante (les XOR sont aléatoires pour lui). Ce compilateur, présenté figure 4.12, n'a pas besoin de techniques d'authentification supplémentaires, ni de modèles idéaux (tels qu'un oracle aléatoire), et il est symétrique dans le sens où tous les joueurs réalisent les mêmes opérations.

Chapitre 5

Le cadre de sécurité de composabilité universelle (UC)

Le cadre de la composabilité universelle (UC) pouvant être très technique, nous le décrivons rapidement dans la première section pour en donner les principes généraux. Tous les points seront repris formellement dans les sections qui suivront. Enfin, la dernière section fera le point sur l'état de l'art des protocoles d'échange de clefs dans le cadre UC et contiendra en particulier les notions utiles dans la suite de ce mémoire.

5.1 Vue d'ensemble

On se place ici dans le cadre très général du calcul partagé, aussi appelé *multiparty computation*, que nous avons brièvement évoqué dans la section 3.3, page 46. Il peut être défini comme l'interaction de n joueurs souhaitant calculer de manière sûre une fonction donnée de leurs entrées. La *sécurité* signifie ici l'exactitude de la sortie ainsi que le respect du secret des entrées, même en cas de triche d'un certain nombre de joueurs.

Concrètement, on se donne n joueurs P_i ayant chacun une entrée x_i, et une fonction f en n variables. On souhaite calculer $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ tel que chaque P_i ait connaissance de y_i et rien de plus.

Reprenons l'exemple parlant donné par Yao dans [Yao82a] pour illustrer cette définition : il s'agit du « problème des millionnaires » . Nous sommes en présence de deux millionnaires qui se rencontrent dans la rue et qui souhaitent établir lequel est le plus riche des deux, sans révéler la somme exacte que chacun des deux possède. La fonction considérée dans ce cas est alors une comparaison entre deux entiers : chacun apprendra le résultat (lequel des deux possède le maximum), mais rien de plus (en particulier pas la somme possédée par l'autre). On peut aussi songer à des schémas de vote : chaque électeur doit connaître le résultat du vote, et être certain de son exactitude, mais pas l'identité du candidat pour lequel a voté son voisin. Ces deux exemples sont des cas particuliers où les y_i sont tous égaux, mais reflètent bien la philosophie du calcul partagé. Pour un exemple où il est bien utile d'avoir des y_i différents, on peut imaginer un protocole de signature en aveugle [Cha83], utile dans les schémas électroniques de paiement : le signataire donne comme entrée sa clef privée et ne reçoit rien, tandis que l'utilisateur fournit le message à faire signer et reçoit la signature.

5.1.1 Objectif

Nous avons vu dans l'introduction la nécessité d'un cadre mathématique et d'une définition de sécurité pour démontrer rigoureusement la sécurité d'un protocole. Dans le cadre des protocoles partagés, la définition d'un tel cadre n'est pas si facile, étant donné la diversité de ces derniers.

Une bonne définition doit nécessairement découler en une approche correcte des cas particuliers (tels que le vote électronique) : on peut par exemple songer à faire une liste des propriétés requises (entrées secrètes, résultat correct, etc.), mais il est difficile de se persuader de son exhaustivité (par exemple, on refuse qu'un joueur se comporte de manière reliée aux joueurs qui l'entourent).

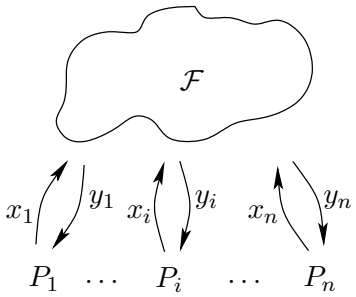
Pour résoudre ce problème, Canetti, qui a défini le cadre UC dans [Can01], est parti d'un point de vue complètement différent, en regardant pour chaque protocole de quelle manière il devrait « idéalement » se comporter.

5.1.2 Monde idéal et monde réel

Un protocole et les attaques qui s'y rapportent s'exécutent dans le *monde réel*. On définit alors un *monde idéal*, dans lequel tout se déroulerait comme prévu : il décrit les spécifications du protocole. L'idée est alors de dire que l'on est en présence d'un bon protocole s'il se comporte de manière indistinguable de son homologue idéal.

Plus formellement, on définit dans le monde idéal une entité que l'on ne peut corrompre, appelée la *fonctionnalité idéale* et généralement notée $\mathcal{F}$.Les joueurs peuvent lui envoyer leurs entrées en privé, et recevoir la sortie correspondante ; il n'y a aucune communication entre les joueurs. $\mathcal{F}$ est alors programmée pour se comporter de manière totalement correcte, ne pas révéler d'autres informations que celles prévues et ne pas pouvoir se faire corrompre par un adversaire : on dit souvent qu'elle est *trivialement sûre et correcte*.

Dans le cas d'un calcul partagé, l'évaluation sûre d'une fonction peut être spécifiée par une autorité qui reçoit toutes les entrées, calcule la sortie, et redonne ces valeurs aux joueurs. L'avantage de l'UC est de permettre de traiter aussi des tâches *réactives*, en remplaçant ce « participant de confiance » par la fonctionnalité idéale, qui est une entité algorithmique plus générale.



On cherche ici à garantir la robustesse des protocoles vis-à-vis de l'environnement d'exécution, c'est-à-dire que l'on considère le protocole de manière indépendante de son environnement, et que l'on cherche à prouver que la composition avec d'autres protocoles est sûre.

On définit donc les tâches pour une instance *unique*, même quand l'utilisation sera effectuée dans le cadre d'instances multiples et concurrentes. Cela permet d'avoir des formulations simples des fonctionnalités idéales et c'est le théorème de composition universelle qui garantira la sécurité dans un cadre multi-instances.

Des exemples de tâches basiques sont ainsi formalisés en réalisant la fonctionnalité adéquate ([Can01]) : authentification des messages (*message authentication*, $\mathcal{F}_{\text{AUTH}}$), transmission sûre des messages (*secure message transmission*, $\mathcal{F}_{\text{SMT}}$), sessions de communication sûre et échange de clefs (*secure communication sessions* et *key exchange*, $\mathcal{F}_{\text{SCS}}$), chiffrement à clef publique (*public-key encryption*), signature (*digital signatures*), primitives bi-partites (*coin-tossing*, *commitment*, *zero-knowledge*, *oblivious-transfer*).

L'objectif d'un protocole π, dans le monde réel et en présence d'un adversaire, est alors de créer une situation équivalente à celle obtenue par $\mathcal{F}$. Cela signifie que la communication entre les joueurs ne donne pas plus d'information que celle donnée par la description de la fonction elle-même, et le résultat de cette dernière (cela peut suffire à empêcher toute forme de sécurité, comme dans le cas de la fonction constante).

5.1.3 Adversaire et environnement

Dans le cadre UC comme dans tout schéma cryptographique, on est en présence d'un adversaire qui observe les joueurs et peut éventuellement intervenir. Mais le cadre de la composition universelle dispose en outre d'une entité appelée l'*environnement*. Comme son nom l'indique, il modélise « tout ce qui est extérieur au protocole en cours d'exécution ». C'est lui qui donne les entrées aux joueurs et qui récupère la sortie à la fin de l'exécution. Cela modélise le fait qu'en général, les entrées ne sont pas aléatoires et choisies au hasard. Elles dépendent

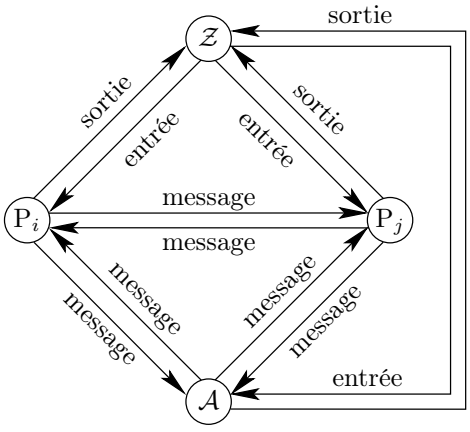
au contraire de l'environnement (au sens français du terme) dans lequel évolue le protocole. Par exemple, l'écriture d'un mail est le plus souvent une réponse, provoquée par la réception d'un mail.

On a dit plus haut qu'un protocole, pour être sûr, devait être capable d'imiter parfaitement son homologue idéal : c'est l'environnement qui sera chargé d'en juger (la sortie finale n'est constituée que d'un seul bit, disant s'il pense avoir interagi avec le protocole réel ou idéal). Informellement, on considère d'un côté le monde réel, avec l'adversaire, les joueurs et le protocole, et de l'autre le monde idéal avec le protocole idéal, des joueurs virtuels et un simulateur de l'adversaire, aussi appelé adversaire idéal. On dit alors dans ce cadre que le protocole π *réalise de manière sûre* une tâche si, pour tout adversaire polynomial $\mathcal{A}$,il existe un adversaire idéal polynomial $\mathcal{S}$tel qu'aucun environnement polynomial $\mathcal{Z}$ne puisse savoir avec avantage non-négligeable s'il est en train d'interagir avec π et $\mathcal{A}$ ou avec le protocole idéal et $\mathcal{S}$. Si un protocole π réalise de manière sûre une fonction f par rapport à ce type d'« environnement interactif », on dit que π *UC-réalise* f.

L'adversaire pouvant interagir à tout moment avec l'environnement, on peut dire que ce dernier a un rôle de *distingueur interactif*. Cela impose comme contrainte au simulateur qu'il n'a pas le droit de « rembobiner » l'environnement, ce qui est pourtant une astuce classique des preuves de sécurité « traditionnelles ».

L'environnement peut mesurer l'influence de l'adversaire $\mathcal{A}$ sur les sorties des joueurs, la fuite d'information, ainsi que le délai de livraison des messages. Les interactions possibles sont schématisées sur le schéma ci-après. L'adversaire a accès à la communication entre les joueurs, mais pas aux entrées et sorties[1]. À l'inverse, l'environnement a accès aux entrées et sorties des joueurs, mais pas à leur communication, ni aux entrées et sorties des sous-routines qu'ils peuvent invoquer. Enfin, il est au courant de l'identité des joueurs corrompus par l'adversaire.

La différence principale avec le cadre habituel est la manière dont l'environnement interagit avec l'adversaire. Dans le cadre UC, ils peuvent interagir librement et à tout moment ; de l'information entre l'instance considérée du protocole et le reste du réseau est échangée n'importe quand.



Adversaires adaptatifs et corruption forte. On a vu dans le chapitre 2 que, selon le cadre de sécurité, le caractère privé de la clef de session est modélisé soit via la « sécurité sémantique » [BR94] ou l'indistinguabilité entre le protocole et son homologue idéal [CK01]. Dans tous les cas, la fuite du secret à long terme (ici le mot de passe) est modélisé par les requêtes de « corruption ». Malheureusement, le secret à long terme peut ne pas être la seule information révélée pendant une corruption. La fuite de secrets éphémères pouvant aussi causer des dégâts importants dans certains contextes [Kra05, KM06], il faut donc considérer les « corruptions fortes » [Sho99]. Cependant, il ne semble pas très réaliste de permettre au concepteur d'un protocole de décider quelle information est révélée par une telle requête.

[1]Quand on évoque l'adversaire, on parle toujours de son comportement par rapport aux seuls joueurs honnêtes (et pas par rapport aux joueurs malhonnêtes ou corrompus, sur lesquels il a tout contrôle).

Le cadre UC propose au contraire une approche différente : quand une corruption forte a lieu, l'adversaire accède à la totalité de la mémoire interne du joueur corrompu (il apprend son état interne) puis il prend son contrôle pour terminer l'exécution à sa place. Cela semble être un scénario beaucoup plus réaliste. Dans une exécution réelle du protocole, on modélise ceci en donnant à l'adversaire le mot de passe et l'état interne du joueur corrompu. De plus, l'adversaire peut à partir de cet instant modifier arbitrairement la stratégie du joueur. Dans une exécution idéale du protocole, le simulateur $\mathcal{S}$ obtient le mot de passe du joueur et doit simuler son état interne, de façon à ce que tout reste cohérent avec ce que l'environnement avait donné au joueur au départ et tout ce qui s'est déroulé depuis.

5.1.4 Principe de la sécurité

Fonctionnalités idéales. La sécurité dans le cadre UC est donc définie en termes d'une fonctionnalité idéale $\mathcal{F}$, qui est une autorité de confiance interagissant avec un ensemble de joueurs pour calculer une fonction donnée f. En particulier, les joueurs donnent leurs paramètres en entrée à $\mathcal{F}$,qui applique f sur les valeurs reçues en entrée et retourne à chaque joueur la valeur adéquate en sortie. Ainsi, dans ce cadre idéal, la sécurité est garantie de façon inhérente, puisqu'un adversaire contrôlant certains joueurs ne peut qu'apprendre (et éventuellement modifier) les données des joueurs corrompus.

Afin de prouver qu'un protocole candidat π réalise la fonctionnalité idéale, on considère un environnement $\mathcal{Z}$, qui est autorisé à donner des valeurs en entrée à tous les participants, et dont l'objectif est de distinguer le cas dans lequel il reçoit les valeurs en sortie provenant d'une exécution réelle du protocole (qui met en jeu tous les joueurs et un adversaire $\mathcal{A}$, qui contrôle certains joueurs et la communication entre tous) du cas dans lequel il reçoit des sorties obtenues par une exécution idéale du protocole (mettant en jeu uniquement des joueurs factices interagissant avec $\mathcal{F}$ et un adversaire idéal $\mathcal{S}$ qui interagit lui aussi avec $\mathcal{F}$).

On dit alors qu'un protocole π réalise la fonctionnalité $\mathcal{F}$ si pour tout adversaire polynomial $\mathcal{A}$, il existe un simulateur polynomial $\mathcal{S}$ tel qu'aucun environnement polynomial $\mathcal{Z}$ ne puisse distinguer une exécution réelle d'une exécution idéale avec un avantage non négligeable. En particulier, le théorème de composition universelle assure que π continue à se comporter comme la fonctionnalité idéale même s'il est exécuté dans un environnement arbitraire.

Identifiants de sessions et de joueurs. Dans le cadre UC, il peut y avoir plusieurs copies de la fonctionnalité idéale tournant en parallèle. Chacune de ces copies est alors supposée posséder un identifiant de session unique (SID). Chaque fois qu'un message est envoyé à une copie spécifique de $\mathcal{F}$, ce message doit contenir le SID de la copie à laquelle il est destiné. Sur le modèle de [CHK+05], on suppose que chaque protocole réalisant $\mathcal{F}$ attend de recevoir des entrées qui contiennent déjà le SID approprié.

Le cadre UC à états joints. Le théorème original de composition universelle permet d'analyser la sécurité d'un système vu comme une unité simple, mais il ne donne aucune information dans le cas où différents protocoles partagent un certain nombre d'états et d'aléa (comme une clef secrète, par exemple). Ainsi, pour les applications en jeu, le théorème UC de base ne peut pas être utilisé tel quel, étant donné que des sessions différentes d'un même protocole partagent des données (telles qu'un oracle aléatoire, un chiffrement idéal ou une CRS).

Pour résoudre ce problème, Canetti et Rabin ont introduit en 2003 la notion de composabilité universelle avec états joints [CR03]. De manière informelle, il s'agit d'une nouvelle opération de composition qui permet à différents protocoles d'avoir des états en commun, tout en préservant la sécurité. Très informellement, ceci est fait en définissant l'*extension multi-session* de $\mathcal{F}$, qui tourne en parallèle des extensions multiples de $\mathcal{F}$. Chaque copie de $\mathcal{F}$ possède un identifiant de sous-session (SSID), ce qui signifie que, si $\widehat{\mathcal{F}}$reçoit un message m avec le SSID ssid, alors elle envoie m à la copie de $\mathcal{F}$ qui a le SSID ssid. Si cette copie n'existe pas, $\widehat{\mathcal{F}}$ en invoque une nouvelle à la volée. En résumé, quand $\widehat{\mathcal{F}}$ est exécutée, le protocole doit spécifier à la fois son SID (l'identifiant de session, comme avec n'importe quelle fonctionnalité idéale) ainsi que le SSID souhaité[2].

[2]Dans les protocoles de ce mémoire, nous nous contenterons souvent du SSID pour alléger les notations.

5.2 Le modèle calculatoire sous-jacent

On veut définir un modèle, à la fois réalisable techniquement et assez général pour toutes les situations réalistes. Le modèle choisi est une extension du modèle des *machines de Turing interactives* (ITM) : les programmes des différents participants sont représentées par des MT, avec des rubans partagés (où toutes peuvent lire et écrire).

L'intérêt des MT est de reproduire fidèlement les interactions dans un réseau, et de représenter au mieux l'effet des communications et la complexité des calculs locaux ; elles constituent une base naturelle pour considérer des ressources bornées, des calculs probabilistes, ainsi que la possibilité d'ordonnancement par un adversaire. En particulier, elles autorisent le traitement asymptotique de la sécurité comme fonction d'un paramètre de sécurité. Un reproche serait qu'elles ont un faible niveau d'abstraction des programmes et protocoles.

Les ITM vont aussi servir à modéliser des entités plus abstraites, comme les adversaires, les environnements et les fonctionnalités idéales.

5.2.1 Machines de Turing interactives (ITM)

Un ruban de machine de Turing est dit *externally writable* (EW) si d'autres ITM peuvent écrire dessus. On suppose de telles machines *write-once* (la tête de lecture a une direction unique). Une ITM est une machine de Turing (dont le code peut inclure des instructions à écrire sur le ruban d'une autre ITM) possédant les rubans suivants :

- ruban d'identité EW (*EW identity tape*) : son contenu est l'identité de M, constituée de la *session identity* (SID) et de la *party identity* (PID)[3] ;
- ruban du paramètre de sécurité EW (*EW security parameter tape*) ;
- ruban d'entrée EW (*EW input tape*) ;
- ruban de communication entrante EW (*EW incoming communication tape*) : son contenu modélise l'information provenant du réseau ; c'est une suite de *messages* avec deux champs : l'émetteur (*sender* : identité d'une ITM) et le contenu (*contents*, arbitraire) ;
- ruban de sortie des sous-routines (*EW subroutine output tape*) : son contenu modélise les sorties des sous-routines de M ; c'est une séquence de *subroutine outputs* ayant deux champs : *subroutine id* (l'identité et le code d'une ITM) et *contents* (contenu, arbitraire)) ;
- ruban de sortie (*output tape*) ;
- ruban d'aléa (*random tape*) ;
- ruban d'activation (*read and write one-bit activation tape*) ;
- ruban de travail (*read and write work tape*).

5.2.2 Systèmes d'ITM

Un *système d'ITM* $\mathrm{S} = (\mathrm{I}, \mathrm{C})$ est défini par une *ITM initiale* I et une *fonction de contrôle*

$$\mathrm{C} : \{0,1\}^* \to \{\text{allow, disallow}\}$$

qui détermine l'effet des instructions EW des ITM dans le système : quand une ITM veut écrire sur le ruban d'une autre, c'est elle qui l'autorise ou non, précise les conditions, etc.

Configurations et instances (ITI). Une *configuration* d'une ITM M est constituée de la description du code, de l'état de contrôle, du contenu de tous les rubans et de la position des têtes. Une configuration est *active* si le ruban d'activation est à 1, *inactive* sinon. Une *instance* (ITI) $\mu = (\mathrm{M}, id)$ d'une ITM est constituée du code (fonction de transition) M et d'une *chaîne identité* $\mathsf{id} \in \{0,1\}^*$. Intuitivement, une ITI représente l'instanciation d'une ITM,

[3] Voir la section 5.2.3 à ce propos.

c'est-à-dire un processus qui fait tourner le code de l'ITM sur une entrée spécifique. Le même programme (ITM) peut avoir de multiples instances (ITI) dans l'exécution d'un système.

Activation et exécution. Une *activation* d'une ITI μ est une séquence de configurations correspondant à un calcul à partir d'une configuration active de μ, jusqu'à ce qu'une configuration inactive soit atteinte. μ attend alors la prochaine activation, sauf si un état spécial *halt* est atteint.

Étant donné un paramètre de sécurité k et une entrée x, une *exécution* du système $\mathrm{S} = (\mathrm{I}, \mathrm{C})$ consiste en une suite d'activations des ITI. La première activation démarre avec la configuration suivante : le code de I, l'entrée x écrite sur le ruban d'entrée, 1^k écrit sur le ruban du paramètre de sécurité, une chaîne aléatoire r suffisamment longue, et l'identité 0. L'*ITI initiale* est donc $(\mathrm{I}, 0)$. L'exécution termine quand l'ITI initiale s'arrête (*ie* quand elle atteint l'état *halt*) et la sortie de l'exécution est le contenu de son ruban de sortie dans cette configuration.

Instructions EW et invocation. Une instruction EW spécifie les paramètres suivants : les codes et identités de la présente ITI (μ) et de l'ITI visée (μ'), le ruban de l'ITI visée sur lequel écrire (*input tape*, *incoming communication tape* ou *subroutine output tape*) et les données à écrire.

L'effet d'une instruction EW dépend de l'identité, du ruban et éventuellement du code de l'ITI destinataire : il est décrit explicitement dans [Can01]. Si l'ITI destinataire n'existait pas dans le système, on dit que μ *invoque* μ'.

L'ordre d'activations est le suivant : à chaque activation, une ITI peut exécuter au plus une instruction EW ; l'ITI destinataire est alors la prochaine ITI activée. Si aucune telle opération n'a eu lieu, l'ITI initiale est la prochaine activée. Par conséquent, à chaque instant, une seule ITI a vu son état local changer depuis sa précédente activation : cela permet de modéliser un calcul distribué par une séquence d'événements locaux.

5.2.3 Problèmes soulevés par la modélisation

La modélisation par systèmes d'ITM introduit la question suivante : de quelle manière sont déterminés les programmes et identités des nouvelles instances ? On choisit de laisser l'« instance d'invocation » déterminer ces valeurs dynamiquement. Pour définir la manière dont cette instance spécifie l'ITI sur le ruban de laquelle elle veut écrire (instruction EW), on suppose que chaque ITI a une chaîne de caractères *identité* déterminée à l'invocation et non modifiable. Ces identités sont globalement uniques et évitent les ambiguïtés lors de la détermination des ITI destinataires.

La nature dynamique d'un système d'ITM soulève une autre question : comment isoler une seule *instance de protocole* lors de l'exécution d'un système ? Informellement, un ensemble d'ITI dans une exécution d'un système est une instance de protocole si celles qui les invoquent en décident ainsi. Pour formaliser cela, on suppose que l'identité d'une ITI est en fait constituée de deux champs séparés : *session ID* (SID) et *party ID* (PID). Un ensemble d'ITI est alors une instance de protocole si elles ont le même programme et le même SID. Les PID servent à séparer les ITI dans une même instance. Cette définition autorise le nombre de participants à être non borné ou inconnu a priori, et à évoluer dynamiquement.

Cette modélisation offre deux méthodes de communication entre les ITI :

- via les *rubans de communication* : *untrusted communication*, l'identité et le programme de l'ITI qui écrit ne sont pas connus du récepteur ;
- via les rubans *input* et *subroutine output* : le récepteur a confiance en l'identité et en le programme de l'ITI en train d'écrire.

5.2.4 Protocoles, sous-routines et fonctionnalités idéales

Sous-routines. On dit que l'ITI μ' est une *sous-routine* de μ si μ a donné une entrée à μ' ou bien si μ' a donné une sortie à μ. C'est une *subsidiary* (filiale) de μ si c'est une sous-routine de μ ou d'un autre *subsidiary* de μ.

États et transcriptions. Un *état* d'un système d'ITM représente une description complète d'un certain instant dans l'exécution du système (suite de toutes les configurations de toutes les activations jusqu'à ce point).

Une *transcription* d'une exécution d'un système est l'état final de l'exécution (l'état dans lequel la dernière configuration est une configuration terminale de l'ITI initiale).

Systèmes étendus. Dans un *système étendu*, la fonction de contrôle peut aussi modifier les requêtes EW, c'est-à-dire pas seulement les autoriser ou les refuser, mais aussi changer l'instruction entière à exécuter. On utilise ceci uniquement pour modifier le code des ITI nouvellement générées.

Sortie des exécutions. On note $\text{OUT}_{\mathcal{Z}}(k, x)$ la variable aléatoire décrivant la sortie de l'exécution du système (I, C) d'ITM où l'entrée de I est x et le paramètre de sécurité k (la probabilité est prise sur le choix des ITM dans le système). On note

$$\text{OUT}_{\mathcal{Z}} = \{\text{OUT}_{\mathcal{Z}}(k, x) \mid k \in \mathbb{N}, x \in \{0,1\}^*\}$$

Protocoles. Un *protocole* est défini comme une seule ITM représentant le code que doit exécuter chaque participant. Étant donné l'état d'un système d'ITM, l'*instance d'un protocole multi-parties* π de SID *sid* est l'ensemble des ITI dans le système (appelées les *parties*, *joueurs* ou *participants*) dont le code est π et la SID *sid*. On suppose que π ignore tous les messages entrants dont l'expéditeur a une SID différente. Une sous-partie (*sub-party*) est une sous-routine d'un joueur ou d'une sous-partie. L'*instance étendue* de π inclut tous les joueurs et sous-parties.

On note π_i ou P_i le i-ième participant exécutant le protocole π (l'ordre est arbitraire, on prend par exemple celui d'invocation ; le i n'a pas besoin d'être connu par l'ITI). On note id_{π_i}, pid_{π_i}, sid_{π_i} l'identité, le PID et le SID de l'ITI. On note $\pi_{(\mathsf{id})}$ ou $\text{P}_{(\mathsf{id})}$ pour désigner l'ITI (π, id).

Fonctionnalité idéale. Une *fonctionnalité idéale* représente la fonctionnalité espérée d'une certaine tâche : cela inclut la *correction* (relations entrée-sortie des joueurs non corrompus) et le *secret* (fuite autorisée d'information envers l'adversaire). C'est une entité algorithmique générale modélisée par une ITM ; elle reçoit de manière répétée des entrées de la part des participants et leur donne en retour les valeurs de sorties appropriées, en garantissant entre temps le maintien de l'état local. Ce cadre garantit que les sorties des participants dans le processus idéal ont les propriétés requises vis-à-vis des entrées, et ceci même si les nouvelles entrées sont choisies de manière adaptative par rapport aux sorties précédentes.

$\mathcal{F}$ a des instructions pour savoir générer les sorties des participants en se basant sur leur entrée. Elle peut aussi échanger des messages avec $\mathcal{A}$: cela permet de prendre en compte l'*influence autorisée* de l'adversaire sur les sorties des participants, et la *fuite autorisée* d'information sur les entrées et sorties vers l'adversaire, ainsi que le *délai autorisé* de livraison des sorties. Techniquement,

- une fonctionnalité idéale $\mathcal{F}$ est une ITM ;
- un certain nombre d'ITI peuvent écrire sur son ruban d'entrée ;
- elle peut écrire sur les rubans de sortie des sous-routines de nombreuses ITI (ces deux dernières propriétés représentent le fait qu'une fonctionnalité idéale se comporte comme une sous-routine pour un certain nombre d'ITI) ;
- son PID est $\perp$ (cela distingue les fonctionnalités idéales des autres ITI), et elle s'attend à ce que toutes les entrées soient écrites par des ITI de SID égale à la SID locale de $\mathcal{F}$;
- son ruban de communication est utilisé pour communiquer avec l'adversaire.

Protocole idéal. Le *protocole idéal* $\text{IDEAL}_{\mathcal{F}}$ pour une fonctionnalité idéale $\mathcal{F}$ est défini de la manière suivante : à la réception d'une entrée v, le protocole demande au participant émetteur de cette entrée de transmettre v comme entrée à l'instance de $\mathcal{F}$ dont le SID est le SID local ; la sortie provenant de $\mathcal{F}$ est alors copiée à la sortie locale. Cela a l'avantage de permettre à l'instance de π de spécifier comme elle le souhaite les SID et PID de l'instance de $\text{IDEAL}_{\mathcal{F}}$. Plus précisément,

- lorsqu'un joueur avec l'identité $(\mathsf{sid}, \mathsf{pid})$ est activé par l'entrée v, il écrit v sur le ruban d'entrée de $\mathcal{F}_{(\mathsf{sid},\perp)}$, ie l'instance de $\mathcal{F}$ dont le SID est sid ;
- si un joueur reçoit v sur son ruban de sortie des sous-routines, il écrit cette valeur sur le ruban correspondant de $\mathcal{Z}$;
- les messages délivrés par $\mathcal{A}$ (corruption incluse) sont ignorés (le but est d'envoyer ces messages de corruption directement à la fonctionnalité idéale, qui décidera de leur effet) ;

Un joueur d'un protocole idéal pour $\mathcal{F}$ est souvent appelée *joueur virtuel*, ou *fictif* (*dummy player* en anglais, car le PID de la fonctionnalité est $\perp$).

Protocole $\mathcal{F}$-hybride. Un *protocole $\mathcal{F}$-hybride* π est un protocole qui inclut des appels au protocole idéal pour $\mathcal{F}$ (sous-routine $\text{IDEAL}_{\mathcal{F}}$). (Plus généralement, on définit un protocole $\mathcal{F}_1, \ldots, \mathcal{F}_n$-hybride.)

5.2.5 ITM et systèmes d'ITM en temps polynomial probabiliste

Dans cette modélisation dynamique, la définition d'un calcul en temps polynomial doit assurer qu'aucun composant (ou le système entier) ne doit surconsommer de ressources, de même qu'on ne doit pas restreindre les composants de manière artificielle (ce qui affecterait le sens de la notion de sécurité). On veut donc borner le temps de calcul total d'un système d'ITM. En particulier, à la fois le temps total de chaque ITI et le nombre total d'instances doivent être bornés.

On dit qu'une ITM M (un programme) est localement en *temps polynomial probabiliste* (ce que l'on note PPT) si, à tout moment de l'exécution d'une ITI μ de code M, le temps total jusqu'à cet instant est borné par un polynôme en le paramètre de sécurité plus la longueur totale en entrée (*ie* le nombre total de bits écrits sur le ruban d'entrée), et si, de plus, le nombre de bits écrits sur les rubans d'entrée des autres ITI, plus le nombre d'autres ITI invoquées par μ, est inférieur à la longueur d'entrée de μ jusqu'à présent.

Plus précisément, le nombre de pas de calcul de M est à tout instant borné par un polynôme en n, où n est calculé de la manière suivante : au paramètre de sécurité k, ajouter le nombre total de bits écrits jusqu'à maintenant sur le ruban d'entrée de M, et soustraire le nombre de bits écrits par M jusqu'à maintenant sur des rubans d'entrée d'ITI. Enfin, soustraire k fois le nombre d'ITI sur les rubans desquelles M a écrit.

Cette condition doit être vraie pour tous les *subsidiaries* de M, et leurs polynômes doivent être plus petits que celui de M. On dit alors que M est P-*bornée*, P étant le polynôme concerné.

La notion de PPT ITM doit être *reconnaissable efficacement*, c'est-à-dire qu'on doit savoir décider si une ITM est PPT ou non ; des règles sont définies pour cela dans [Can01].

5.2.6 Définir la sécurité des protocoles

Cette section a pour but de définir ce que signifie pour un protocole de *réaliser de manière sûre* une fonctionnalité idéale donnée.

Le cadre d'exécution de protocoles. Ce cadre brut décrit un réseau asynchrone, non authentifié et non sûr, dans lequel la communication peut être observée et contrôlée par un adversaire. De plus, personne n'a d'information *a priori* sur les autres participants (identités ou clefs publiques). Il est paramétré par trois ITM :

- π, le protocole à exécuter ;
- $\mathcal{A}$, un adversaire ;
- $\mathcal{Z}$, un environnement.

Le cadre d'exécution de π est le système étendu de PPT ITM $(\mathcal{Z}, \mathcal{C}^{\pi,\mathcal{A}}_{\text{EXEC}})$ (la fonction de contrôle $\mathcal{C}^{\pi,\mathcal{A}}_{\text{EXEC}}$ sera définie dans la suite). On note $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(k,z)$ la variable aléatoire $\text{OUT}_{\mathcal{Z},\mathcal{C}^{\pi,\mathcal{A}}_{\text{EXEC}}}(k,z)$ et $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ l'ensemble $\{\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(k,z) \mid k \in \mathbb{N}, z \in \{0,1\}^*\}$.

On note $\text{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{Z}}(k,z,r)$ la variable aléatoire $\text{EXEC}_{\text{IDEAL}_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(k,z,r)$, et

$$\text{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{Z}} = \{\text{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{Z}}(k,z) \mid k \in \mathbb{N}, z \in \{0,1\}^*\}$$

Comportement de $\mathcal{Z}$. Initialement, $\mathcal{Z}$ (l'ITI initiale) reçoit en entrée l'état initial de l'environnement, autrement dit toutes les entrées extérieures ainsi que les entrées locales de tous les joueurs. La fonction de contrôle impose à $\mathcal{A}$ d'être la première ITI invoquée. $\mathcal{Z}$ peut ensuite invoquer comme sous-routines un nombre arbitraire d'ITI, leur donner des entrées, en récupérer des sorties, à condition qu'elles aient toutes le même SID (imposé par $\mathcal{Z}$) ; leur code est alors fixé à π (ce sont donc des joueurs d'une instance unique du protocole π). L'environnement ne peut interagir avec aucune autre ITI (en particulier des sous-routines des joueurs de cette instance de π). À l'issue de l'exécution, $\mathcal{Z}$ renvoie en sortie un unique bit[4].

Comportement des ITI. La fonction de contrôle autorise les parties et sous-parties de l'instance de π à invoquer des ITI et leur donner des entrées et des sorties (sauf l'adversaire). Elles peuvent aussi écrire des messages sur le ruban de communication entrante de $\mathcal{A}$ (mais pas des autres ITI) : ces derniers peuvent spécifier une identité d'une ITI comme destinataire final. Elles peuvent enfin écrire des résultats sur le ruban de sortie des sous-routines de $\mathcal{Z}$.

Comportement de $\mathcal{A}$.L'adversaire $\mathcal{A}$ peut *délivrer* des messages (ne correspondant pas forcément à ceux réellement envoyés par les joueurs) à n'importe quelle ITI du système, mais pas invoquer de sous-routines[5]. $\mathcal{A}$ peut aussi *corrompre* des joueurs ou sous-parties de l'instance de π, ce qui est modélisé par un message particulier appelé `corrupt`. Il doit pour cela avoir reçu l'ordre de le faire par $\mathcal{Z}$ (ceci afin que l'environnement soit au courant des joueurs corrompus), mais peut avoir ajouté des paramètres. La réponse d'un joueur à ce message (son état interne) est définie dans le protocole plutôt que dans le cadre. Dans le cas byzantin par exemple, l'ITI corrompue envoie à l'adversaire tout son état local courant et suit les instructions de l'adversaire dans toutes ses futures activations en ce qui concerne les instructions EW.

Remarque. En accord avec l'intuition que $\mathcal{Z}$ représente l'environnement (qui procure les entrées et récupère la sortie) et $\mathcal{A}$ représente l'adversaire (qui contrôle les liens de communication sans avoir accès aux entrées et sorties locales) :

- $\mathcal{Z}$ n'a accès qu'aux entrées et sorties des participants, pas à leur communication, ni aux entrées et sorties des sous-routines ;
- $\mathcal{A}$ n'a accès qu'à la communication entre les participants et pas à leurs entrées et sorties.

Ces interactions sont résumées sur le dessin page 60.

5.2.7 Pour résumer

On utilise la modélisation par machines de Turing, et plus précisément par systèmes d'ITM, définis par une ITM initiale et une fonction de contrôle (qui contrôle les communications entre ITM). Le protocole à exécuter, l'environnement et l'adversaire sont modélisés par des ITM. L'environnement est l'ITI initiale et la fonction de contrôle impose que l'adversaire soit la première ITI invoquée.

Une instance (ITI) $\mu = (\text{M}, id)$ d'une ITM est constituée du code M et d'une identité $(\mathsf{sid}, \mathsf{pid})$. Intuitivement, une ITI représente l'instanciation d'une ITM, c'est-à-dire un processus qui fait tourner le code de l'ITM sur une entrée spécifique. Le même programme (ITM) peut avoir de multiples instances (ITI) dans l'exécution d'un système.

[4]On demande que la sortie ne fasse qu'un seul bit pour simplifier les notations ; ce n'est en rien une limitation de ce cadre de sécurité.

[5]Rappelons que l'on ne parle que de ses interactions avec les joueurs honnêtes.

Un protocole π est défini par une ITM de SID sid et code c représentant le code que doit exécuter chaque participant. L'instance d'un protocole π est l'ensemble des ITI, les participants à ce protocole, qui ont tous le code c et le SID sid. π ignore les messages entrants de participants de SID différents. Cela autorise le nombre de participants à être non borné ou à évoluer dynamiquement.

Une seule ITI est active à la fois ; elle peut ensuite envoyer un message à une autre ITI pour l'activer à son tour (elle devient alors inactive ou s'arrête si l'état *halt* est atteint). Si cette autre ITI n'existait pas dans le système, on dit qu'elle l'*invoque*.

5.2.8 Différents modèles de communication

Il y a beaucoup de modèles de communication et d'adversaires pour définir et analyser des protocoles cryptographiques. Souvent, les formulations précises et utilisables des définitions de sécurité leur sont spécifiques. Il existe ainsi de multiples variantes de cadres de définition, un par modèle.

L'approche est ici prise dans le sens contraire : le cadre reste simple et inchangé et les différents modèles de communication sont obtenus en laissant les participants accéder à une fonctionnalité idéale adéquate. Un avantage est que le cadre de base et le théorème de composition ne doivent pas être réénoncés et reprouvés à chaque nouveau modèle de communication.

Le modèle de communication fourni par le cadre UC de base est basique et à peine utilisable : l'adversaire a accès à tous les messages et peut délivrer des messages arbitraires à tous les participants. Aucune garantie n'est donnée concernant le temps de livraison, l'authenticité ou le secret des messages délivrés. Ce sont les fonctionnalités idéales définies par dessus ce modèle qui permettront de le raffiner et obtenir des modèles de communication plus abstraits (communication authentifiée, sûre, synchrone). On peut aussi définir des fonctionnalités idéales qui représentent des propriétés de sécurité pas nécessairement préservées par la composition concurrente : cela permet d'analyser dans le même cadre des protocoles où certaines composantes peuvent être composées de manière concurrente et d'autres non.

5.3 Émulation de protocoles

5.3.1 Indistinguabilité

Un *ensemble de distributions* $\mathcal{X} = \{\mathrm{X}(k,a) \mid k \in \mathbb{N}, a \in \{0,1\}^*\}$ est un ensemble infini de variables aléatoires $\mathrm{X}(k,a)$ associées à chaque $k \in \mathbb{N}$ et $a \in \{0,1\}^*$.

Deux ensembles binaires de distributions $\mathcal{X}$ et $\mathcal{Y}$ sont dits indistinguables (ce qui s'écrit $\mathcal{X} \approx \mathcal{Y}$) si pour tous $c, d \in \mathbb{N}$, il existe $k_0 \in \mathbb{N}$ tel que pour tout $k > k_0$ et tout $a \in \bigcup_{l \leqslant k^d} \{0,1\}^l$,

$$|\Pr[\mathrm{X}(k,a) = 1] - \Pr[\mathrm{Y}(k,a) = 1]| < k^{-c}$$

Rappelons que l'on note $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(k,z)$ la variable aléatoire décrivant la sortie de l'exécution du protocole π en présence de l'adversaire $\mathcal{A}$ et de l'environnement $\mathcal{Z}$ (elle vaut 0 ou 1).

5.3.2 La définition de base

Informellement, un protocole ρ UC-émule un protocole ϕ si les interactions avec ρ et n'importe quel adversaire ou avec ϕ et un simulateur sont indistinguables pour tout environnement.

Définition. *Si ϕ et ρ sont des protocoles PPT (polynomiaux), on dit que ρ UC-émule ϕ si, pour tout adversaire PPT $\mathcal{A}$, il existe un adversaire simulé PPT $\mathcal{S}$ tel que, pour tout environnement PPT $\mathcal{Z}$:*

$$\mathrm{EXEC}_{\phi,\mathcal{S},\mathcal{Z}} \approx \mathrm{EXEC}_{\rho,\mathcal{A},\mathcal{Z}}$$

5.3.3 D'autres formulations

Versions quantitatives. Pour définir quantitativement l'émulation de protocoles PPT, Canetti définit dans [Can01] l'*emulation slack* comme étant la probabilité de distinction par l'environnement des interactions avec π ou ϕ, et le *simulation overhead*, qui est la différence entre la complexité de l'adversaire $\mathcal{A}$ donné et celle de l'adversaire construit $\mathcal{S}$.

L'idée est que si π UC-émule ϕ, alors le temps de l'adversaire croît d'un facteur additif polynomial ne dépendant que de la communication engendrée par $\mathcal{A}$ plutôt que du « temps de calcul interne » de $\mathcal{A}$.

En particulier, il existe un polynôme g fixé tel que, pour toute valeur du paramètre de sécurité k, toute attaque réussie contre ϕ peut devenir une attaque réussie contre π utilisant $g(k)$ fois plus de ressources.

Remarquons que l'émulation est transitive. Mais si le nombre de protocoles n'est pas borné par une constante, alors la complexité de l'adversaire n'est plus bornée par un polynôme.

Définitions équivalentes. Faisons deux remarques préliminaires (voir [Can01]). D'une part, si les sorties ne sont plus binaires, il faut demander en plus que les deux ensembles de sortie $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ et $\text{EXEC}_{\phi,\mathcal{S},\mathcal{Z}}$ soient *calculatoirement indistinguables*. Cette définition est équivalente à celle précédemment donnée. D'autre part, étant donné que l'on autorise l'environnement à recevoir des entrées extérieures arbitraires, il suffit de considérer des environnements déterministes pour obtenir une définition équivalente.

Ensuite, un point important dans la définition de l'émulation est que l'ordre des quantificateurs est sans importance.

Tout d'abord, au lieu d'autoriser un simulateur différent pour chaque adversaire, on peut laisser le simulateur avoir accès à $\mathcal{A}$ comme une *boîte noire*, et demander ensuite que le code de $\mathcal{S}$ soit le même pour tout $\mathcal{A}$. Cette définition est équivalente à la précédente.

On peut aussi déplacer d'une autre manière les quantificateurs de la définition pour associer à chaque $\mathcal{A}$ et $\mathcal{Z}$ un simulateur $\mathcal{S}$. Cette variante est appelée *sécurité par rapport aux simulateurs spécialisés* et est équivalente aux précédentes. L'élément-clef de cette équivalence est l'existence d'un *environnement universel* par rapport à tous les environnements PPT. Ceci est rendu possible par la dépendance du temps de calcul envers non seulement le paramètre de sécurité, mais aussi la taille de l'entrée.

La définition la plus utilisée. On peut simplifier la définition en demandant, au lieu de quantifier sur tous les adversaires possibles, que l'adversaire $\mathcal{S}$ du protocole idéal soit capable de simuler, pour tout environnement $\mathcal{Z}$, le comportement d'un adversaire simple et spécifique, l'*adversaire nul*. C'est « le plus dur à simuler » : il ne délivre aux joueurs que des messages générés par l'environnement, et délivre à l'environnement tous les messages générés par les joueurs (voir ci-dessous).

On pourrait se passer de l'adversaire nul et autoriser l'interaction directe entre les parties et l'environnement : il s'agit de l'*émulation de protocole directement avec l'environnement* (aussi appelée simulation forte). Mais si cela implique l'émulation avec l'adversaire nul, la réciproque n'est pas vraie (cette notion n'est même pas réflexive). En effet, dans le cas de l'émulation directe, l'environnement n'a pas d'adversaire, donc ne s'attend pas à interagir avec un simulateur, qui ne peut donc rien faire.

Cette distinction est une manifestation du fait que la notion des PPT ITM décrite ici ne satisfait pas la *propriété de forward*, spécifiant qu'il soit possible d'insérer une *ITI nulle* (qui ne fait que forwarder les entrées et les messages) entre deux ITI communiquantes sans changer les propriétés du système.

5.3.4 L'adversaire nul

Définition. Comme précisé ci-dessus, l'adversaire nul est un adversaire particulier, et le plus difficile à simuler : il transmet à l'environnement tous les messages reçus des joueurs, et il

transmet aux joueurs les messages qu'il a reçu pour eux de la part de l'environnement. Il sert donc simplement de lien de transmission et donne tous les pouvoirs à l'environnement.

L'*adversaire nul* $\mathcal{D}$ procède de la manière suivante : quand il est activé par un message m sur son ruban de communication entrante, il transmet m (contenant l'identité de l'expéditeur) comme sortie à $\mathcal{Z}$. Lorsqu'il est activé par une entrée (m, id, c) de $\mathcal{Z}$ (message, identité, code), alors il délivre ce message au participant d'identité id. En particulier, lorsque cet adversaire corrompt des joueurs sur ordre de $\mathcal{Z}$, il transmet toute l'information obtenue à $\mathcal{Z}$.

Pour garantir que $\mathcal{D}$ est bien une PPT ITM, on lui impose de s'arrêter si le nombre de bits qu'il a besoin d'écrire sur les rubans de communication entrante d'autres ITI est supérieur au nombre de bits écrits sur son ruban d'entrée. Pour cela, on modifie un peu son comportement : quand il est activé par un message entrant sur son ruban de communication entrante, il transmet (`length`,$|m|$) comme sortie à $\mathcal{Z}$ (`length` étant un message spécial reconnu par l'environnement). S'il reçoit comme entrée $1^{|m|}$, il transmet m comme sortie à $\mathcal{Z}$, sinon il s'arrête (il atteint l'état *halt*).

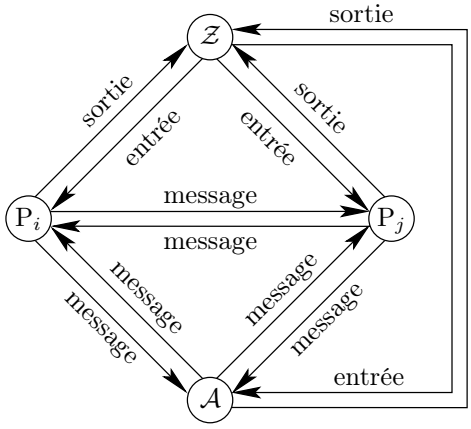
Équivalence avec la précédente définition.

Définition. *Le protocole ρ UC-émule le protocole ϕ par rapport à l'adversaire nul s'il existe un adversaire simulé PPT $\mathcal{S}$ tel que pour tout environnement $\mathcal{Z}$,*

$$\text{EXEC}_{\phi,\mathcal{S},\mathcal{Z}} \approx \text{EXEC}_{\rho,\mathcal{D},\mathcal{Z}}$$

Lemme (Substitution de l'adversaire)**.** *Soient ϕ et ρ des protocoles. Alors ρ UC-émule ϕ si et seulement si ρ UC-émule ϕ par rapport à l'adversaire nul.*

Preuve. Le sens direct est évident, montrons la réciproque. Rappelons les interactions possibles entre les différents participants au protocole sur le schéma ci-après.



Idée de la preuve. Le fonctionnement de l'adversaire nul peut être schématisé de la façon suivante (simplifiée) :

$$\mathrm{P}_i \xrightarrow{m} \mathcal{D} \xrightarrow{m} \mathcal{Z}$$
$$\mathcal{Z} \xrightarrow{m} \mathcal{D} \xrightarrow{m} \mathrm{P}_i$$

En quelque sorte, avec l'accès direct à la communication échangée entre les joueurs, l'environnement peut exécuter lui-même n'importe quel adversaire. Ainsi, la quantification sur les environnements implique déjà la quantification sur les adversaires.

Principe de la preuve. Soient ρ et ϕ des protocoles et $\widehat{\mathcal{S}}$ l'adversaire simulé dans la définition de l'émulation par rapport à l'adversaire nul :

$$\forall \mathcal{Z} \qquad \text{EXEC}_{\rho,\mathcal{D},\mathcal{Z}} \approx \text{EXEC}_{\phi,\widetilde{\mathcal{S}},\mathcal{Z}}$$

On se donne un adversaire $\mathcal{A}$ contre ρ et on construit un adversaire simulé $\mathcal{S}$ (utilisant $\widetilde{\mathcal{S}}$ et $\mathcal{A}$) contre ϕ tel que

$$\forall \mathcal{Z} \qquad \text{EXEC}_{\rho,\mathcal{A},\mathcal{Z}} \approx \text{EXEC}_{\phi,\mathcal{S},\mathcal{Z}}$$

Construction de $\mathcal{S}$. $\mathcal{S}$ doit savoir simuler différents cas de communication ; pour cela, il exécute des instances de $\mathcal{A}$ et de $\widetilde{\mathcal{S}}$ (en boîtes noires) : par rapport à eux, il joue le rôle d'environnement ; par rapport aux joueurs, il joue le rôle d'adversaire.

1. Communication environnement $\leftrightarrow$ adversaire :
   - quand $\mathcal{Z}$ veut envoyer une entrée à $\mathcal{A}$, il l'envoie en réalité à $\mathcal{S}$ et $\mathcal{S}$ la transmet à $\mathcal{A}$;
   - quand $\mathcal{A}$ veut envoyer une entrée à $\mathcal{Z}$, $\mathcal{S}$ l'envoie à $\mathcal{Z}$;
2. Communication $\mathcal{A} \rightarrow$ joueur :
   - quand $\mathcal{A}$ veut envoyer un message m à P_i, $\mathcal{S}$ active $\widetilde{\mathcal{S}}$ avec ce message (en effet, $\widetilde{\mathcal{S}}$ a l'impression de recevoir un message de l'environnement : il va tout de suite le transmettre au joueur concerné) ;
3. Communication $\widetilde{\mathcal{S}} \rightarrow$ environnement :
   - quand $\widetilde{\mathcal{S}}$ génère une sortie de la forme $(\texttt{length}, \ell)$, $\mathcal{S}$ lui envoie 1^ℓ (c'est ce que $\widetilde{\mathcal{S}}$ attend de l'environnement) ;
   - quand il génère une sortie v, alors $\mathcal{S}$ active $\mathcal{A}$ avec la valeur v (en effet, $\widetilde{\mathcal{S}}$ s'apprête à transmettre cette valeur à l'environnement, donc il l'a nécessairement reçue d'un joueur, comme l'adversaire nul) ;
4. Communication $\widetilde{\mathcal{S}} \leftrightarrow$ joueur
   - quand $\widetilde{\mathcal{S}}$ veut envoyer un message m à P_i, alors $\mathcal{S}$ envoie ce message à P_i ;
   - quand P_i veut envoyer un message à $\mathcal{A}$, il l'envoie en réalité à $\mathcal{S}$ et ce dernier l'envoie à $\widetilde{\mathcal{S}}$ (en effet, $\widetilde{\mathcal{S}}$ croit recevoir un message d'un joueur, donc il veut le transmettre à l'environnement ; on applique alors les deux points de 3. et finalement $\mathcal{A}$ reçoit bien le message) ;
5. Temps de calcul
   - si l'instance locale de $\mathcal{A}$ ou $\widetilde{\mathcal{S}}$ atteint sa borne en temps de calcul, $\mathcal{S}$ s'arrête.

$\mathcal{S}$ est PPT. On note $p_{\mathcal{A}}$ le polynôme bornant le temps de calcul de $\mathcal{A}$, $p_{\mathcal{S}}$ le polynôme bornant le temps de calcul de $\mathcal{S}$ et $c_{\mathcal{A}}$ le polynôme bornant le nombre de bits total écrits par $\mathcal{A}$ sur les rubans de communication entrante des autres ITI, plus le nombre de bits reçus par $\mathcal{A}$ sur son ruban de communication entrante. Alors le temps d'exécution de $\mathcal{S}$ sur une entrée de taille n est borné par $p_{\mathcal{A}}(n) + p_{\mathcal{S}}(c_{\mathcal{A}}(n))$.

Construction de $\widetilde{\mathcal{Z}}_0$. Pour prouver la validité de $\mathcal{S}$, on raisonne par l'absurde, en supposant l'existence d'un adversaire $\mathcal{A}_0$ et d'un environnement $\mathcal{Z}_0$ tels que

$$\mathrm{EXEC}_{\rho,\mathcal{A}_0,\mathcal{Z}_0} \not\approx \mathrm{EXEC}_{\phi,\mathcal{S},\mathcal{Z}_0}$$

On construit à partir de là un environnement $\widetilde{\mathcal{Z}}_0$ (utilisant $\mathcal{Z}_0$ et l'adversaire simulé $\mathcal{A}_0$) tel que

$$\mathrm{EXEC}_{\rho,\mathcal{D},\widetilde{\mathcal{Z}}_0} \not\approx \mathrm{EXEC}_{\phi,\widetilde{\mathcal{S}},\widetilde{\mathcal{Z}}_0}$$

et on conclut par contradiction. Il suffit donc d'avoir :

$$\mathrm{EXEC}_{\phi,\widetilde{\mathcal{S}},\widetilde{\mathcal{Z}}_0} = \mathrm{EXEC}_{\phi,\mathcal{S},\mathcal{Z}_0} \tag{1}$$

$$\mathrm{EXEC}_{\rho,\mathcal{D},\widetilde{\mathcal{Z}}_0} = \mathrm{EXEC}_{\rho,\mathcal{A}_0,\mathcal{Z}_0} \tag{2}$$

$\widetilde{\mathcal{Z}}_0$ doit savoir simuler différents cas de communication :

1. Communication adversaire $\leftrightarrow$ environnement :
   - quand l'environnement $\mathcal{Z}_0$ veut envoyer une entrée à $\mathcal{S}$, $\widetilde{\mathcal{Z}}_0$ l'envoie à $\widetilde{\mathcal{S}}$ et donc à $\mathcal{S}$;

- quand $\widetilde{\mathcal{Z}}_0$ est activé avec $(\texttt{length}, \ell)$ provenant de l'adversaire, il donne l'entrée 1^ℓ à l'adversaire ;
- quand il reçoit une autre valeur v de l'adversaire, $\widetilde{\mathcal{Z}}_0$ transmet v à l'adversaire $\mathcal{A}_0$;

2. Communication $\mathcal{A}_0 \rightarrow$ joueur :
   - quand $\mathcal{A}_0$ veut envoyer un message m à P_i, $\widetilde{\mathcal{Z}}$ envoie le message m à P_i ;
3. Communication $\mathcal{A}_0 \leftrightarrow \mathcal{Z}_0$:
   - $\widetilde{\mathcal{Z}}_0$ relaie toute l'information de $\mathcal{A}_0$ à $\mathcal{Z}_0$ et de $\mathcal{Z}_0$ à $\mathcal{A}_0$;
4. Communication environnement $\leftrightarrow$ joueur :
   - $\widetilde{\mathcal{Z}}_0$ relaie toutes les entrées de $\mathcal{Z}_0$ aux joueurs exécutant π ;
   - $\widetilde{\mathcal{Z}}_0$ relaie toutes les sorties des joueurs à $\mathcal{Z}_0$;
5. Sortie de l'exécution
   - la sortie de $\widetilde{\mathcal{Z}}_0$ est la sortie de $\mathcal{Z}_0$.

$\mathcal{S}$ n'arrive pas au bout de son temps de calcul avant que l'adversaire $\mathcal{A}_0$ à l'intérieur de $\mathcal{S}$ n'arrive au bout de son temps de calcul. De même, $\mathcal{D}$ n'arrive jamais au bout de son temps de calcul. Les deux égalités sont alors vérifiées, ce qui achève la preuve.

Nous arrivons ici au point le plus important de la sécurité UC, c'est-à-dire le résultat qui va permettre la composition sûre de protocoles.

5.4 Le théorème fondamental de la sécurité UC

5.4.1 Opération de composition universelle

Soit ρ un protocole qui UC-réalise une fonctionnalité idéale $\mathcal{F}$. Soit π un protocole dans lequel les participants utilisent la communication standard et font aussi des appels à des instances multiples de $\mathcal{F}$ (distinguées par des SID différentes générées par π). On a dit plus haut que π est alors appelé un *protocole $\mathcal{F}$-hybride*.

On construit ensuite le protocole composé π^ρ en remplaçant dans le protocole π chaque appel à une nouvelle instance de $\mathcal{F}$ par une invocation d'une nouvelle instance de ρ (de même pour les messages envoyés et reçus par $\mathcal{F}$).

5.4.2 Construction effective de protocoles

Le théorème de composition universelle n'a pas de preuve constructive ; il ne donne pas de méthode effective pour construire des protocoles sûrs pour la composition universelle. Il se contente de combiner petit à petit des briques de base simples pour obtenir en sortie des constructions compliquées qui donnent par composition des protocoles sûrs, mais il ne donne pas de manière d'obtenir ces briques de base.

Pour construire en pratique un protocole, les opérations habituellement effectuées sont les suivantes :

- définir la fonctionnalité idéale, qui est en quelque sorte la description de ce que l'on veut obtenir (une mise en gage, un échange de clefs, etc.) ; cette fonctionnalité doit être trivialement correcte et sûre ;
- construire un protocole réel qui réalise cette fonctionnalité ;
- faire une preuve directe en montrant qu'il existe un simulateur pour réaliser l'indistinguabilité entre le cas réel et la fonctionnalité.

5.4.3 Énoncé général

Le théorème s'applique pour un protocole tel qu'aucune instance de ce protocole n'échange d'information avec des instances d'un autre protocole. Malgré son fort intérêt théorique, il est donc rarement utilisé en pratique, car il est fréquent de rencontrer des protocoles s'exécutant de manière concurrente en utilisant les mêmes données. Cette propriété est exprimée dans la définition suivante :

Définition. *Une instance d'un protocole ρ est dite* subroutine respecting *si aucune sous-partie de cette instance n'échange des entrées ou des sorties avec des ITI qui ne soient pas des joueurs ou des sous-parties de cette instance (les autres entrées et sorties sont ignorées). Le protocole ρ est alors dit* subroutine respecting *si, quel que soit π, toute instance de ρ dans toute exécution de π^ρ est* subroutine respecting.

Nous sommes alors en mesure d'énoncer le théorème dans toute sa généralité. Le théorème réellement utilisé en pratique sera donné dans la section 5.6. Une représentation visuelle de ce théorème est donnée dans la figure 5.1.

Théorème (Composition universelle : énoncé général)**.** *Soient ϕ un protocole PPT idéal réalisant une fonctionnalité idéale $\mathcal{F}$ et ρ un protocole PPT réel UC-émulant ϕ, tous deux* subroutine respecting. *Soit π un protocole (hybride) utilisant ϕ. Alors le protocole*[6] $\mathrm{UC}(\pi, \rho, \phi) = \pi^{\rho/\phi}$ *dans lequel toutes les instances de ϕ sont remplacées par des instances de ρ UC-émule le protocole π.*

5.5 Preuve du théorème fondamental

5.5.1 Idée de la preuve

Rappelons que dans la définition de l'émulation d'un protocole par un autre, l'ordre des quantificateurs est :

$$\forall \mathcal{A} \quad \exists \mathcal{S}(\mathcal{A}) \quad \forall \mathcal{Z}$$

La preuve utilise l'équivalence avec la formulation faisant intervenir l'adversaire nul, les quantificateurs deviennent alors :

$$\exists \mathcal{S}(\mathcal{D}) \quad \forall \mathcal{Z}$$

L'émulation de ϕ par ρ (que l'on suppose comme hypothèse) s'écrit

$$\exists \widetilde{\mathcal{S}} \quad \forall \widetilde{\mathcal{Z}} \qquad \mathrm{EXEC}_{\rho,\mathcal{D},\widetilde{\mathcal{Z}}} \approx \mathrm{EXEC}_{\phi,\widetilde{\mathcal{S}},\widetilde{\mathcal{Z}}} \tag{1}$$

et celle de π par π^ρ (que l'on veut démontrer) s'écrit

$$\exists \mathcal{S} \quad \forall \mathcal{Z} \qquad \mathrm{EXEC}_{\pi^\rho,\mathcal{D},\mathcal{Z}} \approx \mathrm{EXEC}_{\pi,\mathcal{S},\mathcal{Z}} \tag{2}$$

L'idée de la preuve (par réduction) est donc :

- de construire un adversaire $\mathcal{S}$ à π utilisant $\widetilde{\mathcal{S}}$;
- de supposer qu'il n'est pas valide, c'est-à-dire

$$\exists \mathcal{Z}_0 \qquad \mathrm{EXEC}_{\pi^\rho,\mathcal{D},\mathcal{Z}_0} \not\approx \mathrm{EXEC}_{\pi,\mathcal{S},\mathcal{Z}_0}$$

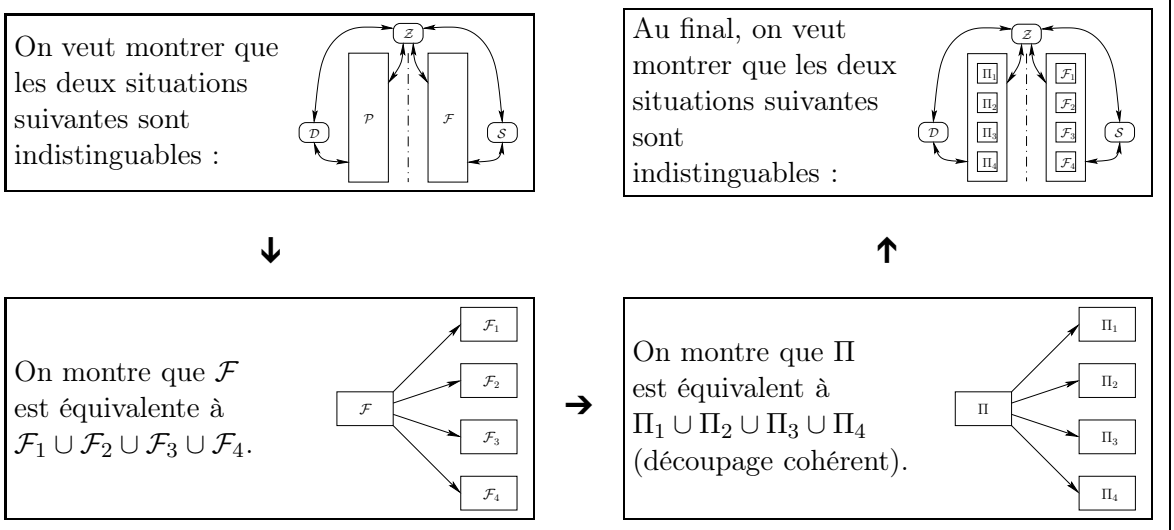
- d'en déduire (en construisant $\widetilde{\mathcal{Z}}_0$)

$$\exists \widetilde{\mathcal{Z}}_0 \qquad \mathrm{EXEC}_{\rho,\mathcal{D},\widetilde{\mathcal{Z}}_0} \not\approx \mathrm{EXEC}_{\phi,\widetilde{\mathcal{S}},\widetilde{\mathcal{Z}}_0}$$

- et de conclure par contradiction.

[6] Noté fréquemment π^ρ si aucune confusion n'est possible.

Fig. 5.1 – Interprétation schématique du théorème UC



Exemple de $\mathcal{F}$ équivalente à $\mathcal{F}_1 \cup \mathcal{F}_2$:
échange de clefs authentifié + schéma de chiffrement symétrique sûr → canaux sécurisés
En réalité, $\mathcal{Z}$ et $\mathcal{A}$ interagissent avec les joueurs, pas avec les protocoles. On a ici choisi de simplifier les notations pour ne pas surcharger les schémas.

5.5.2 Construction de l'adversaire $\mathcal{S}$ à partir de $\widetilde{\mathcal{S}}$

Idée de la construction. Pour obtenir l'indistinguabilité, $\mathcal{Z}$ doit avoir l'impression d'interagir avec $\mathcal{D}$ et π^ρ. Dans la réalité, il interagit avec $\mathcal{S}$ et π. En particulier, $\mathcal{Z}$ croit interagir avec ρ tandis que $\mathcal{S}$ interagit avec ϕ. Ainsi, l'idée est la suivante :

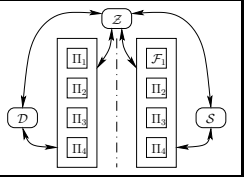
- quand l'adversaire $\mathcal{S}$ doit interagir avec ρ, il exécute $\widetilde{\mathcal{S}}$ qui sait simuler lors de son interaction avec ϕ le comportement de $\mathcal{D}$ avec ρ ;
- quand il doit interagir avec π, il le fait directement en imitant le comportement de $\mathcal{D}$.

Comportement de l'adversaire $\mathcal{S}$. $\mathcal{S}$ exécute des instances simulées de $\widetilde{\mathcal{S}}$; il doit savoir simuler les cas suivants (le cas de la corruption est implicitement traité, car c'est un cas particulier de message délivré par $\mathcal{S}$) :

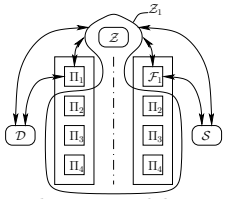
1. Communication environnement → adversaire :
   - si $\mathcal{S}$ reçoit une entrée $(m, \mathsf{id} = (\mathsf{sid}, \mathsf{pid}), \rho)$ provenant de $\mathcal{Z}$, et qu'il n'y a pas d'instance de $\widetilde{\mathcal{S}}$ de SID sid, invoquer cette instance avec (m, id, ρ) ;
   - si $\mathcal{S}$ reçoit une entrée $(m, \mathsf{id} = (\mathsf{sid}, \mathsf{pid}), \rho)$ provenant de $\mathcal{Z}$, et qu'il y a une instance de $\widetilde{\mathcal{S}}$ de SID sid, activer cette instance avec (m, id, ρ) ;
   - si $\mathcal{S}$ reçoit une entrée $(m, \mathsf{id} = (\mathsf{sid}, \mathsf{pid}), \mathit{code})$ provenant de $\mathcal{Z}$, et que $\mathit{code} \neq \rho$, délivrer le message au joueur concerné ;
2. Communication joueur → adversaire :
   - si $\mathcal{S}$ reçoit un message en provenance d'un joueur exécutant ϕ, invoquer (ou activer) l'instance de $\widetilde{\mathcal{S}}$ correspondante ;
   - si $\mathcal{S}$ reçoit un message en provenance d'un joueur (ou d'une sous-partie) exécutant π, imiter le comportement de l'adversaire nul ;
3. Communication $\widetilde{\mathcal{S}}$ → joueur :
   - si une instance de $\widetilde{\mathcal{S}}$ veut délivrer un message m à un joueur exécutant ϕ, délivrer le message à ce joueur ;

Fig. 5.2 – Le théorème UC en pratique

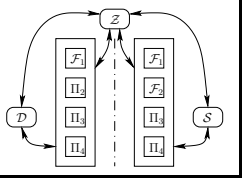
Première étape :
Si $\mathcal{Z}$ peut distinguer les deux situations suivantes...



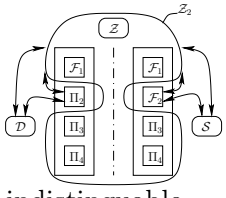
... alors $\mathcal{Z}_1$ peut distinguer les deux situations suivantes, ce qui contredit « Π_1 UC-émule $\mathcal{F}_1$ ». Changer Π_1 en $\mathcal{F}_1$ est indistinguable.



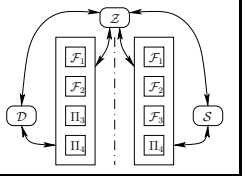
Deuxième étape :
Si $\mathcal{Z}$ peut distinguer les deux situations suivantes...



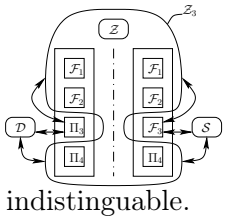
... alors $\mathcal{Z}_2$ peut distinguer les deux situations suivantes, ce qui contredit « Π_2 UC-émule $\mathcal{F}_2$ ». Changer Π_2 en $\mathcal{F}_2$ est indistinguable.



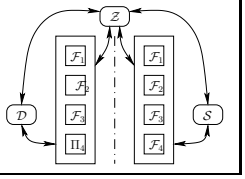
Troisième étape :
Si $\mathcal{Z}$ peut distinguer les deux situations suivantes...



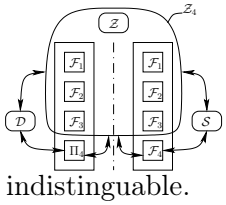
... alors $\mathcal{Z}_3$ peut distinguer les deux situations suivantes, ce qui contredit « Π_3 UC-émule $\mathcal{F}_3$ ». Changer Π_3 en $\mathcal{F}_3$ est indistinguable.



Quatrième étape :
Si $\mathcal{Z}$ peut distinguer les deux situations suivantes...



... alors $\mathcal{Z}_4$ peut distinguer les deux situations suivantes, ce qui contredit « Π_4 UC-émule $\mathcal{F}_4$ ». Changer Π_4 en $\mathcal{F}_4$ est indistinguable.



Comme dans la figure 5.1, les notations sont simplifiées pour ne pas surcharger les schémas.

4. Communication $\widetilde{\mathcal{S}} \rightarrow$ environnement :

   - si une instance de $\widetilde{\mathcal{S}}$ veut donner une sortie à l'environnement, imiter le comportement de l'adversaire nul.

L'adversaire $\mathcal{S}$ est PPT. Par construction, son temps d'exécution est borné par le polynôme bornant le temps de $\widetilde{\mathcal{S}}$ plus un polynôme linéaire (rendant compte du relais des messages concernant le protocole π).

5.5.3 Construction de l'environnement $\widetilde{\mathcal{Z}}_0$ à partir de $\mathcal{Z}_0$

Principe. L'hypothèse est l'existence d'un environnement $\mathcal{Z}_0$ ne satisfaisant pas (2). On construit alors à partir de ce dernier un environnement $\widetilde{\mathcal{Z}}_0$ ne satisfaisant pas (1). On suppose donc l'existence de $\varepsilon > 0$ tel que

$$|\Pr[\text{EXEC}_{\pi^\rho,\mathcal{D},\mathcal{Z}_0}(k,z)=1] - \Pr[\text{EXEC}_{\pi,\mathcal{S},\mathcal{Z}_0}(k,z)=1]| \geqslant \varepsilon \qquad (2')$$

Constructions de variantes de $\mathcal{S}$ et $\mathcal{Z}_0$. On considère une exécution du protocole π avec l'adversaire $\mathcal{S}$ et l'environnement $\mathcal{Z}_0$: on note $t = t(k, |z|)$ un majorant (qui est polynomial) du nombre n d'instances de ϕ dans cette exécution.

On utilise un argument hybride standard : on sait que $\mathcal{Z}_0$ sait distinguer entre une interaction avec π et $\mathcal{S}$ et une interaction avec $\pi^{\rho/\phi}$ et $\mathcal{D}$. L'idée est donc de trouver des « intermédiaires » entre ces interactions, en remplaçant petit à petit les instances de ϕ dans π par des instances de ρ. On saura alors que $\mathcal{Z}_0$ sait distinguer entre deux de ces interactions successives.

Modèle ℓ-hybride du protocole π $(0 \leqslant \ell \leqslant n)$, noté π_ℓ. Informellement, π_ℓ est le protocole π dans lequel les ℓ premières instances de ϕ sont inchangées et les suivantes remplacées par ρ : en particulier $\pi_n = \pi$ et $\pi_0 = \pi^{\rho/\phi}$.

Plus précisément, le protocole π_ℓ utilise le même système étendu d'ITM, mais la fonction de contrôle est modifiée :

- les requêtes EW aux ℓ premières instances de ϕ sont inchangées ;
- les requêtes EW aux instances suivantes de ϕ sont redirigées aux instances des participants exécutant les instances de ρ correspondantes.

Variante de l'environnement $\mathcal{Z}_0$. $\mathcal{Z}_0^{(\ell)}$ se comporte comme $\mathcal{Z}_0$ mais inclut une variable a dans les messages à l'adversaire concernant ρ :

- $a = 1$ si ρ est l'une des ℓ premières instances ;
- $a = 0$ sinon.

Variante de l'adversaire $\mathcal{S}$. De la même manière, $\widehat{\mathcal{S}}$ se comporte comme $\mathcal{S}$ mais inclut une variable supplémentaire s (définie par a) :

- si $s = 1$, $\widehat{\mathcal{S}}$ et $\mathcal{S}$ sont strictement identiques ;
- si $s = 0$, $\widehat{\mathcal{S}}$ n'invoque pas d'instance de $\widetilde{\mathcal{S}}$ dans le point 1 décrit en début de section.

Remarque. Dans l'interaction de $\widehat{\mathcal{S}}$ avec $\mathcal{Z}_0^{(\ell)}$ et π_ℓ, il exécute au plus ℓ instances de $\widetilde{\mathcal{S}}$, les autres étant remplacées par des interactions directes avec ρ ou ses sous-parties. Par conséquent,

$$\mathrm{EXEC}_{\pi_n,\widehat{\mathcal{S}},\mathcal{Z}_0^{(n)}} = \mathrm{EXEC}_{\pi,\mathcal{S},\mathcal{Z}_0}$$

$$\mathrm{EXEC}_{\pi_0,\widehat{\mathcal{S}},\mathcal{Z}_0^{(0)}} = \mathrm{EXEC}_{\pi^\rho,\mathcal{D},\mathcal{Z}_0}$$

En effet, $\pi_n = \pi$ et $a = 1$, donc $\widehat{\mathcal{S}} = \mathcal{S}$ dans le premier cas, et $\pi_0 = \pi^\rho$ et $a = 0$, donc $\widehat{\mathcal{S}} = \mathcal{D}$ dans le second cas.

Conséquences sur la formule (2′). La formule

$$\left|\Pr\left[\mathrm{EXEC}_{\pi^\rho,\mathcal{D},\mathcal{Z}_0}(k,z) = 1\right] - \Pr\left[\mathrm{EXEC}_{\pi,\mathcal{S},\mathcal{Z}_0}(k,z) = 1\right]\right| \geqslant \varepsilon \tag{2′}$$

se réécrit

$$\left|\Pr\left[\mathrm{EXEC}_{\pi_0,\widehat{\mathcal{S}},\mathcal{Z}_0^{(0)}}(k,z) = 1\right] - \Pr\left[\mathrm{EXEC}_{\pi_n,\widehat{\mathcal{S}},\mathcal{Z}_0^{(n)}}(k,z) = 1\right]\right| \geqslant \varepsilon \tag{2″}$$

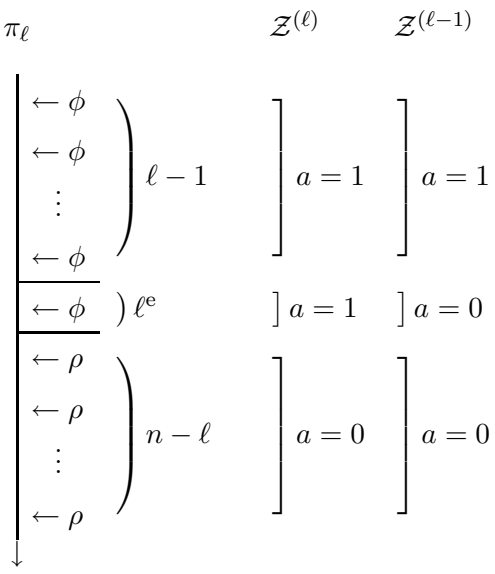
Cette dernière formule implique l'existence de $\ell \in \{1, \ldots, n-1\}$ tel que

$$\left|\Pr\left[\mathrm{EXEC}_{\pi_{\ell-1},\widehat{\mathcal{S}},\mathcal{Z}_0^{(\ell-1)}}(k,z) = 1\right] - \Pr\left[\mathrm{EXEC}_{\pi_\ell,\widehat{\mathcal{S}},\mathcal{Z}_0^{(\ell)}}(k,z) = 1\right]\right| \geqslant \varepsilon/n \tag{2‴}$$

Or,

$$\mathrm{EXEC}_{\pi_\ell,\widehat{\mathcal{S}},\mathcal{Z}_0^{(\ell-1)}} \approx \mathrm{EXEC}_{\pi_\ell,\widehat{\mathcal{S}},\mathcal{Z}_0^{(\ell)}}$$

En effet, l'exécution de π_ℓ dans les deux environnements se schématise ainsi :



et l'on remarque que la valeur de a à la ℓ-ième occurrence de ϕ (la seule différence) ne change rien aux exécutions, car $\widehat{\mathcal{S}}$ n'avait de toute façon pas besoin d'invoquer $\widetilde{\mathcal{S}}$. Finalement,

$$\boxed{\left|\Pr\left[\mathrm{EXEC}_{\pi_{\ell-1},\widehat{\mathcal{S}},\mathcal{Z}_0^{(\ell-1)}}(k,z)=1\right]-\Pr\left[\mathrm{EXEC}_{\pi_\ell,\widehat{\mathcal{S}},\mathcal{Z}_0^{(\ell-1)}}(k,z)=1\right]\right|\geqslant \varepsilon/n} \tag{3}$$

Construction de $\widetilde{\mathcal{Z}}_0$. On souhaite montrer que

$$\left|\Pr\left[\mathrm{EXEC}_{\rho,\mathcal{D},\widetilde{\mathcal{Z}}_0}(k,z)=1\right]-\Pr\left[\mathrm{EXEC}_{\phi,\widetilde{\mathcal{S}},\widetilde{\mathcal{Z}}_0}(k,z)=1\right]\right|\geqslant \varepsilon/n \tag{1'}$$

Pour cela, il suffit de définir $\widetilde{\mathcal{Z}}_0$ tel que :

$$\mathrm{EXEC}_{\pi_{\ell-1},\widehat{\mathcal{S}},\mathcal{Z}_0^{(\ell-1)}} = \mathrm{EXEC}_{\rho,\mathcal{D},\widetilde{\mathcal{Z}}_0} \tag{E0}$$

et

$$\mathrm{EXEC}_{\pi_\ell,\widehat{\mathcal{S}},\mathcal{Z}_0^{(\ell-1)}} = \mathrm{EXEC}_{\phi,\widetilde{\mathcal{S}},\widetilde{\mathcal{Z}}_0} \tag{E1}$$

Dans les membres de droite, on ne fait qu'une seule exécution de ϕ ou ρ. Intuitivement, on va simuler les $(\ell-1)$ premières exécutions et se concentrer sur la suivante. On construit donc $\widetilde{\mathcal{Z}}_0$ comme décrit sur la figure 5.3.

On peut dissocier de cette façon les instances de ϕ et ρ grâce à l'hypothèse « *subroutine-respecting* », qui assure que les instances externes n'échangent pas d'entrées/sorties avec l'instance concernée.

Dans l'exécution de la ℓ-ième instance de ϕ dans le protocole $\pi_{\ell-1}$ en présence de $\mathbb{Z}_0^{\ell-1}$, c'est ρ qui est exécuté, donc $\widehat{\mathcal{S}}$ se comporte comme l'adversaire nul.

En revanche, dans l'exécution de la ℓ-ième instance de ϕ dans le protocole π_ℓ en présence de $\mathbb{Z}_0^{\ell-1}$, c'est ϕ qui est exécuté, donc $\widehat{\mathcal{S}}$ se comporte comme l'adversaire $\widetilde{\mathcal{S}}$.

Les équations (3), (E0) et (E1) donnent alors l'équation (1′) et la contradiction souhaitée, ce qui achève la démonstration.

Remarquons finalement qu'adversaire et environnement doivent pouvoir communiquer à tout moment ; en particulier, l'adversaire ne peut pas « rembobiner » l'environnement.

5.5.4 Extensions

On peut exprimer le théorème de composition universelle de manière quantitative, ce qui revient à dire que, si t est une borne sur le nombre d'instances de ρ dans π^ρ, alors l'*emulation slack* (probabilité de distinction par l'environnement) est multiplié par t tandis que l'*emulation overhead* (rapport entre les temps d'exécution de l'adversaire simulé et de l'adversaire de départ) reste inchangé.

Une preuve alternative au théorème serait de le démontrer dans le cadre d'une instance unique, puis l'appliquer itérativement, mais la qualité de l'émulation s'en trouverait dégradée.

Fig. 5.3 – Description de l'environnement $\widetilde{\mathcal{Z}}_0$

Procédure `simulate`(s, ℓ)

Le paramètre s contient l'état global d'un système représentant l'exécution du protocole π_ℓ en présence de l'adversaire $\widehat{\mathcal{S}}$ et de l'environnement $\mathcal{Z}_0$.

Simuler l'exécution de ce système à partir de l'état s jusqu'à l'apparition de l'un des cas suivants (on note ϕ_ℓ la ℓ-ième instance de ϕ invoquée, et sid_ℓ son SID) :

- un joueur simulé envoie x au joueur d'identité $(\mathsf{sid}_\ell, \mathsf{pid})$: dans ce cas, enregistrer l'état courant du système simulé dans s, donner x au joueur réel d'identité $(\mathsf{sid}_\ell, \mathsf{pid})$ et terminer cette activation.
- l'environnement simulé $\mathcal{Z}_0$ envoie (m, id, c, a) à l'adversaire simulé $\widehat{\mathcal{S}}$, avec $\mathsf{id} = (\mathsf{sid}_\ell, \mathsf{pid})$: dans ce cas, enregistrer l'état courant du système simulé dans s, délivrer le message (m, id, c) au joueur exécutant le code c avec l'identité id et terminer cette activation.
- l'environnement simulé $\mathcal{Z}_0$ s'arrête : dans ce cas, $\widetilde{\mathcal{Z}}_0$ envoie en sortie la sortie de $\mathcal{Z}_0$ et s'arrête.

Programme principal

L'entrée initiale de l'environnement est de la forme $\widetilde{z} = (z, \ell)$, où z est une entrée pour $\mathcal{Z}_0$ et $\ell \in \mathbb{N}$.

1. À la première activation, initialiser une variable s pouvant contenir l'état global d'un système représentant l'exécution du protocole π_ℓ en présence de l'adversaire $\widehat{\mathcal{S}}$ et de l'environnement $\mathcal{Z}_0$. Exécuter ensuite `simulate`(s, ℓ).
2. À une activation suivante (par une valeur x), commencer par mettre à jour l'état s, c'est-à-dire :
   - si la valeur x a été écrite par un joueur $\mathrm{P}_{(\mathsf{id})}$ (exécutant ϕ ou ρ) d'identité id de la forme $(\mathsf{sid}_\ell, \mathsf{pid})$, envoyer (x, id) au joueur simulé qui a invoqué $\mathrm{P}_{(\mathsf{id})}$;
   - si la valeur x a été écrite par l'adversaire, mettre à jour l'état de l'adversaire simulé $\widehat{\mathcal{S}}$ pour y inclure une sortie x générée par $\mathcal{S}_{\mathsf{sid}_\ell}$.

   Exécuter ensuite `simulate`(s, ℓ).

5.5.5 Emboîtement d'instances de protocoles

Des applications répétées de l'opérateur de composition continuent à maintenir la sécurité. Par exemple, pour un nombre d'applications constant : si un protocole ρ_1 UC-émule un protocole ϕ_1, et qu'un protocole ρ_2 UC-émule un protocole ϕ_2 qui utilise des appels à ϕ_1, alors pour tout protocole π qui utilise des appels à ϕ_2, le protocole composé $\pi^{(\rho_2{}^{\rho_1/\phi_1})/\phi_2} = \mathrm{UC}(\pi, \mathrm{UC}(\rho_2, \rho_1, \phi_1), \phi_2)$ UC-émule π.

La sécurité est en fait maintenue si le nombre d'applications (« la profondeur de l'emboîtement ») est polynomiale en le paramètre de sécurité, et s'il existe un polynôme qui borne la complexité de tous les simulateurs. Le théorème est énoncé par simplicité dans le cas où les protocoles remplacés sont les mêmes (ϕ) à tous les niveaux, de même pour ceux qui remplacent (ρ). Si ρ et ϕ sont des protocoles et que ρ est un protocole ϕ-hybride, on note $\rho^{\langle 0 \rangle} = \rho$ et $\rho^{\langle i+1 \rangle} = \mathrm{UC}(\rho^{\langle i \rangle}, \rho, \phi)$.

Théorème (Composition universelle : emboîtement polynomial)**.** *Soit $t : \mathbb{N} \to \mathbb{N}$ un polynôme. Soient ϕ un protocole et ρ un protocole ϕ-hybride qui UC-émule ϕ. Alors le protocole $\rho^{\langle t(k) \rangle}$ UC-émule ϕ.*

5.6 Le théorème de composition universelle avec états joints

Dans les hypothèses du théorème de composition universelle, chaque copie de $\mathcal{F}$ est remplacée par une invocation d'une copie différente de ρ. Cela implique que toutes ces copies ρ ont des états disjoints et des éléments aléatoires indépendants (tout ce qui ne vient pas de l'environnement, les clefs secrètes, aléas, etc.).

Dans la pratique, les copies du protocole ρ pourront être à états joints. Pour continuer à pouvoir appliquer le théorème, l'objectif est de remplacer les invocations de $\mathcal{F}$ par une instance *unique* d'un « protocole joint » $\widehat{\rho}$.

5.6.1 Opération de composition

Fonctionnalité multi-session $\widehat{\mathcal{F}}$ pour $\mathcal{F}$. La fonctionnalité multi-session $\widehat{\mathcal{F}}$ est une fonctionnalité unique pour modéliser plusieurs fonctionnalités $\mathcal{F}$ en une seule. Remarquons que cette fonctionnalité $\widehat{\mathcal{F}}$ n'est pas utilisée par π, c'est juste une manière de formaliser le protocole simulé par $\widehat{\rho}$ (qui peut utiliser des états joints). Elle est définie de la manière suivante :

- elle s'attend à recevoir des messages entrants de la forme $(\mathsf{sid}, \mathsf{ssid}, v)$ où sid est le SID de $\widehat{\mathcal{F}}$ et ssid le SID de la copie de $\mathcal{F}$ concernée ;
- quand elle reçoit un tel message, elle active ou invoque $\mathcal{F}(\mathsf{ssid})$ avec le message (ssid, v) ;
- quand une copie de $\mathcal{F}$ envoie (ssid, v) à P_i, $\widehat{\mathcal{F}}$ envoie $(\mathsf{sid}, \mathsf{ssid}, v)$ à P_i et ssid à $\mathcal{A}$.

Opération de composition. Soient π un protocole $\mathcal{F}$-hybride, et $\widehat{\rho}$ un protocole qui UC-réalise $\widehat{\mathcal{F}}$. Le protocole composé $\pi^{[\widehat{\rho}]}$ est défini comme dans le cadre de la composition universelle classique sauf que :

- P_i invoque une seule copie de $\widehat{\rho}$ et remplace tous les appels aux copies de $\mathcal{F}$ par des activations de cette unique copie ;
- chaque activation de $\widehat{\rho}$ comporte deux SID :
  - celle de $\widehat{\rho}$ est fixée à sid_0 prédéfinie ;
  - le SSID vaut le SID original de l'invocation de $\mathcal{F}$.

Son comportement est défini de la façon suivante :

- à sa première activation par P_i, $\pi^{[\widehat{\rho}]}$ initialise une copie du protocole $\widehat{\rho}$ de SID sid_0 ;
- quand π demande à P_i d'envoyer un message (sid, v) à $\mathcal{F}(\mathsf{sid})$, $\pi^{[\widehat{\rho}]}$ demande à P_i d'invoquer $\widehat{\rho}$ avec $(\mathsf{sid}_0, \mathsf{sid}, v)$;
- quand $\widehat{\rho}$ génère comme sortie $(\mathsf{sid}_0, \mathsf{sid}, v)$, $\pi^{[\widehat{\rho}]}$ procède comme π quand ce dernier reçoit (sid, v) de $\mathcal{F}(\mathsf{sid})$;
- quand $\widehat{\rho}$ veut envoyer un message m à un participant P_j, P_i écrit le message $(\mathsf{sid}_0, \mathsf{sid}, m)$ sur son ruban de communication sortante ;
- quand P_i est activé par un message $(\mathsf{sid}_0, \mathsf{sid}, m)$ provenant de P_j, P_i active $\widehat{\rho}$ avec le message $(\mathsf{sid}_0, \mathsf{sid}, m)$.

5.6.2 Énoncé et preuve

Théorème (Composition universelle avec états joints)**.** *Soient $\mathcal{F}$ une fonctionnalité idéale, $\widehat{\mathcal{F}}$ son extension multi-session, π un protocole $\mathcal{F}$-hybride et $\widehat{\rho}$ un protocole qui UC-réalise $\widehat{\mathcal{F}}$. Alors le protocole $\pi^{[\widehat{\rho}]}$ UC-émule le protocole π.*

Idée de la preuve. Soient $\mathcal{F}$ une fonctionnalité idéale, π un protocole $\mathcal{F}$-hybride et $\widehat{\rho}$ un protocole qui UC-réalise $\widehat{\mathcal{F}}$ (l'extension multi-session de $\mathcal{F}$). On veut montrer que $\pi^{[\widehat{\rho}]}$ UC-émule π. On procède en trois temps :

- on définit le protocole $\widehat{\mathcal{F}}$-hybride $\widetilde{\pi}$ (se comportant comme π) ;
- on remarque que $\widetilde{\pi}^{\widehat{\rho}} = \pi^{[\widehat{\rho}]}$ et on conclut par le théorème UC que $\pi^{[\widehat{\rho}]}$ émule $\widetilde{\pi}$;
- on démontre que $\widetilde{\pi}$ émule π ;
- et on conclut par transitivité.

Définition du protocole $\widehat{\mathcal{F}}$-hybride $\widetilde{\pi}$. Il est construit à partir de π de la façon suivante :

- quand π demande à P_i d'envoyer (sid, v) à $\mathcal{F}(\mathsf{sid})$, $\widetilde{\pi}$ demande à P_i d'envoyer $(\mathsf{sid}_0, \mathsf{sid}, v)$ à $\widehat{\mathcal{F}}$;
- quand un participant P_i exécutant $\widetilde{\pi}$ reçoit un message $(\mathsf{sid}_0, \mathsf{sid}, v)$ de $\widehat{\mathcal{F}}$, P_i suit les instructions de π sur réception de (sid, v) de $\mathcal{F}(\mathsf{sid})$.

Émulation de $\widetilde{\pi}$ par $\pi^{[\widehat{\rho}]}$. On remarque que $\widetilde{\pi}^{\widehat{\rho}} = \pi^{[\widehat{\rho}]}$ (ce sont deux manières de décrire le même protocole). En outre, comme $\widehat{\rho}$ réalise $\widehat{\mathcal{F}}$ de manière sûre, on en déduit par le théorème de composition universelle que $\widetilde{\pi}^{\widehat{\rho}}$ émule $\widetilde{\pi}$. Finalement, $\pi^{[\widehat{\rho}]}$ émule $\widetilde{\pi}$.

Émulation de π par $\widetilde{\pi}$. On considère un adversaire $\widehat{\mathcal{A}}$ interagissant avec $\widetilde{\pi}$. On en déduit la construction d'un adversaire $\mathcal{A}$ interagissant avec π tel que les deux exécutions soient indistinguables pour un environnement (ce qui se vérifie directement) :

- quand $\mathcal{A}$ apprend que $\mathcal{F}(\mathsf{sid})$ a envoyé un message d'identifiant id à P_i, il sauvegarde la paire $(\mathsf{sid}, \mathsf{id})$ et prévient $\widehat{\mathcal{A}}$ que $\widehat{\mathcal{F}}(\mathsf{sid}_0)$ a envoyé un message d'identifiant id à P_i (aucun des deux adversaires ne voit le contenu de ce message) ;
- quand $\widehat{\mathcal{A}}$ délivre un message d'identifiant id de $\widehat{\mathcal{F}}$ à P_i, $\mathcal{A}$ cherche la paire $(\mathsf{sid}, \mathsf{id})$ et délivre le message de $\mathcal{F}(\mathsf{sid})$ à P_i ;
- quand $\widehat{\mathcal{A}}$ corrompt P_i, $\mathcal{A}$ corrompt P_i et obtient son état interne pour le protocole π. Il le traduit en état interne pour $\widetilde{\pi}$ (en remplaçant les appels à de multiples copies de $\mathcal{F}$ par un appel à une copie de $\widehat{\mathcal{F}}$ de sid_0 fixé et de SSID correspondants) et transmet cette information à $\widehat{\mathcal{F}}$.

Cela achève la preuve du théorème.

5.7 Les modèles idéaux et la CRS dans le cadre UC

Comme pour toute primitive, les modèles doivent être formalisés dans le cadre de la composabilité universelle par les fonctionnalités idéales correspondantes. On parlera alors de protocole $\mathcal{F}_{\mathrm{RO}}$-hybride pour un protocole dans le modèle de l'oracle aléatoire, par exemple.

Canetti *et al.* ont montré dans [CHK+05] l'inexistence d'un protocole UC-émulant la fonctionnalité d'échange de clefs à base de mots de passe à deux joueurs (présentée dans la section 5.8.1) dans le cadre de base (c'est-à-dire sans hypothèses de *setup* additionnelles). Ceci justifie l'étude des protocoles hybrides dans les modèles $\mathcal{F}_{\mathrm{RO}}$, $\mathcal{F}_{\mathrm{IC}}$, $\mathcal{F}_{\mathrm{ITC}}$ et $\mathcal{F}_{\mathrm{CRS}}$ présentés dans cette section.

Pour l'oracle aléatoire comme pour les chiffrements idéaux, puisque l'identifiant de session sid sera inclus dans les protocoles dans les entrées à ces fonctions, nous pouvons n'avoir qu'une seule instanciation de chaque, dans l'état joint, et non des instanciations différentes pour chaque session, ce qui est plus réaliste. Notons cependant que dans les cas où seule une partie de l'identifiant de session est inclus, cela pourrait mener à des collisions avec d'autres sessions partageant cette partie du sid et donc à des problèmes dans la simulation, et plus précisément dans la programmation de l'oracle aléatoire et des chiffrements idéaux. Mais la programmation est faite pour des joueurs honnêtes uniquement. Il ne reste donc qu'à s'assurer, lors de la conception d'un protocole, que cette programmation ne devra bien avoir lieu qu'une fois (sauf, bien sûr, si l'adversaire a déjà posé la requête critique, mais ceci n'arrive qu'avec probabilité négligeable). Ceci sera généralement effectué grâce à la fonctionnalité de répartition, que nous présentons dans la section 5.8.3 page 85.

5.7.1 Oracle aléatoire [BR93] dans le cadre UC

La fonctionnalité idéale pour l'oracle aléatoire (ROM, pour *random oracle* en anglais) a été définie par Hofheinz et Müller-Quade dans [HMQ04]. Nous la présentons ici dans la figure 5.4. Il est clair que le modèle de l'oracle aléatoire UC-émule cette fonctionnalité.

Fig. 5.4 – La fonctionnalité $\mathcal{F}_{\mathrm{RO}}$

La fonctionnalité $\mathcal{F}_{\mathrm{RO}}$ procède de la façon suivante, avec le paramètre de sécurité k, les joueurs $\mathrm{P}_1, \ldots, \mathrm{P}_n$ et un adversaire $\mathcal{S}$:

- $\mathcal{F}_{\mathrm{RO}}$ maintient une liste $\mathrm{L}_{\mathsf{sid}}$ (initialement vide) de couples de chaînes de bits.
- À la réception d'une valeur (sid, m) (avec $m \in \{0,1\}^*$) de la part d'un joueur P_i ou de l'adversaire $\mathcal{S}$:
  - S'il existe un couple $(m, \widetilde{h})$ pour un certain $\widetilde{h} \in \{0,1\}^k$ dans la liste $\mathrm{L}_{\mathsf{sid}}$, poser $h := \widetilde{h}$.
  - S'il n'existe pas de tel couple, choisir uniformément $h \in \{0,1\}^k$ et enregistrer le couple $(m, h) \in \mathrm{L}$.

  Une fois h fixé, renvoyer la réponse (sid, h) à la personne qui en fait la demande (c'est-à-dire le joueur P_i ou bien l'adversaire $\mathcal{S}$).

5.7.2 Chiffrement idéal [LRW02] dans le cadre UC

Un chiffrement idéal (IC, pour *ideal cipher* en anglais) est un chiffrement par blocs qui prend en entrée un message clair ou chiffré. Nous avons défini dans [ACCP08] la fonctionnalité idéale $\mathcal{F}_{\mathrm{IC}}$ décrite sur la figure 5.5. Comme pour l'oracle aléatoire, le modèle idéal UC-émule cette fonctionnalité. Il est à noter que cette dernière caractérise une permutation parfaitement aléatoire, en assurant l'injectivité pour la simulation de chaque requête.

5.7.3 Chiffrement idéal étiqueté dans le cadre UC

Comme on l'a expliqué page 22, un chiffrement étiqueté est un chiffrement par blocs qui accepte une deuxième entrée publique, appelée l'étiquette, en plus de la clef et du message clair ou chiffré. C'est cette étiquette, associée à la clef, qui choisit la permutation calculée par le schéma.

La fonctionnalité (que nous avons présentée dans [ACCP09]) maintient, en plus d'une liste $\mathrm{L}_{\mathsf{sid}}$ (intialement vide) de quadruplets de chaînes de bits, un certain nombre d'ensembles (initialement vides) de la forme $\mathrm{C}_{\mathsf{key},\mathsf{label},\mathsf{sid}}, \mathrm{M}_{\mathsf{key},\mathsf{label},\mathsf{sid}}$ définis par

$$\mathrm{C}_{\mathsf{key},\mathsf{label},\mathsf{sid}} = \{c \mid \exists m \ (\mathsf{key}, \mathsf{label}, m, c) \in \mathrm{L}_{\mathsf{sid}}\}$$
$$\mathrm{M}_{\mathsf{key},\mathsf{label},\mathsf{sid}} = \{m \mid \exists c \ (\mathsf{key}, \mathsf{label}, m, c) \in \mathrm{L}_{\mathsf{sid}}\}$$

Elle est décrite sur la figure 5.6.

5.7.4 Le modèle de la CRS [BCNP04]

La fonctionnalité $\mathcal{F}_{\mathrm{CRS}}^{\mathcal{D}}$ est présentée par Barak, Canetti, Nielsen et Pass dans [BCNP04]. À chaque appel à $\mathcal{F}_{\mathrm{CRS}}^{\mathcal{D}}$, cette dernière renvoie la même chaîne de référence, qui a été choisie par elle, suivant une distribution publique connue $\mathcal{D}$. Nous la rappelons ici sur la figure 5.7.

Fig. 5.5 – La fonctionnalité $\mathcal{F}_{\text{IC}}$

La fonctionnalité $\mathcal{F}_{\text{IC}}$ prend en entrée un paramètre de sécurité k, et interagit avec un adversaire $\mathcal{S}$ et un ensemble de joueurs $\text{P}_1, \ldots, \text{P}_n$ au travers des requêtes suivantes :

- $\mathcal{F}_{\text{IC}}$ maintient une liste L_{sid} (initialement vide) contenant des triplets de chaînes de bits et un certain nombre d'ensembles $\text{C}_{\mathsf{key},\mathsf{sid}}, \text{M}_{\mathsf{key},\mathsf{sid}}$ (initialement vides).
- **À la réception d'une requête $(\mathsf{sid}, \mathbf{ENC}, \mathsf{key}, m)$ (avec $m \in \{0,1\}^k$) de la part d'un joueur P_i ou de $\mathcal{S}$, effectuer l'action suivante :**
  - S'il existe un triplet $(\mathsf{key}, m, \widetilde{c})$ pour un certain $\widetilde{c} \in \{0,1\}^k$ dans la liste L_{sid}, poser $c := \widetilde{c}$.
  - Dans le cas contraire, choisir uniformément c dans $\{0,1\}^k - \text{C}_{\mathsf{key},\mathsf{sid}}$, qui est l'ensemble des chiffrés non encore utilisés avec le paramètre key. Ensuite, enregistrer le triplet $(\mathsf{key}, m, c) \in \text{L}_{\mathsf{sid}}$ et poser $\text{C}_{\mathsf{key},\mathsf{sid}} \leftarrow \text{C}_{\mathsf{key},\mathsf{sid}} \cup \{c\}$.

  Une fois c fixé, renvoyer (sid, c) à l'entité qui en a fait la demande.
- **À la réception d'une requête $(\mathsf{sid}, \mathbf{DEC}, \mathsf{key}, c)$ (avec $c \in \{0,1\}^k$) de la part d'un joueur P_i ou de $\mathcal{S}$, effectuer l'action suivante :**
  - S'il existe un triplet $(\mathsf{key}, \widetilde{m}, c)$ pour un certain $\widetilde{m} \in \{0,1\}^k$ apaprtenant à L_{sid}, poser $m := \widetilde{m}$.
  - Dans le cas contraire, choisir uniformément m dans $\{0,1\}^k - \text{M}_{\mathsf{key},\mathsf{sid}}$, qui est l'ensemble des messages clairs non encore utilisés avec le paramètre key. Ensuite, enregistrer le triplet $(\mathsf{key}, m, c) \in \text{L}_{\mathsf{sid}}$ et poser $\text{M}_{\mathsf{key},\mathsf{sid}} \leftarrow \text{M}_{\mathsf{key},\mathsf{sid}} \cup \{m\}$.

  Une fois m fixé, renvoyer (sid, m) à l'entité qui en a fait la demande.

5.8 Les échanges de clefs dans le cadre UC

5.8.1 La fonctionnalité idéale

Une contribution importante des auteurs de [CHK+05] est la définition de la fonctionnalité idéale $\mathcal{F}_{pw\text{KE}}$ pour les échanges de clefs basés sur un mot de passe (voir figure 5.8). L'idée principale de cette fonctionnalité est la suivante : si aucun joueur n'est corrompu, alors ils obtiennent tous les deux la même clef de session uniformément distribuée, et l'adversaire n'apprend aucune information sur cette clef (excepté le fait qu'elle a effectivement été générée). En revanche, si un joueur est corrompu, ou si l'adversaire a bien deviné le mot de passe du joueur (la session étant alors marquée `compromised`), alors il obtient le droit de déterminer complètement la clef de session. On remarque que dès qu'un joueur est corrompu, l'adversaire apprend sa clef.

Les mots de passe. Ce n'est pas la fonctionnalité qui est chargée de donner aux joueurs leurs mots de passe. Ces derniers sont choisis par l'environnement qui les donne alors aux joueurs en entrée. Cela garantit la sécurité[7] même dans le cas où deux joueurs honnêtes exécutent le protocole avec deux mots de passe différents (éventuellement reliés), ce qui modélise, par exemple, le cas où un utilisateur fait une faute de frappe sur son mot de passe. Cette approche permet aussi de modéliser le cas dans lequel des utilisateurs peuvent utiliser le même mot de passe pour des protocoles différents et, plus généralement, le cas dans lequel les mots de passe sont choisis selon une distribution arbitraire (c'est-à-dire pas nécessairement la distribution uniforme). Enfin, cela garantit aussi la *forward secrecy* (voir la section 2.1 page 29).

[7] C'est-à-dire que la probabilité de casser le schéma est essentiellement la même que celle de deviner le mot de passe.

Fig. 5.6 – La fonctionnalité $\mathcal{F}_{\text{ITC}}$

La fonctionnalité $\mathcal{F}_{\text{ITC}}$ prend en entrée un paramètre de sécurité k, et interagit avec un adversaire $\mathcal{S}$ et un ensemble de joueurs $\text{P}_1,\ldots,\text{P}_n$ au travers des requêtes suivantes :

- $\mathcal{F}_{\text{ITC}}$ maintient une liste L_{sid} (initialement vide) contenant des quadruplets de chaînes de bits et un certain nombre d'ensembles $\text{C}_{\mathsf{key,label,sid}}, \text{M}_{\mathsf{key,label,sid}}$ (initialement vides).
- **À la réception d'une requête $(\mathsf{sid}, \mathbf{ENC}, \mathsf{key}, \mathsf{label}, m)$ (avec $m \in \{0,1\}^k$) de la part d'un joueur P_i ou de $\mathcal{S}$, effectuer l'action suivante :**
  - S'il existe un quadruplet $(\mathsf{key}, \mathsf{label}, m, \widetilde{c})$ pour un certain $\widetilde{c} \in \{0,1\}^k$ dans la liste L_{sid}, poser $c := \widetilde{c}$.
  - Dans le cas contraire, choisir uniformément c dans $\{0,1\}^k - \text{C}_{\mathsf{key,label,sid}}$, qui est l'ensemble des chiffrés non encore utilisés avec les paramètres key et label. Ensuite, enregistrer le quadruplet $(\mathsf{key}, \mathsf{label}, m, c) \in \text{L}_{\mathsf{sid}}$ et poser $\text{C}_{\mathsf{key,label,sid}} \leftarrow \text{C}_{\mathsf{key,label,sid}} \cup \{c\}$.

  Une fois c fixé, renvoyer (sid, c) à l'entité qui en a fait la demande.
- **À la réception d'une requête $(\mathsf{sid}, \mathbf{DEC}, \mathsf{key}, \mathsf{label}, c)$ (avec $c \in \{0,1\}^k$) de la part d'un joueur P_i ou de $\mathcal{S}$, effectuer l'action suivante :**
  - S'il existe un quadruplet $(\mathsf{key}, \mathsf{label}, \widetilde{m}, c)$ pour un certain $\widetilde{m} \in \{0,1\}^k$ appartenant à L_{sid}, poser $m := \widetilde{m}$.
  - Dans le cas contraire, choisir uniformément m dans $\{0,1\}^k - \text{M}_{\mathsf{key,label,sid}}$, qui est l'ensemble des messages clairs non encore utilisés avec les paramètres key et label. Ensuite, enregistrer le quadruplet $(\mathsf{key}, \mathsf{label}, m, c) \in \text{L}_{\mathsf{sid}}$ et poser $\text{M}_{\mathsf{key,label,sid}} \leftarrow \text{M}_{\mathsf{key,label,sid}} \cup \{m\}$.

  Une fois m fixé, renvoyer (sid, m) à l'entité qui en a fait la demande.

Fig. 5.7 – Fonctionnalité $\mathcal{F}_{\mathsf{CRS}}^{\mathcal{D}}$

La fonctionnalité $\mathcal{F}_{\text{CRS}}^{\mathcal{D}}$ est paramétrée par une distribution $\mathcal{D}$. Elle interagit avec un ensemble de joueurs et un adversaire de la façon suivante :

- Choisir une valeur $r \xleftarrow{\$} \mathcal{D}$.
- À réception d'une valeur $(\mathsf{CRS}, \mathsf{sid})$ de la part d'un joueur, envoyer $(\mathsf{CRS}, \mathsf{sid}, r)$ à ce joueur.

Définir une nouvelle session. La fonctionnalité commence par une étape d'initialisation durant laquelle elle se contente essentiellement d'attendre que chacun des deux joueurs notifie son intérêt à participer au protocole. Plus précisément, nous supposons que chaque joueur commence une nouvelle session du protocole avec l'entrée $(\texttt{NewSession}, \mathsf{sid}, \text{P}_i, \text{P}_j, \mathsf{pw}_i, \texttt{role})$, où P_i est l'identité du joueur, P_j celle du partenaire avec lequel il souhaite établir une communication, pw_i son mot de passe et `role` son rôle (client ou serveur). La session de ce joueur est alors notée `fresh`.

La modélisation des attaques par dictionnaire en ligne. L'adversaire a la possibilité de tenter de deviner le mot de passe d'un joueur (ce qui modélise la vulnérabilité des mots de passe, et revient à une attaque en ligne dans le monde réel : si l'adversaire a bien deviné le mot de passe, il doit réussir son usurpation d'identité) à l'aide des requêtes `TestPwd`. Toutes les sessions des joueurs sont initialement notées `fresh`. Un test correct de mot de passe s'obtient en notant `compromised` l'enregistrement correspondant, et un essai incorrect en le notant `interrupted`.

Une fois une clef établie, tous les enregistrements seront notés `completed`. Changer le statut `fresh` d'un enregistrement quand un test de mot de passe a lieu, ou qu'une clef (valide) est établie, a pour objectif de limiter le nombre de tests de mots de passe à au plus un par joueur.

La génération de la clef de session. C'est à nouveau l'adversaire qui demande à la fonctionnalité de générer une clef, ce qui lui permet de décider le moment exact auquel la clef doit être envoyée aux joueurs et, en particulier, cela lui permet de choisir le moment exact où les corruptions ou tentatives d'usurpation d'identité doivent avoir lieu ($\mathcal{S}$ ne peut tenter de deviner un mot de passe que lorsque la session d'un joueur est *fresh*, ce qui n'est plus le cas après une requête de clef). Si les deux joueurs ont encore des sessions `fresh` et le même mot de passe, ils reçoivent la même clef, uniformément distribuée. Si une session est `interrupted`, ils obtiennent des clefs aléatoires et choisies de manière indépendante. Enfin, si l'un des joueurs est `compromised` ou corrompu, l'adversaire peut choisir sa clef.

Fig. 5.8 – La fonctionnalité d'échange de clefs à base de mots de passe

La fonctionnalité $\mathcal{F}_{pw\text{KE}}$ est paramétrée par un paramètre de sécurité k. Elle interagit avec un adversaire $\mathcal{S}$ et un ensemble de joueurs $\mathrm{P}_1,\ldots,\mathrm{P}_n$ à travers les requêtes suivantes :

- **Sur réception de la requête (`NewSession`, sid, P_i, P_j, pw, `role`) du joueur P_i :** Envoyer (`NewSession`, sid, P_i, P_j, `role`) à $\mathcal{S}$. Si c'est la première requête `NewSession`, ou si c'est la deuxième requête `NewSession` et qu'il existe une entrée $(\mathrm{P}_j, \mathrm{P}_i, \mathsf{pw}')$, alors enregistrer l'entrée $(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw})$ et la marquer `fresh`.

- **Sur réception de la requête (`TestPwd`, sid, P_i, pw′) de la part de l'adversaire S :**
  S'il existe une entrée de la forme $(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw})$ qui soit `fresh`, alors : si $\mathsf{pw} = \mathsf{pw}'$, noter l'entrée `compromised` et envoyer à $\mathcal{S}$ la réponse « bon mot de passe ». Si $\mathsf{pw} \neq \mathsf{pw}'$, noter l'entrée `interrupted` et lui répondre « mauvais mot de passe ».

- **Sur réception de la requête (`NewKey`, sid, P_i, sk) de la part de l'adversaire S :**
  S'il existe une entrée de la forme $(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw})$, et si c'est la première requête `NewKey` pour P_i, alors :
  - Si l'entrée est `compromised`, ou si P_i ou P_j est corrompu, alors envoyer (sid, sk) au joueur P_i.
  - Si l'entrée est `fresh`, s'il y a une entrée $(\mathrm{P}_j, \mathrm{P}_i, \mathsf{pw}')$ avec $\mathsf{pw}' = \mathsf{pw}$, si une clef sk' a été envoyée à P_j, et si $(\mathrm{P}_j, \mathrm{P}_i, \mathsf{pw})$ était `fresh` à ce moment-là, alors envoyer (sid, sk′) à P_i.
  - Dans tous les autres cas, choisir une nouvelle clef aléatoire sk' de longueur k et envoyer (sid, sk′) à P_i.

  Dans tous les cas, marquer l'entrée $(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw})$ `completed`.

L'authentification du client. Les auteurs de [CHK+05] n'ont pas considéré les authentifications explicites. Nous avons ajouté dans [ACCP08] l'authentification du client, et avons prouvé dans ce cadre une variante d'un protocole EKE [BCP03b], dans le modèle de l'oracle aléatoire et du chiffrement idéal. La sécurité est garantie contre des adversaires adaptatifs (autorisés à corrompre un joueur à tout moment). Ce résultat est présenté dans le chapitre 6.

5.8.2 Une version UC du protocole KOY/GL ([CHK+05])

Nous avons décrit le protocole KOY/GL [GL03] dans la section 3.2. À un haut niveau, les joueurs échangent des mises en gage (chiffrements CCA) du mot de passe, sous la clef publique donnée dans la CRS. Ils calculent ensuite une clef de session en combinant des SPHF des

deux paires (mot de passe, chiffré). La sécurité de ce protocole est basée sur les propriétés de *smoothness* et pseudo-aléa de la SPHF. Mais, comme noté par Canetti *et al* dans [CHK+05], ce protocole ne vérifie pas (*a priori*) la sécurité UC.

Le principal problème qui l'en empêche est que le simulateur dans le monde idéal doit être capable d'extraire le mot de passe utilisé par l'adversaire. On pourrait se dire que comme le simulateur contrôle la CRS, il connaît toutes les clefs privées correspondant aux clefs publiques et il peut donc déchiffrer tous les messages envoyés par l'adversaire. Cela lui permet ainsi de récupérer son mot de passe, mais cela ne suffit pas. Dans le cas où l'adversaire commence à jouer (c'est-à-dire s'il joue le rôle du client), tout marche bien : le simulateur déchiffre le message chiffré généré par l'adversaire et peut alors récupérer le mot de passe qu'il a utilisé. Si l'adversaire a mal deviné le mot de passe, alors la *smoothness* des fonctions de hachage mène à des clefs de session aléatoires et indépendantes. Sinon, s'il a bien deviné, l'exécution peut continuer comme une exécution honnête (étant donné que le simulateur a appris le mot de passe à utiliser).

Maintenant, si le simulateur doit commencer le jeu, à la place du client, c'est-à-dire si l'adversaire joue le rôle d'un serveur, alors il doit envoyer un chiffré du mot de passe avant d'avoir vu quoi que ce soit venir de la part de l'adversaire. Comme décrit ci-dessus, il récupère le mot de passe utilisé par l'adversaire dès que ce dernier a envoyé ses valeurs, mais c'est trop tard. Si la prédiction de l'adversaire était erronée, il n'y a pas de problème grâce à la *smoothness*. Mais sinon, le simulateur est bloqué avec un chiffré faux et ne va pas pouvoir prédire la valeur de la clef de session.

Fig. 5.9 – Une version UC du protocole KOY/GL [CHK+05]

Serveur S		Client U
	$\xleftarrow{(c_0)}$	(U0) $c_0 = \mathcal{E}_{\mathsf{pk}}(\mathsf{pw}_\mathrm{I}; \mathrm{R}_0)$
(S1) $(\mathsf{VK}_\mathrm{J}, \mathsf{SK}_\mathrm{J}) \leftarrow \mathsf{SKG}$ $c_1 = \mathcal{E}_{\mathsf{pk}}(\mathsf{pw}_\mathrm{J}; \mathrm{R}_1)$	$\xrightarrow{(c_1, \mathsf{VK}_\mathrm{J})}$	
		(U2) $\mathsf{hk}_\mathrm{I} = \mathsf{HashKG}(\mathsf{pk})$ $\mathsf{hp}_\mathrm{I} = \mathsf{ProjKG}(\mathsf{hk}_\mathrm{I}; c_1)$
	$\xleftarrow{(c_2, \mathsf{hp}_\mathrm{I}, \mathrm{ZKP}(c_0 \approx c_2))}$	$c_2 = \mathcal{E}_{\mathsf{pk}}(\mathsf{pw}_\mathrm{I}; \mathrm{R}_2)$
(U3) $\mathsf{hk}_\mathrm{J} = \mathsf{HashKG}(\mathsf{pk})$ $\mathsf{hp}_\mathrm{J} = \mathsf{ProjKG}(\mathsf{hk}_\mathrm{J}; c_2)$ $\sigma_\mathrm{J} = \mathsf{Sign}(\mathsf{SK}_\mathrm{J}, (c_2, \mathsf{hp}_\mathrm{J}, \mathsf{hp}_\mathrm{I}))$ $\mathsf{sk}_\mathrm{J} = \mathsf{hashfun}(\mathsf{hk}_\mathrm{I}, \mathsf{pw}_\mathrm{J}, c_1)$ $\mathsf{sk}_\mathrm{I} = \mathsf{projfun}(\mathsf{hp}_\mathrm{J}, \mathsf{pw}_\mathrm{I}, c_2, \mathsf{VK}_\mathrm{I}, \mathrm{R}_\mathrm{J})$ $\mathsf{sk} = \mathsf{sk}_\mathrm{I} \circ \mathsf{sk}_\mathrm{J}$	$\xrightarrow{(\sigma_\mathrm{J}, \mathsf{hp}_\mathrm{J})}$	
		(S4) abandonne si $\mathsf{Ver}(\mathsf{VK}_\mathrm{J}, (c_2, \mathsf{hp}_\mathrm{J}, \mathsf{hp}_\mathrm{I}), \sigma_\mathrm{J}) = 0$ $\mathsf{sk}_\mathrm{I} = \mathsf{hashfun}(\mathsf{hk}_\mathrm{J}, \mathsf{pw}_\mathrm{I}, c_2)$ $\mathsf{sk}_\mathrm{J} = \mathsf{projfun}(\mathsf{hp}_\mathrm{I}, \mathsf{pw}_\mathrm{J}, c_1, \mathrm{R}_\mathrm{I})$ $\mathsf{sk} = \mathsf{sk}_\mathrm{I} \circ \mathsf{sk}_\mathrm{J}$

Pour résoudre ce problème, les auteurs de [CHK+05] ont ajouté un « pré-message » envoyé par le client, qui contient aussi un chiffré du mot de passe. Notons que c'est le client qui envoie ce pré-message pour respecter la convention que c'est toujours le client qui commence. Le reste du protocole est donc inversé par rapport à celui de KOY/GL. Sur la figure 5.9, nous

avons cette fois placé le client à droite afin que le protocole ressemble au maximum à celui de KOY/GL. Le serveur envoie alors son propre chiffré, et finalement le client envoie un dernier chiffré (cette fois, le simulateur est capable d'utiliser le mot de passe correct, récupéré à partir de la valeur envoyée par l'adversaire), ainsi qu'une preuve *zero-knowledge* assurant que les deux chiffrés qu'il a envoyés sont consistants et chiffrent le même mot de passe. Le premier message n'est alors plus jamais utilisé dans le reste du protocole. Cela résout le problème étant donné que, d'une part, le simulateur est bien entendu capable d'envoyer une preuve valide d'une assertion fausse, et d'autre part, le premier chiffré ne sera jamais utilisé ensuite. Le protocole proposé, et rappelé sur la figure 5.9, est sûr contre les adversaires passifs uniquement. Pour simplifier la comparaison avec le protocole KOY/GL, le client est désormais représenté sur la colonne de droite.

5.8.3 Le cas des groupes

La fonctionnalité idéale des échanges de clefs ([KS05]). Katz et Shin [KS05] ont défini la fonctionnalité pour les échanges de clefs pour les groupes (sans mot de passe), et prouvé que la nouvelle notion de sécurité était plus forte que la notion usuelle de Bresson *et al.* [BCPQ01]. En outre, ils ont formalisé un cadre de corruption forte, où les joueurs honnêtes peuvent être compromis à tout moment pendant l'exécution du protocole, et révéler leurs secrets à long et court terme (l'état interne en entier). En outre, ils supposent les identifiants (de session et de groupe) fournis par l'environnement, et ils ont donné une définition formelle de la « confirmation de clef ».

La fonctionnalité idéale des échanges de clefs à base de mots de passe ([ACCP09]). Katz et Shin ont défini une fonctionnalité dans laquelle ils permettent à l'adversaire de choisir la clef dès lors qu'il a corrompu au moins un joueur. Nous verrons dans le chapitre 7 une autre définition (appelée la *contributory*, que nous avons présentée dans [ACCP09]) qui restreint l'adversaire à ne pouvoir biaiser la clef que lorsqu'il a corrompu un nombre suffisant de joueurs. Nous définissons dans le même article la fonctionnalité pour les échanges basés sur un mot de passe, combinant les travaux de Canetti *et al.* [CHK$^+$05], Katz et Shin [KS05], et Barak *et al.* [BCL$^+$05].

La fonctionnalité de répartition ([BCL$^+$05]). Sans aucun mécanisme d'authentification et sur des canaux de communication contrôlés par l'adversaire, ce dernier peut toujours partitionner les joueurs en sous-groupes disjoints et exécuter des sessions indépendantes du protocole avec chaque sous-groupe, en jouant le rôle des autres joueurs. Une telle attaque est inévitable puisque les joueurs ne peuvent pas distinguer le cas dans lequel ils interagissent entre eux du cas dans lequel ils interagissent avec l'adversaire. Nous appelons ceci une *corruption implicite*, ce qui signifie que l'adversaire usurpe l'identité d'un joueur (ou de plusieurs joueurs) depuis le tout début, avant même l'initiation de la génération de clef.

Les auteurs de [BCL$^+$05] ont réglé ce problème en proposant un nouveau cadre basé sur les fonctionnalités de répartition (*split functionalities*), ce qui garantit que cette attaque est la seule possible à l'adversaire. Plus généralement, les auteurs ont considéré des protocoles pour de la *multiparty computation* générique en l'absence de canaux authentifiés. En particulier, ils ont donné une solution générale et conceptuellement simple (bien qu'inefficace) à un certain nombre de problèmes incluant les échanges de clefs de groupe à base de mots de passe.

Cette fonctionnalité de répartition est une construction générique basée sur une fonctionnalité idéale : sa description peut être trouvée sur la figure 5.10 . Dans l'étape d'initialisation, l'adversaire choisit adaptativement des sous-ensembles disjoints de joueurs honnêtes (avec un identifiant de session unique fixé pour toute la durée du protocole). Plus précisément, le protocole commence avec un identifiant de session sid. Ensuite, l'étape d'initialisation génère des valeurs aléatoires qui, combinées aléatoirement et avec sid, créent le nouvelidentifiant de session sid', partagé par tous les joueurs qui ont reçu les mêmes valeurs – c'est-à-dire les joueurs des sous-ensembles disjoints. Le point important ici est que les sous-ensembles créent une *partition* des joueurs, empêchant ainsi toute communication entre les sous-ensembles. Pendant le

calcul, chaque sous-ensemble H active une instance séparée de la fonctionnalité $\mathcal{F}$. Toutes les instances de fonctionnalité sont indépendantes : les exécutions du protocole pour chaque sous-ensemble H ne peuvent être reliées que dans la manière dont l'adversaire choisit les entrées des joueurs qu'il contrôle. Les joueurs $\mathrm{P}_i \in \mathrm{H}$ apportent leurs propres entrées et reçoivent leurs propres sorties (voir le premier point de « calcul » dans la figure 5.10), tandis que l'adversaire joue le rôle de tous les joueurs $\mathrm{P}_j \notin \mathrm{H}$ (voir le deuxième point).

Dans le cas particulier des protocoles d'échange de clefs à base de mots de passe, l'adversaire envoie alors une requête `NewSession` au nom de tels joueurs implicitement corrompus, qui ne deviennent jamais vraiment corrompus mais ont toujours été contrôlés par l'adversaire. Cette situation est donc modélisée dans le monde idéal par les fonctionnalités de répartition $s\mathcal{F}$ qui font appel à une ou plusieurs instances des fonctionnalités normales $\mathcal{F}$ sur des ensembles disjoints de (vrais) joueurs.

Fig. 5.10 – La fonctionnalité de répartition $s\mathcal{F}$

Étant donné une fonctionnalité $\mathcal{F}$, la fonctionnalité de répartition $s\mathcal{F}$ peut être décrite comme suit :

Initialisation :

- À réception de $(\texttt{Init}, \mathsf{sid})$ de la part du joueur P_i, envoyer $(\texttt{Init}, \mathsf{sid}, \mathrm{P}_i)$ à l'adversaire.
- À réception d'un message $(\texttt{Init}, \mathsf{sid}, \mathrm{P}_i, \mathrm{H}, \mathsf{sid}_\mathrm{H})$ de la part de $\mathcal{A}$, où H est un ensemble d'identités de joueurs, vérifier que P_i a déjà envoyé $(\texttt{Init}, \mathsf{sid})$ et que pour tous les enregistrements $(\mathrm{H}', \mathsf{sid}_{\mathrm{H}'})$, soit $\mathrm{H} = \mathrm{H}'$ et $\mathsf{sid}_\mathrm{H} = \mathsf{sid}_{\mathrm{H}'}$, soit H et H' sont disjoints et $\mathsf{sid}_\mathrm{H} \neq \mathsf{sid}_{\mathrm{H}'}$. Si c'est le cas, enregistrer la paire $(\mathrm{H}, \mathsf{sid}_\mathrm{H})$, envoyer $(\texttt{Init}, \mathsf{sid}, \mathsf{sid}_\mathrm{H})$ à P_i, et invoquer une nouvelle fonctionnalité $(\mathcal{F}, \mathsf{sid}_\mathrm{H})$ appelée $\mathcal{F}_\mathrm{H}$ et interagissant avec un ensemble de joueurs honnêtes H.

Calcul :

- À réception de $(\texttt{Input}, \mathsf{sid}, m)$ de la part du joueur P_i, trouver l'ensemble H tel que $\mathrm{P}_i \in \mathrm{H}$ et transmettre m à $\mathcal{F}_\mathrm{H}$.
- À réception de $(\texttt{Input}, \mathsf{sid}, \mathrm{P}_j, \mathrm{H}, m)$ de la part de $\mathcal{A}$, tel que $\mathrm{P}_j \notin \mathrm{H}$, transmettre m à $\mathcal{F}_\mathrm{H}$ comme s'il provenait de P_j.
- Quand $\mathcal{F}_\mathrm{H}$ génère une sortie m pour le joueur $\mathrm{P}_i \in \mathrm{H}$, envoyer m à P_i. Si la sortie est pour $\mathrm{P}_j \notin \mathrm{H}$ ou pour l'adversaire, envoyer m à l'adversaire.

5.9 Conventions de terminologie pour la suite de ce mémoire

Nous avons vu dans cette partie les différents cadres de sécurité dans lesquels étudier les protocoles. On peut remarquer deux différents axes :

- d'une part, le caractère privé de la clef de session peut être modélisé via soit la « sécurité sémantique » [BR94], définie par un jeu de sécurité, soit la simulation, c'est-à-dire l'indistinguabilité entre le protocole et son homologue idéal [BMP00, Can01, CK01]. On parlera ici de *cadre de sécurité* : le *cadre standard* pour le premier cas, et le *cadre UC* pour la sécurité décrite dans ce chapitre. Les autres cadres basés sur l'indistinguabilité [BMP00] ne seront pas utilisés.
- d'autre part, les modèles sur lesquels repose le protocole. On parlera dans ce cas de *modèles idéaux* pour l'oracle aléatoire et le chiffrement (éventuellement étiqueté) idéal, et du *modèle standard* pour les protocoles n'en faisant pas usage.

Cette distinction permettra l'absence de confusion quant au sens à donner au mot *standard*.

Deuxième partie

Protocoles d'échanges de clefs dans le cadre UC et des modèles idéaux

Chapitre 6

Protocole pour deux joueurs

Comme on l'a vu dans la partie précédente, la plupart des échanges de clefs basés sur des mots de passe ont soit des preuves basées sur un jeu de sécurité dans le cadre de Bellare, Pointcheval, and Rogaway (BPR), soit des preuves par simulation dans le cadre de Boyko, MacKenzie, and Patel (BMP). Bien que ces cadres procurent un niveau de sécurité suffisant pour la plupart des applications, ils ne parviennent pas à considérer des scénarios réalistes tel que celui de participants exécutant le protocole avec des mots de passe différents mais éventuellement reliés. Pour pallier ce manque, nous avons vu que Canetti *et al.* [CHK$^+$05] ont proposé un nouveau cadre de sécurité dans le cadre de la composabilité universelle (UC, décrite dans le chapitre 5 page 58) qui ne fait aucune hypothèse sur la distribution des mots de passe utilisés par les participants au protocole. Ils ont aussi proposé un nouveau protocole (décrit dans la section 5.8.2 page 83) basé sur les schémas de Katz, Ostrovsky, and Yung [KOY01] et de Gennaro et Lindell [GL03] et l'ont prouvé sûr dans ce nouveau cadre contre des adversaires statiques sur des hypothèses standard. Ce protocole n'est hélas pas aussi efficace que les protocoles existants dans les cadres BPR et BMP (par exemple, [BCP03b, AP05, KOY01, Mac02]), et ce problème peut significativement limiter son application en pratique.

Compte-tenu de cette limitation, une question naturelle est alors de se demander si certains des protocoles existants, prouvés sûrs dans les cadres BPR ou BMP, le restent dans ce nouveau cadre [CHK$^+$05], et nous avons répondu à cette question par l'affirmative dans un article [ACCP08] publié à la conférence RSA en 2008. Plus précisément, nous montrons que le protocole de Bresson, Chevassut, et Pointcheval (BCP) à CCS 2003 [BCP03b] est aussi sûr dans le nouveau cadre UC. La preuve de sécurité, dans les modèles de l'oracle aléatoire et du chiffrement idéal, fonctionne même en présence d'adversaires adaptatifs, capables de corrompre des joueurs à n'importe quel moment et d'apprendre leurs états internes. C'est la première fois qu'un niveau de sécurité aussi fort est obtenu dans le scénario basé sur des mots de passe, même si ce résultat n'est pas très surprenant étant donné l'utilisation de l'oracle aléatoire et du chiffrement idéal dans la preuve de sécurité.

6.1 Définition de sécurité

6.1.1 La fonctionnalité d'échange de clefs à base de mots de passe avec authentification du client

Nous présentons dans cette section la fonctionnalité $\mathcal{F}_{pw\mathrm{KE}}^{\mathrm{CA}}$ (pour *password-based key-exchange with client authentication*), décrite sur la figure 6.1. Elle est très proche de la définition universellement composable des échanges de clefs basés sur un mot de passe [CHK+05] (voir page 81), et nous y incorporons l'authentification du client (voir définition page 29), qui garantit non seulement aux deux joueurs que personne d'autre ne connaît le secret commun, mais aussi au serveur qu'il partage effectivement un secret commun avec le client. Il aurait été plus facile de modéliser l'authentification mutuelle, mais celle du client est suffisante dans la plupart des cas et procure souvent des protocoles plus efficaces.

La fonctionnalité utilise une liste L pour gérer les joueurs. Les éléments de cette liste sont de la forme [(clef),(valeur)], la clef contenant dans l'ordre l'identifiant de la session, le nom du joueur, le nom de son partenaire, le mot de passe qu'il utilise et son rôle (`client` ou `server`). Ces éléments sont fixes. La valeur contient le statut de l'instance du joueur (`fresh`, `compromised`, `interrupted`, `completed` ou `error`) et la clef qui lui a été attribuée (`sk` ou `error`). Ces valeurs sont initialement vides et les mots-clef sont définis dans la fonctionnalité de Canetti *et al.* page 81.

Les requêtes `NewSession` et `TestPwd` sont définies comme dans [CHK+05] (c'est en particulier l'environnement qui fournit les mots de passe), mais nous introduisons l'authentification du client dans la façon dont la fonctionnalité répond aux requêtes `NewKey`. Tout d'abord, dans la définition de $\mathcal{F}_{pw\mathrm{KE}}^{\mathrm{CA}}$, le serveur reçoit une erreur si les joueurs ne remplissent pas toutes les conditions pour recevoir la même clef, choisie au hasard. Nous aurions pu choisir d'envoyer au serveur une paire constituée d'une clef choisie indépendamment de celle du client, et d'une indication le prévenant que le protocole a échoué, mais nous avons préféré garder la fonctionnalité aussi directe que possible.

Ensuite, nous devons nous préoccuper de l'ordre des requêtes `NewKey` pour gérer l'authentification du client. Plus précisément, dans la fonctionnalité originelle de Canetti *et al.*, si le serveur pose la première requête, il est impossible de lui répondre, car l'adversaire peut éventuellement prendre le contrôle du client ou le corrompre ultérieurement. Par exemple, si la session est `fresh` pour les deux joueurs et que le serveur est le seul à avoir reçu sa clef, la session du client peut devenir `compromised` ou `interrupted` après que le serveur a reçu sa clef, alors que la fonctionnalité aurait dû être capable de déterminer si le serveur aurait dû recevoir une clef ou un message d'erreur. Nous avons résolu ce problème en obligeant l'adversaire à poser la requête pour le serveur après celle pour le client. Ce n'est pas une restriction forte, étant donné que cette situation apparaît fréquemment dans des protocoles réels, et en particulier dans celui que l'on étudie : le serveur doit être sûr que le client est honnête avant de générer la clef de session.

Ainsi, si l'adversaire demande la clef pour un client, tout se déroule comme avant, sauf que l'étiquette `completed` va désormais avoir son rôle : son objectif est de prévenir, quand l'adversaire demande la clef pour le serveur, que le client correspondant a bien déjà obtenu sa clef. Si c'est pour le serveur que l'adversaire demande la clef, le serveur reçoit un message d'erreur dans les cas faciles d'échec (sessions `interrupted` ou `compromised`, joueurs corrompus – si les mots de passe sont différents dans les deux derniers cas). Si la session est `fresh` et que le client correspondant n'a pas encore reçu la clef, on repousse simplement la requête de l'adversaire jusqu'à ce que le client ait reçu sa clef[1] . Quand le client a reçu sa clef, le serveur reçoit la même s'ils ont le même mot de passe, et un message d'erreur sinon.

[1]De manière très formelle, ceci a pour conséquence (dans ce chapitre comme dans les suivants) que nous ne travaillons pas exactement dans le cadre UC, puisque nous n'imposons aucune restriction sur le nombre de requêtes *ignorées*. Cependant, l'attaquant étant polynomial, ce nombre sera donc restreint lui aussi à être polynomial.

Enfin, nous déclarons la session `ready` lorsque deux joueurs se sont manifestés à travers les requêtes `NewSession`, l'un ayant un rôle de client, l'autre de serveur, et les deux évoquant l'autre comme partenaire. À l'inverse de la fonctionnalité originelle, cela permet de restreindre les requêtes de demandes de clefs aux seules sessions dans lesquelles il y a bien deux interlocuteurs ayant des rôles opposés (un client et un serveur).

6.2 Notre schéma

6.2.1 Description du protocole

Le protocole présenté sur la figure 6.2 est basé sur celui de [BCP03b], avec deux légères différences : dans la définition standard de sécurité de Bellare *et al.* [BPR00], l'identifiant de session est obtenu à la fin de l'exécution comme la concaténation des valeurs aléatoires envoyées par les joueurs ; en particulier, il est unique. À l'inverse, dans le cadre de composabilité universelle [Can01], ces identifiants sont déterminés de façon unique à l'avance, avant le début du protocole, ce qui est reflété ici. De même, afin de vérifier la définition de la fonctionnalité, le serveur reçoit un message d'erreur en cas d'échec pour garantir l'authentification du client.

6.2.2 Théorème de sécurité

Nous considérons ici le théorème de composabilité universelle dans sa version à états joints (voir page 61). Soit $\widehat{\mathcal{F}}^{\mathrm{CA}}_{pw\mathrm{KE}}$ l'extension multi-session de $\mathcal{F}^{\mathrm{CA}}_{pw\mathrm{KE}}$ et soient $\mathcal{F}_{\mathrm{RO}}$ et $\mathcal{F}_{\mathrm{IC}}$ les fonctionnalités idéales qui donnent un oracle aléatoire et un chiffrement idéal à tous les joueurs (voir page 79). Il est à noter que seules ces deux fonctionnalités sont dans l'état joint.

Théorème 1. *Le protocole ci-dessus UC-réalise la fonctionnalité $\widehat{\mathcal{F}}^{\mathrm{CA}}_{pw\mathrm{KE}}$ de façon sûre dans le modèle hybride $(\mathcal{F}_{\mathrm{RO}}, \mathcal{F}_{\mathrm{IC}})$, en présence d'un adversaire adaptatif.*

6.3 Preuve du théorème 1

6.3.1 Description de la preuve

Afin de prouver que le protocole UC-réalise la fonctionnalité $\mathcal{F}^{\mathrm{CA}}_{pw\mathrm{KE}}$, on doit montrer que, pour tout adversaire $\mathcal{A}$, on est capable de construire un simulateur $\mathcal{S}$ tel que, pour tout environnement $\mathcal{Z}$, l'interaction, d'une part entre l'environnement, les joueurs (appelons-les Alice et Bob) et l'adversaire (le monde réel), et d'autre part entre l'environnement, la fonctionnalité idéale et le simulateur (le monde idéal) soient indistinguables pour l'environnement.

On définit dans cette preuve une suite de jeux depuis l'exécution réelle du protocole $\mathbf{G}_0$ jusqu'au jeu $\mathbf{G}_7$, que l'on prouve être indistinguable du jeu dans le monde idéal.

Comme il faut gérer les corruptions adaptatives, on considère différents cas en fonction du nombre de corruptions qui ont déjà eu lieu. $\mathbf{G}_0$ est le jeu réel. Dans $\mathbf{G}_1$, on commence par expliquer comment $\mathcal{S}$ simule le chiffrement idéal et l'oracle aléatoire. Ensuite, dans $\mathbf{G}_2$, on se débarrasse des situations dans lesquelles l'adversaire gagne par hasard. Le cas passif, dans lequel aucune corruption n'a lieu avant la fin du protocole, est géré dans $\mathbf{G}_3$. On explique alors la simulation complète du client dans $\mathbf{G}_4$, quelles que soient les corruptions. Pour le serveur, on divise cela en deux étapes : d'abord, on montre dans $\mathbf{G}_5$ comment simuler la dernière étape du protocole, et ensuite, on le simule depuis le début dans $\mathbf{G}_6$. $\mathbf{G}_7$ résume la situation, et on montre qu'il est indistinguable du jeu réel.

Notons que ces jeux sont séquentiels et s'appuient les uns sur les autres. Lorsque l'on dit qu'un jeu considère un cas spécifique, il faut comprendre que dans tous les autres cas, la simulation est effectuée comme dans le jeu précédent.

Fig. 6.1 – Fonctionnalité $\mathcal{F}_{pw\mathrm{KE}}^{\mathrm{CA}}$

La fonctionnalité $\mathcal{F}_{pw\mathrm{KE}}^{\mathrm{CA}}$ utilise un paramètre de sécurité k et correspond à l'identifiant de session sid. Elle maintient une liste L, initialement vide, contenant deux valeurs de la forme $[(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw}_i, \mathtt{client}), (\mathtt{statut}, \mathtt{clef})]$, où `statut` peut être `fresh`, `compromised`, `interrupted`, `completed` ou `error` et la clef qui a été attribuée au joueur $\perp$, sk ou `error`. La fonctionnalité interagit avec un adversaire $\mathcal{S}$ et un ensemble de joueurs $\mathrm{P}_1,\ldots,\mathrm{P}_n$ à travers les requêtes suivantes :

À réception d'une requête $(\mathtt{NewSession}, \mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j, \mathsf{pw}, \mathtt{role})$ de la part de P_i :

- Envoyer $(\mathtt{NewSession}, \mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j, \mathtt{role})$ à $\mathcal{S}$.
- Si c'est la première requête `NewSession`, ou si c'est la seconde et qu'il existe déjà une valeur $[(\mathrm{P}_j, \mathrm{P}_i, \mathsf{pw}', \overline{\mathtt{role}}), (\mathtt{fresh}, \perp)]$ dans la liste L, alors enregistrer l'entrée $[(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw}, \mathtt{role}), (\mathtt{fresh}, \perp)]$ dans L. Dans ce dernier cas, enregistrer $(\mathsf{sid}, \mathtt{ready})$ et l'envoyer à $\mathcal{S}$.

À réception d'une requête $(\mathtt{TestPwd}, \mathsf{sid}, \mathrm{P}_i, \mathsf{pw}')$ de la part de l'adversaire $\mathcal{S}$:

S'il existe un enregistrement de la forme $(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw}, \mathtt{role}) \in \mathrm{L}$ dont le statut soit `fresh`, alors :

- Si $\mathsf{pw} = \mathsf{pw}'$, noter l'enregistrement `compromised` et dire à l'adversaire qu'il a bien deviné le mot de passe.
- Si $\mathsf{pw} \neq \mathsf{pw}'$, noter l'enregistrement `interrupted` et dire à l'adversaire qu'il a mal deviné le mot de passe.

À réception d'une requête $(\mathtt{NewKey}, \mathsf{sid}, \mathrm{P}_i, \mathsf{sk})$ de la part de $\mathcal{S}$, où $|\mathsf{sk}| = k$:

S'il existe une valeur de la forme $(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw}, \mathtt{role}) \in \mathrm{L}$ ainsi qu'une valeur $(\mathsf{sid}, \mathtt{ready})$, et si c'est la première requête `NewKey` pour P_i (autrement dit si son statut est tout sauf `completed`), alors :

Si `role`=`client` :

- Si le statut est `compromised`, ou si l'un des deux joueurs P_i ou P_j est corrompu, alors envoyer $(\mathsf{sid}, \mathsf{sk})$ à P_i, changer le statut en `completed` et compléter l'enregistrement avec sk.
- Sinon, si le statut est `fresh` ou `interrupted`, choisir une clef aléatoire sk' de longueur k et envoyer $(\mathsf{sid}, \mathsf{sk}')$ à P_i. Changer le statut en `completed` et compléter l'enregistrement avec sk.

Si `role`=`server` :

- Si le statut est `compromised`, si l'un des deux joueurs P_i ou P_j est corrompu, et s'il y a deux enregistrements de la forme $(\mathrm{P}_i, \mathrm{P}_j, \mathsf{pw}, \mathtt{server})$ et $(\mathrm{P}_j, \mathrm{P}_i, \mathsf{pw}, \mathtt{client})$, poser $s = \mathsf{sk}$. Sinon, si le statut est `fresh` et s'il existe un élément de la forme $[(\mathrm{P}_j, \mathrm{P}_i, \mathsf{pw}, \mathtt{client}), (\mathtt{completed}, \mathsf{sk}')]$, poser $s = \mathsf{sk}'$.
  - Si $\mathsf{pw} = \mathsf{pw}'$, envoyer (sid, s) à P_i, changer son statut en `completed` dans L et compléter avec s.
  - Si $\mathsf{pw} \neq \mathsf{pw}'$, envoyer $(\mathsf{sid}, \mathtt{error})$ à P_i, changer son statut en `error` dans L et compléter avec `error`.
- Si le statut est `fresh` et s'il n'existe aucun enregistrement dans L de la forme $[(\mathrm{P}_j, \mathrm{P}_i, \mathsf{pw}', \mathtt{client}), (\mathtt{completed}, \mathtt{fresh})]$, alors ne rien faire ;
- Si le statut est `interrupted`, alors envoyer $(\mathsf{sid}, \mathtt{error})$ au joueur P_i, changer son statut en `error` dans L et compléter avec `error`.

Fig. 6.2 – Échange de clefs basé sur un mot de passe pour deux joueurs, avec authentification du client

Client U		Serveur S
$x \xleftarrow{R} [\![1 ; q-1]\!]$		$y \xleftarrow{R} [\![1 ; q-1]\!]$
(U1) $X \leftarrow g^x$	$\xrightarrow{U,X}$	
		(S2) $Y \leftarrow g^y$ $Y^* \leftarrow \mathcal{E}_{ssid\|\|pw}(Y)$
	$\xleftarrow{S,Y^*}$	$K_S \leftarrow X^y$
(U3) $Y = \mathcal{D}_{ssid\|\|pw}(Y^*)$ $K_U \leftarrow Y^x$ $Auth \leftarrow \mathcal{H}_1(ssid\|\|U\|\|S\|\|X\|\|Y\|\|K_U)$ $sk_U \leftarrow \mathcal{H}_0(ssid\|\|U\|\|S\|\|X\|\|Y\|\|K_U)$ `completed`	$\xrightarrow{Auth}$	
		(S4) if $(Auth = \mathcal{H}_1(ssid\|\|U\|\|S\|\|X\|\|Y\|\|K_S))$ then $sk_S \leftarrow \mathcal{H}_0(ssid\|\|U\|\|S\|\|X\|\|Y\|\|K_S)$ `completed` else `error`

6.3.2 Requêtes hybrides et origine des messages

Commençons par décrire deux requêtes *hybrides* utilisées dans les jeux à venir. La requête `GoodPwd` vérifie que le mot de passe d'un certain joueur est bien celui auquel nous pensons. La requête `SamePwd` vérifie que les joueurs partagent bien le même mot de passe, sans le révéler. Dans certains jeux, le simulateur a un accès total aux joueurs. Dans ce cas, une requête `GoodPwd` (ou `SamePwd`) peut facilement être implémentée en laissant simplement le simulateur consulter les mots de passe. Quand les joueurs sont entièrement simulés, $\mathcal{S}$ remplacera les requêtes ci-dessus par des `TestPwd` et des `NewKey`, respectivement.

Suivant [CHK+05], un message est dit *oracle-generated* (généré par un oracle) s'il a été envoyé par un joueur honnête et s'il arrive sans aucune modification au joueur auquel il a été envoyé. Il est dit *non oracle-generated* sinon, c'est-à-dire soit s'il a été envoyé par un joueur honnête et modifié par l'adversaire, soit s'il a été envoyé par un joueur corrompu ou contrôlé par l'attaquant : dans ces deux cas, on dit que l'émetteur est un joueur *attaqué*. Ainsi, la simulation contrôle les aléas des messages *oracle-generated* tandis que l'adversaire contrôle les aléas des messages non *oracle-generated*.

6.3.3 Preuve d'indistinguabilité

Jeu G_0: Jeu réel. G_0 est le jeu réel, dans les modèles de l'oracle aléatoire et du chiffrement idéal.

Jeu G_1: Simulation des oracles. On modifie le jeu précédent en simulant les oracles de hachage, de chiffrement et de déchiffrement de façon usuelle et naturelle.

Pour le chiffrement idéal, on autorise le simulateur à maintenir une liste Λ_ε d'entrées de la forme (question, réponse) : on suppose que ℓ_ε est le nombre de bits de la réponse à une requête de chiffrement, q_ε le nombre de ces requêtes, et de même avec ℓ_d et $q_\mathcal{D}$ pour le déchiffrement. Une telle liste est utilisée par $\mathcal{S}$ pour pouvoir donner des réponses cohérentes avec les contraintes suivantes. D'abord, si le simulateur reçoit deux fois la même question

(chiffrement ou déchiffrement) pour le même mot de passe, il doit donner deux fois la même réponse. Ensuite, il doit s'assurer que le schéma de chiffrement simulé (pour chaque mot de passe) est bien une permutation. Enfin, afin de lui permettre d'extraire plus tard le mot de passe utilisé pour le chiffrement de Y^* dans le premier message, il ne doit pas y avoir deux entrées (question, réponse) avec un chiffré identique et des mots de passe différents. Plus précisément, Λ_ε est composée de deux sous-listes :

$$\Lambda_\varepsilon = \{(\mathsf{ssid}, \mathsf{pw}, Y, \alpha, \mathcal{E}, Y^*)\} \cup \{(\mathsf{ssid}, \mathsf{pw}, Y, \alpha, \mathcal{D}, Y^*)\}$$

La première (resp. seconde) sous-liste indique que l'élément Y (resp. Y^*) a été chiffré (« $\mathcal{E}$ ») (resp. déchiffré (« $\mathcal{D}$ »)) pour produire le chiffré Y^* (resp. le clair Y) via un schéma de chiffrement symétrique qui utilise la clef $\mathsf{ssid}\|\mathsf{pw}$. Le rôle de α sera expliqué plus loin. Le simulateur gère la liste à travers les règles suivantes :

- Pour une requête de chiffrement $\mathcal{E}_{\mathsf{ssid}\|\mathsf{pw}}(Y)$ telle que $(\mathsf{ssid}, \mathsf{pw}, Y, *, *, Y^*)$ apparaît dans la liste Λ_ε, la réponse est Y^*. Sinon, choisir un élément aléatoire $Y^* \in G^* = G \setminus \{1\}$. Si un élément $(*, *, *, *, *, Y^*)$ appartient déjà à la liste Λ_ε, abandonner, sinon ajouter $(\mathsf{ssid}, \mathsf{pw}, Y, \perp, \mathcal{E}, Y^*)$ à la liste.
- Pour une requête de déchiffrement $\mathcal{D}_{\mathsf{ssid}\|\mathsf{pw}}(Y^*)$ telle que $(*, \mathsf{pw}, Y, *, *, Y^*)$ apparaît dans Λ_ε, la réponse est Y. Sinon, choisir un élement aléatoire $\varphi \in {\mathbb{Z}_q}^*$ et évaluer la réponse $Y = g^\varphi$.

Le cas d'abandon sera utile plus tard dans la preuve : quand le simulateur verra un chiffré Y^*, il saura qu'il ne peut avoir été obtenu comme résultat d'un chiffrement qu'avec un unique mot de passe.

En outre, le simulateur maintient aussi une liste $\Lambda_\mathcal{H}$ (les réponses ont pour taille $\ell_{\mathcal{H}_n}$ et le nombre de requêtes est q_h). Cette liste est utilisée pour gérer correctement les requêtes des oracles aléatoires $\mathcal{H}_0$ et $\mathcal{H}_1$ en évitant les collisions. En particulier, le simulateur met à jour $\Lambda_\mathcal{H}$ en utilisant la règle générale suivante (n vaut 0 ou 1) :

- Pour une requête de hachage $\mathcal{H}_n(q)$ telle que (n, q, r) apparaît dans $\Lambda_\mathcal{H}$, la réponse est r. Sinon, choisir un élément aléatoire $r \in \{0,1\}^{\ell_{\mathcal{H}_n}}$. Si $(n, *, r)$ appartient déjà à la liste $\Lambda_\mathcal{H}$, abandonner, sinon ajouter (n, q, r) à la liste.

Le simulateur utilisera plus tard deux oracles privés $\mathcal{H}'_0$ et $\mathcal{H}'_1$, qu'il gèrera de la même manière.

Les cas d'abandon pour le chiffrement et le hachage étant exactement le paradoxe des anniversaires (de probabilités $(q_\varepsilon)^2/2^{\ell_\varepsilon}$ et $(q_h)^2/2^{\ell_{\mathcal{H}_n}}$), $\mathbf{G}_1$ est indistinguable du jeu réel $\mathbf{G}_0$.

Jeu $\mathbf{G}_2$: Cas où l'adversaire gagne par hasard. Ce jeu est pratiquement le même que le précédent. La seule différence est que l'on autorise le simulateur à abandonner si l'adversaire parvient à deviner Auth sans avoir posé la requête correspondante à l'oracle. Cet événement apparaît avec probabilité négligeable, donc $\mathbf{G}_2$ et $\mathbf{G}_1$ sont indistinguables.

Jeu $\mathbf{G}_3$: Cas passif : aucune corruption avant l'étape 4. On s'occupe dans ce jeu du cas passif dans lequel aucune corruption n'a lieu avant l'étape 4. Le simulateur a un contrôle partiel sur les joueurs engagés dans le protocole, c'est-à-dire qu'on suppose qu'il a accès à chaque joueur en tant qu'oracle pour les trois premiers tours du protocole. On lui demande alors de simuler complètement leur comportement en S4 si aucune corruption n'a eu lieu. Plus précisément, on considère deux cas distincts. Si aucune corruption n'a eu lieu avant S4, on demande à $\mathcal{S}$ de simuler l'exécution du protocole au nom des deux joueurs. Si, au contraire, un joueur a été corrompu avant de commencer S4, alors $\mathcal{S}$ ne fait rien. Notons que dans chacun des cas, on l'autorise toujours à connaître les mots de passe des deux joueurs.

Si au début de S4, les deux joueurs sont encore honnêtes et que les messages ont été *oracle-generated*, alors le simulateur pose une requête `SamePwd`. Comme on suppose que $\mathcal{S}$ connaît toujours les deux mots de passe, cela revient à vérifier que ce sont les mêmes.

On distingue alors à nouveau deux cas. Si les mots de passe sont les mêmes, $\mathcal{S}$ choisit une clef aléatoire K dans l'espace des clefs et la « donne » aux deux joueurs. Sinon, il choisit une clef aléatoire et la donne au client tandis que le serveur reçoit un message d'erreur.

Si les deux joueurs ont le même mot de passe, une telle stratégie rend le jeu indistinguable du précédent. À l'inverse, s'ils ne partagent pas le même mot de passe, une exécution du protocole dans ce jeu est indistinguable d'une exécution réelle au risque de collisions près, qui est négligeable. En effet, si deux joueurs ne partagent pas le même mot de passe, le serveur va calculer un authentifiant Auth différent, obtenant ainsi un message d'erreur, sauf avec probabilité négligeable. Ainsi, $\mathbf{G}_3$ et $\mathbf{G}_2$ sont indistinguables.

Jeu $\mathbf{G}_4$: Simulation du client depuis le début du protocole. Dans ce jeu, $\mathcal{S}$ simule le client non-corrompu depuis le début du protocole, mais désormais sans avoir accès à son mot de passe. La simulation se déroule ainsi : en S1, le client choisit un x aléatoire et envoie le X correspondant au serveur. En S3, s'il est encore honnête, il ne demande pas de requête de déchiffrement pour Y^*.

Si tous les messages sont *oracle-generated*, le simulateur calcule alors Auth avec un oracle $\mathcal{H}'_1$ privé : $\mathsf{Auth} = \mathcal{H}'_1(\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}^*)$ à la place de $\mathcal{H}_1$. Il peut y avoir un problème si le serveur se fait corrompre, ce que nous décrivons formellement plus loin. Sinon, si le message reçu par le client n'est pas *oracle-generated*, nous distinguons deux cas :

- Le simulateur connaît le mot de passe du serveur s'il a été corrompu plus tôt dans le protocole (récupération automatique), ou si la valeur Y^* envoyée par l'adversaire en S2 a été obtenue via une requête de chiffrement (avec l'aide de la liste de chiffrement). Ainsi, quand il reçoit Y^*, le client pose une requête `GoodPwd` à la fonctionnalité pour le client avec le mot de passe du serveur. Si le mot de passe est correct, alors $\mathcal{S}$ utilise $\mathcal{H}_1$ pour le client, sinon il utilise son oracle privé $\mathcal{H}'_1$: $\mathsf{Auth} = \mathcal{H}'_1(\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}^*)$.
- Si l'adversaire n'a pas obtenu Y^* via une requête de chiffrement, il y a une chance négligeable qu'il connaisse la valeur y correspondante et le client utilise aussi $\mathcal{H}'_1$ dans ce cas. L'événement `AskH` peut alors faire abandonner le jeu (nous bornerons cette probabilité plus tard ; il suffit pour l'instant de remarquer qu'elle est négligeable et reliée au CDH) :

  > `AskH` : $\mathcal{A}$ pose l'une des deux requêtes suivantes à l'un des oracles $\mathcal{H}_0$ ou $\mathcal{H}_1$: $\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{K}_\mathrm{U}$ ou $\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{K}_\mathrm{S}$, c'est-à-dire la valeur commune de $\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathsf{CDH}(\mathrm{X},\mathrm{Y})$.

Nous montrons maintenant comment simuler la deuxième partie de U3 (le calcul de sk_U). Nous devons séparer les cas dans lesquels le client reste honnête, et ceux dans lesquels il se fait corrompre.

- Si le client reste honnête, il reçoit sk_U à travers une requête à un oracle privé $\mathcal{H}'_0$ si Auth a été obtenue à travers une requête à $\mathcal{H}'_1$ et s'il n'y a eu aucune corruption. Il la reçoit par une requête à $\mathcal{H}_0$ dans les autres cas (si Auth a été obtenue à travers une requête à $\mathcal{H}_1$ ou si Auth a été obtenue à travers une requête à $\mathcal{H}'_1$ et qu'il y a eu une corruption ensuite).
- Si le client se fait corrompre pendant U3, l'adversaire doit recevoir son état interne : le simulateur connaît déjà x et apprend son mot de passe ; il est donc capable de calculer un Y correct. Quand il reçoit une requête d'authentification de la part de l'adversaire pour le client, $\mathcal{S}$ calcule alors Auth via une requête à $\mathcal{H}_1$ (il n'y a pas besoin que cette requête donne la même valeur que celle calculée précédemment par la requête à $\mathcal{H}'_1$, étant donné que Auth n'a pas été publié). De la même manière, le simulateur utilisera $\mathcal{H}_0$ pour calculer la clef sk_U du client.

Si les deux joueurs sont honnêtes au début de S4 et que tous les messages ont été *oracle-generated*, il n'y aura pas de problème étant donné que dans le jeu précédent nous avons empêché le serveur de calculer Auth. S'il se fait corrompre après que Auth a été envoyé, et si les mots de passe sont les mêmes, le simulateur reprogramme les oracles de façon à ce que, d'une part, $\mathcal{H}_1(\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{K}_\mathrm{U}) = \mathcal{H}'_1(\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}^*)$ et, d'autre part, $\mathcal{H}_0(\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{K}_\mathrm{U}) = \mathcal{H}'_0(\mathsf{ssid}\|\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}^*)$. Cette programmation ne va échouer que si cette requête à $\mathcal{H}_1$ ou $\mathcal{H}_0$ a déjà été posée avant la corruption, auquel cas l'événement `AskH` apparaît.

Enfin, si le client se fait corrompre, $\mathcal{S}$ fait la même reprogrammation.

Ainsi, en ignorant les événements `AskH`, dont nous calculerons la probabilité plus tard, les jeux $\mathbf{G}_4$ et $\mathbf{G}_3$ sont indistinguables.

Jeu $\mathbf{G}_5$: Simulation du serveur dans la dernière étape du protocole. Dans ce jeu, on demande à $\mathcal{S}$ de simuler le serveur non-corrompu dans l'étape S4. Plus précisément, durant ce jeu, on considère deux cas. Si aucune corruption n'a lieu avant S4 et que tous les messages ont été *oracle-generated*, le comportement de $\mathcal{S}$ a été décrit au jeu $\mathbf{G}_3$. Si, à l'inverse, le client a été corrompu avant le début de S4, ou si un message a été non *oracle-generated*, la simulation est effectuée comme suit :

- Si le client est corrompu ou contrôlé par l'adversaire qui a déchiffré Y^* pour obtenir la valeur Y envoyée dans Auth, alors le serveur récupère le mot de passe utilisé (par la corruption ou la liste de déchiffrement) et il vérifie le Diffie-Hellman envoyé par le client. S'il est incorrect, le serveur reçoit une erreur. S'il est correct, alors le simulateur pose une requête `GoodPwd` pour le serveur avec ce mot de passe. Si ce dernier est correct, alors le serveur reçoit la même clef que le client ; sinon, il reçoit un message d'erreur.
- Si le client est contrôlé par l'adversaire qui a envoyé quoi que ce soit d'autre, on abandonne le jeu. Ceci n'arrive que s'il a deviné Y par hasard, c'est-à-dire avec probabilité négligeable.

Finalement, si le serveur se fait corrompre pendant S4, l'adversaire reçoit y et Y. Plus précisément, le simulateur récupère le mot de passe du serveur et donne à l'adversaire des valeurs cohérentes avec les listes. Ainsi, $\mathbf{G}_5$ et $\mathbf{G}_4$ sont indistinguables.

Jeu $\mathbf{G}_6$: Simulation du serveur depuis le début du protocole. Dans ce jeu, $\mathcal{S}$ simule les joueurs non-corrompus depuis le début du protocole. On a déjà vu comment $\mathcal{S}$ simule le client. La simulation pour un serveur non corrompu est effectuée comme suit.

En S2, le serveur envoie une valeur Y^* aléatoire (choisie sans poser de requête à l'oracle de chiffrement). S'il se fait corrompre, le simulateur récupère son mot de passe, et peut alors donner à l'adversaire des valeurs adéquates de y et Y à l'aide des listes de chiffrement et déchiffrement. La simulation de S4 a déjà été décrite dans le jeu précédent.

$\mathbf{G}_6$ est indistinguable de $\mathbf{G}_5$, étant donné que si les deux joueurs restent honnêtes jusqu'à la fin du jeu, ils ont la même clef dépendant uniquement de leurs mots de passe dans $\mathbf{G}_3$. Quant au cas où l'un des deux a été corrompu, il a été géré dans les deux jeux précédents, et l'exécution ne dépend pas de la valeur de Y^*, puisque le chiffrement de $G \to G$ est tel qu'il existe toujours un clair correspondant à un chiffré.

Jeu $\mathbf{G}_7$: Résumé de la simulation et remplacement des requêtes hybrides. On modifie le jeu précédent en remplaçant les requêtes hybrides `GoodPwd` et `SamePwd` par leurs versions idéales. Si une session est abandonnée ou terminée, alors $\mathcal{S}$ le signale à $\mathcal{A}$.

La figure 6.3 résume la simulation jusqu'à ce point et décrit complètement le comportement du simulateur. Au début d'une étape du protocole, le joueur est supposé honnête (sinon nous n'avons pas à le simuler), et il peut être corrompu à la fin de l'étape. On suppose que U3(1) doit être exécutée avant U3(2) et U3(3). Mais les deux dernières étapes peuvent être exécutées dans n'importe quel ordre. Par simplicité, on suppose dans la suite que cet ordre est respecté.

Montrons que $\mathbf{G}_7$ est indistinguable du jeu idéal en rappelant d'abord que les requêtes `GoodPwd` sont remplacées par des requêtes `TestPwd` à la fonctionnalité et les `SamePwd` par des `NewKey` : ce sont les seules différences entre $\mathbf{G}_6$ et $\mathbf{G}_7$. On dit que les joueurs ont des sessions compatibles s'ils partagent le même identifiant ssid, ont des rôles opposés (client et serveur) et partagent les mêmes valeurs de X et Y^*.

D'abord, si les deux joueurs restent honnêtes jusqu'à la fin du jeu, ils obtiendront une clef aléatoire, à la fois dans $\mathbf{G}_7$ et IWE (*ideal world experiment*, le jeu idéal), puisqu'il n'y a pas de requêtes `TestPwd` et que les sessions restent `fresh`.

Nous devons montrer qu'un client honnête va recevoir la même clef qu'un serveur honnête dans $\mathbf{G}_7$ si et seulement si cela arrive dans IWE.

Fig. 6.3 – Simulation et corruptions adaptatives

	Client	Serveur	Simulation
U1	honnête	honnête	x aléatoire, $\mathrm{X} = g^x$
		adversaire	
	se fait corrompre	honnête	révèle x à $\mathcal{A}$
		adversaire	
S2	honnête	honnête	Y^* aléatoire
	adversaire		
	honnête	se fait corrompre	apprend pw calcule y et Y via une requête $\mathcal{D}$ révèle $\mathrm{X}, y, \mathrm{Y}$ à $\mathcal{A}$
	adversaire		
U3 (1)	honnête	honnête	pas de requête $\mathcal{D}$ sur Y^*
		adversaire	
	se fait corrompre	honnête	apprend pw calcule y et Y via une requête $\mathcal{D}$ révèle $x, \mathrm{X}, \mathrm{Y}$ à $\mathcal{A}$
		adversaire	
U3 (2)	honnête	honnête	utilise $\mathcal{H}'_1$ pour Auth
		adversaire	GoodPwd(pw) faux, utilise $\mathcal{H}'_1$
			GoodPwd(pw) correct, utilise $\mathcal{H}_1$
			si pw inconnu, abandonne
	se fait corrompre	honnête	apprend pw calcule y et Y via une requête $\mathcal{D}$ révèle $x, \mathrm{X}, \mathrm{Y}$ à $\mathcal{A}$
		adversaire	
U3 (3)	honnête	honnête	utilise $\mathcal{H}'_0$ pour Auth
		adversaire	GoodPwd(pw) faux, utilise $\mathcal{H}'_0$
			GoodPwd(pw) correct, utilise $\mathcal{H}_0$
S4	honnête	honnête	si SamePwd correct, alors même clef sk
			si SamePwd incorrect, alors erreur
	adversaire		si pw inconnu, alors abandonne
			si pw connu, DH faux, alors erreur
			si pw connu, DH correct, GoodPwd(pw) correct, alors même clef
			si pw connu, DH correct, GoodPwd(pw) faux, alors erreur

On gère d'abord le cas d'un client et un serveur avec des sessions compatibles. S'ils ont le même mot de passe, alors dans $\mathbf{G}_7$, ils vont recevoir la même clef : s'ils sont honnêtes, ils la reçoivent depuis $\mathbf{G}_3$; si le client est honnête avec un serveur corrompu, ils la reçoivent depuis $\mathbf{G}_4$; enfin, si le client est corrompu, ils la reçoivent depuis $\mathbf{G}_5$. Dans IWE, la fonctionnalité va recevoir deux requêtes `NewSession` avec le même mot de passe. Si les deux joueurs sont honnêtes, elle ne va pas recevoir de requête `TestPwd`, et donc la clef va être la même pour les deux. Et si l'un des deux est corrompu et qu'une requête `TestPwd` a lieu (et qu'elle est correcte, vu qu'ils partagent le mot de passe), alors ils vont aussi avoir la même clef, choisie par l'adversaire.

S'ils n'ont pas le même mot de passe, alors dans $\mathbf{G}_7$, le serveur va recevoir une erreur. Dans IWE, c'est simplement la définition de la fonctionnalité.

On s'occupe désormais du cas dans lequel le client et le serveur n'ont pas de sessions compatibles. Il est clair dans $\mathbf{G}_7$ que dans ce cas, les clefs de session d'un client et d'un serveur vont être indépendantes car elles ne sont fixées dans aucun jeu. Dans IWE, la seule façon pour qu'ils reçoivent des clefs identiques est que la fonctionnalité reçoive deux requêtes `NewSession` avec les mêmes mots de passe, et que $\mathcal{S}$ envoie des requêtes `NewKey` sans avoir envoyé aucune requête `TestPwd`. Mais si ces deux sessions n'ont pas de conversation compatible, elles doivent différer dans X, Y^* ou Auth. La probabilité qu'ils partagent la même paire $(\mathrm{X}, \mathrm{Y}^*)$ est bornée par q_ε^2/q et donc négligeable, q_ε étant le nombre de requêtes de chiffrement à l'oracle.

Gérons maintenant les corruptions : si le client se fait corrompre avant la fin du jeu, alors dans $\mathbf{G}_7$, le simulateur récupère le mot de passe et l'utilise dans une requête `TestPwd` pour le serveur à la fonctionnalité. Si la réponse est incorrecte, le serveur reçoit un message d'erreur, et si elle est correcte, il reçoit la même clef que le client (et cette dernière a été choisie par le simulateur). C'est exactement le comportement de la fonctionnalité dans IWE. Si le serveur se fait corrompre, il y a une requête `TestPwd` concernant le client dans $\mathbf{G}_7$. Si le mot de passe est correct, le serveur recevra la clef donnée par le simulateur dans la requête `NewKey`, sinon, elle sera choisie par la fonctionnalité. La même chose arrive dans IWE.

6.3.4 Simuler les exécutions à travers le problème CDH

Comme dans [BCP03b], on calcule la probabilité de l'événement `AskH` avec l'aide d'une réduction au problème CDH, étant donnée une instance $\mathsf{CDH}(\mathrm{A}, \mathrm{B})$. Plus précisément, `AskH` signifie qu'il existe une session dans laquelle on a remplacé les oracles aléatoires $\mathcal{H}_0$ et $\mathcal{H}_1$ par $\mathcal{H}'_0$ et $\mathcal{H}'_1$ respectivement, et dans laquelle $\mathcal{A}$ a posé la requête de hachage correspondante. On choisit alors au hasard une session, appelée ssid, et on injecte l'instance CDH dans cette session spécifique (on a choisi la bonne session avec probabilité $1/q_s$). Dans cette session ssid, on maintient une liste Λ_B, et

- le client pose $\mathrm{X} = \mathrm{A}$;
- le serveur choisit Y^* au hasard, mais le comportement du déchiffrement est modifié sur cette entrée spécifique Y^*, quelle que soit la clef, mais seulement pour cette session ssid : on choisit un élément aléatoire $\beta \in {\mathbb{Z}_q}^*$ on calcule $\mathrm{Y} = \mathrm{B}g^\beta$, et on enregistre (β, Y) dans la liste Λ_B, ainsi que l'élément usuel dans Λ_ε. Si Y appartient déjà à la liste, on abandonne comme avant.

On remarque que cela n'affecte que la session critique ssid et que cela ne change rien d'autre. Contrairement à la simulation précédente, on ne connaît pas les valeurs de x et φ (les logarithmes discrets de X et Y), mais on n'en a plus besoin étant donné que les valeurs de $\mathrm{K_U}$ et $\mathrm{K_S}$ ne sont plus requises pour calculer l'authentifiant et la clef de session : l'événement `AskH` qui apparaît pour cette session (X, Y) signifie que l'adversaire a posé la requête $\mathrm{U}\|\mathrm{S}\|\mathrm{X}\|\mathrm{Y}\|\mathrm{Z}$ aux oracles aléatoires $\mathcal{H}_0$ ou $\mathcal{H}_1$, avec $\mathrm{Z} = \mathsf{CDH}(\mathrm{X}, \mathrm{Y})$. En choisissant aléatoirement dans la liste $\Lambda_\mathcal{H}$, on obtient ce triplet Diffie-Hellman avec probabilité $1/q_h$, où q_h est le nombre de requêtes de hachage. On peut alors simplement regarder dans la liste Λ_B pour les valeurs β telles que $\mathrm{Y} = \mathrm{B}g^\beta$: $\mathsf{CDH}(\mathrm{X}, \mathrm{Y}) = \mathsf{CDH}(\mathrm{A}, \mathrm{B}g^\beta) = \mathsf{CDH}(\mathrm{A}, \mathrm{B})\mathrm{A}^\beta$.

On note cependant qu'en cas de corruption, on peut avoir besoin de révéler des états internes, avec x et φ :

- Si la corruption a lieu avant la fin de U3, avec la publication de Auth, il n'y a pas de problème puisque les oracles aléatoires ne vont pas être remplacés par des oracles privés, et donc le choix de la session n'était pas correct.
- Si la corruption a lieu après la fin de U3, avec la publication de Auth, il n'y a pas non plus de problème :
  - la corruption du client ne révèle pas son état interne, puisqu'il a terminé son exécution ;
  - la corruption du serveur mène à une « reprogrammation » des oracles publiques qui mène immédiatement à l'événement `AskH` si la requête a déjà été posée. On peut alors terminer notre simulation, et extraire la valeur Diffie-Hellman de la liste $\Lambda_\mathcal{H}$, sans avoir à attendre la fin de toute l'attaque.

Chapitre 7

Protocole de groupe

Nous considérons ici des échanges de clefs de groupe basés sur des mots de passe résistant aux corruptions fortes dans un cadre adaptatif, dans lequel les adversaires sont autorisés à corrompre les joueurs de manière adaptative, basée sur l'information recueillie jusqu'à ce point (voir le chapitre 5 page 85). Les échanges de clefs adaptativement sûrs permettent l'établissement de canaux sécurisés même en présence d'un adversaire qui peut corrompre les joueurs à n'importe quel moment et obtenir leur état interne.

En dépit des nombreux travaux sur les échanges de clefs dans le cadre de groupes, peu de schémas ont été prouvés capables de résister aux corruptions fortes dans le cas d'adversaires adaptatifs. Dans le contexte de GAKE, Katz et Shin [KS05] ont proposé un compilateur qui convertit n'importe quel protocole sûr dans un cadre de corruption faible en un protocole sûr dans un cadre de corruption forte, mais leur protocole repose sur des signatures et ne fonctionne pas dans un scénario à base de mots de passe. Dans le cas des GPAKE, le seul schéma résistant à notre connaissance aux corruptions adaptatives est dû à Barak *et al.* [BCL$^+$05] et il utilise des technique générales de *multiparty computation*. Malheureusement, leur schéma est inutilisable en pratique.

Dans le cadre de sécurité de Katz and Shin [KS05], on souhaite seulement empêcher l'adversaire de déterminer complètement la clef s'il n'a corrompu aucun utilisateur : il est totalement libre de la choisir dès qu'il a un corrompu un joueur dans le groupe. Il serait en fait préférable d'avoir une contrainte plus forte sur le caractère aléatoire de la clef : autrement dit, l'adversaire ne devrait pas être capable de biaiser la clef s'il n'a pas corrompu « suffisamment » de joueurs.

Il y a plusieurs avantages à adopter une telle notion de *contributory* plus forte pour les échanges de clefs de groupes. En premier lieu, cela crée une distinction claire entre les protocoles de *distribution* de clefs et ceux d'*accord* (ou d'*échange*) de clefs en décrivant en termes concrets la notion que, dans un protocole d'accord de clefs, chaque joueur devrait contribuer de manière égale à la clef. Ensuite, cela rend le protocole plus robuste aux pannes puisqu'on garantit que la clef de session est distribuée uniformémement même si certains joueurs ne choisissent pas leurs contributions correctement (en cas de mauvais fonctionnement matériel par exemple).

Cela évite également des scénarios dans lesquels des adversaires infiltrés laissent secrètement fuir la valeur de la clef de session à une tierce personne, soit en imposant une valeur spécifique pour sa contribution à la clef de session, soit en biaisant sa distribution, afin que la clef puisse en être déduite facilement, et la communication alors espionnée en temps réel. Par exemple, on peut imaginer que la tierce personne soit une agence d'espionnage, telle que la CIA, et les infiltrés des pièces matérielles ou logicielles malicieuses (comme un générateur pseudo-aléatoire) installées par l'agence d'espionnage.

Enfin, si l'absence de canaux subliminaux [Sim84] peut être garantie pendant la durée de vie de la clef de session à partir du moment où les joueurs ont rassemblé suffisamment d'information pour calculer cette clef (une propriété qui doit être étudiée indépendamment), alors aucun adversaire infiltré ne devrait être capable d'aider un tiers à espionner la communication en temps réel. De manière intéressante, étant donné que toutes les fonctions utilisées dans les derniers messages sont déterministes, cette propriété semble être satisfaite par notre schéma. Bien sûr, on ne peut pas éviter qu'un infiltré ne révèle plus tard la clef, mais il serait alors peut-être trop tard pour que cette information ait un quelconque intérêt. D'un autre côté, des protections physiques ou une surveillance du réseau peuvent assurer qu'aucun flux de données ne puisse être envoyé à l'extérieur par les joueurs pendant une discussion confidentielle. La propriété de *contributory* ainsi que l'absence de canaux subliminaux garantissent alors l'impossibilité d'espionnage en temps réel, ce qui mène à une nouvelle propriété de sécurité. Cependant, l'étude des canaux subliminaux est hors-sujet ici.

Nous donnons en premier lieu une définition formelle des protocoles *contributory* [BVS07, DPSW06, BM08] et nous définissons une fonctionnalité idéale pour les échanges de clefs de groupe basés sur des mots de passe avec authentification mutuelle explicite et *contributory* dans le cadre UC. Comme dans les définitions précédentes dans ce cadre (voir la section 5.8 page 81), nos définitions ne supposent aucune distribution particulière sur les mots de passe ni l'indépendance entre les mots de passe des différents joueurs.

Nous proposons ensuite les premiers pas pour la réalisation de cette fonctionnalité dans le cadre adaptatif fort décrit ci-dessus en analysant un protocole efficace existant et en montrant qu'il réalise la fonctionnalité idéale sous l'hypothèse CDH. Nous montrons ainsi qu'une petite variante du GPAKE d'Abdalla *et al.* [ABCP06], basée sur le protocole de Burmester et Desmedt [BD94, BD05] et décrit dans la section 7.2, réalise de manière sûre la nouvelle fonctionnalité, même contre des attaques adaptatives.

La preuve, dans les modèles du chiffrement étiqueté idéal et de l'oracle aléatoire [BR93], est donnée dans la section 7.3. Même si, d'un point de vue mathématique, il serait préférable d'avoir une preuve dans le modèle standard, il est à remarquer que le protocole présenté ici est le premier échange de clefs de groupe basé sur des mots de passe qui réalise des notions de sécurité aussi fortes, c'est-à-dire la sécurité adaptative contre les corruptions fortes dans le cadre UC et la $(n/2, n)$-*contributory* (définie ci-dessous). En outre, notre protocole est assez efficace. Nous donnons aussi une modification pour atteindre la $(n-1, n)$-*contributory*. Les résultats de ce chapitre ont été présentés à la conférence Africacrypt 2009 [ACCP09].

7.1 Cadre de sécurité

7.1.1 Notion de *contributory*

Nous considérons ici un cadre de corruption contre les attaques internes (*insiders*) plus fort que celui proposé par Katz et Shin dans [KS05], dans lequel ils autorisent l'adversaire à choisir la clef de session dès qu'il y a une corruption (ce qui correspond à $t = 1$ dans la définition ci-dessous). Nous définissons ici une notion de protocole *contributory*, qui garantit à l'inverse que la distribution des clefs de session reste aléatoire tant qu'il y a suffisamment de participants honnêtes dans la session : l'adversaire ne peut pas biaiser la distribution des clefs à moins de contrôler un grand nombre de joueurs. Bien entendu, on ne peut pas empêcher un attaquant infiltré d'apprendre la clef, et éventuellement de la révéler (ou même toute

la communication) à l'extérieur. Mais l'on peut espérer empêcher la fuite « subliminale » d'information qui autoriserait de l'espionnage en temps réel.

Plus précisément, on dit qu'un protocole est (t, n)-*contributory* si le groupe possède n personnes et que l'adversaire ne peut pas biaiser la clef tant qu'il a corrompu moins de t joueurs. Concrètement, le protocole que nous proposons est $(n/2, n)$-*contributory*, ce qui signifie que l'adversaire ne peut pas biaiser la clef tant qu'il y a au moins la moitié des joueurs honnêtes. Nous donnons ensuite une extension qui montre qu'on peut obtenir un protocole $(n-2, n)$-*contributory* en effectuant des exécutions parallèles.

7.1.2 La fonctionnalité d'échange de clefs basée sur des mots de passe avec authentification mutuelle

Nous décrivons ici la fonctionnalité $\mathcal{F}_{\text{GPAKE}}$ (voir la figure 7.1). L'extension multi-session de cette fonctionnalité serait similaire à celle proposée par Canetti et Rabin [CR03]. Nos points de départ sont la fonctionnalité d'échange de clefs de groupe décrite par Katz et Shin [KS05] et celle pour deux joueurs basée sur des mots de passe donnée par Canetti *et al.* dans [CHK+05]. Nous avons combiné les deux et ajouté l'authentification mutuelle et la (t, n)-*contributory*. La nouvelle définition reste très générale : en posant $t = 1$, nous retrouvons le cas dans lequel l'adversaire peut choisir la clef dès qu'il contrôle au moins un joueur, comme dans [CHK+05, KS05].

Notre fonctionnalité garantit l'authentification mutuelle (voir chapitre 2.1.1 page 29), qui assure explicitement que, si un joueur accepte la clef de session, alors aucun autre joueur n'a pu accepter une autre clef de session. Notons cependant que l'adversaire peut modifier les messages suivants et ainsi faire rejeter certains joueurs pendant que les autres acceptent. Le protocole fait inévitablement fuir de l'information à l'adversaire, qui apprend au final si les joueurs partagent un secret commun ou non.

Remarquons d'abord comme dans le chapitre précédent que la fonctionnalité n'est pas chargée de donner les mots de passe aux participants : c'est le rôle de l'environnement.

Dans toute la suite, nous appelons n le nombre de joueurs impliqués dans une exécution donnée du protocole. Comme dans le chapitre précédent, la fonctionnalité commence par une étape d'initialisation durant laquelle elle se contente essentiellement d'attendre que chaque joueur notifie son intérêt à participer au protocole. Plus précisément, nous supposons que chaque joueur commence une nouvelle session du protocole avec l'entrée (`NewSession`, sid, P_i, Pid, pw_i), où P_i est l'identité du joueur, pw_i est son mot de passe et Pid représente l'ensemble des (identités des) joueurs avec lesquelles il a l'intention de partager une clef de session. Une fois que tous les joueurs (partageant les mêmes sid et Pid) ont reçu leur message de notification, $\mathcal{F}_{\text{GPAKE}}$ informe l'adversaire qu'il est prêt à commencer une nouvelle session du protocole, grâce au mot-clef `ready`.

En principe, une fois l'étape d'initialisation terminée, tous les joueurs sont prêts à recevoir la clef de session. Cependant, la fonctionnalité attend que $\mathcal{S}$ envoie un message `NewKey` avant de passer à la suite. Cela autorise $\mathcal{S}$ à décider le moment exact où la clef doit être envoyée aux joueurs et, en particulier, cela lui permet de choisir le moment exact où les corruptions doivent avoir lieu ($\mathcal{S}$ peut en effet décider de corrompre un joueur P_i avant que la clef soit envoyée mais après que P_i a décidé de participer à une session donnée du protocole, voir [KS05]).

Une fois que la fonctionnalité a reçu un message (sid, Pid, `NewKey`, sk) de la part de $\mathcal{S}$, il passe à la phase de génération de clef, de la façon suivante. Si tous les joueurs de Pid partagent le même mot de passe et que (strictement) moins de t joueurs sont corrompus, alors la fonctionnalité choisit une clef sk' uniformément et aléatoirement dans l'espace des clefs approprié. Si tous les joueurs dans Pid partagent le même mot de passe mais que t joueurs ou plus sont corrompus, alors la fonctionnalité autorise $\mathcal{S}$ à déterminer complètement la clef en posant $\mathsf{sk}' = \mathsf{sk}$. Dans tous les autres cas, aucune clef n'est établie.

Remarquons que par simplicité, nous avons choisi d'intégrer la *contributory* dans la fonctionnalité idéale. Ce n'était cependant pas indispensable et l'on aurait pu définir la fonctionnalité pour des échanges de clef pour des groupes basés sur des mots de passe, et étudier ensuite

la *contributory* comme une propriété supplémentaire. Mais on aurait alors eu besoin de deux preuves de sécurité.

Notre définition de la fonctionnalité $\mathcal{F}_{\text{GPAKE}}$ gère les corruptions d'une façon assez similaire à celle de $\mathcal{F}_{\text{GAKE}}$ dans [KS05], dans le sens où si un adversaire a corrompu certains participants, il peut déterminer la clef de session, mais uniquement s'il y a assez de joueurs corrompus. Il est cependant à noter que $\mathcal{S}$ n'a ce pouvoir qu'avant que la clef ne soit établie. Une fois cette clef fixée, les corruptions autorisent l'adversaire à connaître la clef mais pas à la choisir.

Les requêtes `TestPwd` ne semblent pas nécessaires dans le contexte des fonctionnalités de répartition (voir la section 5.8.3 page 85), étant donné qu'en divisant le groupe entier en sous-groupes d'un joueur chacun, l'adversaire peut déjà tester un mot de passe par joueur. Cependant, ajouter cette requête modélise les attaques par dictionnaire en ligne, en permettant simplement à l'adversaire de tester un mot de passe supplémentaire par joueur, ce qui ne change pas significativement son pouvoir en pratique. En outre, ces requêtes sont nécessaires dans l'analyse de sécurité de notre protocole, que nous avons essayé de conserver aussi efficace que possible. Construire un protocole efficace tout en se passant de ces requêtes est un problème ouvert intéressant.

Plus précisement, les requêtes `TestPwd` permettent de modéliser le fait que dans un sous-groupe particulier (déterminé grâce à la fonctionnalité de répartition), l'adversaire est capable de diviser une fois de plus l'ensemble des utilisateurs dans des plus petits groupes en testant localement les mots de passe de ces utilisateurs. Nous appellerons cela des composantes connexes, voir page 107. En fait, dans la plupart des protocoles basés sur celui de Burmester et Desmedt [BD05], comme celui de Abdalla *et al.* [ABCP06], l'adversaire peut tester la valeur du mot de passe d'un utilisateur simplement en jouant le rôle des voisins de celui-ci. Ceci étant dit, on remarque toutefois que même si les requêtes `TestPwd` pourraient être évitées selon le protocole considéré, leur utilisation n'affaiblit pas particulièrement le cadre de sécurité puisque l'adversaire est encore limité à deux tests de mots de passe par utilisateur honnête pour chaque session dans laquelle il joue un rôle actif, en séparant le groupe en sous-groupes d'un utilisateur honnête chacun. Ainsi, on doit juste ajouter un bit d'entropie supplémentaire au mot de passe pour contrebalancer cet effet.

Dans tous les cas, après la génération de la clef, la fonctionnalité prévient l'adversaire du résultat, c'est-à-dire que l'adversaire apprend si une clef a été effectivement établie ou non. En particulier, cela signifie qu'il apprend aussi si les joueurs partagent le même mot de passe ou non. À première vue, cela peut sembler une information dangereuse à lui donner, mais cela n'est cependant pas le cas dans notre cadre. En effet, comme tous les mots de passe sont choisis par l'environnement, une telle information pourrait déjà être connue de l'adversaire. De plus, cela ne semble pas critique de cacher le statut du protocole (c'est-à-dire s'il s'est terminé correctement ou non), puisqu'en pratique cette information peut aisément être obtenue en surveillant l'exécution (si les joueurs arrêtent subitement la communication, le protocole n'a pas dû se terminer correctement).

Finalement, la clef est envoyée aux joueurs selon le planning choisi par $\mathcal{S}$. Ceci est formellement modélisé par le biais de requêtes de livraison de clefs. Nous supposons qu'une fois que $\mathcal{S}$ a demandé la livraison d'une clef à un joueur, cette dernière est envoyée immédiatement.

Remarquez que l'authentification mutuelle signifie en fait que si l'un des joueurs accepte, alors tous les joueurs partagent de quoi établir la même clef (et pas une autre) ; mais cela ne signifie pas qu'ils acceptent tous. En fait, on ne peut pas supposer que tous les messages sont correctement transmis par l'adversaire : il peut modifier un seul message, ou au minimum oublier/refuser d'en délivrer un. Cette attaque, appelée *déni de service*, est modélisée dans la fonctionnalité par la livraison de la clef : l'adversaire peut choisir s'il veut que le joueur reçoive ou non la clef (donc les messages) simplement à l'aide du mot-clef `b` fixé à `yes` ou `no`.

Fig. 7.1 – Fonctionnalité $\mathcal{F}_{\text{GPAKE}}$

La fonctionnalité $\mathcal{F}_{\text{GPAKE}}$ utilise un paramètre de sécurité k et le paramètre t de la *contributory*. Elle correspond à l'identifiant de session sid. Elle maintient une liste L, initialement vide, contenant des valeurs de la forme $[(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i), (\mathtt{completed}, \mathsf{sk})]$ et interagit avec un adversaire $\mathcal{S}$ et un ensemble de joueurs $\mathrm{P}_1,\ldots,\mathrm{P}_n$ à travers les requêtes suivantes :

- **Initialisation.** À réception d'une requête $(\mathtt{NewSession}, \mathsf{sid}, \mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i)$ de la part du joueur P_i pour la première fois, où Pid est un ensemble d'au moins deux identités distinctes contenant P_i, enregistrer $(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i)$ avec le statut $\mathtt{fresh}$, et envoyer $(\mathsf{sid}, \mathrm{P}_i, \mathsf{Pid})$ à $\mathcal{S}$. Ignorer toutes les requêtes $(\mathtt{NewSession}, \mathsf{sid}, \mathrm{P}_j, \mathsf{Pid}', \mathsf{pw}_j)$ suivantes où $\mathsf{Pid}' \neq \mathsf{Pid}$.

  S'il y a au moins $|\mathsf{Pid}| - 1$ triplets $(\mathsf{sid}, \mathrm{P}_j, \mathsf{Pid}, \mathsf{pw}_j)$ pour des joueurs $\mathrm{P}_j \in \mathsf{Pid} \setminus \{\mathrm{P}_i\}$, alors enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathtt{ready})$ et l'envoyer à $\mathcal{S}$.
- **Tests de mots de passe.** À réception d'une requête $(\mathtt{TestPwd}, \mathsf{sid}, \mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}')$ de la part de $\mathcal{S}$, s'il existe un enregistrement de la forme $[(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i), (\mathtt{fresh}, \perp)]$:
  - Si $\mathsf{pw}_i = \mathsf{pw}'$, noter l'enregistrement $\mathtt{compromised}$ et répondre à $\mathcal{S}$ qu'il a effectué un test correct.
  - Si $\mathsf{pw}_i \neq \mathsf{pw}'$, noter l'enregistrement $\mathtt{interrupted}$ et répondre à $\mathcal{S}$ qu'il a effectué un test incorrect.
- **Géneration de la clef.** À réception d'un message $(\mathtt{NewKey}, \mathsf{sid}, \mathsf{Pid}, \mathsf{sk})$ de la part de $\mathcal{S}$ et s'il existe un triplet enregistré $(\mathsf{sid}, \mathsf{Pid}, \mathtt{ready})$, alors, en notant n_c le nombre de joueurs corrompus,
  - Si tous les $\mathrm{P}_i \in \mathsf{Pid}$ ont le même mot de passe et si $n_c < t$, choisir $\mathsf{sk}' \in \{0,1\}^k$ uniformément de façon aléatoire et enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{sk}')$. Ensuite, pour tous les $\mathrm{P}_i \in \mathsf{Pid}$ noter l'enregistrement $(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i)$ $\mathtt{completed}$.
  - si tous les $\mathrm{P}_i \in \mathsf{Pid}$ ont le même mot de passe et si $n_c \geqslant t$, enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{sk})$. Alors, pour tous les $\mathrm{P}_i \in \mathsf{Pid}$, noter l'enregistrement $(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i)$ $\mathtt{completed}$.
  - Dans tous les autres cas, enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$. Pour tous les $\mathrm{P}_i \in \mathsf{Pid}$, noter l'enregistrement $(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i)$ $\mathtt{error}$.

  Quand la clef est fixée, donner le résultat (soit $\mathtt{error}$ soit $\mathtt{completed}$) à $\mathcal{S}$.
- **Livraison de la clef.** À réception d'un message $(\mathtt{KeyDelivery}, \mathsf{sid}, \mathrm{P}_i, \mathtt{b})$ de la part de $\mathcal{S}$, alors si $\mathrm{P}_i \in \mathsf{Pid}$ et s'il existe un triplet $(\mathsf{sid}, \mathsf{Pid}, \alpha)$ où $\alpha \in \{0,1\}^k \cup \{\mathtt{error}\}$, envoyer $(\mathsf{sid}, \mathsf{Pid}, \alpha)$ à P_i si $\mathtt{b}$ est égal à $\mathtt{yes}$ ou $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$ si $\mathtt{b}$ est égal à $\mathtt{no}$. Compléter l'enregistrement $(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i)$ avec α ou $\mathtt{error}$.
- **Corruption d'un joueur.** Si $\mathcal{S}$ a corrompu $\mathrm{P}_i \in \mathsf{Pid}$ et s'il existe un enregistrement $(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i)$, alors révéler pw_i à $\mathcal{S}$. S'il existe aussi un triplet $(\mathsf{sid}, \mathsf{Pid}, \mathsf{sk})$, et que la dernière valeur de l'enregistrement $(\mathrm{P}_i, \mathsf{Pid}, \mathsf{pw}_i)$ vaut encore $\perp$, alors envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{sk})$ à $\mathcal{S}$.

7.2 Notre protocole

Description. Notre solution est basée sur un protocole précédent proposé par Abdalla *et al.* [ABCP06] et elle est décrite dans la figure 7.2. Soient $\mathcal{E}$ et $\mathcal{D}$ les schémas de chiffrement et de déchiffrement d'un schéma de chiffrement idéal étiqueté (*tweakable*). Notons $\mathcal{E}^{\ell}_{\mathsf{pw}}(m)$ le chiffrement du message m utilisant l'étiquette ℓ et le mot de passe pw. De façon similaire, le déchiffrement est noté $\mathcal{D}^{\ell}_{\mathsf{pw}}(c)$. Ce protocole utilise cinq oracles aléatoires différents, que l'on note $\mathcal{H}_i$ pour tout $i = 0,\ldots,4$. On appelle q_{h_i} le nombre de requêtes faites à l'oracle $\mathcal{H}_i$ ($q_h = q_{h_0} + \cdots + q_{h_4}$), et $k_i = 2^{\ell_i}$ la taille de sortie : $\mathcal{H}_i \colon \{0,1\}^k \to \{0,1\}^{\ell_i}$. Pour une instanciation optimale, on suppose que pour tout i, $\ell_i = 2k$ (la probabilité de demander par hasard

une requête vaut $q_{h_i}/2^{\ell_i}$ et celle de collisions est ${q_{h_i}}^2/2^{\ell_i}$), où k est un paramètre de sécurité. Finalement, on suppose l'existence d'un schéma de *one-time* signature $(\mathsf{SKG}, \mathsf{Sign}, \mathsf{Ver})$, SKG étant l'algorithme de génération de clef de signature, Sign l'algorithme de signature et Ver l'algorithme de vérification (voir page 24). Notons que l'algorithme ne nécessite pas une *one-time* signature *forte* : ici, l'adversaire est autorisé à poser une seule requête d'oracle de signature, et il ne doit pas être capable de construire une signature d'un autre authentifiant.

Informellement, et en oubliant les détails, l'algorithme peut être décrit comme suit : d'abord, chaque joueur choisit un exposant aléatoire x_i et calcule $z_i = g^{x_i}$ et un chiffrement z_i^* de z_i. Il applique alors SKG pour générer une paire $(\mathsf{SK}_i, \mathsf{VK}_i)$ de clefs de signature, et il met en gage dans c_i les valeurs VK_i et z_i^*. À la seconde étape, il révèle ces valeurs (l'utilité de la mise en gage sera expliquée plus loin). Nous soulignons que le deuxième tour ne commence pas avant que toutes les mises en gage aient été reçues. À ce stade, l'identifiant de session devient $\mathsf{ssid}' = \mathsf{ssid}\|c_1\|\ldots\|c_n$: il sera inclus, et vérifié, dans toutes les valeurs hachées suivantes. Ensuite, après vérification des mises en gage des autres joueurs, chaque couple $(\mathrm{P}_i, \mathrm{P}_{i-1})$ de joueurs calcule une valeur Diffie-Hellman commune $\mathrm{Z}_i = g^{x_i\, x_{i-1}}$, ce qui mène à une valeur hachée X_i pour chaque joueur. Chaque joueur met alors en gage cette valeur X_i, et une fois toutes ces mises en gage reçues, il révèle cette valeur X_i. À l'étape suivante, les joueurs vérifient ce deuxième tour de mises en gage et calculent un authentifiant et la signature associée. Finalement, les joueurs vérifient ces authentifiants et leurs signatures, et s'ils sont tous corrects, ils calculent la clef de session et notent leur session `completed`.

Dès qu'une valeur reçue par P_i ne coïncide pas avec la valeur espérée, il abandonne, en posant sk_i = `error`. En particulier, chaque joueur vérifie que $c_i = \mathcal{H}_3(\mathsf{ssid}, z_i^*, \mathsf{VK}_i, i)$, $c_i' = \mathcal{H}_4(\mathsf{ssid}', \mathrm{X}_i, i)$ et $\mathsf{Ver}(\mathsf{VK}_i, \mathsf{Auth}_i, \sigma_i) = 1$.

Nous mettons maintenant en évidence quelques-unes des différences entre le schéma proposé dans [ABCP06] et le nôtre, décrit sur la figure 7.2 . D'abord, notre construction ne requiert pas de nonce aléatoire dans la première étape (l'identifiant de session construit après le premier tour est suffisant). En outre, pour implémenter correctement la fonctionnalité, nous retournons un message d'erreur aux joueurs quand ils ne partagent pas le même mot de passe (authentification mutuelle). Par simplicité, nous avons aussi ajouté deux modifications supplémentaires, pas directement reliées à la fonctionnalité. D'une part, nous utilisons un chiffrement idéal étiqueté à la place d'une clef symétrique différente pour chaque joueur. D'autre part, les valeurs X_i sont calculées ici comme le xor de deux hachés et non comme un quotient.

Grâce à la fonctionnalité de répartition, les joueurs sont partitionnés suivant les valeurs reçues pendant le premier tour (*i.e.* avant la ligne pointillée de la figure 7.2). Ils partagent tous les c_i – et donc les z_i^* et VK_i grâce à l'oracle aléatoire $\mathcal{H}_3$ – et l'identifiant de session devient $\mathsf{ssid}' = \mathsf{ssid}\|c_1\|\ldots\|c_n$. Au troisième tour, la signature ajoutée à l'authentifiant empêche l'adversaire d'être capable de changer un authentifiant en une autre valeur. Au début de chaque tour, les joueurs attendent d'avoir reçu toutes les autres valeurs des tours précédents avant d'envoyé la nouvelle. Ceci est particulièrement important entre les tours `flow(1a)` et `flow(1b)` et de même entre `flow(2a)` et `flow(2b)`. Comme l'identifiant de session ssid' est inclus dans toutes les valeurs hachées et dans la signature, seuls les joueurs d'un même sous-groupe peuvent accepter et obtenir une clef commune.

La propriété de *contributory* est assurée par la modification suivante : dans les première et seconde étapes, chaque joueur commence par envoyer une mise en gage de la valeur qu'il vient de calculer (en utilisant un oracle aléatoire), notée c_i puis c_i'. Grâce à cette mise en gage, il est impossible à un joueur de calculer z_i^* (ou X_i) après avoir vu les valeurs des autres : chaque joueur doit mettre en gage son z_i^* (ou X_i) au même moment, et cette valeur ne peut pas dépendre des autres valeurs envoyées par les joueurs.

Enfin, signalons que nous n'avons pas besoin de supposer dans notre preuve de sécurité que les joueurs effacent une valeur éphémère avant la fin du calcul de la clef de session.

Hypothèse de sécurité. Le protocole repose sur l'hypothèse CDH, définie page 19. Il est facile de voir que si P_i et P_{i+1} ont les mêmes mots de passe, alors ils vont calculer à l'étape 2 la même valeur $\mathrm{Z}_{i+1} = (z_{i+1})^{x_i} = g^{x_i x_{i+1}} = (z_i)^{x_{i+1}}$. Si les mots de passe sont différents, on

Fig. 7.2 – Description du protocole pour le joueur P_i, avec l'indice i et le mot de passe pw_i

(1a) $x_i \xleftarrow{\mathrm{R}} \mathbb{Z}_q^*$ $\quad z_i = g^{x_i}$ $\quad z_i^* = \mathcal{E}_{\mathsf{pw}_i}^{\mathsf{ssid},i}(z_i)$
$(\mathsf{VK}_i, \mathsf{SK}_i) \leftarrow \mathsf{SKG}$ $\quad c_i = \mathcal{H}_3(\mathsf{ssid}, z_i^*, \mathsf{VK}_i, i)$ $\xrightarrow{\quad c_i \quad}$

. . . . . .

Après ce point, l'identifiant de session devient $\mathsf{ssid}' = \mathsf{ssid} \| c_1 \| \ldots \| c_n$.

(1b) envoie z_i^* et VK_i $\xrightarrow{\quad z_i^*, \mathsf{VK}_i \quad}$

(2a) vérifie si $c_j = \mathcal{H}_3(\mathsf{ssid}, z_j^*, \mathsf{VK}_j, j) \quad \forall j \neq i$
et abandonne si l'une de ces valeurs est incorrecte
$z_{i-1} = \mathcal{D}_{\mathsf{pw}_i}^{\mathsf{ssid},i-1}(z_{i-1}^*)$ $\quad \mathrm{Z}_i = (z_{i-1})^{x_i}$
$z_{i+1} = \mathcal{D}_{\mathsf{pw}_i}^{\mathsf{ssid},i+1}(z_{i+1}^*)$ $\quad \mathrm{Z}_{i+1} = (z_{i+1})^{x_i}$
$h_i = \mathcal{H}_2(\mathrm{Z}_i)$ $\quad h_{i+1} = \mathcal{H}_2(\mathrm{Z}_{i+1})$
$\mathrm{X}_i = h_i \oplus h_{i+1}$ $\quad c_i' = \mathcal{H}_4(\mathsf{ssid}', \mathrm{X}_i, i)$ $\xrightarrow{\quad c_i' \quad}$

(2b) envoie X_i $\xrightarrow{\quad \mathrm{X}_i \quad}$

(3) vérifie si $c_j' = \mathcal{H}_4(\mathsf{ssid}', \mathrm{X}_j, j) \quad \forall j \neq i$
et abandonne si l'une de ces valeurs est incorrecte
$h_{j+1} = \mathrm{X}_j \oplus h_j \qquad \forall j = i, \ldots, n+i-1 \pmod n$
$\mathsf{Auth}_i = \mathcal{H}_1(\mathsf{ssid}', (z_1^*, \mathrm{X}_1, h_1), \ldots, (z_n^*, \mathrm{X}_n, h_n), i)$
$\sigma_i = \mathsf{Sign}(\mathsf{SK}_i, \mathsf{Auth}_i)$ $\xrightarrow{\quad \sigma_i \quad}$

(4) calcule tous les Auth_j et vérifie $\mathsf{Ver}(\mathsf{VK}_j, \mathsf{Auth}_j, \sigma_j)$. Si ces valeurs sont correctes, noter la session `completed` et fixer $sk_i = \mathcal{H}_0(\mathsf{ssid}', h_1, \ldots, h_n)$. Sinon, fixer $sk_i = \texttt{error}$.

appelle $\mathrm{Z}_i^{\mathrm{R}}$ la valeur calculée par P_i sur son côté droit, et $\mathrm{Z}_{i+1}^{\mathrm{L}}$ la valeur calculée par P_{i+1} sur son côté gauche. On a alors :

$$\mathrm{Z}_i^{\mathrm{R}} = \mathsf{CDH}_g(\mathcal{D}_{\mathsf{pw}_i}^{\mathsf{ssid},i-1}(z_{i-1}^*), \mathcal{D}_{\mathsf{pw}_i}^{\mathsf{ssid},i}(z_i^*)) \qquad h_i^{\mathrm{R}} = \mathcal{H}_2(\mathrm{Z}_i^{\mathrm{R}})$$
$$\mathrm{Z}_{i+1}^{\mathrm{L}} = \mathsf{CDH}_g(\mathcal{D}_{\mathsf{pw}_i}^{\mathsf{ssid},i}(z_i^*), \mathcal{D}_{\mathsf{pw}_i}^{\mathsf{ssid},i+1}(z_{i+1}^*)) \qquad h_{i+1}^{\mathrm{L}} = \mathcal{H}_2(\mathrm{Z}_{i+1}^{\mathrm{L}})$$

(ici, CDH_g symbolise la fonction Diffie-Hellman en base g qui, sur l'entrée g^a, g^b renvoie g^{ab}), et les valeurs $\mathrm{Z}_{i+1}^{\mathrm{L}}$ et $\mathrm{Z}_{i+1}^{\mathrm{R}}$ ont de bonnes chances d'être différentes.

Graphiquement, la situation peut être résumée comme suit :

$$\mathrm{P}_{i-1} \xleftrightarrow{\mathrm{Z}_i^{\mathrm{L}} \neq \mathrm{Z}_i^{\mathrm{R}}} \mathrm{P}_i \xleftrightarrow{\mathrm{Z}_{i+1}^{\mathrm{L}} \neq \mathrm{Z}_{i+1}^{\mathrm{R}}} \mathrm{P}_{i+1}$$

Chaque joueur P_i calcule $\mathrm{X}_i = h_i^{\mathrm{R}} \oplus h_{i+1}^{\mathrm{L}}$ et ainsi, une fois les valeurs X_j publiées, tous les joueurs peuvent itérativement calculer tous les h_j requis pour obtenir les authentifiants Auth_i et ensuite la clef de session sk_i.

(collisions),

Le théorème principal. Soit $\widehat{s\mathcal{F}}_{\mathrm{GPAKE}}$ l'extension multi-session de la fonctionnalité de répartition $s\mathcal{F}_{\mathrm{GPAKE}}$ et soient $\mathcal{F}_{\mathrm{RO}}$ et $\mathcal{F}_{\mathrm{ITC}}$ les fonctionnalités idéales qui procurent un oracle aléatoire et un chiffrement idéal étiqueté à tous les joueurs.

Théorème 2. *Le protocole présenté sur la figure 7.2 réalise de manière sûre la fonctionnalité* $\widehat{s\mathcal{F}}_{\mathrm{GPAKE}}$ *dans le modèle* $(\mathcal{F}_{\mathrm{RO}}, \mathcal{F}_{\mathrm{ITC}})$*-hybride, en présence d'adversaires adaptatifs, et il est* $(n/2, n)$-contributory.

Corollaire 2.1. *Ce protocole, modifié selon les indications décrites dans la section 7.4.1 (avec $n/2$ exécutions parallèles du schéma original), réalise de manière sûre $\widehat{s\mathcal{F}}_{\text{GPAKE}}$ dans le modèle $(\mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{ITC}})$-hybride, en présence d'adversaires adaptatifs, et il est $(n-2, n)$*-contributory.

Il est à noter que si le schéma de signature est vierge de tout canal subliminal [Sim84] (par exemple si une seule signature σ est valide pour une paire donnée $(\mathsf{VK}, \mathsf{Auth})$) alors, après le premier message $(1a)$, tout est déterministe, et aucun canal subliminal n'est accessible à un adversaire pour faire fuir de l'information à un espion.

7.3 Preuve du théorème 2

7.3.1 Idée de la preuve

Nous montrons à présent que le protocole est $(n/2, n)$-*contributory*, et nous expliquerons dans la section 7.4.1 comment obtenir la $(n-2, n)$-*contributory*, en utilisant des exécutions parallèles. Ici, $n/2$ signifie implicitement $\lfloor n/2 \rfloor$ lorsque n est impair.

Commençons par exhiber une attaque montrant que la $(n/2+1)$-*contributory* ne peut pas être atteinte. Par simplicité, supposons n impair, et considérons la situation suivante dans laquelle il y a $\lfloor n/2 \rfloor$ joueurs honnêtes (appelés P_i) et $\lfloor n/2 \rfloor + 1$ joueurs corrompus (appelés $\mathcal{A}_i$, puisqu'ils sont sous le contrôle de l'adversaire $\mathcal{A}$) :

$$\mathrm{P}_1 \quad \mathcal{A}_1 \quad \mathrm{P}_2 \quad \mathcal{A}_2 \quad \mathrm{P}_3 \ldots \mathrm{P}_{\lfloor n/2 \rfloor} \quad \mathcal{A}_{\lfloor n/2 \rfloor} \quad \mathcal{A}_{\lfloor n/2 \rfloor + 1}$$

Comme $\mathcal{A}$ connaît les aléas des joueurs corrompus, il apprend à partir des messages $(1b)$ les valeurs Z_i et donc $h_i = \mathcal{H}_2(\mathrm{Z}_i)$ pour $i = 1, \ldots, n-1$, avant de participer au nom de $\mathcal{A}_{\lfloor n/2 \rfloor + 1}$. Même s'il ne peut plus modifier l'élément z_n de l'étape $(1b)$ (déjà mise en gage dans c_n), il peut choisir à l'étape $(2a)$ la valeur Z_n pour biaiser h_n, et donc la clef finale, définie comme $sk = \mathcal{H}_0(\mathsf{ssid}', h_1, \ldots, h_n)$. Cette attaque est possible étant donné qu'aucun joueur honnête ne peut vérifier la valeur de Z_n : c'est un Diffie-Hellman entre deux joueurs corrompus.

Plus généralement, si $\mathcal{A}$ contrôle suffisamment de joueurs pour que chaque joueur honnête soit entre deux joueurs corrompus, alors il peut apprendre, à l'aide des messages $(1b)$, les valeurs X_i envoyées dans les messages $(2b)$. Si deux joueurs corrompus sont voisins, ils peuvent envoyer la valeur X_i de leur choix, puisqu'elle provient d'un Diffie-Hellman entre eux deux. Dans l'attaque ci-dessus, l'adversaire pourrait apprendre tous les h_i suffisamment tôt, de telle sorte que son contrôle sur h_n pourrait biaiser la clef. À l'inverse, s'il peut simplement contrôler un h_i, mais sans connaître les autres valeurs, il reste suffisamment d'entropie dans la dérivation de clef, ce qui montre que la clef finale est uniformément distribuée et donne la propriété de *contributory*.

On prouve désormais la $(n/2, n)$-*contributory*, en utilisant l'intuition décrite ci-dessus. On doit construire, pour tout adversaire réel $\mathcal{A}$ interagissant avec de vrais joueurs exécutant le protocole, un adversaire idéal $\mathcal{S}$ interagissant avec des joueurs virtuels et la fonctionnalité $\widehat{s\mathcal{F}}_{\text{GPAKE}}$ de telle manière qu'aucun environnement $\mathcal{Z}$ ne puisse distinguer une exécution avec $\mathcal{A}$ dans le monde réel d'une exécution avec $\mathcal{S}$ dans le monde idéal avec probabilité non-négligeable.

On utilise comme à la page 93 des requêtes hybrides : `GoodPwd` (pour `TestPwd`) vérifie si le mot de passe d'un joueur est celui auquel on pense ; `SamePwd` (pour `NewKey`) vérifie si les joueurs partagent le même mot de passe, sans le révéler ; `Delivery` (pour `KeyDelivery`) donne la clef de session au joueur en question. Notons qu'ici, le simulateur connaît toujours les mots de passe mis en gage par l'adversaire pour les utilisateurs corrompus grâce au chiffrement idéal.

On utilise également les notions de messages *oracle-generated* et non *oracle-generated* décrites au même endroit.

On définit une suite incrémentale de jeux à partir de l'exécution réelle du protocole dans le monde réel jusqu'au jeu $\mathbf{G_8}$ que nous prouvons être indistinguable du jeu idéal. Le point-clef de cette preuve va être $\mathbf{G_7}$. $\mathbf{G_0}$ est le jeu réel. Dans $\mathbf{G_1}$, on simule les requêtes de chiffrement, déchiffrement et hachage, en éliminant certains événements improbables (comme les collisions). Grâce au chiffrement idéal (voir les détails dans la section 7.3.2), on peut extraire les mots

de passe utilisés par l'adversaire $\mathcal{A}$ pour les joueurs corrompus depuis le début de la session. $\mathbf{G_2}$ et $\mathbf{G_3}$ permettent à $\mathcal{S}$ d'être sûr que les authentifiants pour les joueurs honnêtes sont toujours *oracle-generated*. Dans $\mathbf{G_4}$, on montre comment simuler les premiers messages. Dans $\mathbf{G_5}$, on gère le cas passif, où tous les messages sont *oracle-generated*. Dans $\mathbf{G_6}$, on s'occupe des messages (éventuellement) non *oracle-generated* à partir de la deuxième étape. $\mathbf{G_7}$ est le jeu crucial, dans lequel on montre comment simuler les joueurs honnêtes sans connaître leurs mots de passe, même dans le cas où des corruptions ont eu lieu avant la deuxième étape. Finalement, on montre que $\mathbf{G_8}$, dans lequel on ne fait que remplacer les requêtes hybrides par les requêtes réelles, est indistinguables du jeu idéal.

Comme on considère la fonctionnalité de répartition, les joueurs ont été partitionnés en sous-ensembles en fonction des valeurs reçues dans le premier message `flow(1a)`. On peut donc supposer dans la suite que tous les joueurs ont reçu les mêmes `flow(1a)` et `flow(1b)` grâce à la propriété de *binding* de la mise en gage $\mathcal{H}_3$ (la définition de cette propriété peut être trouvée page 25). Des messages `flow(1a)` *oracle-generated* ont été envoyés par des joueurs qui seront considérés honnêtes dans cette session, tandis que les messages non *oracle-generated* ont été envoyés par l'adversaire, et les joueurs correspondants vont être considérés comme corrompus depuis le début de la session : l'adversaire a donc choisi leurs mots de passe. Un avantage de ce modèle (qui utilise un oracle aléatoire pour la mise en gage et un chiffrement idéal) est que l'on connaît les mots de passe utilisés par l'adversaire (et donc les joueurs corrompus) dans le premier message, simplement en jetant un œil aux tables qui seront définies dans $\mathbf{G_2}$, et ces mots de passe sont les mêmes dans la vue de n'importe quel joueur honnête. L'extraction d'un mot de passe peut échouer si l'adversaire a calculé le chiffré ou la mise en gage au hasard, mais alors il n'a aucune chance de mener le protocole à terme avec succès.

De la même manière, si `flow(2a)` est *oracle-generated*, alors `flow(2b)` doit l'être aussi (sauf avec probabilité négligeable) grâce à $\mathcal{H}_4$, comme ci-dessus. Ainsi, on pose $\mathsf{sk}_i = \texttt{error}$ dès qu'une incohérence est repérée par un joueur (ouverture de mise en gage incorrecte ou signature invalide). Ce dernier abandonne alors l'exécution.

Corruptions adaptatives et composantes connexes. Par simplicité, considérons que le simulateur maintient, pour chaque joueur honnête, une liste symbolisant son état interne :

$$\Lambda_i = (\mathsf{pw}_i, \mathrm{SK}_i, x_i, z_i, z_i^*, c_i, \mathrm{Z}_i^{\mathrm{R}}, \mathrm{Z}_{i+1}^{\mathrm{L}}, h_i^{\mathrm{R}}, h_{i+1}^{\mathrm{L}}, \mathrm{X}_i, c_i'),$$

où les indices L et R indiquent avec quel voisin la valeur est partagée (gauche ou droite). Quand un joueur se fait corrompre, le simulateur doit donner cette liste à l'adversaire. La plupart des champs vont être choisis au hasard pendant la simulation, et les autres valeurs sont initialisées à $\perp$. Dès qu'un joueur se fait corrompre, le simulateur récupère son mot de passe, qui va l'aider à remplir les autres champs de façon cohérente, grâce à l'oracle aléatoire programmable et au chiffrement idéal.

La connaissance du mot de passe de P_i aide ainsi le simulateur à remplir son état interne (voir ci-dessous). Cela permet aussi à $\mathcal{S}$ d'envoyer les valeurs (en particulier X_i) de façon cohérente avec la vision de l'adversaire et de l'environnement. Informellement, $\mathcal{S}$ réalise cela en partitionnant l'ensemble des joueurs en un certain nombre de composantes connexes (le détail du partitionnement sera expliqué dans le jeu $\mathbf{G}_7$ page 112). Chaque composante consiste en tous les joueurs voisins partageant le même mot de passe (celui utilisé pour générer le premier message). On montre plus loin que tout ce que $\mathcal{S}$ a à faire pour faire marcher la simulation est de s'assurer que les valeurs produites sont cohérentes pour les joueurs appartenant aux mêmes composantes. Ainsi, pour des joueurs voisins dans des composantes différentes, $\mathcal{S}$ peut pratiquement envoyer des valeurs complètement indépendantes, sans craindre d'être pris en faute, étant donné que les valeurs z_i déchiffrées n'ont aucun lien entre elles, et ce depuis le début du protocole.

Simulateur : initialisation de la session. L'objectif du premier message est de créer les sous-groupes H de joueurs intervenant dans la même (sous-)exécution du protocole (voir la fonctionnalité de répartition, décrite sur la figure 5.10 page 86). $\mathcal{S}$ choisit les valeurs $(\mathsf{SK}_i, \mathsf{VK}_i)$ au nom des joueurs honnêtes et fixe au hasard la valeur z_i^* plutôt que de demander une requête

de déchiffrement (il ne connaît pas les mots de passe des joueurs). Il calcule alors les mises en gage c_i et les envoie à l'adversaire. L'environnement initialise une session pour chaque joueur (virtuel) honnête, ce qui est modélisé par les requêtes `Init` envoyées à la fonctionnalité de répartition. À partir de sa vision des c_i des joueurs honnêtes, l'adversaire choisit les sous-groupes qu'il a envie de créer et envoie alors c_i au nom des joueurs qu'il veut contrôler (ils vont devenir corrompus dès le début de la session). Ceci définit les ensembles H suivant les valeurs c_j reçues : les joueurs honnêtes qui ont reçu le même ensemble $\{c_j\}$ de valeurs (éventuellement modifiées par l'adversaire) sont dans le même sous-groupe H. Le simulateur transmet ces ensembles (qui forment une partition de tous les joueurs honnêtes) à la fonctionnalité de répartition. Cette dernière initialise alors les fonctionnalités idéales avec ssid_H pour chaque sous-groupe H : tous les joueurs dans la même session (ssid_H) ont alors reçu le même $\{c_j\}$. L'environnement reçoit en retour la répartition via les joueurs virtuels, et envoie alors les requêtes `NewSession` en leur nom, selon le ssid_H approprié. Le simulateur utilise les mises en gage envoyées par l'adversaire pour extraire le mot de passe utilisé (grâce au chiffrement idéal), et envoie alors les requêtes `NewSession` appropriées au nom des joueurs corrompus (notons que si aucun mot de passe n'est extrait, un mot de passe aléatoire est utilisé). Ainsi, on se concentre sur une session spécifique $\mathsf{ssid}' = \mathsf{ssid}_\mathrm{H}$ pour un certain ensemble H.

Simulateur : idée principale. La simulation du reste du protocole dépend de la connaissance des mots de passe par le simulateur. D'abord, s'il connaît le mot de passe d'un joueur, il fait tout honnêtement pour chaque joueur de sa composante connexe. Sinon, il fait tout au hasard. En cas de corruption, $\mathcal{S}$ apprend son mot de passe et peut programmer les oracles et remplir son état interne (et celui de tous les joueurs dans sa composante connexe) de façon cohérente. La dernière phase (la programmation des oracles) peut échouer, mais seulement si l'adversaire peut résoudre un problème difficile (CDH).

Plus précisément, dans la plupart des cas, le simulateur $\mathcal{S}$ se contente de suivre le protocole au nom des joueurs honnêtes. La principale différence entre les joueurs simulés et les joueurs honnêtes est que $\mathcal{S}$ ne s'engage pas sur un mot de passe particulier. Cependant, si $\mathcal{A}$ génère ou modifie un message `flow(1a)` ou `flow(1b)` envoyé à P_i dans la session ssid', alors $\mathcal{S}$ extrait le mot de passe pw grâce au chiffrement idéal et l'utilise dans une requête `TestPwd` pour P_i à la fonctionnalité. Si c'est correct, alors $\mathcal{S}$ utilise ce mot de passe au nom de P_i et continue la simulation.

Le point-clef de la simulation consiste à envoyer des X_i cohérents, dont les valeurs déterminent complètement la clef de session. Dans ce but, on considère deux cas. Premièrement, si les joueurs sont honnêtes et partagent le même mot de passe, les h_i sont choisis au hasard mais de façon identique entre deux voisins. S'ils ne partagent pas les mêmes mots de passe, ces valeurs sont fixées au hasard. Deuxièmement, s'il y a des joueurs corrompus, le simulateur détermine les composantes connexes comme dans le jeu $\mathbf{G}_7$ expliqué page 112 et l'astuce consiste à rendre la simulation cohérente uniquement dans ces composantes, étant donné que l'adversaire n'a aucun moyen de deviner ce qui s'est passé entre deux composantes (les mots de passe sont différents).

Si une session échoue ou termine, $\mathcal{S}$ le rapporte à $\mathcal{A}$. Si la session termine avec la clef de session sk, alors $\mathcal{S}$ pose une requête `KeyDelivery` à $\widehat{\mathcal{F}}_{\mathrm{GPAKE}}$, en spécifiant la clef de session. Mais sauf si suffisamment de joueurs sont corrompus, $\widehat{\mathcal{F}}_{\mathrm{GPAKE}}$ va ignorer la clef spécifiée par $\mathcal{S}$: il est donc inutile de s'en préoccuper dans ce cas.

7.3.2 Description des jeux

Jeu $\mathbf{G}_0$: Le jeu $\mathbf{G}_0$ est le jeu réel dans les modèles idéaux ROM (*random oracle model*) et ITCM (*ideal tweakable cipher model*) : $\mathcal{S}$ exécute le protocole pour les joueurs honnêtes comme ces derniers le feraient, en utilisant les mots de passe envoyés par l'environnement.

Jeu $\mathbf{G}_1$: Nous modifions le jeu précédent en simulation les oracles de hachage et de chiffrement/déchiffrement, d'une manière naturelle et usuelle, de façon similaire à la preuve du cha-

pitre précédent, page 93 : la seule différence est l'ajout d'une étiquette en plus de la clef. Ainsi, pour le chiffrement idéal étiqueté, nous autorisons $\mathcal{S}$ à maintenir une liste Λ_ε d'entrées (questions, réponses) de longueur $q_\varepsilon + q_\mathcal{D}$, composée de deux sous-listes : $\{(\mathsf{pw}, (\mathsf{ssid}, i), \mathrm{Y}, \alpha, \mathcal{E}, \mathrm{Y}^*)\}$ pour les requêtes de chiffrement, et $\{(\mathsf{pw}, (\mathsf{ssid}, i), \mathrm{Y}, \alpha, \mathcal{D}, \mathrm{Y}^*)\}$ (pour les déchiffrements). La première (resp. seconde) sous-liste est utilisée pour indiquer que l'élément Y (resp. Y^*) a été chiffré (« $\mathcal{E}$ ») (resp. déchiffré (« $\mathcal{D}$ »)) pour produire le chiffré Y^* (resp. le clair Y) via un algorithme de chiffrement symétrique qui utilise la clef pw et l'étiquette (ssid, i). Plus précisément, pour une nouvelle requête de chiffrement, Y^* est choisi aléatoirement, et α est fixé égal à $\perp$. Pour une nouvelle requête de déchiffrement (d'une valeur qui n'a pas été obtenue par chiffrement), α est choisi aléatoirement dans $\mathbb{Z}_q^*$, et Y est fixé égal à g^α. Une telle liste est utilisée par $\mathcal{S}$ pour être capable de donner des réponses cohérentes avec les contraintes suivantes :

1. la même question pour la même paire (mot de passe, étiquette) va recevoir la même réponse ;
2. le schéma simulé (pour chaque paire (mot de passe, étiquette)) est une permutation sur G ;
3. afin d'aider $\mathcal{S}$ à extraire plus tard le mot de passe utilisé dans le premier message, il ne doit pas y avoir d'entrées de chiffrement de la forme (question, réponse) avec un chiffré et une étiquette identiques, mais des mots de passe différents : un chiffré (obtenu par chiffrement) doit correspondre à un mot de passe unique. De même, tout déchiffrement d'un chiffré non obtenu par chiffrement va permettre d'obtenir le logarithme discret en même temps.

Le simulateur gère cette liste de requêtes de chiffrement et déchiffrement à travers les règles suivantes :

- Pour une requête de chiffrement $\mathcal{E}_{\mathsf{pw}}^{\mathsf{ssid},i}(\mathrm{Y})$ telle que $(\mathsf{pw}, (\mathsf{ssid}, i), \mathrm{Y}, *, *, \mathrm{Y}^*)$ apparaît dans la liste Λ_ε, la réponse est Y^*. Sinon, choisir aléatoirement un élément $\mathrm{Y}^* \in \mathrm{G}^*$. Si $(*, (\mathsf{ssid}, i), *, *, *, \mathrm{Y}^*)$ appartient déjà à Λ_ε, alors abandonner, sinon ajouter la nouvelle entrée $(\mathsf{pw}, (\mathsf{ssid}, i), \mathrm{Y}, \perp, \mathcal{E}, \mathrm{Y}^*)$ à la liste.
- Pour une requête de déchiffrement $\mathcal{D}_{\mathsf{pw}}^{\mathsf{ssid},i}(\mathrm{Y}^*)$ telle que $(\mathsf{pw}, (\mathsf{ssid}, i), \mathrm{Y}, *, *, \mathrm{Y}^*)$ apparaît dans la liste Λ_ε, la réponse est Y. Sinon, choisir aléatoirement un élément $\mathrm{Y} \in \mathrm{G}^*$. Si $(*, (\mathsf{ssid}, i), \mathrm{Y}, *, *, \mathrm{Y}^*)$ appartient déjà à Λ_ε, alors abandonner, sinon ajouter la nouvelle entrée $(\mathsf{pw}, (\mathsf{ssid}, i), \mathrm{Y}, \perp, \mathcal{D}, \mathrm{Y}^*)$ à la liste.

$\mathcal{S}$ maintient aussi une liste $\Lambda_\mathcal{H}$ de triplets (i, q, r) où $\mathcal{H}_i(q) = r$, utilisée pour gérer proprement les requêtes aux oracles aléatoires $\mathcal{H}_i$, en excluant aussi les collisions. Il utilise pour cela la règle générale suivante (où n appartient à $\{0, 1, 2, 3, 4\}$) :

- Pour une requête de hachage $\mathcal{H}_n(q)$ telle que (n, q, r) apparaît dans la liste $\Lambda_\mathcal{H}$, la réponse est r. Sinon, choisir aléatoirement un élément $r \in \{0,1\}^{\ell_n}$. Si $(n, *, r)$ appartient déjà à $\Lambda_\mathcal{H}$, abandonner, sinon ajouter (n, q, r) à la liste.

Le paradoxe des anniversaires implique que $\mathbf{G}_1$ et $\mathbf{G}_0$ sont statistiquement indistinguables pour des longueurs de sortie des oracles bien choisies.

Jeu $\mathbf{G}_2$: Dans ce jeu, le simulateur rejette tout authentifiant qui a été envoyé sans avoir été demandé à l'oracle $\mathcal{H}_1$. Cela fait une différence si jamais cela lui fait rejeter un authentifiant qui était en fait valide. Mais ceci ne peut bien sûr arriver qu'avec probabilité négligeable : $\mathbf{G}_2$ et $\mathbf{G}_1$ sont statistiquement indistinguables.

Jeu $\mathbf{G}_3$: Le simulateur rejette de plus tout authentifiant non *oracle-generated* si le joueur concerné est encore honnête : l'adversaire ne connaît pas la clef de signature, et ne pourra donc pas signer correctement cet authentifiant. On peut facilement montrer qu'une différence entre $\mathbf{G}_2$ et $\mathbf{G}_3$ mènerait à une attaque contre le schéma de *one-time* signature.

Jeu $\mathbf{G}_4$: Dans ce jeu, on modifie formellement la façon dont on simule les joueurs honnêtes, dont le simulateur connaît encore les mots de passe. Dans la première étape, $\mathcal{S}$ envoie une valeur c_i au nom de P_i et ensuite, quand tous les c_j ont été envoyés, il envoie une valeur z_i^* aléatoire (choisie sans requête à l'oracle de chiffrement, puisque nous essayons d'éviter l'utilisation des mots de passe) ainsi que la clef de vérification VK_i, générée honnêtement. Finalement, $\mathcal{S}$ pose $c_i = \mathcal{H}_3(\mathsf{ssid}, z_i^*, \mathsf{VK}_i, i)$ en programmant l'oracle. Il y a un risque négligeable que la simulation échoue, si l'adversaire a déjà posé cette requête à l'oracle (cette probabilité est bornée par q_{h_3}/q). Au deuxième tour, $\mathcal{S}$ continue la simulation en utilisant cette fois le mot de passe pw_i : il pose trois requêtes de déchiffrement, pour z_i^*, z_{i-1}^*, et z_{i+1}^*, avec la clef pw_i et les étiquettes appropriées. Étant donné la manière dont l'oracle de déchiffrement est simulé, on apprend z_{i-1}, z_i, et z_{i+1}, ainsi que le logarithme discret x_i en base g de z_i, sauf si les requêtes de chiffrement correspondantes ont été posées au préalable (auquel cas x_i n'est pas initialisé). Comme notre simulation ne calcule aucun chiffrement, ceci n'arrive que si z_i^* a été obtenu par un chiffrement par l'adversaire, c'est-à-dire avec probabilité négligeable. Une fois que l'on a x_i, on peut alors conclure en calculant Z_i et Z_{i+1}, et ensuite h_i, h_{i+1} et X_i. Un tel changement dans le processus n'altère pas la vue d'un adversaire puisque que c'est une réécriture purement syntaxique si on a exclu les échecs avant.

Corruptions. Si P_i se fait corrompre, le simulateur doit donner son état interne (en particulier son exposant privé x_i) à $\mathcal{A}$. Il s'agit en fait du logarithme discret obtenu pendant la simulation de l'oracle de déchiffrement sur z_i^*. Ceci est possible avec la même probabilité (négligeable) d'échec.

$\mathbf{G}_4$ est alors statistiquement indistinguable de $\mathbf{G}_3$. Notons que l'on n'utilise pas le mot de passe avant la deuxième étape. Nous allons désormais essayer d'éviter complètement de l'utiliser avant une corruption.

Jeu $\mathbf{G}_5$: Dans ce jeu, nous gérons le cas passif, dans lequel tous les messages sont *oracle-generated*. La simulation des trois premiers messages est effectuée comme dans le jeu précédent $\mathbf{G}_4$ et nous considérons ensuite deux cas. Si tous les messages étaient *oracle-generated* jusqu'à la quatrième étape, on demande à $\mathcal{S}$ de simuler la fin de l'exécution du protocole au nom de tous les joueurs. Sinon, il suit simplement le protocole comme avant.

Dans le premier cas (avec uniquement des messages *oracle-generated* jusqu'à la quatrième étape), le simulateur commence cette étape en posant une requête `SamePwd`. Comme on suppose que $\mathcal{S}$ connaît tous les mots de passe, cela revient à vérifier que tous les mots de passe sont en fait les mêmes. On distingue alors deux cas : s'ils sont tous les mêmes, $\mathcal{S}$ pose une requête `Delivery` avec une clef aléatoire et le mot-clef `b` égal à `yes` à la fonctionnalité pour chaque joueur. Sinon, tous les joueurs reçoivent un message d'erreur. Un problème n'apparaît que si $\mathcal{A}$ pose la requête correspondante à $\mathcal{H}_0$, mais alors il doit avoir obtenu les valeurs h_i et l'événement `AskH`, décrit ci-dessous, apparaît.

Remarque 1. Notons que si tous les joueurs ont le même mot de passe, une telle stratégie rend le jeu (presque) indistinguable du précédent. Si, à l'inverse, les joueurs ne partagent pas tous le même mot de passe, ils vont finalement calculer des Auth_i différents, obtenant ainsi un message d'erreur (sauf avec probabilité négligeable). Cependant, un problème peut être soulevé si l'adversaire pose les requêtes correctes à $\mathcal{H}_2$, ce qui est la seule façon, étant donné tous les X_i, d'avoir suffisamment d'information sur tous les h_i. En effet, tous les X_i forment $(n-1)$ combinaisons linéaires indépendantes des n variables h_i. Fixer l'un des h_{i_0} à une valeur aléatoire est une autre équation indépendante. On a donc une solution unique $(h_1, \ldots, h_n)$ à ce système de n équations et n inconnues, quelle que soit la valeur donnée à ce h_{i_0}. Ceci signifie que la valeur de h_{i_0} est imprédictible et ne peut pas être devinée sans avoir posé la requête correcte à $\mathcal{H}_2$. Cependant, un tel événement ne peut arriver qu'avec probabilité négligeable, nous l'appelons `AskH` :

> `AskH` : $\mathcal{A}$ pose à l'oracle $\mathcal{H}_2$ la requête sur la valeur r telle qu'il existe i vérifiant $r = \mathrm{CDH}_g(\mathcal{D}_{\mathsf{pw}_i}^{\mathsf{ssid},i-1}(z_{i-1}^*), \mathcal{D}_{\mathsf{pw}_i}^{\mathsf{ssid},i}(z_i^*))$ et qu'il n'existe pas d'entrée de la forme $(\mathsf{pw}_i, (\mathsf{ssid}, i-1), *, *, \mathcal{E}, z_{i-1}^*)$ ou $(\mathsf{pw}_i, (\mathsf{ssid}, i), *, *, \mathcal{E}, z_i^*)$ dans la liste Λ_ε.

La condition sur les requêtes de chiffrement est importante : si z^*_{i-1} avait été obtenu par une requête de chiffrement, l'adversaire connaîtrait l'exposant secret et pourrait alors calculer le CDH de manière triviale. Comme la probabilité de `AskH` est négligeable sous l'hypothèse DDH (voir la preuve page 115), $\mathbf{G}_5$ et $\mathbf{G}_4$ sont (calculatoirement) indistinguables.

Jeu $\mathbf{G}_6$: Dans ce jeu, nous supposons que $\mathcal{S}$ connaît encore les mots de passe des joueurs et nous considérons deux cas. Soit un joueur a déjà été corrompu avant la deuxième étape, et alors le simulateur exécute tout le protocole honnêtement, comme dans $\mathbf{G}_4$. Soit aucune corruption n'a eu lieu avant la deuxième étape, et nous allons alors modifier la simulation afin de ne plus utiliser les mots de passe. Leur connaissance est encore nécessaire dans le premier cas (qui sera géré dans $\mathbf{G}_7$). Nous nous concentrons sur le deuxième cas dans ce jeu : aucune corruption n'a donc eu lieu avant la deuxième étape.

$\mathcal{S}$ commence le protocole en envoyant, au nom de chaque joueur non-corrompu P_i, des valeurs aléatoires c_i, z^*_i et VK_i, comme dans le jeu précédent. Si aucune corruption n'a eu lieu, il commence alors la deuxième étape en demandant une requête `SamePwd`. Par simplicité, nous considérons deux cas ici, en fonction de la réponse qui peut être 'yes' ou 'no'. Rappelons que les joueurs ne peuvent obtenir une clef valide partagée que dans le premier cas ; dans le second, ils vont obtenir une erreur.

Cas 1 – Les joueurs partagent tous le même mot de passe ($\mathbf{G}_{6\mathtt{a}}$) : Le simulateur envoie une valeur aléatoire c'_i au nom de chaque joueur P_i. Une fois toutes les valeurs envoyées, il choisit aléatoirement les valeurs h^{L}_i et h^{R}_i, avec les contraintes $h^{\mathrm{L}}_i = h^{\mathrm{R}}_{i-1}$ et $h^{\mathrm{R}}_i = h^{\mathrm{L}}_{i+1}$ (afin d'assurer que $\mathrm{X}_1 \oplus \ldots \oplus \mathrm{X}_n$ soit bien égal à 0 comme prévu). Finalement, il programme l'oracle $\mathcal{H}_4$ de telle sorte que $\mathcal{H}_4(\mathsf{ssid}', \mathrm{X}_i, i) = c'_i$. La simulation n'échoue que si $\mathcal{A}$ a déjà posé la requête à l'oracle pour au moins l'un des X_i, et cette probabilité est bornée par $q_{h_4}/2^{\ell_2}$. À ce stade, chaque liste contient au moins les valeurs suivantes : $(\perp, \mathsf{SK}_i, \perp, \perp, z^*_i, c_i, \perp, \perp, h^{\mathrm{R}}_i, h^{\mathrm{L}}_{i+1}, \mathrm{X}_i, c'_i)$.

Corruptions. Si un joueur P_i se fait corrompre ensuite, le simulateur apprend son mot de passe pw_i et pose des requêtes de déchiffrement pour z^*_i, z^*_{i-1}, et z^*_{i+1} avec la clef pw_i et les étiquettes appropriées, comme dans $\mathbf{G}_4$, ce qui lui permet de récupérer x_i avec probabilité d'échec négligeable. $\mathcal{S}$ peut alors calculer toutes les valeurs dans la liste et donner à $\mathcal{A}$ l'état interne du joueur. Il programme finalement l'oracle aléatoire $\mathcal{H}_2$ de telle sorte que les valeurs $\mathrm{Z}^{\mathrm{R}}_i$ et $\mathrm{Z}^{\mathrm{L}}_{i+1}$ calculées soient cohérentes avec les valeurs h^{R}_i et h^{L}_{i+1} déjà envoyées, c'est-à-dire $\mathcal{H}_2(\mathrm{Z}^{\mathrm{R}}_i) = h^{\mathrm{R}}_i$ et $\mathcal{H}_2(\mathrm{Z}^{\mathrm{L}}_{i+1}) = h^{\mathrm{L}}_{i+1}$. L'événement `AskH` peut faire échouer cette programmation, comme expliqué dans la remarque 1 page 110.

Comme l'on suppose que tous les joueurs ont le même mot de passe (une seule composante connexe), on peut remplir par un raisonnement similaire les listes de chaque joueur restant exactement de la même manière (et avec la même probabilité négligeable d'échec).

Notons que si un joueur reçoit une valeur X_i non *oracle-generated*, alors il va recevoir des valeurs Auth_i et σ_i non *oracle-generated*, qui vont donc être refusées, grâce à $\mathbf{G}_3$. De même, si un joueur reçoit un Auth_i non *oracle-generated*, il va refuser la signature correspondante.

À la troisième étape, le simulateur pose honnêtement les requêtes à $\mathcal{H}_1$, afin de donner aux joueurs des authentifiants corrects, et il calcule (honnêtement) les signatures correspondantes. À la quatrième étape, il pose à la fonctionnalité une requête `Delivery` avec une clef aléatoire pour chaque joueur. Le mot-clef `b` est fixé à `yes` sauf pour les joueurs qui ont reçu des authentifiants non *oracle-generated*. Si un joueur a été corrompu avant, $\mathcal{S}$ reçoit sa clef de session et peut alors reprogrammer l'oracle $\mathcal{H}_0$ en conséquence : en cas d'échec, l'événement `AskH` a nécessairement eu lieu si $\mathcal{A}$ a posé la requête correspondante à $\mathcal{H}_0$, comme dans le jeu précédent.

Corruptions. Notons que dans le cas de corruptions dans les étapes 3 et 4, le simulateur peut facilement calculer l'état interne comme ci-dessus, en ne programmant que $\mathcal{H}_2$.

Cas 2 – Certains joueurs ont des mots de passe différents ($\mathbf{G}_{6\mathtt{b}}$) : Le simulateur donne à chaque joueur une valeur aléatoire c'_i choisie indépendamment des autres. Quand toutes ces valeurs ont été envoyées, il leur donne une valeur X_i choisie aléatoirement et programme $\mathcal{H}_4$ de

sorte que $c'_i = \mathcal{H}_4(\mathsf{ssid}', \mathrm{X}_i, i)$. Cette programmation échoue avec probabilité $q_{h_4}/2^{\ell_2}$. (Notons que dans le jeu réel, il peut aussi arriver que $\sum \mathrm{X}_i = 0$ avec probabilité négligeable .) Il remplit alors les listes d'états internes avec z_i^* et X_i : toutes les autres valeurs sont inconnues du simulateur. Les corruptions sont gérées comme dans le premier cas (quoique cette fois il va y avoir un indice j tel que $h_j^{\mathrm{R}} \neq h_{j+1}^{\mathrm{L}}$ ou $h_j^{\mathrm{L}} \neq h_{j-1}^{\mathrm{R}}$).

À la troisième étape, le simulateur donne des valeurs aléatoires et indépendantes aux joueurs en réponse à leurs requêtes d'authentification, et les signatures correspondantes correctes. Finalement, tous les joueurs reçoivent un message d'erreur à la quatrième étape. Les corruptions sont gérées comme dans le jeu précédent.

Jeu $\mathbf{G}_7$: Nous allons désormais conclure, en montrant comment le simulateur gère les sessions où une corruption a lieu avant la deuxième étape. Nous allons essayer de ne pas utiliser la connaissance des mots de passe des joueurs honnêtes.

Grâce à l'utilisation de la fonctionnalité de répartition de [BCL+05], nous pouvons supposer que tous les joueurs ont reçu les mêmes valeurs pendant la première étape puisque nous avons créé les groupes en fonction des premiers messages reçus (voir page 107). En particulier, aucun message n'a été modifié et $\mathcal{S}$ peut extraire les mots de passe utilisés dans les messages `flow(1a)` et `flow(1b)` non *oracle-generated* (qui correspondent à des joueurs corrompus depuis le début), grâce à z_i^* et la liste Λ_ε. Cette extraction est unique grâce à la restriction sur les collisions incluse dans la simulation (décrite dans $\mathbf{G}_1$ page 108). Bien sûr, cette stratégie peut échouer si $\mathcal{A}$ n'a jamais posé la requête de chiffrement correspondante. Cependant, dans un tel cas, $\mathcal{A}$ n'a aucun contrôle sur le clair (et en particulier aucune idée sur son logarithme discret), et on gère ce cas comme si $\mathcal{A}$ avait un mot de passe incorrect par rapport à son voisin (voir plus bas).

Nous comparons chaque joueur corrompu avec l'ensemble de ses voisins consécutifs non-corrompus qui appartiennent à la même composante connexe (ce qui signifie qu'ils partagent le même mot de passe). En considérant toutes les situations possibles dans lesquelles un joueur P_i non corrompu peut être par rapport à ses voisins (voir la figure 7.3), on distingue alors 6 situations mutuellement exclusives. Dans le premier schéma de cette figure, « $=$ » signifie que le simulateur sait que P_i a le même mot de passe que les deux joueurs corrompus (notés $\mathcal{A}$) les plus proches (à gauche et à droite), le mot de passe étant celui extrait des requêtes de chiffrement, tandis que « $\neq$ » signale des mots de passe différents et « **?** » une relation inconnue entre les mots de passe. Plusieurs cas partagent le même indice pour montrer qu'ils sont en fait symétriques et peuvent être étudiés comme un seul cas.

Il est facile de vérifier que ces cas recouvrent le spectre entier des possibilités. Nous expliquons désormais comment le simulateur peut facilement déterminer à quel cas chaque joueur appartient grâce aux requêtes `GoodPwd` déjà posées. Dès que $\mathcal{S}$ récupère le mot de passe utilisé par un joueur corrompu, il pose une requête `GoodPwd` à ses deux voisins. Pour chaque requête, si elle retourne « non », $\mathcal{S}$ arrête. Sinon, il pose la même requête au joueur suivant. Ce processus est itéré jusqu'à ce que soit une requête retourne « non », soit $\mathcal{S}$ se retrouve dans une situation où il doit poser une requête pour un joueur pour lequel une requête `GoodPwd` a déjà été posée (rappelons qu'il est interdit de poser plus d'une telle requête par joueur). C'est pourquoi nous pouvons avoir un « **?** », même à côté d'un joueur corrompu. Notons que dans le cas où nous ne récupérons pas le mot de passe utilisé par un joueur corrompu, on considère (sans poser de requête `GoodPwd`) qu'il est dans le cas (4b) des deux côtés : on considère que ce joueur ne partage pas le même mot de passe que ses deux voisins.

Nous décrivons maintenant la simulation de la deuxième étape dans ces différents cas. Notons tout d'abord que seul le premier cas mène à une vraie clef partagée. Dans tous les autres cas, au moins un mot de passe est différent des autres, et ainsi les joueurs reçoivent tous un message d'erreur. Mais nous devons cependant être capable de révéler correctement les états internes, en cas de corruption.

Cas 1 ($\mathbf{G}_{7\mathbf{a}}$) : Il existe une suite de joueurs $\mathrm{P}_1, \ldots, \mathrm{P}_k$ (incluant le joueur P_i que l'on cherche à simuler), entre deux joueurs corrompus, partageant tous le même mot de passe (cas 1 de la figure 7.3). S'il y a plus de $n/2$ joueurs corrompus, rappelons que $\mathcal{A}$ doit avoir le droit de

Fig. 7.3 – Les différents cas de composantes connexes

$\mathcal{A} \ldots \leftarrow = \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow = \rightarrow \ldots \mathcal{A}$ (1) $\quad \mathcal{A} \ldots \leftarrow \neq \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow = \rightarrow \ldots \mathcal{A}$ (2)

$\mathcal{A} \ldots \leftarrow = \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow \neq \rightarrow \ldots \mathcal{A}$ (2) $\quad \mathcal{A} \ldots \leftarrow \neq \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow \neq \rightarrow \ldots \mathcal{A}$ (4)

$\mathcal{A} \ldots \leftarrow = \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow ? \rightarrow \ldots \mathcal{A}$ (3) $\quad \mathcal{A} \ldots \leftarrow \neq \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow ? \rightarrow \ldots \mathcal{A}$ (5)

$\mathcal{A} \ldots \leftarrow ? \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow = \rightarrow \ldots \mathcal{A}$ (3)

$\mathcal{A} \ldots \leftarrow ? \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow \neq \rightarrow \ldots \mathcal{A}$ (5)

$\mathcal{A} \ldots \leftarrow ? \rightarrow \ldots \mathrm{P}_i \ldots \leftarrow ? \rightarrow \ldots \mathcal{A}$ (6)

---

(1) $\underset{\rightarrow}{\mathrm{Z}_0^{\mathrm{R}}}$ $\underset{\mathcal{A}}{\mathrm{Z}_1^{\mathrm{L}}=\mathrm{Z}_1^{\mathrm{R}}}$ $\underset{\mathrm{X}_0}{\longleftrightarrow}$ $\underset{\mathrm{P}_1}{\mathrm{Z}_2^{\mathrm{L}}=\mathrm{Z}_2^{\mathrm{R}}}$ $\underset{\mathrm{X}_1}{\longleftrightarrow}$ $\ldots$ $\mathrm{Z}_i^{\mathrm{L}}=\mathrm{Z}_i^{\mathrm{R}}$ $\underset{\mathrm{P}_i}{\longleftrightarrow}$ $\mathrm{Z}_{i+1}^{\mathrm{L}}=\mathrm{Z}_{i+1}^{\mathrm{R}}$ $\underset{\mathrm{X}_i}{\longleftrightarrow}$ $\ldots$ $\mathrm{Z}_k^{\mathrm{L}}=\mathrm{Z}_k^{\mathrm{R}}$ $\underset{\mathrm{P}_k}{\longleftrightarrow}$ $\mathrm{Z}_{k+1}^{\mathrm{L}}=\mathrm{Z}_{k+1}^{\mathrm{R}}$ $\underset{\mathrm{X}_k}{\longleftrightarrow}$ $\underset{\mathcal{A}}{\mathrm{Z}_{k+2}^{\mathrm{L}}}$ $\underset{\mathrm{X}_{k+1}}{\leftarrow}$

---

(2) $\underset{\rightarrow}{\mathrm{Z}_0^{\mathrm{R}}}$ $\underset{\mathcal{A}}{\mathrm{Z}_1^{\mathrm{L}}=\mathrm{Z}_1^{\mathrm{R}}}$ $\underset{\mathrm{X}_0}{\longleftrightarrow}$ $\underset{\mathrm{P}_1}{\mathrm{Z}_2^{\mathrm{L}}=\mathrm{Z}_2^{\mathrm{R}}}$ $\underset{\mathrm{X}_1}{\longleftrightarrow}$ $\ldots$ $\mathrm{Z}_k^{\mathrm{L}}=\mathrm{Z}_k^{\mathrm{R}}$ $\underset{\mathrm{P}_k}{\longleftrightarrow}$ $\mathrm{Z}_{k+1}^{\mathrm{L}}\neq\mathrm{Z}_{k+1}^{\mathrm{R}}$ $\underset{\mathrm{X}_k}{\leftarrow\neq\rightarrow}$ $\underset{\mathrm{P}_{k+1}}{\mathrm{Z}_{k+2}^{\mathrm{L}}}$ $\leftarrow$

---

(3) $\underset{\rightarrow}{\mathrm{Z}_0^{\mathrm{R}}}$ $\underset{\mathcal{A}}{\mathrm{Z}_1^{\mathrm{L}}=\mathrm{Z}_1^{\mathrm{R}}}$ $\underset{\mathrm{X}_0}{\longleftrightarrow}$ $\underset{\mathrm{P}_1}{\mathrm{Z}_2^{\mathrm{L}}=\mathrm{Z}_2^{\mathrm{R}}}$ $\underset{\mathrm{X}_1}{\longleftrightarrow}$ $\ldots$ $\mathrm{Z}_k^{\mathrm{L}}=\mathrm{Z}_k^{\mathrm{R}}$ $\underset{\mathrm{P}_k}{\longleftrightarrow}$ $\mathrm{Z}_{k+1}^{\mathrm{L}}$ $\underset{\mathrm{X}_k}{\leftarrow ? \rightarrow}$ $\mathrm{Z}_{k+1}^{\mathrm{R}}$ $\underset{\mathrm{P}_{k+1}}{}$ $\mathrm{Z}_{k+2}^{\mathrm{L}}$ $\leftarrow$

---

(4) $(4a) \rightarrow \mathcal{A} \longleftrightarrow \ldots \leftarrow \neq \rightarrow \ldots \leftarrow ? \rightarrow \mathrm{P}_i \leftarrow \neq \rightarrow \mathcal{A} \leftarrow$ ou $(4b) \rightarrow \mathcal{A} \leftarrow \neq \rightarrow \mathrm{P}_i \leftarrow \neq \rightarrow \mathcal{A} \leftarrow$

ou $(4c) \rightarrow \mathcal{A} \longleftrightarrow \ldots \leftarrow \neq \rightarrow \ldots \leftarrow ? \rightarrow \mathrm{P}_i \leftarrow ? \rightarrow \ldots \leftarrow \neq \rightarrow \ldots \longleftrightarrow \mathcal{A} \leftarrow$

---

(5) $\underset{\rightarrow}{\mathrm{Z}_0^{\mathrm{R}}}$ $\underset{\mathcal{A}}{\mathrm{Z}_1^{\mathrm{L}}=\mathrm{Z}_1^{\mathrm{R}}}$ $\underset{\mathrm{X}_0}{\longleftrightarrow}$ $\underset{\mathrm{P}_1}{\mathrm{Z}_2^{\mathrm{L}}=\mathrm{Z}_2^{\mathrm{R}}}$ $\underset{\mathrm{X}_1}{\longleftrightarrow}$ $\ldots$ $\mathrm{Z}_{i-1}^{\mathrm{L}}=\mathrm{Z}_{i-1}^{\mathrm{R}}$ $\underset{\mathrm{P}_{i-1}}{\longleftrightarrow}$ $\mathrm{Z}_i^{\mathrm{L}}\neq\mathrm{Z}_i^{\mathrm{R}}$ $\underset{\mathrm{X}_{i-1}}{\leftarrow\neq\rightarrow}$ $\underset{\mathrm{P}_i}{?}$ $\leftarrow$

---

(6) $\rightarrow \mathcal{A} \ldots \leftarrow ? \rightarrow \mathrm{P}_i \leftarrow ? \rightarrow \ldots \mathcal{A} \leftarrow$

choisir la clef de session. Comme $\mathcal{S}$ connaît le mot de passe de cette composante connexe, il peut tout calculer correctement. En outre, *avant* d'envoyer un quelconque X_i, il récupère les h_i des joueurs corrompus grâce aux c'_i et à la liste des oracles aléatoires, et il pose la requête à l'oracle $\mathcal{H}_0$ pour obtenir la clef de session. Il pose alors une requête `SamePwd` à la fonctionnalité, qui fixe la clef soit à cette valeur soit à `error`. Ainsi, $\mathcal{S}$ a fixé la clef à la valeur que $\mathcal{A}$ a choisie, ceci grâce aux valeurs envoyées par les joueurs corrompus. Il sera capable de renvoyer à $\mathcal{A}$ cette clef particulière quand ce dernier posera la requête correspondante à $\mathcal{H}_0$ – cette requête étant imprédictible à $\mathcal{A}$ jusqu'à ce qu'il reçoive les X_i.

S'il y a au maximum $\lfloor n/2 - 1/2 \rfloor$ joueurs corrompus, il existe au moins une composante de joueurs honnêtes $\mathrm{P}_1, \ldots, \mathrm{P}_k$ pour laquelle $k \geqslant 2$. Ceci sera le point-clef de la généralisation présentée dans la section 7.4.1 page 115. Dans les composantes où $k = 1$, comme le simulateur connaît le mot de passe de P_1, il peut le simuler parfaitement. Dans les composantes où $k \geqslant 2$, le simulateur donne aux joueurs des valeurs c'_j aléatoires choisies indépendamment. Une fois l'étape $(2a)$ terminée, il calcule honnêtement les valeurs h_1^{L} et h_k^{R} mais choisit au hasard toutes les autres, avec la contrainte que $h_i^{\mathrm{L}} = h_i^{\mathrm{R}}$ partout de manière itérative (voir la remarque 1 page 110). Comme avant, $\mathcal{S}$ programme aussi l'oracle $\mathcal{H}_4$ afin que $\mathcal{H}_4(\mathsf{ssid}', \mathrm{X}_j, j) = c'_j$, avec probabilité d'échec négligeable . Avant d'envoyer un quelconque X_i, $\mathcal{S}$ pose une requête `SamePwd` et une requête `Delivery` pour un joueur corrompu, avec une clef aléatoire et le mot-clef `yes`, obtenant ainsi soit la valeur de la clef soit un message d'erreur. Si une clef a été fixée, $\mathcal{S}$ récupère les h_i des joueurs corrompus et programme $\mathcal{H}_0$ sur cette valeur. Cette programmation n'échoue que si l'événement `AskH` apparaît. Dans cette composante, tous les X_j sont donc imprédictibles pour l'adversaire. À ce stade, chaque liste contient au moins les valeurs suivantes : $(\bot, \bot, \bot, z_i^*, c_i, \bot, \bot, h_i^{\mathrm{R}}, h_{i+1}^{\mathrm{L}}, \mathrm{X}_i, c'_i)$. En cas de corruptions ultérieures, on procède comme dans le jeu $\mathbf{G}_{6a}$.

Cas 2 et 3 ($\mathbf{G_{7b}}$) : Il existe une suite de joueurs $\mathrm{P}_1, \ldots, \mathrm{P}_k$ (incluant le joueur P_i que l'on simule), à côté d'un joueur corrompu, partageant tous le même mot de passe, mais pas nécessairement avec P_{k+1} (la suite de requêtes `GoodPwd` réussies a pris fin avec lui ou bien une requête `GoodPwd` a déjà échoué depuis le côté droit de P_{k+1}). Ce sont les cas 2 et 3 de la figure 7.3.

À nouveau, la simulation est parfaite pour les k premiers joueurs, puisque le simulateur connaît le mot de passe partagé entre $\mathrm{P}_1, \ldots, \mathrm{P}_k$. Ensuite, $\mathcal{S}$ envoie un X_{k+1} aléatoire au nom de P_{k+1}. En cas de corruptions ultérieures, remarquons qu'à la première étape, les deux joueurs P_k et P_{k+1} (non corrompus) ont envoyé les valeurs z_k^* et z_{k+1}^* sans avoir posé de requête de chiffrement. Cela signifie qu'il existe une probabilité négligeable que $\mathcal{A}$ ait posé la requête de chiffrement correspondante avec les étiquettes (ssid, k) et $(\mathsf{ssid}, k+1)$, respectivement. En fait, poser la requête $\mathcal{H}_2(\mathrm{Z}_{k+1})$ dans ce contexte revient exactement à l'événement `AskH`, ce qui implique que $\mathcal{A}$ ne peut pas distinguer X_{k+1} d'une valeur exacte. La programmation ne va donc échouer qu'avec une probabilité négligeable, comme dans $\mathbf{G}_5$.

Cas 4, 5 et 6 ($\mathbf{G_{7c}}$) : P_i ne partage le mot de passe d'aucun de ses deux voisins corrompus les plus proches (cas 4 de la figure 7.3), ou de l'un des deux (cas 5), ou on ne sait rien (cas 6). Dans ce cas, X_i est aléatoire, mais on ne connaît rien d'autre excepté z_i^* et X_i pour P_i. Notons que P_i n'est pas corrompu et qu'il a envoyé z_i^* sans poser de requête de chiffrement. Premier cas, si le voisin de P_i (disons P_{i+1}) est un joueur corrompu, il n'a pas posé de requête de chiffrement menant à z_{i+1}^* avec le mot de passe pw_i et l'étiquette (ssid, i) (simplement parce qu'il a posé une requête à l'oracle de déchiffrement avec pw et l'étiquette $(\mathsf{ssid}, i+1)$, et que nous supposons que $\mathsf{pw}_i \neq \mathsf{pw}$). Deuxième cas, si P_{i+1} est un joueur honnête, alors il a envoyé z_{i+1}^* sans requête à l'oracle de chiffrement. Ainsi, dans tous les cas, $\mathcal{A}$ ne peut se rendre compte que X_i n'était pas choisi correctement que si l'événement `AskH` intervient. Les corruptions ultérieures sont gérées comme dans $\mathbf{G}_5$.

Dernières étapes : Elles sont gérées exactement comme dans le jeu précédent, selon la réponse à la requête `SamePwd`. Seul le premier cas doit être géré avec soin : $\mathcal{S}$ ne donne pas une clef aléatoire, mais la clef obtenue par la première requête `Delivery` posée ci-dessus.

Ainsi, en ignorant l'événement `AskH`, dont la probabilité (négligeable) est calculée dans la section 7.3.3, $\mathbf{G}_7$ et $\mathbf{G}_6$ sont indistinguables.

Jeu $\mathbf{G}_8$: Ce jeu est pratiquement le même que le précédent sauf que nous formalisons le comportement du simulateur en introduisant les requêtes à la fonctionnalité, c'est-à-dire en remplaçant les requêtes `GoodPwd`, `SamePwd` et `Delivery` par leurs équivalents idéaux. Informellement, $\mathcal{S}$ se comporte non pas en fonction des messages envoyés, mais en fonction des messages reçus (éventuellement modifiés par l'adversaire). À la première étape, $\mathcal{S}$ envoie une valeur z_i^* aléatoire au nom de chaque joueur non corrompu. Pour la deuxième étape, voir les jeux $\mathbf{G}_6$ et $\mathbf{G}_7$. Dans la troisième étape, $\mathcal{S}$ pose des requêtes de livraison de clef avec une clef aléatoire, ou la clef obtenue plus tôt, voir $\mathbf{G}_{7a}$. À la quatrième étape, $\mathcal{S}$ fixe `b` à `no` pour les joueurs qui ont reçu un message non *oracle-generated* à l'étape précédente et à `yes` pour les autres. Si une session abandonne ou termine, $\mathcal{S}$ le rapporte à $\mathcal{A}$. Montrons désormais que $\mathbf{G}_8$ est indistinguable du jeu idéal, en disant que les joueurs ont des sessions partenaires s'ils partagent le même ssid' (ce qui implique qu'ils partagent les mêmes VK_i et z_i^* grâce à l'utilisation de la fonctionnalité de répartition).

Il est clair que les joueurs partageant le même mot de passe vont obtenir une clef aléatoire, à la fois dans $\mathbf{G}_8$ (depuis $\mathbf{G}_5$) et le jeu idéal IWE (sauf pour les joueurs qui reçoivent des messages non *oracle-generated*, ce qui est modélisé par le bit `b`). Cette clef ne va pas être choisie par l'adversaire sauf s'il y a suffisamment de joueurs corrompus (puisqu'il y aura toujours deux joueurs honnêtes côte à côte, voir $\mathbf{G}_7$). Finalement, des joueurs qui ne partagent pas le même mot de passe vont recevoir une erreur. Maintenant, nous devons montrer que deux joueurs vont recevoir la même clef dans $\mathbf{G}_8$ si et seulement si c'est le cas dans IWE.

C'est clairement le cas pour des joueurs qui ont des sessions partenaires (avec ou sans le même mot de passe). Cela vient des jeux $\mathbf{G}_5$ ou $\mathbf{G}_7$ dans le monde réel, et des requêtes

`NewSession` mentionnant les mêmes groupes de joueurs dans le monde idéal. Finalement, considérons le cas de joueurs sans sessions partenaires. Il est clair que dans $\mathbf{G}_8$ les clefs de session de ces joueurs vont être indépendantes puisqu'elles ne sont fixées dans aucun des jeux. Dans IWE, la seule manière pour qu'ils reçoivent des clefs identiques est que la fonctionnalité reçoive deux requêtes `NewSession` avec le même ssid' et un groupe dans lequel tous les joueurs partagent le même mot de passe, ce qui n'est pas le cas puisqu'ils n'ont pas de sessions partenaires.

7.3.3 Probabilité de AskH

Nous montrons désormais que l'événement `AskH` a lieu avec probabilité négligeable, à l'aide d'une réduction à un problème CDH, étant donné une instance $\mathsf{CDH}(\mathrm{U}, \mathrm{V})$. Au début du jeu, globalement pour toutes les sessions, $\mathcal{S}$ choisit deux requêtes de déchiffrement. Avec probabilité $1/({q_\mathcal{D}}^2)$, $q_\mathcal{D}$ étant le nombre de requêtes de déchiffrement, ces requêtes vont correspondre à celles réellement posées par l'adversaire. On utilise U et V pour simuler ces deux requêtes. Plus précisément, pour la première requête de déchiffrement z_i^*, si $(*, (\mathsf{ssid}, i), *, *, *, z_i^*) \in \Lambda_\varepsilon$, il abandonne. Sinon, il ajoute $(\mathsf{pw}, (\mathsf{ssid}, i), \mathrm{U}, \perp, \mathcal{D}, z_i^*)$ à la liste. Pour la seconde, z_j^*, si $(*, (\mathsf{ssid}, j), *, *, *, z_j^*) \in \Lambda_\varepsilon$, il abandonne. Sinon, il ajoute $(\mathsf{pw}, (\mathsf{ssid}, j), \mathrm{V}, \perp, \mathcal{D}, z_j^*)$ à la liste. Soit $\texttt{AskH}'$ l'événement dans lequel $\mathcal{A}$ a posé la requête

$$\mathrm{CDH}(\mathcal{D}_{\mathsf{pw}}^{(\mathsf{ssid},i)}(z_i^*), \mathcal{D}_{\mathsf{pw}}^{(\mathsf{ssid},j)}(z_j^*))$$

à l'oracle $\mathcal{H}_2$. Ceci implique que $|\Pr[\texttt{AskH}']| = {q_\mathcal{D}}^2\, q_{\mathcal{H}_2} |\Pr[\texttt{AskH}]|$, $q_{\mathcal{H}_2}$ étant le nombre de requêtes à l'oracle aléatoire $\mathcal{H}_2$. Comme on sait que $\mathrm{CDH}(z_i, z_j) = \mathrm{CDH}(\mathrm{U}, \mathrm{V})$, notons finalement que si $\mathcal{A}$ fait intervenir l'événement `AskH`, il a résolu le problème CDH, ce qui n'est possible qu'avec probabilité négligeable .

7.4 Généralisation

7.4.1 $(n-2, n)$-*Contributory*

On vient de montrer que s'il existe deux joueurs voisins non corrompus quelque part dans le cycle (voir $\mathbf{G}_{7a}$), alors la clef est complètement imprédictible pour l'adversaire, ce qui est exactement la définition de la $(n/2, n)$-*contributory*. On explique maintenant comment étendre ce résultat à la $(n-2, n)$-*contributory*.

On peut facilement remarquer que l'on peut construire $p = \binom{n}{2}$ cycles tels que tous les joueurs soient voisins de chaque autre joueur dans au moins un cycle. Pour cela, on considère toutes les paires de joueurs possibles et on complète le cycle de manière arbitraire. Mais en fait, il est même possible d'obtenir un nombre linéaire de cycles (voir le lemme 1).

On montre maintenant comment une telle construction peut permettre au protocole d'atteindre la $(n-2, n)$-*contributory*. Considérons un protocole consistant en p exécutions parallèles du protocole décrit plus haut, dans lequel le ssid inclut le numéro de l'exécution. Pour éviter des attaques de malléabilité dans lesquelles l'adversaire rejouerait des messages dans des ordres différents dans la même session, on demande aussi aux joueurs de ne pas envoyer des valeurs séparées de c_i et c_i' pour chaque exécution, mais d'envoyer des valeurs uniques de c_i et c_i' pour toutes les exécutions. De même, une clef unique de signature et vérification suffit : chaque joueur signe un authentifiant qui met en jeu les éléments nécessaires aux clefs de toutes les exécutions parallèles. À la fin de ces dernières, chaque joueur a obtenu p clefs. Si l'une d'entre elles est un message d'erreur, il reçoit un message d'erreur. Sinon, la clef de session finale est définie comme le haché de toutes ces clefs.

Supposons que tous les joueurs partagent le même mot de passe. Alors, s'il y a au moins deux joueurs non-corrompus, on considère l'exécution dans laquelle ils sont voisins. On a montré dans $\mathbf{G}_7$ que, dans ce cas, la clef de cette exécution est imprédictible pour un adversaire. Ceci mène à une clef finale imprédictible pour l'adversaire, et donc à la $(n-2, n)$-*contributory*.

Lemme 1. *On peut construire* $\lceil n/2 \rceil$ *cycles des* n *joueurs tels que pour chaque paire* $(\mathrm{P}_i, \mathrm{P}_j)$ *de joueurs,* P_i *et* P_j *sont voisins dans au moins un cycle.*

Démonstration. On peut facilement vérifier le résultat pour $n = 2$ et $n = 3$ et on le démontre par récurrence. On considère un entier n impair, et on suppose que le résultat est vrai pour $n - 1$ joueurs. On appelle $\mathcal{A}_{n-1}$ l'algorithme pour $n - 1$ joueurs et on construit à partir de là $\mathcal{A}_n$ et $\mathcal{A}_{n+1}$, ce qui prouvera le résultat pour n et $n + 1$.

On commence par appliquer $\mathcal{A}_{n-1}$ aux joueurs $\mathrm{P}_1, \ldots, \mathrm{P}_{n-1}$ et on insère dans chaque cycle le joueur P_n entre deux joueurs qu'il n'a jamais rencontrés. Cette manipulation est possible jusqu'à la $i_0 = \lfloor n/2 \rfloor$-ième étape.

En effet, appelons « o » un joueur qui a déjà été un voisin de P_n et « n » un joueur qui ne l'a jamais été. Il faut montrer qu'il existe toujours deux voisins de type « n » jusqu'au rang i_0. Le cas le pire est « (on)(on)… (on)nn… nn », où tous les joueurs vus sont entre deux joueurs non vus, ce qui empêche P_n de se placer à cet endroit à l'étape suivante.

Soit i un cycle dans lequel on peut trouver une place pour P_n entre deux joueurs de type « n ». Comme P_n voit deux nouveaux joueurs différents à chaque étape, il a déjà vu $2(i - 1)$ joueurs jusqu'à l'étape i. Le nombre de places libres (entre deux « n » est alors égal à $(n - 1) - [2(i - 1)] = n + 1 - 2i$. Pour être capable d'insérer P_n, on a besoin d'au moins une place libre, donc $n + 1 - 2i \geqslant 1$ et $i \leqslant n/2$.

On effectue cette manipulation des étapes 1 à $(n - 1)/2$. P_n a alors vu $n - 1$ joueurs comme voisins, c'est-à-dire tout le monde. Et l'algorithme $\mathcal{A}_{n-1}$ est aussi fini, puisqu'il consiste en $(n-1)/2$ étapes. Il reste à traiter le cas des seuls joueurs qui ne se sont pas encore rencontrés, qui sont exactement ceux qui se seraient rencontrés si l'on n'avait pas inséré P_n entre eux. Mais les paires séparées par P_n sont toutes distinctes par construction, donc on peut les faire se rencontrer dans un cycle unique (on les juxtapose simplement). Finalement, on a donc construit un algorithme pour n joueurs en $(n - 1)/2 + 1 = \lceil n/2 \rceil$ étapes.

Montrons désormais que le résultat est aussi vrai pour $n + 1$ sous les mêmes hypothèses. Comme précédemment, on utilise $\mathcal{A}_n$ et on insère P_{n+1} entre deux joueurs non vus jusqu'à ce que cela devienne impossible. De même, le dernier rang auquel c'est possible est $(n + 1)/2$, mais on choisit d'arrêter à l'étape $(n - 1)/2$. À ce stade, P_{n+1} a vu $n - 1$ des joueurs comme voisins, c'est-à-dire tous sauf un. Mais la dernière étape de $\mathcal{A}_n$ consiste en la juxtaposition de paires indépendantes. Ainsi, on peut insérer P_{n+1} à côté du dernier voisin, sans rien changer à la correction de $\mathcal{A}_n$. Finalement, si n est impair, on a montré comment construire $\mathcal{A}_n$ et $\mathcal{A}_{n+1}$ étant donné $\mathcal{A}_{n-1}$. □

7.4.2 Moyens d'authentification

Lorsque nous avons défini la fonctionnalité idéale pour les GPAKE dans la section 7.1, on a demandé que tous les joueurs aient le même mot de passe pour pouvoir établir une clef secrète commune. Cependant, il y a d'autres manières de définir la fonctionnalité idéale, qui pourraient être plus adaptées pour des applications particulières. Par exemple, considérons le cas dans lequel chaque paire d'utilisateurs a un mot de passe différent associé à cette paire. Dans ce scénario, on pourrait envisager une définition qui permette à des utilisateurs d'établir un secret commun si chaque utilisateur dans le groupe partage un mot de passe avec chacun de ses voisins (en supposant un ordre particulier sur les joueurs). Nous avons opté pour la première formulation ici pour des raisons de simplicité.

En outre, comme dans [BCL+05], nos résultats peuvent aussi être étendus au cas de réseaux partiellement authentifiés, dans lesquels certains joueurs en jeu dans le protocole peuvent avoir des liens authentifiés à leur disposition. Comme noté par Barak *et al.* [BCL+05], cela semble être un scénario plus réaliste que le scénario standard qui suppose que soit toutes les paires de joueurs sont connectées par des canaux authentifiés, soit aucune d'entre elles ne l'est avec une autre.

Troisième partie

Protocoles d'échanges de clefs dans le cadre UC et le modèle standard

Chapitre 8

Smooth projective hash functions

Nous avons décrit les *smooth projective hash functions* (SPHF) dans la section 1.2.9 page 25. Ici, nous allons décrire comment construire des SPHF pour des langages plus complexes, qui peuvent être décrits en termes de disjonctions et conjonctions de langages plus simples pour lesquels ces fonctions existent. Nous nous basons pour cela sur la version de Gennaro et Lindell [GL03]. Par exemple, si H_m représente une famille de SPHF pour le langage $\{(c)\}$, où c est un chiffré de m sous une clef publique donnée, nous pouvons construire une famille de SPHF pour le langage $\{(c)\}$, où c est un chiffré de 0 *ou* de 1, en combinant H_0 et H_1.

L'un des avantages de construire des SPHF pour des langages plus complexes est de permettre de simplifier la création de primitives auxquelles elles sont associées. Nous illustrons donc ensuite comment leur utilisation peut être efficacement associée à des schémas de mise en gage extractibles et ainsi éviter l'utilisation de preuves *zero-knowledge*. Dans la plupart des protocoles dans lesquels des mises en gage extractibles sont utilisées, la capacité d'extraire le message engagé dépend en général de la génération de la mise en gage (proprement ou non). Pour atteindre ce but et obliger la génération correcte de cette mise en gage, on ajoute souvent des mécanismes additionnels, tels que des preuves *zero-knowledge*. C'est le cas, par exemple, de plusieurs protocoles où une phase spécifique de certification de clef publique est nécessaire, tels que la plupart des protocoles cryptographiques avec des groupes dynamiques (multisignatures [Bol03, LOS+06], signatures de groupe [DP06], etc). Un tel cadre est souvent

appelé *registered public-key setting*, dans lequel une preuve de connaissance de la clef secrète est requise avant toute certification.

Pour être capable de construire des schémas de mise en gage extractibles plus efficaces et éviter l'utilisation de preuves *zero-knowledge* concurrentes éventuellement coûteuses, un deuxième point dans ce chapitre est de généraliser le concept de mises en gage extractibles afin que l'extractibilité puisse échouer si la mise en gage n'est pas proprement générée. Plus précisément, nous introduisons une nouvelle notion de mises en gage L-*extractibles* dans lesquelles l'extraction est seulement garantie si la valeur engagée appartient au langage L et peut échouer sinon. L'intuition principale derrière cette généralisation est que, lorsqu'on utilise ces mises en gage conjointement avec une SPHF pour le langage L, les cas dans lesquels l'extraction peut échouer ne vont pas être importants puisque la sortie de la SPHF va alors être statistiquement indistinguable d'une valeur aléatoire.

Nous expliquons finalement comment appliquer ces résultats pour donner des solutions plus efficaces à deux problèmes cryptographiques connus (nous donnons un peu plus de détail dans les deux paragraphes suivants) : une certification de clef publique qui garantit la connaissance de la clef privée par l'utilisateur sans oracle aléatoire ou preuves *zero-knowledge* ; et la sécurité adaptative pour des protocoles d'échange de clefs authentifiés basés sur des mots de passe dans le cadre de composabilité universelle avec effacements et sans hypothèses idéales. Tous les résultats de ce chapitre et du suivant ont été publiés à la conférence Crypto 2009 [ACP09].

Registered Public-Key Setting. Il existe beaucoup de protocoles cryptographiques dans lesquels le simulateur décrit dans la preuve de sécurité doit pouvoir extraire la clef privée des utilisateurs autorisés afin de prouver la sécurité même quand les utilisateurs peuvent se joindre de manière dynamique au système. Ceci permet d'éviter les *rogue-attacks* [Bol03] et s'appelle le *registered public-key setting*. Ceci devrait de toute façon être la bonne manière de procéder pour une autorité de certification : elle certifie la clef publique d'un utilisateur si et seulement si ce dernier procure une preuve de connaissance de la clef privée associée. Cependant, afin de permettre la concurrence, des preuves *zero-knowledge* complexes sont requises, ce qui rend le processus de certification sûr uniquement dans le modèle de l'oracle aléatoire [BR93], ou inefficace dans le modèle standard.

Nous montrons ici comment les SPHF avec des mises en gage conditionnellement extractibles peuvent aider à résoudre ce problème efficacement, dans le modèle standard, en établissant un canal sécurisé entre les protagonistes, avec des clefs qui sont soit les mêmes pour les deux si la mise en gage a été construite correctement, soit parfaitement indépendantes sinon.

Schémas PAKE adaptativement sûrs. Nous étudions ensuite des schémas d'échange de clefs plus évolués en utilisant une approche différente de celle de Barak *et al.* [BCL+05]. Au lieu d'utiliser des techniques générales de MPC, nous étendons la méthodologie de Gennaro et Lindell [GL03] décrite page 45 pour gérer les corruptions adaptatives en utilisant un schéma de mise en gage non-malléable, conditionnellement extractible et équivocable avec une famille de SPHF associée (les définitions de ces primitives se trouvent dans le chapitre 1 pages 25 et 25). Ce nouveau schéma est adaptativement sûr dans le modèle de la CRS dans le cadre UC avec effacements sous des hypothèses de complexités standards.

Nous rappelons dans la section 8.2 les primitives basiques nécessaires. Nous décrivons ensuite dans la section 8.3 la construction des familles SPHF pour des conjonctions et disjonctions de langages. Nous combinons alors cela avec des mises en gage conditionnellement extractibles dans la section 8.4. Nous nous concentrons sur des mises en gage basées sur le schéma ElGamal, puisque c'est suffisant pour construire des protocoles de certification de clef publique plus efficaces. Nous ajoutons ensuite dans la section 8.5 l'équivocabilité à cette mise en gage, en empruntant des techniques au schéma de Canetti et Fischlin [CF01]. Finalement, nous ajoutons la propriété de non-malléabilité, grâce au schéma de chiffrement Cramer-Shoup, et la mise en gage résultante peut alors être utilisée pour construire un PAKE adaptativement sûr dans le cadre UC, basé sur la définition de Gennaro et Lindell [GL03], dans le chapitre 9.

8.1 Schémas de chiffrement étiquetés

La primitive de chiffrement étiqueté est décrit dans la section 1.2.4 page 21. Nous utilisons ici les mêmes notations. L'un des avantages de l'utiliser, que l'on exploite ici, est que l'on peut aisément combiner plusieurs tels schémas IND-CCA à l'aide d'un schéma de signature *one-time* et résistant aux contrefaçons de telle sorte que le schéma résultant reste IND-CCA [DK05].

8.2 Mises en gage

Dans toute la suite, on se base sur des mises en gage (*commitments*) de type Pedersen (telles que présentées dans la section 1.2.8), et des certifications de clefs publiques à la Schnorr, c'est-à-dire qu'on travaille dans le cadre du logarithme discret. Afin d'obtenir des mises en gage extractibles, on utilise donc des schémas de chiffrement de la même famille : le chiffrement ElGamal [ElG85] et la version étiquetée du chiffrement de Cramer-Shoup [CS98] (pour obtenir la non-malléabilité).

Dans toute la suite, on utilise des schémas de chiffrement afin de construire des mises en gage, ce qui implique immédiatement les propriétés de *hiding*, *binding* et d'extraction, comme on l'a vu ci-dessus. En revanche, si l'on utilise les versions additives des schémas de chiffrement ElGamal ou Cramer-Shoup, l'extractibilité (ou le déchiffrement) n'est possible que si les valeurs mises en gage (ou les messages clairs) sont suffisamment petites. Ceci justifie notre notion de *mises en gage* L-*extractibles* (voir la section 8.4), ce qui signifie qu'une mise en gage est extractible si la valeur engagée appartient au langage L, et pas forcément sinon. Plus précisément, nous séparons la valeur engagée en petits morceaux (chacun dans le langage L), mais il faut alors s'assurer qu'ils appartiennent effectivement à ce langage pour garantir l'extractibilité du tout. On introduit pour cela les *smooth projective hash functions* afin d'autoriser les communications uniquement dans le cas où les mises en gage sont valides.

8.3 *Smooth projective hash functions* sur les conjonctions et disjonctions de langages

Nous avons vu page 25 la description informelle de cette primitive. Nous donnons ici leur définition formelle, et présentons de nouvelles constructions de SPHF pour gérer des langages plus complexes, tels que des disjonctions ou des conjonctions de n'importe quels langages. Les constructions sont présentées pour deux langages mais peuvent aisément être étendues à un nombre polynomial de langages. Les pertes possible d'information sont discutées à la fin de cette section. Les propriétés de correction, *smoothness* et pseudo-aléa sont facilement vérifiées par ces nouveaux SHS (les preuves formelles peuvent être trouvées dans la section 8.3.6).

8.3.1 Notation des langages

Puisque l'on veut utiliser ici des langages plus généraux, on a besoin de notations plus détaillées. Soit **LPKE** un schéma de chiffrement étiqueté avec la clef publique pk (*labeled public-key encryption*). Soit X l'espace des chiffrés. Voici trois exemples utiles de langages L dans X :

- les chiffrés valides c de m sous pk, $\mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,m)} = \{c | \exists r\ c = \mathcal{E}^{\ell}_{\mathsf{pk}}(m;r)\}$;
- les chiffrés valides c de m_1 ou m_2 sous pk (c'est-à-dire une disjonction de deux versions des langages précédents), $\mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,m_1 \vee m_2)} = \mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,m_1)} \cup \mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,m_2)}$;
- les chiffrés valides c sous pk, $\mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,*)} = \{c | \exists m\ \exists r\ c = \mathcal{E}^{\ell}_{\mathsf{pk}}(m;r)\}$.

Si le schéma de chiffrement est $\mathsf{IND-CPA}$, les deux premiers sont des *hard partitioned subsets* de X. Le dernier peut aussi en être un dans certains cas : pour le chiffrement de Cramer-Shoup, $\mathrm{L} \subsetneq \mathrm{X} = \mathrm{G}^4$ et, afin de distinguer un chiffré valide d'un chiffré invalide, on

doit casser le problème DDH. Cependant, pour le schéma de chiffrement ElGamal, tous les chiffrés sont valides, c'est-à-dire que $\mathrm{L} = \mathrm{X} = \mathrm{G}^2$.

Des langages plus complexes peuvent être définis, avec des disjonctions comme ci-dessus, ou des conjonctions : par exemple, les paires de chiffrés (a, b) tels que $a \in \mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,0\vee1)}$ et $b \in \mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,2\vee3)}$. Cet ensemble peut être obtenu par $(\mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,0\vee1))} \times \mathrm{X}) \cap (\mathrm{X} \times \mathrm{L}_{(\mathbf{LPKE},\mathsf{pk}),(\ell,2\vee3)})$.

De même, on peut définir des langages plus généraux basés sur d'autres primitives telles que des schémas de mise en gage. La définition serait similaire à celle ci-dessus, avec pk jouant le rôle des paramètres communs, $\mathcal{E}_{\mathsf{pk}}$ jouant le rôle de l'algorithme de mise en gage, (m, ℓ) jouant le rôle du message en entrée, et c jouant le rôle de la mise en gage.

Plus généralement, dans la suite, on utilise la notation générique $\mathrm{L}_{(\mathbf{Sch},\rho),\mathsf{aux}}$ où aux symbolise tous les paramètres utiles pour caractériser le langage (tels que l'étiquette utilisée, ou un message clair), ρ les paramètres publics tels que la clef publique pk, et **Sch** la primitive utilisée pour définir le langage, telle que le schéma de chiffrement **LPKE** ou un schéma de mise en gage **Com**. Quand il n'y a pas d'ambiguïté, la primitive associée **Sch** sera omise.

8.3.2 Définitions formelles des SPHF

Comme défini dans [GL03], une famille de SPHF, pour un langage $\mathrm{L}_{\mathsf{pk},\mathsf{aux}} \subset \mathrm{X}$, sur l'ensemble G, basée sur un schéma de chiffrement étiqueté à clef publique pk ou sur un schéma de mise en gage avec pk comme paramètre public consiste en quatre algorithmes, et est notée de la façon suivante : $\mathbf{HASH}(\mathsf{pk}) = (\mathsf{HashKG}, \mathsf{ProjKG}, \mathsf{Hash}, \mathsf{ProjHash})$. Notons que X est l'espace des chiffrés ou des mises en gage.

L'algorithme probabiliste de génération de clefs produit des clefs de hachage via $\mathsf{hk} \xleftarrow{\$} \mathsf{HashKG}(\mathsf{pk}, \mathsf{aux})$. L'algorithme de génération de clefs de projection produit les clefs projetées via $\mathsf{hp} = \mathsf{ProjKG}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c)$, dans lequel c est soit un chiffré soit une mise en gage dans X suivant les cas. L'algorithme de hachage Hash calcule, sur $c \in \mathrm{X}$, la valeur hachée $g = \mathsf{Hash}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c) \in \mathrm{G}$, en utilisant la clef de hachage hk. Finalement, l'algorithme de hachage projeté $\mathsf{ProjHash}$ calcule, sur $c \in \mathrm{X}$, la valeur hachée $g = \mathsf{ProjHash}(\mathsf{hp}; \mathsf{pk}, \mathsf{aux}, c; w) \in \mathrm{G}$, en utilisant la clef projetée hp et un témoin w du fait que $c \in \mathrm{L}_{\mathsf{pk},\mathsf{aux}}$.

Décrivons désormais formellement les trois propriétés d'un *smooth hash system* (SHS).

CORRECTNESS (CORRECTION). Soient $c \in \mathrm{L}_{\mathsf{pk},\mathsf{aux}}$ et w un témoin de cette appartenance. Alors, pour toutes les clefs de hachage $\mathsf{hk} \xleftarrow{\$} \mathsf{HashKG}(\mathsf{pk}, \mathsf{aux})$ et les clefs projetées correspondantes $\mathsf{hp} = \mathsf{ProjKG}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c)$, $\mathsf{Hash}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c) = \mathsf{ProjHash}(\mathsf{hp}; \mathsf{pk}, \mathsf{aux}, c; w)$.

SMOOTHNESS. Pour tout c n'appartenant *pas* à $\mathrm{L}_{\mathsf{pk},\mathsf{aux}}$, la valeur hachée $g = \mathsf{Hash}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c)$ est statistiquement proche de l'uniforme et indépendante des valeurs hp, pk, aux et c. Pour une clef de hachage hk choisie uniformément, les deux distributions suivantes sont statistiquement indistinguables :

$$\{\mathsf{pk}, \mathsf{aux}, c, \mathsf{hp} = \mathsf{ProjKG}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c), \quad g = \mathsf{Hash}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c)\}$$
$$\{\mathsf{pk}, \mathsf{aux}, c, \mathsf{hp} = \mathsf{ProjKG}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c), \quad g \xleftarrow{\$} \mathrm{G}\}$$

PSEUDORANDOMNESS (CARACTÈRE PSEUDO-ALÉATOIRE). Si $c \in \mathrm{L}_{\mathsf{pk},\mathsf{aux}}$, alors sans un témoin w de cette appartenance, la valeur hachée $g = \mathsf{Hash}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c)$ est calculatoirement indistinguable d'une valeur aléatoire. Pour une clef de hachage hk choisie uniformément, les deux distributions suivantes sont calculatoirement indistinguables :

$$\{\mathsf{pk}, \mathsf{aux}, c, \mathsf{hp} = \mathsf{ProjKG}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c), \quad g = \mathsf{Hash}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c)\}$$
$$\{\mathsf{pk}, \mathsf{aux}, c, \mathsf{hp} = \mathsf{ProjKG}(\mathsf{hk}; \mathsf{pk}, \mathsf{aux}, c), \quad g \xleftarrow{\$} \mathrm{G}\}$$

Soit $\mathrm{L}_{\mathsf{pk},\mathsf{aux}}$ tel qu'il soit difficile de distinguer un élément aléatoire appartenant à $\mathrm{L}_{\mathsf{pk},\mathsf{aux}}$ d'un élément aléatoire n'appartenant pas à $\mathrm{L}_{\mathsf{pk},\mathsf{aux}}$. Gennaro et Lindell ont formalisé le ca-

ractère pseudo-aléatoire en montrant dans [GL03] que les deux expériences suivantes sont indistinguables (en notant D le distingueur) :

`Expt-Hash`(D) : soit D un adversaire ayant accès à deux oracles : Ω et `Hash`. Le premier oracle reçoit une entrée vide et retourne $x \in \mathrm{L}_{\mathsf{pk},\mathsf{aux}}$ choisi suivant la distribution de $\mathrm{L}_{\mathsf{pk},\mathsf{aux}}$. L'oracle Hash reçoit x en entrée. Si x n'a pas été obtenu au préalable par Ω, il ne renvoie rien. Sinon, il choisit une clef hk et renvoie la paire $(\mathsf{ProjKG}(\mathsf{hk};\mathsf{pk},\mathsf{aux},x),\mathsf{Hash}(\mathsf{hk};\mathsf{pk},\mathsf{aux},x))$. La sortie de l'expérience est celle de D.

`Expt-Unif`(D) : cette expérience est définie exactement comme la précédente sauf que l'oracle `Hash` est remplacé par l'oracle `Unif` suivant : lorsqu'il reçoit x, si x n'a pas été précédemment obtenu par Ω, il ne renvoie rien. Sinon, il choisit une clef hk et un élément aléatoire g et renvoie la paire $(\mathsf{ProjKG}(\mathsf{hk};\mathsf{pk},\mathsf{aux},x),g)$. La sortie de l'expérience est celle de D.

Dans le cas où le langage $\mathrm{L}_{\mathsf{pk},\mathsf{aux}}$ est associé à un schéma de chiffrement étiqueté, nous appelons `Enc` l'oracle Ω. Dans le cas où ce langage est associé à un schéma de mise en gage, nous appelons cet oracle `Commit`.

8.3.3 Conjonction de deux SPHF génériques

Considérons un schéma de chiffrement ou de mise en gage défini par des paramètres publics et une clef publique agrégés dans ρ. X est l'ensemble des éléments que l'on veut étudier (chiffrés, n-uplets de chiffrés, mises en gage, etc.), et $\mathrm{L}_1 = \mathrm{L}_{1,\rho,\mathsf{aux}}$ et $\mathrm{L}_2 = \mathrm{L}_{2,\rho,\mathsf{aux}}$ sont des *hard partitioned subsets* de X, qui spécifient les propriétés attendues (chiffrés valides, chiffrés d'un clair spécifique, etc.). On considère les situations dans lesquelles X possède une structure de groupe, ce qui est le cas si l'on considère des chiffrés ou des n-uplets de chiffrés obtenus par un schéma de chiffrement homomorphe. On note donc $\oplus$ la loi commutative de groupe (et $\ominus$ la loi opposée, telle que $c \oplus a \ominus a = c$).

On suppose l'existence de deux *smooth hash systems* SHS_1 et SHS_2, sur les ensembles correspondants aux langages L_1 et L_2 : $\mathsf{SHS}_i = \{\mathsf{HashKG}_i, \mathsf{ProjKG}_i, \mathsf{Hash}_i, \mathsf{ProjHash}_i\}$. Ici, HashKG_i et ProjKG_i symbolisent les générateurs de la clef de hachage et de la clef projetée, et Hash_i et $\mathsf{ProjHash}_i$ les algorithmes qui calculent la fonction de hachage utilisant hk_i et hp_i.

Soient c un élément de X, et r_1 et r_2 deux éléments choisis au hasard. On appelle les clefs de hachage $\mathsf{hk}_1 = \mathsf{HashKG}_1(\rho,\mathsf{aux},r_1)$ et $\mathsf{hk}_2 = \mathsf{HashKG}_2(\rho,\mathsf{aux},r_2)$, et les clefs projetées $\mathsf{hp}_1 = \mathsf{ProjKG}_1(\mathsf{hk}_1;\rho,\mathsf{aux},c)$, et $\mathsf{hp}_2 = \mathsf{ProjKG}_2(\mathsf{hk}_2;\rho,\mathsf{aux},c)$. Un SHS pour le langage $\mathrm{L} = \mathrm{L}_1 \cap \mathrm{L}_2$ est alors défini comme suit, si $c \in \mathrm{L}_1 \cap \mathrm{L}_2$ et w_i est un témoin que $c \in \mathrm{L}_i$, pour $i = 1, 2$:

$$\begin{aligned}
\mathsf{HashKG}_{\mathrm{L}}(\rho,\mathsf{aux},r = r_1\|r_2) &= \mathsf{hk} = (\mathsf{hk}_1,\mathsf{hk}_2)\\
\mathsf{ProjKG}_{\mathrm{L}}(\mathsf{hk};\rho,\mathsf{aux},c) &= \mathsf{hp} = (\mathsf{hp}_1,\mathsf{hp}_2)\\
\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk};\rho,\mathsf{aux},c) &= \mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},c) \oplus \mathsf{Hash}_2(\mathsf{hk}_2;\rho,\mathsf{aux},c)\\
\mathsf{ProjHash}_{\mathrm{L}}(\mathsf{hp};\rho,\mathsf{aux},c;(w_1,w_2)) &= \mathsf{ProjHash}_1(\mathsf{hp}_1;\rho,\mathsf{aux},c;w_1)\\
&\quad \oplus \mathsf{ProjHash}_2(\mathsf{hp}_2;\rho,\mathsf{aux},c;w_2)
\end{aligned}$$

8.3.4 Disjonction de deux SPHF génériques

Soient L_1 et L_2 deux langages comme décrits ci-dessus. On suppose que deux systèmes SHS_1 et SHS_2 sont donnés par rapport à ces langages. On définit $\mathrm{L} = \mathrm{L}_1 \cup \mathrm{L}_2$ et on construit un SHS pour ce langage comme suit :

$$\begin{aligned}
\mathsf{HashKG}_{\mathrm{L}}(\rho,\mathsf{aux},r = r_1\|r_2) &= \mathsf{hk} = (\mathsf{hk}_1,\mathsf{hk}_2)\\
\mathsf{ProjKG}_{\mathrm{L}}(\mathsf{hk};\rho,\mathsf{aux},c) &= \mathsf{hp} = (\mathsf{hp}_1,\mathsf{hp}_2,\mathsf{hp}_\Delta = \mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},c)\\
&\quad \oplus \mathsf{Hash}_2(\mathsf{hk}_2;\rho,\mathsf{aux},c))\\
\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk};\rho,\mathsf{aux},c) &= \mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},c)\\
\mathsf{ProjHash}_{\mathrm{L}}(\mathsf{hp};\rho,\mathsf{aux},c;w) &= \mathsf{ProjHash}_1(\mathsf{hp}_1;\rho,\mathsf{aux},c;w) \qquad \text{si } c \in \mathrm{L}_1\\
&\text{ou } \mathsf{hp}_\Delta \ominus \mathsf{ProjHash}_2(\mathsf{hp}_2;\rho,\mathsf{aux},c;w) \qquad \text{si } c \in \mathrm{L}_2
\end{aligned}$$

où w est un témoin que $c \in \mathrm{L}_i$ pour un $i \in \{1,2\}$. Alors $\mathsf{ProjHash}_i(\mathsf{hp}_i;\rho,\mathsf{aux},c;w) = \mathsf{Hash}_i(\mathsf{hk}_i;\rho,\mathsf{aux},c)$. Le joueur chargé de calculer cette valeur est supposé connaître le témoin w, et en particulier le langage auquel appartient c et donc l'indice i.

8.3.5 Uniformité et indépendence

Dans la définition ci-dessus de SPHF (contrairement à la définition originale de Cramer-Shoup [CS02]), la valeur de la clef projetée dépend formellement du chiffré ou de la mise en gage c. Cependant, dans certains cas, on ne souhaite pas nécessairement révéler d'information sur cette dépendance. En fait, dans certains cas comme dans la construction de SPHF pour des mises en gage équivocables et extractibles dans la section 8.5, on peut même ne pas vouloir révéler une quelconque information sur les éléments auxiliaires aux. Quand aucune information n'est révélée sur aux, cela signifie que les détails sur le langage exact vont être dissimulés.

On ajoute alors une notion similaire à la *smoothness*, mais pour la clef projetée : elle peut dépendre (ou non) de c (et aux) dans le calcul, mais sa distribution n'en dépend pas : notons $\mathrm{D}_{\rho,\mathsf{aux},c}$ la distribution $\{\mathsf{hp} \mid \mathsf{hk} = \mathsf{HashKG}_{\mathrm{L}}(\rho,\mathsf{aux},r)$ et $\mathsf{hp} = \mathsf{ProjKG}_{\mathrm{L}}(\mathsf{hk};\rho,\mathsf{aux},c)\}$ sur les clefs projetées. Si, pour tout $c,c' \in \mathrm{X}$, $\mathrm{D}_{\rho,\mathsf{aux},c'}$ et $\mathrm{D}_{\rho,\mathsf{aux},c}$ sont indistinguables, alors on dit que le SHS a la propriété de *1-uniformité*. Si, pour tout $c,c' \in \mathrm{X}$, et pour tous éléments auxiliaires aux et aux', $\mathrm{D}_{\rho,\mathsf{aux}',c'}$ et $\mathrm{D}_{\rho,\mathsf{aux},c}$ sont indistinguables, on l'appelle propriété de *2-uniformité*.

En plus de l'indistinguabilité des distributions, la clef projetée hp elle-même peut ne pas dépendre du tout de c, comme dans la définition de Cramer et Shoup. On dit alors que le SHS garantit la propriété de *1-indépendance* (resp. *2-indépendance* si elle ne dépend pas non plus de aux). Il est à noter que ces notions d'indépendance impliquent immédiatement les notions d'uniformité respectives.

Comme exemple, le SHS associé au cryptosystème d'ElGamal (voir section 8.4.1 page 126) garantit la 2-indépendance. En revanche, le système analogue associé au chiffrement de Cramer-Shoup (voir la section 8.6.1) ne garantit que la 2-uniformité. Pour des combinaisons de SHS, on peut noter que dans le cas de disjonctions, on peut obtenir au maximum la propriété d'uniformité, puisque des calculs de hachage sur les mises en gage sont nécessaires pour obtenir la clef projetée. En outre, ceci n'est satisfait que sous la condition que les deux SHS sous-jacents satisfont déjà cette propriété (voir la section 8.3.6 pour plus de détails et les preuves).

Finalement, on doit noter qu'en cas de disjonction, la valeur hachée projetée pourrait laisser fuir de l'information à propos du sous-langage auquel appartient l'entrée, si un adversaire envoie une fausse hp_Δ. L'adversaire peut en fait vérifier si $\mathsf{ProjHash}_{\mathrm{L}}(\mathsf{hp};\rho,\mathsf{aux},c;w)$ est égal à $\mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},c)$ ou $\mathsf{hp}_\Delta \ominus \mathsf{Hash}_2(\mathsf{hk}_2;\rho,\mathsf{aux},c)$. Mais d'abord, cela ne contredit aucune notion de sécurité pour les SHS ; ensuite, dans toutes les applications ci-dessous, la valeur hachée projetée ne sera jamais révélée.

8.3.6 Preuves

On se concentre sur les preuves pour les disjonctions, puisque le cas $\mathrm{L} = \mathrm{L}_1 \cap \mathrm{L}_2$ peut être traité de la même façon, en plus simple : $(\mathsf{hp}_1,\mathsf{hp}_2,\mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},x) \oplus \mathsf{Hash}_2(\mathsf{hk}_2;\rho,\mathsf{aux},x))$ est statistiquement (si $x \notin \mathrm{L}_1$ ou $x \notin \mathrm{L}_2$) ou calculatoirement (si $x \in \mathrm{L}_1 \cap \mathrm{L}_2$) indistinguable du triplet $(\mathsf{hp}_1,\mathsf{hp}_2,g)$.

Nous supposons donc $\mathrm{L} = \mathrm{L}_1 \cup \mathrm{L}_2$, et nous étudions l'information supplémentaire donnée par la clef projetée, qui contient hp_1 et hp_2, mais aussi $\mathsf{hp}_\Delta = \mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},x) \oplus \mathsf{Hash}_2(\mathsf{hk}_2;\rho,\mathsf{aux},x)$. Comme SHS_1 et SHS_2 sont deux SPHF, la propriété pseudo-aléatoire de chacun d'entre eux (ou même la *smoothness*, si x n'appartient à aucun des deux langages) garantit que les paires $(\mathsf{hp}_i,\mathsf{Hash}_i(\mathsf{hk}_i;\rho,\mathsf{aux},x))$ sont (statistiquement ou calculatoirement) indistinguables de (hp_i,g_i). Par conséquent, on voit facilement que $(\mathsf{hp}_1,\mathsf{hp}_2,\mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},x),$ $\mathsf{Hash}_2(\mathsf{hk}_2;\rho,\mathsf{aux},x))$ est (statistiquement ou calculatoirement) indistinguable de $(\mathsf{hp}_1,\mathsf{hp}_2,$ $g_1,g_2)$, où g_1 et g_2 sont indépendantes. Ceci implique de plus que $(\mathsf{hp}_1,\mathsf{hp}_2,\mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},x)$ $\oplus$ $\mathsf{Hash}_2(\mathsf{hk}_2;\rho,\mathsf{aux},x))$ est (statistiquement ou calculatoirement) indistinguable du triplet $(\mathsf{hp}_1,\mathsf{hp}_2,g)$: l'élément hp_Δ ne procure pas donc plus d'information que hp_1 ou hp_2.

Hachage efficace à partir de la clef. Étant donné un élément quelconque $x \in \mathrm{X}$ et une clef hk, il est possible de calculer efficacement $\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk};\rho,\mathsf{aux},x)$.

Démonstration. Ceci provient des hachages efficaces des deux SPHF sous-jacentes, et en particulier de SHS_1, puisque $\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk};\rho,\mathsf{aux},x) = \mathsf{Hash}_1(\mathsf{hk}_1;\rho,\mathsf{aux},x)$. □

Hachage efficace à partir de la clef projetée. Étant donné un élément $x \in \mathrm{L}$, un témoin w de cette appartenance, et la clef projetée $\mathsf{hp} = \mathsf{ProjKG}_{\mathrm{L}}(\mathsf{hk}; \rho, \mathsf{aux}, x)$, il est possible de calculer efficacement la valeur hachée projetée $\mathsf{ProjHash}_{\mathrm{L}}(\mathsf{hp}; \rho, \mathsf{aux}, x, w) = \mathsf{Hash}_{\mathrm{L}}(\mathsf{hk}; \rho, \mathsf{aux}, x)$.

Démonstration. Si $x \in \mathrm{L}_1$, alors

$$\begin{aligned}\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk}; \rho, \mathsf{aux}, x) &= \mathsf{Hash}_1(\mathsf{hk}_1; \rho, \mathsf{aux}, x)\\ &= \mathsf{ProjHash}_1(\mathsf{hp}_1; \rho, \mathsf{aux}, x, w) = \mathsf{ProjHash}_{\mathrm{L}}(\mathsf{hp}; \rho, \mathsf{aux}, x, w),\end{aligned}$$

qui peut être calculé efficacement puisque SHS_1 est une SPHF. Si $x \in \mathrm{L}_2$, alors

$$\begin{aligned}\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk}; \rho, \mathsf{aux}, x) &= \mathsf{Hash}_1(\mathsf{hk}_1; \rho, \mathsf{aux}, x)\\ &= \mathsf{ProjHash}_1(\mathsf{hp}_1; \rho, \mathsf{aux}, x, w) = \mathsf{hp}_\Delta \ominus \mathsf{ProjHash}_2(\mathsf{hp}_2; \rho, \mathsf{aux}, x, w),\end{aligned}$$

qui peut être calculé efficacement puisque SHS_2 est une SPHF. □

Smoothness. Pour tout élément $x \in \mathrm{X} \setminus \mathrm{L}$, $\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk}; \rho, \mathsf{aux}, x)$ est uniformément distribué, étant donné la clef projetée.

Démonstration. Considérons $x \notin \mathrm{L}$. Alors $x \notin \mathrm{L}_1$ et $x \notin \mathrm{L}_2$. Si $\mathsf{hk} = (\mathsf{hk}_1, \mathsf{hk}_2)$ est une clef aléatoire, et $\mathsf{hp} = (\mathsf{hp}_1, \mathsf{hp}_2, \mathsf{hp}_\Delta)$ la clef projetée correspondante, alors l'analyse précédente a montré l'indistinguabilité statistique entre $(\mathsf{hp}_1, \mathsf{hp}_2, \mathsf{Hash}_1(\mathsf{hk}_1; \rho, \mathsf{aux}, x), \mathsf{Hash}_2(\mathsf{hk}_2; \rho, \mathsf{aux}, x))$ et $(\mathsf{hp}_1, \mathsf{hp}_2, g_1, g_2)$, où g_1 et g_2 sont des éléments aléatoires et indépendants. Ceci implique l'indistinguabilité statistique entre $(\mathsf{hp}_1, \mathsf{hp}_2, \mathsf{hp}_\Delta, \mathsf{Hash}_1(\mathsf{hk}_1; \rho, \mathsf{aux}, x))$ et $(\mathsf{hp}_1, \mathsf{hp}_2, g_1, g_2)$. Par suite, le couple $(\mathsf{hp}, \mathsf{Hash}_{\mathrm{L}}(\mathsf{hk}; \rho, \mathsf{aux}, x))$ est statistiquement indistinguable de (hp, g), ce qui est la définition de la *smoothness* pour le nouveau système. □

Caractère pseudo-aléatoire. Pour chaque élément $x \in \mathrm{L}$, la valeur $\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk}; \rho, \mathsf{aux}, x)$ est calculatoirement indistinguable d'une valeur uniforme, étant donnée la clef projetée.

Démonstration. On peut effectuer exactement la même analyse qu'au paragraphe précédent, mais avec l'indistinguabilité calculatoire quand $x \in \mathrm{L}_1$ ou $x \in \mathrm{L}_2$. On obtient alors l'indistinguabilité calculatoire du couple $(\mathsf{hp}, \mathsf{Hash}_{\mathrm{L}}(\mathsf{hk}; \rho, \mathsf{aux}, x))$ par rapport à (hp, g), ce qui est la définition du caractère pseudo-aléatoire pour le nouveau système. □

Préservation des propriétés d'uniformité et d'indépendance. Si les deux SHS sous-jacents vérifient la 1-uniformité (resp. la 2-uniformité), alors le SHS pour la conjonction ou la disjonction vérifie cette propriété. Si les deux SHS sous-jacents vérifient la 1-indépendance (resp. la 2-indépendance), alors le SHS pour la conjonction *uniquement* vérifie cette propriété. Nous insistons sur le fait que l'indépendance ne se propage pas à la disjonction, puisque la valeur hachée (qui nécessite aux et x) est incluse dans la clef projetée.

Démonstration. Nous ne prouvons que le résultat de 1-uniformité pour la disjonction (la preuve est la même dans les autres cas).

Si (ρ, aux) sont les paramètres pour les langages et x et x' deux éléments de L, alors $\mathrm{D}_{1,\rho,\mathsf{aux},x} \approx \mathrm{D}_{1,\rho,\mathsf{aux},x'}$ et $\mathrm{D}_{2,\rho,\mathsf{aux},x} \approx \mathrm{D}_{2,\rho,\mathsf{aux},x'}$. Grâce à la forme de hp, ceci assure que les deux premiers éléments de hp sont indistinguables. Considérons à présent la troisième partie. Sans perte de généralité, nous pouvons supposer que $x \in \mathrm{L}_1$. Alors, grâce au caractère pseudo-aléatoire du premier SHS, la valeur $\mathsf{Hash}_1(\mathsf{hk}_1; \rho, \mathsf{aux}, x)$ est calculatoirement indistinguable d'une valeur uniforme. C'est au moins la même chose pour la valeur $\mathsf{Hash}_2(\mathsf{hk}_2; \rho, \mathsf{aux}, x)$. Puisque les clefs projetées dépendent de valeurs et de langages aléatoires et indépendants, les deux membres du $\oplus$ sont indépendants : la valeur $\mathsf{Hash}_{\mathrm{L}}(\mathsf{hk}_{\mathrm{L}}; \rho, \mathsf{aux}, x)$ est alors indistinguable d'une valeur uniforme. Par suite, $\mathrm{D}_{\rho,\mathsf{aux},x} \approx \mathrm{D}_{\rho,\mathsf{aux},x'}$. □

8.4 Une mise en gage conditionnellement extractible

8.4.1 Mise en gage ElGamal et SPHF associée

La mise en gage ElGamal est réalisée dans le modèle de la CRS, où la CRS ρ contient $(\mathrm{G}, \mathsf{pk})$, comme définie dans la section 8.2, pour le schéma de chiffrement ElGamal. En pratique, sk ne devrait être connue de personne, mais dans l'analyse de sécurité, sk va être la trappe d'extraction. Supposons que l'entrée d'un algorithme de mise en gage est un scalaire $\mathrm{M} \in \mathbb{Z}_q$. L'algorithme de mise en gage consiste à choisir une valeur aléatoire r et calculer le chiffré ElGamal suivant sous l'aléa r : $\mathrm{C} = \mathsf{EG}^+_{\mathsf{pk}}(\mathrm{M}, r) = (u_1 = {g_1}^r, e = h^r g^{\mathrm{M}})$.

Le langage $\mathrm{L} = \mathrm{L}_{(\mathbf{EG}^+,\rho),\mathrm{M}} \subset \mathrm{X} = \mathrm{G}^2$ des chiffrés ElGamal additifs C de M sous les paramètres globaux et la clef publique définie par ρ est un *hard partitioned subset* de $\mathrm{X} = \mathrm{G}^2$ sous l'hypothèse DDH (sécurité sémantique du schéma de chiffrement ElGamal).

La SPHF associée à ce schéma de mise en gage et au langage L est donc la famille basée sur le schéma de chiffrement ElGamal, comme défini dans [GL03]. Pour $\mathrm{C} = \mathsf{EG}^+_{\mathsf{pk}}(\mathrm{M}, r)$, et avec le témoin r, les algorithmes sont donc définis de la façon suivante en utilisant les mêmes notations que dans [GL03] :

$$\begin{aligned}
&\mathsf{HashKG}((\mathbf{EG}^+, \rho), \mathrm{M}) = \mathsf{hk} = (\gamma_1, \gamma_3) \xleftarrow{\$} \mathbb{Z}_q \times \mathbb{Z}_q \\
&\mathsf{ProjKG}(\mathsf{hk}; (\mathbf{EG}^+, \rho), \mathrm{M}, \mathrm{C}) = \mathsf{hp} = (g_1)^{\gamma_1} (h)^{\gamma_3} \\
&\mathsf{Hash}(\mathsf{hk}; (\mathbf{EG}^+, \rho), \mathrm{M}, \mathrm{C}) = (u_1)^{\gamma_1} (e g^{-\mathrm{M}})^{\gamma_3} \\
&\mathsf{ProjHash}(\mathsf{hp}; (\mathbf{EG}^+, \rho), \mathrm{M}, \mathrm{C}; r) = (\mathsf{hp})^r
\end{aligned}$$

8.4.2 Mises en gage L-extractible et SPHF correspondantes

Notons que la valeur g^{M} serait aisément extractible à partir de cette mise en gage (vue comme le chiffrement ElGamal multiplicatif). Cependant, on ne peut extraire M lui-même (la valeur en gage) que si sa taille est suffisamment petite afin que sa valeur puisse être obtenue comme la solution au problème du logarithme discret. Afin d'obtenir l'« extractibilité », on doit donc plutôt mettre en gage M bit par bit.

On écrit $\mathrm{M} \in \mathbb{Z}_q$ sous la forme $\sum_{i=1}^{m} \mathrm{M}_i \cdot 2^{i-1}$, avec $\mathrm{M}_i \in \{0, 1\}$, où $m \leqslant n$. Sa mise en gage est $\mathsf{comEG}_{\mathsf{pk}}(\mathrm{M}) = (b_1, \ldots, b_m)$, où $b_i = \mathsf{EG}^+_{\mathsf{pk}}(\mathrm{M}_i \cdot 2^{i-1}, r_i) = (u_{1,i} = {g_1}^{r_i}, e_i = h^{r_i} g^{\mathrm{M}_i \cdot 2^{i-1}})$, pour $i = 1, \ldots, m$. La propriété homomorphe du schéma de chiffrement permet d'obtenir, à partir de ce m-uplet, la mise en gage simple de M ci-dessus

$$\mathrm{C} = \mathsf{EG}^+_{\mathsf{pk}}(\mathrm{M}, r) = (u_1, e) = (\prod u_{1,i}, \prod e_i) = \prod b_i, \quad \text{avec } r = \sum r_i$$

Précisons désormais ce que l'on entend par « extractibilité » : ici, la mise en gage va être extractible si les messages M_i sont des bits (ou au moins des éléments suffisamment petits), mais on ne peut pas assurer qu'elle sera extractible sinon. Plus généralement, cela mène à une nouvelle notion de *mises en gage* $\mathrm{L}-$*extractible*, ce qui signifie que l'on autorise la primitive à ne pas être extractible si le message n'appartient pas à un certain langage L (par exemple le langage des chiffrés de 0 ou 1). Ce langage L est informellement le langage de toutes les mises en gage valides et « de la bonne forme », et est aussi inclus dans l'ensemble X de toutes les mises en gage.

Pour le protocole ci-dessus, on a besoin d'un SHS pour le langage $\mathrm{L} = \mathrm{L}_1 \cap \mathrm{L}_2$, où $\mathrm{L}_1 = \{(b_1, \ldots, b_m) \mid \forall i, b_i \in \mathrm{L}_{(\mathbf{EG}^+,\rho),0\vee1}\}$, $\mathrm{L}_2 = \{(b_1, \ldots, b_m) \mid \mathrm{C} = \prod_i b_i \in \mathrm{L}_{(\mathbf{EG}^\times,\rho),g^{\mathrm{M}}}\}$, à un facteur près (correspondant au décalage 2^{i-1}) avec

$$\begin{aligned}
\mathrm{L}_{(\mathbf{EG}^+,\rho),0\vee1} &= \mathrm{L}_{(\mathbf{EG}^+,\rho),0} \cup \mathrm{L}_{(\mathbf{EG}^+,\rho),1} & \mathrm{L}_{(\mathbf{EG}^+,\rho),0} &= \{\mathrm{C} \mid \exists r\ \mathrm{C} = \mathsf{EG}^+_{\mathsf{pk}}(0, r)\} \\
\mathrm{L}_{(\mathbf{EG}^\times,\rho),g^{\mathrm{M}}} &= \{\mathrm{C} \mid \exists r\ \mathrm{C} = \mathsf{EG}^\times_{\mathsf{pk}}(g^{\mathrm{M}}, r)\} & \mathrm{L}_{(\mathbf{EG}^+,\rho),1} &= \{\mathrm{C} \mid \exists r\ \mathrm{C} = \mathsf{EG}^+_{\mathsf{pk}}(1, r)\}
\end{aligned}$$

Il est facile de voir que ceci revient à construire un SHS correspondant à une conjonction et disjonction de langages, comme présenté dans la section précédente.

8.4.3 Certification de clefs publiques

Description. Une application classique des mises en gage extractibles est la certification des clefs publiques (lorsque l'on veut être sûr qu'une personne joignant le système connaît effectivement la clef privé associée). Supposons qu'un utilisateur U possède une paire de clefs secrète et publique, et voudrait avoir sa clef publique certifiée par l'autorité. Une propriété naturelle est que l'autorité ne va pas certifier cette clef publique sans être sûre que l'utilisateur possède effectivement la clef privée reliée, ce qui est assuré par une preuve de connaissance *zero-knowledge* : l'utilisateur connaît la clef privée s'il existe un extracteur de cette dernière.

Nous présentons ici une construction qui possède la même propriété sans requérir une preuve explicite de connaissance, de plus de manière concurrente étant donné qu'il n'y aucun besoin de rembobiner :

- l'utilisateur envoie sa clef publique g^{M}, ainsi qu'une mise en gage bit-à-bit L-extractible de la clef privée M, c'est-à-dire un m-uplet $\mathsf{comEG}_{\mathsf{pk}}(\mathrm{M}) = (b_1, \ldots, b_m)$ comme décrit plus haut, à partir duquel on peut dériver $\mathrm{C} = \prod b_i = \mathsf{EG}^{+}_{\mathsf{pk}}(\mathrm{M}, r) = \mathsf{EG}^{\times}_{\mathsf{pk}}(g^{\mathrm{M}}, r)$.
- Nous pouvons définir le SHS relié au langage $\mathrm{L}_1 \cap \mathrm{L}_2$, avec $\mathrm{L}_1 = \cap_i \mathrm{L}_{1,i}$, où $\mathrm{L}_{1,i}$ est le langage des m-uplets dans lesquels la i-ième composante b_i est un chiffré de 0 ou 1, et L_2 est le langage des m-uplets dans lesquels le $\mathrm{C} = \prod b_i$ dérivé est un chiffré de la clef publique g^{M} (sous le ElGamal multiplicatif, comme dans la section 8.4.1 page 126).

  Notons que lorsque le m-uplet $(b_1, \ldots, b_m)$ appartient à $\mathrm{L}_1 \cap \mathrm{L}_2$, cela correspond à une mise en gage extractible de la clef privée M associée à la clef publique g^{M} : chaque b_i chiffre un bit, et peut alors être déchiffré, ce qui donne le i-ième bit de M.
- L'autorité calcule une clef de hachage hk, la clef projetée correspondante hp sur le m-uplet $(b_1, \ldots, b_m)$ et la valeur hachée reliée Hash sur $(b_1, \ldots, b_m)$. Il envoie hp à U ainsi que $\mathsf{Cert} \oplus \mathsf{Hash}$, où Cert est le certificat espéré. Notons que si Hash n'est pas assez grand, un générateur pseudo-aléatoire peut être utilisé pour l'étendre.
- L'utilisateur est alors capable de récupérer ce certificat si et seulement si il peut calculer Hash. Cette valeur peut être calculée avec l'algorithme $\mathsf{ProjHash}$ sur $(b_1, \ldots, b_m)$, à partir de hp, mais il requiert aussi un témoin w prouvant que le m-uplet $(b_1, \ldots, b_m)$ appartient à $\mathrm{L}_1 \cap \mathrm{L}_2$, ce qui signifie que l'utilisateur connaît la clef privée.

À l'aide des propriétés du SHS, si l'utilisateur calcule correctement la mise en gage, il connaît le témoin w, et peut obtenir le même masque Hash pour extraire le certificat. Si l'utilisateur a triché, la propriété de *smoothness* rend Hash parfaitement imprédictible : aucune information ne fuit à propos du certificat.

Analyse de sécurité. Voici un aperçu de la preuve de sécurité du protocole décrit ci-dessus. Le cadre de sécurité est le suivant : personne ne peut obtenir de certificat sur une clef publique sans connaître la clef privée associée (c'est-à-dire si aucun simulateur ne peut extraire la clef privée). En d'autres termes, l'adversaire gagne s'il est capable de renvoyer $(g^{\mathrm{M}}, \mathsf{Cert})$ et qu'aucun simulateur ne peut produire M.

Le jeu d'attaque formel peut alors être décrit comme suit : l'adversaire $\mathcal{A}$ interagit plusieurs fois avec l'autorité, en envoyant des clefs publiques et des mises en gage, et il demande les certificats correspondants. Il renvoie alors une paire $(g^{\mathrm{M}}, \mathsf{Cert})$ et gagne si aucun simulateur ne peut extraire M à partir de la transcription de la communication.

Le simulateur est défini ainsi : il a accès à un oracle de (signature de) certification, et génère des clefs publique et privée $(\mathsf{sk}, \mathsf{pk})$ pour le chiffrement ElGamal. La clef publique est fixée comme la CRS qui définit le schéma de mise en gage. La clef privée va alors être la trappe d'extraction.

Quand le simulateur reçoit une demande de certification, avec une clef publique et une mise en gage, il commence d'abord par essayer d'extraire la clef privée associée, étant donnée la trappe d'extraction. En cas de succès, le simulateur pose une requête à l'oracle de signature pour la fournir en tant que certificat correspondant sur la clef publique, et il termine le procédé comme décrit dans le protocole. Cependant, l'extraction peut échouer si les mises en gage ne

sont pas bien construites (pas dans $\mathrm{L}_1 \cap \mathrm{L}_2$). Dans un tel cas, le simulateur envoie alors une chaîne de bits aléatoire de taille appropriée. En cas d'extraction réussie, la réponse reçue par l'utilisateur est exactement celle attendue. En cas d'échec, elle est parfaitement indistinguable aussi étant donné que la propriété de *smoothness* de la fonction de hachage rend le masque Hash parfaitement aléatoire (puisque l'entrée n'est pas dans le langage).

Après plusieurs interactions, $\mathcal{A}$ renvoie une paire $(g^{\mathrm{M}}, \mathsf{Cert})$, transmise par le simulateur. Soit g^{M} a été posée à l'oracle de signature, ce qui signifie que l'extraction a réussi, le simulateur connaît M et l'adversaire n'a pas gagné le jeu d'attaque, soit c'est une signature valide sur un nouveau message, et c'est alors une attaque de contrefaçon existentielle à messages choisis (*existential forgery under chosen-message attack*).

8.5 Une mise en gage conditionnellement extractible et équivocable

Dans cette section, on améliore les précédents schémas de mise en gage en leur ajoutant l'équivocabilité, ce qui n'est pas une tâche triviale si l'on veut aussi conserver la propriété d'extraction. On commence par construire une mise en gage extractible et équivocable malléable basée sur ElGamal (voir section 8.4.1 page 126), et on ajoute ensuite la non-malléabilité en construisant simplement la mise en gage à l'aide du schéma Cramer-Shoup. Tous les détails de cette extension sont donnés dans la section 8.6. Dans la suite, si b est un bit, on note son complémentaire $\overline{b}$ (c'est-à-dire $\overline{b} = 1 - b$). On note en outre $x[j]$ le j-ième bit de la chaîne de bits x et plus généralement les valeurs correspondant à ce bit, en l'occurrence les $r_{i,\delta}[j]$ et $b_{i,\delta}[j]$.

8.5.1 Équivocabilité

Obtenir des mises en gage à la fois extractibles et équivocables semble être un objectif difficile à atteindre. Canetti et Fischlin [CF01] ont proposé une solution dans le modèle de la CRS, mais uniquement pour un bit. Ils ont par ailleurs aussi proposé la fonctionnalité idéale pour cette primitive et montré que vérifier ces propriétés simultanément était impossible dans le modèle de base (sans CRS ni hypothèse additionnelle). Damgård et Nielsen [DN02] ont ensuite proposé une autre construction. Mais pour des raisons d'efficacité essentiellement (des preuves de relation dans la construction de Damgård et Nielsen se révèlent assez inefficaces), dans notre contexte spécifique, on étend la première proposition, qui est bien adaptée à notre objectif de lui associer une SPHF. Dans cette section, on améliore donc notre précédente mise en gage (déjà L-extractible) pour la rendre équivocable, en utilisant l'approche de Canetti et Fischlin. Le chapitre 9 utilisera alors une version non-malléable de cette nouvelle mise en gage ainsi que la SPHF associée pour construire un protocole d'échange de clefs basé sur des mots de passe résistant contre les attaques adaptatives dans le cadre UC [Can01]. Le protocole résultant est raisonnablement efficace et en particulier, plus efficace que le protocole de Barak *et al.* [BCL$^+$05], qui est à notre connaissance le seul à atteindre le même niveau de sécurité dans le modèle standard.

Description de la mise en gage. Notre schéma de mise en gage est une extension naturelle du schéma de Canetti et Fischlin [CF01], de façon bit par bit. Il utilise le schéma de chiffrement à clef publique ElGamal, pour chaque bit de la chaîne.

Soient $y_1, \ldots, y_m$ des éléments aléatoires dans G. Cette mise en gage est réalisée dans le modèle de la CRS, cette dernière ρ contenant $(\mathrm{G}, \mathsf{pk})$, où pk est une clef publique ElGamal telle que la clef privée soit inconnue de tout le monde, à part l'extracteur de la mise en gage. Elle inclut aussi le m-uplet $(y_1, \ldots, y_m)$, dont les logarithmes discrets en base g sont inconnus de tout le monde, à part l'équivocateur de la mise en gage.

Supposons que l'entrée de l'algorithme de mise en gage soit une chaîne de bits de la forme $\pi = \sum_{i=1}^{m} \pi_i \cdot 2^{i-1}$. L'algorithme fonctionne de la façon suivante :

- Pour $i = 1, \ldots, m$, il choisit une valeur aléatoire $x_{i,\pi_i} = \sum_{j=1}^{n} x_{i,\pi_i}[j] \cdot 2^{j-1}$ et pose $x_{i,\overline{\pi}_i} = 0$.
- Pour $i = 1, \ldots, m$, l'algorithme met en gage π_i, en utilisant l'aléa x_{i,π_i} :

$$a_i = \mathsf{comPed}(\pi_i, x_{i,\pi_i}) = g^{x_{i,\pi_i}} y_i^{\pi_i}$$

  et en définissant $\mathbf{a} = (a_1, \ldots, a_m)$.
- Pour $i = 1, \ldots, m$, il calcule les mises en gage ElGamal (voir la section précédente) de $x_{i,\delta}$, pour $\delta = 0, 1$: $(\mathbf{b}_{i,\delta} = (b_{i,\delta}[j])_j = \mathsf{comEG}_{\mathsf{pk}}(x_{i,\delta}, r_{i,\delta})$, dans lesquelles $b_{i,\delta}[j] = \mathsf{EG}^{+}_{\mathsf{pk}}(x_{i,\delta}[j] \cdot 2^{j-1}, r_{i,\delta}[j])$. On peut directement extraire du calcul des $b_{i,\delta}[j]$ un chiffré $\mathrm{B}_{i,\delta}$ de $x_{i,\delta}$: $\mathrm{B}_{i,\delta} = \prod_j b_{i,\delta}[j] = \mathsf{EG}^{+}_{\mathsf{pk}}(x_{i,\delta}, r_{i,\delta})$, où $r_{i,\delta}$ est la somme des aléas $r_{i,\delta}[j]$.

La chaîne aléatoire entière pour cette mise en gage est (avec n la longueur en bits de l'ordre premier q du groupe G)

$$\mathrm{R} = (x_{1,\pi_1}, (r_{1,0}[1], r_{1,1}[1], \ldots, r_{1,0}[n], r_{1,1}[n]), \ldots, x_{m,\pi_m}, (r_{m,0}[1], \ldots, r_{m,1}[n]))$$

À partir de là, toutes les valeurs $r_{i,\overline{\pi}_i}[j]$ peuvent être effacées, laissant les données révélées à l'ouverture (témoins de la valeur engagée) limitées à

$$\mathsf{w} = (x_{1,\pi_1}, (r_{1,\pi_1}[1], \ldots, r_{1,\pi_1}[n]), \ldots, x_{m,\pi_m}, (r_{m,\pi_m}[1], \ldots, r_{m,\pi_m}[n]))$$

La sortie de l'algorithme de mise en gage, de la chaîne de bits π, en utilisant l'aléa R, est

$$\mathsf{com}_\rho(\pi; \mathrm{R}) = (\mathbf{a}, \mathbf{b})$$

où $\quad \mathbf{a} = (a_i = \mathsf{comPed}(\pi_i, x_{i,\pi_i}))_i, \mathbf{b} = (b_{i,\delta}[j] = \mathsf{EG}^{+}_{\mathsf{pk}}(x_{i,\delta}[j] \cdot 2^{j-1}, r_{i,\delta}[j]))_{i,\delta,j}$

Ouverture. Afin d'ouvrir la mise en gage sur π, le témoin w ci-dessus (ainsi que la valeur π) est en fait suffisant : on peut reconstruire, pour tous i et j, $b_{i,\pi_i}[j] = \mathsf{EG}^{+}_{\mathsf{pk}}(x_{i,\pi_i}[j] \cdot 2^{j-1}, r_{i,\pi_i}[j])$, et les vérifier avec $\mathbf{b}$. On peut alors calculer à nouveau tous les $a_i = \mathsf{comPed}(\pi_i, x_{i,\pi_i})$, et les vérifier avec $\mathbf{a}$. Les éléments aléatoires effacés aideraient à vérifier les chiffrés des zéros, ce que l'on ne souhaite pas, puisque la propriété d'équivocabilité va exploiter ce point.

Propriétés. Vérifions rapidement les propriétés de sécurité, qui sont formellement prouvées dans la section 8.5.2. D'abord, grâce au fait que la mise en gage de Pedersen est parfaitement *hiding*, sauf si de l'information fuit à propos des $x_{i,\delta}[j]$, aucune information ne fuit à propos des π_i. Le caractère privé est aussi garanti grâce à la sécurité sémantique du schéma de chiffrement ElGamal. Comme la mise en gage de Pedersen est calculatoirement *binding*, les a_i ne peuvent pas être ouverts de deux manières différentes : une seule paire (π_i, x_{i,π_i}) est possible. Considérons maintenant les nouvelles propriétés suivantes :

- l'extractibilité (conditionnelle) est procurée par le chiffrement bit à bit. Avec la clef de déchiffrement sk, on peut déchiffrer tous les $b_{i,\delta}[j]$ et obtenir tous les $x_{i,\delta}$ (sauf si les chiffrés contiennent des valeurs autres que 0 ou 1, ce qui va être une condition pour l'extractibilité). On peut alors vérifier, pour $i = 1, \ldots, m$, que $a_i = \mathsf{comPed}(0, x_{i,0})$ ou $a_i = \mathsf{comPed}(1, x_{i,1})$, ce qui donne π_i (sauf si aucune des égalités n'est satisfaites, ce qui va être une autre condition pour l'extractibilité).
- l'équivocabilité est possible en utilisant la trappe de la mise en gage de Pedersen. Au lieu de prendre un x_{i,π_i} aléatoire et poser ensuite $x_{i,\overline{\pi}_i} = 0$, ce qui spécifie π_i comme le bit mis en gage, on prend un $x_{i,0}$ aléatoire, on calcule $a_i = \mathsf{comPed}(0, x_{i,0})$, mais on extrait aussi $x_{i,1}$ tel que $a_i = \mathsf{comPed}(1, x_{i,1})$ (ce qui est possible grâce à la connaissance du logarithme discret de y_i en base g, qui est la trappe de la mise en gage de Pedersen). Le reste de la procédure de mise en gage reste identique, mais maintenant on peut ouvrir n'importe quelle chaîne de bits pour π, en utilisant le x_{i,π_i} approprié et les éléments aléatoires correspondants (le simulateur n'a rien effacé).

8.5.2 Preuves

EXTRACTIBILITÉ. La clef d'extraction est sk, la clef de déchiffrement ElGamal (ou Cramer-Shoup, dans la version non-malléable de cette primitive, présentée page 131). Le simulateur commence par essayer de déchiffrer tous les $b_{i,\delta}[j]$ pour obtenir $x_{i,\delta}[j]$ et abandonne si l'un d'entre eux n'est ni 0 ni 1. Il construit alors les $x_{i,\delta}$, et vérifie que $a_i = g^{x_{i,0}}$ ou $a_i = g^{x_{i,1}} y_i$, ce qui lui permet de récupérer π_i (sauf en cas d'échec). Cette extraction n'échoue que dans les trois cas suivants :

- si les chiffrés ne chiffrent ni 0 ni 1 ;
- si a_i ne satisfait aucune des égalités ;
- si a_i satisfait les deux égalités.

Les deux premières raisons vont être exclues dans le langage L, tandis que la troisième contredirait la propriété de *binding* de la mise en gage de Pedersen, ce qui mènerait au logarithme discret d'un certain y_i en base g. Ceci n'arrive donc qu'avec probabilité négligeable.

EQUIVOCABILITÉ. Étant donné les logarithmes discrets de $(y_1, \ldots, y_m)$, on est capable de calculer, pour tout $i \in \{1, \ldots, m\}$, à la fois $x_{i,0}$ et $x_{i,1}$. L'équivocation consiste alors à calculer, pour tout i, un chiffrement à la fois de $x_{i,0}$ et $x_{i,1}$ (et pas de 0). Si l'on n'efface aucun aléa, cela permet de changer d'avis et d'ouvrir chaque mise en gage sur n'importe lequel des π_i (0 ou 1).

Prouvons à présent que mettre en gage toutes les chaînes de bits π dans une mise en gage unique ne change pas la vue d'un adversaire. La preuve est basée sur un argument hybride *Left-or-Right*, voir [BDJR97]. Supposons qu'il existe un oracle répondant soit $\mathcal{E}(g^x)$ soit $\mathcal{E}(g^0)$ quand on lui envoie g^x (g^0 étant donné implicitement). Nous définissons alors des jeux hybrides dans lesquels les chiffrements de $x_{i,\pi_i}[j]$ dans la mise en gage sont calculés à l'aide de cet oracle. Le premier jeu hybride, dans lequel l'oracle chiffre toujours g^0, est équivalent au calcul réel, dans lequel uniquement une chaîne de bits est mise en gage (parfaitement *binding*). De même, le dernier jeu, dans lequel il chiffre toujours g^x, est équivalent à la simulation, dans laquelle toutes les chaînes de bits sont mises en gage (équivocabilité).

G_{1a}		$\mathrm{G}_{1b} = \mathrm{G}_{2a}$		$\mathrm{G}_{2b} = \mathrm{G}_{3a}$		$\mathrm{G}_{mn,b}$
1		$g^{x_{1,\overline{\pi}_1}[1]}$		$g^{x_{1,\overline{\pi}_1}[1]}$		$g^{x_{1,\overline{\pi}_1}[1]}$
1		1		$g^{x_{1,\overline{\pi}_1}[2]}$		$g^{x_{1,\overline{\pi}_1}[2]}$
$\vdots$		$\vdots$		$\vdots$	$\cdots$	$\vdots$
1		1		1		$g^{x_{m,\overline{\pi}_m}[n]}$
	$\longleftrightarrow$		$\longleftrightarrow$		$\longleftrightarrow$	
	$\mathsf{Adv}_{\mathsf{cpa}}(\mathcal{E})$		$\mathsf{Adv}_{\mathsf{cpa}}(\mathcal{E})$		$\mathsf{Adv}_{\mathsf{cpa}}(\mathcal{E})$	

HIDING. Cette propriété suit directement de l'équivocabilité de la mise en gage.

BINDING. Supposons d'abord que l'on ne connaisse pas les logarithmes discrets de $(y_1, \ldots, y_m)$ en base g et que l'adversaire ait réussi à envoyer une mise en gage qui peut être ouverte sur π et sur π'. Alors, pour un certain bit $i = 1, \ldots, m$, $\pi_i \neq \pi'_i$, et l'adversaire est capable de donner $x_i \neq x'_i$ tel que $g^{x_i} y^{\pi_i} = g^{x'_i} y^{\pi'_i}$. Cet événement revient alors à casser le problème du logarithme discret, ce qui n'arrive qu'avec probabilité négligeable. Cela signifie que si l'équivocabilité n'est pas utilisée, la mise en gage est calculatoirement *binding*, sous le problème du logarithme discret.

Maintenant, puisqu'une mise en gage réelle (avec un chiffré de 0) et une mise en gage équivocable (sans chiffré de 0) sont indistinguables pour un adversaire, la vue de mises en gage équivocables n'aide pas l'adversaire à casser la propriété de *binding*, ou sinon il casserait la propriété IND-CPA pour le schéma de chiffrement sous-jacent.

8.5.3 La SPHF associée

Comme on l'a vu ci-dessus, notre nouveau schéma de mise en gage est conditionnellement extractible (on peut récupérer les $x_{i,\delta}$, et la valeur engagée π), sous la condition que tous les chiffrés ElGamal chiffrent soit 0 soit 1, et que les a_i soient des mises en gage de 0 ou 1, avec pour aléa $x_{i,0}$ ou $x_{i,1}$.

Comme auparavant, on veut faire en sorte que les deux valeurs de hachage (le calcul direct et celui à partir de la clef projetée) soient les mêmes si les deux joueurs utilisent la même entrée π et parfaitement indépendantes s'ils utilisent des entrées différentes (*smoothness*). On veut en outre pouvoir contrôler que chaque a_i est effectivement une mise en gage de Pedersen de π_i utilisant l'aléa chiffré x_{i,π_i}, et donc $g^{x_{i,\pi_i}} = a_i/y_i^{\pi_i}$: la valeur extraite x_{i,π_i} est vraiment la clef privée M reliée à une clef publique donnée g^{M} qui est $a_i/y_i^{\pi_i}$ dans notre cas. En utilisant les mêmes notations que dans la section 8.4.1 page 126, on veut définir un SHS montrant que, pour tout i, δ, j, $b_{i,\delta}[j] \in \mathrm{L}_{(\mathbf{EG}^+,\rho),0\vee1}$ et, pour tout i, $\mathrm{B}_{i,\pi_i} \in \mathrm{L}_{(\mathbf{EG}^\times,\rho),(a_i/y_i{}^{\pi_i})}$, où $\mathrm{B}_{i,\pi_i} = \prod_j b_{i,\pi_i}[j]$.

Combinaisons de ces SHS. Soit C la mise en gage de π ci-dessus utilisant l'aléa R comme défini dans la section 8.5.1 page 128. On précise maintenant le langage $\mathrm{L}_{\rho,\pi}$, consistant informellement en toutes les mises en gage valides « de la bonne forme » :

$$\mathrm{L}_{\rho,\pi} = \left\{ \mathrm{C} \;\middle|\; \begin{array}{ll} \exists \mathrm{R} \text{ tel que } \mathrm{C} = \mathsf{com}_\rho(\pi, \mathrm{R}) & \text{et } \forall i\ \forall j \quad b_{i,\pi_i}[j] \in \mathrm{L}_{(\mathbf{EG}^+,\rho),0\vee1} \\ & \text{et } \forall i \quad \mathrm{B}_{i,\pi_i} \in \mathrm{L}_{(\mathbf{EG}^\times,\rho),a_i/y_i{}^{\pi_i}} \end{array} \right\}$$

Le SHS pour ce langage repose sur les SHS décrits précédemment, en utilisant la construction générique pour les conjonctions et disjonctions décrite dans la section 8.3. La définition précise de ce langage (construit comme des conjonctions et disjonctions de langages plus simples) se trouve dans la section 8.6, en omettant les étiquettes et en remplaçant le chiffrement Cramer-Shoup $\mathbf{CS}^+$ par le chiffrement ElGamal $\mathbf{EG}^+$.

Propriétés : uniformité et indépendance. Leur énoncé et leur preuve est similaire à celles de la section 8.6.

Estimation de la complexité. Globalement, chaque opération (mise en gage, clef projetée, hachage et hachage projeté) requiert $\mathcal{O}(mn)$ exponentiations dans G, avec des constantes petites (au plus 16). Le détail peut être trouvé dans la section 8.6.

8.6 Une mise en gage non-malléable conditionnellement extractible et équivocable

Dans cette section, nous montrons comment améliorer le schéma de mise en gage décrit dans la section 8.5 en lui ajoutant la non-malléabilité : brièvement, il suffit simplement d'étendre la mise en gage ElGamal au chiffrement étiqueté Cramer-Shoup à l'aide de *one-time* signatures. Comme avant, si b est un bit, on note son complémentaire $\overline{b}$ (c'est-à-dire $\overline{b} = 1 - b$). On note en outre $x[i]$ le i-ième bit de la chaîne de bits x.

8.6.1 Non-malléabilité

La non-malléabilité est un souhait usuel pour des schémas de chiffrement ou des mises en gage [GL03]. Notre objectif est donc désormais de parvenir à réaliser cette propriété. Nous utilisons donc pour cela le schéma de chiffrement étiqueté de Cramer-Shoup à la place de ElGamal, et y ajoutons une *one-time* signature. En utilisant les résultats de Dodis et Katz [DK05] pour la sécurité à chiffrés choisis du chiffrement multiple, on peut facilement voir que la sécurité à chiffrés choisis (et donc la non-malléabilité) du schéma de chiffrement utilisé pour calculer un vecteur de chiffrés $(b_1, \ldots, b_m)$ provient de la sécurité à chiffrés choisis du schéma sous-jacent Cramer-Shoup étiqueté et de la propriété de *strong unforgeability* du schéma de *one-time* signature utilisé pour lier tous les chiffrés ensemble.

Plus précisément, si M est défini comme précédemment, et si ℓ est une étiquette, la mise en gage $\mathsf{comCS}^{\ell}_{\mathsf{pk}}(\mathrm{M})$ est obtenue comme suit. D'abord, l'utilisateur génère une paire de clefs $(\mathsf{VK}, \mathsf{SK})$ pour un schéma de *one-time* signature. Ensuite, il calcule les valeurs suivantes, avec $\ell' = \ell \circ \mathsf{VK}$:

$$\forall i \qquad b_i = \mathsf{CS}^{+\,\ell'}_{\mathsf{pk}}(\mathrm{M}_i \cdot 2^{i-1}, r_i) = (u_{1,i} = g_1^{r_i}, u_{2,i} = g_2^{r_i}, e_i = h^{r_i} g^{\mathrm{M}_i \cdot 2^{i-1}}, v_i = (cd^{\theta_i})^{r_i})$$

En définissant $\mathbf{b} = (b_1, \ldots, b_m)$, il calcule $\sigma = \mathsf{Sign}(\mathsf{SK}, \mathbf{b})$. La mise en gage finale est alors définie par $\mathsf{comCS}^{\ell'}_{\mathsf{pk}}(\mathrm{M}) = (\mathbf{b}, \mathsf{VK}, \sigma)$. On peut obtenir, pour tout b_i, un chiffrement ElGamal de $\mathrm{M}_i \cdot 2^{i-1}$: $\mathrm{B}_i = \mathsf{EG}^{+}_{\mathsf{pk}}(\mathrm{M}_i \cdot 2^{i-1}, r_i) = (u_{1,i} = g_1^{r_i}, e_i = h^{r_i} g^{\mathrm{M}_i \cdot 2^{i-1}})$. La propriété homomorphe du schéma de chiffrement permet d'obtenir

$$\mathrm{B} = \mathsf{EG}^{\times}_{\mathsf{pk}}(g^{\mathrm{M}}, \textstyle\sum r_i) = \mathsf{EG}^{+}_{\mathsf{pk}}(\mathrm{M}, \sum r_i) = (\prod u_{1,i}, \prod e_i) = \prod \mathrm{B}_i$$

Afin de définir la SPHF associée à ce schéma de mise en gage, nous rappelons la famille de SPHF pour le schéma de chiffrement Cramer-Shoup étiqueté sous-jacent, donnée dans [GL03]. Soient $\mathrm{X}' = \mathrm{G}^4$ et $\mathrm{L}' = \mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell,\mathrm{M})}$ le langage des éléments C tels que C est un chiffré Cramer-Shoup valide de M sous l'étiquette ℓ (aux est définie par (ℓ, M)). Sous l'hypothèse DDH, L' est un *hard partitioned subset* de X'. En notant $\mathrm{C} = \mathsf{CS}^{+\,\ell}_{\mathsf{pk}}(\mathrm{M}, r) = (u_1, u_2, e, v)$, le SHS associé est le suivant :

$$\begin{aligned}
\mathsf{HashKG}((\mathbf{CS}^+, \rho), (\ell, \mathrm{M})) &= \mathsf{hk} = (\gamma_1, \gamma_2, \gamma_3, \gamma_4) \xleftarrow{\$} \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \\
\mathsf{ProjKG}(\mathsf{hk}; (\mathbf{CS}^+, \rho), (\ell, \mathrm{M}), \mathrm{C}) &= \mathsf{hp} = (g_1)^{\gamma_1} (g_2)^{\gamma_2} (h)^{\gamma_3} (cd^{\theta})^{\gamma_4} \\
\mathsf{Hash}(\mathsf{hk}; (\mathbf{CS}^+, \rho), (\ell, \mathrm{M}), \mathrm{C}) &= (u_1)^{\gamma_1} (u_2)^{\gamma_2} (e g^{\mathrm{M}})^{\gamma_3} (v)^{\gamma_4} \\
\mathsf{ProjHash}(\mathsf{hp}; (\mathbf{CS}^+, \rho), (\ell, \mathrm{M}), \mathrm{C}; r) &= (\mathsf{hp})^r
\end{aligned}$$

À partir de ces définitions, nous considérons le langage

$$\mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell,0\vee1)} = \mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell,0)} \cup \mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell,1)}$$

dans lequel

$$\mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell,0)} = \{\mathrm{C} \mid \exists r\ \mathrm{C} = \mathsf{CS}^{+\,\ell}_{\mathsf{pk}}(0, r)\} \quad \text{et} \quad \mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell,1)} = \{\mathrm{C} \mid \exists r\ \mathrm{C} = \mathsf{CS}^{+\,\ell}_{\mathsf{pk}}(1, r)\}$$

et nous souhaitons définir un SHS montrant que

$$\forall i \quad b_i \in \mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell',0\vee1)} \qquad \text{et} \qquad \mathrm{B} = \prod \mathrm{B}_i \in \mathrm{L}_{(\mathbf{EG}^\times,\rho),g^{\mathrm{M}}}$$

Notons que, pour tout i, la première appartenance implique que $\mathrm{B}_i \in \mathrm{L}_{(\mathbf{EG}^+,\rho),0\vee1}$, puisque les chiffrés ElGamal B_i sont extraits des chiffrés Cramer-Shoup b_i correspondants : il est ainsi suffisant de vérifier cette validité pour obtenir l'extractibilité, ce qui signifie que l'on peut extraire la clef privée M mise en gage, associée à la clef publique g^{M}. La signature doit bien sûr aussi être vérifiée, mais cela peut être effectuée de manière publique, à partir de VK.

8.6.2 Équivocabilité

Nous décrivons ici entièrement notre mise en gage par complétude, mais notons qu'elle est très similaire à celle décrite dans la section 8.5.

Description de la mise en gage. Notre schéma utilise le chiffrement Cramer-Shoup étiqueté et une *one-time* signature, afin d'obtenir la non-malléabilité comme expliqué dans la section précédente. Le chiffrement Cramer-Shoup étiqueté [CS98], utilisé pour chaque bit de la chaîne de bits est vu ici comme une extension du chiffrement ElGamal homomorphe [ElG85].

Supposons que l'entrée de l'algorithme de mise en gage soit une chaîne de bits de la forme $\pi = \sum_{i=1}^{m} \pi_i \cdot 2^{i-1}$ et une étiquette ℓ. Cet algorithme fonctionne comme suit :

- Pour $i = 1, \ldots, m$, il choisit une valeur aléatoire $x_{i,\pi_i} = \sum_{j=1}^{n} x_{i,\pi_i}[j] \cdot 2^{j-1}$ et fixe $x_{i,\overline{\pi}_i} = 0$. Il génère aussi une paire de clefs $(\mathsf{VK}, \mathsf{SK})$ pour la *one-time* signature.
- Pour $i = 1, \ldots, m$, il met en gage π_i en utilisant l'aléa x_{i,π_i} :

$$a_i = \mathsf{comPed}(\pi_i, x_{i,\pi_i}) = g^{x_{i,\pi_i}} y_i^{\pi_i}$$

  Il définit ensuite $\mathbf{a} = (a_1, \ldots, a_m)$.

- Pour $i = 1, \ldots, m$, il calcule avec l'aide de $\ell_i = \ell \circ \mathsf{VK} \circ \mathbf{a} \circ i$ les mises en gage Cramer-Shoup de $x_{i,\delta}$, pour $\delta = 0, 1$ (voir la section précédente) :

  $$(\mathbf{b}_{i,\delta} = (b_{i,\delta}[j])_j, \mathsf{VK}, \sigma_{i,\delta}) = \mathsf{comCS}_{\mathsf{pk}}^{\ell_i}(x_{i,\delta}), \text{ où } b_{i,\delta}[j] = \mathsf{CS}_{\mathsf{pk}}^{+\,\ell_i}(x_{i,\delta}[j] \cdot 2^{j-1}, r_{i,\delta}[j])$$

  Le calcul des $b_{i,\delta}[j]$ définit implicitement $\mathrm{B}_{i,\delta}[j] = \mathsf{EG}_{\mathsf{pk}}^{+}(x_{i,\delta}[j] \cdot 2^{j-1}, r_{i,\delta}[j])$, depuis lequel on peut directement extraire un chiffré $\mathrm{B}_{i,\delta}$ de $x_{i,\delta}$: $\mathrm{B}_{i,\delta} = \prod_j \mathrm{B}_{i,\delta}[j] = \mathsf{EG}_{\mathsf{pk}}^{+}(x_{i,\delta}, r_{i,\delta})$, où $r_{i,\delta}$ est la somme des aléas $r_{i,\delta}[j]$.

Si n est la longueur en bits de l'ordre premier q du groupe G, la chaîne entière d'aléas pour cette mise en gage est

$$\mathrm{R} = (x_{1,\pi_1}, (r_{1,0}[1], r_{1,1}[1], \ldots, r_{1,0}[n], r_{1,1}[n]), \ldots, x_{m,\pi_m}, (r_{m,0}[1], \ldots, r_{m,1}[n]), \mathsf{SK})$$

À partir de là, toutes les valeurs $r_{i,\overline{\pi}_i}[j]$ peuvent être effacées, en laissant la valeur d'ouverture (témoin de la valeur mise en gage) devenir

$$\mathsf{w} = (x_{1,\pi_1}, (r_{1,\pi_1}[1], \ldots, r_{1,\pi_1}[n]), \ldots, x_{m,\pi_m}, (r_{m,\pi_m}[1], \ldots, r_{m,\pi_m}[n]))$$

La sortie de l'algorithme de mise en gage sur la chaîne de bits π, avec l'étiquette ℓ, et utilisant l'aléa R, est

$$\mathsf{com}_\rho(\ell, \pi; \mathrm{R}) = (\ell, \mathbf{a}, \mathbf{b}, \mathsf{VK}, \sigma)$$

où $\quad \mathbf{a} = (a_i = \mathsf{comPed}(\pi_i, x_{i,\pi_i}))_i$, $\mathbf{b} = (b_{i,\delta}[j] = \mathsf{CS}_{\mathsf{pk}}^{+\,\ell_i}(x_{i,\delta}[j] \cdot 2^{j-1}, r_{i,\delta}[j]))_{i,\delta,j}$

et $\quad \sigma = (\sigma_{i,\delta} = \mathsf{Sign}(\mathsf{SK}, (b_{i,\delta}[j])_j))_{i,\delta}$

Propriétés. La mise en gage est ouverte comme dans la section 8.5 et les preuves données dans la section 8.5.2 sont toujours valables.

En outre, montrons désormais que la vue de mises en gage équivocables n'aide pas l'adversaire à construire une mise en gage valide mais non-extractible (qui pourrait s'ouvrir de n'importe quelle façon, et dont l'extraction mènerait donc à plusieurs possibilités) grâce à la propriété CCA du chiffrement. Comme dans la section 8.5.2, nous utilisons un argument *Left-or-Right*, voir [BDJR97]. L'oracle *Left*, qui donne toujours au joueur un chiffrement $\mathcal{E}(g^0)$, est équivalent au jeu dans lequel aucune mise en gage n'est équivocable, et l'oracle *Right*, qui donne toujours $\mathcal{E}(g^x)$ sur une entrée x, est équivalent au jeu dans lequel les mises en gage sont équivocables. Ainsi, si l'adversaire avait plus de chance de construire une mise en gage valide mais non-extractible dans le dernier cas que dans le premier, on pourrait construire un distingueur aux oracles *Left-or-Right*. Cependant, contrairement à la preuve dans la section 8.5.2, un oracle de déchiffrement est requis pour vérifier si la mise en gage est valide mais non-extractible (tandis que dans la section 8.5.2 l'adversaire casse la propriété *binding* avec deux valeurs d'ouverture différentes). Par conséquent, produire des mises en gage valides mais non-extractibles n'est pas plus facile quand l'adversaire reçoit des mises en gage équivocables que lorsqu'il n'en reçoit pas, sous la sécurité IND-CCA du schéma de chiffrement. En outre, une mise en gage valide mais non-extractible avec un oracle de déchiffrement mène à deux valeurs d'ouvertures différentes pour la mise en gage, et donc à une attaque contre la propriété de *binding*, ce qui mène à une solution au problème du logarithme discret.

La non-malléabilité provient de la conjonction de la propriété IND-CCA et de la *one-time* signature, comme expliqué au début de cette section 8.6.

8.6.3 La SPHF associée

Comme expliqué ci-dessus, notre nouveau schéma de mise en gage est conditionnellement extractible (on peut obtenir les $x_{i,\delta}$ puis π), sous la condition que tous les chiffrés Cramer-Shoup chiffrent 0 ou 1, et que les a_i soient des mises en gage de 0 ou 1, avec l'aléa $x_{i,0}$ ou $x_{i,1}$.

Comme nous voulons l'appliquer ensuite dans le cadre de mots de passe, nous voulons faire en sorte que les deux valeurs hachées (calcul direct, et calcul depuis la clef projetée) soient les

mêmes si les deux joueurs utilisent le même mot de passe π, mais parfaitement indépendantes s'ils utilisent des mots de passe différents. En utilisant leur mot de passe $\pi = \sum_{i=1}^{m} \pi_i \cdot 2^{i-1}$, on veut de plus assurer que chaque a_i est en fait une mise en gage de Pedersen de π_i utilisant l'aléa chiffré x_{i,π_i}, et donc $g^{x_{i,\pi_i}} = a_i / y_i^{\pi_i}$: la valeur x_{i,π_i} extraite est réellement la clef privée M reliée à une clef publique donnée g^{M} qui est $a_i/y_i^{\pi_i}$ dans notre cas. En utilisant les mêmes notations que dans la section 8.6.1, nous souhaitons définir un SHS montrant que, pour tout i, δ, j, $b_{i,\delta}[j] \in \mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell_i,0\vee1)}$ et, pour tout i, $\mathrm{B}_{i,\pi_i} \in \mathrm{L}_{(\mathbf{EG}^\times,\rho),(a_i/y_i{}^{\pi_i})}$, où $\mathrm{B}_{i,\pi_i} = \prod_j \mathrm{B}_{i,\pi_i}[j]$. Comme avant, notons que, pour tout i, δ, j, la première appartenance implique aussi que $\mathrm{B}_{i,\delta}[j] \in \mathrm{L}_{(\mathbf{EG}^+,\rho),0\vee1}$ puisque cette valeur est extraite de la valeur correspondante $b_{i,\delta}[j]$: il est ainsi suffisant de vérifier la validité de cette dernière.

Combinaisons de ces SHS. Soit C la mise en gage ci-dessus de π utilisant l'étiquette ℓ et l'aléa R comme définie dans la section 8.6.2. Précisons désormais le langage $\mathrm{L}_{\rho,(\ell,\pi)}$, consistant informellement en toutes les mises en gage valides « de la bonne forme » :

$$\mathrm{L}_{\rho,(\ell,\pi)} = \left\{ \mathrm{C} \;\middle|\; \begin{array}{l} \exists \mathrm{R} \text{ tel que } \mathrm{C} = \mathsf{com}_\rho(\ell, \pi, \mathrm{R}) \\ \quad \text{et } \forall i\ \forall \delta\ \forall j \quad b_{i,\delta}[j] \in \mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell_i,0\vee1)} \\ \quad \text{et } \forall i \quad \mathrm{B}_{i,\pi_i} \in \mathrm{L}_{(\mathbf{EG}^\times,\rho),a_i/y_i{}^{\pi_i}} \end{array} \right\}$$

Le SHS pour ce langage repose sur les SHS décrits précédemment, en utilisant la construction générique pour les conjonctions et disjonctions de la section 8.3. Plus précisément, en ajoutant les valeurs $\mathrm{B}_{i,\delta}$ facilement calculées à partir des valeurs $b_{i,\delta}[j]$, la mise en gage peut être convertie en une liste de la forme suivante (on omet la partie correspondant à la signature, puisqu'elle doit être vérifiée avant de calculer un hachage quelconque) :

$$\begin{array}{ll} (\ell, & a_1, \ldots, a_m, \quad b_{1,0}[1], b_{1,1}[1], \ldots, b_{1,0}[n], b_{1,1}[n], \ldots, b_{m,0}[1], b_{m,1}[1], \ldots, b_{m,0}[n], b_{m,1}[n], \\ & \mathrm{B}_{1,0}, \mathrm{B}_{1,1}, \ldots, \mathrm{B}_{m,0}, \mathrm{B}_{m,1}) \in \{0,1\}^* \times \mathrm{G}^m \times (\mathrm{G}^4)^{2mn} \times (\mathrm{G}^2)^{2m} \end{array}$$

Notons $\mathrm{L}^{i,\delta,j}_{(\mathbf{CS}^+,\rho),(\ell_i,0\vee1)}$ le langage qui restreint la valeur $b_{i,\delta}[j]$ à un chiffré Cramer-Shoup valide de 0 ou 1 :

$$\underbrace{\{0,1\}^*}_{\ell} \times \underbrace{\mathrm{G}^m}_{\mathbf{a}} \times \underbrace{(\mathrm{G}^4)^{2n}}_{b_{1,*}[*]} \times \ldots \times (\underbrace{\mathrm{G}^4}_{b_{i,0}[1]} \times \underbrace{\mathrm{G}^4}_{b_{i,1}[1]} \times \ldots \times \underbrace{\mathrm{L}_{(\mathbf{CS}^+,\rho),(\ell_i,0\vee1)}}_{b_{i,\delta}[j]} \times \ldots \times \underbrace{\mathrm{G}^4}_{b_{i,0}[n]} \times \underbrace{\mathrm{G}^4}_{b_{i,1}[n]})$$
$$\times \ldots \times \underbrace{(\mathrm{G}^4)^{2n}}_{b_{1,m}[*]} \times \underbrace{(\mathrm{G}^2)^{2m}}_{\mathrm{B}_{*,*}}$$

et $\mathrm{L}^i_{(\mathbf{EG}^\times,\rho)}$ le langage qui restreint les chiffrés ElGamal B_{i,π_i} :
si $\pi_i = 0$,

$$\mathrm{L}^i_{(\mathbf{EG}^\times,\rho)} = \underbrace{\{0,1\}^*}_{\ell} \times \underbrace{\mathrm{G}^m}_{\mathbf{a}} \times \underbrace{(\mathrm{G}^4)^{2mn}}_{b_{*,*}[*]} \times \underbrace{(\mathrm{G}^2)^2}_{\mathrm{B}_{1,*}} \times \ldots \times (\underbrace{\mathrm{L}_{(\mathbf{EG}^\times,\rho),a_i}}_{\mathrm{B}_{i,0}} \times \underbrace{\mathrm{G}^2}_{\mathrm{B}_{i,1}}) \times \ldots \times \underbrace{(\mathrm{G}^2)^2}_{\mathrm{B}_{m,*}}$$

si $\pi_i = 1$,

$$\mathrm{L}^i_{(\mathbf{EG}^\times,\rho)} = \underbrace{\{0,1\}^*}_{\ell} \times \underbrace{\mathrm{G}^m}_{\mathbf{a}} \times \underbrace{(\mathrm{G}^4)^{2mn}}_{b_{*,*}[*]} \times \underbrace{(\mathrm{G}^2)^2}_{\mathrm{B}_{1,*}} \times \ldots \times (\underbrace{\mathrm{G}^2}_{\mathrm{B}_{i,0}} \times \underbrace{\mathrm{L}_{(\mathbf{EG}^\times,\rho),a_i/y_i}}_{\mathrm{B}_{i,1}}) \times \ldots \times \underbrace{(\mathrm{G}^2)^2}_{\mathrm{B}_{m,*}}$$

Le langage $\mathrm{L}_{\rho,(\ell,\pi)}$ est alors la conjonction de tous ces langages, où les $\mathrm{L}^{i,\delta,j}_{(\mathbf{CS}^+,\rho),(\ell_i,0\vee1)}$ sont des disjonctions :

$$\mathrm{L}_{\rho,(\ell,\pi)} = \left(\bigcap_{i,\delta,j} \mathrm{L}^{i,\delta,j}_{(\mathbf{CS}^+,\rho),(\ell_i,0\vee1)} \right) \cap \left(\bigcap_i \mathrm{L}^i_{(\mathbf{EG}^\times,\rho)} \right)$$

Propriétés : uniformité et indépendance. Avec une telle mise en gage et une SPHF associée, il est possible d'améliorer l'établissement d'un canal sécurisé entre deux joueurs, à partir

de celui présenté dans la section 8.4.3. Plus précisément, deux joueurs peuvent s'accorder sur une clef commune s'ils partagent un mot de passe π commun (de faible entropie). Cependant, un protocole plus évolué que celui proposé dans la section 8.4.3 est nécessaire pour obtenir toutes les propriétés requises pour un protocole d'échange de clefs basé sur des mots de passe, comme cela sera expliqué et prouvé dans le chapitre suivant.

Malgré tout, il semble y avoir une fuite d'information à cause du langage qui dépend du mot de passe π : la clef projetée hp semble contenir de l'information sur π, qui peut être utilisée dans une autre exécution par un adversaire. D'où les notions d'indépendance et d'uniformité présentées dans la section 8.3.5 page 124, qui assurent que hp ne contient aucune information sur π :

- pour les langages $\mathrm{L}^{i}_{(\mathbf{EG}^{\times},\rho)}$, la SPHF satisfait la propriété de 2-indépendance, puisque la clef projetée pour un langage $\mathrm{L}_{(\mathbf{EG}^{\times},\rho),\mathrm{M}}$ dépend de la clef publique (et donc de ρ) uniquement. On génère donc une seule clef pour chaque paire $(\mathrm{B}_{i,0},\mathrm{B}_{i,1})$, et on utilise le chiffré correct par rapport au π_i désiré quand on évalue la valeur hachée. Mais cette distinction sur π_i apparaît uniquement dans le calcul de la valeur hachée, qui est pseudo-aléatoire. La clef projetée est totalement indépendante de π_i.
- pour les langages $\mathrm{L}^{i,\delta,j}_{(\mathbf{CS}^{+},\rho),(\ell_i,0\vee1)}$, la SPHF satisfait la propriété de 2-uniformité uniquement, mais pas la 2-indépendance. Ainsi, la clef projetée et (aux,ρ) sont statistiquement indépendantes, mais la première dépend, dans son calcul, de la deuxième. Si nous voulons obtenir la propriété d'équivocabilité pour la mise en gage (comme cela sera nécessaire dans le chapitre suivant), nous devons inclure toutes les paires $(b_{i,0}[j],b_{i,1}[j])$, et pas seulement les $b_{i,\pi_i}[j]$, afin que nous puissions ouvrir plus tard de n'importe quelle façon. En raison de la 2-uniformité et non 2-indépendance, nous avons besoin d'une clef pour chaque élément de la paire, et pas seulement une comme précédemment.

Estimation de la complexité. Globalement, chaque opération (mise en gage, clef projetée, hachage et hachage projeté) requiert $\mathcal{O}(mn)$ exponentiations dans G, avec des petites constantes (maximum 16).

Considérons d'abord l'opération de mise en gage, sur un mot de passe secret π de m bits, sur un groupe G de taille n bits. On doit d'abord faire m mises en gage de Pedersen d'un bit (m exponentiations) et ensuite $2mn$ chiffrés additifs de Cramer-Shoup (2 exponentiations et 2 multi-exponentiations chacun, et donc approximativement le coût de $8mn$ exponentiations).

Pour la SPHF, la génération de la clef de hachage consiste uniquement à générer des éléments aléatoires dans $\mathbb{Z}_q$: 8 pour chaque chiffré Cramer-Shoup, afin de montrer qu'ils chiffrent 0 ou 1, et 2 pour chaque paire de chiffrés ElGamal, donc globalement $2m(8n+1)$ éléments aléatoires dans $\mathbb{Z}_q$. Les clefs projetées nécessitent des exponentiations : $m(4n+1)$ multi-exponentiations, et $4mn$ évaluations de hachage (une multi-exponentiation chacune). Ainsi, globalement, le coût de $m(8n+1)$ exponentiations dans G est requis pour la clef projetée. Finalement, les calculs de hachage sont essentiellement les mêmes en utilisant la clef de hachage ou la clef projetée, puisque pour les sous-fonctions, la première consiste en une multi-exponentiation et la deuxième en une exponentiation. Elles coûtent toutes les deux $m(4n+1)$ exponentiations, après les multiplications nécessaires pour calculer les B_{i,π_i}, qui sont négligeables.

Chapitre 9

Protocole d'échange de clefs pour deux joueurs

La primitive décrite dans la section 8.6 page 131, qui utilise le schéma de chiffrement de Cramer-Shoup, est une mise en gage non-malléable conditionnellement extractible et équivocable. On explique dans ce chapitre comment utiliser cette nouvelle primitive de façon à construire le premier protocole efficace d'échange de clefs à deux joueurs basés sur un mot de passe qui soit sûr contre les attaques adaptatives dans le cadre UC avec effacements, dans le modèle standard. Nous présentons ce protocole dans la section 9.1. Dans la preuve, les mots de passe seront inconnus au début de la simulation : $\mathcal{S}$ va parvenir à rétablir a posteriori le bon (grâce à l'équivocabilité), mais sans effacements il resterait des indices sur la façon dont les calculs ont été menés, ce qui donnerait des indications sur les mots de passe utilisés à un attaquant (qui récupérerait tout l'état interne passé lors d'une corruption).

Notre protocole est basé sur KOY/GL [KOY01, GL03] et son extension UC par Canetti *et al.* [CHK+05] décrits dans les chapitre 3 (page 45) et 5 (page 83). Malheureusement, la modification qu'ils ont effectuée dans ce dernier article ne semble pas fonctionner pour gérer des adversaires adaptatifs, ce qui est le cas qui nous intéresse. Comme expliqué précédemment, c'est parce que le simulateur ne peut pas ouvrir correctement la mise en gage quand l'adversaire corrompt le client après que le pré-message a été envoyé. Une remarque similaire s'applique dans le cas où le serveur se fait corrompre après avoir envoyé son premier message. En conséquence, en plus d'être extractible, le schéma de mise en gage doit aussi être équivocable afin que le simulateur puisse procurer une vue cohérente à l'adversaire. Comme l'utilisation d'un tel schéma de mise en gage paraît aussi permettre de prouver directement le protocole original de Gennaro-Lindell dans le cadre UC (sans utiliser la modification de [CHK+05]), nous avons opté pour ce protocole comme point de départ pour le nôtre.

Ces remarques sont en fait suffisantes (avec quelques détails techniques) pour obtenir la sécurité adaptative. Ainsi, notre solution consiste essentiellement en l'utilisation de notre schéma de mise en gage non-malléable extractible et équivocable dans les deux premiers messages du protocole Gennaro-Lindell. Comme expliqué dans le chapitre précédent, l'extractibilité peut être conditionnelle : on inclut donc cette condition dans le langage de la *smooth projective hash function* (notons que les clefs projetées envoyées ne laissent fuir aucune information à propos du mot de passe).

9.1 Description du protocole

Comme décrit dans le chapitre 5 page 83, Canetti *et al.* [CHK+05] ont proposé une variante simple de la méthode de Gennaro-Lindell [GL03] pouvant être prouvée sûre dans le cadre UC. Bien qu'assez efficace, leur protocole n'est apparemment pas sûr contre les attaques adaptatives. Le seul PAKE sûr contre de telles attaques dans le cadre UC sous des hypothèses standards est celui proposé par Barak *et al.* [BCL+05] et il utilise des techniques générales de *multiparty computation*. Il mène ainsi à des schémas assez inefficaces.

La description complète du protocole se trouve sur la figure 9.2, tandis qu'un schéma informel est présenté sur la figure 9.1. Nous utilisons l'indice I pour une valeur reliée au client (Alice) et J pour une valeur reliée au serveur (Bob).

Correction. Dans une exécution honnête du protocole, si les joueurs partagent le même mot de passe ($\mathsf{pw}_\mathrm{I} = \mathsf{pw}_\mathrm{J} = \mathsf{pw}$), il est facile de vérifier que les deux joueurs vont terminer en acceptant et en calculant la même valeur pour la clef de session, égale à $\mathsf{Hash}(\mathsf{hk}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}), \mathsf{com}_\mathrm{J}) + \mathsf{Hash}(\mathsf{hk}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}), \mathsf{com}_\mathrm{I})$ (les notations sont celles du chapitre précédent).

Securité. L'intuition derrière la sécurité de notre protocole est assez simple et basée sur celle du protocole de Gennaro-Lindell. Le point-clef permettant de garantir la sécurité adaptative est l'utilisation de mises en gage, qui autorisent l'extraction et l'équivocation à n'importe quel moment, ne demandant donc pas au simulateur d'avoir connaissance de corruptions futures. Le théorème 3, dont la preuve complète sera donnée dans la section suivante, statue que le protocole est sûr dans le cadre UC. La fonctionnalité idéale $\mathcal{F}_{pw\mathrm{KE}}$ est celle de Canetti *et al.*, déjà présentée au chapitre 5 page 81. Comme nous utilisons la version à états joints du théorème UC, nous considérons implicitement l'extension multi-session de cette fonctionnalité. En particulier, notons que les mots de passe dépendent de la session considérée. Par simplicité, nous les notons pw_I et pw_J, mais on doit implicitement lire $\mathsf{pw}_{\mathrm{I},\mathsf{ssid}}$ et $\mathsf{pw}_{\mathrm{J},\mathsf{ssid}}$.

Théorème 3. *Soient* com *le schéma de mise en gage non-malléable (conditionnellement) extractible et équivocable décrit dans la section 8.6,* $\mathcal{H}$ *une* SPHF *par rapport à cette mise en gage, et* $\mathcal{SIG}$ *un schéma de* one-time *signature. Notons* $\widehat{\mathcal{F}}_{pw\mathrm{KE}}$ *l'extension multi-session de la fonctionnalité* $\mathcal{F}_{pw\mathrm{KE}}$ *d'échange de clefs basé sur un mot de passe (voir figure 5.8.1 page 81), et soit* $\mathcal{F}_{\mathrm{CRS}}$ *la fonctionnalité idéale qui procure une CRS* $(\mathrm{G}, \mathsf{pk}, (y_1, \ldots, y_m))$ *aux deux joueurs, où* G *est un groupe cyclique,* $y_1, \ldots, y_m$ *des éléments aléatoires de ce groupe, et* pk *une clef publique pour le schéma Cramer-Shoup. Le protocole décrit dans les figures 9.1 et 9.2 réalise alors de manière sûre* $\widehat{\mathcal{F}}_{pw\mathrm{KE}}$ *dans le modèle* $\mathcal{F}_{\mathrm{CRS}}$*-hybride, en présence d'adversaires adaptatifs.*

9.2 Preuve du théorème 3

9.2.1 Idée de la preuve

Afin de prouver le théorème 3, nous devons construire, pour tout adversaire réel $\mathcal{A}$ (interagissant avec des joueurs réels exécutant le protocole), un adversaire idéal $\mathcal{S}$ (interagissant avec des joueurs virtuels et la fonctionnalité $\mathcal{F}_{pw\mathrm{KE}}$) tel qu'aucun environnement $\mathcal{Z}$ ne puisse distinguer une exécution avec $\mathcal{A}$ dans le monde réel d'une exécution avec $\mathcal{S}$ dans le monde idéal avec probabilité non-négligeable.

Comme dans le chapitre 6 page 93, nous utilisons les requêtes hybrides `GoodPwd` et `SamePwd`, ainsi que les terminologies de messages *oracle-generated* et *non oracle-generated*. On appellera *attaqué* un joueur corrompu ou contrôlé par l'adversaire.

On définit dans cette preuve une suite de jeux depuis l'exécution réelle du protocole $\mathbf{G}_0$ jusqu'au jeu $\mathbf{G}_8$, que l'on prouve être indistinguable du jeu idéal :

- $\mathbf{G}_0$ est le jeu réel.
- À partir de $\mathbf{G}_1$, $\mathcal{S}$ peut programmer la trappe d'extractabilité de la CRS.
- À partir de $\mathbf{G}_2$, $\mathcal{S}$ peut toujours extraire le mot de passe engagé par l'adversaire et abandonne quand l'extraction échoue si la mise en gage est valide pour deux mots de passe ou plus. Cela mènerait à une attaque contre la propriété de *binding* du schéma de mise en gage.

Fig. 9.1 – Description du protocole pour les joueurs $(\mathbf{P_I}, \mathbf{ssid})$, avec indice $\mathbf{I}$ et mot de passe $\boldsymbol{pw_I}$ et $(\mathbf{P_J}, \mathbf{ssid})$, avec indice $\mathbf{J}$ et mot de passe $\boldsymbol{pw_J}$.

Client U		Serveur S
(U1) $(\mathsf{VK}_\mathrm{I}, \mathsf{SK}_\mathrm{I}) \leftarrow \mathsf{SKG}$ $\ell_\mathrm{I} = \mathrm{J} \circ \mathrm{I} \circ \mathsf{ssid} \circ \mathsf{VK}_\mathrm{I}$ $\mathsf{com}_\mathrm{I} = \mathsf{com}_\rho(\ell_\mathrm{I}, \mathsf{pw}_\mathrm{I}; \mathrm{R}_\mathrm{I})$	$\xrightarrow{(\texttt{flow-one}, \mathsf{com}_\mathrm{I}, \mathsf{VK}_\mathrm{I})}$	
		(S2) vérifie (publiquement) la validité de com_I $(\mathsf{VK}_\mathrm{J}, \mathsf{SK}_\mathrm{J}) \leftarrow \mathsf{SKG}$ $\ell_\mathrm{J} = \mathrm{J} \circ \mathrm{I} \circ \mathsf{ssid} \circ \mathsf{VK}_\mathrm{J}$ $\mathsf{hk}_\mathrm{J} = \mathsf{HashKG}(\rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{J}), r_\mathrm{J})$ $\mathsf{com}_\mathrm{J} = \mathsf{com}_\rho(\ell_\mathrm{J}, \mathsf{pw}_\mathrm{J}; \mathrm{R}_\mathrm{J})$ $\mathsf{hp}_\mathrm{J} = \mathsf{ProjKG}(\mathsf{hk}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{I})$ $\mathtt{Hash}_\mathrm{J} = \mathsf{Hash}(\mathsf{hk}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{I})$ efface hk_J
	$\xleftarrow{(\texttt{flow-two}, \mathsf{com}_\mathrm{J}, \mathsf{VK}_\mathrm{J}, \mathsf{hp}_\mathrm{J})}$	
(U3) vérifie (publiquement) la validité de com_J $\mathsf{hk}_\mathrm{I} = \mathsf{HashKG}(\rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{I}), r_\mathrm{I})$ $\mathsf{hp}_\mathrm{I} = \mathsf{ProjKG}(\mathsf{hk}_\mathrm{I}; (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{J})$ $\sigma_\mathrm{I} = \mathsf{Sign}(\mathsf{SK}_\mathrm{I}, (\mathsf{com}_\mathrm{I}, \mathsf{com}_\mathrm{J}, \mathsf{hp}_\mathrm{I}, \mathsf{hp}_\mathrm{J}))$ $\mathsf{sk}_\mathrm{I} = \mathsf{ProjHash}(\mathsf{hp}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{I}; \mathsf{w}_\mathrm{I})$ $+ \mathsf{Hash}(\mathsf{hk}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{J})$ efface hk_I		
	$\xrightarrow{(\texttt{flow-three}, \sigma_\mathrm{I}, \mathsf{hp}_\mathrm{I})}$	
		(S4) abandonne si $\mathsf{Ver}(\mathsf{VK}_\mathrm{I}, (\mathsf{com}_\mathrm{I}, \mathsf{com}_\mathrm{J}, \mathsf{hp}_\mathrm{I}, \mathsf{hp}_\mathrm{J}), \sigma_\mathrm{I}) = 0$ $\sigma_\mathrm{J} = \mathsf{Sign}(\mathsf{SK}_\mathrm{J}, (\mathsf{com}_\mathrm{I}, \mathsf{com}_\mathrm{J}, \mathsf{hp}_\mathrm{I}, \mathsf{hp}_\mathrm{J}))$ $sk_\mathrm{J} = \mathsf{ProjHash}(\mathsf{hp}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{J}; \mathsf{w}_\mathrm{J})$ $+ \mathtt{Hash}_\mathrm{J}$ renvoie $(\mathsf{sid}, \mathsf{ssid}, sk_\mathrm{J})$ efface tout marque la session `accepted`
	$\xleftarrow{(\texttt{flow-four}, \sigma_\mathrm{J})}$	
(U5) abandonne si $\mathsf{Ver}(\mathsf{VK}_\mathrm{J}, (\mathsf{com}_\mathrm{I}, \mathsf{com}_\mathrm{J}, \mathsf{hp}_\mathrm{I}, \mathsf{hp}_\mathrm{J}), \sigma_\mathrm{J}) = 0$ renvoie $(\mathsf{sid}, \mathsf{ssid}, sk_\mathrm{I})$ efface tout marque la session `accepted`		

À la fin des tours (U1) et (S2), les joueurs effacent la partie non nécessaire dans la suite de leurs aléas R_I et R_J utilisés dans les mises en gage, conservant uniquement w_I et w_J.

- À partir de $\mathbf{G}_3$, $\mathcal{S}$ simule toutes les mises en gage et les rend équivocables grâce à la CRS simulée (totalement). Elles demeurent *binding* et *hiding* pour l'environnement et l'adversaire, d'après la propriété CCA2 du chiffrement. Notons aussi que la connaissance des mots de passe n'est plus nécessaire pour la simulation de l'étape de mise en gage.
- À partir de $\mathbf{G}_4$, $\mathcal{S}$ simule le client honnête sans plus utiliser son mot de passe (sauf pour les requêtes hybrides). Alice reçoit une clef aléatoire dans (U3) dans le cas où `flow-two` a été *oracle-generated* et que le serveur n'était pas corrompu. Si le serveur était corrompu, $\mathcal{S}$ récupère son mot de passe pw_J et fait une requête `GoodPwd` à la fonctionnalité pour Alice. Si elle est incorrecte, Alice reçoit aussi une clef aléatoire, mais si elle est correcte, la clef est calculée honnêtement en utilisant ce mot de passe. Si aucun mot de passe n'est récupéré, Alice reçoit aussi une clef aléatoire. Elle abandonne alors en (U5) si la signature du serveur est invalide. Si le serveur était corrompu avant (U5), $\mathcal{S}$ récupère son mot de passe et fonctionne exactement comme décrit précédemment. Ceci est indistinguable du jeu précédent grâce à la propriété pseudo-aléatoire de la fonction de hachage.
- À partir de $\mathbf{G}_5$, dans le cas où `flow-two` n'a pas été *oracle-generated*, $\mathcal{S}$ extrait pw_J depuis la valeur com_J et procède comme décrit dans $\mathbf{G}_4$: il pose une requête `GoodPwd` pour Alice et lui renvoie soit une valeur aléatoire soit une valeur calculée honnêtement pour sk_I. De même, si aucun mot de passe n'est récupéré, Alice reçoit une clef aléatoire. Elle abandonne à l'étape (U5) si la signature du serveur est invalide. Une corruption du serveur avant (U5) est gérée comme dans $\mathbf{G}_4$. Ce jeu est indistinguable du précédent grâce à la propriété de *smoothness* de la fonction de hachage.
- À partir de $\mathbf{G}_6$, $\mathcal{S}$ simule le serveur sans plus utiliser son mot de passe. Il abandonne si la signature reçue de la part du client est invalide. Sinon, le serveur Bob reçoit une clef aléatoire en (S4) dans le cas où `flow-one` était *oracle-generated* et le client non corrompu. Une corruption du client avant (S4) est gérée comme dans $\mathbf{G}_4$: $\mathcal{S}$ pose une requête `GoodPwd` pour Bob et lui donne soit une valeur aléatoire soit une valeur calculée honnêtement pour sk_J (si aucun mot de passe n'est récupéré, Bob reçoit une clef aléatoire). Ce jeu est indistinguable du précédent grâce au caractère pseudo-aléatoire de la fonction de hachage.
- À partir de $\mathbf{G}_7$, dans le cas où `flow-one` n'a pas été *oracle-generated*, $\mathcal{S}$ extrait pw_I depuis la valeur com_I et procède comme décrit dans $\mathbf{G}_4$: il pose une requête `GoodPwd` pour Bob et lui donne soit une valeur aléatoire soit une valeur calculée honnêtement pour sk_J (si aucun mot de passe n'est récupéré, Bob reçoit une clef aléatoire). Ce jeu est indistinguable du précédent grâce à la *smoothness* de la fonction de hachage.
- Finalement, les requêtes hybrides sont remplacées par les vraies dans $\mathbf{G}_8$, qu'on montre être indistinguable du jeu idéal.

9.2.2 Description du simulateur

La description du simulateur est basée sur celle des chapitres précédents. Une fois initialisé avec le paramètre de sécurité k, le simulateur commence par exécuter l'algorithme de génération de clefs du schéma de chiffrement $\mathcal{E}$, obtenant ainsi la paire $(\mathsf{sk}, \mathsf{pk})$. Il choisit aussi au hasard m éléments $(y_1, \ldots, y_m)$ dans G. Il initialise alors l'adversaire réel $\mathcal{A}$, en lui donnant $(\mathrm{G}, \mathsf{pk}, (y_1, \ldots, y_m))$ comme CRS.

À partir de ce moment-là, le simulateur interagit avec l'environnement $\mathcal{Z}$, la fonctionnalité $\mathcal{F}_{pw\mathrm{KE}}$ et sa sous-routine $\mathcal{A}$. Pour la plus grande partie, le simulateur $\mathcal{S}$ se contente de suivre le protocole au nom de tous les joueurs honnêtes. La différence principale entre les joueurs simulés et les vrais joueurs honnêtes est que $\mathcal{S}$ ne s'engage pas sur un mot de passe particulier pour ces derniers. Cependant, si $\mathcal{A}$ modifie un message `flow-one` ou `flow-two` délivré au joueur P dans la session ssid, alors $\mathcal{S}$ déchiffre cet élément (en utilisant sk) et utilise le message récupéré pw dans une requête `TestPwd` à la fonctionnalité. Si le mot de passe est correct, alors $\mathcal{S}$ l'utilise au nom de P, et continue la simulation. Plus de détails suivent.

Fig. 9.2 – Description du protocole pour les deux joueurs Alice et Bob. Alice est le client $\mathbf{P_I}$, avec indice I et mot de passe $\mathbf{pw_I}$, et Bob est le serveur $\mathbf{P_J}$, avec indice J et mot de passe $\mathbf{pw_J}$.

Common reference string. Un $m+2$-uplet $(\mathrm{G}, \mathsf{pk}, (y_1, \ldots, y_m))$, où G est un groupe cyclique, $y_1, \ldots, y_m$ des éléments aléatoires de ce groupe, et pk une clef publique pour le schéma Cramer-Shoup.

Étapes du protocole.

1. Quand $\mathrm{P_I}$ est activé avec l'entrée $(\texttt{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathrm{I}, \mathrm{J}, \mathsf{pw_I}, \texttt{role})$, il y a deux cas : si $\texttt{role} = \texttt{server}$, il ne fait rien. Si $\texttt{role} = \texttt{client}$, il utilise SKG pour générer une paire $(\mathsf{VK_I}, \mathsf{SK_I})$ pour un schéma de *one-time* signature, fixe le label (public) $\ell_\mathrm{I} = \mathrm{J} \circ \mathrm{I} \circ \mathsf{ssid} \circ \mathsf{VK_I}$, calcule $\mathsf{com_I} = \mathsf{com}_\rho(\ell_\mathrm{I}, \mathsf{pw_I}; \mathrm{R_I})$ et envoie le message $(\texttt{flow-one}, \mathsf{com_I}, \mathsf{VK_I})$ à $\mathrm{P_J}$. À partir de là, on suppose que $\mathrm{P_I}$ est un joueur activé avec l'entrée $(\texttt{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathrm{I}, \mathrm{J}, \mathsf{pw_I}, \texttt{client})$ et que $\mathrm{P_J}$ est un joueur activé avec l'entrée $(\texttt{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathrm{I}, \mathrm{J}, \mathsf{pw_J}, \texttt{server})$. Rappelons que $\mathrm{P_I}$ efface quasiment tous les aléas (dans $\mathrm{R_I}$) utilisés dans le calcul de $\mathsf{com_I}$ (voir la section 8.6). Plus précisément, il conserve uniquement dans $\mathrm{R_I}$ les valeurs présentes dans le témoin $\mathsf{w_I}$ qui seront utilisées dans le calcul de la fonction de hachage.

2. Quand $\mathrm{P_J}$ reçoit un message $(\texttt{flow-one}, \mathsf{com_I}, \mathsf{VK_I})$, il vérifie (publiquement) que $\mathsf{com_I}$ est bien construit, sinon il abandonne. Il utilise alors SKG pour générer une paire de clefs $(\mathsf{VK_J}, \mathsf{SK_J})$ pour un schéma de *one-time* signature, et HashKG pour générer une clef $\mathsf{hk_J}$ pour la SPHF $\mathcal{H}$ par rapport à ρ. Il fixe le label (public) $\ell_\mathrm{J} = \mathrm{J} \circ \mathrm{I} \circ \mathsf{ssid} \circ \mathsf{VK_J}$ et calcule la projection $\mathsf{hp_J} = \mathsf{ProjKG}(\mathsf{hk_J}; \rho, (\ell_\mathrm{I}, \mathsf{pw_J}), \mathsf{com_I})$. Il calcule alors $\mathsf{com_J} = \mathsf{com}_\rho(\ell_\mathrm{J}, \mathsf{pw_J}; \mathrm{R_J})$ et envoie le message $(\texttt{flow-two}, \mathsf{com_J}, \mathsf{VK_J}, \mathsf{hp_J})$ à $\mathrm{P_I}$. Il calcule finalement $\mathsf{Hash_J} = \mathsf{Hash}(\mathsf{hk_J}; \rho, (\ell_\mathrm{I}, \mathsf{pw_J}), \mathsf{com_I})$ et efface $\mathsf{hk_J}$ et les valeurs de $\mathrm{R_J}$ qui ne sont pas présentes dans le témoin $\mathsf{w_J}$.

3. Quand $\mathrm{P_I}$ reçoit un message $(\texttt{flow-two}, \mathsf{com_J}, \mathsf{VK_J}, \mathsf{hp_J})$, il vérifie (publiquement) que $\mathsf{com_J}$ est bien construit, sinon il abandonne. Il utilise alors HashKG pour générer une clef $\mathsf{hk_I}$ pour la SPHF $\mathcal{H}$ par rapport à pk et calcule la projection $\mathsf{hp_I} = \mathsf{ProjKG}(\mathsf{hk_I}; \rho, (\ell_\mathrm{J}, \mathsf{pw_I}), \mathsf{com_J})$. Il calcule alors $\sigma_\mathrm{I} = \mathsf{Sign}(\mathsf{SK_I}, (\mathsf{com_I}, \mathsf{com_J}, \mathsf{hp_I}, \mathsf{hp_J}))$ et envoie le message $(\texttt{flow-three}, \sigma_\mathrm{I}, \mathsf{hp_I})$ à $\mathrm{P_J}$. Il calcule la clef de session $sk_\mathrm{I} = \mathsf{ProjHash}(\mathsf{hp_J}; \rho, (\ell_\mathrm{I}, \mathsf{pw_I}), \mathsf{com_I}; \mathsf{w_I}) + \mathsf{Hash}(\mathsf{hk_I}; \rho, (\ell_\mathrm{J}, \mathsf{pw_I}), \mathsf{com_J})$, et efface toutes les données privées sauf $\mathsf{pw_I}$ et $\mathsf{sk_I}$, conservant les données publiques en mémoire.

4. Quand $\mathrm{P_J}$ reçoit un message $(\texttt{flow-three}, \sigma_\mathrm{I}, \mathsf{hp_I})$, il commence par vérifier que $\mathsf{Ver}(\mathsf{VK_I}, (\mathsf{com_I}, \mathsf{com_J}, \mathsf{hp_J}), \sigma_\mathrm{I}) = 1$. Si ce n'est pas le cas, il abandonne la session en ne renvoyant rien. Sinon, il calcule $\sigma_\mathrm{J} = \mathsf{Sign}(\mathsf{SK_J}, (\mathsf{com_I}, \mathsf{com_J}, \mathsf{hp_I}, \mathsf{hp_J}))$ et envoie le message $(\texttt{flow-four}, \sigma_\mathrm{J})$ to $\mathrm{P_I}$. Il calcule alors la clef de session $\mathsf{sk_J} = \mathsf{Hash_J} + \mathsf{ProjHash_I}(\mathsf{hp_I}; \rho, (\ell_\mathrm{J}, \mathsf{pw_J}), \mathsf{com_J}; \mathsf{w_J})$, et note la session $\texttt{accepted}$, ce qui signifie en particulier qu'il efface tout sauf $\mathsf{pw_J}$ et $\mathsf{sk_J}$ et termine l'exécution.

5. Quand $\mathrm{P_I}$ reçoit un message $(\texttt{flow-four}, \sigma_\mathrm{J})$, il commence par vérifier l'égalité $\mathsf{Ver}(\mathsf{VK_J}, (\mathsf{com_I}, \mathsf{com_J}, \mathsf{hp_I}, \mathsf{hp_J}), \sigma_\mathrm{J}) = 1$. Si ce n'est pas le cas, il abandonne la session en ne renvoyant rien. Sinon, il note la session $\texttt{accepted}$ et la termine.

Corruptions. Nous considérons des attaques adaptatives, qui peuvent intervenir à n'importe quel moment durant l'exécution du protocole. Étant donné le mot de passe d'un joueur, le simulateur doit donc être capable de donner à l'adversaire un état interne coïncidant avec toutes les données déjà envoyées (sans la connaissance du mot de passe du joueur). Pour

gérer de telles corruptions, le point-clef est la propriété équivocable de la mise en gage. Plus précisément, au lieu de s'engager sur un mot de passe particulier, le simulateur met en gage *tous* les mots de passe, et il est ainsi capable d'ouvrir sur n'importe lequel d'entre eux. Mettre en gage tous les mots de passe signifie simuler les mises en gage de telle sorte que les chiffrés de tous les x_i, correspondant à tous les pw_i, soient envoyés au lieu de chiffrés de 0 (voir la partie sur l'équivocabilité de la section 8.6 page 131 pour les détails). Rappelons que les valeurs hk et hp ne dépendent pas du mot de passe, si bien qu'elles n'engagent le joueur sur aucun d'eux.

Initialisation de la session. Sur réception du message $(\texttt{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathrm{I}, \mathrm{J}, \texttt{role})$ de la part de $\mathcal{F}_{pw\mathrm{KE}}$, $\mathcal{S}$ commence à simuler une nouvelle session du protocole pour le joueur $\mathrm{P_I}$, avec partenaire $\mathrm{P_J}$, identifiant de session ssid, et CRS $(\mathrm{G}, \mathsf{pk}, (y_1, \ldots, y_m))$. Appelons cette session $(\mathrm{P_I}, \mathsf{ssid})$. Si $\mathsf{role} = \texttt{client}$, alors $\mathcal{S}$ genère un message `flow-one` en mettant en gage tous les mots de passe et en choisissant une paire de clefs $(\mathsf{SK_I}, \mathsf{VK_I})$ pour un schéma de *one-time* signature. Il envoie ce message à $\mathcal{A}$ au nom de $(\mathrm{P_I}, \mathsf{ssid})$.

Si $(\mathrm{P_I}, \mathsf{ssid})$ se fait corrompre à ce stade, alors $\mathcal{S}$ récupère son mot de passe $\mathsf{pw_I}$ et il est capable (grâce à l'équivocabilité) d'ouvrir cette mise en gage sur $\mathsf{pw_I}$. Il peut alors donner à $\mathcal{A}$ des données cohérentes.

Étapes du protocole. Supposons que $\mathcal{A}$ envoie un message m à une session active d'un joueur. Si ce message est formatté différent de ce que la session attend, alors $\mathcal{S}$ abandonne cette session et prévient $\mathcal{A}$. Sinon, nous sommes face aux cas suivants (où l'on note $\mathrm{P_I}$ un client et $\mathrm{P_J}$ un serveur) :

1. Supposons que le joueur $(\mathrm{P_J}, \mathsf{ssid})$ reçoive un message $m = (\texttt{flow-one}, \mathsf{com_I}, \mathsf{VK_I})$. Alors $\mathrm{P_J}$ est nécessairement un serveur et m est le premier message reçu par $\mathrm{P_J}$. Si $\mathsf{com_I}$ n'est pas une mise en gage générée par $\mathcal{S}$ pour un message `flow-one`, $\mathcal{S}$ utilise la clef secrète sk pour déchiffrer et obtenir $\mathsf{pw_I}$ ou rien. Ne rien obtenir est considéré comme obtenir un mot de passe invalide ci-dessous grâce à la construction du SHS relié à la mise en gage. Quand l'extraction réussit, grâce à la propriété de *binding*, un seul $\mathsf{pw_I}$ est possible ($\mathcal{A}$ tente une attaque par dictionnaire en ligne), et $\mathcal{S}$ fait alors une requête $(\texttt{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathrm{J}, \mathsf{pw_I})$ à la fonctionnalité. Si le mot de passe est correct, alors $\mathcal{S}$ fixe le mot de passe du serveur à $\mathsf{pw_I}$, sinon c'est un mot de passe invalide. Dans tous les cas, $\mathcal{S}$ produit une mise en gage $\mathsf{com_J}$ sur tous les mots de passe (il utilise la propriété d'équivocabilité), choisit une paire de clefs $(\mathsf{SK_J}, \mathsf{VK_J})$ pour un schéma de *one-time* signature, exécute les algorithmes de génération de clef du SHS sur $\mathsf{com_I}$ pour produire $(\mathsf{hk_J}, \mathsf{hp_J})$ et envoie le message `flow-two` $(\mathsf{com_J}, \mathsf{VK_J}, \mathsf{hp_J})$ à $\mathcal{A}$ au nom de $(\mathrm{P_J}, \mathsf{ssid})$.

   Si l'expéditeur $(\mathrm{P_I}, \mathsf{ssid})$ de ce message ou le destinataire $(\mathrm{P_J}, \mathsf{ssid})$ se fait corrompre d'ici la fin de cette étape, $\mathcal{S}$ gère cette corruption exactement comme si ce joueur s'était fait corrompre à la fin de l'étape d'initialisation. Notons en outre que $\mathcal{S}$ est capable de calculer et donner à $\mathcal{A}$ une valeur correcte pour $\mathsf{Hash_J}$, la clef projetée étant indépendante du mot de passe (voir la discussion page 134).

2. Supposons que le joueur $(\mathrm{P_I}; \mathsf{ssid})$ reçoive un message $m = (\texttt{flow-two}, \mathsf{com_J}, \mathsf{VK_J}, \mathsf{hp_J})$. Alors $\mathrm{P_I}$ doit être un client qui a envoyé un message `flow-one` et qui attend maintenant la réponse. On dit que $(\mathrm{P_J}, \mathsf{ssid})$ est une session partenaire de $(\mathrm{P_I}, \mathsf{ssid}')$ si $\mathsf{ssid} = \mathsf{ssid}'$, si le joueur $(\mathrm{P_I}, \mathsf{ssid})$ a comme partenaire $\mathrm{P_J}$, si le joueur $(\mathrm{P_J}, \mathsf{ssid})$ a comme partenaire $\mathrm{P_I}$, et ces deux sessions ont des rôles opposés (`client`/`server`). Si la paire $(\mathsf{com_J}, \mathsf{VK_J})$ n'est pas égale à la paire $(\mathsf{com_J}, \mathsf{VK_J})$ qui a été générée par $\mathcal{S}$ pour un message `flow-one` de la session partenaire $(\mathrm{P_J}, \mathsf{ssid})$ (ou si aucun tel chiffré n'a déjà été généré, ou si aucune telle session partenaire n'existe) alors $\mathcal{S}$ utilise sa clef secrète sk pour calculer $\mathsf{pw_J}$, ou rien, ce qui est considéré comme un mot de passe invalide ci-dessous (comme avant). Notons aussi que $\mathcal{S}$ peut récupérer $\mathsf{pw_J}$ si $(\mathrm{P_J}, \mathsf{ssid})$ a été corrompu après avoir envoyé sa mise en gage. Si $\mathsf{pw_J}$ a été récupéré, $\mathcal{S}$ fait une requête $(\texttt{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathrm{I}, \mathsf{pw_J})$ à la fonctionnalité. Si le mot de passe est correct, $\mathcal{S}$ fixe le mot de passe de ce client à $\mathsf{pw_J}$, sinon c'est un mot de passe invalide.

Il exécute alors l'algorithme de génération de clefs du SHS sur $\mathsf{com}_{\mathrm{J}}$ pour produire $(\mathsf{hk}_{\mathrm{I}}, \mathsf{hp}_{\mathrm{I}})$, ainsi que l'algorithme de signature avec SK_{I} pour calculer σ_{I}. Il envoie alors le message `flow-three` $(\sigma_{\mathrm{I}}, \mathsf{hp}_{\mathrm{I}})$ à $\mathcal{A}$ au nom de $(\mathrm{P}_{\mathrm{I}}, \mathsf{ssid})$.

Notons que dans le cas précédent (mot de passe correct), $\mathcal{S}$ calcule honnêtement la clef de session en utilisant le mot de passe pw_{J}, sans le révéler. Sinon, P_{I} reçoit une clef choisie au hasard.

Si $(\mathrm{P}_{\mathrm{J}}, \mathsf{ssid})$ se fait corrompre à la fin de cette étape, $\mathcal{S}$ gère cette corruption comme lorsque ce joueur se fait corrompre à la fin de l'étape précédente.

Si c'est $(\mathrm{P}_{\mathrm{I}}, \mathsf{ssid})$ qui se fait corrompre, nous faisons face à deux cas. Si un test correct de mot de passe a eu lieu précédemment, alors $\mathcal{S}$ a tout calculé honnêtement et peut donner chaque valeur à $\mathcal{A}$ (rappelons que la clef de projection ne dépend pas du mot de passe). Sinon, si $\mathcal{S}$ a fixé la clef de session au hasard, rappelons qu'il n'a encore rien envoyé, et donc que l'adversaire ignore totalement les valeurs calculées. $\mathcal{S}$ récupère alors le mot de passe de $(\mathrm{P}_{\mathrm{I}}, \mathsf{ssid})$ et il est capable de calculer les valeurs et de les donner à l'adversaire.

3. Supposons qu'un joueur $(\mathrm{P}_{\mathrm{J}}, \mathsf{ssid})$ reçoive un message $m = (\texttt{flow-three}, \sigma_{\mathrm{I}}, \mathsf{hp}_{\mathrm{I}})$. Alors $(\mathrm{P}_{\mathrm{J}}, \mathsf{ssid})$ doit être un serveur qui a envoyé un message `flow-two` et qui attend maintenant la réponse. $\mathcal{S}$ abandonne si la signature σ_{I} est invalide. Si `flow-one` n'était pas *oracle-generated*, alors $\mathcal{S}$ a extrait le mot de passe pw_{I} à partir de la mise en gage (ou a échoué à l'extraire). De même, si la session partenaire P_{I} (existe et) a été corrompue plus tôt dans le protocole, alors $\mathcal{S}$ connaît son mot de passe pw_{I} (qui a en particulier été utilisé dans la mise en gage). Dans tous les cas, le simulateur pose une requête $(\texttt{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathrm{J}, \mathsf{pw}_{\mathrm{I}})$ à la fonctionnalité pour vérifier la compatibilité des deux mots de passe. Dans le cas d'une réponse correcte, $\mathcal{S}$ fixe le mot de passe de la session serveur à pw_{I} et calcule honnêtement la clef de session en utilisant pw_{I}. Sinon, P_{J} reçoit une clef choisie au hasard.

   Ensuite, $\mathcal{S}$ exécute l'algorithme de signature avec SK_{J} pour calculer σ_{J} et envoie ce message `flow-four` à $\mathcal{A}$ au nom de $(\mathrm{P}_{\mathrm{J}}, \mathsf{ssid})$.

   $\mathcal{S}$ gère une corruption de $(\mathrm{P}_{\mathrm{I}}, \mathsf{ssid})$ comme il l'a fait à la fin de l'étape précédente. Et rappelons qu'aucune autre corruption de $(\mathrm{P}_{\mathrm{J}}, \mathsf{ssid})$ ne peut avoir lieu puisqu'il a déclaré sa session `completed` et qu'il a effacé son état interne.

4. Supposons qu'une session $(\mathrm{P}_{\mathrm{I}}, \mathsf{ssid})$ reçoive un message $m = (\texttt{flow-four}, \sigma_{\mathrm{J}})$. $(\mathrm{P}_{\mathrm{I}}, \mathsf{ssid})$ doit alors être un client qui a envoyé un message `flow-three` et attend désormais la réponse. $\mathcal{S}$ abandonne si la signature σ_{J} est invalide. Si `flow-two` n'a pas été *oracle-generated*, alors $\mathcal{S}$ a extrait le mot de passe pw_{J} de la mise en gage (ou a échoué à l'extraire). De manière similaire, si la session partenaire P_{J} (existe et) a été corrompue plus tôt dans le protocole, alors $\mathcal{S}$ connaît son mot de passe pw_{J} (qui a en particulier été utilisé dans la mise en gage). Dans tous les cas, le simulateur effectue une requête $(\texttt{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathrm{J}, \mathsf{pw}_{\mathrm{J}})$ à la fonctionnalité pour vérifier la compatibilité des deux mots de passe. Dans le cas d'une réponse correcte, $\mathcal{S}$ fixe le mot de passe de la session client à pw_{J} et calcule honnêtement sa clef de session. Sinon, il fixe sa clef de session au hasard. Notons finalement qu'aucune corruption ne peut avoir lieu à ce stade.

Si une session abandonne ou termine, $\mathcal{S}$ le rapporte à $\mathcal{A}$. Si cette session termine avec une clef de session sk, alors $\mathcal{S}$ fait une requête `NewKey` à $\mathcal{F}_{pw\mathrm{KE}}$, en spécifiant la clef de session. Mais rappelons que sauf si la session est *compromised* ou *interrupted*, $\mathcal{F}_{pw\mathrm{KE}}$ va ignorer la clef spécifiée par $\mathcal{S}$ et nous n'avons donc pas à nous inquiéter de la valeur de la clef dans ces cas.

9.2.3 Description des jeux

Nous donnons désormais la description complète des jeux. La preuve détaillée de l'indistinguabilité entre certains d'entre eux peut être trouvée dans la section 9.2.4.

Jeu $\mathbf{G}_0$: $\mathbf{G}_0$ est le jeu réel.

Jeu $\mathbf{G}_1$: À partir de ce jeu, le simulateur est autorisé à programmer partiellement la CRS, ce qui lui permet de connaître les trappes d'extractibilité (mais pas d'équivocabilité).

Jeu $\mathbf{G}_2$: Ce jeu est pratiquement le même que le précédent. La seule différence est que $\mathcal{S}$ essaie toujours d'extraire le mot de passe mis en gage par l'adversaire (sans utiliser sa connaissance du mot de passe pour le moment) dès que ce dernier essaie de contrôler l'un des joueurs. Le simulateur est autorisé à abandonner si cette extraction échoue car l'adversaire a généré une mise en gage valide pour deux mots de passe ou plus. Grâce à la propriété de *binding* de la mise en gage (voir la section 8.6 page 131), la probabilité que l'adversaire parvienne à réaliser cela est négligeable. Notons que l'extraction peut aussi échouer si les valeurs envoyées n'étaient pas des chiffrés de 0 ou 1 mais le simulateur n'abandonne pas dans ce cas. Pour le moment, le simulateur connaît toujours les mots de passe des joueurs, et dans les jeux suivants, quand ce ne sera plus le cas, nous montrerons que les résultats des SPHF seront aléatoires si bien que cet échec n'aura pas de mauvaise conséquence. Ceci montre que $\mathbf{G}_2$ et $\mathbf{G}_1$ sont indistinguables.

Jeu $\mathbf{G}_3$: Dans ce jeu, $\mathcal{S}$ connaît toujours les mots de passe des joueurs, mais il programme désormais totalement la CRS. Comme elle est indistinguable, cela ne change rien au jeu précédent. En revanche, $\mathcal{S}$ connaît désormais la trappe d'équivocabilité et il simule les mises en gage en s'engageant sur tous les mots de passe possibles (afin de pouvoir équivoquer ensuite, voir la section 8.6 pour plus de détails). Notons que puisque le commitment est *hiding*, ceci ne change pas la vision d'un environnement. En outre, la mise en gage reste *binding* (même sous l'accès à des mises en gage équivocables, comme expliqué dans les sections 8.6 et 8.5.2). Notons aussi que la génération des clefs projetées pour les SPHF (voir la section 8.6) est effectuée sans nécessiter la connaissance du mot de passe. Ainsi, $\mathbf{G}_3$ et $\mathbf{G}_2$ sont indistinguables.

Remarquons au passage que nous venons de prouver que la connaissance des mots de passe n'est plus nécessaire pour les étapes (U1) et (S2). Si un joueur se fait corrompre, le simulateur récupère son mot de passe et est capable d'équivoquer la mise en gage et de donner à l'adversaire des données cohérentes (puisque la clef projectée pour la SPHF ne dépend pas du mot de passe engagé).

Jeu $\mathbf{G}_4$: Dans ce jeu, nous supposons que `flow-one` a été *oracle-generated*. Nous nous situons donc au début de l'étape (U3) et nous souhaitons simuler le client (honnête). Nous supposons que le simulateur connaît encore le mot de passe du serveur, mais plus celui du client. Considérons tout d'abord le cas dans lequel Alice a reçu un message `flow-two` *oracle-generated*. Dans un tel cas, le simulateur choisit une valeur $\mathsf{Hash}(\mathsf{hk}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{J})$ aléatoire. Ensuite, si le serveur reste honnête jusqu'à (S4), le simulateur pose une requête `SamePwd` à la fonctionnalité. Si la réponse est oui (c'est-à-dire si $\mathsf{pw}_\mathrm{I} = \mathsf{pw}_\mathrm{J}$), il donne à Bob la même valeur aléatoire pour $\mathsf{ProjHash}(\mathsf{hp}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{J}; \mathsf{w}_\mathrm{J})$ et calcule honnêtement $\mathsf{Hash}(\mathsf{hk}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{I})$, déterminant ainsi complètement la clef sk_J. Sinon, il calcule correctement la clef en entier. Si les deux joueurs restent honnêtes jusqu'à (U5), alors, s'ils partagent le même mot de passe, $\mathcal{S}$ fixe $\mathsf{ProjHash}(\mathsf{hp}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{I}; \mathsf{w}_\mathrm{I}) = \mathsf{Hash}(\mathsf{hk}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{I})$. Sinon, $\mathcal{S}$ fixe sk_I au hasard.

La description des corruptions et la preuve d'indistinguabilité entre $\mathbf{G}_4$ et $\mathbf{G}_3$ peut être trouvée dans la section 9.2.4 (elle provient du caractère pseudo-aléatoire de la SPHF).

Jeu $\mathbf{G}_5$: Dans ce jeu, on continue à supposer que `flow-one` a été *oracle-generated*, mais l'on considère désormais le cas dans lequel `flow-two` n'a pas été *oracle-generated*. Nous sommes désormais au début de l'étape (U3) et nous souhaitons simuler le client (honnête). Le simulateur connaît toujours le mot de passe du serveur.

Puisque le deuxième message n'est pas *oracle-generated*, com_J, VK_J ou hk_J a été généré par l'adversaire. Si com_J était un rejeu, l'adversaire ne pourrait pas modifier VK_J à cause de l'étiquette utilisée pour com_J, et alors la vérification de la signature échouerait. com_J est donc nécessairement une nouvelle mise en gage. Rappelons que depuis $\mathbf{G}_2$, le simulateur extrait le mot de passe de la mise en gage du serveur, grâce à la clef secrète. L'extraction va échouer avec probabilité négligeable si $\mathcal{S}$ récupère deux mots de passe différents (grâce à la propriété de *binding*, voir $\mathbf{G}_2$ et $\mathbf{G}_3$). Dans un tel cas, le simulateur abandonne le jeu. Il peut aussi arriver que l'extraction ne mène à aucun mot de passe, ou bien qu'elle échoue si les valeurs envoyées n'étaient pas des chiffrés de 0 ou de 1.

Ensuite, s'il a récupéré un mot de passe, le simulateur pose une requête `GoodPwd` à la fonctionnalité pour le client. Si le mot de passe proposé est correct, il calcule honnêtement la clef de session du client. Sinon, ou s'il n'a récupéré aucun mot de passe dans l'extraction, il fixe la valeur $\mathsf{Hash}(\mathsf{hk}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{J})$ au hasard (la clef entière sk_I sera fixée à l'étape (U5)). Notons que la valeur renvoyée par la SPHF sera nécessairement aléatoire par construction si les valeurs envoyées dans les mises en gage n'étaient pas des chiffrés de 0 ou de 1.

Les corruptions qui peuvent faire suite à cette étape sont gérées comme dans le jeu précédent (rappelons que les clefs projetées ne dépendent pas du mot de passe). Grâce à la *smoothness* de la fonction de hachage, ce jeu est indistinguable du précédent (les détails sont donnés dans la section 9.2.4 page 147).

Jeu $\mathbf{G}_6$: Dans ce jeu, nous gérons le cas dans lequel tous les messages reçus par le client jusqu'à (S4) sont *oracle-generated*. Nous sommes désormais au début de (S4) et nous souhaitons simuler le serveur (honnête). Nous supposons que le simulateur ne connaît plus aucun mot de passe. Considérons en premier le cas dans lequel `flow-three` a été *oracle-generated*. Notons que `flow-one` a alors nécessairement été *oracle-generated* aussi, car sinon la signature σ_1 aurait été rejetée.

Le simulateur pose alors une requête `SamePwd` à la fonctionnalité. Si la réponse est correcte, il fixe la valeur de $\mathsf{ProjHash}(\mathsf{hp}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{J}; \mathsf{w}_\mathrm{J})$ à $\mathsf{Hash}(\mathsf{hk}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{J})$, et il fixe aussi celle de $\mathsf{Hash}(\mathsf{hk}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{I})$ au hasard. Sinon, il fixe ces valeurs au hasard.

À l'étape (U5), si les mots de passe d'Alice et Bob sont les mêmes et que les deux sont encore honnêtes, on fixe la valeur $\mathsf{ProjHash}(\mathsf{hp}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{I}; \mathsf{w}_\mathrm{I}) = \mathsf{Hash}(\mathsf{hk}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{I})$: sk_I et sk_J sont alors égales, comme requis. Notons que puisque $\mathsf{Hash}(\mathsf{hk}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{J})$ est déjà fixé au hasard depuis $\mathbf{G}_4$ ou $\mathbf{G}_5$, toutes les clefs sont déjà aléatoires.

Si les mots de passe ne sont pas les mêmes mais que les deux joueurs sont encore honnêtes, Alice va recevoir en (U5) une clef choisie indépendamment et aléatoirement. Ici, la valeur $\mathsf{ProjHash}(\mathsf{hp}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{I}), \mathsf{com}_\mathrm{I}; \mathsf{w}_\mathrm{I})$ n'a pas besoin d'être programmée, puisque les clefs n'ont aucune raison d'être identiques. Dans ce cas, comme dans $\mathbf{G}_4$, le caractère pseudo-aléatoire des fonctions de hachage assure l'indistinguabilité (voir la section 9.2.4 page 147 pour le traitement des corruptions).

Jeu $\mathbf{G}_7$: Dans ce jeu, nous gérons toujours le cas dans lequel tous les messages reçus par le client jusqu'en (S4) ont été *oracle-generated*. À nouveau, nous nous plaçons au début de (S4) et nous souhaitons simuler le serveur (honnête), sans connaître aucun mot de passe, mais l'on suppose maintenant que `flow-three` n'était pas *oracle-generated*. Notons que dans ce cas `flow-one` ne peut pas avoir été *oracle-generated* car sinon, la signature σ_1 aurait été rejetée.

Rappelons que, depuis $\mathbf{G}_2$, le simulateur extrait le mot de passe à partir de la mise en gage du client, grâce à la clef secrète. Cette extraction peut échouer avec probabilité négligeable si $\mathcal{S}$ récupère deux mots de passe différents (grâce à la propriété de *binding*, voir $\mathbf{G}_2$ et $\mathbf{G}_3$). Dans un tel cas, le simulateur abandonne le jeu. Il peut aussi arriver que l'extraction ne mène à aucun mot de passe (par exemple si les valeurs envoyées dans la mise en gage n'était pas des chiffrés de 0 ou 1). Ensuite, s'il a récupéré un mot de passe, le simulateur pose une requête `GoodPwd` à la fonctionnalité. Si elle est correcte, il calcule alors honnêtement la clef de session du serveur. Sinon, ou s'il n'a récupéré aucun mot de passe dans l'extraction, il fixe les valeurs $\mathsf{Hash}(\mathsf{hk}_\mathrm{J}; \rho, (\ell_\mathrm{I}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{I})$ et $\mathsf{ProjHash}(\mathsf{hp}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{J}; \mathsf{w}_\mathrm{J})$ aléatoirement. Rappelons que si les valeurs envoyées dans les mises en gage n'étaient pas des chiffrés de 0 ou 1, la valeur renvoyée par la SPHF va être aléatoire.

Les corruptions pouvant suivre cette étape sont gérées comme dans le jeu précédent. Grâce à la *smoothness* de la fonction de hachage, ce jeu est indistinguable du précédent. La preuve est exactement la même que pour $\mathbf{G}_5$, en plus simple, puisque la clef du client est déjà aléatoire.

Jeu $\mathbf{G}_8$: Dans ce jeu, nous remplaçons les requêtes `GoodPwd` par des requêtes `TestPwd`, et les requêtes `SamePwd` par des requêtes `NewKey`. Si une session abandonne ou termine, $\mathcal{S}$ le rapporte à $\mathcal{A}$. Si une session termine avec la clef de session sk, alors $\mathcal{S}$ fait une requête `NewKey` à la fonctionnalité, en spécifiant la clef de session sk. Mais rappelons que sauf si la session

est `compromised`, la fonctionnalité va ignorer la clef spécifiée par $\mathcal{S}$. Nous montrons dans la section 9.2.4 page 148 que ce jeu est indistinguable du jeu idéal.

9.2.4 Détails de la preuve

Indistinguabilité de $\mathbf{G_4}$ et $\mathbf{G_3}$. Décrivons ce qui se passe dans le cas de corruptions. Tout d'abord, si Alice se fait corrompre après l'exécution de (U3), alors le simulateur récupère son mot de passe et il est capable de tout calculer correctement. Rappelons que la clef projetée ne dépend pas du mot de passe.

Ensuite, si Bob se fait corrompre après (U3) ou pendant (S4), alors le simulateur est capable de tout calculer correctement. En particulier, dans le second cas, l'adversaire va obtenir une valeur cohérente de sk_J. Le simulateur va alors poser à la fonctionnalité une requête `GoodPwd` pour Alice avec le mot de passe de Bob. Si le mot de passe est le même, il a alors récupéré celui d'Alice, et comme il n'a pas réellement effacé ses données, il va être capable de calculer sk_I exactement comme l'adversaire l'aurait fait pour Bob (ceci est en particulier dû au fait que la clef de projection ne dépend pas du mot de passe). Sinon, il donne à Alice une clef aléatoire : il n'y a pas besoin que les joueurs obtiennent la même clef s'ils ne partagent pas le même mot de passe – rappelons que toutes les données privées sont effacées afin que l'adversaire ne puisse pas vérifier quoi que ce soit.

Finalement, si Alice se fait corrompre d'ici la fin de (U5), le simulateur va récupérer son mot de passe et poser une requête `SamePwd` à la fonctionnalité. Si σ_J est correcte et si les joueurs partagent le même mot de passe, alors le simulateur calcule sk_I exactement comme l'adversaire l'aurait fait pour Bob (une fois encore grâce au fait que la clef de projection ne dépend pas du mot de passe). Sinon, si la signature est correcte, mais que leurs mots de passe sont différents, alors Alice reçoit une valeur aléatoire (comme ses données sont effacées, l'adversaire n'est pas supposé être capable de les vérifier). Sinon, si la signature est incorrecte, le simulateur abandonne le jeu. Nous pouvons voir ici la nécessité de l'étape (U5) pour garantir la sécurité adaptative.

Montrons désormais qu'un environnement capable de distinguer entre $\mathbf{G}_4$ et $\mathbf{G}_3$ peut être utilisé pour construire un distingueur entre `Expt-Hash` et `Expt-Unif` tels que décrit dans [GL03] (voir la section 8.3.2 page 122), ce qui contredit leur corollaire 3.3.

Commençons par décrire des jeux hybrides H_i comme suit. Notons tout d'abord que les sessions sont ordonnées par rapport aux étapes (U1). Dans toutes les sessions avant la i-ième, le calcul est effectué aléatoirement, et dans toutes les sessions suivantes $(i, i+1, \ldots)$, les valeurs sont réelles. Dans tous les cas « aléatoires », on fixe $\mathsf{ProjHash}(\mathsf{hp}_\mathrm{I}; \rho, (\ell_\mathrm{J}, \mathsf{pw}_\mathrm{J}), \mathsf{com}_\mathrm{J}; \mathsf{w}_\mathrm{J})$ à la même valeur pendant la simulation du serveur si « tout se passe bien », c'est-à-dire si aucune corruption n'a eu lieu et si les mots de passe sont les mêmes. Avec ces notations, le cas $i = 1$ est exactement $\mathbf{G}_3$ et le cas $i = q_s + 1$ est exactement $\mathbf{G}_4$. Graphiquement,

$$\begin{cases} \text{Random} & 1, \ldots, i-1 \\ \text{Real} & i, \ldots, q_s \end{cases}$$

Notre objectif est désormais de prouver l'indistinguabilité entre H_i et H_{i+1}. Nous définissons l'événement E_i : « il y a eu une corruption entre les étapes de mise en gage (U1) et de hachage (U3) dans la i-ième session. » Notons que la probabilité de cet événement est le même dans les deux jeux H_i et H_{i+1}. Ceci est vrai en dépit du fait que l'on considère des sessions concurrentes. Pour voir cela, notons que même si E_i peut dépendre de sessions avec un indice plus élevé, la seule différence entre ces deux jeux concerne l'étape (U3) de la i-ième session : tout est identique avant ce point. Étant donné qu'aucune corruption n'a lieu avant cette étape, la probabilité que l'adversaire corrompe un joueur dans la i-ième session, qui ne dépend que de ce qui s'est passé au préalable, est la même dans les deux cas.

Nous notons alors $\mathrm{OUT}_\mathcal{Z}$ la sortie de l'environnement à la fin de l'exécution et nous calculons la différence entre $\Pr[\mathrm{OUT}_\mathcal{Z} = 1]$ dans les deux différents jeux :

$$
\begin{aligned}
&\left|\Pr_{\mathrm{H}_{i+1}}[\mathrm{OUT}_{\mathcal{Z}}=1]-\Pr_{\mathrm{H}_i}[\mathrm{OUT}_{\mathcal{Z}}=1]\right| \\
=\ &\left|\Pr_{\mathrm{H}_{i+1}}[\mathrm{OUT}_{\mathcal{Z}}=1\wedge \mathrm{E}_i]+\Pr_{\mathrm{H}_{i+1}}[\mathrm{OUT}_{\mathcal{Z}}=1\wedge\neg\mathrm{E}_i]\right. \\
&\qquad \left.-\Pr_{\mathrm{H}_i}[\mathrm{OUT}_{\mathcal{Z}}=1\wedge \mathrm{E}_i]-\Pr_{\mathrm{H}_i}[\mathrm{OUT}_{\mathcal{Z}}=1\wedge\neg\mathrm{E}_i]\right| \\
\leqslant\ &\left|\Pr_{\mathrm{H}_{i+1}}[\mathrm{OUT}_{\mathcal{Z}}=1\wedge \mathrm{E}_i]-\Pr_{\mathrm{H}_i}[\mathrm{OUT}_{\mathcal{Z}}=1\wedge \mathrm{E}_i]\right| \\
&\qquad +\left|\Pr_{\mathrm{H}_{i+1}}[\mathrm{OUT}_{\mathcal{Z}}=1\wedge\neg\mathrm{E}_i]-\Pr_{\mathrm{H}_i}[\mathrm{OUT}_{\mathcal{Z}}=1\wedge\neg\mathrm{E}_i]\right| \\
\leqslant\ &\left|\Pr_{\mathrm{H}_{i+1}}[\mathrm{OUT}_{\mathcal{Z}}=1|\mathrm{E}_i]-\Pr_{\mathrm{H}_i}[\mathrm{OUT}_{\mathcal{Z}}=1|\mathrm{E}_i]\right|\ \left|\Pr_{\mathrm{H}_i}[\mathrm{E}_i]\right| \\
&\qquad +\left|\Pr_{\mathrm{H}_{i+1}}[\mathrm{OUT}_{\mathcal{Z}}=1|\neg\mathrm{E}_i]-\Pr_{\mathrm{H}_i}[\mathrm{OUT}_{\mathcal{Z}}=1|\neg\mathrm{E}_i]\right|\ \left|\Pr_{\mathrm{H}_i}[\neg\mathrm{E}_i]\right|
\end{aligned}
$$

Considérons d'abord le premier terme de cette somme. S'il y a une corruption, nous apprenons le mot de passe et aussi (enfin, nous simulons de façon cohérente) l'aléa, ce qui nous permet de tout calculer correctement. Ainsi, ce terme est égal à 0.

Considérons désormais le deuxième terme, correspondant au cas dans lequel il n'y a pas de corruption, et montrons que $|\Pr[\texttt{Expt-Hash}(\mathrm{D})=1]-\Pr[\texttt{Expt-Unif}(\mathrm{D})=1]|$ borne (à une quantité négligeable près) la probabilité que l'environnement distingue H_{i+1} de H_i. Plus précisément, considérons le jeu H suivant dans lequel les oracles `Commit` et `Hash` apparaissent uniquement dans la i-ième session sous l'hypothèse $\neg\mathrm{E}_i$: soit D une machine recevant une clef publique pk_{J} choisie aléatoirement et qui émule le jeu H_i avec les changements suivants pour la i-ième session. Sur réception d'un message `flow-one` *oracle-generated*, D ne calcule pas directement $\mathsf{com}_{\mathrm{J}}$ mais il pose en fait une requête à l'oracle `Commit`() et il fixe $\mathsf{com}_{\mathrm{J}}$ à la valeur retournée. Si Alice reçoit le message $\mathsf{com}_{\mathrm{J}}$ non modifié dans le message `flow-two`, alors D pose la requête $\texttt{Hash}(\mathsf{com}_{\mathrm{J}})$ et reçoit une paire $(s_{\mathrm{I}},\eta_{\mathrm{I}})$. Il fixe alors $\mathsf{hp}_{\mathrm{I}}=s_{\mathrm{I}}$ et $\mathsf{ProjHash}(\mathsf{hp}_{\mathrm{I}};\rho,(\ell_{\mathrm{J}},\mathsf{pw}_{\mathrm{I}}),\mathsf{com}_{\mathrm{J}};\mathsf{w}_{\mathrm{J}})=\eta_{\mathrm{I}}$. Notons que w_{J} représente ici l'aléa utilisé par l'oracle `Commit` dans la requête qui a généré $\mathsf{com}_{\mathrm{J}}$. Ensuite, si Bob reçoit la clef projetée hp_{I} non modifiée, D utilise aussi η_{I} pour la portion appropriée de la clef de session – dans le cas où ils partagent le même mot de passe. Finalement, la sortie de D est égale à la sortie de l'environnement. Il est facile de voir que $\mathrm{H}=\mathrm{H}_{i+1}$ dans le cas où l'oracle `Hash` retourne une valeur aléatoire, et qu'il est égal à H_i sinon.

Indistinguabilité de $\mathbf{G_5}$ et $\mathbf{G_4}$. En premier lieu, remarquons que les corruptions qui peuvent suivre cette étape sont gérées comme dans $\mathbf{G}_4$, et que la simulation de $\mathbf{G}_5$ est compatible avec $\mathbf{G}_4$.

Si le mot de passe récupéré du serveur est correct, la simulation est effectuée honnêtement, de sorte que ce jeu est parfaitement équivalent au précédent. Sinon, si le mot de passe est incorrect, ou si aucun mot de passe n'a été récupéré, alors $\mathsf{com}_{\mathrm{J}}$ est invalide.

Étant donné un $\mathsf{com}_{\mathrm{J}}$ invalide, avec le mot de passe pw_{I} de $(\mathrm{P}_{\mathrm{I}},\mathsf{ssid})$ et l'étiquette ℓ_{I}, la distribution $\{\mathsf{pk}_{\mathrm{I}},\mathsf{com}_{\mathrm{I}},\ell_{\mathrm{I}},\mathsf{pw}_{\mathrm{I}},\mathsf{hp}_{\mathrm{I}},\mathsf{Hash}(\mathsf{hk}_{\mathrm{I}};\rho,(\ell_{\mathrm{J}},\mathsf{pw}_{\mathrm{I}}),\mathsf{com}_{\mathrm{J}})\}$ est statistiquement proche de la distribution $\{\mathsf{pk}_{\mathrm{I}},\mathsf{com}_{\mathrm{I}},\ell_{\mathrm{I}},\mathsf{pw}_{\mathrm{I}},\mathsf{hp}_{\mathrm{I}},z\}$ dans laquelle z est un élément aléatoire du groupe G (grâce à la *smoothness* de la fonction de hachage). Puisque $\mathsf{Hash}(\mathsf{hk}_{\mathrm{I}};\rho,(\ell_{\mathrm{J}},\mathsf{pw}_{\mathrm{I}}),\mathsf{com}_{\mathrm{J}})$ est un composant de la clef de session sk_{I} pour $(\mathrm{P}_{\mathrm{I}},\mathsf{ssid})$, alors la clef de session générée est statistiquement proche d'un élément uniforme.

Ainsi, puisque $\mathsf{com}_{\mathrm{J}}$ est invalide, $(\mathrm{P}_{\mathrm{J}},\mathsf{ssid})$ va calculer honnêtement la clef, mais il ne va pas obtenir la même clef de session puisque les mots de passe sont différents. Ce comportement est équivalent à ce qui se passe dans la fonctionnalité idéale : soit les sessions correspondantes de $(\mathrm{P}_{\mathrm{I}},\mathsf{ssid})$ et $(\mathrm{P}_{\mathrm{J}},\mathsf{ssid})$ n'ont pas de conversation partagée, soit elles ont obtenu des mots de passe différents de la part de l'environnement, de sorte que $(\mathrm{P}_{\mathrm{J}},\mathsf{ssid})$ ne va pas recevoir la même clef de session que $(\mathrm{P}_{\mathrm{I}},\mathsf{ssid})$.

Indistinguabilité de $\mathbf{G_6}$ et $\mathbf{G_5}$. C'est le caractère pseudo-aléatoire des fonctions de hachage qui montre l'indistinguabilité de $\mathbf{G}_6$ et $\mathbf{G}_5$. En effet, l'environnement ne peut pas se rendre compte que les clefs ont été choisies au hasard. Plus précisément, on effectue la même manipulation que dans la preuve de $\mathbf{G}_4$, mais en considérant cette fois la SPHF Hash par rapport à $\mathsf{com}_{\mathrm{J}}$. Notons que dans les jeux hybrides, les sessions sont ordonnées par rapport aux étapes (S2).

Considérons désormais le cas dans lequel Bob se fait corrompre pendant l'exécution de (S4). Le simulateur récupère alors son mot de passe et il capable de tout calculer correctement (rappelons que les clefs de projection ne dépendent pas du mot de passe). Il pose alors une requête `GoodPwd` pour le client. Si leurs mots de passe sont différents, Alice reçoit une clef aléatoire. Sinon, $\mathcal{S}$ lui donne la même clef qu'à Bob.

Notons enfin que par simplicité, nous ne calculons $\mathsf{Hash}_{\mathrm{J}}$ qu'à l'étape (S4) dans la simulation. Mais si le serveur est corrompu après (S2), le simulateur récupère son mot de passe et il est capable de donner à l'adversaire une valeur $\mathsf{Hash}_{\mathrm{J}}$ correcte (à nouveau car la clef de projection ne dépend pas du mot de passe).

Indistinguabilité de $\mathbf{G_8}$ et du jeu idéal. La seule différence entre $\mathbf{G}_7$ et $\mathbf{G}_8$ est que les requêtes `GoodPwd` sont remplacées par des requêtes `TestPwd` à la fonctionnalité, et les requêtes `SamePwd` par des requêtes `NewKey`. Des joueurs sont dits avoir des sessions partenaires s'ils partagent le même ssid et s'ils sont d'accord sur les valeurs de $\mathsf{com}_{\mathrm{I}}$, $\mathsf{com}_{\mathrm{J}}$, hp_{I} et hp_{J} (c'est-à-dire sur toutes les valeurs qui déterminent la clef). Nous montrons désormais que $\mathbf{G}_8$ et le jeu idéal sont indistinguables.

D'abord, si les deux joueurs partagent le même mot de passe et restent honnêtes jusqu'à la fin du jeu, et si l'adversaire ne prend le contrôle sur aucun d'entre eux, ils vont obtenir une clef aléatoire, à la fois dans $\mathbf{G}_8$ (depuis $\mathbf{G}_6$) et dans le jeu idéal, puisqu'il n'y a pas de requêtes `TestPwd` et que les sessions restent `fresh`. Ensuite, s'ils partagent le même mot de passe mais qu'il y a des tentatives de prise de contrôle par l'adversaire, ils reçoivent des clefs aléatoires choisies indépendamment (depuis $\mathbf{G}_5$ ou $\mathbf{G}_7$). Enfin, s'ils ne partagent pas le même mot de passe, ils reçoivent des clefs aléatoires choisies indépendamment. Maintenant, nous devons montrer que deux joueurs reçoivent la même clef dans $\mathbf{G}_8$ si et seulement si c'est le cas dans le jeu idéal.

Considérons en premier lieu le cas de joueurs avec des sessions partenaires et le même mot de passe. Si les deux joueurs restent honnêtes jusqu'à la fin du jeu, ils vont recevoir la même clef depuis $\mathbf{G}_6$. Sinon, elle sera identique depuis $\mathbf{G}_5$ ou $\mathbf{G}_7$. Rappelons que s'il y a des tentatives de prises de contrôle, les clefs vont être aléatoires et indépendantes, à la fois dans $\mathbf{G}_8$ et dans le jeu idéal. Dans ce dernier, la fonctionnalité va recevoir deux requêtes `NewSession` avec le même mot de passe. Si les deux joueurs sont honnêtes, elle ne va pas recevoir de requête `TestPwd`, de sorte que la clef va être la même pour les deux. Et si l'un des deux est corrompu et qu'une requête `TestPwd` (correcte) a eu lieu, ils vont aussi recevoir la même clef, choisie par l'adversaire.

Ensuite, considérons le cas de joueurs avec des sessions partenaires mais pas le même mot de passe. S'ils restent tous les deux honnêtes jusqu'à la fin du jeu, ils vont recevoir des clefs aléatoires et choisies indépendamment depuis $\mathbf{G}_6$. Sinon, il en sera de même grâce à $\mathbf{G}_7$. Dans le jeu idéal, la fonctionnalité va recevoir deux requêtes `NewSession` avec des mots de passe différents. Par définition, elle va alors leur donner des clefs différentes.

Enfin, considérons le cas de joueurs sans session partenaire. Il est clair que dans $\mathbf{G}_8$ les clefs de session de ces joueurs vont être indépendantes puisqu'elles ne sont fixées dans aucun des jeux. Dans le jeu idéal, la seule façon que les joueurs reçoivent des clefs correspondantes est que la fonctionnalité reçoive deux requêtes `NewSession` avec le même mot de passe, et que $\mathcal{S}$ envoie une requête `NewKey` pour ces sessions sans avoir envoyé aucune requête `TestPwd`. Mais si les deux sessions n'ont pas de conversation commune, elles doivent différer au moins sur $\mathsf{com}_{\mathrm{I}}$, $\mathsf{com}_{\mathrm{J}}$, hp_{I} ou hp_{J}. Dans ce cas, elles vont refuser la signature de l'autre joueur et abandonner l'exécution.

Si l'un des deux joueurs est corrompu d'ici la fin du jeu, le simulateur récupère son mot de passe et l'utilise dans une requête `TestPwd` à la fonctionnalité pour l'autre joueur, comme expliqué dans $\mathbf{G}_4$. Si le résultat est correct, alors les deux joueurs reçoivent la même clef. Sinon, ils reçoivent des clefs aléatoires choisies indépendamment. C'est exactement le comportement de la fonctionnalité dans le jeu idéal, ce qui achève la preuve.

Chapitre 10

Extraction d'aléa optimale à partir d'un élément de groupe aléatoire

Nous avons vu dans tout ce qui précède que beaucoup de schémas cryptographiques sont basés sur la technique Diffie-Hellman et reposent sur l'hypothèse DDH. Sous cette hypothèse, l'élément Diffie-Hellman obtenu suite aux protocoles d'échange de clefs peut être vu comme un élément parfaitement aléatoire dans le groupe $\mathbb{G} = \langle g \rangle$. Cependant, un élément uniformément distribué dans $\mathbb{G}$ n'est pas une chaîne de bits parfaitement aléatoire. Pour l'usage envisagé de la mise en accord de clefs, qui doit conduire à une clef cryptographique symétrique, nous souhaitons donc le convertir en une chaîne de bits (plus courte) uniformément distribuée (parfaitement, ou au moins statistiquement).

Extraction d'aléa. Une solution classique à ce problème est d'utiliser une fonction de hachage, ce qui permet d'obtenir une chaîne de bits aléatoire, mais dans le modèle de l'oracle aléatoire [BR93]. On remarque qu'il existe $\log_2(q)$ bits d'entropie dans l'élément de groupe, mais on ne sait pas où ils sont exactement : on ne peut pas les extraire directement à partir de la représentation de l'élément dans $\mathbb{G}$. C'est l'objectif d'un extracteur d'aléa. Le *Leftover Hash Lemma* [HILL99, GKR04], que nous avons évoqué page 21, est le plus célèbre d'entre eux. Probabiliste, il peut extraire de l'entropie à partir de n'importe quelle source aléatoire qui ait une min-entropie suffisante. En plus de requérir l'utilisation de fonctions de hachage deux-à-deux indépendantes, qui ne sont pas en standard dans les librairies cryptographiques, son plus grand inconvénient est qu'il demande de l'aléa supplémentaire parfait. Une version calculatoire de ce lemme a aussi été proposé et analysé dans [FPZ08], version qui a l'avantage d'utiliser des fonctions pseudo-aléatoires pour l'extraction d'aléa et pas de telles fonctions de hachage. Ces deux solutions sont génériques, ce qui montre l'intérêt de trouver une solution *déterministe* dédiée à l'extraction d'aléa d'un élément aléatoire de $\mathbb{G}$, qui éviterait l'utilisation d'aléa supplémentaire.

La solution la plus prometteuse dans ce sens est clairement de conserver les bits de poids faible de l'élément de groupe (vue comme une « pré-clef maître », *pre-master key*) et d'es-

pérer que la chaîne de bits résultante soit uniforme, comme c'est proposé dans plusieurs articles [BV96, BS01, JV08]. La troncature est une manipulation simple et déterministe, ce qui a un grand intérêt.

À ICALP 2006, Fouque *et al.* [FPSZ06] ont utilisé cette idée et montré que sous l'hypothèse DDH, les bits de poids faible de g^{ab} sont presque uniformément distribués, étant donné g^a et g^b, si le groupe $\mathbb{G}$ est un sous-groupe multiplicatif suffisamment grand (d'ordre premier q) d'un corps fini $\mathbb{Z}_p$, c'est-à-dire, si q n'est pas trop petit par rapport à p. La taille de q est le plus gros inconvénient puisque qu'il doit valoir au moins la moitié de p, ce qui rend le protocole assez inefficace. Pour montrer ce résultat, les auteurs ont majoré la distance statistique, en évaluant directement la norme L_1, à l'aide de sommes d'exponentielles.

Puisque la cryptographie à base de courbes elliptiques utilise de grands sous-groupes en pratique, le même résultat pour ces courbes pourrait avoir un intérêt pratique. Gürel [Gür05] a étudié le cas des courbes elliptiques définies sur des extensions quadratiques d'un corps fini, avec une grande fraction de bits, et sur un corps fini d'ordre premier, mais avec les mêmes limitations que ci-dessus sur le nombre de bits extraits. Il majore aussi directement la distance statistique à l'aide de la norme L_1, mais en utilisant une somme de caractères de Legendre. Sa technique n'utilise que ce caractère, ce qui n'est pas suffisant dans le cas de $\mathbb{Z}_p$. En conséquence, la technique des auteurs de [FPSZ06] nécessite de sommer sur tous les caractères.

Objectif. Informellement, nous montrons ici que les bits de poids faible d'un élément aléatoire de $\mathbb{G} \subset \mathbb{Z}_p^*$ ou de l'abscisse d'un point aléatoire de $\mathcal{E}(\mathbb{F}_p)$ sont indistinguables d'une chaîne de bits uniforme. Une telle opération est très efficace. C'est aussi un bon extracteur, puisque l'on montre qu'elle peut extraire environ le même nombre de bits que le *Leftover Hash Lemma* (LHL) peut le faire pour la plupart des paramètres de courbes elliptiques, et pour les grands sous-groupes des corps finis. En outre, elle est déterministe.

Ce résultat étend ceux de Canetti *et al.* [CFK+00] et de Fouque *et al.* [FPSZ06] puisque nous sommes capables d'extraire deux fois plus de bits que précédemment. Ce nouveau résultat est obtenu en introduisant une nouvelle technique pour majorer la distance statistique. Tandis que les techniques précédentes s'attachaient à majorer la norme L_1, ce qui est difficile à cause de la valeur absolue, nous majorons ici la norme euclidienne L_2, ce qui est beaucoup plus facile car uniquement des carrés sont en jeu. Nous pouvons aussi, dans certains cas, améliorer nos résultats en utilisant des techniques classiques de sommes d'exponentielles.

Cependant, puisque le résultat ne s'applique qu'à de grands sous-groupes, nous l'étendons ensuite aux groupes de courbes elliptiques, en montrant que, si le groupe $\mathbb{G}$ est généré par P d'ordre premier q et si c est uniformément distribué dans $\mathbb{Z}_q$, les distributions suivantes sont statistiquement indistinguables :

$$\mathrm{U}_k \approx \mathtt{lsb}_k(x(c\mathrm{P}))$$

où U_k est la distribution uniforme sur les chaînes de k bits, $\mathtt{lsb_k}()$ est la fonction qui tronque les k bits de poids faible d'une chaîne et $x()$ est la fonction abscisse sur les points d'une courbe elliptique.

En général, le cofacteur de ces groupes est petit : inférieur à 8, et même égal à 1 pour les courbes du NIST sur des corps finis d'ordre premier. On obtient alors le résultat mentionné ci-dessus en utilisant des techniques plus évoluées sur les sommes d'exponentielles sur des fonctions définies sur les points de la courbe elliptique. Plus précisément, nous montrons que les 82 (resp. 214 et 346) bits de poids faibles de l'abscisse d'un élément aléatoire des courbes du NIST sur des corps finis d'ordre premier à 256 (resp. 384 et 521) bits sont indistinguables d'une chaîne aléatoire. Ils peuvent donc directement être utilisés comme clef symétrique. Pour comparer avec le résultat de Gürel dans [Gür05], pour une courbe elliptique définie sur un corps fini d'ordre premier de 200 bits, il extrait 50 bits avec une distance statistique de 2^{-42}, tandis qu'avec la même distance, on peut extraire 102 bits. Notons que sa preuve était plus facile à comprendre, mais nous n'avons pas réussi à évaluer la norme L_2 de sommes de caractères de Legendre et à la généraliser. Les résultats de ce chapitre ont été présentés à la conférence Eurocrypt 2009 [CFPZ09].

10.1 Notations

Nous introduisons dans cette section les notions utilisées dans l'extraction d'aléa. Dans toute la suite, une source d'aléa est vue comme une distribution de probabilité.

10.1.1 Mesures d'aléa

Pour mesurer l'aléa dans une variable aléatoire, nous utilisons deux mesures distinctes : la *min-entropie* et l'*entropie de collision*. La min-entropie mesure la difficulté qu'a un adversaire de deviner la valeur de la variable aléatoire, tandis que l'entropie de collision mesure la probabilité que deux éléments tirés suivant cette distribution coïncident. L'entropie de collision est ici utilisée comme outil intermédiaire pour établir des résultats, qui sont alors reformulés à l'aide de la min-entropie.

Définition 1 (Min-Entropie)**.** *Soit* X *une variable aléatoire à valeurs dans un ensemble fini* $\mathcal{X}$*. La* probabilité du point le plus probable *de* X*, notée* $\gamma(\mathrm{X})$*, est la probabilité* $\max_{x\in\mathcal{X}}(\Pr[\mathrm{X}=x])$*. La min-entropie de* X *est alors :* $\mathrm{H}_\infty(\mathrm{X}) = -\log_2(\gamma(\mathrm{X}))$*.*

Par exemple, lorsque X suit la distribution uniforme sur un ensemble de taille N, sa min-entropie est $\log_2(\mathrm{N})$.

Définition 2 (Entropie de collision)**.** *Soient* X *et* X′ *deux variables aléatoires indépendantes et identiquement distribuées à valeurs dans un ensemble fini* $\mathcal{X}$*. La probabilité de collision de* X*, appelée* $\mathrm{Col}(\mathrm{X})$*, est la probabilité* $\Pr[\mathrm{X}=\mathrm{X}'] = \sum_{x\in\mathcal{X}}\Pr[\mathrm{X}=x]^2$*. L'entropie de collision de* X *est* $\mathbf{H}_2(\mathrm{X}) = -\log_2(\mathrm{Col}(\mathrm{X}))$*.*

L'entropie de collision est aussi appelée *entropie de Renyi*. Il existe une relation facile entre ces deux types d'entropie : $\mathbf{H}_\infty(\mathrm{X}) \leqslant \mathbf{H}_2(\mathrm{X}) \leqslant 2\cdot\mathbf{H}_\infty(\mathrm{X})$. Pour comparer deux variables aléatoires, nous utilisons la classique distance statistique.

Définition 3 (Distance statistique)**.** *Soient* X *et* Y *deux variables aléatoires à valeurs dans un ensemble fini* $\mathcal{X}$*. La distance statistique entre* X *et* Y *est la valeur de l'expression suivante :*

$$\mathbf{SD}(\mathrm{X},\mathrm{Y}) = \tfrac{1}{2}\sum_{x\in\mathcal{X}}|\Pr[\mathrm{X}=x]-\Pr[\mathrm{Y}=x]|$$

Notons U_k une variable aléatoire uniformément distribuée sur $\{0,1\}^k$. On dit qu'une variable aléatoire X à valeurs dans $\{0,1\}^k$ est δ-*uniforme* si la distance statistique entre X et U_k est majorée par δ.

Lemme 2. *Soient* X *une variable aléatoire à valeurs dans un ensemble* $\mathcal{X}$ *de taille* $|\mathcal{X}|$ *et* $\varepsilon = \mathbf{SD}(\mathrm{X},\mathrm{U}_\mathcal{X})$ *la distance statistique entre* X *et* $\mathrm{U}_\mathcal{X}$*, la variable uniformément distribuée sur* $\mathcal{X}$*. Alors*

$$\mathrm{Col}(\mathrm{X}) \geqslant \frac{1+4\varepsilon^2}{|\mathcal{X}|} \tag{10.1}$$

Démonstration. La preuve provient de [Sho05] et est une conséquence de l'inégalité de Cauchy-Schwarz, qui implique que plus la distance statistique est petite, plus l'inéquation est fine (si X est uniformément distribuée, alors l'inégalité est une égalité). Pour montrer ce lemme, nous avons besoin du résultat suivant, qui implique que la norme 1 est plus petite que la norme 2.

Lemme 3. *Soient* $\mathcal{X}$ *un ensemble fini et* $(\alpha_x)_{x\in\mathcal{X}}$ *une suite de nombres réels. Alors,*

$$\frac{\left(\sum_{x\in\mathcal{X}}|\alpha_x|\right)^2}{|\mathcal{X}|} \leqslant \sum_{x\in\mathcal{X}}\alpha_x^2 \tag{10.2}$$

Cette inégalité est une conséquence directe de l'inégalité de Cauchy-Schwarz :

$$\sum_{x\in\mathcal{X}} |\alpha_x| = \sum_{x\in\mathcal{X}} |\alpha_x| \cdot 1 \leqslant \sqrt{\sum_{x\in\mathcal{X}} \alpha_x^2} \cdot \sqrt{\sum_{x\in\mathcal{X}} 1^2} \leqslant \sqrt{|\mathcal{X}|}\sqrt{\sum_{x\in\mathcal{X}} \alpha_x^2}$$

Maintenant, si X est une variable aléatoire à valeurs dans $\mathcal{X}$ et si nous considérons que $\alpha_x = \Pr[\mathrm{X} = x]$, alors, puisque la somme des probabilités est égale à 1, et puisqu'on a l'égalité $\mathrm{Col}(\mathrm{X}) = \sum_{x\in\mathcal{X}} \Pr[\mathrm{X} = x]^2$,

$$\frac{1}{|\mathcal{X}|} \leqslant \mathrm{Col}(\mathrm{X}) \tag{10.3}$$

Nous pouvons alors prouver le lemme 2. Si $\varepsilon = 0$, le résultat est une conséquence facile de l'équation 10.3 . Supposons ε différent de 0. Définissant $q_x = |\Pr[\mathrm{X} = x] - 1/|\mathcal{X}||/(2\varepsilon)$, on a $\sum_x q_x = 1$. D'après l'équation 10.2 ,

$$\begin{aligned}\tfrac{1}{|\mathcal{X}|} &\leqslant \sum_{x\in\mathcal{X}} q_x^2 = \sum_{x\in\mathcal{X}} \tfrac{(\Pr[\mathrm{X}=x]-1/|\mathcal{X}|)^2}{4\varepsilon^2} = \tfrac{1}{4\varepsilon^2}\left(\sum_{x\in\mathcal{X}} \Pr[\mathrm{X} = x]^2 - 1/|\mathcal{X}|\right)\\ &\leqslant \tfrac{1}{4\varepsilon^2}\left(\mathrm{Col}(\mathrm{X}) - 1/|\mathcal{X}|\right)\end{aligned}$$

Le lemme peut en être déduit facilement. □

10.1.2 De la min-entropie à la δ-uniformité

La méthode la plus classique pour obtenir une source δ-uniforme est d'extraire de l'aléa à partir de sources de chaînes de bits de grandes entropies, en utilisant un *extracteur d'aléa*. Le plus célèbre d'entre eux est très certainement celui donné par le *Leftover Hash Lemma* [HILL99, IZ89] (LHL), qui requiert l'utilisation de familles de fonctions de hachage universelles (voir page 21).

Soient $(h_i)_{i\in\{0,1\}^d}$ une famille de fonctions de hachage universelles, i une variable aléatoire uniformément distribuée sur $\{0,1\}^d$, U_k une variable aléatoire uniformément distribuée sur $\{0,1\}^k$, et X une variable aléatoire à valeurs dans $\{0,1\}^n$, avec i et X mutuellement indépendantes et telle que la min-entropie de X soit plus grande que m, c'est-à-dire $\mathbf{H}_\infty(\mathrm{X}) \geqslant m$. Le LHL [HILL99, IZ89, Sho05] assure que $\mathbf{SD}(\langle i, h_i(\mathrm{X})\rangle, \langle i, \mathrm{U}_k\rangle) \leqslant 2^{(k-m)/2-1}$.

En d'autres termes, si l'on veut extraire de l'entropie à partir de la variable aléatoire X, on génère une variable aléatoire *i uniformément distribuée* et on calcule $h_i(\mathrm{X})$. Le LHL garantit une sécurité en 2^{-e}, si l'on impose que

$$k \leqslant m - 2e + 2 \tag{10.4}$$

Le LHL extrait quasiment toute l'entropie disponible *quelles que soient les sources d'aléa*, mais il a besoin d'investir quelques bits additionnels vraiment aléatoires. Pour surmonter ce problème, il a été proposé d'utiliser des fonctions déterministes. Elles n'ont pas besoin de bits aléatoires supplémentaires, mais n'existent que pour certaines sources spécifiques d'aléa.

Définition 4 (Extracteur déterministe)**.** *Soit f une fonction de $\{0,1\}^n$ dans $\{0,1\}^k$. Soit* X *une variable aléatoire à valeur dans $\{0,1\}^n$ et soit U_k une variable aléatoire uniformément distribuée sur $\{0,1\}^k$, où U_k et* X *sont indépendants. f est un extracteur (X,ε)-déterministe si*

$$\mathbf{SD}\left(f(\mathrm{X}), \mathrm{U}_k\right) < \varepsilon$$

10.1.3 Sommes de caractères sur les groupes abéliens et inégalité de Polya–Vinogradov

Rappelons un lemme standard sur les groupes de caractères des groupes abéliens.

Lemme 4. *Soient* H *un groupe abélien et $\widehat{\mathrm{H}} = \mathrm{Hom}(\mathrm{H}, \mathbb{C}^*)$ son groupe dual. Alors, pour tout élément χ de $\widehat{\mathrm{H}}$, on a l'égalité suivante, dans laquelle χ_0 est le caractère trivial :*

$$\frac{1}{|\mathrm{H}|}\sum_{h\in\mathrm{H}} \chi(h) = \begin{cases} 1 & \text{si } \chi = \chi_0 \\ 0 & \text{si } \chi \neq \chi_0 \end{cases}$$

Nous donnons dans cette section une preuve de l'inégalité de Polya–Vinogradov et introduisons tous les résultats basiques à propos des sommes de caractères requis dans la section suivante.

Rappelons d'abord les notations. Soient p un nombre premier, G un sous-groupe de $(\mathbb{Z}_p^*, \cdot)$ d'ordre q, et g un élément qui génère G.

Un caractère est un homomorphisme de $(\mathbb{Z}_p, +)$ dans $(\mathbb{C}^*, \cdot)$. Dans cette section, nous ne nous intéressons qu'au caractère suivant, noté e_p : pour tout $y \in \mathbb{Z}_p$, $e_p(y) = e^{\frac{2i\pi y}{p}} \in \mathbb{C}^*$. Plus précisément, on se focalise sur des sommes particulières : $\sum_{x\in \mathrm{G}} e_p(ax)$. Pour tout $a \in \mathbb{Z}^*$, nous introduisons la notation

$$\mathrm{S}(a, \mathrm{G}) = \sum_{x\in \mathrm{G}} e_p(ax)$$

Lemme 5. *Soient p un nombre premier et* G *un sous-groupe de $(\mathbb{Z}_p^*, \cdot)$. Alors,*

(1) si $a = 0$, $\sum_{x=0}^{p-1} e_p(ax) = \sum_{x\in\mathbb{Z}_p} e_p(ax) = p$;

(2) pour tout $a \in \mathbb{Z}_p^*$, $\sum_{x=0}^{p-1} e_p(ax) = \sum_{x\in\mathbb{Z}_p} e_p(ax) = 0$;

(3) pour tout $x_0 \in \mathrm{G}$ et tout $a \in \mathbb{Z}_p^*$, $\mathrm{S}(ax_0, \mathrm{G}) = \mathrm{S}(a, \mathrm{G})$.

Démonstration. Le point (1) est immédiat puisque $e_p(0) = 1$.

Si $a \neq 0$, la somme $\sum_{x=0}^{p-1} e_p(ax)$ est la somme d'une suite géométrique de raison $e_p(a) \neq 1$, donc elle est égale à $(1 - e_p(ap))/(1 - e_p(a)) = 0$, ce qui prouve le point (2).

Finalement, comme la fonction de G dans G reliant chaque x à $x \cdot x_0$ est bijective, le changement de variable $x' = x \cdot x_0$ permet de prouver que $\sum_{x\in \mathrm{G}} e_p(ax \cdot x_0) = \sum_{x'\in \mathrm{G}} e_p(ax')$, ce qui est le point (3). □

Notons $\mathbb{Z}_p^*/\mathrm{G}$ le groupe-quotient de G dans $\mathbb{Z}_p^*$. Le point (3) du lemme 5 implique que pour toute classe $\omega \in \mathbb{Z}_p^*/\mathrm{G}$, $\mathrm{S}(\omega, \mathrm{G})$ est bien définie : $\mathrm{S}(\omega, \mathrm{G}) = \mathrm{S}(x_0, \mathrm{G})$, où x_0 est un représentant quelconque de la classe ω. Les résultats précédents permettent de prouver le lemme suivant.

Lemme 6. *Soit* G *un sous-groupe de $\mathbb{Z}_p^*$. On a alors l'égalité suivante :*

$$\sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}} |\mathrm{S}(\omega, \mathrm{G})|^2 = p - |\mathrm{G}|$$

Démonstration. Montrons que $\sum_{a\in\mathbb{Z}_p} |\mathrm{S}(a, \mathrm{G})|^2 = p|\mathrm{G}|$. En effet, puisque $\overline{\mathrm{S}(a, \mathrm{G})} = \mathrm{S}(-a, \mathrm{G})$,

$$\begin{aligned}
\sum_{a\in\mathbb{Z}_p} |\mathrm{S}(a, \mathrm{G})|^2 &= \sum_{a\in\mathbb{Z}_p} \mathrm{S}(a, \mathrm{G})\mathrm{S}(-a, \mathrm{G}) \\
&= \sum_{a\in\mathbb{Z}_p} \sum_{x_1\in\mathrm{G}} e_p(ax_1) \sum_{x_2\in\mathrm{G}} e_p(-ax_2) \\
&= \sum_{a\in\mathbb{Z}_p} \sum_{x_1,x_2\in\mathrm{G}} e_p(a(x_1 - x_2)) \\
&= \sum_{x_1,x_2\in\mathrm{G}} \sum_{a\in\mathbb{Z}_p} e_p(a(x_1 - x_2)) \\
&= \sum_{x_1=x_2\in\mathrm{G}} p \\
\sum_{a\in\mathbb{Z}_p} |\mathrm{S}(a, \mathrm{G})|^2 &= p|\mathrm{G}|
\end{aligned}$$

l'avant-dernière égalité provenant des points (1) et (2) du lemme 5. Ce résultat implique alors que $\sum_{a\in\mathbb{Z}_p^*} |\mathrm{S}(a, \mathrm{G})|^2 = \sum_{a\in\mathbb{Z}_p} |\mathrm{S}(a, \mathrm{G})|^2 - |\mathrm{S}(0, \mathrm{G})|^2 = p|\mathrm{G}| - |\mathrm{G}|^2$. En outre, on a

$$\sum_{a\in\mathbb{Z}_p^*} |\mathrm{S}(a, \mathrm{G})|^2 = \sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}} \sum_{a\in\omega} |\mathrm{S}(a, \mathrm{G})|^2 = \sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}} \sum_{a\in\omega} |\mathrm{S}(\omega, \mathrm{G})|^2 = \sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}} |\mathrm{S}(\omega, \mathrm{G})|^2 \, |\mathrm{G}|$$

d'où l'on peut déduire le lemme aisément. □

L'inégalité de Polya–Vinogradov est une conséquence facile de ces résultats. Elle donne une majoration de $|\mathrm{S}(a, \mathrm{G})|$, pour $a \neq 0$, et a été montrée par Polya et Vinogradov indépendamment.

Théorème 4 (Inégalité de Polya–Vinogradov)**.** *Soient p un nombre premier,* G *un sous-groupe de $(\mathbb{Z}_p^*, \cdot)$. Pour tout $a \in \mathbb{Z}_p^*$,*

$$\left|\sum_{x \in \mathrm{G}} e_p(ax)\right| \leqslant \sqrt{p}$$

Démonstration. D'après le lemme précédent, on sait que pour tout $a \in \mathbb{Z}_p^*$, si $a \in \omega$ avec $\omega \in \mathbb{Z}_p^*/\mathrm{G}$, on a $|\mathrm{S}(a, \mathrm{G})|^2 = |\mathrm{S}(\omega, \mathrm{G})|^2 \leqslant \sum_{\omega \in \mathbb{Z}_p^*/\mathrm{G}} |\mathrm{S}(\omega, \mathrm{G})|^2$, qui est plus petit que p. Ainsi, $|\mathrm{S}(a, \mathrm{G})|^2$ est plus petit que p, ce qui est le résultat escompté. □

Cette majoration est non-triviale dès que $\sqrt{p} \leqslant |\mathrm{G}|$. Plus de résultats sur les sommes de caractères peuvent être trouvés dans [KS99].

10.1.4 Courbes elliptiques

Soient p un nombre premier et $\mathcal{E}$ une courbe elliptique sur $\mathbb{F}_p$ donnée par l'équation de Weierstrass

$$y^2 + (a_1 x + a_3) \cdot y = x^3 + a_2 x^2 + a_4 x + a_6$$

Nous notons $\mathcal{E}(\mathbb{F}_p)$ le groupe des éléments de $\mathcal{E}$ sur $\mathbb{F}_p$ et $\mathbb{F}_p(\mathcal{E})$ le corps de fonctions de la courbe $\mathcal{E}$, défini comme le corps des fractions sur les points de $\mathcal{E}$: $\mathbb{F}_p(\mathcal{E}) = \mathbb{F}_p[\mathrm{X}, \mathrm{Y}]/\mathcal{E}(\mathbb{F}_p)$. Il est généré par les fonctions x et y, satisfaisant l'équation de Weierstrass de $\mathcal{E}$, et telles que $\mathrm{P} = (x(\mathrm{P}), y(\mathrm{P}))$ pour tout point $\mathrm{P} \in \mathcal{E}(\mathbb{F}_p) \setminus \{\mathrm{O}\}$. Soit $f_a \in \mathbb{F}_p(\mathcal{E})$ l'application $f_a = a \cdot x$ où $a \in \mathbb{Z}_p^*$. Si $f \in \mathbb{F}_p(\mathcal{E})$, on note $\deg(f)$ son degré, c'est-à-dire $\sum_{i=1}^{t} n_i \deg(\mathrm{P}_i)$ si $\sum_{i=1}^{t} n_i \mathrm{P}_i$ est le diviseur des pôles de f. Finalement, on note $\Omega = \mathrm{Hom}(\mathcal{E}(\mathbb{F}_p), \mathbb{C}^*)$ le groupe des caractères sur $\mathcal{E}(\mathbb{F}_p)$, et ω_0 le caractère trivial (tel que $\omega_0(\mathrm{P}) = 1$ pour tout P).

10.2 Extraction d'aléa dans les corps finis

Dans cette section, nous étendons les résultats de Fouque *et al.* [FPSZ06], afin d'extraire des bits d'éléments aléatoires d'un sous-groupe multiplicatif d'un corps fini. La même technique permet ensuite d'améliorer le résultat de Canetti *et al.* [CFK+00], mais nous ne détaillons pas ce point ici.

Nous étudions désormais l'extracteur d'aléa qui consiste à conserver les bits de poids faible d'un élément aléatoire d'un sous-groupe G de $\mathbb{Z}_p^*$. La technique de preuve présentée ici nous permet d'extraire deux fois plus de bits que dans Fouque *et al.*. Dans le cas particulier $q \geqslant p^{3/4}$, où q est le cardinal de G, on prouve même un meilleur résultat : on peut extraire autant de bits qu'avec le LHL. Ceci signifie que, dans le cas où $q \geqslant p^{3/4}$, notre extracteur est aussi bon que le LHL, mais calculatoirement plus efficace et plus facile à utiliser dans les protocoles, puisqu'il ne requiert pas d'élément public parfaitement aléatoire supplémentaire.

Dans le papier original, Fouque *et al.* majorent directement la distance statistique entre les bits extraits et la distribution uniforme, en utilisant des sommes d'exponentielles. Nous les utilisons toujours ici, mais proposons d'appliquer cette technique pour majorer la probabilité de collision des bits extraits. L'inégalité de Cauchy-Schwarz permet de relier la distance statistique et la probabilité de collision et de conclure. Comme la distribution des bits extraits est très proche de la distribution uniforme, l'inégalité de Cauchy-Schwarz est très fine. C'est la raison pour laquelle nous ne perdons pas beaucoup avec ce détour. Au contraire, nous sommes capables de trouver une bonne majoration pour la résistance aux collisions, et donc la majoration globale est améliorée.

Le résultat dans le cas où $q \geqslant p^{3/4}$ est élaboré à partir de la même idée de base, mais requiert des techniques plus élaborées sur les sommes d'exponentielles pour être établi.

Théorème 5. *Soient p un nombre premier à n bits, G un sous-groupe de $\mathbb{Z}_p^*$ de cardinal q (on appelle $\ell = \log_2(q) \in \mathbb{R}$), $\mathrm{U_G}$ une variable aléatoire uniformément ditribuée sur G et k un entier positif. Alors*

$$\mathbf{SD}(\mathsf{lsb}_k(\mathrm{U_G}), \mathrm{U}_k) \leqslant \begin{cases} 2^{3n/4-\ell-1} + 2^{(k-\ell)/2} & \left(\text{si } p^{3/4} \leqslant q\right) \\ 2^{(k+n+\log_2 n)/2-\ell} & \left(\text{si } (2^{-8}p)^{2/3} \leqslant q \leqslant p^{3/4}\right) \\ 2^{(k+n/2+\log_2 n+4)/2-5\ell/8} & \left(\text{si } p^{1/2} \leqslant q \leqslant (2^{-8}p)^{2/3}\right) \\ 2^{(k+n/4+\log_2 n+4)/2-3\ell/8} & \left(\text{si } (2^{16}p)^{1/3} \leqslant q \leqslant p^{1/2}\right) \end{cases}$$

Rappelons que ces inégalités sont non-triviales seulement si elles sont plus petites que 1.

Démonstration. Posons $\mathrm{K} = 2^k$ et $u_0 = \mathsf{msb}_{n-k}(p-1)$. Les deux principaux intérêts des sommes d'exponentielles sont qu'elles permettent de construire des fonctions caractéristiques et que dans certains cas nous en connaissons de bonnes majorations. Grâce à ces fonctions caractéristiques, on peut évaluer la taille de certains ensembles et, en manipulant des sommes, on peut majorer la taille de ces ensembles.

Dans notre cas, on construit $\mathbb{1}(x,y,u) = \frac{1}{p} \times \sum_{a=0}^{p-1} e_p(a(g^x - g^y - \mathrm{K}u))$, où $\mathbb{1}(x,y,u)$ est la fonction caractéristique égale à 1 si $g^x - g^y = \mathrm{K}u \bmod p$ et 0 sinon. Ainsi, on peut évaluer $\mathrm{Col}(\mathsf{lsb}_k(\mathrm{U_G}))$ où $\mathrm{U_G}$ est uniformément distribuée dans G :

$$\begin{aligned} \mathrm{Col}(\mathsf{lsb}_k(\mathrm{U_G})) &= \tfrac{1}{q^2} \times \left| \{(x,y) \in [\![\, 0\,;\, q-1 \,]\!]^2 \,|\, \exists u \leqslant u_0,\, g^x - g^y = \mathrm{K}u \bmod p\} \right| \\ &= \tfrac{1}{q^2 p} \times \sum_{x=0}^{q-1} \sum_{y=0}^{q-1} \sum_{u=0}^{u_0} \sum_{a=0}^{p-1} e_p(a(g^x - g^y - \mathrm{K}u)) \end{aligned}$$

Changeons l'ordre des sommes, et découpons la somme sur les a en deux termes :

1. le premier provient du cas $a = 0$, et il est égal à $(u_0+1)/p$, c'est-à-dire environ à $1/2^k$;
2. le deuxième provient du reste, et va être le terme principale dans la distance statistique dans laquelle on peut séparer les sommes sur x et sur u.

L'expression précédente est alors égale à

$$\begin{aligned} \mathrm{Col}(\mathsf{lsb}_k(\mathrm{U_G})) &= \frac{u_0+1}{p} + \frac{1}{q^2 p} \sum_{a=1}^{p-1} \left(\sum_{x=0}^{q-1} e_p(ag^x) \right) \left(\sum_{y=0}^{q-1} e_p(-ag^y) \right) \left(\sum_{u=0}^{u_0} e_p(-a\mathrm{K}u) \right) \\ &= \frac{u_0+1}{p} + \frac{1}{q^2 p} \sum_{a=1}^{p-1} \mathrm{S}(a,\mathrm{G})\mathrm{S}(-a,\mathrm{G}) \left(\sum_{u=0}^{u_0} e_p(-a\mathrm{K}u) \right) \\ \mathrm{Col}(\mathsf{lsb}_k(\mathrm{U_G})) &= \frac{u_0+1}{p} + \frac{1}{q^2 p} \sum_{a=1}^{p-1} |\mathrm{S}(a,\mathrm{G})|^2 \left(\sum_{u=0}^{u_0} e_p(-a\mathrm{K}u) \right) \end{aligned} \tag{10.5}$$

d'où

$$\begin{aligned} \mathrm{Col}(\mathsf{lsb}_k(\mathrm{U_G})) &\leqslant \tfrac{u_0+1}{p} + \tfrac{1}{q^2 p} \sum_{a=1}^{p-1} |\mathrm{S}(a,\mathrm{G})|^2 \left| \sum_{u=0}^{u_0} e_p(-a\mathrm{K}u) \right| && \text{(valeur absolue)} \\ &\leqslant \tfrac{u_0+1}{p} + \tfrac{\mathrm{M}^2}{q^2 p} \sum_{a=1}^{p-1} \left| \sum_{u=0}^{u_0} e_p(-au) \right| && \text{avec } \mathrm{M} = \max_a(|\mathrm{S}(a,\mathrm{G})|) \end{aligned}$$

Les trois dernières bornes. À partir de ce point, la preuve des trois dernières inéquations est différente de celle de la première inéquation.

$$\begin{aligned} \sum_{a=1}^{p-1} \left| \sum_{u=0}^{u_0} e_p(-a\mathrm{K}u) \right| &= \sum_{a=1}^{p-1} \left| \sum_{u=0}^{u_0} e_p(-au) \right| && \text{(changement de variables)} \\ &= \sum_{a=1}^{p-1} \left| \tfrac{1-e_p(-a(u_0+1))}{1-e_p(-a)} \right| && \text{(suite géométrique)} \\ &= \sum_{a=1}^{p-1} \left| \tfrac{\sin(\frac{\pi a(u_0+1)}{p})}{\sin(\frac{\pi a}{p})} \right| \end{aligned}$$

$$\begin{aligned}\sum_{a=1}^{p-1}\left|\sum_{u=0}^{u_0} e_p(-a\mathrm{K}u)\right| &= 2\sum_{a=1}^{\frac{p-1}{2}}\left|\frac{\sin(\frac{\pi a(u_0+1)}{p})}{\sin(\frac{\pi a}{p})}\right| \\ &\leqslant 2\sum_{a=1}^{\frac{p-1}{2}}\left|\frac{1}{\sin(\frac{\pi a}{p})}\right| \\ &\leqslant \sum_{a=1}^{\frac{p-1}{2}}\left|\tfrac{p}{a}\right| \leqslant p\log_2(p)\end{aligned}$$

car $\sin(y) \geqslant 2y/\pi$ si $0 \leqslant y \leqslant \pi/2$. La probabilité de collision est alors majorée par

$$\mathrm{Col}(\mathsf{lsb}_k(\mathrm{X})) \leqslant \tfrac{u_0+1}{p} + \tfrac{\mathrm{M}^2\log_2(p)}{q^2}$$

Utilisons alors le lemme 2 pour obtenir une relation entre la distance statistique ε entre $\mathsf{lsb}_k(\mathrm{X})$ et la distribution uniforme et la probabilité de collision : $\mathrm{Col}(\mathsf{lsb}_k(\mathrm{U_G})) \geqslant \frac{1+4\varepsilon^2}{2^k}$. La majoration précédente donne alors

$$4\varepsilon^2 \leqslant \frac{2^k(u_0+1)}{p} - 1 + \frac{2^k\mathrm{M}^2\log_2(p)}{q^2}$$

Remarquons alors que $\left|(u_0+1)/p - 1/2^k\right| \leqslant 1/p$, puisque $\mathrm{K} = 2^k$, $u_0 = \left\lfloor \frac{p-1}{\mathrm{K}} \right\rfloor$ et :

$$-\frac{1}{p} \leqslant -\frac{1}{\mathrm{K}p} \leqslant \left(1 + \left\lfloor\frac{p-1}{\mathrm{K}}\right\rfloor\right)\frac{1}{p} - \frac{1}{\mathrm{K}} \leqslant \frac{\mathrm{K}-1}{\mathrm{K}p} \leqslant \frac{1}{p}$$

En résumé,

$$2\varepsilon \leqslant \sqrt{\frac{2^k}{p} + \frac{2^k\mathrm{M}^2\log_2(p)}{q^2}} \tag{10.6}$$

On conclut la preuve en utilisant les majorations suivantes pour M :

$$\mathrm{M} \leqslant \begin{cases} p^{1/2} & \left(\text{intéressant si } p^{2/3} \leqslant q\right) \\ 4p^{1/4}q^{3/8} & \left(\text{intéressant si } p^{1/2} \leqslant q \leqslant p^{2/3}\right) \\ 4p^{1/8}q^{5/8} & \left(\text{intéressant si } 2^{16/3}p^{1/3} \leqslant q \leqslant p^{1/2}\right) \end{cases}$$

La première borne est celle de Polya–Vinogradov, et les autres proviennent de [KS99, HBK00]. Elles sont asymptotiques dans les papiers d'origine, et les constantes impliquées ne sont pas explicites. Cependant, on peut les trouver en regardant avec attention les preuves dans [HBK00] et [KS99]. Ces bornes ont déjà été utilisées dans [FPSZ06]. À l'aide des inégalités $2^{n-1} < p < 2^n$, $2^{\ell-1} \leqslant q < 2^\ell$, et $\sqrt{a+b} \leqslant \sqrt{a}+\sqrt{b}$, où $a, b \geqslant 0$, on obtient le résultat suivant :

$$2\varepsilon \leqslant \begin{cases} 2^{(k-n+1)/2} + 2^{(k+n+\log_2 n)/2-\ell} & \left(\text{si } p^{2/3} \leqslant q\right) \\ 2^{(k-n+1)/2} + 2^{(k+n/2+\log_2 n+4)/2-5\ell/8} & \left(\text{si } p^{1/2} \leqslant q \leqslant p^{2/3}\right) \\ 2^{(k-n+1)/2} + 2^{(k+n/4+\log_2 n+4)/2-3\ell/8} & \left(\text{si } p^{1/3} \leqslant q \leqslant p^{1/2}\right) \end{cases}$$

La première borne. Pour établir la première inégalité, utilisons l'équation 10.5 qui donne

$$\begin{aligned}\mathrm{Col}(\mathsf{lsb}_k(\mathrm{X})) &= \frac{u_0+1}{p} + \frac{1}{q^2p}\sum_{a=1}^{p-1}|\mathrm{S}(a,\mathrm{G})|^2\left(\sum_{u=0}^{u_0} e_p(-a\mathrm{K}u)\right) \\ &= \frac{u_0+1}{p} + \frac{1}{q^2p}\sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}}|\mathrm{S}(\omega,\mathrm{G})|^2\sum_{a\in\omega}\sum_{u=0}^{u_0} e_p(-a\mathrm{K}u) \\ &= \frac{u_0+1}{p} + \frac{1}{q^2p}\sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}}|\mathrm{S}(\omega,\mathrm{G})|^2\sum_{u=0}^{u_0}\sum_{a\in\omega} e_p(-a\mathrm{K}u)\end{aligned}$$

$$\begin{aligned}\mathrm{Col}(\mathsf{lsb}_k(\mathrm{X})) &= \frac{u_0+1}{p} + \frac{1}{q^2 p}\sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}} |\mathrm{S}(\omega,\mathrm{G})|^2 \sum_{u=0}^{u_0} \mathrm{S}(-\omega \mathrm{K} u, \mathrm{G}) \\ &\leqslant \frac{u_0+1}{p} + \frac{1}{q^2 p}\sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}} |\mathrm{S}(\omega,\mathrm{G})|^2 \left(\sum_{u=1}^{u_0} |\mathrm{S}(-\omega \mathrm{K} u, \mathrm{G})| + q\right)\end{aligned}$$

Utilisons alors l'inégalité de Polya–Vinogradov combinée avec l'inégalité $\sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}} |\mathrm{S}(\omega,\mathrm{G})|^2 \leqslant p$ pour montrer que :

$$\begin{aligned}\mathrm{Col}(\mathsf{lsb}_k(\mathrm{X})) &\leqslant \frac{u_0+1}{p} + \frac{u_0\sqrt{p}+q}{q^2 p}\sum_{\omega\in\mathbb{Z}_p^*/\mathrm{G}} |\mathrm{S}(\omega,\mathrm{G})|^2 \\ &\leqslant \frac{u_0+1}{p} + \frac{u_0\sqrt{p}+q}{q^2 p}\cdot p \\ \mathrm{Col}(\mathsf{lsb}_k(\mathrm{X})) &\leqslant \frac{u_0+1}{p} + \frac{u_0\sqrt{p}+q}{q^2}\end{aligned}$$

Terminons comme précédemment pour obtenir, puisque $u_0 \leqslant 2^{n-k}$:

$$2\varepsilon \leqslant \sqrt{\frac{2^k}{p} + \frac{2^n\sqrt{p}+2^k q}{q^2}} \tag{10.7}$$

puis $$2\varepsilon \leqslant 2^{(k-n+1)/2} + 2^{3n/4-\ell} + 2^{(k-\ell)/2}$$

Comme $\ell \leqslant n-1$, ceci donne la majoration espérée. □

Comme la min-entropie de $\mathrm{U_G}$, vue comme un élément de $\mathbb{Z}_p^*$ uniformément distribué dans G, est égale à $\ell = \log_2(|\mathrm{G}|) = \log_2(q)$, la proposition précédente mène à :

Corollaire 5.1. *Soit e un entier positif et supposons que l'une de ces inéquations est vraie :*

$$k \leqslant \begin{cases} \ell - (2e+2) & \text{et } 2^e \cdot p^{3/4} \leqslant q \\ 2\ell - (n + 2e + \log_2(n)) & \text{et } (2^{-8}\cdot p)^{2/3} \leqslant q \leqslant 2^e \cdot p^{3/4} \\ 5\ell/4 - (n/2 + 2e + \log_2(n) + 4) & \text{et } p^{1/2} \leqslant q \leqslant (2^{-8}\cdot p)^{2/3} \\ 3\ell/4 - (n/4 + 2e + \log_2(n) + 4) & \text{et } (2^{16}\cdot p)^{1/3} \leqslant q \leqslant p^{1/2} \end{cases}$$

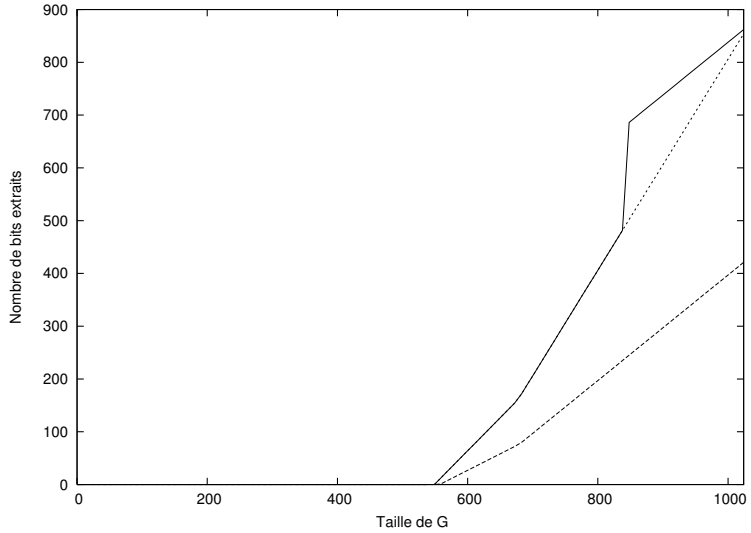
Alors l'application Ext_k est un extracteur $(\mathrm{U_G}, 2^{-e})$-déterministe.

Ceci signifie que si l'on souhaite une sécurité en 2^{-e}, et si $(2^{-8}\cdot p)^{2/3} \leqslant q \leqslant 2^e \cdot p^{3/4}$, on peut extraire k bits avec $k \leqslant 2(\ell - (n/2 + e + \log_2 n/2))$.

La plupart du temps, en pratique, la deuxième borne est la plus adéquate. Cependant, il s'agit parfois de l'une des autres. Par exemple, avec $n = 1024$, $\ell = 600$ et $e = 80$, la deuxième borne assure que l'on peut extraire 6 bits, tandis que la troisième borne montre que l'on peut en fait en extraire 64.

Si l'on souhaite extraire une chaîne de 256 bits, pour les mêmes valeurs de n et de e, on a besoin d'un groupe de taille plus grande que 2^{756}. La figure 10.1 présente nos majorations ainsi que celles originales de Fouque *et al.* [FPSZ06], dans le cas où $n = 1024$ et $e = 80$.

Fig. 10.1 – Nombre de bits extraits en fonction de la taille du groupe, pour $n = 1024$ et $e = 80$



La longue ligne pointillée représente les résultats de Fouque *et al.* [FPSZ06] et la ligne pleine les nôtres. Notons le saut pour $q = p^{3/4}$. La petite ligne pointillée représente nos résultats sans améliorations particulières dans le cas où $q \geqslant p^{3/4}$.

10.3 Extraction d'aléa dans les courbes elliptiques

Nous montrons désormais comment l'extracteur d'aléa étudié dans la section précédente, qui consistait à conserver les bits de poids faible d'un élément aléatoire d'un sous-groupe G de $\mathbb{Z}_p^*$, peut être étendu à une autre structure, en l'occurrence le groupe des éléments d'une courbe elliptique. L'idée principale est d'évaluer l'élément « M » de la preuve du théorème 5, qui est une majoration de $\mathrm{S}(a, \mathrm{G})$, tel que défini dans la section précédente.

10.3.1 Une borne pour $\mathrm{S}(a, \mathrm{G})$

Si G est un sous-groupe de $\mathcal{E}(\mathbb{F}_p)$, $f \in \mathbb{F}_p(\mathcal{E})$ et $\omega \in \Omega$, on définit

$$\mathrm{S}(\omega, f, \mathrm{G}) = \sum_{\mathrm{P} \in \mathrm{G}} \omega(\mathrm{P}) e_p(f(\mathrm{P}))$$

En particulier, puisque $e_p \in \Omega$,

$$\mathrm{S}(a, \mathrm{G}) = \mathrm{S}(\omega_0, f_a, \mathrm{G}) = \sum_{\mathrm{P} \in \mathrm{G}} e_p(f_a(\mathrm{P}))$$

L'objectif de cette section est de montrer le résultat suivant :

Théorème 6. *Soient $\mathcal{E}$ une courbe elliptique définie sur $\mathbb{F}_p$ et $f \in \mathbb{F}_p(\mathcal{E})$. Alors,*

$$\mathrm{S}(\omega, f, \mathcal{E}(\mathbb{F}_p)) \leqslant 2 \deg(f) \sqrt{p}$$

Par conséquent, si $a \in \mathbb{Z}^*$, $\mathrm{S}(a, \mathcal{E}(\mathbb{F}_p)) \leqslant 4\sqrt{p}$

Plus spécifiquement, dans le cas où le cofacteur n'est pas égal à 1, nous sommes intéressés par son corollaire. Notons que dans le cas de $\mathbb{F}_p$, toutes les courbes recommandées par le NIST ont un cofacteur égal à 1.

Corollaire 6.1. *Soient $\mathcal{E}$ une courbe elliptique sur $\mathbb{F}_p$, $a \in \mathbb{Z}^*$ et* G *un sous-groupe de $\mathcal{E}(\mathbb{F}_p)$. Alors,*

$$\mathrm{S}(\omega, f, \mathrm{G}) \leqslant 2 \deg(f) \sqrt{p} \qquad \textit{et} \qquad \mathrm{S}(a, \mathrm{G}) \leqslant 4\sqrt{p}$$

Démonstration. Notons d'abord que nous utilisons la version complète du théorème 6, qui implique que le résultat est aussi valide pour la somme $\mathrm{S}(\omega, a, \mathcal{E}(\mathbb{F}_p)) = \sum_{x \in \mathcal{C}(\mathbb{Z}_p)} \omega(x) e_p(f_a(x))$, où $\omega \in \Omega$ (la preuve est identique).

Notons $\Omega_\mathrm{G} \subset \Omega$ l'ensemble des caractères $\chi \in \Omega$ tels que $\mathrm{Ker}\,(\chi)$ contienne G. Puisque Ω_G est le dual de $\mathcal{E}(\mathbb{F}_p)/\mathrm{G}$, on peut appliquer le lemme ci-dessus et obtenir les égalités suivantes :

$$\begin{aligned}
\mathrm{S}(f, \mathrm{G}) &= \sum_{x \in \mathrm{G}} e_p(f(x)) \\
&= \sum_{x \in \mathrm{G}} \left(\frac{1}{|\Omega_\mathrm{G}|} \sum_{\chi \in \Omega_\mathrm{G}} \chi(x) \right) e_p(f(x)) \\
\mathrm{S}(f, \mathrm{G}) &= \frac{1}{|\Omega_\mathrm{G}|} \sum_{\chi \in \Omega_\mathrm{G}} \mathrm{S}(\chi, f, \mathcal{E}(\mathbb{F}_p))
\end{aligned}$$

à partir desquelles le résultat suit en appliquant le théorème 6 à $\mathrm{S}(\chi, f, \mathcal{E}(\mathbb{F}_p))$. □

Donnons désormais un cas particulier de la conjecture d'Artin et de l'hypothèse de Riemann pour les corps de fonctions, pour une courbe elliptique et le caractère trivial. Ces formes sont utilisées dans la preuve du théorème 6 :

Théorème 7 (Conjecture d'Artin)**.** *Soient $\mathcal{E}$ une courbe elliptique et $f \in \mathbb{F}_p(\mathcal{E})$. La fonction* L *définie par $t \mapsto \mathrm{L}(t, f, \mathcal{E}(\mathbb{F}_p)) = \exp\left(\sum_{m=1}^{+\infty} \mathrm{S}(f, \mathcal{E}(\mathbb{F}_p)) t^m / m \right)$ est un polynôme et son degré vaut $\mathrm{D} = \deg(f)$.*

Théorème 8 (Hypothèse de Riemann)**.** *Soient $\mathcal{E}$ une courbe elliptique et $f \in \mathbb{F}_p(\mathcal{E})$. Les zéros de la fonction* L

$$t \mapsto \mathrm{L}(t, f, \mathcal{E}(\mathbb{F}_p)) = \exp\left(\sum_{m=1}^{+\infty} \mathrm{S}(f, \mathcal{E}(\mathbb{F}_p)) t^m / m \right)$$

ont pour module $1/\sqrt{q}$.

Démonstration (du théorème 6). Par souci de simplicité, nous ne considérons que le cas dans lequel $\omega = \omega_0$ afin d'utiliser des notations plus simples. Nous suivons la preuve de Bombieri dans [Bom66] et celle de Kohel et Shparlinski dans [KS00], en considérant tout d'abord la somme $\mathrm{S}_m(f, \mathcal{E}(\mathbb{F}_p)) = \mathrm{S}(\sigma \circ f, \mathcal{E}(\mathbb{F}_{p^m}))$ où σ est la trace de $\mathbb{F}_{p^m}$ dans $\mathbb{F}_p$. Notons que la somme qui nous intéresse correspond à $m = 1$.

Cette somme provient du caractère $e_p \circ f$, qui définit une extension d'Artin-Schreier (informellement, une extension de degré p) du corps de fonctions $\mathbb{F}_p(\mathcal{E})$, et ensuite un revêtement d'Artin-Schreier de $\mathcal{E}(\mathbb{F}_p)$. Un moyen facile d'évaluer cette somme est de considérer cette fonction L reliée à ce revêtement d'Artin-Schreier. Les fonctions L sont un moyen standard pour assembler divers éléments dans un objet unique (une série), de la même manière qu'une série génératrice, voir par exemple [Was03, chap. 14]. Bombieri a montré que cette fonction L est définie comme suit, pour $t \in \mathbb{C}$ tel que $|t| < q^{-1}$:

$$\mathrm{L}(t, f, \mathcal{E}(\mathbb{F}_p)) = \exp\left(\sum_{m=1}^{+\infty} \mathrm{S}(f, \mathcal{E}(\mathbb{F}_p)) t^m / m \right)$$

D'après la conjecture d'Artin, dont la preuve a été donnée par Weil dans [Wei48], cette fonction est un polynôme de degré $\mathrm{D} = \deg(f)$. Notons ses D racines complexes (non nécessairement distinctes) $\theta_i = {\omega_i}^{-1}$.

On a alors les deux équations suivantes :

$$\mathrm{L}(t, f, \mathcal{E}(\mathbb{F}_p)) = \sum_{i=0}^{+\infty} \frac{1}{i!} \left(\sum_{m=1}^{+\infty} \mathrm{S}_m(f, \mathcal{E}(\mathbb{F}_p)) t^m / m \right)^i$$

$$\mathrm{L}(t, f, \mathcal{E}(\mathbb{F}_p)) = \prod_{i=1}^{\mathrm{D}} (1 - \omega_i t)$$

La première équation peut être réécrite de la façon suivante :

$$1 + \sum_{m=1}^{+\infty} \mathrm{S}_m(f, \mathcal{E}(\mathbb{F}_p))t^m/m + \frac{1}{2}\sum_{m=1}^{+\infty}\sum_{n=1}^{+\infty} \frac{\mathrm{S}_m(f, \mathcal{E}(\mathbb{F}_p))\mathrm{S}_n(f, \mathcal{E}(\mathbb{F}_p))}{m\ n} t^{m+n} + \dots$$

Si l'on considère le coefficient d'ordre 1 du polynôme, on obtient :

$$\mathrm{S}_1(f, \mathcal{E}(\mathbb{F}_p)) = -\sum_{i=1}^{\mathrm{D}} \omega_i$$

L'hypothèse de Riemann pour les corps de fonctions (voir [Wei48] pour la preuve) montre que chaque zéro de la fonction L vérifie $|\theta_i| = 1/\sqrt{p}$. Ceci revient à $|\mathrm{S}_1(f, \mathcal{E}(\mathbb{F}_p))| \leqslant \deg(f)\sqrt{p}$, ce qui est le résultat requis. Finalement, on conclut en remarquant que $\deg(f_a) = 2$. □

10.3.2 Extraction d'aléa

Montrons un équivalent du théorème 5 :

Théorème 9. *Soient p un nombre premier à n bits, G un sous-groupe de $\mathcal{E}(\mathbb{F}_p)$ de cardinal q généré par P_0, q étant un nombre premier de ℓ bits, U_G une variable aléatoire uniformément distribuée dans G et k un entier positif. Alors,*

$$\mathbf{SD}(\mathsf{lsb}_k(\mathrm{U}_\mathrm{G}), \mathrm{U}_k) \leqslant 2^{(k+n+\log_2 n)/2+3-\ell}$$

Démonstration. Nous suivons la preuve du théorème 5, en construisant la fonction

$$\mathbb{1}(r, s, u) = \tfrac{1}{p} \times \sum_{a=0}^{p-1} e_p(a(f(r\mathrm{P}_0) - f(s\mathrm{P}_0) - \mathrm{K}u))$$

où $\mathbb{1}(r, s, u)$ est la fonction caractéristique qui est égale à 1 si $f(r\mathrm{P}_0) - f(s\mathrm{P}_0) = \mathrm{K}u \bmod p$ et 0 sinon. On peut ainsi évaluer $\mathrm{Col}(\mathsf{lsb}_k(x(\mathrm{U}_\mathrm{G})))$ où U_G est uniformément distribuée dans G, exactement de la même manière, et injecter $\mathrm{M} \leqslant 4\sqrt{p}$ dans l'équation 10.6 pour obtenir :

$$2\varepsilon \leqslant \sqrt{\frac{2^k}{p}} + \frac{2^{k/2+2}\sqrt{p}\sqrt{\log_2(p)}}{q}$$

Nous concluons comme avant en utilisant les deux inégalités $2^{n-1} < p \leqslant 2^n$ et $2^{\ell-1} < q \leqslant 2^\ell$ et en remarquant que le premier terme est négligeable par rapport au deuxième :

$$2\varepsilon \leqslant 2^{(k-n-1)/2} + 2^{(k+n+\log_2(n))/2+3-\ell}$$ □

En utilisant le cofacteur $\alpha = |\mathcal{E}(\mathbb{F}_p)| \,/\, |\mathrm{G}| \leqslant 2^{n-\ell}$ de la courbe elliptique, nous obtenons le résultat suivant :

Corollaire 9.1. *Soit e un entier positif et supposons cette inéquation vraie :*

$$k \leqslant 2\ell - (n + 2e + \log_2(n) + 6) = n - (2\log_2(\alpha) + 2e + \log_2(n) + 6)$$

Dans ce cas, l'application Ext_k est un extracteur $(\mathrm{U}_\mathrm{G}, 2^{-e})$-déterministe.

10.4 Conclusion et applications

Dans ce contexte d'extraction de clef d'un protocole d'échange de clefs utilisé pour créer un canal sécurisé, le LHL montre que l'utilisation des fonctions de hachage universelles est sûre dans le modèle standard et que, si le cardinal du groupe $\mathbb{G}$ est égal à q et si l'on veut obtenir une sécurité en 2^{-e}, alors on peut extraire $k = \log_2 q - 2e$ bits. Cependant, cette solution requiert des bits aléatoires supplémentaires, publiques et parfaitement aléatoires, ce qui augmente à la fois les complexités en temps et en communication des protocoles sous-jacents.

La troncature de la représentation en bits de l'élément aléatoire est clairement l'extracteur d'aléa le plus efficace, puisqu'elle est déterministe, et qu'elle ne requiert aucun calcul. Cependant, les résultats originaux présentés dans [FPSZ06, Gür05] n'étaient pas aussi bons

que le LHL, du point de vue du nombre de bits extraits. On pouvait extraire beaucoup moins de $\log_2 q - 2e$ bits. Dans les travaux présentés dans ce chapitre, pour des grands sous-groupes de $\mathbb{Z}_p^*$ (où l'ordre q est plus grand que $p^{3/4} \cdot 2^e$), on peut extraire jusqu'à $\log_2 q - 2e$ bits, ce qui est aussi bien que le LHL. Pour de grands sous-groupes d'une courbe elliptique sur $\mathbb{F}_p$, on extrait $n - 2e - \log_2(n) - 2\log_2(\alpha) - 6$ bits où α est le cofacteur de la courbe elliptique, ce qui n'est pas loin du LHL puisque, en pratique, α est très petit (souvent égal à 1). Et alors, pour des tailles de corps fini usuelles (p compris entre 256 et 512), on peut extraire environ $n - 2e - 16$.

Même avec notre amélioration, l'extracteur simple peut ne pas sembler très praticable pour des sous-groupes de $\mathbb{Z}_p^*$, puisque de grands sous-groupes sont nécessaires. En effet, pour générer un chaîne de 256 bits, avec une sécurité de 80 bits et un premier p de 1024 bits, on aurait besoin d'un sous-groupe de 725 bits, alors que le LHL nécessiterait un sous-groupe d'ordre 416 seulement : le temps d'exponentiation est approximativement doublé. Notons, cependant, que l'on gagne le temps de génération d'aléa supplémentaire. En revanche, sur les courbes elliptiques, l'amélioration a beaucoup de sens, puisque les groupes utilisés sont déjà gros. Les courbes elliptiques du NIST on un cofacteur de A, et on peut alors extraire 82 bits, avec une sécurité de 80 bits, sur une courbe elliptique sur un corps fini à 256 bits. Sur un corps fini à 384 bits, on peut extraire 214 bits, tandis que l'on peut extraire 346 bits sur un corps à 521 bits. Ceci est clairement suffisant comme matériel pour une clef symétrique pour de l'authentification ou du chiffrement, sans coût additionnel.

Pour conclure, mentionnons brièvement le récent générateur DRBG [NIS07] (*Dual Random Bit Generator*) SP 800-90 du NIST, approuvé dans le standard ANSI X9.82 en 2006. Basé sur les courbes elliptiques, la définition de ce générateur aléatoire de bits (RNG : *Random Bit Generator*) est adapté du générateur de Blum-Micali. En 2007 à Crypto, Brown et Gjøsteen [BG07] ont adapté la preuve de sécurité du RNG de Blum et Micali pour montrer que la sortie du DRBG est indistinguable d'une chaîne de bits uniformément distribuée, pour tout adversaire calculatoirement limité. Notre résultat permet d'améliorer ce résultat de sécurité dans deux directions, que nous ne détaillons pas ici. La première amélioration réduit le nombre d'hypothèses sur lesquelles repose la preuve de sécurité (en éliminant le TPP, *truncated point problem*). La seconde réduit la borne implicite de sécurité dans [BG07].

Quatrième partie

Calculs distribués à base de mots de passe

Chapitre 11

Définition de la primitive

Traditionnellement, il est impossible de faire de la cryptographie à clef publique à partir de petits mots de passe. La raison est qu'une clef privée de faible entropie va rapidement succomber à une attaque par dictionnaire hors ligne, ce qui est rendu possible par la publication de la clef publique, qui peut alors être utilisée comme une fonction test non-interactive. Comme les attaques hors-ligne sont très efficaces contre les secrets faibles, il est impératif que les clefs privées dans les systèmes à clef publique soient aléatoires et complexes, mais cela les rend impossibles à retenir par des humains.

Mais que se passe-t-il si, au lieu d'être une entité indivisible, la clef privée est découpée en plusieurs petites pièces, chacune d'entre elles étant retenue indépendamment par une personne différente dans un groupe d'amis ou de collègues ? Les composantes de la clef seraient alors conservées de manière sûre dans les mémoires respectives des membres du groupe, du moins tant qu'elles ne seraient pas utilisées. La seule complication est la nécessité de rassembler la clef privée entière à partir des composantes distinctes, afin que des opérations à clef privée puissent être effectuées. Naturellement, les personnes ne devraient pas réassembler effectivement la clef, mais plutôt effectuer un calcul distribué de l'opération à clef privée dont ils ont besoin, sans jamais avoir à se rencontrer ou même reconstituer la clef.

Nous introduisons donc dans ce chapitre la notion de cryptographie distribuée à base de mots de passe, où une clef privée virtuelle de haute entropie est définie implicitement comme la concaténation de mots de passe de faible entropie possédés par différents utilisateurs. Ces derniers peuvent effectuer ensemble des opérations à clef privée en échangeant des messages à travers un canal public arbitraire, à partir de leurs mots de passe respectifs, et sans jamais partager ces mots de passe ou reconstituer explicitement la clef. Tous les résultats de ce chapitre et du suivant ont été publiés à la conférence PKC 2009 [ABCP09].

Exigences inhabituelles. Même si l'on peut effectuer des calculs à clef privée sans effectivement réassembler la clef, il existe des vulnérabilités plus subtiles.

Pour commencer, on ne peut pas simplement supposer que la clef privée (virtuelle) est simplement faite d'un certain nombre de composantes aléatoires (une par utilisateur) générées indépendamment et uniformément au hasard. Au contraire, on est obligé de supposer que les diverses composantes sont arbitraires et éventuellement corrélées, et que certaines d'entre elles peuvent être très faibles et facilement devinées. Ceci provient de notre souhait d'avoir des mots de passe faciles à retenir pour des humains : il est donc impératif que les utilisateurs les choisissent eux-mêmes et sans contrainte.

Une autre conséquence est la possibilité que les mots de passe soient réutilisés plusieurs fois : même si c'est une mauvaise pratique qui devrait être découragée, elle est aussi très

commune et nous devons la gérer du mieux possible, au lieu de considérer que c'est un cas impossible.

En outre, comme les différentes personnes ne se font pas forcément confiance, il est nécessaire qu'elles puissent choisir leur secret de manière totalement privée. Notre solution doit donc gérer des corruptions ou collusions d'utilisateurs, et demeurer aussi sûre que la somme totale des composantes de clefs des utilisateurs honnêtes restants.

Finalement, nous devons avoir une notion de *leader* de groupe, qui est la personne à qui va « appartenir » la clef privée distribuée virtuelle. Appartenir signifie ici qu'il va être la seule personne capable d'utiliser la clef, c'est-à-dire obtenir les résultats de tout calcul privé l'utilisant, avec l'aide des autres membres du groupe. Nous insistons sur le fait que ni le leader ni aucun autre joueur ne doit apprendre la clef elle-même.

Une différence importante entre nos exigences et celles de tout protocole distribué à base de mots de passe (tel que GPAKE) est qu'ici les secrets sont choisis arbitrairement et de manière privée par tous les utilisateurs. Ceci est différent des notions habituelles, telles que GPAKE (où les mots de passe sont tous identiques) ou la cryptographie à seuil. Tout le système devrait donc : (1) rester sûr même si certains mots de passe sont révélés, tant que l'entropie combinée des autres mots de passe reste grande ; (2) préserver le caractère privé des autres mots de passe à toutes les étapes du processus (durant le calcul initial de la clef publique et toute utilisation ultérieure de la clef privée virtuelle).

La notion de leader de groupe est nécessaire pour notre application. La plupart des protocoles à base de mots de passe ont une finalité *symétrique*, par exemple les utilisateurs s'authentifient *mutuellement*, ou bien s'accordent sur une clef de session *partagée*. Ici, au contraire, le désir de créer un couple clef privée/clef publique doit provenir d'un joueur particulier, qui va devenir le leader, et qui cherche l'aide de personnes dont il peut avoir à peu près confiance pour l'aider à se « souvenir » de la clef. (Il pourra ensuite partager le résultat d'un calcul privé, mais cela ne rentre pas dans le cadre du protocole.) Remarquons que s'il est facile de laisser ensuite le leader décider de partager un résultat privé avec les autres, ce serait impossible de restreindre un tel résultat au leader si le calcul le révélait à tous.

Approche générale. L'objectif est donc de montrer comment effectuer de la cryptographie asymétrique à partir d'un ensemble distribué de mots de passe faibles. Comme la cryptographie à clef publique à base d'un mot de passe unique est irrémédiablement non sûre, le mieux que nous pouvons espérer est de la baser sur des ensembles distribués de mots de passe de taille raisonnable. Étant donné un système (de signature, chiffrement ou IBE par exemple), nous définissons un couple de protocoles qui prennent en entrée des mots de passe d'utilisateurs indépendants et qui, de manière distribuée, (1) génèrent une clef publique publiable qui correspond à l'ensemble des mots de passe et (2) effectuent des calculs privés sur cette clef privée virtuelle.

Pour créer un couple de clefs, un groupe de joueurs menés par un leader s'engage dans un protocole de génération de clef distribuée. Il s'effectue sur des canaux de communication non authentifiés et, si tout se passe bien, renvoie en sortie une clef publique explicite. La clef privée n'est pas calculée et reste implicitement définie par l'ensemble des mots de passe. Pour utiliser la clef privée, le même groupe de joueurs s'engage dans un autre protocole, en utilisant les mêmes mots de passe que dans le protocole de génération de clef. Ce protocole s'effectue à nouveau sur des canaux de communication non authentifiés et, si tout se passe bien, le leader, et seulement lui, obtient le résultat du calcul. À nouveau, la clef privée n'est pas calculée explicitement, et les mots de passe restent privés à leurs utilisateurs respectifs.

À l'inverse des *cryptosystèmes traditionnels à clef publique*, la clef privée n'est jamais stockée ou utilisée directement dans son ensemble ; elle reste virtuelle et délocalisée, et l'opération à clef privée est effectuée à l'aide d'un protocole interactif. À l'inverse de la *cryptographie à seuil*, où les parts des joueurs sont uniformément réparties et typiquement aussi longues que la clef elle-même, ici les mots de passe sont arbitraires et sélectionnés par les utilisateurs eux-mêmes. À l'inverse du *chiffrement à base de mot de passe*, les attaques hors-ligne sont évacuées à l'aide de l'entropie élevée de l'ensemble des mots de passe distincts, qui doivent être devi-

nés tous à la fois. Les attaques en ligne contre des mots de passe uniques ne peuvent pas être éliminées, mais elles sont très lentes car elles requièrent une mise en gage en ligne pour chaque essai. Enfin, à l'inverse des *protocoles d'échange de clefs authentifiés à base de mots de passe*, les mots de passe ne sont pas les mêmes ou reliés les uns aux autres : ils sont purement personnels.

Contributions. Nous formalisons ici cette classe de protocoles et leurs cadres de sécurité : il nous a semblé judicieux d'effectuer cela dans le cadre UC [Can01], qui se prête bien à l'analyse de protocoles à base de mots de passe, comme nous avons pu le voir dans les chapitres précédents. Nous décrivons donc les fonctionnalités idéales de génération de clef distribuée et de calcul privé distribué.

Nous donnerons dans le chapitre suivant un exemple de protocole qui les réalise, basé sur le chiffrement ElGamal [ElG85], soit dans le modèle de l'oracle aléatoire (pour l'efficacité) ou de la CRS.

Enfin, nous conclurons, toujours dans le chapitre suivant, en montrant que nos constructions se généralisent facilement à une grande classe de cryptosystèmes à clef publique basés sur le logarithme discret, ce qui inclue par exemple le chiffrement IBE, et en particulier tous les schéma dérivés de BF et BB [BF01, BB04].

Comparaison avec les travaux existants. Bien qu'il n'y ait pas de travaux antérieurs sur la cryptographie distribuée à partir de mots de passe, cette notion est bien entendue reliée à la fois au PAKE et à la *multiparty computation* (MPC).

Les protocoles de MPC (dont le premier et plus connu est celui de Yao [Yao82a]), autorisent deux participants possédant des entrées secrètes à calculer une fonction publique de leurs entrées, sans révéler autre chose que la sortie de cette fonction [GMW87b, GMW87a, BGW88, CCD88]. Ils supposent généralement que la communication entre les joueurs est authentique, c'est-à-dire qu'il existe généralement un mécanisme externe pour éviter les modifications ou l'insertion de messages. Le revers de la médaille est que de tels protocoles tendent à devenir non sûrs lorsque le nombre de joueurs malhonnêtes atteint un certain seuil du total (généralement 1/2 ou 1/3 du nombre total), ce qui leur permet de prendre le contrôle du calcul et de récupérer à partir de là les entrées des autres joueurs [RB89, BG89, HKK06].

Plusieurs travaux ont considéré la MPC sur canaux non authentifiés [CHH00, FGH$^+$02, BCL$^+$05], en faisant précéder le calcul distribué proprement dit par une sorte d'authentification basée sur des mises en gage ou signatures non-malléables [DDN00]. Les avancées de Barak *et al.* [BCL$^+$05] en particulier donnent des conditions générales sur ce qui peut et ne peut pas être réalisé en MPC non-authentifié (voir la section 5.8.3 page 85) : ils montrent qu'un adversaire peut toujours partitionner l'ensemble des joueurs en « ilôts » disjoints qui vont calculer des valeurs indépendantes, en se contenant de rejeter des messages ou les relayer correctement. Ils montrent aussi comment transformer toute (réalisation d'une) fonctionnalité UC en une version multi-joueurs de la même qui se contente de laisser l'adversaire effectuer cette partition. Ils montrent aussi comment construire un GPAKE à partir de cette notion en commençant par créer une clef de session aléatoire pour le groupe en exécutant un protocole MPC sans authentification, puis en vérifiant que tous les joueurs possèdent la même clef à l'aide d'une fonctionnalité d'« égalité de chaînes de caractères ». Ici, à l'inverse, nous forçons les utilisateurs à mettre en gage d'abord leurs mots de passe, puis effectuer le calcul proprement dit basé sur ces mises en gage.

Bien qu'il soit clair que, comme beaucoup de domaines en cryptographie, la cryptographie distribuée à base de mots de passe puisse être vu comme un cas particulier de MPC non authentifié, notre contribution n'est pas tant dans la définition conceptuelle de cette notion que dans les spécifications de fonctionnalités convenables pour ce problème non-trivial (et leur implémentation efficace). En particulier, la définition provient de l'exigence que chaque utilisateur possède son propre mot de passe (éventuellement réutilisable dans d'autres contextes), au lieu d'un mot de passe commun pour tout le groupe, ce qui est le cas dans les applications considérées par exemple dans [BCL$^+$05].

11.1 Cadre de sécurité

Nous décrivons notre protocole dans le cadre UC et utilisons les fonctionnalités de répartition, comme décrites dans la section 5.8.3 page 85. Dans toute la suite, alors que nous décrivons nos deux fonctionnalités générales $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ et $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$, il faut garder en mémoire qu'un attaquant contrôlant les canaux de communication peut toujours choisir de les voir comme les fonctionnalités partagées $s\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ et $s\mathcal{F}_{\mathsf{pwDistPrivateComp}}$, consistant implicitement en des instances multiples de $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ et $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ pour des sous-ensembles disjoints des joueurs originaux. En outre, on ne peut pas empêcher $\mathcal{A}$ de retenir des messages, qui ne vont jamais arriver. Comme dans le chapitre 7, ceci est modélisé dans nos fonctionnalités (décrites sur les figures 11.1 et 11.2 page 169 et 171) par un bit `b`, qui spécifie si le message parvient réellement au joueur ou non.

Dans la suite, nous notons n le nombre d'utilisateurs impliqués dans une exécution donnée du protocole. L'un de ces utilisateurs joue un rôle particulier et est appelé le *leader du groupe*, tandis que les autres sont simplement appelés des joueurs. Les groupes peuvent être formés arbitrairement, et chacun d'eux est *défini* par son leader (qui « possède » le groupe en étant celui qui reçoit le résultat de tout calcul privé) et un nombre arbitraire d'autres joueurs dans un ordre spécifique (qui « assistent » le leader dans son utilisation de la clef privée virtuelle du groupe).

Nous insistons sur le fait que la composition et l'ordonnancement d'un groupe est ce qui le définit et ne peut pas être changé : ceci assure qu'un tiers utilisant la clef publique du groupe sait exactement comment la clef privée va être utilisée. Si un autre joueur veut être le leader, il doit former un nouveau groupe. (Même si un tel nouveau groupe peut contenir le même ensemble de joueurs avec éventuellement des mots de passe inchangés, les deux groupes vont être distincts et avoir des paires de clefs différentes et incompatibles.)

Comme précédemment (voir section 5.8.1 page 81), la fonctionnalité n'est pas en charge de donner les mots de passe aux participants : ces derniers sont choisis par l'environnement et donnés aux joueurs en entrée.

Comme les fonctionnalités sont censées modéliser des protocoles distribués basés sur des mots de passe pour la génération de clef et des opérations à clef privée sur une primitive à clef publique arbitraire, nous représentons tous les algorithmes de la primitive comme des paramètres en boîte noire dans nos définitions. En général, on requiert :

- une fonction `SecretKeyGen` pour combiner un vecteur de mots de passe dans une clef secrète unique ;
- une fonction `PublicKeyGen` pour calculer une clef publique correspondant à un vecteur de mots de passe ;
- un prédicat `PublicKeyVer` pour vérifier une telle clef publique par rapport à n'importe quel vecteur de mots de passe : c'est important pour la correction des fonctionnalités idéales, mais cela simplifie aussi l'utilisation du théorème UC dans l'état joint (voir la section 5.1.4 page 61) puisqu'il devient inutile de considérer les mots de passe dans les données jointes ;
- une fonction `PrivateComp` pour exécuter l'opération souhaitée en utilisant la clef privée : cela peut être la fonction de déchiffrement `Dec` d'un schéma de chiffrement à clef publique, la fonction de signature `Sign` d'un schéma de signature, ou la fonction d'extraction de clef basée sur l'identité `Extract` d'un système IBE.

Les deux fonctionnalités commencent par une étape d'initialisation, dans laquelle les utilisateurs notifient leur intérêt à calculer une clef publique ou effectuer un calcul privé, selon le cas. Une telle notification est réalisée à l'aide de requêtes `NewSession` (contenant l'identifiant de session sid de l'instance du protocole, l'identité de l'utilisateur P_i et celle du groupe Pid, le mot de passe de l'utilisateur pw_i, et lors du calcul de la fonction privée, une clef publique pk et l'entrée in) envoyées par les utilisateurs. Une fois que tous les utilisateurs (partageant les mêmes sid et Pid) ont envoyé leur message de notification, la fonctionnalité informe l'adversaire qu'elle est prête à continuer.

En principe, après cette étape d'initialisation, les utilisateurs sont prêts à recevoir le résultat. La fonctionnalité attend cependant l'envoi par $\mathcal{S}$ d'un message `compute` avant de poursuivre. Cela autorise $\mathcal{S}$ à décider le moment exact où la clef peut être envoyée aux utilisateurs et, en particulier, cela lui permet de choisir le moment exact où les corruptions peuvent avoir lieu (par exemple, $\mathcal{S}$ peut décider de corrompre un joueur P_i avant que la clef ne soit envoyée mais après que P_i a décidé de participer à une session donnée du protocole, comme dans [KS05]). En outre, même si dans la fonctionnalité de génération de clef tous les utilisateurs ont normalement le droit de recevoir la clef publique, dans la fonctionnalité de calcul privé il est important que seul le leader du groupe reçoive le résultat (bien qu'il puisse choisir de le révéler ensuite à d'autres, en dehors du protocole, selon l'application).

Fig. 11.1 – La fonctionnalité distribuée de génération de clef $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$

La fonctionalité $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ est paramétrée par un paramètre de sécurité k et une fonction efficacement calculable $\mathtt{PublicKeyGen} : (\mathsf{pw}_1, \mathsf{pw}_2, \ldots, \mathsf{pw}_n) \mapsto \mathsf{pk}$ qui dérive une clef publique pk à partir d'un ensemble de mots de passe. On appelle `role` les propriétés `player` ou `leader`. La fonctionnalité interagit avec un adversaire $\mathcal{S}$ et un ensemble de joueurs $\mathrm{P}_1,\ldots,\mathrm{P}_n$ à travers les requêtes suivantes :

- **Initialisation.** À réception d'une requête $(\mathtt{NewSession}, \mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pw}_i, \mathtt{role})$ de la part d'un utilisateur P_i pour la première fois, où Pid est un ensemble de cardinal $\geqslant 2$ d'identités distinctes contenant P_i, l'ignorer si $\mathtt{role} = \mathtt{leader}$ et s'il y a déjà un enregistrement de la forme $(\mathsf{sid}, \mathsf{Pid}, *, *, \mathtt{leader})$. Enregistrer $[(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pw}_i, \mathtt{role}), \mathtt{fresh}]$ et envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathtt{role})$ à $\mathcal{S}$. Ignorer toutes les requêtes $(\mathtt{NewSession}, \mathsf{sid}, \mathsf{Pid}', *, *, *)$ ultérieures telles que $\mathsf{Pid}' \neq \mathsf{Pid}$.

  S'il y a déjà $|\mathsf{Pid}| - 1$ enregistrements de la forme $[(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_j, \mathsf{pw}_j, \mathtt{role}), \mathtt{fresh}]$ pour $\mathrm{P}_j \in \mathsf{Pid} \setminus \{\mathrm{P}_i\}$, et exactement un parmi eux tel que $\mathtt{role} = \mathtt{leader}$, alors, tout en enregistrant le $|\mathsf{Pid}|$-ième d'entre eux, enregistrer aussi $(\mathsf{sid}, \mathsf{Pid}, \mathtt{ready})$ et l'envoyer à $\mathcal{S}$. S'il n'y a pas de `leader` parmi eux, enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$ et envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$ à $\mathcal{S}$.
- **Calcul de la clef.** À réception d'un message $(\mathtt{compute}, \mathsf{sid}, \mathsf{Pid})$ de la part de l'adversaire $\mathcal{S}$, s'il y a un enregistrement de la forme $(\mathsf{sid}, \mathsf{Pid}, \mathtt{ready})$, alors calculer $\mathsf{pk} = \mathtt{PublicKeyGen}(\mathsf{pw}_1, \ldots, \mathsf{pw}_n)$ et enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk})$.
- **Livraison de la clef au leader.** À réception d'un message $(\mathtt{leaderDeliver}, \mathsf{sid}, \mathsf{Pid}, \mathsf{b})$ de la part de l'adversaire $\mathcal{S}$ pour la première fois, s'il y a un triplet $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk})$ et un enregistrement de la forme $[(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pw}_i, \mathtt{leader}), \mathtt{fresh}]$, envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk})$ à P_i et à $\mathcal{S}$ si $\mathsf{b} = 1$, ou $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$ sinon. Enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathtt{sent})$ et l'envoyer à $\mathcal{S}$. Changer le statut `fresh` du joueur en `completed` ou `error`.
- **Livraison de la clef à un autre joueur.** À réception d'un message de la forme $(\mathtt{playerDeliver}, \mathsf{sid}, \mathsf{Pid}, \mathsf{b}, \mathrm{P}_i)$ de la part de l'adversaire $\mathcal{S}$, ne rien faire s'il n'y a pas d'enregistrement de la forme $(\mathsf{sid}, \mathsf{Pid}, \mathtt{sent})$. Sinon, s'il y a des enregistrements de la forme $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk})$, $[(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pw}_i, \mathtt{player}), \mathtt{fresh}]$ et $(\mathsf{sid}, \mathsf{Pid}, \mathtt{sent})$, envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk})$ à P_i si $\mathsf{b} = 1$, ou $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$ sinon. Changer le statut `fresh` du joueur en `completed` ou `error`.
- **Corruption d'un utilisateur.** Si $\mathcal{S}$ corrompt $\mathrm{P}_i \in \mathsf{Pid}$ tel qu'il existe un enregistrement $(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pw}_i, \mathtt{role})$, alors révéler pw_i à $\mathcal{S}$. S'il y a aussi un enregistrement de la forme $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk})$ et si $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk})$ n'a pas encore été envoyé à P_i, envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk})$ à $\mathcal{S}$.

11.2 La fonctionnalité de génération de clef distribuée (fig. 11.1)

L'objectif de cette fonctionnalité est de procurer une clef publique aux utilisateurs, calculée en fonction de leurs mots de passe selon la fonction `PublicKeyGen` mentionnée précédemment donnée en paramètre. Elle doit aussi s'assurer que le leader ne reçoive jamais une clef incorrecte au final, quoi que fasse l'adversaire. Le protocole commence par une phase d'initialisation telle que déjà décrite, suivie par une phase de calcul de clef, déclenchée par une requête `compute` explicite.

Après que la clef est calculée, l'adversaire peut choisir si le leader du groupe doit effectivement recevoir cette clef. Si l'adversaire refuse la livraison, alors personne ne reçoit la clef, et c'est comme si elle n'avait jamais été calculée. Si la livraison est accordée, alors le leader et $\mathcal{S}$ reçoivent tous deux la clef publique. Ce comportement capture le fait que la clef publique générée est prévue pour être accessible à tous, à commencer par l'opposant. (Ceci va aussi permettre d'éliminer de mauvais protocoles qui ne pourraient être sûrs que si la clef publique restait inaccessible à $\mathcal{S}$.) Une fois qu'ils ont reçu cette clef publique, les autres joueurs peuvent être autorisés à la recevoir aussi, selon le planning prévu par $\mathcal{S}$, ce qui est modélisé par des requêtes de livraison de clefs de la part de $\mathcal{S}$. Une fois qu'il demande de livrer une clef à un joueur, la clef est envoyée immédiatement.

Notons qu'étant donnée la clef publique, si l'adversaire connaît assez de mots de passe afin que l'entropie combinée des mots de passe restants soit suffisamment faible, il va être capable de récupérer ces mots de passe par une attaque brutale. Ceci est inévitable et explique l'absence de requête `TestPwd` dans la fonctionnalité. (Ceci n'a rien à voir avec le fait que notre système soit distribué : des attaques hors-ligne sont toujours possibles en principe dans des systèmes à clef publique, et deviennent faisables dès qu'une portion suffisante de la clef privée est connue.)

11.3 La fonctionnalité de calcul privé distribué (figure 11.2)

L'objectif ici est d'effectuer un calcul privé pour le bénéfice unique du leader du groupe. Ce dernier est responsable de l'exactitude du calcul ; en outre, c'est l'unique utilisateur qui reçoive le résultat final.

Cette fonctionnalité va donc calculer une fonction d'une entrée donnée in, cette fonction dépendant d'un ensemble de mots de passe qui doivent définir une clef secrète correspondant à une clef publique donnée. Plus précisément, la fonctionnalité va être capable de vérifier la compatibilité des mots de passe avec la clef publique grâce à la fonction `PublicKeyVer` de vérification, et si c'est correct elle va alors calculer la clef secrète sk à l'aide de la fonction `SecretKeyGen`, et à partir de là évaluer $\texttt{PrivateComp}(\mathsf{sk}, \mathsf{in})$ et donner le résultat au leader. Notons que les deux fonctions `SecretKeyGen` et`PublicKeyVer` sont naturellement liées à la fonction `PublicKeyGen` appelée par la précédente fonctionnalité. En toute généralité, sauf si `SecretKeyGen` et `PublicKeyGen` sont toutes les deux supposées être déterministes, nous avons besoin du prédicat `PublicKeyVer` afin de vérifier qu'une clef publique est « correcte » sans forcément être « égale » (à une clef publique canonique). Notons aussi que la fonction `SecretKeyGen` n'est pas supposée être injective, car cela restreindrait le nombre d'utilisateurs et la taille totale de leurs mots de passe.

Phases et requêtes. Pendant la phase d'initialisation, chaque utilisateur reçoit en entrée un mot de passe pw_i comme décrit plus tôt, mais aussi une entrée in et une clef publique pk. Nous insistons sur le fait que la sécurité est garantie même si les utilisateurs ne partagent pas les mêmes valeurs pour in et pk, car alors la fonctionnalité échoue directement à la fin de la phase d'initialisation. À la fin de cette étape, cet adversaire apprend aussi la valeur in commune ainsi que pk (vu qu'ils sont supposés publics).

Après cette étape d'initialisation, mais avant le calcul effectif, l'adversaire $\mathcal{S}$ a l'opportunité d'essayer de deviner *simultanément* un ou plusieurs mots de passe, en posant une unique requête de test de mot de passe, afin de modéliser une attaque de type « *man-in-the-middle* » en essayant de prendre le contrôle d'un sous-ensemble d'utilisateurs. La requête doit indiquer le sous-ensemble d'utilisateur(s) concerné(s) par l'attaque, et quel(s) mot(s) de passe $\mathcal{S}$ souhaite tester pour cet(s) utilisateur(s). Si tous les mots de passe sont compatibles avec pk, les utilisateurs affectés sont notés `compromised`, sinon ils sont notés `interrupted`. Des utilisateurs non concernés sont notés `fresh`. Observons que le meilleur intérêt de l'opposant est de viser uniquement un utilisateur unique dans la requête de test de mot de passe pour optimiser la probabilité de compromission.

Fig. 11.2 – La fonctionnalité distribuée de calcul privé $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$

$\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ est paramétrée par un paramètre de sécurité k et trois fonctions efficacement calculables. `PublicKeyVer` est une fonction booléenne $\mathtt{PublicKeyVer} : (\mathsf{pw}_1, \mathsf{pw}_2, \ldots, \mathsf{pw}_n, \mathsf{pk}) \mapsto b$, où $b = 1$ si les mots de passe et la clef publique sont compatibles, $b = 0$ sinon. `SecretKeyGen` est une fonction $\mathtt{SecretKeyGen} : (\mathsf{pw}_1, \mathsf{pw}_2, \ldots, \mathsf{pw}_n) \mapsto \mathsf{sk}$, où sk est la clef secrète obtenue à partir des mots de passe. Finalement, `PrivateComp` est une fonction à clef privée $\mathtt{PrivateComp} : (\mathsf{sk}, \mathsf{in}) \mapsto \mathsf{out}$, où sk est la clef privée, in l'entrée de la fonction (par exemple un chiffré) et out le résultat privé du calcul (par exemple le message déchiffré). Nous notons `role` les propriétés `player` ou `leader`. La fonctionnalité interagit avec un adversaire $\mathcal{S}$ et un ensemble de joueurs $\mathrm{P}_1, \ldots, \mathrm{P}_n$ à travers les requêtes suivantes :

- **Initialisation.** À réception d'une requête $(\mathtt{NewSession}, \mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pk}, \mathsf{in}, \mathsf{pw}_i, \mathtt{role})$ de la part d'un utilisateur P_i pour la première fois, où Pid est un ensemble de cardinal $\geqslant 2$ d'identités distinctes contenant P_i, l'ignorer si $\mathtt{role} = \mathtt{leader}$ et s'il y a déjà un enregistrement de la forme $(\mathsf{sid}, \mathsf{Pid}, *, *, *, *, \mathtt{leader})$. Enregistrer $[(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pk}, \mathsf{in}, \mathsf{pw}_i, \mathtt{role}), \mathtt{fresh}]$ et envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pk}, \mathsf{in}, \mathtt{role})$ à $\mathcal{S}$. Ignorer toutes les requêtes $(\mathtt{NewSession}, \mathsf{sid}, \mathsf{Pid}', *, *, *, *, *)$ ultérieures où $\mathsf{Pid}' \neq \mathsf{Pid}$.

  S'il y a déjà $|\mathsf{Pid}| - 1$ enregistrements de la forme $[(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_j, \mathsf{pk}, \mathsf{in}, \mathsf{pw}_j, \mathtt{role}), \mathtt{fresh}]$ pour $\mathrm{P}_j \in \mathsf{Pid} \setminus \{\mathrm{P}_i\}$, et exactement un parmi eux tel que $\mathtt{role} = \mathtt{leader}$, alors, tout en enregistrant le $|\mathsf{Pid}|$-ième d'entre eux, vérifier que les valeurs de c et pk sont les mêmes pour tous les utilisateurs. Si les $|\mathsf{Pid}| - 1$ enregistrements ne vérifient pas toutes ces conditions, envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$ à $\mathcal{S}$ et abandonner. Sinon, enregistrer aussi $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk}, \mathsf{in}, \mathtt{ready})$ et l'envoyer à $\mathcal{S}$.
- **Test de mot de passe.** À réception d'une première requête de la part de l'adversaire $\mathcal{S}$ de la forme $(\mathtt{TestPwd}, \mathsf{sid}, \mathsf{Pid}, \{\mathrm{P}_{i_1}, \ldots, \mathrm{P}_{i_l}\}, \{\mathsf{pw}_{i_1}, \ldots, \mathsf{pw}_{i_l}\})$, s'il existe l enregistrements $[(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_{i_k}, \mathsf{pk}, \mathsf{in}, \mathsf{pw}'_{i_k}, *), \mathtt{fresh}]$, et un enregistrement $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk}, \mathsf{in}, \mathtt{ready})$, alors noter $\mathsf{pw}_{j_{l+1}}, \ldots, \mathsf{pw}_{j_n}$ les mots de passe des $n - \ell$ autres utilisateurs du groupe. Si $\mathtt{PublicKeyVer}(\mathsf{pw}_1, \ldots, \mathsf{pw}_n, \mathsf{pk}) = 1$, éditer les enregistrements de $\mathrm{P}_{i_1}, \ldots, \mathrm{P}_{i_l}$ pour changer leur statut en `compromised` et répondre à $\mathcal{S}$ que sa requête est correcte. Sinon, noter les enregistrements des utilisateurs $\mathrm{P}_{i_1}, \ldots, \mathrm{P}_{i_l}$ `interrupted` et répondre à $\mathcal{S}$ que sa requête est incorrecte. Ignorer toutes les requêtes ultérieures de la forme $(\mathtt{TestPwd}, \mathsf{sid}, *, *, *)$.
- **Calcul privé.** À réception d'un message $(\mathtt{compute}, \mathsf{sid}, \mathsf{Pid})$ de la part de l'adversaire $\mathcal{S}$, s'il y a un enregistrement de la forme $(\mathsf{sid}, \mathsf{Pid}, \mathsf{pk}, \mathsf{in}, \mathtt{ready})$, si aucun joueur n'est `interrupted` et si $\mathtt{PublicKeyVer}(\mathsf{pw}_1, \ldots, \mathsf{pw}_n, \mathsf{pk}) = 1$, alors calculer $\mathsf{sk} = \mathtt{SecretKeyGen}(\mathsf{pw}_1, \ldots, \mathsf{pw}_n)$ et $\mathsf{out} = \mathtt{PrivateComp}(\mathsf{sk}, \mathsf{in})$, et enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{out})$; ensuite, pour tout $\mathrm{P}_i \in \mathsf{Pid}$ tel que $\mathtt{role} = \mathtt{player}$, changer le statut de l'enregistrement $(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pk}, \mathsf{in}, \mathsf{pw}_i, \mathtt{player})$ en `completed`. Dans tous les autres cas, enregistrer $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$ et changer le statut des joueurs (dont le rôle est `player`) en `error`. Quand le résultat du calcul est fixé, donner le statut (soit `error` soit `completed`) à $\mathcal{S}$.
- **Livraison de la clef au leader.** À réception d'un message $(\mathtt{leaderDeliver}, \mathsf{sid}, \mathsf{Pid}, \mathsf{b})$ de la part de l'adversaire $\mathcal{S}$ pour la première fois, s'il y a un triplet $(\mathsf{sid}, \mathsf{Pid}, \mathsf{out})$ tel que $\mathsf{out} \in \{\textit{messages bien formés}\} \cup \{\mathtt{error}\}$, et s'il existe un enregistrement de la forme $(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pk}, \mathsf{in}, \mathsf{pw}_i, \mathtt{leader})$, envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{out})$ à P_i si $\mathsf{b} = 1$, ou $(\mathsf{sid}, \mathsf{Pid}, \mathtt{error})$ sinon. Si le leader du groupe P_i est corrompu ou `compromised`, alors envoyer aussi $(\mathsf{sid}, \mathsf{Pid}, \mathsf{out})$ à $\mathcal{S}$ (notons que $\mathcal{S}$ reçoit out automatiquement si P_i est corrompu). Changer alors son statut en `completed` ou `error`.
- **Corruption d'un utilisateur.** Si $\mathcal{S}$ corrompt $\mathrm{P}_i \in \mathsf{Pid}$ tel qu'il existe un enregistrement $(\mathsf{sid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pk}, \mathsf{in}, \mathsf{pw}_i, \mathtt{role})$, alors révéler pw_i à $\mathcal{S}$. Si $\mathtt{role} = \mathtt{leader}$, s'il y a aussi un enregistrement de la forme $(\mathsf{sid}, \mathsf{Pid}, \mathsf{out})$ et si $(\mathsf{sid}, \mathsf{Pid}, \mathsf{out})$ n'a pas encore été envoyé à P_i, envoyer $(\mathsf{sid}, \mathsf{Pid}, \mathsf{out})$ à $\mathcal{S}$.

Une fois que la fonctionnalité a reçu un message de la forme $(\mathtt{compute}, \mathsf{sid}, \mathsf{Pid})$ de la part de $\mathcal{S}$, il procède à la phase de calcul. Si (1) tous les enregistrements sont `fresh` ou `compromised`, et (2) les mots de passe sont compatibles avec la clef publique commune pk, alors la fonctionnalité calcule la clef privée sk et ensuite la sortie out. Dans tous les autres cas, aucun message n'est calculé.

Dans tous les cas, après la génération de clef, la fonctionnalité informe l'adversaire du résultat, ce qui signifie que $\mathcal{S}$ apprend si un message a été effectivement calculé ou non. En particulier, cela signifie que l'adversaire apprend aussi si les mots de passe des utilisateurs sont compatibles avec pk ou non. À première vue, cela semble être une information critique à donner à l'adversaire. Ce n'est cependant pas le cas dans notre cadre (comme dans celui du chapitre 7). Déjà, apprendre le statut du protocole (c'est-à-dire, s'il a réussi ou non) sans avoir aucune connaissance des mots de passe est complètement inutile, et la seule connaissance que l'adversaire peut avoir de ces mots de passe est celle qu'il a utilisée dans sa requête `TestPwd`. Ainsi, comme on peut le supposer, la seule chose utile que l'adversaire peut apprendre du statut du protocole est si les tests de mots de passe qu'il a fait étaient *tous* corrects ou non (une simple réponse oui/non), mais rien d'autre. Ensuite, même si l'adversaire pouvait apprendre des informations plus utiles à partir du statut du protocole, ce dernier peut difficilement être dissimulé dans des scénarios réels puisqu'il est facile de le déduire du comportement ultérieur des joueurs.

Ensuite, comme dans la première fonctionnalité, le résultat final peut soit être révélé au leader du groupe, soit lui être masqué. Cependant, cette fois, puisque le résultat final est une sortie privée, il n'y a pas de possibilité de le donner aux autres joueurs. De même, $\mathcal{S}$ ne récupère le message que si le leader a été précédemment corrompu ou s'il est dans l'état `compromised` (c'est-à-dire soit s'il est tombé sous le contrôle de $\mathcal{S}$, soit si $\mathcal{S}$ a pris sa place avec succès dans le protocole en devinant son mot de passe).

Discussion. Nous insistons sur le fait que dans ce cadre seul le leader et aucun autre joueur ne reçoit le résultat final. Ceci a l'avantage de rendre la construction plus simple, c'est aussi le choix qui semble le plus raisonnable, et cela rend notre protocole beaucoup plus résistant aux révélations de mots de passe dans les attaques en ligne. Pour voir cela, supposons que la sortie finale ait en fait été envoyée à tous les joueurs. Alors, casser le mot de passe d'un seul utilisateur serait en fait suffisant pour casser tout le système : ajouter des utilisateurs réduirait la sécurité totale, car un groupe plus grand amènerait une probabilité plus grande qu'un utilisateur choisisse un mot de passe faible. Au contraire, dans le cadre tel qu'on l'a défini, casser le mot de passe d'un utilisateur ordinaire n'a pas de conséquence grave : la sécurité du protocole va simplement continuer à reposer sur le reste des mots de passe. Puisque des utilisateurs ordinaires `compromised` n'apportent aucune autre information que celle contenue dans leurs mots de passe, c'est comme si les mots de passe révélés étaient retirés de la clef dans des exécutions futures du protocole, ou encore s'ils n'avaient en fait jamais contribué.

Bien sûr, casser le mot de passe du leader va compromettre tout le groupe et garantir l'accès aux calculs privés (toujours avec l'aide des autres joueurs), mais c'est naturel puisque le groupe *appartient* au leader. C'est une distinction importante entre l'exposition du mot de passe d'un joueur ordinaire et de celui du leader : le leader représente le groupe par rapport à des tiers, c'est-à-dire que lorsque des joueurs utilisent la clef publique du groupe, leur intention est de communiquer avec le leader. Au contraire, les joueurs ordinaires ne sont pas censés être dignes de confiance et leur inclusion dans le groupe est un choix du leader pour l'aider à renforcer la sécurité de la clef privée – ou la laisser inchangée si ce joueur devient `compromised` – mais jamais à la diminuer.

11.4 Corruptions d'utilisateurs

Notre définition de la fonctionnalité $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ gère les corruptions d'utilisateurs d'une manière assez différente de celle d'autres protocoles de groupe basés sur des mots de passe. Par exemple, dans la fonctionnalité d'échange de clefs de groupe de Katz et Shin [KS05] (dont nous avons repris les idées dans le chapitre 7), si l'adversaire a obtenu les mots de passe de certains participants (à travers des essais ou des corruptions), il peut librement fixer la clef de session résultante à n'importe quelle valeur. Ici, nos fonctionnalités sont beaucoup plus exigeantes sur deux points : $\mathcal{S}$ est tout d'abord beaucoup plus contraint dans la façon dont il peut tester en ligne des mots de passe ; ensuite, $\mathcal{S}$ ne peut pas altérer le calcul une fois qu'il a commencé (il ne peut pas choisir le résultat).

Tests de mots de passe. La première différence est que la requête `TestPwd` ne peut être posée qu'une seule fois, tôt dans le protocole, et elle ne teste pas vraiment le mot de passe des utilisateurs, mais plutôt la compatibilité entre (1) les mots de passe testés d'un groupe spécifique d'utilisateurs, (2) les vrais mots de passe du reste du groupe (connus par la fonctionnalité grâce aux requêtes `NewSession`), et (3) la clef publique (dont on garantit à ce stade qu'elle est la même pour tous les utilisateurs). Cette forme inhabituelle pour la requête `TestPwd` procure un niveau de sécurité très élevé, car (A) au plus un seul ensemble de mots de passe peut être testé pour un joueur dans toute instance de protocole, et (B) si $\mathcal{S}$ choisit de tester un ensemble de plus d'un mot de passe, alors, pour obtenir une réponse positive, tous les mots de passe doivent être corrects simultanément (et comme cette probabilité devient exponentiellement faible, l'adversaire astucieux devrait tester un mot de passe à la fois seulement). Les attaques en ligne sont inévitables mais vraiment inefficaces ici.

Après le calcul privé et sa livraison, tous les enregistrements, initialement dans l'état `fresh`, `compromised`, ou `interrupted`, deviennent soit `completed` soit `error`. Plus aucune requête `TestPwd` n'est acceptée à ce stade, car une fois que les utilisateurs ont terminé leur tâche, il est trop tard pour que $\mathcal{S}$ puisse prendre le contrôle sur eux (bien que des requêtes de corruption puissent encore être effectuées pour apprendre leur état interne). Notons qu'une seule requête `TestPwd` est autorisée pour chaque instance de $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$, mais que plusieurs instances peuvent être invoquées par la fonctionnalité de répartition $s\mathcal{F}_{\mathsf{pwDistPrivateComp}}$.

Robustesse. La seconde différence avec le cadre de [KS05] est que l'on ne donne pas à l'adversaire le droit d'altérer le résultat du calcul lorsqu'il corrompt des utilisateurs ou apprend certains mots de passe. Ceci signifie en particulier que soit le leader du groupe reçoit quelque chose de cohérent, soit il reçoit une erreur ; il ne peut pas recevoir quelque chose de faux, ce qui rend le protocole *robuste*. La robustesse est en fait automatique si nous faisons l'hypothèse que la fonction de calcul `PrivateComp` est déterministe ; par simplicité, c'est le cas du protocole générique décrit en détail dans ce papier. Nous mentionnerons cependant ensuite certaines applications qui requièreront de l'aléa dans le calcul. Sans rentrer dans les détails, on peut tout de même conserver la robustesse du protocole dans ce cas en demandant à tous les joueurs de mettre en gage leurs aléas dans la première étape, de la même manière qu'ils vont mettre en gage leurs mots de passe (voir la description du protocole plus loin). Ceci nous permet de traiter ces aléas comme une entrée privée standard dans le cadre, et donc d'interdire l'adversaire de les modifier une fois que le calcul a commencé.

Remarquons que bien que l'adversaire ne puisse pas truquer le calcul, l'environnement apprend la terminaison du protocole, et pourrait donc distinguer entre les mondes réel et idéal si l'adversaire gagnait plus souvent dans l'un que dans l'autre. On doit bien sûr s'attendre à ce que l'environnement apprenne une telle information dans la réalité, et donc il est naturel que notre cadre puisse le gérer. (Nos implémentations vont donc devoir s'assurer que les conditions de succès sont les mêmes dans les deux mondes.)

Chapitre 12

Application au déchiffrement distribué

Nous proposons ici deux protocoles satisfaisant les fonctionnalités présentées dans le chapitre précédent pour un cas particulier de calcul privé distribué non authentifié [BCL$^+$05]. Informellement, si l'on suppose que s est une clef privée, l'objectif du protocole va être de calculer la valeur c^s étant donné un élément c du groupe. Nous nous focalisons ici sur du déchiffrement ElGamal, reposant sur l'hypothèse DDH.

Il peut ensuite être modifié pour reposer sur l'hypothèse linéaire décisionnelle afin d'être compatible avec des groupes bilinéaires [BBS04] : nous donnerons les grands lignes de cette modification à la fin de ce chapitre. Ce calcul pourra ainsi être utilisé pour obtenir des signatures BLS distribuées [BLS01], des déchiffrements ElGamal [ElG85] ou linéaires [BBS04], et des extractions de clefs BF et BB1 basées sur l'identité [BF01, BB04].

12.1 Définitions et notations

Soient $\mathbb{G}$ un groupe d'ordre premier p, et g un générateur de ce groupe. On suppose en outre qu'un élément $h \in \mathbb{G}$ est donné comme CRS. On utilise les blocs de base suivants :

12.1.1 Sélection des mots de passe

Chaque utilisateur P_i possède un mot de passe pw_i choisi de manière privée, qui tient le rôle de la i-ième part de la clef secrète sk (voir ci-dessous). Par simplicité, on écrit $\mathsf{pw}_i = \mathsf{pw}_{i,1} \dots \mathsf{pw}_{i,\ell} \in \{0, \dots, 2^{\mathrm{L}\ell} - 1\}$, c'est-à-dire que l'on divise à nouveau chaque mot de passe pw_i en ℓ blocs $\mathsf{pw}_{i,j} \in \{0, \dots, 2^{\mathrm{L}} - 1\}$ de L bits chacun, où $p < 2^{\ell \mathrm{L}}$. Cette segmentation en blocs est un point technique pour obtenir des mises en gage extractibles efficaces de longs mots de passe (elles requièrent un calcul de logarithme discret) : dans le schéma concret, par exemple, nous allons utiliser des blocs à un seul bit afin de réussir l'extraction la plus efficace possible (c'est-à-dire $\mathrm{L} = 1$ et $\ell = 160$ pour un entier premier p de 160 bits). Notons que bien que nous autorisions des grands mots de passe contenant jusqu'à $\mathrm{L}\ell$ bits (la taille de p), les utilisateurs ont bien entendu le droit de choisir des mots de passe plus courts.

12.1.2 Combinaison des mots de passe

La clef privée sk est définie comme la combinaison (virtuelle) de tous les mots de passe pw_i. La définition précise de cette combinaison importe peu, tant qu'elle est reproductible et préserve l'entropie jointe de l'ensemble des mots de passe (jusqu'à $\log_2 p$ bits, puisque c'est la longueur de sk). Par exemple, s'il y a n utilisateurs, tous avec des mots de passe courts $\mathsf{pw}_i^* \in \{0, \dots, \Delta - 1\}$ tels que $\Delta^n < p$, définir $\mathsf{pw}_i = \Delta^i \mathsf{pw}_i^*$ et choisir $\mathsf{sk} = \sum_i \mathsf{pw}_i$ suffit à assurer qu'il n'y a pas de recouvrements, ou d'annulation mutuelle de deux ou plusieurs mots de passe, qui pourraient être reliés (car ils ne sont pas supposés indépendants).

En général, il est préférable que chaque utilisateur transforme indépendamment son véritable mot de passe pw_i^* en un mot de passe effectif pw_i en appliquant un extracteur convenable $\mathsf{pw}_i = \mathsf{H}(i, \mathsf{pw}_i^*, \mathrm{Z}_i)$ où Z_i est une information publique appropriée quelconque, telle qu'une description du groupe et de son objectif. On peut alors utiliser de manière sûre $\mathsf{sk} = \sum_i \mathsf{pw}_i$ et être assuré que l'entropie de sk va être très proche de l'entropie jointe du vecteur $(\mathsf{pw}_1^*, \dots, \mathsf{pw}_n^*)$ des mots de passe pris ensemble. Un tel pré-calcul de mot de passe en utilisant des fonctions de hachage est très standard, mais se situe en dehors de la définition des fonctionnalités proprement dite.

12.1.3 Clef privée et clef publique

Les mots de passe (effectifs) pw_i sont utilisés pour définir une paire de clefs $(\mathsf{sk}, \mathsf{pk} = g^{\mathsf{sk}})$ pour un mécanisme d'encapsulation de clefs ElGamal basé sur un mot de passe (KEM). On définit les fonctions décrites dans le cadre de sécurité page 168 :

$$\mathsf{sk} = \mathtt{SecretKeyGen}(\mathsf{pw}_1, \dots, \mathsf{pw}_n) \stackrel{\mathsf{def}}{=} \sum_{i=1}^{n} \mathsf{pw}_i$$
$$\mathsf{pk} = \mathtt{PublicKeyGen}(\mathsf{pw}_1, \dots, \mathsf{pw}_n) \stackrel{\mathsf{def}}{=} g^{\sum \mathsf{pw}_i}$$

La fonction de vérification de clef publique est alors

$$\mathtt{PublicKeyVer}(\mathsf{pw}_1, \dots, \mathsf{pw}_n, \mathsf{pk}) \stackrel{\mathsf{def}}{=} \left(\mathsf{pk} \stackrel{?}{=} g^{\sum \mathsf{pw}_i}\right)$$

L'opération de clef publique du KEM ElGamal est l'encapsulation $\mathtt{Enc} : (\mathsf{pk}, r) \mapsto (c = g^r, m = \mathsf{pk}^r)$, qui renvoie une clef de session aléatoire m et un chiffré c. L'opération de clef privée est la décapsulation $\mathtt{Dec} : (\mathsf{sk}, c) \mapsto m = c^{\mathsf{sk}}$, qui est ici déterministe. Observons que bien que `Dec` instancie `PrivateComp` dans les fonctionnalités, `Enc` est prévu pour un usage public extérieur et n'apparaît jamais dans les protocoles privés.

12.1.4 Préservation de l'entropie

Afin que les faibles entropies des mots de passe se combinent proprement pour former la clef secrète $\mathsf{sk} = \sum_i \mathsf{pw}_i$, le mot de passe pw_i effectif doit être correctement « découplé » pour éviter des annulations mutuelles, comme discuté ci-dessus, soit avec un décalage pour éviter

des superpositions comme dans $\mathsf{pw}_i = \Delta^i \mathsf{pw}'_i$ (où Δ est une base spécifique), soit randomisé de manière distincte comme dans $\mathsf{pw}_i = \mathrm{H}(i, \mathsf{pw}'_i, \mathrm{Z}_i)$.

Notons que, même avec ce type de randomisation, il est possible que l'entropie actuelle de sk soit plus petite que sa valeur maximale de $\log_2 p$ bits, par exemple s'il n'y a pas assez d'utilisateurs honnêtes ou si leurs mots de passe sont trop petits. Malgré tout, il n'y a pas d'attaque effective connue contre le logarithme discret et les problèmes reliés qui pourrait mettre à profit la faible entropie de sk. Spécifiquement, indépendamment de la combinaison effective des mots de passe, on peut facilement prouver qu'aucune attaque générique [Sho97] ne peut résoudre le problème du logarithme discret ou du DDH en moins de $\sqrt{2^h}$ opérations, où h est la min-entropie de la clef privée sk conditionnée sur tous les mots de passe connus par l'adversaire.

12.1.5 Mises en gage homomorphes extractibles

La première étape de notre protocole de déchiffrement distribué consiste, pour chaque utilisateur, à mettre en gage son mot de passe (les détails sont donnés dans la section suivante). La mise en gage (voir page 25) doit être extractible, homomorphe et compatible avec la forme de la clef publique. De façon générale, on a besoin d'une mise en gage $\mathsf{Commit}(\mathsf{pw}, r)$ additivement homomorphe sur pw et avec certaines propriétés sur r. Afin de simplifier la description des protocoles, nous avons choisi d'utiliser le schéma ElGamal [ElG85], qui est additif sur la valeur aléatoire r, et donné par $\mathsf{Commit}_v(\mathsf{pw}, r) = (v^{\mathsf{pw}} h^r, g^r)$. La sécurité sémantique repose sur l'hypothèse DDH. L'extractibilité est possible grâce à la clef x de déchiffrement, telle que $h = g^x$, dans le modèle de la CRS (voir page 27).

12.1.6 Preuves *Zero-Knowledge Simulation-Sound* non interactives

Informellement, un système de preuve *zero-knowledge* (voir page 24) est dit *simulation-sound* si un adversaire ne peut pas donner une preuve convaincante d'un faux énoncé, même s'il a accès en tant qu'oracle au simulateur *zero-knowledge*. La non-malléabilité est aussi requise, c'est-à-dire qu'une preuve d'un théorème ne peut pas être convertie en une preuve d'un autre théorème. De Santis *et al.* ont prouvé dans [DDO+01] l'existence d'un tel schéma, possédant la propriété additionnelle d'être non-interactif, si l'on suppose l'existence de permutations à sens unique à trappe. Notons que leur schéma autorise des simulations multiples avec une CRS unique, ce qui est crucial pour le cadre multi-session. Si nous instancions toutes les preuves *simulation-sound non-interactive zero-knowledge* (SSNIZK) de nos protocoles par ces dernières, alors ils sont UC-sûrs dans le modèle de la CRS.

Toutefois, par souci d'efficacité, nous pouvons à la place les instancier en utilisant des preuves de type Schnorr d'égalité de logarithmes discrets [FP01], qui reposent sur le modèle de l'oracle aléatoire [BR93] (voir page 26), mais sont significativement plus utilisables en pratique. Ces preuves SSNIZK sont bien connues (voir les détails dans la section 12.6 page 186 et les preuves dans [FP01]).

Nous utiliserons ici la notation générique $\mathsf{SSNIZK}_{\mathcal{L}}(x; w)$ pour une preuve que x appartient au langage $\mathcal{L}$, avec pour témoin w. Une telle preuve peut être paramétrée par une étiquette, que l'on place alors en exposant : $\mathsf{SSNIZK}^{\mathrm{H}}$. En outre, la notation $\mathsf{CDH}(g, \mathrm{G}, h, \mathrm{H})$ signifie que $(g, \mathrm{G}, h, \mathrm{H})$ appartient au langage CDH : il existe un exposant commun x tel que $\mathrm{G} = g^x$ et $\mathrm{H} = h^x$. Nous utiliserons en particulier les deux types de preuves suivants :

- $\mathsf{SSNIZK}_{\mathsf{CDH}}((g, \mathrm{G}, h, \mathrm{H}); r)$, ce qui signifie que $\mathrm{G} = g^r$ et $\mathrm{H} = h^r$;
- $\mathsf{SSNIZK}_{\mathsf{CDH}\vee\mathsf{CDH}}(((g, \mathrm{G}, h, \mathrm{H}), (g', \mathrm{G}', h', \mathrm{H}')); r)$, ce qui signifie que l'on a soit $\mathrm{G} = g^r$ et $\mathrm{H} = h^r$, soit $\mathrm{G}' = g'^r$ et $\mathrm{H}' = h'^r$.

12.2 Intuition

Nous décrivons en premier lieu l'algorithme de calcul distribué. Tous les utilisateurs possèdent un mot de passe pw_i, une clef publique pk et un chiffré c. L'un d'eux est le leader du groupe, appelé P_1, et les autres sont notés $\mathrm{P}_2, \ldots, \mathrm{P}_n$. Pour ce chiffré donné $c \in \mathbb{G}$, le leader

souhaite obtenir $m = c^{\mathsf{sk}}$. Mais avant de calculer cette valeur, tout le monde veut s'assurer que les utilisateurs sont honnêtes, ou au moins que la combinaison des mots de passe est compatible avec la clef publique.

Fig. 12.1 – Étapes du protocole de génération de clef distribuée

$(1a)$ $r_{i,j} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^*$ $\quad \mathrm{C}_{i,j} = \mathsf{Commit}_g(\mathsf{pw}_{i,j}, r_{i,j}) = (g^{\mathsf{pw}_{i,j}} h^{r_{i,j}}, g^{r_{i,j}})$
$\Pi^0_{i,j} = \mathsf{SSNIZK}_{\mathsf{CDH}}(((g, \mathrm{C}^{(2)}_{i,j}, h, \mathrm{C}^{(1)}_{i,j}), (g, \mathrm{C}^{(2)}_{i,j}, h, \mathrm{C}^{(1)}_{i,j}/g)); r_{i,j})$
$\mathrm{C}_i = \{\mathrm{C}_{i,j}\}_j, \{\Pi^0_{i,j}\}_j$ $\xrightarrow{\mathrm{C}_i}$

$(1b)$ $\mathrm{H} = \mathcal{H}(\mathrm{C}_1, \ldots, \mathrm{C}_n)$ $\quad s_i \xleftarrow{\mathrm{R}} \mathbb{Z}_q^*$
$\mathrm{C}'_i = \mathsf{Commit}^{\mathrm{H}}_g(\mathsf{pw}_i, s_i) = (g^{\mathsf{pw}_i} h^{s_i}, g^{s_i}, \mathrm{H})$
$\Pi^1_i = \mathsf{SSNIZK}^{\mathrm{H}}_{\mathsf{CDH}}((g, \mathrm{C}'^{(2)}_i / \prod_j \mathrm{C}^{(2)}_{i,j}, h, \mathrm{C}'^{(1)}_i / \prod_j \mathrm{C}^{(1)}_{i,j}); s_i - \sum_j r_{i,j})$ $\xrightarrow{\mathrm{C}'_i, \Pi^1_i}$

$(1c)$ $\gamma^{(1)}_0 = \prod_i \mathrm{C}'^{(1)}_i = g^{\sum_i \mathsf{pw}_i} h^{\sum_i s_i}$ $\quad \gamma^{(2)}_0 = h$
étant donné, pour tout $j = 1, \ldots, i-1$, $\quad (\gamma^{(1)}_j, \gamma^{(2)}_j, \Pi^2_j)$
vérifier $\Pi^2_j \stackrel{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((\gamma^{(1)}_{j-1}, \gamma^{(1)}_j, \gamma^{(2)}_{j-1}, \gamma^{(2)}_j); \alpha_j)$
$\alpha_i \xleftarrow{\mathrm{R}} \mathbb{Z}_q^*$ $\quad \gamma^{(1)}_i = (\gamma^{(1)}_{i-1})^{\alpha_i}$ $\quad \gamma^{(2)}_i = (\gamma^{(2)}_{i-1})^{\alpha_i}$
$\Pi^2_i = \mathsf{SSNIZK}_{\mathsf{CDH}}((\gamma^{(1)}_{i-1}, \gamma^{(1)}_i, \gamma^{(2)}_{i-1}, \gamma^{(2)}_i); \alpha_i)$ $\xrightarrow{\gamma^{(1)}_i, \gamma^{(2)}_i, \Pi^2_i}$

$(1d)$ étant donné $\gamma^{(1)}_n = g^{\alpha \sum_i \mathsf{pw}_i} h^{\alpha \sum_i s_i}$ et $\gamma^{(2)}_n = h^{\alpha}$
vérifier $\Pi^2_n \stackrel{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((\gamma^{(1)}_{n-1}, \gamma^{(1)}_n, \gamma^{(2)}_{n-1}, \gamma^{(2)}_n); \alpha_n)$
$h_i = (\gamma^{(2)}_n)^{s_i}$ $\quad \Pi^3_i = \mathsf{SSNIZK}_{\mathsf{CDH}}((g, \mathrm{C}'^{(2)}_i, \gamma^{(2)}_n, h_i); s_i)$ $\xrightarrow{h_i, \Pi^3_i}$

$(1e)$ étant donné, pour tout $j = 1, \ldots, n$, $\quad (h_j, \Pi^3_j)$
vérifier $\Pi^3_j \stackrel{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((g, \mathrm{C}'^{(2)}_j, \gamma^{(2)}_n, h_j); s_j)$
$\zeta_{n+1} = \gamma^{(1)}_n / \prod_j h_j = g^{\alpha \sum_j \mathsf{pw}_j}$
étant donné, pour tout $j = n, \ldots, i+1$, $\quad (\zeta_j, \Pi^4_j)$
vérifier $\Pi^4_j \stackrel{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((\gamma^{(1)}_{j-1}, \gamma^{(1)}_j, \zeta_j, \zeta_{j+1}); 1/\alpha_j)$
$\zeta_i = (\zeta_{i+1})^{1/\alpha_i}$ $\quad \Pi^4_i = \mathsf{SSNIZK}_{\mathsf{CDH}}((\gamma^{(1)}_{i-1}, \gamma^{(1)}_i, \zeta_i, \zeta_{i+1}); 1/\alpha_i)$ $\xrightarrow{\zeta_i, \Pi^4_i}$

$(1f)$ étant donné, pour tout $j = i-1, \ldots, 1$, $\quad (\zeta_j, \Pi^4_j)$
vérifier $\Pi^4_j \stackrel{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((\gamma^{(1)}_{j-1}, \gamma^{(1)}_j, \zeta_j, \zeta_{j+1}); 1/\alpha_j)$
$\mathsf{pk} = \zeta_1$

Le protocole commence par vérifier que les joueurs vont être capables de déchiffrer le chiffré, et donc qu'ils connaissent effectivement une représentation en parts de la clef de déchiffrement. Chaque utilisateur envoie deux mise en gage consécutives C_i et C'_i de son mot de passe. C'est la deuxième uniquement qui sera utilisée dans la suite, la première ne sert qu'à « tricher » sur les mots de passe utilisés. Comme on le voit dans la preuve (section 9.2.1), la première mise en gage doit être extractible afin que le simulateur soit capable de retrouver les mots de passe utilisés par l'adversaire : c'est une contrainte du cadre UC, comme dans [CHK+05] et les chapitres précédents. Plus précisément, le simulateur doit être capable de tout simuler sans connaître les mots de passe. Il les récupère donc en les extrayant des mises en gage C_i effectuées par l'adversaire à la première étape, ce qui lui permet d'ajuster ses propres valeurs avant la mise en gage suivante, afin que tous les mots de passe soient compatibles avec la clef publique (si ce doit bien être le cas). Si l'on réfléchit en termes de chiffrement ElGamal, l'extraction est proportionnelle en la racine carrée de la taille de l'alphabet, ce qui est utilisable en pratique pour des mots de passe de 20 bits mais pas pour des mots de passe de 160 bits (et même si les mots de passe sont généralement petits, nous ne souhaitons pas restreindre leur taille). C'est la raison pour laquelle nous segmentons les mots de passe en petits blocs afin de les mettre en gage par petits morceaux. Dans notre description concrète, ces derniers seront de taille 1, ce qui nous aidera à faire la preuve de validité (de chiffrement ElGamal de 1 bit).

Une fois cette première étape achevée, les utilisateurs mettent à nouveau en gage leurs mots de passe. Ce seront ces mises en gage C'_i qui seront utilisées dans tout le reste du protocole. Elles n'ont pas besoin d'être découpées puisque l'on n'extrait rien à partir d'elles, mais on demande aux utilisateurs de prouver qu'elles sont compatibles avec les précédentes. Notons qu'elles

utilisent les trois valeurs $\mathrm{H} = \mathcal{H}(\mathrm{C}_1, \ldots, \mathrm{C}_n)$ (où $\mathcal{H}$ est une fonction de hachage résistante aux collisions), pk et c, comme « étiquettes » (*labels*), afin d'éviter la malléabilité et le rejeu de sessions précédentes, grâce aux preuves SSNIZK qui incluent et vérifient ces étiquettes.

Ensuite, les utilisateurs mettent une nouvelle fois en gage leurs mots de passe, mais cette fois ils réalisent un chiffrement ElGamal de pw_i en base c plutôt qu'en base g. Chaque utilisateur calcule donc $\mathrm{A}_i = (c^{\mathsf{pw}_i} h^{t_i}, g^{t_i})$. La mise en gage C'_i va être utilisée pour vérifier la possibilité du déchiffrement (*ie* que les mots de passe sont cohérents avec $\mathsf{pk} = g^{\mathsf{sk}}$), tandis que A_i va être utilisée pour calculer le déchiffrement c^{sk}. Ceci explique les deux bases g et c dans C'_i et A_i.

Les utilisateurs envoient ces deux dernières mises en gage à tout le monde, ainsi que les preuves SSNIZK que le même mot de passe a été utilisé les trois fois. Ces preuves sont « étiquetées » par H, pk et c, et la vérification par les autres utilisateurs ne va réussir que si leurs étiquettes sont les mêmes. Ceci permet à tous les joueurs de vérifier que tout le monde partage la même clef publique pk et le même chiffré c et évite donc les situations dans lesquelles un leader avec une clef incorrecte obtiendrait un message déchiffré correct (ce qui contredirait la fonctionnalité idéale). Le protocole va donc échouer si H, pk ou c n'est pas partagé par tout le monde, ce qui est le résultat requis par la fonctionnalité. Notons qu'il va aussi échouer si l'adversaire retient ou modifie un message reçu par un utilisateur, même si tout était correct (mots de passe compatibles, même clef publique, même chiffré). Cette situation est modélisée dans la fonctionnalité par le bit b des requêtes de livraison, pour les cas de déni de service.

Après ces étapes de mise en gage, une étape de vérification permet au leader, mais aussi à tous les joueurs, de vérifier que la clef publique et les mots de passe sont bien compatibles. Notons qu'à ce point, tout est devenu publiquement vérifiable si bien que le leader ne va pas être capable de tricher et faire croire aux autres joueurs que tout est correct quand ce n'est pas le cas. La vérification commence par les mises en gage $\mathrm{C}'_i = (\mathrm{C}'^{(1)}_i, \mathrm{C}'^{(2)}_i) = (g^{\mathsf{pw}_i} h^{s_i}, g^{s_i})$, et fait entrer en compte deux « tours de masquage » pour évaluer les deux valeurs $\prod_i \mathrm{C}'^{(1)}_i = g^{\mathsf{sk}} h^{\sum s_i}$ et h à une puissance aléatoire distribuée $\alpha = \sum_i \alpha_i$. Le rapport des valeurs masquées permet d'annuler le terme $h^{\alpha \sum_i s_i}$, laissant $g^{\alpha \mathsf{sk}}$. Un « tour de démasquage » final est appliqué pour retirer l'exposant α et révéler g^{sk}. Ceci termine avec une décision du leader sur l'abandon du protocole (quand les mots de passe sont incompatibles avec la clef publique) ou la poursuite du calcul. Nous insistons sur le fait que chaque utilisateur est capable de vérifier la validité de la décision du leader : une exécution malhonnête ne peut pas continuer sans qu'un joueur honnête ne s'en rende compte (et ne l'abandonne). Notons cependant qu'une exécution honnête peut aussi être arrêtée par un utilisateur si l'adversaire modifie un message qui lui est destiné, comme c'est reflété par le bit b dans la fonctionnalité idéale.

Si le leader décide de poursuivre, les joueurs l'aident dans le calcul de c^{sk}, à nouveau à l'aide de deux tours de masquage et démasquage, à partir des mises en gage A_i. Notons que si à un moment un utilisateur ne parvient pas à envoyer sa valeur à tout le monde (par exemple à cause d'une attaque de type déni de service) ou si l'adversaire modifie un message (dans une attaque de type « *man-in-the-middle* »), le protocole va échouer. Dans le monde idéal, cela signifie que le simulateur pose une requête de livraison avec le bit b fixé à 0. Grâce aux preuves SSNIZK, ce sont exactement les mêmes mots de passe que dans les premières étapes qui doivent être utilisés par les joueurs dans ces étapes de déchiffrement. Ceci implique nécessairement la compatibilité avec la clef publique, mais peut être une condition plus forte.

Observons enfin que tous les tours de masquage dans les étapes de vérification et de calcul pourraient être effectués de manière concurrente plutôt que séquentielle, afin de simplifier le protocole. Notons toutefois que le dernier tour de démasquage de c^{sk} dans l'étape de calcul ne doit être effectué que lorsque l'on sait que la clef publique et les mots de passe engagés sont compatibles, et que les mots de passe sont les mêmes dans les deux suites de mises en gage, c'est-à-dire après que les étapes de vérification ont réussi.

Nous montrons dans la section 12.5 page 183 que nous pouvons *efficacement* simuler ces calculs sans la connaissance des mots de passe, ce qui signifie qu'ils ne révèlent rien de plus sur les autres mots de passe que pk ne révèle déjà. Plus précisément, nous montrons que de tels calculs sont indistinguables pour $\mathcal{A}$ sous l'hypothèse DDH.

Fig. 12.2 – Étapes du protocoles de calcul privé distribué

$(2a)$ $= (1a)$ $\xrightarrow{\{\mathrm{C}_{i,j},\Pi^0_{i,j}\}_j}$

$(2b)$ $= (1b)$ sauf

$\mathrm{C}'_i = \mathsf{Commit}_g^{\mathrm{H},\mathsf{pk},c}(\mathsf{pw}_i, s_i) = (g^{\mathsf{pw}_i} h^{s_i}, g^{s_i}, \mathrm{H}, \mathsf{pk}, c)$

$\mathrm{A}_i = \mathsf{Commit}_c(\mathsf{pw}_i, t_i) = (c^{\mathsf{pw}_i} h^{t_i}, g^{t_i})$

$\Pi^1_i = \mathsf{SSNIZK}^{\mathrm{H},\mathsf{pk},c}_{\mathsf{CDH}}((g, {\mathrm{C}'_i}^{(2)} / \prod_j \mathrm{C}^{(2)}_{i,j}, h, {\mathrm{C}'_i}^{(1)} / \prod_j \mathrm{C}^{(1)}_{i,j}); s_i - \sum_j r_{i,j})$

$\bar{\Pi}^1_i = \mathsf{SSNIZK}(\mathrm{C}'_i \overset{g,c}{\approx} \mathrm{A}_i)$ $\xrightarrow{\mathrm{C}'_i,\mathrm{A}_i,\Pi^1_i,\bar{\Pi}^1_i}$

$(2c)$ $= (1c)$ $\xrightarrow{\gamma^{(1)}_i,\gamma^{(2)}_i,\Pi^2_i}$

$(2d)$ $= (1d)$ $\xrightarrow{h_i,\Pi^3_i}$

$(2e)$ $= (1e)$ $\xrightarrow{\zeta_i,\Pi^4_i}$

$(2f)$ $= (1f)$ $\quad \mathsf{pk} \overset{?}{=} \zeta_1$

$(3a)$ $\delta^{(1)}_0 = \prod_i \mathrm{A}^{(1)}_i = c^{\sum_i \mathsf{pw}_i} h^{\sum_i t_i} \quad \delta^{(2)}_0 = h$

étant donné, pour tout $j = 1, \ldots, i-1$, $\quad (\delta^{(1)}_j, \delta^{(2)}_j, \Pi^5_j)$

vérifier $\Pi^5_j \overset{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((\delta^{(1)}_{j-1}, \delta^{(1)}_j, \delta^{(2)}_{j-1}, \delta^{(2)}_j); \beta_j)$

$\beta_i \overset{\mathrm{R}}{\leftarrow} \mathbb{Z}^*_q \quad \delta^{(1)}_i = (\delta^{(1)}_{i-1})^{\beta_i} \quad \delta^{(2)}_i = (\delta^{(2)}_{i-1})^{\beta_i}$

$\Pi^5_i = \mathsf{SSNIZK}_{\mathsf{CDH}}((\delta^{(1)}_{i-1}, \delta^{(1)}_i, \delta^{(2)}_{i-1}, \delta^{(2)}_i); \beta_i)$ $\xrightarrow{\delta^{(1)}_i,\delta^{(2)}_i,\Pi^5_i}$

$(3b)$ étant donné $\delta^{(1)}_n = c^{\beta \sum_i \mathsf{pw}_i} h^{\beta \sum_i t_i}$ et $\beta^{(2)}_n = h^\beta$

vérifier $\Pi^5_j \overset{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((\delta^{(1)}_{j-1}, \delta^{(1)}_j, \delta^{(2)}_{j-1}, \delta^{(2)}_j); \beta_j)$

$h'_i = (\delta^{(2)}_n)^{t_i} \quad \Pi^6_i = \mathsf{SSNIZK}_{\mathsf{CDH}}((g, {\mathrm{A}_i}^{(2)}, \delta^{(2)}_n, h'_i); t_i)$ $\xrightarrow{h'_i,\Pi^6_i}$

$(3c)$ étant donné, pour tout $j = 1, \ldots, n$, $\quad (h'_j, \Pi^6_j)$

vérifier $\Pi^6_j \overset{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((g, {\mathrm{A}_j}^{(2)}, \delta^{(2)}_n, h'_j); t_j)$

$\zeta'_{n+1} = \delta^{(1)}_n / \prod_j h'_j = c^{\beta \sum_j \mathsf{pw}_j}$

Si $i \neq 1$, étant donné, pour tout $j = n, \ldots, i+1$, $\quad (\zeta'_j, \Pi^7_j)$

vérifier $\Pi^7_j \overset{?}{=} \mathsf{SSNIZK}_{\mathsf{CDH}}((\delta^{(1)}_{j-1}, \delta^{(1)}_j, \zeta'_j, \zeta'_{j+1}); \beta_j)$

$\zeta'_i = (\zeta'_{i+1})^{1/\beta_i} \quad \Pi^7_i = \mathsf{SSNIZK}_{\mathsf{CDH}}((\delta^{(1)}_{i-1}, \delta^{(1)}_i, \zeta'_i, \zeta'_{i+1}); \beta_i)$ $\xrightarrow{\zeta'_i,\Pi^7_i}$

$(3d)$ P_1 obtient $\zeta'_1 = (\zeta'_2)^{1/\beta_1} = c^{\sum \mathsf{pw}_i} = c^{\mathsf{sk}}$

Le protocole de génération de clef (calcul de $\mathsf{pk} = g^{\mathsf{sk}}$) est un cas particulier du protocole de déchiffrement expliqué ci-dessus (calcul de g^{sk}, test $g^{\mathsf{sk}} = \mathsf{pk}$, calcul de $m = c^{\mathsf{sk}}$), mais en plus simple. Effectivement, nous n'avons besoin que d'un seul ensemble de mises en gage pour les dernières étapes de masquage/démasquage, puisque nous omettons toutes les vérifications précédentes (il n'y a rien à vérifier quand la clef est fabriquée pour la première fois).

12.3 Détails de la génération de clef distribuée

Nous décrivons dans cette section le protocole de génération de clef distribuée, qui réalise de manière sûre la fonctionnalité $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ (décrite sur la figure 12.1 page 178).

12.3.1 Première mise en gage (1*a*)

Chaque utilisateur P_i engage son mot de passe pw_i (divisé en ℓ blocs $\mathsf{pw}_{1,1}, \ldots, \mathsf{pw}_{i,\ell}$ de longueur L – dans notre description, L = 1), morceau de la clef secrète sk : il calcule $\mathrm{C}_{i,j} = (\mathrm{C}^{(1)}_{i,j}, \mathrm{C}^{(2)}_{i,j}) = (g^{\mathsf{pw}_{i,j}} h^{r_{i,j}}, g^{r_{i,j}})$, pour $j = 1, \ldots, \ell$, et « publie » (c'est-à-dire essaie d'envoyer à tout le monde) $\mathrm{C}_i = (\mathrm{C}_{i,1}, \ldots, \mathrm{C}_{i,\ell})$, avec des preuves SSNIZK que chaque élément l'engage effectivement sur un bloc de L bits.

12.3.2 Deuxième mise en gage (1*b*)

Chaque utilisateur P_i calcule le label $\mathrm{H} = \mathcal{H}(\mathrm{C}_1, \ldots, \mathrm{C}_n)$, et engage à nouveau son mot de passe pw_i, mais cette fois comme un bloc unique et avec l'« étiquette » H : il calcule $\mathrm{C}'_i = (\mathrm{C}'^{(1)}_i, \mathrm{C}'^{(2)}_i, \mathrm{C}'^{(3)}_i) = (g^{\mathsf{pw}_i} h^{s_i}, g^{s_i}, \mathrm{H})$, et le publie ainsi qu'une preuve SSNIZK que les mots de passe engagés sont les mêmes dans les deux mises en gage. Le langage considéré pour cette preuve d'appartenance est le suivant :

$$\mathrm{L}(g,h,\mathrm{H}) = \left\{ (\{\mathrm{C}_{i,1}, \ldots, \mathrm{C}_{i,\ell}\}, \mathrm{C}'_i) \;\middle|\; \exists (r_{i,1}, \ldots, r_{i,\ell}, s_i) \text{ tel que } \begin{bmatrix} \mathrm{C}^{(2)}_{i,j} = g^{r_{i,j}}, \;\; \mathrm{C}'^{(2)}_i = g^{s_i}, \\ \prod_{j=1}^{\ell} \mathrm{C}^{(1)}_{i,j} / h^{r_{i,j}} = \mathrm{C}'^{(1)}_i / h^{s_i}, \\ \mathrm{C}'^{(3)}_i = \mathrm{H} \end{bmatrix} \right\}$$

Notons que cette définition de L implique l'égalité des mots de passe entre les mises en gage : ils sont présents à l'intérieur de $\mathrm{C}^{(1)}_{i,j}$ et $\mathrm{C}'^{(1)}_i$. La réalisation de ces preuves est décrite sur la figure 12.1, étape (1*b*).

12.3.3 Première étape du calcul (1*c*)

Si l'une des preuves reçues par un utilisateur est incorrecte, il abandonne le jeu. Sinon, ils partagent les mêmes H et C'_i : ils sont donc capables de calculer le même $\gamma_0 = (g^{\sum \mathsf{pw}_i} h^{\sum s_i}, h) = (\gamma_0^{(1)}, \gamma_0^{(2)})$ en multipliant les premières parties des mises en gage C'_i entre elles.

Le leader du groupe P_1 veut calculer $g^{\sum \mathsf{pw}_i} = \mathsf{pk}$. Pour $i = 1, \ldots, n$, séquentiellement, P_i choisit $\alpha_i \in \mathbb{Z}$ aléatoirement et calcule $\gamma_i = (\gamma_i^{(1)}, \gamma_i^{(2)}) = ((\gamma_{i-1}^{(1)})^{\alpha_i}, (\gamma_{i-1}^{(2)})^{\alpha_i})$. Il produit alors une preuve SSNIZK qu'il a utilisé le même α_i dans les calculs de $\gamma_i^{(1)}$ et de $\gamma_i^{(2)}$, à partir de $\gamma_{i-1}^{(1)}$ et de $\gamma_{i-1}^{(2)}$ respectivement, et il publie γ_i accompagné d'une telle preuve, dont le langage est l'égalité suivante de logarithme – voir la figure 12.1, étape (1*c*) :

$$\mathrm{L}_1 = \left\{ (\gamma_i, \gamma_{i-1}) \mid \exists \alpha_i \text{ tel que } \gamma_i^{(1)} = \left(\gamma_{i-1}^{(1)}\right)^{\alpha_i} \text{ et } \gamma_i^{(2)} = \left(\gamma_{i-1}^{(2)}\right)^{\alpha_i} \right\}$$

Si la preuve n'est pas valide, l'utilisateur suivant abandonne. Si tout se passe bien, les utilisateurs vont avoir effectué à la fin un « tour de masquage », dans lequel chaque utilisateur P_i aura contribué avec son propre exposant aléatoire éphémère α_i.

12.3.4 Deuxième étape du calcul (1*d*)

Notons $\alpha = \prod \alpha_i$. Quand les utilisateurs reçoivent le dernier élément à l'issue du tour de masquage $\gamma_n = (g^{\alpha \sum \mathsf{pw}_i} h^{\alpha \sum s_i}, h^{\alpha})$, ils calculent tous et publient la valeur $h_i = h^{\alpha s_i}$, ainsi qu'une preuve SSNIZK que leur valeur aléatoire s_i est la même que celle utilisée dans C'_i. Le langage de cette preuve est à nouveau une égalité de logarithmes discrets – voir la figure 12.1, étape (1*d*) :

$$\mathrm{L}_2(g, h^{\alpha}) = \left\{ (\gamma_n, \mathrm{C}'_i, h_i) \mid \exists s_i \text{ tel que } \mathrm{C}'^{(2)}_i = g^{s_i} \text{ et } h_i = (h^{\alpha})^{s_i} \right\}$$

12.3.5 Troisième étape du calcul (1*e*)

À ce point, chaque utilisateur est capable de calculer $h^{\alpha \sum s_i}$ en multipliant tous les h_i ensemble, et ensuite, en divisant $\gamma_n^{(1)}$ par cette valeur, d'obtenir $g^{\alpha \sum \mathsf{pw}_i} = \zeta_{n+1}$. Ainsi, pour $i = n, \ldots, 2$, séquentiellement, l'utilisateur P_i calcule $\zeta_i = (\zeta_{i+1})^{1/\alpha_i}$, ainsi qu'une preuve SSNIZK que la valeur α_i est la même que précédemment. C'est une preuve d'égalité de logarithmes discrets, dont le langage est – voir la figure 12.1, étape (1*e*) :

$$\mathrm{L}_3 = \left\{ (\gamma_i, \gamma_{i-1}, \zeta_i, \zeta_{i+1}) \mid \exists \alpha_i \text{ tel que } \gamma_i^{(1)} = \left(\gamma_{i-1}^{(1)}\right)^{\alpha_i} \text{ et } \gamma_i^{(2)} = \left(\gamma_{i-1}^{(2)}\right)^{\alpha_i} \text{ et } \zeta_i = (\zeta_{i+1})^{1/\alpha_i} \right\}$$

Chaque joueur publie alors séquentiellement ζ_i et la preuve, autorisant ainsi les joueurs restants à continuer. Ce « tour de démasquage » inverse retire le masquage α dans l'ordre inverse de celui où il a été appliqué.

12.3.6 Dernière étape du calcul (1*f*)

Le dernier joueur à prendre part dans le démasquage est le leader du groupe P_1, qui est donc le premier à obtenir la clef publique démasquée $\zeta_1 = (\zeta_2)^{1/\alpha_1} = g^{\sum \mathsf{pw}_i} = \mathsf{pk}$.

Pour communiquer la clef aux autres, il publie ζ_1 et la preuve SSNIZK reliée. Tous les utilisateurs peuvent alors effectuer la dernière étape de démasquage pour eux-mêmes et être certains que la clef résultante correspond aux mises en gage initiales des mots de passe.

12.4 Détails du déchiffrement distribué

Nous décrivons dans cette section le protocole de déchiffrement distribué, qui réalise de manière sûre la fonctionnalité $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ (décrite sur la figure 12.2 page 180).

12.4.1 Vérification commune de la clef publique (2*a*) − (2*f*)

Ces étapes sont quasiment les mêmes que dans le protocole précédent, excepté (2*b*) qui diffère de (1*b*) sur l'étiquette qui ne contient pas seulement H, mais aussi la clef publique pk et le chiffré c : $\mathrm{C}'_i = (\mathrm{C}'^{(1)}_i, \mathrm{C}'^{(2)}_i, \mathrm{C}'^{(3)}_i, \mathrm{C}'^{(4)}_i, \mathrm{C}'^{(5)}_i) = (g^{\mathsf{pw}_i} h^{s_i}, g^{s_i}, \mathrm{H}, \mathsf{pk}, c)$. Cette étiquette étendue va assurer que tous les joueurs utilisent les mêmes données. Nous anticipons aussi l'objectif de cette fonctionnalité : le calcul de c^{sk}. Chaque utilisateur P_i met donc en gage pw_i en base c, envoyant $\mathrm{A}_i = (\mathrm{A}^{(1)}_i, \mathrm{A}^{(2)}_i) = (c^{\mathsf{pw}_i} h^{t_i}, g^{t_i})$, ainsi qu'une preuve SSNIZK que le même mot de passe pw_i a été utilisé à la fois dans C'_i et A_i, avec les bases g et c.

La preuve est un peu plus complexe, mais consiste en plusieurs preuves d'égalité de logarithmes discrets. On calcule et publie d'abord les éléments suivants, pour un élément $u_i \xleftarrow{\$} \mathbb{Z}_q^*$ aléatoire :

$$\begin{cases} a_i = (\mathrm{C}'^{(1)}_i)^{u_i} = g^{u_i \mathsf{pw}_i} h^{u_i s_i} = g^{\pi_i} h^{\sigma_i} \\ b_i = (\mathrm{C}'^{(2)}_i)^{u_i} = g^{u_i s_i} = g^{\sigma_i} \\ c_i = (\mathrm{A}^{(1)}_i)^{u_i} = c^{u_i \mathsf{pw}_i} h^{u_i t_i} = c^{\pi_i} h^{\tau_i} \\ d_i = (\mathrm{A}^{(2)}_i)^{u_i} = g^{u_i t_i} = g^{\tau_i} \\ \mathrm{G}_i = g^{\pi_i} \\ \Gamma_i = c^{\pi_i} \end{cases}$$

et on publie aussi des preuves SSNIZK assurant que

- le même u_i est utilisé pour calculer a_i, b_i, c_i, d_i à partir de $\mathrm{C}'^{(1)}_i$, $\mathrm{C}'^{(2)}_i$, $\mathrm{A}^{(1)}_i$ et $\mathrm{A}^{(2)}_i$ respectivement :

  $$\mathsf{CDH}(\mathrm{C}'^{(1)}_i, a_i, \mathrm{C}'^{(2)}_i, b_i) \quad \wedge \quad \mathsf{CDH}(\mathrm{C}'^{(1)}_i, a_i, \mathrm{A}^{(1)}_i, c_i) \quad \wedge \quad \mathsf{CDH}(\mathrm{C}'^{(1)}_i, a_i, \mathrm{A}^{(2)}_i, d_i)$$

- le même π_i est utilisé pour calculer G_i et Γ_i, à partir de g et c respectivement : $\mathsf{CDH}(g, \mathrm{G}_i, c, \Gamma_i)$;
- σ_i est en fait mis en gage en base g dans (a_i, b_i) : $\mathsf{CDH}(g, b_i, h, a_i/\mathrm{G}_i)$;
- τ_i est en fait mis en gage en base c dans (c_i, d_i) : $\mathsf{CDH}(g, d_i, h, c_i/\Gamma_i)$.

Si l'une des preuves reçues par un utilisateur est incorrecte, cet utilisateur abandonne le jeu. Sinon, comme ils partagent les valeurs H et C'_i, chaque utilisateur est capable de calculer $\gamma_0 = \left(g^{\sum \mathsf{pw}_i} h^{\sum s_i}, h\right)$ en multipliant les premières parties des mises en gage C'_i entre elles.

Le leader P_1 veut vérifier que $g^{\sum \mathsf{pw}_i} = \mathsf{pk}$. Ceci est effectué à l'aide de deux tours de masquage et démasquage et leurs preuves SSNIZK associées, exactement comme dans l'étape de calcul du protocole précédent : le leader publie à la fin $\zeta_1 = g^{\sum \mathsf{pw}_i}$ (accompagné de la preuve correspondante) et chaque joueur est capable de vérifier si le résultat est correct ou non.

12.4.2 Calcul du leader utilisant la clef privée virtuelle (3*a*) − (3*d*)

Le calcul de c^{sk} commence comme celui de g^{sk}, à l'aide de deux tours de masquage et démasquage (en utilisant des aléas t_i à la place des s_i et des valeurs aléatoires β_i différentes

des α_i) et les suites de preuves SSNIZK correspondantes. Cette fois, les joueurs utilisent les mises en gage en base c, c'est-à-dire les A_i, et le leader obtient à la fin la valeur $c^{\sum \mathsf{pw}_i}$ mais ne la publie pas. Comme le leader commence le calcul et ne publie pas le résultat final $c^{\sum \mathsf{pw}_i}$, il est le seul à apprendre le message obtenu (et même le seul à être au courant que le déchiffrement a réussi, même si une telle information peut être difficile à cacher dans une application de la vie réelle).

Notons que dans le protocole réel, un joueur est `compromised` si l'adversaire $\mathcal{A}$ a deviné un mot de passe compatible dans le premier message. Grâce aux preuves SSNIZK, dans ce cas l'adversaire est le seul capable d'envoyer les messages suivants de manière acceptable.

Les preuves de ces théorèmes se trouvent dans la section 12.5.

Théorème 10. *Soit $\widehat{s\mathcal{F}}_{\mathsf{pwDistPublicKeyGen}}$ l'extension concurrente multi-session de la fonctionnalité de répartition $s\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$. Le protocole de génération de clef distribuée décrit sur la figure 12.1 réalise de manière sûre $\widehat{s\mathcal{F}}_{\mathsf{pwDistPublicKeyGen}}$ pour une génération de clef ElGamal, dans le modèle de la CRS, en présence d'adversaires statiques, sous l'hypothèse que le* DDH *soit difficile dans $\mathbb{G}$, $\mathcal{H}$ soit résistante aux collisions, et des preuves SSNIZK pour le langage* CDH *existent.*

Théorème 11. *Soit $\widehat{s\mathcal{F}}_{\mathsf{pwDistPrivateComp}}$ l'extension concurrente multi-session de la fonctionnalité de répartition $s\mathcal{F}_{\mathsf{pwDistPrivateComp}}$. Le protocole de déchiffrement distribué de la figure 12.2 réalise de manière sûre $\widehat{s\mathcal{F}}_{\mathsf{pwDistPrivateComp}}$ pour un déchiffrement ElGamal, dans le modèle de la CRS, en présence d'adversaires statiques, sous l'hypothèse que le* DDH *soit difficile dans $\mathbb{G}$, $\mathcal{H}$ soit résistante aux collisions, et des preuves SSNIZK pour le langage* CDH *existent.*

Rappelons que ces protocoles sont seulement prouvés sûrs que contre des adversaires statiques. Contrairement aux adversaires adaptatifs, les adversaires statiques ne sont autorisés à corrompre les participants au protocole qu'avant le début de l'exécution du protocole.

12.5 Preuve du protocole

Dans cette section, nous donnons les grandes idées de la preuve que les protocoles décrits sur les figures 12.1 et 12.2 réalisent respectivement les fonctionnalités spécifiées sur les figures 11.1 et 11.2. La preuve ressemble beaucoup à celles des chapitres précédents : les détails peuvent être trouvés dans la version complète de [ABCP09].

La preuve du protocole de génération de clef distribuée est similaire à celle du déchiffrement distribué donnée ci-dessous, avec la simplification supplémentaire qu'il n'y a pas d'étape de vérification et la différence que tous les utilisateurs reçoivent le résultat à la fin (ce qui correspond exactement à ce qui se passe dans le protocole de déchiffrement à la fin de l'étape de vérification, où tout le monde reçoit aussi le résultat). Nous nous contentons donc de donner la preuve du second protocole pour expliquer la simulation. Les simplifications additionnelles impliquées par le fait que l'adversaire reçoive le résultat à la fin seront données en remarques.

12.5.1 Grandes lignes de la preuve

L'objectif de la preuve est de construire, pour tout adversaire réel $\mathcal{A}$, un simulateur $\mathcal{S}$ dans le monde idéal, afin que le comportement de $\mathcal{A}$ dans le monde réel et celui de $\mathcal{S}$ dans le monde idéal soient indistinguables pour tout environnement $\mathcal{Z}$. Les fonctionnalités idéales sont spécifiées dans les figures 11.1 et 11.2, et décrites dans la section 11.1 page 168. Comme on utilise la version à états joints du théorème UC (voir page 78), nous considérons implicitement les extensions multi-sessions de ces fonctionnalités, et remplaçons donc tous les sid par $(\mathsf{sid}, \mathsf{ssid})$. Par simplicité, nous n'utilisons que ssid. Notons que les mots de passe des utilisateurs dépendent de la sous-session considérée. Par souci de simplicité, on les note pw_i, mais on devrait lire implicitement $\mathsf{pw}_{i,ssid}$.

Dans le jeu réel, on sait que le protocole ne peut pas continuer après les deux étapes initiales de mises en gage s'il y a des incohérences (entre les mots de passe pw_i utilisés dans les

mises en gage, et entre les copies de pk et c possédées par les utilisateurs). Toute incohérence va contredire le langage L des preuves SSNIZK, et comme le système de preuve avec *setup* honnête est supposé être *sound*, il est difficile pour quiconque de prouver un énoncé faux (puisque la connaissance de la trappe ne change pas le *setup*, donc c'est indistinguable). De même, les deux étapes de masquage et démasquage servent respectivement à vérifier la cohérence de pk avec les pw_i, et à calculer le résultat final c^{sk}. Pour être précis, la sécurité de ces étapes provient de nos hypothèses : tricher dans le calcul des étapes de masquage et démasquage sans se faire prendre requiert une contrefaçon de la preuve SSNIZK ; tandis que distinguer le déchiffrement final c^{sk} d'une valeur aléatoire par n'importe qui à part le leader du groupe se réduit à la résolution du problème DDH dans $\mathbb{G}$.

12.5.2 Description du simulateur

Cette description est basée sur celles des chapitres précédents. Quand il est initialisé avec le paramètre de sécurité k, le simulateur commence par choisir un élément aléatoire $h = g^x \in \mathbb{G}$, et utilise le simulateur *zero-knowledge* pour obtenir sa CRS γ. Il initialise finalement l'adversaire réel $\mathcal{A}$, en lui donnant la CRS (h, γ).

À partir de là, $\mathcal{S}$ interagit avec l'environnement $\mathcal{Z}$, la fonctionnalité $\widehat{s\mathcal{F}}_{\mathsf{pwDistPrivateComp}}$ et sa sous-routine $\mathcal{A}$. Pour la plus grande part, cette interaction consiste uniquement pour $\mathcal{S}$ à choisir un mot de passe quelconque et suivre le protocole au nom des joueurs honnêtes. En outre, au lieu de suivre la stratégie d'un prouveur honnête, $\mathcal{S}$ utilise le simulateur *zero-knowledge* dans toutes les preuves (ce qui est indistinguable grâce à la propriété *zero-knowledge* du protocole de preuve). Si une session abandonne ou termine, alors $\mathcal{S}$ le rapporte à $\mathcal{A}$.

Rappelons que l'on utilise les fonctionnalités de répartition 5.8.3 de [BCL+05] et que les utilisateurs sont partitionnés en sessions disjointes selon ce qu'ils ont reçu à la première étape. On suppose donc dans la suite que les joueurs ont reçu les mêmes valeurs $\mathrm{C}_{i,j}$. Ceci est particulièrement utile quand $\mathcal{A}$ contrôle un ensemble d'utilisateurs puisque ces mises en gage sont extractibles. Notons en effet que choisir g et h autorise $\mathcal{S}$ à connaître le logarithme discret x de h en base g. Comme les mises en gage sont des chiffrés ElGamal, la connaissance de $\log_g h$ va permettre au simulateur de déchiffrer et donc extraire les mots de passe utilisés par $\mathcal{A}$ dans les mises en gage de la première étape. (La véritable extraction requiert le calcul de logarithmes discrets, mais ceci peut être effectué efficacement en utilisant des méthodes génériques, car les logarithmes discrets à extraire sont les sous-blocs du mot de passe de L bits, qui ont par définition un domaine très petit. En outre, leur petite taille est imposée par une SSNIZK.) Une fois ces mots de passe récupérés, $\mathcal{S}$ pose une requête `TestPwd` à la fonctionnalité. Si la réponse est correcte, cela procure l'équation suivante :

$$g^{\mathrm{P}_i \sum_{\text{honnête}} \mathsf{pw}_i} = \mathsf{pk} \Big/ g^{\mathrm{P}_i \sum_{\text{corrompu}} \mathsf{pw}_i} \tag{12.1}$$

Nous donnons maintenant plus de détails sur la simulation des différents messages. Notons que si quoi que ce soit se passe mal ou si $\mathcal{S}$ reçoit un message formatté différemment de ce qui est attendu par la session, alors $\mathcal{S}$ abandonne la session et prévient $\mathcal{A}$.

Initialisation de la session. À réception du message $(\mathsf{ssid}, \mathsf{Pid}, \mathrm{P}_i, \mathsf{pk}, c, \mathtt{role})$ de la part de $\widehat{s\mathcal{F}}_{\mathsf{pwDistPrivateComp}}$, le simulateur $\mathcal{S}$ commence à simuler une nouvelle session du protocole pour le joueur P_i, le groupe Pid, l'identifiant de session ssid, et la CRS $(h = g^x, \gamma)$. Nous notons ce joueur $(\mathrm{P}_i, \mathsf{ssid})$.

Étape (1*a*). $\mathcal{S}$ choisit au hasard n_h mots de passe pour chacun des n_h joueurs honnêtes, calcule les mises en gage de la première étape, et les envoie accompagnées des preuves correspondantes (et simulées). Comme la mise en gage est calculatoirement *hiding* sous l'hypothèse DDH, ceci est indistinguable d'une exécution réelle. $\mathcal{S}$ apprend alors de la part de la fonctionnalité si les utilisateurs partagent tous les mêmes c et pk ou non. Dans le second cas, $\mathcal{S}$ abandonne le jeu.

Étape (1*b*). Si tous les utilisateurs sont honnêtes, $\mathcal{S}$ pose une requête `Private Computation` accompagnée d'une requête `Leader Decryption Delivery` à la fonctionnalité, qui retourne `completed` ou `error`. Si elle retourne `completed`, alors $\mathcal{S}$ continue à utiliser les $n_h - 1$ derniers mots de passe, mais il fixe la valeur $g^{\mathsf{pw}'_1}$ du leader du groupe P_1 tel qu'ils soient compatibles avec la valeur $\mathsf{pk} = g^{\mathsf{sk}}$ (sans connaître le mot de passe pw'_1 correspondant) : le calcul va donc être correct. $\mathcal{S}$ envoie alors les deuxièmes mises en gage, ainsi que les preuves correspondantes (et simulées). Notons que le simulateur ne va pas apprendre c^{sk}, donc il va devoir mettre en gage une certaine valeur $c^{\mathsf{pw}''_1}$ pour le leader, non compatible avec c^{sk}. Mais cela n'a pas d'importance puisqu'il ne va jamais être révélé et que cela ne va pas faire rater le protocole. Notons aussi que, puisque C_1 et C'_1 ne vont pas être compatibles, $\mathcal{S}$ va devoir prouver un énoncé faux pour P_1, ce qui est indistinguable d'une exécution réelle puisque les preuves sont simulées.

Remarque. Notons que les choses auraient été plus simples pour le protocole de génération de clef. Dans ce cas, le simulateur ne reçoit pas seulement `completed` ou `error`, mais aussi la valeur exacte du résultat $\mathsf{pk} = g^{\mathsf{sk}}$ (analogue de c^{sk} ici, puisqu'il n'y a pas d'étape de vérification dans la génération de clef). Il est donc capable de modifier g^{pw_1} en $g^{\mathsf{pw}'_1}$ sans connaître ce nouveau mot de passe pw'_1 afin que les mots de passe soient compatibles avec la clef publique. Le même argument apparaît ci-dessous.

Si la requête retourne `error`, le simulateur continue à utiliser les mots de passe initiaux (il n'y a pas besoin qu'ils soient compatibles) et envoie les nouvelles mises en gage avec les preuves correspondantes (simulées).

Si certains utilisateurs sont corrompus, $\mathcal{S}$ extrait les mots de passe et pose une requête `TestPwd`. Si la réponse est correcte, $\mathcal{S}$ conserve les $n_h - 1$ premiers mots de passe et change la dernière valeur $g^{\mathsf{pw}_{i_{n_h}}}$ en $g^{\mathsf{pw}'_{i_{n_h}}}$ (sans connaître $g\mathsf{pw}'_{i_{n_h}}$) afin qu'elle soit compatible avec l'équation (12.1) (sans connaître le mot de passe correspondant). À la même étape, $\mathcal{S}$ doit produire une autre série de mises en gage de mots de passe, cette fois comme des chiffrés ElGamal de c^{pw_i} et non de g^{pw_i}. On doit désormais considérer deux cas. Premièrement, si le leader du groupe est attaqué, $\mathcal{S}$ calcule les mises en gage normalement pour les $n_h - 1$ premiers joueurs honnêtes en utilisant les mots de passe simulés. Pour le dernier joueur $\mathrm{P}_{i_{n_h}}$, il pose une requête `Private Computation` ainsi qu'une requête `Leader Computation Delivery` afin de récupérer c^{sk}, et calcule la mise en gage manquante comme un chiffré ElGamal de $c^{\mathsf{pw}''_{i_{n_h}}}$, qui est donné par

$$c^{\mathsf{pw}''_{i_{n_h}}} = c^{\mathsf{sk}} \Big/ \left(c^{\mathrm{P}_i \text{ honnête, } i \neq i_{n_h}} {}^{\sum \mathsf{pw}_i} \; c^{\mathrm{P}_i \text{ corrompu}} {}^{\sum \mathsf{pw}_i} \right)$$

Sinon, si le leader du groupe n'est pas attaqué, le simulateur ne va pas récupérer c^{sk}. Nous procédons alors comme dans le cas honnête, en utilisant des valeurs incorrectes c^{pw_i}.

Finalement, si la requête `TestPwd` est incorrecte, alors comme l'étape de vérification va échouer, $\mathcal{S}$ peut conserver tous les mots de passe (autant en base g qu'en base c), et envoie ces mises en gage ainsi que les preuves correspondantes (simulées).

L'indistinguabilité entre la simulation de cette étape et l'exécution réelle repose sur la non-malléabilité et la propriété de *hiding* calculatoire des mises en gage (qui repose sur l'hypothèse DDH). L'adversaire ne peut pas se rendre compte que les mots de passe ne sont pas les bons, ou que le mot de passe pour l'utilisateur P_1 ou $\mathrm{P}_{i_{n_h}}$ a changé entre les deux étapes de mises en gage.

Étapes suivantes. Tout est fixé à ce point. Si les utilisateurs sont honnêtes et partagent des mots de passe compatibles, alors $\mathcal{S}$ continue honnêtement le protocole, en choisissant des valeurs aléatoires α_i et β_i et en exécutant les quatre tours comme décrits dans le protocole. Rappelons que la toute dernière étape va échouer, sans conséquence sur le résultat final, et uniquement dans la vue du leader du groupe. $\mathcal{S}$ pose finalement une requête `Leader Computation Delivery`, en fixant le bit `b` à 1 si l'exécution a réussi et à 0 sinon (par exemple si certains messages n'ont pas été *oracle-generated*).

S'il y a des joueurs corrompus, qui partagent des mots de passe compatibles avec le reste du groupe, $\mathcal{S}$ continue aussi le jeu honnêtement avec les quatre tours. Tout marche bien si le leader du groupe est attaqué. Sinon, la dernière étape échoue, sans aucune mauvaise conséquence étant donné que le leader termine la boucle (comme avant). Le bit `b` est choisi comme décrit ci-dessus : si quelque chose se passe mal et que le protocole termine, alors $\mathtt{b} = 0$, sinon $\mathtt{b} = 1$.

Si les joueurs ne partagent pas des mots de passe compatibles, alors le simulateur continue honnêtement le protocole (sans connaître les véritables mots de passe des utilisateurs) jusqu'à tant qu'il échoue, à l'étape de vérification. Le simulateur pose alors une requête `Private Computation`, ainsi qu'une requête Leader Computation Delivery, en fixant le bit `b` à 0.

Finalement, la simulation est indistinguable du monde idéal, puisque le leader du groupe reçoit un message correct dans la simulation si et seulement si il reçoit un message correct dans le monde idéal.

REMARQUE. Ce qui rend les preuves plus faciles dans ce nouveau cadre que dans les protocoles d'échange de clefs est qu'à la fin il n'est pas nécessaire de s'assurer que tous les participants obtiennent la même clef dans le monde réel si et seulement si c'est le cas dans le monde idéal. Dans notre cadre, nous n'avons qu'à nous préoccuper de la clef reçue par un unique joueur, en l'occurrence le leader du groupe.

Ceci établit que, étant donné un adversaire $\mathcal{A}$ qui attaque le protocole Π dans le monde réel, nous pouvons construire un simulateur $\mathcal{S}$ qui interagit avec la fonctionnalité $\mathcal{F}$ dans le monde idéal, de telle sorte qu'un environnement ne puisse pas distinguer le monde dans lequel il se situe.

12.6 Instanciation des SSNIZK dans le modèle de l'oracle aléatoire

Nous rappelons rapidement comment construire efficacement les preuves SSNIZK nécessaires dans le modèle de l'oracle aléatoire [BR93]. Rappelons que nous n'avons besoin de prouver que deux types de langages :

- l'égalité de logarithmes discrets, c'est-à-dire que, pour $(g, \mathrm{G}, h, \mathrm{H}) \in \mathbb{G}^4$, on souhaite prouver $\mathsf{CDH}(g, \mathrm{G}, h, \mathrm{H})$, connaissant k tel que $\mathrm{G} = g^k$ et $\mathrm{H} = h^k$;
- des chiffrés ElGamal de 0 ou 1, c'est-à-dire une preuve de disjonction d'égalité de logarithmes discrets. Étant donné $(g, h, \mathrm{G}, \mathrm{H}) \in \mathbb{G}^4$, on veut prouver $\mathsf{CDH}(g, \mathrm{G}, h, \mathrm{H})$ ou $\mathsf{CDH}(g, \mathrm{G}, h, \mathrm{H}/g)$, connaissant k tel que $\mathrm{G} = g^k$ et $\mathrm{H} = h^k$ (pour un chiffré de 0) ou $\mathrm{H} = gh^k$ (chiffré de 1).

Nous autorisons aussi qu'une étiquette ℓ soit incluse dans la preuve, et dans la vérification.

12.6.1 Égalité de logarithmes discrets

Étant donné $(g, h, \mathrm{G} = g^k, \mathrm{H} = h^k)$,

- on commence par choisir $r \xleftarrow{\mathrm{R}} \mathbb{Z}_q^*$, puis on calcule $\mathrm{G}' = g^r$ et $\mathrm{H}' = h^r$;
- on génère alors le challenge $c = \mathcal{H}(\ell, g, h, \mathrm{G}, \mathrm{H}, \mathrm{G}', \mathrm{H}') \in \mathbb{Z}_q$;
- on calcule finalement $s = r - kc \bmod q$.

La preuve consiste en le couple (c, s). Pour la vérifier, on doit d'abord calculer les valeurs espérées pour G' et H' : $\mathrm{G}'' = g^s \mathrm{G}^c$ et $\mathrm{H}'' = h^s \mathrm{H}^c$; et vérifier que $c \stackrel{?}{=} \mathcal{H}(\ell, g, h, \mathrm{G}, \mathrm{H}, \mathrm{G}'', \mathrm{H}'')$. Cette preuve est bien SSNIZK [FP01], dans le modèle de l'oracle aléatoire.

12.6.2 Chiffrement ElGamal de 0 ou 1

Nous souhaitons désormais combiner deux des preuves ci-dessus, afin de montrer que l'une des deux est vraie : étant donné $(g, h, \mathrm{G}, \mathrm{H})$, nous voulons montrer qu'il existe k tel que $\mathrm{G} = g^k$ et soit $\mathrm{H} = h^k$ soit $\mathrm{H} = gh^k$. Supposons que $\mathrm{H} = g^b h^k$, pour $b \in \{0, 1\}$.

- on choisit d'abord $r_b \overset{\mathrm{R}}{\leftarrow} \mathbb{Z}_q$ et l'on calcule $\mathrm{G}'_b = g^{r_b}$ et $\mathrm{H}'_b = h^{r_b}$;
- on choisit aussi $c_{\bar{b}}, s_{\bar{b}} \in \mathbb{Z}_q$, et l'on calcule $\mathrm{G}'_{\bar{b}} = g^{s_{\bar{b}}}\mathrm{G}^{c_{\bar{b}}}$, ainsi que $\mathrm{H}'_{\bar{b}} = h^{s_{\bar{b}}}(\mathrm{H}/g^{\bar{b}})^{c_{\bar{b}}}$;
- on génère alors le challenge $c = \mathcal{H}(\ell, g, h, \mathrm{G}, \mathrm{H}, \mathrm{G}'_0, \mathrm{H}'_0, \mathrm{G}'_1, \mathrm{H}'_1) \in \mathbb{Z}_q$;
- on calcule $c_b = c - c_{\bar{b}} \bmod q$, et $s_b = r_b - kc_b \bmod q$.

La preuve consiste en (c_0, c_1, s_0, s_1). Afin de la vérifier, on calcule d'abord les valeurs espérées pour G'_0, G'_1, H'_0 et H'_1 : $\mathrm{G}''_0 = g^{s_0}\mathrm{G}^{c_0}$, $\mathrm{H}''_0 = h^{s_0}\mathrm{H}^{c_0}$; $\mathrm{G}''_1 = g^{s_1}\mathrm{G}^{c_1}$, et $\mathrm{H}''_1 = h^{s_1}(\mathrm{H}/g)^{c_1}$, et vérifie alors que $c_0 + c_1 \overset{?}{=} \mathcal{H}(\ell, g, h, \mathrm{G}, \mathrm{H}, \mathrm{G}''_0, \mathrm{H}''_0, \mathrm{G}''_1, \mathrm{H}''_1)$.

12.7 Discussion et conclusion

Les deux protocoles explicites construits, pour la génération de clef et le déchiffrement distribué, sont spécifiques au cas du chiffrement ElGamal, et la preuve repose sur l'hypothèse DDH. Mais l'approche décrite dans ces deux chapitres est en fait assez générale et a un certain nombre d'applications. En particulier, les fonctionnalitées idéales définies devraient pouvoir s'appliquer sans changement à la plupart des primitives à clef publique. Une application importante est d'étendre l'IBE pour définir l'*IBE distribué à base de mots de passe*, où le générateur de clef privée est décentralisé sur un ensemble de joueurs, chacun d'eux possédant seulement un petit mot de passe privé de son choix, qui ensemble forment la clef maître.

La première modification est d'introduire de l'aléa dans les fonctions, déterministes jusqu'à présent. On utilise pour cela la même méthode de calcul de c^{sk} à partir d'une clef secrète distribuée $\mathsf{sk} = \langle \mathsf{pw}_1, \ldots, \mathsf{pw}_n \rangle$, en remarquant qu'elle peut aussi permettre de calculer des paires de vecteurs $(\mathbf{c}_i^{\mathsf{sk}}, \mathbf{c}_j^r)$ pour un aléa r auquel tous les joueurs ont contribué – ou, plus précisément, pour $r = \sum_i r_i$ où chaque r_i est initialement mis en gage par chaque joueur, de façon similaire à la mise en gage de leur mot de passe. Grâce aux propriétés *hiding* et *binding* de ces mises en gage (voir page 25), ceci garantit que r est uniforme et imprédictible si au moins un joueur a choisi r_i aléatoirement.

La deuxième modification est de travailler dans des groupes bilinéaires, où l'hypothèse n'est plus le DDH mais le linéaire décisionnel [BBS04]. Ce sont des travaux en cours, nous ne citons ici que les idées principales. L'avantage est de supprimer toutes les preuves SSNIZK des tours de masquage et démasquage, qui sont remplacées par des simples vérification de CDH (possibles par un simple couplage). En revanche, pour les mises en gage, on utilise des SSNIZK construites à partir de couplages bilinéaires, basées sur le système de preuve de Groth-Sahai [GS08] sous l'hypothèse linéaire [BBS04]. Ceci ne permet plus d'ajouter des étiquettes dans les preuves pour réaliser facilement la fonctionnalité de répartition. Nous les avons remplacées par une signature des premiers messages envoyés.

La principale difficulté est ensuite que l'on continue à faire du déchiffrement ElGamal, mais qu'il devient vérifiable quand on passe dans un groupe bilinéaire. L'argument « le leader n'obtient pas le bon résultat mais personne ne s'en rendra compte » ne tient donc plus. On peut résoudre cette difficulté en masquant à nouveau le résultat du masquage (le leader ajoute un exposant de son choix avant de révéler le dernier tour du masquage).

Une fois ces problèmes surmontés, cette extension a un certain nombre d'applications très utiles :

- **IBE Boneh-Franklin (BF) à base de mots de passe [BF01] :** nous devons calculer $d_{\mathsf{id}} = \mathrm{H}(\mathsf{id})^{\mathsf{sk}}$ où $\mathrm{H}(\mathsf{id})$ est le haché public de l'identité d'un utilisateur. Ceci est analogue à c^{sk}, et donc le protocole est inchangé.

- **IBE Boneh-Boyen (BB_1) à base de mots de passe [BB04] :** ici, d_{id} est randomisé et de la forme $({g_0}^{\mathsf{sk}}({g_1}^{\mathsf{id}}g_2)^r, g_3^r)$. Ceci a la forme générale de ce que l'on peut calculer en ajoutant des valeurs éphémères à notre protocole comme expliqué ci-dessus. Plus précisément, ${g_0}^{\mathsf{sk}}$ est une valeur privée, donc le protocole peut être adapté ainsi :
  - Dans les mises en gage, l'utilisateur s'engage aussi (une fois) dans $(2a)$ sur une valeur r_i, qui va être son morceau de r.
  - Jusqu'à $(2f)$, tout marche comme avant afin de vérifier pk (il n'y a pas besoin de vérifier r qui est construit à la volée).
  - Les tours de masquage sont faits en parallèle, l'un pour $({h_0}^{\mathsf{sk}})^\beta$, un deuxième pour $(({h_1}^{\mathsf{id}}h_2)^r)^\beta$, et un troisième pour $({h_3}^r)^\beta$, le CDH étant vérifié pour assurer que le même r est utilisé les deux fois, et le même β_i à chaque fois.
  - Les joueurs obtiennent $({h_0}^{\mathsf{sk}}({h_1}^{\mathsf{id}}h_2)^r)^\beta$ et le tour de démasquage est réalisé globalement pour cette valeur. Un autre tour de démasquage est effectué pour $({h_3}^r)^\beta$, avec la même vérification des exposants β_i.
- **déchiffrement linéaire [BBS04] :** soient $(f = g^{1/x}, g, h = g^{1/y})$ la clef publique d'un schéma de chiffrement linéaire, et (x, y) la clef privée associée. En posant $z = y/x$, cette clef publique peut être vue comme $\mathsf{pk} = (h^z, h^y, h)$ et la clef privée est alors $\mathsf{sk} = (y, z)$. En utilisant ces notations,

$$c = \mathcal{E}_{\mathsf{pk}}(m, r) = (c_1, c_2, c_3) = (f^r, h^s, mg^{r+s})$$

et

$$m = \mathcal{D}_{\mathsf{sk}}(c) = c_3({c_1}^x{c_2}^y)^{-1} = mg^{r+s}g^{-r}g^{-s}$$

Dans le premier protocole, les joueurs doivent utiliser deux mots de passe z_i et y_i pour créer la clef publique pk. Dans le second, les étapes de mise en gage sont doublées pour mettre en gage à la fois z_i et y_i. Dès que pk est vérifiée, les tours de masquage sont effectués séparément, d'un côté pour $({c_1}^x)^\beta$ et de l'autre pour $({c_2}^y)^\beta$. Les joueurs obtiennent $({c_1}^x{c_2}^y)^\beta$ et le tour de démasquage peut être effectué globalement pour cette valeur. Dans les deux tours, le CDH est vérifié pour assurer que le même β_i est utilisé à chaque fois.

Table des figures

Index des notions

Index des notations

Bibliographie

[ABC+06] Michel Abdalla, Emmanuel Bresson, Olivier Chevassut, Bodo Möller, et David Pointcheval. « Provably Secure Password-Based Authentication in TLS ». Dans Ferng-Ching Lin, Der-Tsai Lee, Bao-Shuh Lin, Shiuhpyng Shieh, et Sushil Jajodia, éditeurs, *ASIACCS 06 : 1st Conference on Computer and Communications Security*, pages 35–45. ACM Press, mars 2006.

[ABCP06] Michel Abdalla, Emmanuel Bresson, Olivier Chevassut, et David Pointcheval. « Password-Based Group Key Exchange in a Constant Number of Rounds ». Dans Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, et Tal Malkin, éditeurs, *PKC 2006 : 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 de *Lecture Notes in Computer Science*, pages 427–442. Springer, avril 2006.

[ABCP09] Michel Abdalla, Xavier Boyen, Céline Chevalier, et David Pointcheval. « Distributed Public-Key Cryptography from Weak Secrets ». Dans Stanislaw Jarecki et Gene Tsudik, éditeurs, *PKC 2009 : 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 de *Lecture Notes in Computer Science*, pages 139–159. Springer, mars 2009.

[ABGS07] Michel Abdalla, Jens-Matthias Bohli, Maria Isabel Gonzalez Vasco, et Rainer Steinwandt. « (Password) Authenticated Key Establishment : From 2-Party to Group ». Dans Salil P. Vadhan, éditeur, *TCC 2007 : 4th Theory of Cryptography Conference*, volume 4392 de *Lecture Notes in Computer Science*, pages 499–514. Springer, février 2007.

[ACCP08] Michel Abdalla, Dario Catalano, Céline Chevalier, et David Pointcheval. « Efficient Two-Party Password-Based Key Exchange Protocols in the UC Framework ». Dans Tal Malkin, éditeur, *Topics in Cryptology – CT-RSA 2008*, volume 4964 de *Lecture Notes in Computer Science*, pages 335–351. Springer, avril 2008.

[ACCP09] Michel Abdalla, Dario Catalano, Céline Chevalier, et David Pointcheval. « Password-Authenticated Group Key Agreement with Adaptive Security and Contributiveness ». Dans *AFRICACRYPT '09 : Proceedings of the 2nd International Conference on Cryptology in Africa*, pages 254–271, Berlin, Heidelberg, 2009. Springer-Verlag.

[ACP05] Michel Abdalla, Olivier Chevassut, et David Pointcheval. « One-Time Verifier-Based Encrypted Key Exchange ». Dans Serge Vaudenay, éditeur, *PKC 2005 : 8th International Workshop on Theory and Practice in Public Key Cryptography*, volume 3386 de *Lecture Notes in Computer Science*, pages 47–64. Springer, janvier 2005.

[ACP09] Michel Abdalla, Céline Chevalier, et David Pointcheval. « Smooth Projective Hashing for Conditionally Extractable Commitments ». Dans Shai Halevi, éditeur, *Advances in Cryptology – CRYPTO 2009*, Lecture Notes in Computer Science, pages 671–689. Springer, août 2009.

[AFP05] Michel Abdalla, Pierre-Alain Fouque, et David Pointcheval. « Password-Based Authenticated Key Exchange in the Three-Party Setting ». Dans Serge Vaudenay, éditeur, *PKC 2005 : 8th International Workshop on Theory and*

Practice in Public Key Cryptography, volume 3386 de *Lecture Notes in Computer Science*, pages 65–84. Springer, janvier 2005.

[AIR01] William AIELLO, Yuval ISHAI, et Omer REINGOLD. « Priced Oblivious Transfer : How to Sell Digital Goods ». Dans Birgit PFITZMANN, éditeur, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 de *Lecture Notes in Computer Science*, pages 119–135. Springer, mai 2001.

[AP05] Michel ABDALLA et David POINTCHEVAL. « Simple Password-Based Encrypted Key Exchange Protocols ». Dans Alfred MENEZES, éditeur, *Topics in Cryptology – CT-RSA 2005*, volume 3376 de *Lecture Notes in Computer Science*, pages 191–208. Springer, février 2005.

[AP06] Michel ABDALLA et David POINTCHEVAL. « A Scalable Password-Based Group Key Exchange Protocol in the Standard Model ». Dans Xuejia LAI et Kefei CHEN, éditeurs, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 de *Lecture Notes in Computer Science*, pages 332–347. Springer, décembre 2006.

[AST98] Giuseppe ATENIESE, Michael STEINER, et Gene TSUDIK. « Authenticated Group Key Agreement and Friends ». Dans *ACM CCS 98 : 5th Conference on Computer and Communications Security*, pages 17–26. ACM Press, novembre 1998.

[BB04] Dan BONEH et Xavier BOYEN. « Efficient Selective-ID Secure Identity Based Encryption Without Random Oracles ». Dans Christian CACHIN et Jan CAMENISCH, éditeurs, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 de *Lecture Notes in Computer Science*, pages 223–238. Springer, mai 2004.

[BBS04] Dan BONEH, Xavier BOYEN, et Hovav SHACHAM. « Short Group Signatures ». Dans Matthew FRANKLIN, éditeur, *Advances in Cryptology – CRYPTO 2004*, volume 3152 de *Lecture Notes in Computer Science*, pages 41–55. Springer, août 2004.

[BCK98] Mihir BELLARE, Ran CANETTI, et Hugo KRAWCZYK. « A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols (Extended Abstract) ». Dans *30th Annual ACM Symposium on Theory of Computing*, pages 419–428. ACM Press, mai 1998.

[BCL+05] Boaz BARAK, Ran CANETTI, Yehuda LINDELL, Rafael PASS, et Tal RABIN. « Secure Computation Without Authentication ». Dans Victor SHOUP, éditeur, *Advances in Cryptology – CRYPTO 2005*, volume 3621 de *Lecture Notes in Computer Science*, pages 361–377. Springer, août 2005.

[BCNP04] Boaz BARAK, Ran CANETTI, Jesper Buus NIELSEN, et Rafael PASS. « Universally Composable Protocols with Relaxed Set-Up Assumptions ». Dans *45th Annual Symposium on Foundations of Computer Science*, pages 186–195. IEEE Computer Society Press, octobre 2004.

[BCP01] Emmanuel BRESSON, Olivier CHEVASSUT, et David POINTCHEVAL. « Provably Authenticated Group Diffie-Hellman Key Exchange – The Dynamic Case ». Dans Colin BOYD, éditeur, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 de *Lecture Notes in Computer Science*, pages 290–309. Springer, décembre 2001.

[BCP02a] Emmanuel BRESSON, Olivier CHEVASSUT, et David POINTCHEVAL. « Dynamic Group Diffie-Hellman Key Exchange under Standard Assumptions ». Dans Lars R. KNUDSEN, éditeur, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 de *Lecture Notes in Computer Science*, pages 321–336. Springer, avril / mai 2002.

[BCP02b] Emmanuel BRESSON, Olivier CHEVASSUT, et David POINTCHEVAL. « Group Diffie-Hellman Key Exchange Secure against Dictionary Attacks ». Dans Yuliang ZHENG, éditeur, *Advances in Cryptology – ASIACRYPT 2002*, volume 2501 de *Lecture Notes in Computer Science*, pages 497–514. Springer, décembre 2002.

[BCP03a] Emmanuel BRESSON, Olivier CHEVASSUT, et David POINTCHEVAL. « The Group Diffie-Hellman Problems ». Dans Kaisa NYBERG et Howard M. HEYS, éditeurs,

SAC 2002 : 9th Annual International Workshop on Selected Areas in Cryptography, volume 2595 de *Lecture Notes in Computer Science*, pages 325–338. Springer, août 2003.

[BCP03b] Emmanuel Bresson, Olivier Chevassut, et David Pointcheval. « Security Proofs for an Efficient Password-Based Key Exchange ». Dans Sushil Jajodia, Vijayalakshmi Atluri, et Trent Jaeger, éditeurs, *ACM CCS 03 : 10th Conference on Computer and Communications Security*, pages 241–250. ACM Press, octobre 2003.

[BCP04] Emmanuel Bresson, Olivier Chevassut, et David Pointcheval. « New Security Results on Encrypted Key Exchange ». Dans Feng Bao, Robert Deng, et Jianying Zhou, éditeurs, *PKC 2004 : 7th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2947 de *Lecture Notes in Computer Science*, pages 145–158. Springer, mars 2004.

[BCP07] Emmanuel Bresson, Olivier Chevassut, et David Pointcheval. « A Security Solution for IEEE 802.11's Ad-hoc Mode : Password Authentication and Group Diffie-Hellman Key Exchange ». *International Journal of Wireless and Mobile Computing*, 2(1) :4–13, 2007.

[BCPQ01] Emmanuel Bresson, Olivier Chevassut, David Pointcheval, et Jean-Jacques Quisquater. « Provably Authenticated Group Diffie-Hellman Key Exchange ». Dans *ACM CCS 01 : 8th Conference on Computer and Communications Security*, pages 255–264. ACM Press, novembre 2001.

[BD94] Mike Burmester et Yvo Desmedt. « A Secure and Efficient Conference Key Distribution System (Extended Abstract) ». Dans Alfredo De Santis, éditeur, *Advances in Cryptology – EUROCRYPT'94*, volume 950 de *Lecture Notes in Computer Science*, pages 275–286. Springer, mai 1994.

[BD05] Mike Burmester et Yvo Desmedt. « A Secure and Scalable Group Key Exchange System ». *Information Processing Letters*, 94(3) :137–143, mai 2005.

[BDJR97] Mihir Bellare, Anand Desai, Eric Jokipii, et Phillip Rogaway. « A Concrete Security Treatment of Symmetric Encryption ». Dans *38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE Computer Society Press, octobre 1997.

[BEWS00] Peter Buhler, Thomas Eirich, Michael Waidner, et Michael Steiner. « Secure Password-Based Cipher Suite for TLS ». Dans *ISOC Network and Distributed System Security Symposium – NDSS 2000*. The Internet Society, février 2000.

[BF01] Dan Boneh et Matthew K. Franklin. « Identity-Based Encryption from the Weil Pairing ». Dans Joe Kilian, éditeur, *Advances in Cryptology – CRYPTO 2001*, volume 2139 de *Lecture Notes in Computer Science*, pages 213–229. Springer, août 2001.

[BG89] Donald Beaver et Shafi Goldwasser. « Multiparty Computation with Faulty Majority ». Dans *30th Annual Symposium on Foundations of Computer Science*, pages 468–473. IEEE Computer Society Press, octobre / novembre 1989.

[BG07] Daniel R. L. Brown et Kristian Gjøsteen. « A Security Analysis of the NIST SP 800–90 Elliptic Curve Random Number Generator ». Dans Alfred Menezes, éditeur, *Advances in Cryptology – CRYPTO 2007*, volume 4622 de *Lecture Notes in Computer Science*, pages 466–481. Springer, août 2007.

[BGS06] Jens-Matthias Bohli, Maria Isabel Gonzalez Vasco, et Rainer Steinwandt. « Password-Authenticated Constant-Round Group Key Establishment with a Common Reference String ». Cryptology ePrint Archive, Report 2006/214, 2006. `http://eprint.iacr.org/`.

[BGW88] Michael Ben-Or, Shafi Goldwasser, et Avi Wigderson. « Completeness Theorems for Noncryptographic Fault-Tolerant Distributed Computations ». Dans *20th*

Annual ACM Symposium on Theory of Computing, pages 1–10. ACM Press, mai 1988.

[BLS01] Dan BONEH, Ben LYNN, et Hovav SHACHAM. « Short Signatures from the Weil Pairing ». Dans Colin BOYD, éditeur, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 de *Lecture Notes in Computer Science*, pages 514–532. Springer, décembre 2001.

[BM84] Manuel BLUM et Silvio MICALI. « How to generate cryptographically strong sequences of pseudorandom bits ». *SIAM Journal on Computing*, 13(4) :850–864, 1984.

[BM92] Steven M. BELLOVIN et Michael MERRITT. « Encrypted Key Exchange : Password-Based Protocols Secure against Dictionary Attacks ». Dans *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, mai 1992.

[BM08] Emmanuel BRESSON et Mark MANULIS. « Securing group key exchange against strong corruptions and key registration attacks ». *Int. J. Appl. Cryptol.*, 1(2) :91–107, 2008.

[BMN01] Colin BOYD, Paul MONTAGUE, et Khanh Quoc NGUYEN. « Elliptic Curve Based Password Authenticated Key Exchange Protocols ». Dans *ACISP '01 : Proceedings of the 6th Australasian Conference on Information Security and Privacy*, pages 487–501, London, UK, 2001. Springer-Verlag.

[BMP00] Victor BOYKO, Philip D. MACKENZIE, et Sarvar PATEL. « Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman ». Dans Bart PRENEEL, éditeur, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 de *Lecture Notes in Computer Science*, pages 156–171. Springer, mai 2000.

[Bol03] Alexandra BOLDYREVA. « Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme ». Dans Yvo DESMEDT, éditeur, *PKC 2003 : 6th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2567 de *Lecture Notes in Computer Science*, pages 31–46. Springer, janvier 2003.

[Bom66] Enrico BOMBIERI. « On Exponential Sums in Finite Fields ». Dans *American Journal of Mathematics*, volume 88, pages 71–105. The Johns Hopkins University Press, 1966.

[Boy99] Maurizio Kliban BOYARSKY. « Public-Key Cryptography and Password Protocols : The Multi-User Case ». Dans *ACM CCS 99 : 6th Conference on Computer and Communications Security*, pages 63–72. ACM Press, novembre 1999.

[BPR00] Mihir BELLARE, David POINTCHEVAL, et Phillip ROGAWAY. « Authenticated Key Exchange Secure against Dictionary Attacks ». Dans Bart PRENEEL, éditeur, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 de *Lecture Notes in Computer Science*, pages 139–155. Springer, mai 2000.

[BR93] Mihir BELLARE et Phillip ROGAWAY. « Random Oracles are Practical : A Paradigm for Designing Efficient Protocols ». Dans V. ASHBY, éditeur, *ACM CCS 93 : 1st Conference on Computer and Communications Security*, pages 62–73. ACM Press, novembre 1993.

[BR94] Mihir BELLARE et Phillip ROGAWAY. « Entity Authentication and Key Distribution ». Dans Douglas R. STINSON, éditeur, *Advances in Cryptology – CRYPTO'93*, volume 773 de *Lecture Notes in Computer Science*, pages 232–249. Springer, août 1994.

[BR95] Mihir BELLARE et Phillip ROGAWAY. « Provably Secure Session Key Distribution : The Three Party Case ». Dans *27th Annual ACM Symposium on Theory of Computing*, pages 57–66. ACM Press, mai / juin 1995.

[BR00] Mihir Bellare et Phillip Rogaway. « The AuthA Protocol for Password-based Authenticated Key Exchange ». Contributions to IEEE P1363, mars 2000.

[BS01] Dan Boneh et Igor Shparlinski. « On the Unpredictability of Bits of the Elliptic Curve Diffie-Hellman Scheme ». Dans Joe Kilian, éditeur, *Advances in Cryptology – CRYPTO 2001*, volume 2139 de *Lecture Notes in Computer Science*, pages 201–212. Springer, août 2001.

[BV96] Dan Boneh et Ramarathnam Venkatesan. « Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes ». Dans Neal Koblitz, éditeur, *Advances in Cryptology – CRYPTO'96*, volume 1109 de *Lecture Notes in Computer Science*, pages 129–142. Springer, août 1996.

[BVS07] Jens-Matthias Bohli, Mar&#x00ed ;a Isabel Gonz&#x00e1 ;lez Vasco, et Rainer Steinwandt. « Secure group key establishment revisited ». *Int. J. Inf. Secur.*, 6(4) :243–254, 2007.

[Can01] Ran Canetti. « Universally Composable Security : A New Paradigm for Cryptographic Protocols ». Dans *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145. IEEE Computer Society Press, octobre 2001.

[CCD88] David Chaum, Claude Crépeau, et Ivan Damgård. « Multiparty Unconditionally Secure Protocols ». Dans *20th Annual ACM Symposium on Theory of Computing*, pages 11–19. ACM Press, mai 1988.

[CF01] Ran Canetti et Marc Fischlin. « Universally Composable Commitments ». Dans Joe Kilian, éditeur, *Advances in Cryptology – CRYPTO 2001*, volume 2139 de *Lecture Notes in Computer Science*, pages 19–40. Springer, août 2001.

[CFK+00] R. Canetti, J. Friedlander, S. Konyagin, M. Larsen, D. Lieman, et I. Shparlinski. « On the Statistical Properties of Diffie-Hellman Distributions ». *Israel Journal of Mathematics*, 120 :23–46, 2000.

[CFPZ09] Céline Chevalier, Pierre-Alain Fouque, David Pointcheval, et Sébastien Zimmer. « Optimal Randomness Extraction from a Diffie-Hellman Element ». Dans Antoine Joux, éditeur, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 de *Lecture Notes in Computer Science*, pages 572–589. Springer, avril 2009.

[CGH98] Ran Canetti, Oded Goldreich, et Shai Halevi. « The Random Oracle Methodology, Revisited (Preliminary Version) ». Dans *30th Annual ACM Symposium on Theory of Computing*, pages 209–218. ACM Press, mai 1998.

[Cha83] David Chaum. « Blind Signatures for Untraceable Payments ». Dans David Chaum, Ronald L. Rivest, et Alan T. Sherman, éditeurs, *Advances in Cryptology – CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1983.

[CHH00] Ran Canetti, Shai Halevi, et Amir Herzberg. « Maintaining Authenticated Communication in the Presence of Break-Ins ». *Journal of Cryptology*, 13(1) :61–105, 2000.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, et Philip D. MacKenzie. « Universally Composable Password-Based Key Exchange ». Dans Ronald Cramer, éditeur, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 de *Lecture Notes in Computer Science*, pages 404–421. Springer, mai 2005.

[CI09] Jean-Sébastien Coron et Thomas Icart. « A Random Oracle into Elliptic Curves ». Cryptology ePrint Archive, Report 2009/340, 2009. `http://eprint.iacr.org/`.

[CK01] Ran Canetti et Hugo Krawczyk. « Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels ». Dans Birgit Pfitzmann, éditeur, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 de *Lecture Notes in Computer Science*, pages 453–474. Springer, mai 2001.

[CPP04] Dario CATALANO, David POINTCHEVAL, et Thomas PORNIN. « IPAKE : Isomorphisms for Password-based Authenticated Key Exchange ». Dans Matthew FRANKLIN, éditeur, *Advances in Cryptology – CRYPTO 2004*, volume 3152 de *Lecture Notes in Computer Science*, pages 477–493. Springer, août 2004.

[CPS08] Jean-Sébastien CORON, Jacques PATARIN, et Yannick SEURIN. « The Random Oracle Model and the Ideal Cipher Model Are Equivalent ». Dans David WAGNER, éditeur, *Advances in Cryptology – CRYPTO 2008*, volume 5157 de *Lecture Notes in Computer Science*, pages 1–20. Springer, août 2008.

[CR03] Ran CANETTI et Tal RABIN. « Universal Composition with Joint State ». Dans Dan BONEH, éditeur, *Advances in Cryptology – CRYPTO 2003*, volume 2729 de *Lecture Notes in Computer Science*, pages 265–281. Springer, août 2003.

[CS98] Ronald CRAMER et Victor SHOUP. « A Practical Public Key Cryptosystem Provably Secure Against Adaptive Chosen Ciphertext Attack ». Dans Hugo KRAWCZYK, éditeur, *Advances in Cryptology – CRYPTO'98*, volume 1462 de *Lecture Notes in Computer Science*, pages 13–25. Springer, août 1998.

[CS02] Ronald CRAMER et Victor SHOUP. « Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption ». Dans Lars R. KNUDSEN, éditeur, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 de *Lecture Notes in Computer Science*, pages 45–64. Springer, avril / mai 2002.

[DB06] Ratna DUTTA et Rana BARUA. « Password-Based Encrypted Group Key Agreement ». *International Journal of Network Security*, 3(1) :30–41, juillet 2006. `http://isrc.nchu.edu.tw/ijns`.

[DBS04] Ratna DUTTA, Rana BARUA, et Palash SARKAR. « Provably Secure Authenticated Tree Based Group Key Agreement ». Dans Javier LÓPEZ, Sihan QING, et Eiji OKAMOTO, éditeurs, *ICICS 04 : 6th International Conference on Information and Communication Security*, volume 3269 de *Lecture Notes in Computer Science*, pages 92–104. Springer, octobre 2004.

[DDN00] Danny DOLEV, Cynthia DWORK, et Moni NAOR. « Nonmalleable Cryptography ». *SIAM Journal on Computing*, 30(2) :391–437, 2000.

[DDO+01] Alfredo DE SANTIS, Giovanni DI CRESCENZO, Rafail OSTROVSKY, Giuseppe PERSIANO, et Amit SAHAI. « Robust Non-interactive Zero Knowledge ». Dans Joe KILIAN, éditeur, *Advances in Cryptology – CRYPTO 2001*, volume 2139 de *Lecture Notes in Computer Science*, pages 566–598. Springer, août 2001.

[DH76] Whitfield DIFFIE et Martin E. HELLMAN. « New Directions in Cryptography ». *IEEE Transactions on Information Theory*, 22(6) :644–654, 1976.

[DK05] Yevgeniy DODIS et Jonathan KATZ. « Chosen-Ciphertext Security of Multiple Encryption ». Dans Joe KILIAN, éditeur, *TCC 2005 : 2nd Theory of Cryptography Conference*, volume 3378 de *Lecture Notes in Computer Science*, pages 188–209. Springer, février 2005.

[DKOS01] Giovanni DI CRESCENZO, Jonathan KATZ, Rafail OSTROVSKY, et Adam SMITH. « Efficient and Non-interactive Non-malleable Commitment ». Dans Birgit PFITZMANN, éditeur, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 de *Lecture Notes in Computer Science*, pages 40–59. Springer, mai 2001.

[DN02] Ivan DAMGÅRD et Jesper Buus NIELSEN. « Perfect Hiding and Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor ». Dans Moti YUNG, éditeur, *Advances in Cryptology – CRYPTO 2002*, volume 2442 de *Lecture Notes in Computer Science*, pages 581–596. Springer, août 2002.

[DP06] Cécile DELERABLÉE et David POINTCHEVAL. « Dynamic Fully Anonymous Short Group Signatures ». Dans Phong Q. NGUYEN, éditeur, *Progress in Cryptology -*

VIETCRYPT 06 : 1st International Conference on Cryptology in Vietnam, volume 4341 de *Lecture Notes in Computer Science*, pages 193–210. Springer, septembre 2006.

[DPSW06] Yvo DESMEDT, Josef PIEPRZYK, Ron STEINFELD, et Huaxiong WANG. « A Non-malleable Group Key Exchange Protocol Robust Against Active Insiders ». Dans Sokratis K. KATSIKAS, Javier LOPEZ, Michael BACKES, Stefanos GRITZALIS, et Bart PRENEEL, éditeurs, *ISC 2006 : 9th International Conference on Information Security*, volume 4176 de *Lecture Notes in Computer Science*, pages 459–475. Springer, août / septembre 2006.

[ElG85] Taher ELGAMAL. « A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms ». Dans G. R. BLAKLEY et David CHAUM, éditeurs, *Advances in Cryptology – CRYPTO'84*, volume 196 de *Lecture Notes in Computer Science*, pages 10–18. Springer, août 1985.

[FGH+02] Matthias FITZI, Daniel GOTTESMAN, Martin HIRT, Thomas HOLENSTEIN, et Adam SMITH. « Detectable byzantine agreement secure against faulty majorities ». Dans *21st ACM Symposium Annual on Principles of Distributed Computing*, pages 118–126. ACM Press, juillet 2002.

[FP01] Pierre-Alain FOUQUE et David POINTCHEVAL. « Threshold Cryptosystems Secure against Chosen-Ciphertext Attacks ». Dans Colin BOYD, éditeur, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 de *Lecture Notes in Computer Science*, pages 351–368. Springer, décembre 2001.

[FPSZ06] Pierre-Alain FOUQUE, David POINTCHEVAL, Jacques STERN, et Sébastien ZIMMER. « Hardness of Distinguishing the MSB or LSB of Secret Keys in Diffie-Hellman Schemes ». Dans Michele BUGLIESI, Bart PRENEEL, Vladimiro SASSONE, et Ingo WEGENER, éditeurs, *ICALP 2006 : 33rd International Colloquium on Automata, Languages and Programming, Part II*, volume 4052 de *Lecture Notes in Computer Science*, pages 240–251. Springer, juillet 2006.

[FPZ08] Pierre-Alain FOUQUE, David POINTCHEVAL, et Sébastien ZIMMER. « HMAC is a randomness extractor and applications to TLS ». Dans Masayuki ABE et Virgil D. GLIGOR, éditeurs, *ASIACCS*, pages 21–32. ACM, 2008.

[GKR04] Rosario GENNARO, Hugo KRAWCZYK, et Tal RABIN. « Secure Hashed Diffie-Hellman over Non-DDH Groups ». Dans Christian CACHIN et Jan CAMENISCH, éditeurs, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 de *Lecture Notes in Computer Science*, pages 361–381. Springer, mai 2004.

[GL01] Oded GOLDREICH et Yehuda LINDELL. « Session-Key Generation Using Human Passwords Only ». Dans Joe KILIAN, éditeur, *Advances in Cryptology – CRYPTO 2001*, volume 2139 de *Lecture Notes in Computer Science*, pages 408–432. Springer, août 2001. `http://eprint.iacr.org/2000/057`.

[GL03] Rosario GENNARO et Yehuda LINDELL. « A Framework for Password-Based Authenticated Key Exchange ». Dans Eli BIHAM, éditeur, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 de *Lecture Notes in Computer Science*, pages 524–543. Springer, mai 2003. `http://eprint.iacr.org/2003/032.ps.gz`.

[GL06] Oded GOLDREICH et Yehuda LINDELL. « Session-Key Generation Using Human Passwords Only ». *Journal of Cryptology*, 19(3) :241–340, juillet 2006.

[GM84] Shafi GOLDWASSER et Silvio MICALI. « Probabilistic Encryption ». *Journal of Computer and System Sciences*, 28(2) :270–299, 1984.

[GMR88] Shafi GOLDWASSER, Silvio MICALI, et Ronald L. RIVEST. « A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks ». *SIAM Journal on Computing*, 17(2) :281–308, avril 1988.

[GMR06] Craig GENTRY, Philip MACKENZIE, et Zulfikar RAMZAN. « A Method for Making Password-Based Key Exchange Resilient to Server Compromise ». Dans Cynthia

DWORK, éditeur, *Advances in Cryptology – CRYPTO 2006*, volume 4117 de *Lecture Notes in Computer Science*, pages 142–159. Springer, août 2006.

[GMW87a] Oded GOLDREICH, Silvio MICALI, et Avi WIGDERSON. « How to Play Any Mental Game, or A Completeness Theorem for Protocols with Honest Majority ». Dans Alfred AHO, éditeur, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229. ACM Press, mai 1987.

[GMW87b] Oded GOLDREICH, Silvio MICALI, et Avi WIGDERSON. « How to Prove all NP-Statements in Zero-Knowledge, and a Methodology of Cryptographic Protocol Design ». Dans Andrew M. ODLYZKO, éditeur, *Advances in Cryptology – CRYPTO'86*, volume 263 de *Lecture Notes in Computer Science*, pages 171–185. Springer, août 1987.

[GMW91] Oded GOLDREICH, Silvio MICALI, et Avi WIGDERSON. « Proofs That Yield Nothing But Their Validity Or All Languages in NP Have Zero-Knowledge Proof Systems ». *Journal of the ACM*, 38(3) :691–729, 1991.

[GS08] Jens GROTH et Amit SAHAI. « Efficient Non-interactive Proof Systems for Bilinear Groups ». Dans Nigel P. SMART, éditeur, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 de *Lecture Notes in Computer Science*, pages 415–432. Springer, avril 2008.

[Gür05] N. GÜREL. « Extracting bits from coordinates of a point of an elliptic curve ». Cryptology ePrint Archive, Report 2005/324, 2005. `http://eprint.iacr.org/`.

[HBK00] D. R. HEATH-BROWN et S. KONYAGIN. « New bounds for Gauss sums derived from k^{th} powers, and for Heilbronn's exponential sum ». *Q. J. Math.*, 51(2) :221–235, 2000.

[HILL99] Johan HÅSTAD, Russell IMPAGLIAZZO, Leonid A. LEVIN, et Michael LUBY. « A Pseudorandom Generator from any One-way Function ». *SIAM Journal on Computing*, 28(4) :1364–1396, 1999.

[HK98] Shai HALEVI et Hugo KRAWCZYK. « Public-Key Cryptography and Password Protocols ». Dans *ACM CCS 98 : 5th Conference on Computer and Communications Security*, pages 122–131. ACM Press, novembre 1998.

[HKK06] Dan HOLTBY, Bruce M. KAPRON, et Valerie KING. « Lower bound for scalable Byzantine Agreement ». Dans Eric RUPPERT et Dahlia MALKHI, éditeurs, *25th ACM Symposium Annual on Principles of Distributed Computing*, pages 285–291. ACM Press, juillet 2006.

[HMQ04] Dennis HOFHEINZ et Jörn MÜLLER-QUADE. « Universally Composable Commitments Using Random Oracles ». Dans Moni NAOR, éditeur, *TCC 2004 : 1st Theory of Cryptography Conference*, volume 2951 de *Lecture Notes in Computer Science*, pages 58–76. Springer, février 2004.

[Ica09] Thomas ICART. « How to Hash into Elliptic Curves ». Dans Shai HALEVI, éditeur, *Advances in Cryptology – CRYPTO 2009*, Lecture Notes in Computer Science, pages 303–316. Springer, août 2009.

[IEE01] « IEEE P1363a Committee. IEEE P1363a / D9 — Standard Specifications for Public Key Cryptography : Additional Techniques ». `http://grouper.ieee.org/groups/1363/index.html/`, juin 2001. Draft Version 9.

[IEE04] « IEEE Draft Standard P1363.2. Password-based Public Key Cryptography ». `http://grouper.ieee.org/groups/1363/passwdPK`, mai 2004. Draft Version 15.

[IZ89] R. IMPAGLIAZZO et D. ZUCKERMAN. « How to Recycle Random Bits ». Dans *Proc. of the 30th FOCS*, pages 248–253. IEEE, New York, 1989.

[Jab96] David JABLON. « Strong Password-Only Authenticated Key Exchange ». *Computer Communication Review*, 26(5) :5–26, 1996.

[Jab02] David JABLON. « SRP-4 », 2002. Submission to IEEE P1363.2.

[JV08] Dimitar Jetchev et Ramarathnam Venkatesan. « Bits Security of the Elliptic Curve Diffie-Hellman Secret Keys ». Dans David Wagner, éditeur, *Advances in Cryptology – CRYPTO 2008*, volume 5157 de *Lecture Notes in Computer Science*, pages 75–92. Springer, août 2008.

[Kal05] Yael Tauman Kalai. « Smooth Projective Hashing and Two-Message Oblivious Transfer ». Dans Ronald Cramer, éditeur, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 de *Lecture Notes in Computer Science*, pages 78–95. Springer, mai 2005.

[KLL04] Hyun-Jeong Kim, Su-Mi Lee, et Dong Hoon Lee. « Constant-Round Authenticated Group Key Exchange for Dynamic Groups ». Dans Pil Joong Lee, éditeur, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 de *Lecture Notes in Computer Science*, pages 245–259. Springer, décembre 2004.

[KM06] Neal Koblitz et Alfred Menezes. « Another Look at "Provable Security". II. (Invited Talk) ». Dans Rana Barua et Tanja Lange, éditeurs, *Progress in Cryptology - INDOCRYPT 2006 : 7th International Conference in Cryptology in India*, volume 4329 de *Lecture Notes in Computer Science*, pages 148–175. Springer, décembre 2006.

[KOY01] Jonathan Katz, Rafail Ostrovsky, et Moti Yung. « Efficient Password-Authenticated Key Exchange Using Human-Memorable Passwords ». Dans Birgit Pfitzmann, éditeur, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 de *Lecture Notes in Computer Science*, pages 475–494. Springer, mai 2001.

[KOY02] Jonathan Katz, Rafail Ostrovsky, et Moti Yung. « Forward Secrecy in Password-Only Key Exchange Protocols ». Dans Stelvio Cimato, Clemente Galdi, et Giuseppe Persiano, éditeurs, *SCN 02 : 3rd International Conference on Security in Communication Networks*, volume 2576 de *Lecture Notes in Computer Science*, pages 29–44. Springer, septembre 2002.

[Kra05] Hugo Krawczyk. « HMQV : A High-Performance Secure Diffie-Hellman Protocol ». Dans Victor Shoup, éditeur, *Advances in Cryptology – CRYPTO 2005*, volume 3621 de *Lecture Notes in Computer Science*, pages 546–566. Springer, août 2005.

[KS99] S. V. Konyagin et I. Shparlinski. *Character Sums With Exponential Functions and Their Applications*. Cambridge University Press, Cambridge, 1999.

[KS00] David R. Kohel et Igor E. Shparlinski. « On Exponential Sums and Group Generators for Elliptic Curves over Finite Fields ». Dans *Proceedings of the 4th International Symposium on Algorithmic Number Theory*, volume 1838 de *LNCS*, pages 395–404, 2000.

[KS05] Jonathan Katz et Ji Sun Shin. « Modeling Insider Attacks on Group Key-Exchange Protocols ». Dans Vijayalakshmi Atluri, Catherine Meadows, et Ari Juels, éditeurs, *ACM CCS 05 : 12th Conference on Computer and Communications Security*, pages 180–189. ACM Press, novembre 2005.

[KY03] Jonathan Katz et Moti Yung. « Scalable Protocols for Authenticated Group Key Exchange ». Dans Dan Boneh, éditeur, *Advances in Cryptology – CRYPTO 2003*, volume 2729 de *Lecture Notes in Computer Science*, pages 110–125. Springer, août 2003.

[LHL04] Su-Mi Lee, Jung Yeon Hwang, et Dong Hoon Lee. « Efficient Password-Based Group Key Exchange ». Dans Sokratis K. Katsikas, Javier Lopez, et Günther Pernul, éditeurs, *TrustBus 2004 : Trust and Privacy in Digital Business, 1st International Conference*, volume 3184 de *Lecture Notes in Computer Science*, pages 191–199. Springer, août / septembre 2004.

[LOS+06] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, et Brent Waters. « Sequential Aggregate Signatures and Multisignatures Without Random

Oracles ». Dans Serge VAUDENAY, éditeur, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 de *Lecture Notes in Computer Science*, pages 465–485. Springer, mai / juin 2006.

[LRW02] Moses LISKOV, Ronald L. RIVEST, et David WAGNER. « Tweakable Block Ciphers ». Dans Moti YUNG, éditeur, *Advances in Cryptology – CRYPTO 2002*, volume 2442 de *Lecture Notes in Computer Science*, pages 31–46. Springer, août 2002.

[Luc97] Stefan LUCKS. « Open Key Exchange : How to Defeat Dictionary Attacks Without Encrypting Public Keys ». Dans *Workshop on Security Protocols*, École Normale Supérieure, 1997.

[Mac01] Phil MACKENZIE. « On the Security of the SPEKE Password-Authenticated Key Exchange Protocol », 2001. Cryptology ePrint Archive : Report 2001/057.

[Mac02] Philip D. MACKENZIE. « The PAK suite : Protocols for Password-Authenticated Key Exchange ». Contributions to IEEE P1363.2, 2002.

[NIS07] NIST. « Recommendation for Random Number Generation Using Deterministic Random Bit Generators », mars 2007. NIST Special Publications 800-90. `http://csrc.nist.gov/publications/PubsSPs.html`.

[NP01] Moni NAOR et Benny PINKAS. « Efficient Oblivious Transfer Protocols ». Dans *12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 448–457. ACM-SIAM, janvier 2001.

[NV04] Minh-Huyen NGUYEN et Salil P. VADHAN. « Simpler Session-Key Generation from Short Random Passwords ». Dans Moni NAOR, éditeur, *TCC 2004 : 1st Theory of Cryptography Conference*, volume 2951 de *Lecture Notes in Computer Science*, pages 428–445. Springer, février 2004.

[Pai99] Pascal PAILLIER. « Public-Key Cryptosystems Based on Composite Degree Residuosity Classes ». Dans Jacques STERN, éditeur, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 de *Lecture Notes in Computer Science*, pages 223–238. Springer, mai 1999.

[Rab79] Michael O. RABIN. « Digital signatures and public key functions as intractable as factorization ». Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology, janvier 1979.

[RB89] Tal RABIN et Michael BEN-OR. « Verifiable Secret Sharing and Multiparty Protocols with Honest Majority ». Dans *21st Annual ACM Symposium on Theory of Computing*, pages 73–85. ACM Press, mai 1989.

[RSA78] Ronald RIVEST, Adi SHAMIR, et Leonard ADLEMAN. « A Method for Obtaining Digital Signatures and Public-Key Cryptosystems ». *Commun. ACM*, 21(2) :120–126, 1978.

[Sho97] Victor SHOUP. « Lower Bounds for Discrete Logarithms and Related Problems ». Dans Walter FUMY, éditeur, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 de *Lecture Notes in Computer Science*, pages 256–266. Springer, mai 1997.

[Sho99] Victor SHOUP. « On Formal Models for Secure Key Exchange ». Rapport Technique RZ 3120, IBM, 1999.

[Sho01a] Victor SHOUP. « OAEP Reconsidered ». Dans Joe KILIAN, éditeur, *Advances in Cryptology – CRYPTO 2001*, volume 2139 de *Lecture Notes in Computer Science*, pages 239–259. Springer, août 2001.

[Sho01b] Victor SHOUP. « A Proposal for an ISO Standard for Public Key Encryption ». Cryptology ePrint Archive, Report 2001/112, 2001. `http://eprint.iacr.org/`.

[Sho04] Victor SHOUP. « ISO 18033-2 : An Emerging Standard for Public-Key Encryption ». `http://shoup.net/iso/std6.pdf`, décembre 2004. Final Committee Draft.

[Sho05] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, Cambridge, 2005.

[Sim84] Gustavus J. Simmons. « The Prisoners' Problem and the Subliminal Channel ». Dans David Chaum, éditeur, *Advances in Cryptology – CRYPTO'83*, pages 51–67. Plenum Press, New York, USA, 1984.

[Was03] Lawrence Washington. *Elliptic Curves : Number Theory and Cryptography*. CRC Press, 2003.

[Wei48] André Weil. « Sur les courbes algébriques et les variétés qui s'en déduisent ». Dans *Actualités scientifiques et industrielles, Publications de l'institut de Mathématique de l'université de Strasbourg*, volume 1041, Paris, 1948. Hermann.

[Wu98] Thomas D. Wu. « The Secure Remote Password Protocol ». Dans *ISOC Network and Distributed System Security Symposium – NDSS'98*. The Internet Society, mars 1998.

[Wu02] Thomas Wu. « SRP-6 : Improvements and Refinements to the Secure Remote Password Protocol », 2002. Submission to IEEE P1363.2.

[Yao82a] Andrew C. Yao. « Protocols for Secure Computations ». Dans *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society Press, novembre 1982.

[Yao82b] Andrew C. Yao. « Theory and applications of trapdoor functions ». Dans *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91. IEEE Computer Society Press, novembre 1982.

Résumé

Une propriété fondamentale procurée par la cryptographie est la création de canaux de communication sûrs, c'est-à-dire garantissant l'authentification, l'intégrité et la confidentialité des données transférées. L'authentification, qui permet à plusieurs utilisateurs de se convaincre de l'identité de leurs interlocuteurs, est généralement une étape préalable à la communication proprement dite, et ce procédé est souvent couplé à la génération d'une clef de session secrète qui permet ensuite de chiffrer toute les messages échangés. Nous nous concentrons ici sur un type d'authentification particulier, basé sur des mots de passe.

Nous rappelons tout d'abord les différents cadres de sécurité, ainsi que les protocoles d'échange de clefs existants, en insistant particulièrement sur un nouveau cadre, dit de composabilité universelle. Nous montrons ensuite que deux variantes de protocoles existants restent sûres dans ce contexte, sous des hypothèses de sécurité très fortes, et dans les modèles de l'oracle aléatoire et du chiffrement idéal. Dans un troisième temps, nous étendons une primitive (les *smooth hash functions*) pour obtenir un protocole avec un niveau de sécurité équivalent, mais cette fois dans le modèle standard. Ce dernier aboutit non plus sur une chaîne de bits mais sur un élément de groupe aléatoire. Nous présentons alors un algorithme d'extraction d'aléa à partir d'un tel élément de groupe, pour obtenir finalement une chaîne de bits aléatoire. Enfin, nous montrons comment étendre l'utilisation des mots de passe à des primitives à clef publique en définissant la nouvelle notion de cryptographie distribuée à base de mots de passe.

Mots-clefs : cryptographie, preuve de sécurité, composabilité universelle, mots de passe, authentification, échange de clefs, extraction d'entropie.

Abstract

A fundamental property fulfilled by cryptography is the creation of secure communication channels, which guarantee authentication, integrity and confidentiality of the data transfered. Authentication, which allows several users to be convinced of the identities of their interlocutors, is generally nothing but a preliminary step to the proper communication, and is often coupled with the generation of a secret session key, which then enables the encryption of the following messages. We focus here on a particular type of authentication, based on passwords.

We first recall the different security frameworks, as well as the existing protocols, particularly insisting on the new framework of universal composability. We show next that two variants of existing protocols remain secure in this context, under strong security hypothesis, and in the random oracle and ideal cipher models. In a third step, we extend the smooth hash functions to obtain a protocol with an equivalent level of security, but this time in the standard model. This protocol does not output a bitstring anymore, but a random group element. We then present a randomness extractor from such a group element, to obtain a random bitstring. Finally, we show how to extend the use of passwords to public key primitives, by defining the new notion of distributed cryptography from passwords.

Keywords : cryptography, security proof, universal composability, passwords, authentication, key exchange, key extraction.