Méthodes pour la vérification des protocoles cryptographiques dans le modèle calculatoire

Mathilde Duclos

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial :

Présentée par

Mathilde Duclos

Thèse dirigée par **Yassine Lakhnech**
et codirigée par **Pierre Corbineau**

préparée au sein **Laboratoire VERIMAG**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information et de l'Informatique (MSTII)**

Méthodes pour la vérification des protocoles cryptographiques dans le modèle calculatoire

Thèse soutenue publiquement le **29 janvier 2016**,
devant le jury composé de :

Jean-Guillaume Dumas
Professeur, Université Grenoble-Alpes, Président
Christine Paulin
Professeur, Université Paris Sud, INRIA - Saclay, Rapporteur
Véronique Cortier
Directrice de recherche, CNRS, Nancy, Rapporteur
Bruno Blanchet
Directeur de recherche, INRIA Paris - Rocquencourt, Examinateur
Steve Kremer
Directeur de recherche, INRIA Nancy - Grand Est, Examinateur
Jean-François Monin
Professeur, Université Grenoble-Alpes, Examinateur
Yassine Lakhnech
Professeur, Université Grenoble-Alpes, Directeur de thèse
Pierre Corbineau
Maître de conférence, Université Grenoble-Alpes, Co-Directeur de thèse

Résumé

Les preuves de sécurité pour les systèmes cryptographiques peuvent être effectuées dans différents modèles qui correspondent chacun à des hypothèses de sécurité différentes. Dans le modèle symbolique, on considère qu'un attaquant ne peut deviner aucun secret et qu'il a seulement la possibilité d'appliquer un ensemble prédéfini d'actions. À l'inverse, dans le modèle calculatoire, l'adversaire peut espérer deviner des secrets et appliquer n'importe quelle opération qui se déroule en temps polynomial. Les propriétés de sécurité sont plus dures à établir et à vérifier dans ce dernier modèle.

Au cours des travaux exposés dans cette thèse, nous avons étendu un cadre de travail pour la certification de preuves d'indiscernabilité calculatoire. Ces preuves sont modélisées à l'aide de l'assistant de preuve Coq, et basées sur CIL (Computationnal Indistinguishability Logic), une logique spécialisée pour le modèle calculatoire qui peut être appliquée et pour les primitives et pour les protocoles. Nous montrons comment CIL et sa modélisation en Coq permettent des preuves qui vont au-delà du modèle calculatoire classique dit *boîte noire*, où un adversaire n'effectue que des échanges de type requête/réponse avec le système sans avoir d'information sur l'état du système.

Nous avons étendu CIL pour parvenir à raisonner dans ces modèles. En particulier pour prendre en compte la fuite d'information de systèmes honnêtes, autorisant l'adversaire à connaître partiellement des données secrètes. Pour ce faire, nous nous sommes basés sur le modèle à taille de mémoire bornée, qui limite la quantité d'information pouvant être retrouvée par l'adversaire. Nous avons utilisé cette extension pour la preuve de sécurité d'un protocole d'échange de clefs résistant aux intrusions. En modélisant l'intrusion hors de la logique en elle-même, nous avons fait en sorte que CIL garde une utilisation large et non spécifique au modèle.

De cette façon, nous montrons qu'il est possible de modéliser des fuites d'informations des participants honnêtes de protocoles sans modification majeure de CIL. L'intrusion est prise en compte au niveau de la modélisation du protocole, en ayant soin de ne pas rendre cette preuve trop complexe pour l'utilisateur. Cette stratégie a été choisie pour sa capacité d'adaptation à d'autres modèles de sécurité et constitue la motivation majeure de ces travaux.

Pour mener à bien la preuve de ce protocole, nous avons développé diverses bibliothèques générales (comme la gestion des distributions) ainsi que des procédures permettant d'automatiser les parties les plus triviales des preuves. De plus, nous avons développé une architecture complexe pour modéliser la preuve de ce protocole qui s'articule sur plusieurs primitives différentes, ce qui implique une preuve complexe devant s'articuler autour des propriétés de sécurité de ces primitives.

Il résulte de ces travaux une preuve complexe d'un protocole dans un modèle de sécurité sortant des modèles conventionnels. Cette preuve s'appuie sur les hypothèses de sécurité des primitives utilisées par le protocole, ainsi que sur les règles de CIL pour parvenir à certifier la sécurité du protocole en présence d'agent d'observation.

Mots clefs : cryptographie, sécurité, preuve, méthodes formelles, certification, résistance aux intrusions, modèle calculatoire, modèle à taille de mémoire borné, Coq

Abstract

Security proofs for cryptographic systems can be carried out in different models which reflect different kinds of security assumptions. In the symbolic model, an attacker cannot guess a secret at all and can only apply a pre-defined set of operations, whereas in the computational model, he can hope to guess secrets and apply any polynomial-time operation. Security properties in the computational model are more difficult to establish and to check.

During this work, we improved a framework for certified proofs of computational indistinguishability, written using the Coq proof assistant, and based on CIL, a specialized logic for computational frames that can be applied to primitives and protocols. We demonstrate how CIL and its Coq-formalization allow proofs beyond the classical black-box security framework, where an attacker only uses the input/output relation of the system by executing on chosen inputs without having additional information on the state.

We enlarged CIL to take into account other model derived from the computational model. Specifically, we adapt it to include information leakage from honest machine in the model such that the adversary can retrieve some secret data. This model is called the bounded storage memory model. We illustrate this improvement on the security proof of an intrusion-resilient key exchange protocol. We choose to model the intrusion outside of the logic itself so that CIL would keep a broad and non specific usage.

With this strategy, we show that one may model information leakage from honest machines (and possibly other specificity of other models) without any major modification of CIL. The intrusion is modeled at the core of the proof, while keeping the proof simple to the user. This method is meant to be adaptive to other security models and is the main concern of this work.

To reach this goal, we developped several general libraries (as distribution over bitstrings), and procedures allowing the automation of the most trivial parts of the proof. Moreover, we developed a complex architecture to modelise this protocol proof, which is built upon the security of several different primitives. The result is a complex proof constructed from the security statements of these primitives.

Keywords: cryptography, security, proof, formal methods, certification, intrusion resilience, computationnal model, bounded storage memory model, Coq

À mes parents, ces fabuleux sponsors de la recherche pendant mes jeunes années.

Remerciements

Je voudrais tout d'abord remercier l'ensemble des membres de mon jury, en particulier les rapporteurs Christine Paulin et Véronique Cortier d'avoir accepté de lire ce manuscrit. Ensuite, je voudrais remercier les examinateurs, Bruno Blanchet, Steve Kremer, Jean-François Monin et Jean-Guillaume Dumas pour leur intérêt pour ces travaux.

Mes directeurs de thèse ont toute ma gratitude pour m'avoir encadré tout au long de cette longue thèse. Je remercie grandement mon directeur de thèse, Yassine Lakhnech, pour m'avoir proposé un sujet de thèse à la fois captivant et exigeant. Je ne remercierai jamais assez mon codirecteur de thèse, Pierre Corbineau, qui a eu la patience de reprendre des explications à de nombreuses reprises sur la bibliothèque ALEA ou celle de CIL, et qui a eu le bon sens de me rassurer aux bons moments. Ce travail lui doit beaucoup.

J'aimerais aussi remercier Gilles Barthe (IMDEA software, Madrid) pour toutes les conversations qui m'ont amenée à mieux comprendre CIL et à mieux l'adapter au modèle à mémoire de taille bornée. Je remercie également Pascal Lafourcade (Université d'Auvergne), pour ses nombreuses conversations, ses encouragements toujours présents et sa bonne humeur. De même, j'aimerais remercier Cristian Ené (laboratoire VERIMAG, Grenoble) pour toutes les conversations que nous avons eues sur les protocoles cryptographiques qui m'ont amenée à les regarder sous un angle nouveau.

Je remercie chaleureusement toute l'équipe du laboratoire VERIMAG, qui fut pour moi une seconde maison depuis mes jeunes années. J'ai une pensée pour Nicolas Halbwachs, Susanne Graf et Florence Maraninchi et tous les autres permanents du labo. Je remercierai tout particulièrement Claire Maiza pour ses encouragements et ses relectures multiples de mon manuscrit. La fluidité de celui-ci est aussi due à son exigence quant à ma rédaction. Plus largement, je remercie toute l'équipe DCS du laboratoire (et ses anciens membres), en particulier le pôle sécurité : Marie-Laure Potet, Marion Daubignard, Martin, Jannik pour les conversations que nous avons eues sur la vérification de protocoles et primitives cryptographique. J'aimerais également remercier les secrétaires (Sandrine, Christine, Rosen...) pour nous avoir aidés pendant tout ce temps à ne pas nous laisser ensevelir dans toutes nos démarches administratives, et les administrateurs réseau (Jean-Noël, Philippe Genin et Patrick Fulconis).

Je remercie profondément Julien et (une fois encore) Marion, sans qui je me serais laissée décourager plus d'une fois au cours de cette thèse, que ce soit

dans l'écriture de la preuve, ou dans celle de ce manuscrit.

Je me souviendrai longtemps de l'ambiance du labo, du verifaim des premières années au vericlimb et veriski qui ont occupé mes congés de thèse. Je pourrais citer Laure, Marc, Jacques, Sophie, Valentin, Benoit, Florent, Claude, Pierre, Hugo, Christian, Laurie, Giovanni, Tommaso, Tayeb, Selma, Jérome, Nicolas, mais la liste est encore longue et je ne veux vexer personne.

Pour finir, je voudrais remercier mes parents, dont le soutien n'a jamais failli, ainsi que mes amis (Audrey en particulier) et Vanessa, qui a du supporter mes frustrations au quotidien durant ces longues années.

Table des matières

Table des figures

Liste des tableaux

Table des jeux de sécurité

Chapitre 1

Introduction et motivation

L'Homme a toujours cherché à protéger ses intérêts : les biens matériels tout d'abord, puis les techniques. Ces techniques sont devenues des informations jalousement gardées car elles apportent un avantage à son détenteur, qu'il soit de nature commerciale (secret industriel), stratégique (secret militaire) ou politique, voire même personnelle.

L'apparition de l'écriture a fourni un excellent moyen pour la sauvegarde et le partage d'information. Par la nature même de ces qualités, l'écriture est également devenue un nouveau vecteur de fuite de secrets. Protéger les informations est devenu rapidement un enjeu majeur. Les techniques utilisées ont donc évolué au cours du temps et des avancées scientifiques. Ainsi le plus vieux document chiffré retrouvé à ce jour (selon wikipédia [Wik13]) est une recette secrète gravée sur une tablette d'argile, de laquelle on avait retiré les consonnes et modifié l'orthographe des mots. Elle est datée du XVI^{e} siècle avant J.C.

D'autres moyens furent utilisés, comme la stéganographie (l'art de cacher un message). On trouve dès le VI^{e} siècle avant notre ère quelques procédés : par exemple, Énée le Tacticien liste dans [lTJC] au chapitre 31 plusieurs procédés stéganographiques comme écrire le message sur de fines lamelles de plomb, que l'on enroule sur elles-mêmes et dont on pare une jolie jeune femme. Certaines de ces techniques furent même employées dans des situations critiques. Par exemple, Hérodote écrit qu'alors que Xerxès, roi de Perse, préparait une immense armée pour conquérir la Grèce, Demarate, sparte banni en terre persienne, prévient sa patrie ainsi : il retire la cire d'une tablette, grave son message, et fait fondre de la cire sur celui-ci. La tablette, vierge d'apparence, a pu voyager sans encombres et les populations grecques furent averties à temps : la bataille navale de Salamine eut raison de l'invasion perse.

Ces procédés protègent certes les messages d'un examen rapide, mais dès lors que l'ennemi connait ces méthodes, les risques de voir un message tomber entre de mauvaises mains sont importants. Ces techniques ont donc une durée de vie courte et un pouvoir de protection de l'information limité. On voit naître alors la *cryptographie* (l'art de chiffrer les messages).

Pour assurer à ses messages une protection optimale, et ceci même après interception,

César utilisait un système simple : il décalait les lettres de l'alphabet de 3 positions vers la droite. La méthode de chiffrement (ici le décalage de l'alphabet) est appelé *système cryptographique*, le nombre de positions de décalage est appelé *clef*. De ce fait, 'a' s'écrivait 'd', 'b' s'écrivait 'e' et ainsi de suite. Le chiffré "FHVDU XWLOLVDLW XQ FKLIIUH" se déchiffre alors "CESAR UTILISAIT UN CHIFFRE". Pour quiconque l'intercepte, ce chiffré est illisible si le lecteur ignore la façon dont le message a été chiffré et la clef qu'il faut utiliser. Intuitivement, on remarque que plus le nombre de clefs possibles est grand, plus la clef sera difficile à deviner (et donc plus le chiffré sera dur à casser).

Ce type de chiffrement vit la fin de sa suprématie avec l'invention de l'analyse fréquentielle au cours du IX^{e} siècle par Al Kindi. Cette méthode de *cryptanalyse* (l'art de casser les chiffrements) repose sur l'observation qu'en utilisant un chiffrement de substitution, la lettre chiffrée (par exemple 'H' dans le code césar) apparait autant de fois dans le message chiffré que la lettre claire (ici 'E') dans le message clair original. Ainsi, en comptant la lettre la plus fréquente d'un message, on en déduit qu'elle remplace la lettre la plus fréquente du texte clair. Cette lettre est probablement une des lettres les plus fréquentes du langage utilisé : pour le français, les trois lettres les plus utilisées sont 'e', puis 'a' et 'i'. En testant successivement les clefs associées à ces décalages, on arrive à retrouver le clair à partir du chiffré. Cette méthode s'est avérée extrêmement efficace pour les chiffrements basés sur les substitutions.

Il s'ensuit une véritable course-poursuite entre cryptographes et cryptanalystes : un système cryptographique est considéré comme sûr jusqu'à ce qu'une attaque efficace contre celui-ci ait été trouvée et révélée. Les systèmes utilisés étaient donc de plus en plus complexes et l'une des cryptanalyses les plus célèbres fut celle d'Enigma au cours de la seconde guerre mondiale, machine utilisée par les sous-marins de l'Axe pour communiquer en toute sécurité.

Dans les années 30, un mathématicien polonais, Marian Rejewski a proposé une cryptanalyse basée sur la redondance des informations : l'entête des messages contenait le chiffrement de clef utilisée ce jour-ci par l'opérateur.

Le système s'est avéré néanmoins sûr jusqu'à l'acquisition par les Alliés d'une de ces machines et de son manuel d'utilisation. Sa cryptanalyse, faite par Alan Turing et les analystes de Bletchley park à l'aide des premières machines conçues (par Turing et Gordon Welchman) pour casser des chiffrements, reposait principalement sur deux erreurs d'utilisation :

— les textes chiffrés débutaient toujours par la date et par l'identité de l'émetteur et du destinataire,

— les clefs étaient changées suivant un protocole qui réduisait fortement l'ensemble des clefs possibles.

Il est essentiel de noter qu'une mauvaise utilisation d'un système cryptographique peut le mettre en danger, alors même que le système n'a pas de failles connues. Nous en donnons d'ailleurs un exemple dans la section 3.1.1. [1]

1. Le lecteur curieux trouvera bien d'autres anecdotes sur la cryptographie en feuilletant [Sin99].

Avec l'ère numérique, les débits d'information augmentent de façon exponentielle. Les états, les entreprises et les particuliers utilisent de façon intensive internet pour partager des connaissances, s'informer et réaliser des transactions.

Les informations sensibles (allant de la simple protection de la vie privée à la sécurisation d'une opération bancaire) circulant sur ce réseau accessible à tous, notamment aux pirates informatiques, sont monnaie courante [2]. Le vote en ligne est également en plein essor : en France, les expatriés peuvent voter par ce moyen.

Parallèlement, les nombreuses avancées techniques en informatique permettaient de grandes avancées en cryptanalyse, ce qui accéléra la course-poursuite entre cryptographes et cryptanalystes. Les systèmes de sécurité, qui englobent chiffrement, authentification et intégrité des données, ont atteint une complexité telle qu'il est difficile, même pour des spécialistes de raisonner à leur propos.

Or il est devenu essentiel de pouvoir accorder une confiance importante aux infrastructures et aux moyens de protections de l'information. Cette confiance optimale passe par la preuve que les systèmes utilisés protègent les données sensibles transitant sur le réseau. Le besoin de chiffrement et protocole sûrs (hors de porté des cryptanalystes) se fait sentir. Pour se mettre à l'abri d'une attaque pour le moment inconnue, on cherche à prouver mathématiquement la sécurité du système employé contre toutes les attaques possibles : le simple test de résistance aux failles connues ne suffit plus, comme l'explique Stern dans [Ste03].

Afin de garantir ce niveau de sécurité, il faut utiliser de nouvelles méthodes pour raisonner sur ces systèmes de sécurité. De telles méthodes ont déjà été développées pour les systèmes critiques : ce sont les méthodes formelles (voir [Mon00]). Elles apportent un grand nombre de méthodes et d'outils pour raisonner sur des systèmes d'informations. On cherche à prouver des propriétés (appelées spécifications) de systèmes d'information. Cela va de la résilience à la panne (par exemple dans le cas d'un réseau de capteur) à la correction d'un système.

Les premiers modèles ont été développés pour prendre en compte les propriétés spécifiques de la cryptographie comme le secret, l'authentification ou l'intégrité des données, et raisonner sur ces propriétés. L'un de ces modèles, le modèle calculatoire (comme celui introduit par Goldwasser et Micali [GM84]), présente l'intérêt principal d'être réaliste : on considère tous les objets comme des chaînes de bits. Théoriquement, on peut donc modéliser tout types d'attaques sur les systèmes, ce qui procure une certaine flexibilité du modèle.

Dans ce contexte, Halevi recommande l'utilisation d'outils automatiques pour vérifier les propriétés de sécurité dans [Hal05]. Entre autres, une logique d'indiscernabilité, CIL (pour *Computational Indistinguishability Logic*), a été développée par Barthe et. al. [BDKL10]. Elle repose sur l'observation suivante : s'il est impossible de distinguer le comportement d'un système du comportement souhaité (ou idéal), alors le système original est sûr. Pour être complète, cette logique développe également un volet concernant la proba-

2. Le lundi précédant Noël 2013, le site américain Amazon a annoncé avoir vendu 36.8 millions de biens à travers le monde, ce qui représente autant de transactions à sécuriser.

bilité d'un évènement, qui permet de quantifier la possibilité de provoquer une faille dans le système. CIL reste délibérément haut niveau, ne faisant pas d'hypothèses sur l'implantation de l'adversaire, tout en permettant de travailler sur un modèle de sécurité plus précis lors de la modélisation du système étudié.

Ce travail est une mise à l'épreuve de cette logique. Nous avons volontairement choisi un protocole dont les propriétés de sécurité vont au delà des modèles de sécurité traditionnels, afin de démontrer la capacité de CIL à modéliser de nouveaux modèles. Ainsi, nous démontrons toute l'efficacité de CIL en raisonnant sur les fuites d'information au travers d'agents d'observation (en anglais *spyware*), c'est-à-dire de petits programmes installés sur un ordinateur à l'insu de son utilisateur, renvoyant à un attaquant des informations sur les données stockées (ou produites) par l'ordinateur. Ces propriétés ne sont pas prises en compte dans les modèles de sécurité classiques, qui ne permettent pas à l'adversaire l'accès aux machines honnêtes. On considère que le système cryptographique interagit avec un adversaire par question/réponse uniquement (où aucune fuite d'information sur l'état du système n'est envisagée).

Dans ce travail, nous élargissons CIL dans un premier temps afin de pouvoir l'utiliser pour prouver la sécurité d'un protocole en présence d'agents d'observation. Les hypothèses de travail sont les suivantes :

— L'agent d'observation ne peut envoyer qu'une quantité limitée d'information (sans quoi l'administrateur du système s'apercevrait de cette fuite).

— À un moment, l'agent d'observation sera découvert et mis hors d'état de nuire sur les machines honnêtes participants au protocole. C'est durant ces périodes où les machines honnêtes seront saines que l'on veut montrer la sécurité des échanges (hors de ces périodes, les machines sont de toute façon corrompues par l'agent d'observation).

— On ne considère pas le cas où l'adversaire est capable, par le biais de l'agent d'observation, de modifier l'état ou le comportement des machines.

Nous sommes capables de prendre en compte ces hypothèses dans l'utilisation de CIL, sans modifier outre mesure cette logique. Ainsi, nous pouvons continuer d'utiliser CIL avec d'autres modèles de sécurité.

Nous présentons au cours du chapitre 2 les méthodes formelles, en particulier les éléments de logique nécessaires à la compréhension de l'outil formel que nous utilisons, Coq. Le chapitre 3 est consacré à la cryptographie et ses modèles et preuves de sécurité. Le protocole ayant servi à cette mise à l'épreuve est présenté dans le chapitre 4.2.2. Le chapitre 5 présente la logique d'indiscernabilité mise à l'épreuve et modifiée dans ce travail, dont nous montrons le lien avec les définitions de sécurité asymptotiques et concrètes dans la section 5.4.2. La certification du protocole, majeure partie de ce travail, est présentée dans le chapitre 6. Enfin, nous présentons nos conclusions dans le chapitre 7.

Chapitre 2

Méthodes formelles pour la sécurité des systèmes

Sommaire

La cryptologie a évolué au cours de son histoire au cours de la course poursuite opposant cryptographes (ceux qui crééent les chiffrements) et cryptanalystes (ceux qui tentent de casser ces chiffrements). Mais existe-il des méthodes, des preuves pour savoir si un outil cryptographique est sûr ? Cette question découle d'une autre question, bien plus large : peut-on (et si oui comment) prouver des propriétés pour des systèmes de calculs automatisés ?

C'est une des grandes questions de l'informatique, et ce domaine est en pleine expansion depuis une quarantaine d'années. On appelle l'ensemble des moyens utilisés pour parvenir à une assurance de propriété d'un programme *les méthodes formelles*.

2.1 Motivation

Depuis l'avènement de l'ère du numérique, la façon de vérifier le fonctionnement d'un logiciel n'a que très peu changé et repose essentiellement sur les tests, bien que la façon de développer un logiciel ait énormément évolué. Ainsi, un logiciel courant, n'ayant pas d'application stratégique (peu d'enjeux au niveau humain, sociétal ou économique) connaîtra ce type de cycle de vie en V : on spécifie du plus haut niveau d'abstraction jusqu'au plus bas, ce qui correspond au cahier des charges dans d'autres industries. Puis on teste les réalisations de ces spécifications, depuis les fonctions jusqu'au programme entier, en passant par les modules. Le cycle en V assure une certaine réactivité du développement du logiciel en cas d'erreur et une certaine confiance sur le logiciel obtenu.

Une autre famille de méthodes est courament utilisée : les méthodes agiles qui impliquent l'utilisateur du logiciel tout au long de la conception, afin d'appréhender au mieux ses attentes. Cette organisation met l'accent sur la réactivité des programmeurs et impose des cycles courts et des tests immédiats tout au long du projet.

Ces méthodes, couramment employées en génie logiciel, sont systèmatiquement utilisées lors de la production de logiciels aux enjeux limités, car les tests successifs permettent effectivement de se prémunir des bogues les plus courants.

Cependant, cette confiance est relative : toute la vérification se base sur des tests, et vouloir tester tous les cas possibles d'un logiciel (toutes les entrées possibles par l'utilisateur, par exemple, toutes les valeurs que peuvent renvoyer les capteurs, tous les états de la mémoire possible, etc.) serait illusoire.

C'est pourquoi nous devons nous tourner vers les méthodes formelles, si nous voulons une confiance optimale envers le bon fonctionnement de certains logiciels dits critiques.

Définition 2.1 (Correction d'un programme)**.** La propriété de bon fonctionnement d'un programme est appelée correction d'un programme. Elle caractérise le bon comportement d'un programme en regard de sa spécification.

Les méthodes formelles cherchent à prouver cette correction : on cherche à s'appuyer le plus possible sur les mathématiques, les outils de logiques et de manière plus générale tous les outils nous permettant de raisonner sur les programmes pour nous assurer, avec le plus de confiance possible, du bon fonctionnement, en toutes circonstances, du programme.

Cependant, ces méthodes sont difficiles à mettre en place : elles requièrent plus de temps de développement, un personnel formé pour les mener à terme (ce qui est plutôt rare) et une maintenance dédiée. Dans quels cas utiliser ces méthodes ? En raison de leur coût, on restreint leurs usages aux applications critiques.

Pour qu'un système soit qualifié de critique, il faut qu'il y ait soit :

— un fort impact humain (en terme de vies humaines), avec des applications dans l'aviation, les transports automatisés, les voitures (semi-)automatisées, les chaînes de production de médicaments, les automates de contrôle d'appareils médicaux à rayonnement ;

— un fort impact sociétal, avec par exemple le vote électronique, la protection des données privées ;

— un fort impact économique, avec des applications comme le commerce électronique, le payement par carte bancaire, les transactions par voie électronique (à la bourse, par exemple) ;

— les trois à la fois : la production d'énergie avec le contrôle des centrales nucléaires, les applications militaires...

Quand utiliser des méthodes formelles ? Dès lors que le risque est bien plus grand que le coût des méthodes formelles employées. Si l'erreur informatique n'est pas une option, on aura alors recours aux méthodes formelles les plus poussées qui soient. On peut garder en tête des cas où les bogues informatiques ont couté cher : dans les dernières décennies, on peut retenir l'incident spatial d'Ariane 5, qui a explosé en vol peu après son décollage à cause d'un bogue informatique. On peut aussi parler de la sonde martienne Mars Climate Orbiter, qui s'est désintégrée au cours de son approche, à cause de l'incohérence du système de mesure utilisé : d'un côté des données calculées dans le système anglo-saxon, de l'autre des données interprétées dans le système métrique. Cette dernière erreur aurait pu être évitée avec une spécification soignée du système, et un choix définitif des unités de mesure.

Quelle est la méthode à employer ? Elle dépend du niveau de confiance dont on a besoin pour le système développé. Par exemple, un bogue dans un petit jeu pour terminal mobile (tablettes, téléphones dits intelligents) n'aura d'autres conséquences que d'ennuyer le joueur. Dans ce cas, l'utilisation des méthodes formelles semble coûteuse pour des avantages limités.

En revanche, considérons un logiciel qui gère les données médicales des patients d'un hôpital. L'impact est fort pour les usagers, sans être critique pour autant. Un soin particulier lors des définitions du cahier des charges et de la spécification sera nécessaire. Une preuve sur papier, faite à la main en langage naturel, montrant que la spécification répond effectivement au cahier des charges peut être indiquée. On parle alors de méthodes semi-formelles. Nous ne les détaillerons pas plus avant.

Pour des systèmes critiques ne supportant pas la moindre erreur (la ligne 14 du métro parisien par exemple, qui est entièrement automatisée), une telle preuve papier ne suffit plus : les informaticiens comme les mathématiciens peuvent faire des erreurs coûteuses dans leurs démonstrations, qui peuvent être du reste compliquées à vérifier en raison des raccourcis que peuvent employer les rédacteurs d'une preuve. Il faut donc s'affranchir de la confiance que l'on peut avoir en eux. Pour cela, nous pouvons utiliser des outils informatiques dont les rôles sont :

— générer des preuves reposant sur un nombre fini et petit d'axiomes que l'on connaît et dont on sait qu'ils ne sont pas contradictoires entre eux,

— vérifier ces preuves de façon indépendante.

Ces preuves formelles reposent donc sur la modélisation du système et de ses propriétés, et sur l'outil informatique utilisé. On réduit donc ainsi l'ensemble des parties en lesquelles

il faut avoir confiance, ce qui rend la démonstration plus sûre. Dans ce cas, on parle de certification. Nous détaillerons ces méthodes dans la section 2.2.

Les méthodes formelles ont peu à peu été introduites dans le monde industriel. Il a donc fallu développer des normes pour s'assurer du sérieux de l'utilisation de ces méthodes. Les critères communs (Common Criteria for Information Technology Security Evaluation, voir [Com12]) permettent une harmonisation à l'échelle mondiale d'un niveau de conception formelle ou semi-formelle du point de vue de la sécurité du produit.

2.2 Méthodes formelles

Les différents standards demandent l'utilisation d'un système formel pour les plus hauts niveaux de certification. Ces systèmes formels définissent des modèles mathématiques dans lesquels on prouve les propriétés essentielles du logiciel.

2.2.1 Système formel

Une méthode formelle est l'utilisation d'un système formel (une logique, un outil basé sur une logique, etc..) comme cadre pour un raisonnement. Il faut modéliser à la fois le logiciel et le raisonnement. Le but est d'aboutir à la preuve d'une (ou plusieurs) propriété(s) souhaitée(s) pour le système. Le modèle doit être particulièrement bien conçu, en regard de ce qu'il faut garantir : la terminaison, l'absence d'arrêt intempestif, la vérification de telle ou telle propriété (par ex une machine à voter doit bien compter, ne pas divulguer le vote d'une personne, etc.).

On obtient donc une preuve mathématique que la modélisation du logiciel répond bien aux attentes demandées. Néanmoins, même si la preuve est tangible en soi, il peut rester de grosses différences entre le produit fini (le code exécuté du logiciel) et le modèle sur lequel on travaille. On a en revanche gagné en confiance sur la preuve : les erreurs ne viennent plus de la preuve en soi, mais plutôt de la modélisation du problème ou encore d'une incohérence dans le système formel.

Ces systèmes sont par ailleurs surveillés de très près, et lorsque qu'une preuve est trouvée, il y de très fortes chances qu'elle soit juste.

2.2.2 Méthode B

La méthode B [Abr96] fonctionne sur le principe de raffinements successifs : on part de la spécification pour arriver à la réalisation. On prouve à chaque raffinement qu'il respecte la spécification, et on obtient en bout de chaîne un logiciel respectant le cahier des charges : les chances de provoquer des bogues sont négligeables.

La méthode B fournit donc un modèle mathématique servant à formaliser les spécifications en langage B, mais également un formalisme à utiliser pour effectuer les raffinements successifs, allant jusqu'à la génération de code. Ainsi, on peut accorder une très grande

confiance au logiciel obtenu. De puissants outils comme l'atelier B ont été développés pour aider les développeurs à spécifier puis réaliser les logiciels.

Cette méthode a été mise en place notamment sur les parties critiques du logiciel de pilotage des lignes automatiques 14 et 1 du métro parisien avec le projet METEOR.

2.2.3 Isabelle

Isabelle [Pau89] est un assistant de preuve de programme basé sur une logique d'ordre supérieur avec un petit noyau d'axiomes. Il possède des outils aidant au raisonnement, comme la réécriture de termes, ainsi que plusieurs procédures de décisions. Il a été utilisé à plusieurs reprise pour prouver des théorèmes mathématiques célèbres, comme le théorème de complétude de Gödel [Mar04].

Il a également été utilisé par Hewlett-Packard pour la conception d'une série de serveurs [Cam97], ainsi que par l'équipe NICTA pour la vérification d'un micro noyau [KEH+09], aboutissant chaque fois à la découverte de nombreux bogues.

2.2.4 Coq

Coq est un assistant de preuve de théorèmes basé sur les calculs des constructions [CH88]. À partir de cette théorie, il implémente le langage de programmation fonctionnel à type dépendant Gallina. Il est conçu en deux parties indépendantes. La première est le noyau, qui contient les bases de la théorie des calculs et certifie la validité d'une preuve en vérifiant que l'arbre de preuve est correctement typé. La seconde partie est la blibliothèque de théories déjà prouvées à l'aide de cet algorithme, de nombreuses tactiques aidant à écrire la preuve, ainsi que d'un interpréteur de Gallina.

Coq a en outre la particularité d'offrir une possibilité d'extraire un programme en Caml certifié à partir d'une preuve (la preuve étant en ce cas la construction d'un algorithme répondant à une propriété donnée).

Il a été conçu pour aider à la preuve de théorèmes mathématiques, en fournissant des certificats de preuve pouvant être vérifié par un logiciel indépendant. C'est un outil puissant pour les méthodes formelles, tant par son expressivité pour modéliser des systèmes que pour sa capacité à fournir un code exécutable à partir d'une preuve de satisfaction d'une spécification.

La confiance se base finalement sur :

— la théorie sur laquelle repose Coq,

— le noyau de Coq : ses axiomes et leur cohérence,

— le compilateur de Caml, qui peut théoriquement introduire des bogues, ce qui est très improbable selon les développeurs de Coq eux-même [1],

1. Là encore, Coq repose sur le noyau de Caml, c'est-à-dire les structures et les instructions les plus basiques et éprouvées. Un bogue à ce niveau d'implantation est rarissime, et il est fortement improbable qu'un tel bogue impacte Coq sans impacter les fonctionnalités de Coq ou d'autres logiciels compilés avec caml.

— les axiomes de l'utilisateur, qui devrait en théorie veiller à ce qu'ils soient cohérents avec le noyau et la théorie de Coq [2],

— la plateforme d'exécution du logiciel Coq (dont une erreur d'exécution serait très improbable. On peut toutefois essayer une autre plateforme pour s'assurer de la justesse d'une preuve).

On obtient donc un certificat auquel on peut accorder la plus grande confiance.

2.3 Éléments de logique

Coq est basé sur le calcul des constructions, qui découle lui même du lambda-calcul (noté λ-calcul) et de la théorie des types. Cette théorie a été utilisée pour modéliser la logique d'ordre supérieur, que nous utilisons pour prouver des théorèmes en Coq de manière plus intuitive pour le logicien.

Le calcul des constructions repose sur l'isomorphisme de Curry-Howard, qui définit une relation entre les algorithmes et les preuves de propositions ainsi qu'entre types et propositions. Les algorithmes sont originellement décrits en λ-calcul simplement typé, et l'on peut exprimer une correspondance entre une proposition et un type, ainsi qu'entre un λ-terme et une preuve.

Dans cette théorie, prouver une proposition, c'est apporter un terme dont le type est celui de la proposition. Nous pouvons également voir une preuve comme un algorithme, par exemple une preuve de $A \Rightarrow B$ est un algorithme qui à une preuve de A associe une preuve de B.

Avec un système de déduction et des règles de calculs conçus de façon claire et concordante, Coq a défini un logiciel aidant à la preuve de propositions. Un ensemble d'outils, comme l'interpréteur et les bibliothèques modélisant les objets mathématiques les plus courants ainsi que leur propriétés, permettent à l'utilisateur de s'abstraire en partie des bases théoriques de la conception de Coq.

Dans cette section, nous donnons les éléments logiques nécessaires à la compréhension basique et l'utilisation de Coq.

2.3.1 Logique du premier ordre

Pour être exprimée dans un contexte mathématique clair et précis, une logique se dote d'une sémantique et d'une syntaxe claire, dont nous allons exprimer les bases. Ces fondements sont indispensables pour toute réflexion formelle, d'autant plus si l'on est amené à utiliser des outils pour raisonner à l'aide de ces logiques.

Dans ce travail, nous nous plaçons dans une logique du premier ordre, qui permet de raisonner sur les propriétés et les relations entre des objets, les fonctions et introduit les quantifications sur les variables. Nous tenons pour acquis cette logique, dont on peut facilement trouver les définitions (par exemple dans [DLL12]).

2. En pratique cette vérification est dure à effectuer. Fort heureusement, une telle incohérence entre les axiomes de Coq et les axiomes de l'utilisateur est rare.

2.3.2 Système d'inférences

Les systèmes d'inférences sont basés sur des règles de base ; utilisées pour déduire un ensemble de connaissances sur des axiomes.

Par exemple, la déduction naturelle (présentée en 1935 par Gentzen [Gen35]) est un système d'inférence, dont les règles de base comprennent le *modus ponens* (ou élimination de l'implication).

Définition 2.2 (Règle)**.** Une règle $\mathcal{R}$ est un couple composé d'un ensemble de formules $H_1, ..., H_n$, dites hypothèses ou prémisses, et d'une formule C dite conclusion. On la note ainsi :

$$\frac{H_1 \qquad ... \qquad H_n}{C}\,\mathcal{R}$$

Ces règles peuvent être utilisées par une logique pour construire un raisonnement, mais elles sont également utilisées en théorie des types, comme nous allons le voir dans la section suivante.

2.3.3 λ-calcul simplement typé

Le λ-calcul simplement typé découle de la théorie des types appliqué au λ-calcul. Dans cette théorie, chaque terme possède un type qui délimite l'ensemble des fonctions que l'on peut lui appliquer. Le λ-calcul repose sur la construction de types avec le constructeur $\rightarrow$. On peut ainsi écrire des fonctions sous la forme de λ-termes et composer des fonctions entre elles. Les fonctions sont donc des objets de premier rang : une variable peut être une fonction, et être applicable. Pour éviter des incohérences comme le paradoxe de Russell, la théorie des types a été développée. On attribue donc des types à des variables et des fonctions.

On considère un ensemble de types de bases dits types atomiques.

Définition 2.3 (Type)**.** Les types sont définis inductivement ainsi :

- les types atomiques sont des types,
- si T et U sont des types, alors $T \rightarrow U$ est un type.

Définition 2.4 (Contexte de typage)**.** Un contexte de typage Γ est un ensemble de couples (x, T) où x est une variable et T est un type. On note généralement $x : T$.

On considère que chaque type comporte au moins un élément.

Définition 2.5 (λ-termes)**.** Les termes sont définis inductivement ainsi :

- les variables sont des termes,
- si t et u sont des termes, alors $(t\ u)$ est un terme (c'est l'*application*),
- si x est une variable, T un type et t un terme alors $\lambda x : T.t$ est un terme (c'est l'*abstraction*).

Définition 2.6 (Règles et jugement de typage)**.** Un jugement de typage est l'affirmation d'un type à un terme dans un contexte. C'est un triplet $\Gamma \vdash x : T$ où Γ est un contexte de typage, x est une variable et T est un type. On dit que x a pour type T (ou x est bien typé) dans Γ.

Les règles de typages sont définies pour chaque λ-terme :

Variable $\Gamma, x : T \vdash x : T (\text{Var})$

Abstraction $\dfrac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x.M : T_1 \to T_2} (\text{Abs})$

Application $\dfrac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash x : T_1}{\Gamma \vdash Mx : T_2} (\text{App})$

À l'aide de ces règles, on peut inférer le type des termes. Il peut arriver cependant que le type d'un terme soit ambiguë selon le contexte.

Par exemple, $\lambda x : T.x$ est la fonction identité de type $T \to T$. Si l'on type $(\lambda x.x)(\lambda x.x)$, le terme de droite admet pour type $T \to T$ (T étant un type arbitraire), celui de gauche a pour type $(T \to T) \to (T \to T)$, le type du terme complet étant $T \to T$ (étant donné que l'on applique la fonction identité à une fonction).

2.3.4 La correspondance de Curry-Howard

La correspondance de Curry-Howard établit un lien étroit entre les notions de preuves (en logique intuitionniste) et de programmes. Si l'on considère une formule logique comme un type, alors une preuve est une instance de ce type. De même, on peut considérer la spécification d'un programme comme un type, et une implantation comme une de ces instances y répondant. Partant de cette observation, une preuve et un programme sont des objets de même nature, mais considérés de points de vue différents.

De nombreux systèmes de types ont été élaborés au fil du temps, entre autres le système de Curry[CFC59], le système Automath de De Bruijn [dB70], le système $\mathcal{F}$ de Girard [Gir72] ainsi que la théorie des constructions de Coquand et Huet [CH88].

Si l'on se place en logique du premier ordre (avec ses règles usuelles), l'isomorphisme qui en découle peut être décrit ainsi :

— à la déduction A correspond un objet $x : A$,

— à la déduction $\dfrac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \Rightarrow B} \Rightarrow \text{intro}$ correspond le terme $\lambda x : A.v$ où v correspond à B et $x : A$ correspond à A,

— à la déduction $\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & A \Rightarrow B\end{array}}{B} \Rightarrow \text{elim}$ correspond le terme tu où t correspond à la déduction de $A \Rightarrow B$ et u correspond à la déduction de A.

On peut donc considérer une preuve de $A \Rightarrow B$ comme un algorithme qui à une preuve de A associe une preuve de B : ainsi en notant ϕ l'isomorphisme décrit ci-dessus, on a $\phi(A \Rightarrow B) = \phi(A) \to \phi(B)$.

La conception de Coq repose sur cette correspondance. Coq définit un langage de spécification, appelé Gallina [dt14], qui permet de définir des termes (types de données structurés, fonctions, formules du calcul des prédicats d'ordre supérieur). C'est en vérifiant que le type de ce terme correspond bien à la formule à prouver que Coq certifie la preuve. Tout repose donc sur la formalisation de cet isomorphisme, le choix de la syntaxe et de la sémantique tâchant de s'approcher le plus possible d'une preuve rédigée en langage naturel.

2.4 Conclusion

Nous avons présenté dans ce chapitre les motivations pour l'utilisation des méthodes formelles, ainsi que les bases théoriques sur lesquelles elles sont fondées. Celles-ci sont recommandées dans toutes les situations critiques (que ce soit au niveau de coût financier, sociétal ou en vie humaine).

Ceci implique que dans bien des situations où les systèmes cryptographiques sont employés, les méthodes formelles doivent être utilisées. Les systèmes de sécurité ont développé leurs propres modèles afin de pouvoir raisonner sur les propriétés de sécurité de ces systèmes : intégrité des données, authentification, confidentialité de l'information, et enfin accessibilité des données. Les bases théoriques de la cryptographie et les modèles que nous utilisons pour prouver la sécurité des systèmes sont présentés dans le chapitre suivant.

Chapitre 3

Cryptographie

Sommaire

La cryptographie a longtemps été étudiée et utilisée, mais l'avènement de l'informatique et de la théorie de l'information a constitué une véritable révolution des techniques employées jusqu'alors pour raisonner sur les chiffres. On a pu alors chiffrer et déchiffrer mécaniquement de grands volumes d'information. En contre-partie, ces grands volumes d'information permettent des attaques mécanisées exploitant des similitudes établies entre messages chiffrés, ou entre messages clairs et messages chiffrés.

Nous exposons tout d'abord dans la section 3.1 les bases théoriques utilisées pour construire des chiffrements sûrs, puis dans la section 3.2 nous expliquons les différents modèles mathématiques utilisés pour raisonner et prouver des propriétés de sécurité des systèmes cryptographiques.

3.1 Bases théoriques de la Cryptographie

La cryptographie se distingue d'abord à la fin du XIXè siècle des vieux usages avec la principale loi de Kerckhoff : l'ennemi est tenu de connaître tôt ou tard le système cryptographique utilisé. C'est à dire que l'adversaire dispose de l'algorithme permettant de procéder au chiffrement et déchiffrement d'information. Seule la clef utilisée lui fait défaut. C'est sous cette hypothèse de travail que les cryptographes conçoivent leur système.

Ensuite elle est révolutionnée par la naissance de la théorie de l'information de Claude Shannon qui définit une façon de quantifier la quantité d'information contenue dans un

message. Grâce à cela, on peut définir la confidentialité d'un texte chiffré à partir de la quantité d'information qu'il transporte (si la clef utilisée pour le chiffrer est inconnue). Ainsi, un chiffré parfaitement confidentiel aura une quantité d'information nulle : sans la clef, on ne peut le distinguer d'une suite de symboles aléatoire.

3.1.1 Shannon et la théorie de l'information

Shannon entame la théorisation de l'information avec [Sha48]. Il y définit l'information comme étant observable et mesurable de façon complètement indépendante du support qui sert à transporter cette information (que ce soit du papier ou des ondes radio).

Cette définition repose sur la probabilité qu'un signe (un bit, par exemple) apparaisse dans un message. Shannon se place donc du point de vue de l'incertitude du signal (ou entropie). D'une part, au cours d'un échange de messages standard, on cherche à réduire cette entropie pour que l'information passe de façon sûre (c'est-à dire non ambiguë) à son destinataire. D'autre part, en cryptographie, on cherche à faire en sorte que cette entropie soit maximale pour assurer la confidentialité de ce message.

De là, Shannon définit dans [Sha49] la notion de confidentialité parfaite. La confidentialité d'un message (au travers d'un chiffré) est parfaite si le chiffré n'apporte aucune information sur le message.

Toute l'information repose ainsi sur la clef utilisée. Dans le cadre d'un chiffrement parfait, elle doit être aléatoire et ne pas être réutilisée. Il faut donc maximiser l'entropie des clefs lors de leur génération.

Shannon a ainsi prouvé que le chiffrement "*one-time pad*", qui consiste à ajouter une clef au texte clair, utilisé avec une clef aléatoire de même longueur que le message, avait une confidentialité parfaite. Ce chiffrement est malheureusement impraticable : il demande

— la génération de clefs parfaitement aléatoires (ce qui est loin d'être trivial),

— l'échange de telles clefs, qui doivent être aussi longues que les messages à chiffrer,

— une utilisation (ainsi qu'une conservation et une destruction) attentive de ces clefs, qui ne doivent jamais être compromises, ni être réutilisées.

Shannon souligne ainsi l'aspect critique de la clef d'un chiffrement : si la clef est interceptée, alors tous les chiffrés l'utilisant sont compromis. L'échange de clefs devient alors critique.

3.1.2 La Cryptographie asymétrique

Toute la sécurité d'un système de chiffrement repose sur la clef. Le problème était dorénavant l'échange de cette clef pour garantir la sécurité des envois de messages chiffrés. La solution vint d'un domaine similaire : l'authentification. Needham étudiait l'authentification par mots de passe : ceux-ci étaient à l'origine conservés dans une table contenant les identifiants et leurs mots de passe. Son idée fut de ne plus sauvegarder le mot de passe, mais une empreinte de celui-ci. La vérification consisterait à appliquer la même transformation au mot de passe fourni, et de vérifier que les deux empreintes sont identiques.

La sécurité de ce système repose donc sur la difficulté de retrouver le mot de passe originel à partir de l'empreinte. Une fonction pour laquelle il est extrêmement difficile de retrouver l'antécédent d'une image (autrement que par des essais exhaustifs), tout en pouvant calculer les images facilement est appelée *fonction à sens unique*.

Diffie et Hellman utilisèrent ce type de fonction dans [DH76] pour montrer que l'échange de clef est possible en échangeant des données publiques uniquement. Ce protocole repose sur la facilité de calculer un exposant et la difficulté de calculer un logarithme discret. Ce protocole est le premier système cryptographique *asymétrique* : il combine des clefs publiques et secrètes. Nous présentons ce protocole un peu plus loin, dans la section 3.1.2.

A partir de là, on utilise des problèmes connus pour leur difficulté calculatoire pour concevoir des systèmes de cryptographie (le plus souvent asymétriques). Le but est simple : à partir d'un élément secret (la clef privée), construire une instance dure d'un problème à résoudre. Le chiffrement asymétrique le plus connu est RSA, qui repose sur la difficulté à factoriser le produit de deux grands entiers.

Comme la clef de chiffrement est dorénavant publique, les systèmes sont exposés aux attaques à texte clairs connus : si le chiffrement est déterministe, alors "oui" sera toujours chiffré de la même façon. Pour contrer ces attaques, on utilise un masquage probabiliste du message chiffré.

Cryptographie probabiliste

Cette dernière innovation en cryptographie est apparue dans les années 80. Il suffit de remarquer que la seule relation obligatoire pour un système de chiffrement $S = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ (où $\mathcal{K}$ est l'algorithme générateur de clefs, $\mathcal{E}$ est l'algorithme de chiffrement et $\mathcal{D}$ est l'algorithme de déchiffrement) est la *correction fonctionnelle* du système :

$$\forall m, (sk, pk) \xleftarrow{\$} \mathcal{K}(\eta) ; \mathcal{D}_{sk}(\mathcal{E}_{pk}(m)) = m$$

Autrement dit, peu importe que le chiffrement ne soit pas déterministe, du moment que son déchiffrement renvoie le message original. Goldwasser et Micali proposent ainsi un schéma de chiffrement probabiliste [GM84], ainsi que ElGamal [ElG85] et Paillier [Pai99].

Exemple d'une attaque : le protocole Diffie-Hellman

Protocole Diffie et Hellman utilisèrent des fonctions à sens-unique dans [DH76] pour montrer que l'échange de clef est possible en échangeant des données publiques uniquement. Ce protocole repose sur la difficulté de calculer un logarithme discret, combiné à la facilité de calculer une puissance.

Soit G un groupe abélien d'ordre premier p et d'un générateur de ce groupe g. Supposons qu'Alice et Bob souhaitent mettre en place une clef commune secrète, voici comment ils opèrent :

— Alice et Bob choisissent chacun une clef secrète (respectivement $a, b \in \{0, \cdots, p-2\}$),

— Alice envoie sa clef publique $A \stackrel{\text{def}}{=} g^a \mod p$ à Bob,

— Bob envoie sa clef publique $B \stackrel{\text{def}}{=} g^b \mod p$ à Alice,

— Alice peut calculer la clef commune $K \stackrel{\text{def}}{=} B^a = g^{a \cdot b} \mod p$,

— Bob peut calculer la clef commune $K \stackrel{\text{def}}{=} A^b = g^{a \cdot b} \mod p$.

Connaissant uniquement A et B, retrouver K est difficile, selon l'hypothèse de décision Diffie-Hellman. Une façon de retrouver K serait de calculer les logarithmes de A et B, ce qui est un problème notoirement complexe. [1] Retrouver K à partir des données publiques étant réputé dur, le système est sûr, c'est à dire que personne ne pourra calculer rapidement la clef à partir de cet échange.

Attaque : l'homme du milieu (ou *man in the middle*) Si l'adversaire (Eve) est capable d'intercepter et d'envoyer ses propres messages, alors elle peut exploiter une faille du protocole. Elle peut se faire passer soit pour Alice, soit pour Bob.

— Alice et Bob conviennent d'un groupe abélien fini G d'ordre premier p et d'un générateur de ce groupe g (qui sont des données publiques),

— Alice et Bob choisissent chacun une clef secrète (respectivement $a, b \in \{0, \cdots, p-2\}$), ainsi que Eve ($m \in \{0, \cdots, p-2\}$).

— Alice envoie sa clef publique $A \stackrel{\text{def}}{=} g^a$ à Bob, mais elle est interceptée par Eve.

— Bob envoie sa clef publique $B \stackrel{\text{def}}{=} g^b$ à Alice, mais elle est interceptée par Eve.

— Eve envoie à Alice et Bob sa clef publique, $M \stackrel{\text{def}}{=} g^m$.

— Alice peut calculer la clef commune $K' \stackrel{\text{def}}{=} M^a = g^{a \cdot m}$,

— Bob peut calculer la clef commune $K'' \stackrel{\text{def}}{=} M^b = g^{m \cdot b}$,

— Eve calcule les deux clefs secrètes correspondantes, K' et K''.

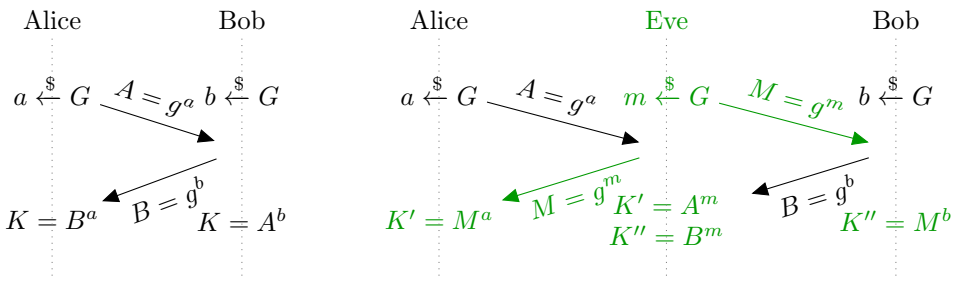
Eve peut continuer d'intercepter tous les messages, les déchiffrer et les transmettre (ou non) à leur destinataire. Le protocole et son attaque sont illustrés dans la figure 3.1.

Il faut donc assortir tout schéma cryptographique d'une preuve de sécurité, pour s'assurer que le protocole résiste bien aux types d'attaques les plus connus.

3.2 Cryptographie prouvable

Les primitives (par exemple les algorithmes de chiffrement) et les protocoles (par exemple pour échanger une clef) cryptographiques sont impliqués dans une variété d'applications critiques : les banques en ligne, le vote électronique, la vente aux enchères électroniques... En tant que tels, ils doivent être formellement vérifiés et prouvés [Ste03], en utilisant si possible des outils dédiés [Hal05]. Les preuves usuelles qui assurent que les propriétés requises sont effectivement satisfaites sont des preuves mathématiques.

1. Ce problème est différent car l'attaquant doit retrouver la clef K (et non le logarithme) avec toutes les données publiques : on ne sait pas si ces deux problèmes sont équivalents.



A gauche, on donne une session ordinaire du protocole. À droite, on illustre le protocole lors de l'attaque de l'homme du milieu. L'adversaire apparait en vert, ainsi que les clefs corrompues.

FIGURE 3.1 – Protocole Diffie-Hellman et l'attaque de l'homme du milieu.

En tant que preuves mathématiques, elles reposent sur un modèle mathématique (comme conseillé par [GM84]). L'un des types de modèles le plus utilisé à ce jour est dit boîte noire. Dans ces modèles, l'adversaire a accès aux primitives comme à des boîtes noires : l'adversaire peut seulement obtenir des réponses à ses requêtes sans pouvoir observer l'état de la machine exécutant l'algorithme. Combiné au modèle calculatoire, l'adversaire a la liberté d'effectuer tout calcul qu'il souhaite, pour autant qu'il se termine en temps polynomial du paramètre de sécurité (habituellement la taille de la clef).

Par exemple, pour un protocole de cryptologie dont le but est d'échanger une clef entre deux parties, on définit le modèle ainsi :

— les deux parties disposent d'un espace mémoire et d'une capacité de calcul raisonnables (qui terminent en temps polynomial par rapport au paramètre de sécurité) mais non illimités,

— l'adversaire (qui cherche à connaître la clef, ou à en imposer une) dispose de ces mêmes ressources,

— toutes ces parties peuvent communiquer sur un réseau commun (où ils ne sont pas authentifiés)

— l'adversaire a une totale maîtrise de ce réseau : il peut à sa guise retarder, modifier voire empêcher d'arriver tous les messages qui sont envoyés.

Ce modèle est volontairement pessimiste pour garantir la sécurité du protocole, s'il existe une preuve de sa sécurité dans ce modèle. Cependant, il est possible qu'il ne soit pas aussi complet que ce que l'on veut atteindre en termes de sécurité : par exemple, que peut-on dire du protocole si l'adversaire est capable d'envoyer un agent d'observation chez les parties honnêtes ? Il devient nécessaire de définir un modèle de sécurité pour chaque famille de protocoles.

Toutefois, si les propriétés de chaque protocole peuvent être très différentes, il y a un cadre et un esprit propre aux preuves cryptographiques. Dolev et Yao dans [DY83] et par Goldwasser et Micali dans [GM84] fondent les deux modèles sur lesquels sont basées les

preuves de sécurité des systèmes cryptographiques : les modèles symboliques et les modèles calculatoires.

3.2.1 Modèles symboliques

Dans les modèles symboliques, par exemple celui présenté par Dolev et Yao dans [DY83], on considère les primitives cryptographiques comme des boîtes noires dont on suppose qu'elles sont parfaitement sûres. Cela impose par exemple de posséder la clef privée pour pouvoir déchiffrer un message, ou signer un document. L'adversaire est considéré comme ayant tous les pouvoirs sur le réseau : il peut laisser transiter les messages, les retarder, les intercepter, les modifier et même en créer. La liste des opérations qu'il peut effectuer est finie : il peut, si il a la clef, déchiffrer un message ou le signer, etc.

Ce type de modèle permet d'automatiser la vérification de certaines propriétés de sécurité, comme la confidentialité ("Le contenu de ce message est-il accessible à l'adversaire ?") et l'authentification ("L'adversaire peut-il se faire passer pour une autre personne ?") pour un nombre en général majoré de sessions. De ce fait, de nombreux outils ont été développés sur ce modèle :

— ProVerif [Bla01] repose sur les clauses de Horn. C'est un transformateur automatique de π-calcul (modélisant des agents distribués) appliqué en clauses de Horn, largement optimisé pour vérifier les protocoles. L'outil utilise des approximations : il est valide (si il déduit que le protocole satisfait une propriété, alors c'est le cas) mais pas complet (il peut donner de fausses attaques, voire ne pas terminer).

— avispa [ABB+05] fournit une interface unique transformant la sécurité d'un protocole en une instance résolue par un autre outil logique.

— Scyther [Cre08] est basé sur un algorithme de raffinement de motifs, ce qui lui permet de pouvoir représenter des traces (possiblement infinies) de manière concise. Après avoir calculé toutes les traces possibles, il les classe et vérifie qu'aucune ne peut contenir d'attaques.

— AKISS [CCK12] raisonne sur une modélisation abstraite des traces d'exécutions de protocole dans des clauses de Horn de premier ordre. Il cherche à montrer des équivalences de traces.

En utilisant ces systèmes, on modélise le protocole pour lancer la vérification. Une attention particulière doit être accordée à cette modélisation, qui impose souvent une certaine distance avec l'implantation du protocole : il est possible de trouver une attaque efficace contre l'implantation d'un système prouvé sûr (par exemple les attaques par mesure de temps et différence de potentiels).

Ceci est dû à ce que par nature, leur utilisation en tant qu'opérateurs abstraits limite leur expressivité, notamment pour modéliser les propriétés de résistance face aux fuites d'information. En effet, l'approche symbolique se retrouve vite limitée si l'on veut modéliser l'accès au support physique qui sert aux calculs cryptographiques par l'adversaire. La

modélisation d'une attaque par consommation de temps ou d'énergie n'est pas naturelle, ainsi que la modélisation d'agent d'observation.

La perte immédiate d'information partielle parait donc difficile à modéliser dans les modèles symboliques. En cela, le modèle calculatoire se distingue du modèle symbolique pour prouver la sécurité d'un système cryptographique.

3.2.2 Modèles calculatoires

Les modèles calculatoires sont plus permissifs pour l'adversaire : d'une part, tous les messages et arguments sont des chaînes de bits. D'autre part, l'adversaire est considéré comme étant probabiliste et ayant des ressources calculatoires (essentiellement du temps et de l'espace mémoire) majorées de façon polynomiale en fonction d'un paramètre de sécurité. En d'autres termes, il doit être efficace. Il peut donc effectuer n'importe quelle opération sur les messages et les arguments, contrairement au modèle symbolique où seul un nombre limité d'actions étaient possibles à l'adversaire.

L'objectif de l'adversaire est, entre autres :

— d'obtenir des informations sur une clef échangée,

— d'obtenir des informations sur un message clair à partir de son chiffré,

— se faire passer pour l'un ou l'autre des participants d'un protocole,

— trouver un mot de passe à partir de son empreinte...

Pour tout adversaire probabiliste, on lui associe une probabilité d'y parvenir. On cherche alors à majorer cette probabilité.

Le fait qu'il soit impossible de violer la confidentialité d'un message chiffré, de compromettre une clef ou d'impersonnifier un participant d'un protocole sont des propriétés de sécurité. Ces dernières sont alors modélisées par des jeux que l'adversaire doit remporter. Une propriété est dite violée lorsque l'adversaire a une probabilité notablement plus forte qu'une réponse faite au hasard de gagner (c'est-à-dire que la différence des deux probabilités est non négligeable).

On gagne alors en description des attaques possibles. Pour le chiffrement, on travaille le plus souvent sur la notion d'indiscernabilité. L'adversaire doit résoudre un défi : deviner à quel texte clair (entre deux possibles) correspond un chiffré donné. L'idée étant que si un adversaire arrive à différencier deux clairs, cela veut dire qu'il a réussi à obtenir une information à partir du chiffré.

Exemple intuitif de jeu d'indiscernabilité

On peut, par exemple, imaginer un jeu d'indiscernabilité où l'adversaire évolue dans deux mondes possibles (avec la même probabilité d'évoluer dans l'un ou l'autre). Imaginons que l'adversaire joue à pile ou face. Dans un des mondes (le monde réel), le challengeur utilise une pièce normale, bien équilibrée, où pile et face on la même probabilité d'arriver, et dans l'autre monde une pièce biaisée où la pièce possède deux côtés faces. L'adversaire doit alors deviner dans quel monde il évolue. Ce jeu est illustré dans la figure 3.2.

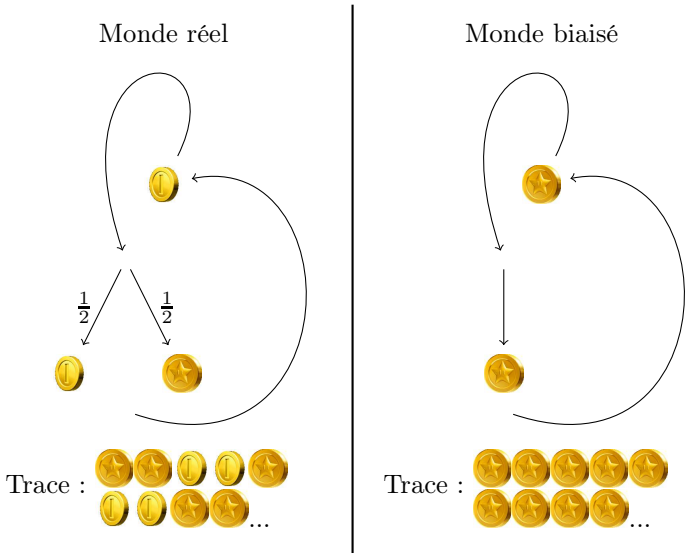


FIGURE 3.2 – Jeu d'indiscernabilité réel ou biaisé

L'expérience se déroule en deux phases :

1. Le challengeur effectue n jets de pièce.
2. L'adversaire doit deviner dans quel monde il évolue.

À la fin de l'expérience, nous obtenons une trace de ce qu'il s'est passé (ici, le résultat de tous les jets de pièces). Si au cours de l'expérience, l'adversaire a vu un côté pile apparaître (autrement dit, si pile appartient à la trace), alors il est sûr d'être dans le monde réel. Sinon, il y a une chance sur 2^n que pile n'apparaisse pas dans la trace du monde réel et toutes les chances d'arriver dans le monde biaisé. Pour répondre au mieux, l'adversaire dira qu'il est dans le monde réel si il y a eu un côté pile qui est apparu au cours des jets, sinon, il répond qu'il est dans le monde biaisé. On peut également imaginer un adversaire qui répondrait au hasard.

L'avantage d'un adversaire (exprimé en général en fonction d'un paramètre de sécurité) peut être défini par la différence de sa probabilité de gagner le défi et de la probabilité de gagner le défi en répondant au hasard (dans le cas de l'indiscernabilité, une chance sur deux, soit $\frac{1}{2}$), ou encore par la différence entre les probabilités de donner une réponse correcte et une réponse fausse en interagissant dans un monde donné. On mesure l'efficacité d'un adversaire grâce à son avantage. Si cet avantage est suffisamment petit, alors la propriété de sécurité est préservée.

Nous calculons d'abord la probabilité d'être dans le monde réel en sachant que le côté pile n'est jamais apparu. Soient F l'évènement où le côté face apparaît sur tous les lancers et R l'évènement où l'adversaire évolue dans le monde réel. Calculons la probabilité de n'avoir que des faces :

$$\Pr[F] = \frac{1}{2^{n+1}} + \frac{1}{2} = \frac{1+2^n}{2^{n+1}}$$

Puis la probabilité d'être dans le monde réel :

$$\begin{aligned}\Pr[R|F] &= \frac{\Pr[F|R]\Pr[R]}{\Pr[F]} \\ &= \frac{1}{2^n} \times \frac{1}{2} \times \frac{2^{n+1}}{1+2^n} \\ &= \frac{2^{n+1}}{2^{n+1}(2^n+1)} \\ &= \frac{1}{2^n+1}\end{aligned}$$

Nous pouvons maintenant calculer l'avantage AVG de l'adversaire $\mathbb{A}$:

$$\begin{aligned}AVG_{\mathbb{A}}^{IND_{rob}}(n) &= \left|\Pr\left[b' \xleftarrow{\$} IND^0(\mathbb{A},n)|b'=0\right] - \Pr\left[b' \xleftarrow{\$} IND^1(\mathbb{A},n)|b'=0\right]\right| \\ &= \left|1 - \frac{1}{2^n+1}\right| \\ &= \frac{2^n}{2^n+1}\end{aligned}$$

, où $\mathbb{A}$ est l'adversaire, b' est la réponse de l'adversaire, 1 représente le monde réel et 0 le monde biaisé, $IND^0(\mathbb{A},n)$ note l'exécution du jeu d'indistingabilité dans le monde 0 (le monde biaisé), $IND^1(\mathbb{A},n)$ note l'exécution du jeu d'indistingabilité dans le monde 1 (le monde réel), et $\xleftarrow{\$}$ dénote un résultat probabiliste. Dans le cas où l'adversaire est dans le monde biaisé, il répond toujours 0 : $\Pr\left[b' \xleftarrow{\$} IND^0(\mathbb{A},n)|b'=0\right] = 1$. Dans le cas où l'adversaire évolue dans le monde réel, la probabilité qu'il se trompe est égale à la probabilité que tous les tirages soient des faces, soit $\Pr\left[b' \xleftarrow{\$} IND^1(\mathbb{A},n)|b'=0\right] = \Pr[R|F]$.

Ici, l'avantage de l'adversaire est énorme : plus le nombre de jets est grand, plus l'adversaire a de chance de deviner correctement.

De la même façon, on définit des jeux d'indiscernabilité pour les propriétés de sécurité cryptographique.

Jeux d'indiscernabilité pour le chiffrement

On distingue trois niveaux d'attaques :

— les attaques de type textes clairs choisis (CPA pour *Chosen Plaintext Attack*) : l'adversaire peut chiffrer les messages qu'il souhaite (c'est le minimum requis pour un chiffrement asymétrique) ;

— les attaques de type textes chiffrés choisis (CCA1 pour *Chosen Cipher Attack*, connues aussi sous le nom de *lunch-time attack*) : l'adversaire peut déchiffrer les messages qu'il souhaite ;

— les attaques adaptatives de type textes chiffrés choisis (CCA2 pour *Chosen Cipher Attack*) : l'adversaire peut déchiffrer les messages qu'il souhaite, même après que le défi lui soit donné, à l'exception du chiffré qui lui est donné à résoudre.

On peut également travailler sur la notion de malléabilité (l'adversaire ne devrait pas être capable de modifier un chiffré en un autre chiffré valide, c'est-à-dire chiffrant un autre message) qui est plus forte que l'indiscernabilité, ou de sens unique d'une fonction (l'adversaire ne devrait pas être capable de retrouver un message clair à partir d'un chiffré, ce qui est une propriété plus faible que l'indiscernabilité).

Définition 3.1 (Fonction négligeable)**.** Une fonction $\mu : \mathbb{N} \to \mathbb{R}^+$ est dite négligeable si, pour tout polynôme P, il existe un $X \in \mathbb{R}^+$ tel que

$$\forall x > X, \mu(x) < \frac{1}{|P(x)|}$$

Un système est sûr dans un jeu de sécurité si pour tout adversaire représenté par une machine de Turing probabiliste en temps polynomial (PPT pour *Probabilistic Polynomial time Turing machine*) son avantage est négligeable.

On définit maintenant formellement le jeu de IND-CCA1. Soit un système de chiffrement $S = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ où $\mathcal{K}$ est l'algorithme de génération des clefs, $\mathcal{E}$ est l'algorithme de chiffrement et $\mathcal{D}$ est l'algorithme de déchiffrement. L'adversaire est une machine de Turing probabiliste polynomiale en temps. Il doit envoyer deux messages au challengeur, puis décider si le chiffré qu'il reçoit au cours du jeu est le gauche ou le droit qu'il avait proposé.

Soit η le paramètre de sécurité. Le jeu se déroule comme dans le jeu 3.1.

$\mathbb{A}^{\mathbb{O}}$ signifie que $\mathbb{A}$ a accès à un oracle $\mathbb{O}$ (c'est-à-dire uniquement question-réponse). On définit ainsi l'avantage lié à un jeu d'indiscernabilité :

Définition 3.2 (Avantage d'indiscernabilité CCA1 d'un adversaire)**.** Soit η le paramètre de sécurité. L'avantage $AVG^{IND_{CCA1}}$ d'un adversaire $\mathbb{A}$ contre le système de chiffrement $S = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ dans un jeu d'indiscernabilité 3.1 est défini par

$$AVG_{S,\mathbb{A}}^{IND_{CCA1}}(\eta) = \Big| \Pr\left[b' \xleftarrow{\$} IND^0_{CCA1}(\mathbb{A}, \eta) : b' = 0\right] - \Pr\left[b' \xleftarrow{\$} IND^1_{CCA1}(\mathbb{A}, \eta) : b' = 0\right] \Big|$$

Définition 3.3 (Chiffrement sûr contre les attaques à chiffrés choisis)**.** Un système de chiffrement $S = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ où $\mathcal{K}$ est l'algorithme de génération des clefs, $\mathcal{E}$ est l'algorithme de chiffrement et $\mathcal{D}$ est l'algorithme de déchiffrement est dit sémantiquement sûr si et seulement si, pour toute PPT $\mathbb{A}$ en temps polynomial en η (le paramètre de sécurité), l'avantage $AVG_{S,\mathbb{A}}^{IND_{CCA1}}(\eta)$ est négligeable dans le jeu IND-CCA1.

Il existe d'autres jeux de sécurité pour les propriétés CPA [2] qui ne donne pas accès à l'oracle de déchiffrement et CCA2 qui donnent accès à l'adversaire à un oracle de déchif-

2. Il est démontré par [GM84] que les notions de CPA-indiscernabilité et de sécurité sémantique sont équivalentes. Par abus de langage, nous parlerons uniquement de sécurité sémantique.

Monde gauche (0)

1. Le challengeur génère la paire de clefs utilisée $\mathcal{K}(\eta) = (sk, pk)$.
2. Le challengeur fournit la clef publique pk à l'adversaire.
3. L'adversaire exécute un nombre polynomial d'opérations, en ayant accès à l'oracle de chiffrement $\mathcal{E}$ et de déchiffrement $\mathcal{D}$.
4. L'adversaire donne ensuite deux messages clairs (m_0, m_1) sur lesquels il souhaite être testé, et de l'information i (de longueur arbitraire) qu'il souhaite utiliser au cours de la seconde phase.
5. Le challengeur calcule le chiffré correspondant au message gauche $c \stackrel{\text{def}}{=} \mathcal{E}_{pk}(\mathbf{m_0})$ et renvoie ensuite (i, c) à l'adversaire.
6. L'adversaire peut effectuer un nombre polynomial d'opérations (avec accès à $\mathcal{E}$) avant de donner sa supposition sur le monde dans lequel il évolue.

Monde droit (1)

1. Le challengeur génère la paire de clefs utilisée $\mathcal{K}(\eta) = (sk, pk)$.
2. Le challengeur fournit la clef publique pk à l'adversaire.
3. L'adversaire exécute un nombre polynomial d'opérations, en ayant accès à l'oracle de chiffrement $\mathcal{E}$ et de déchiffrement $\mathcal{D}$.
4. L'adversaire donne ensuite deux messages clairs (m_0, m_1) sur lesquels il souhaite être testé, et de l'information i (de longueur arbitraire) qu'il souhaite utiliser au cours de la seconde phase.
5. Le challengeur calcule le chiffré correspondant au message droit $c \stackrel{\text{def}}{=} \mathcal{E}_{pk}(\mathbf{m_1})$ et renvoie ensuite (i, c) à l'adversaire.
6. L'adversaire peut effectuer un nombre polynomial d'opérations (avec accès à $\mathcal{E}$) avant de donner sa supposition sur le monde dans lequel il évolue.

Jeu 3.1 – Indiscernabilité CCA1 de chiffrement droit ou gauche

frement (après l'envoi du défi pour CCA2). Dans notre preuve, nous utilisons la propriété IND-CCA1.

Toutefois, le chiffrement n'est pas l'unique primitive cryptographique.

Jeu d'indiscernabilité pour les codes d'authentification

Il existe par exemple les codes d'authentification de messages (appelés MAC pour *Message Authentification Code*), qui authentifient l'intégrité ainsi que la provenance d'un message. Nous devons donc définir un nouveau jeu pour prendre en compte la contrefaçon d'un tel code.

Soit un système MAC $S = (\mathcal{K}, G, V)$ où $\mathcal{K}$ est l'algorithme de génération des clefs, G est l'algorithme de génération de MAC et V est l'algorithme de vérification. L'adversaire $\mathbb{A}$ a un accès oracle au système et doit produire une paire message/code originale (c'est à dire que $\mathbb{A}$ n'a pas demandé le code d'authentification du message produit) correcte vis à vis du système MAC.

Soit η le paramètre de sécurité. Le jeu d'existence de contrefaçon est décrit dans le jeu 3.2.

1. Le challengeur génère la clef utilisée $k = \mathcal{K}(\eta)$.
2. L'adversaire peut opérer un nombre polynomial d'opérations en η, en ayant un accès oracle aux algorithmes de génération G et de vérification V de MAC.
3. L'adversaire doit retourner un couple (x, m) où x est le code d'authentification de m.
4. Le challengeur vérifie que m n'a jamais été l'objet d'une demande de génération de code par $\mathbb{A}$ (auquel cas l'adversaire a perdu).

Jeu 3.2 – Existence de contrefaçon

L'avantage associé à ce jeu est défini ainsi :

$$AVG_{S,\mathbb{A}}^{ex-forge}(\eta) \stackrel{\text{def}}{=} \Pr\left[(x, m) \leftarrow \mathbb{A}^{(G,V)} : V(k, x, m) = \mathsf{true}\right]$$

Définition 3.4 (Système MAC sûr)**.** Un système d'authentification de message $S = (\mathcal{K}, G, V)$ où $\mathcal{K}$ est l'algorithme de génération des clefs, G est l'algorithme de génération de MAC et V est l'algorithme de vérification de MAC est dit sûr contre l'existence de contrefaçons si et seulement si, pour toute PPT $\mathbb{A}$ en temps polynomial en η (le paramètre de sécurité), l'avantage $AVG_{S,\mathbb{A}}^{ex-forge}(\eta)$ est négligeable.

Les outils existant

Certains outils ont déjà été développés pour faciliter les preuves dans le domaine calculatoire. La plupart repose sur les séquences de jeux.

— Cryptoverif [Bla06] qui pour prouver la sécurité d'un protocole génère automatiquement une séquence de jeux, en remplaçant les primitives cryptographiques par leur

définition de sécurité,

— Certicrypt [BGZ09], développé en Coq, qui permet également de certifier une séquence de jeux prouvant la sécurité de systèmes cryptographiques. Cet outil est cependant difficile à utiliser, car les preuves restent manuelles.

— Easycrypt [BGHB11] cherche à combler cette lacune en aidant à la génération les preuves d'indiscernabilité entre les différents jeux en utilisant des solveurs SMT (Satisfaisabilité Modulo Théorie).

Ces méthodes efficaces bridents cependant l'expressivité d'une preuve : ces outils sont loin de prendre en compte toutes les preuves cryptographiques apportées par les cryptographes. En particulier les fuites d'informations (les attaques par canaux cachés, par exemple) ne sont pas modélisées. Cela motive la conception, l'implantation et l'utilisation d'une logique dédiée, qui puisse ensuite être adaptée pour vérifier d'autres types de preuves.

Modèle pour le stockage d'information borné (par l'adversaire)

Jusqu'à présent, les preuves de sécurité considéraient des environments d'exécutions protégés. L'algorithme est connu de tous, mais l'adversaire n'a aucun accès à son exécution. Toutefois, la cryptographie est maintenant embarquée dans des puces (cartes à puce), utilisée sur des réseaux (des *spywares* peuvent infecter des machines honnêtes), etc. Les attaques par canaux cachés comme l'analyse de consommation de courant ou du temps d'exécution, ainsi que l'intrusion sont de réelles menaces, et de nombreuses implantations ont été cassées en les utilisant.

Par exemple RSA a été cassé dans [Koc96] par une analyse du temps d'exécution sur cartes à puce. Malheureusement, ce type d'attaques n'est pas pris en compte par les modèles boîtes noires habituels. S'ensuivit une nouvelle bataille entre cryptanalystes et cryptographes : à chaque nouvelle attaque, des contres-mesures ad hoc étaient conçues. Cette approche est limitée, et on ne peut pas faire confiance à de tels systèmes sur le long terme.

Ces dernières années, les théoriciens ont fait leur première tentative pour capter la résistance aux intrusions dans leur modèles, avec le travail de [IR02], suivi du travail sur la résistance aux fuites [DP08a]. Des modèles pour la résistance à la fois aux intrusions et aux fuites ont été proposés par [Mau92, MR04].

Le modèle à mémoire bornée (noté BSM - *Bounded-Storage Memory model*), introduit par [Mau92], a pour but de représenter théoriquement l'intrusion dans un système : un agent d'observation (en anglais *spyware*) installé dans une machine honnête par exemple.

On part d'un adversaire, Eve, ayant tous les pouvoirs sur le réseau. Elle contrôle tous les messages : elle peut les lire, les retarder, les modifier, les remplacer par ses propres messages.

Le modèle que nous utilisons (dont on peut trouver la définition formelle dans Dziembowski [Dzi06]) autorise Eve à envoyer un agent d'observation (défini formellement comme un circuit) dans une machine honnête (qui l'exécute sur sa mémoire), et à récupérer sa

réponse à la fin des calculs de celui-ci. L'agent en lui-même n'a pas de contrainte de taille ou de capacité de calcul. La seule restriction est la taille de sa réponse. Cette limitation est modélisée également par le fait que Eve a une mémoire bornée. Pour parvenir à leur fin, on autorise Alice et Bob à partager une (très) longue clef secrète K aléatoire, bien plus grande que la taille de la mémoire de Eve. Ainsi, Alice et Bob sont en mesure d'utiliser cette longue clef à chaque session pour obtenir, à chaque session, une entropie maximale pour des clefs privées (par exemple).

Les restrictions de ce modèle s'expliquent par le fait qu'un système d'information est monitoré par un administrateur système et qu'une trop grande fuite d'information vers une destination inhabituelle serait rapidement repérée.

Pour raisonner dans ce modèle, nous nous sommes basés sur la logique CIL.

Le modèle calculatoire dans son ensemble, et la notion d'indiscernabilité en particulier (c.f. def. 3.2.2), sont la base de CIL (Computational Indistinguishability Logic). Celle-ci définit la sécurité d'un système cryptographique comme une indiscernabilité d'un système idéal ayant la même fonction (bien qu'elle puisse également raisonner sur la probabilité qu'un mauvais évènement se produise). Nous expliquons dans le chapitre 5 comment CIL articule ses différentes règles autour de ce modèle.

Chapitre 4

Protocole d'échanges de clef résistant aux intrusions

Sommaire

Le protocole décrit ci-après, décrit dans [Dzi06] est un protocole de génération de clef de session entre deux parties résistant aux intrusions. Il cherche à générer pour deux parties des clefs de session de manière sécurisée (dans le sens de la résistance aux intrusions).

Tout d'abord, nous décrivons le modèle dans lequel le protocole est considéré.

4.1 Modèle BSM pour la génération de clef de session

Le modèle à taille de mémoire bornée a pour but de modéliser un adversaire qui parviendrait à infecter des machines honnêtes et à récupérer une quantité de données de ces machines. Cette quantité d'information est volontairement limitée dans le modèle de

sécurité : il est très probable qu'une fuite d'information importante soit repérée par l'administrateur du réseau infecté.

4.1.1 Description naïve du modèle

Alice et Bob sont attaqués par un adversaire Eve dont la puissance est bornée. Nous examinons un nombre borné de sessions. L'objectif d'Alice et Bob est de mettre en place une clef de session non compromise, et l'objectif d'Eve est de compromettre une clef de session, à savoir :

— Alice et Bob obtiennent et acceptent des clefs de session différentes,

— Alice ou Bob accepte une clef (partiellement) connue [1],

— obtenir des informations sur la clef qu'Alice et Bob ont générée.

Pour ce faire, Eve a le contrôle total du réseau : elle peut retarder, modifier ou même stopper les messages entre les parties, et envoyer ses propres messages. Eve est autorisée à installer un agent d'observation sur chaque machine d'Alice et de Bob en début de session. La taille de l'agent d'observation est sans limite, et l'agent d'observation est considéré comme non interactif avec Eve (ce qui simplifie la modélisation tout en conservant un certain réalisme car Eve peut envoyer son propre algorithme comme agent d'observation et simuler cette interaction). Elle récupère le résultat de cet agent d'observation à la fin de la session. Une session est dite compromise si un agent d'observation est installé sur au moins l'une des machines.

Alice et Bob partagent une clef secrète à long terme K (pour toutes les sessions), et ont des paramètres aléatoires frais (et indépendants) au début de chaque session. À la fin de la session, ils obtiennent tous deux une clef de session κ_i (qui est généralement utilisée pour sécuriser leurs nouveaux échanges, mais ce n'est pas la question ici). Sans restreindre la généralité, nous supposons que Alice démarre la session.

L'agent d'observation peut faire ce qui suit :

— lire toutes les données internes de la machine sur laquelle il est installé (K, les informations précédentes non supprimées...),

— calculer une fonction arbitraire Γ sur ces données. Cette fonction est modélisée comme un circuit booléen, et sa seule restriction (en plus de sa taille polynomiale, comme Eve est polynomialement bornée) est que son résultat est de taille limitée,

— renvoyer le résultat à l'adversaire en fin de session.

Eve gagne la partie si la clef de session qu'elle a réussi à compromettre n'est pas d'une session compromise : on remarquera que pour une session compromise, l'agent d'observation peut facilement renvoyer à Eve la clef de session (ou du moins suffisamment d'informations sur la clef pour pouvoir la distinguer d'une clef aléatoire).

Nous ne considérons que l'exécution séquentielle des sessions, c'est-à-dire qu'il n'y pas deux sessions en cours en même temps. Nous supposons également que le protocole ne se déroule qu'entre deux parties A et B, par souci de simplicité.

1. On entend par partiellement connue qu'Eve a des informations significatives sur cette clef.

4.1.2 Une Description plus formelle du modèle

La description qui suit est tirée de [Dzi06].

Le système de génération de clef de session est un 6-uplet $(A, B, \alpha, \beta, \gamma, \delta, \chi)$, où α, β, γ, δ, χ sont des polynômes ; tels que $\forall n, \gamma(n)\chi(n) \ll \alpha(n)$ et A et B sont des machines de Turing interactives, en prenant en entrée un paramètre de sécurité 1^η et une clef secrète $K \in \{0,1\}^{\alpha(\eta)}$. L'adversaire Eve ($E$) est une machine de Turing Probabiliste en Temps Polynomial (PPT) prenant en entrée 1^η. L'exécution est divisée en sessions $T_1, T_2, \cdots, T_{\chi(\eta)}$.

Une session se déroule ainsi :

1. Les machines A et B reçoivent des paramètres aléatoires choisis uniformément (et indépendamment) $N_A \xleftarrow{\$} \{0,1\}^{\beta(\eta)}$ et $N_B \xleftarrow{\$} \{0,1\}^{\beta(\eta)}$ (respectivement).
2. E choisit si elle veut corrompre ou non la session. Si tel est le cas, elle produit une description d'un circuit booléen C (qui modélise l'agent d'observation) calculant une fonction $\Gamma : \{0,1\}^{\alpha(\eta)} \times \{0,1\}^{\beta(\eta)} \to \{0,1\}^{\gamma(\eta)}$.[2] La taille de C est arbitraire[3]. C est envoyé à A et B[4].
3. Les machines commencent à échanger des messages. E peut intercepter les messages. Elle peut aussi empêcher certains des messages envoyés de parvenir à destination et fabriquer de nouveaux messages. A commence par un message d'initialisation à B.
4. À la fin de la session, les machines génèrent (en privé) une clef convenue $\kappa_i \xleftarrow{\$} \{0,1\}^{\delta(\eta)}$. Si le trafic n'a pas été perturbé par E alors ces deux clefs doivent être identiques. Si la session est corrompue, A renvoie $\Gamma(K, N_A)$ et B renvoie $\Gamma(K, N_B)$ à E.

Si la session T_i est finie quand Eve décide de la compromettre, elle peut avoir accès à une description des états de A et B à la fin de T_i (mais elle ne peut pas le faire si une session est en cours). Pour l'auteur de l'article original, il modélise le fait que κ_i peut être récupéré par E une fois la session terminée.

Soit $\mathcal{T}$ l'ensemble de toutes les sessions non compromises. De toute évidence, E gagne si, pour une séance $T_i \in \mathcal{T}$ les utilisateurs A et B génèrent deux clefs différentes.

Si ce n'est pas le cas, alors à la fin de l'exécution E choisit $T_{\text{Test}} \in \mathcal{T}$ sa session test. Sa tâche sera de distinguer κ_{Test} d'une clef véritablement aléatoire de même longueur. Nous devons exiger qu'au moins une des parties A ou B ait effectivement émis une clef κ_{Test}.

Le jeu de distinction est donné dans le jeu 4.1.

Définition 4.1 (Système de génération de clefs résistant aux intrusions)**.** Soit l'avantage d'un adversaire $\mathbb{A}$ défini par :

$$AVG_{\mathbb{A}}^{IND}(\eta) = \Big|\Pr\left[b' \xleftarrow{\$} IND^0(\mathbb{A}, \eta) : b' = 0\right] - \Pr\left[b' \xleftarrow{\$} IND^1(\mathbb{A}, \eta) : b' = 0\right]\Big|$$

2. Il est clair que nous devons toujours avoir $\gamma(\eta)\chi(\eta) \ll \alpha(\eta)$, sinon Eve pourrait récupérer la totalité de la clef secrète K.

3. Cependant, elle doit être polynomiale au regard du paramètre de sécurité, comme E est borné en temps polynomial

4. A et B ne sont pas conscients d'avoir été corrompus.

Monde réel	**Monde aléatoire**
1. E effectue un nombre de session du protocole borné par $\chi(\eta)$.	1. E effectue un nombre de session du protocole borné par $\chi(\eta)$.
2. E choisit $T_{\text{Test}} \in \mathcal{T}$	2. E choisit $T_{\text{Test}} \in \mathcal{T}$
3. Le challengeur renvoie κ_{Test} à E.	3. Le challengeur renvoie $\kappa \xleftarrow{\$} \{0,1\}^{\delta(\eta)}$ à E.
4. E peut effectuer un nombre polynomial d'opération avant de donner sa supposition sur le monde dans lequel il évolue.	4. E peut effectuer un nombre polynomial d'opération avant de donner sa supposition sur le monde dans lequel il évolue.

Jeu 4.1 – Indiscernabilité de clefs réelle ou aléatoire dans le modèle BSM

sur le jeu d'indiscernabilité 4.1.

On dit qu'un schéma de génération de clef $(A, B, \alpha, \beta, \gamma, \delta, \chi)$ comme décrit ci-dessus est résistant aux intrusions si, pour tout PPT $\mathbb{A}$:

— la probabilité que, dans les sessions $T_i \in \mathcal{T}$ les machines A et B acceptent deux clefs différentes est négligeable (en η),

— l'avantage de $\mathbb{A}$ est négligeable (en η).

4.2 Description du protocole

Le protocole original décrit dans [Dzi06] est en deux phases : d'abord, les deux parties calculent une clef d'authentification commune intermédiaire S' à partir de leur longue mémoire partagée à l'aide d'une fonction d'extension de clef résistante aux intrusions. Puis, en utilisant cette clef avec une fonction MAC, ils échangent la clef de session, en utilisant un schéma de chiffrement asymétrique.

Nous commençons par présenter la définition d'une fonction résistante aux intrusions, puis nous présenterons le protocole.

4.2.1 Fonction résistante aux intrusions

La définition originale est donnée par [DM04]. Nous adaptons leur définition à nos besoins, notamment en définissant la sécurité d'une fonction résistante aux intrusions en termes d'indiscernabilité réel ou aléatoire. Nous utilisons par ailleurs la plupart de leur notations.

Soit η le paramètre de sécurité. Nous supposons que les deux parties honnêtes, soient Alice et Bob, partagent une longue chaîne de bits aléatoires de longueur $\alpha(\eta)$ tirée d'une distribution uniforme sur $\mathcal{R} \stackrel{\text{def}}{=} \{0,1\}^{\alpha(\eta)}$, c'est-à-dire un champ aléatoire $K \in \mathcal{R}$ qui est

— soit accessible temporairement par tous,

— soit envoyé à tous par Alice ou Bob.

On suppose que l'adversaire Eve a une mémoire de taille $\sigma(\eta)$ très inférieure à $\alpha(\eta)$. Eve ne peut donc stocker que des informations partielles sur K, de taille au plus $\sigma(\eta)$, au travers de n'importe quelle fonction $\Gamma : \mathcal{R} \to \{0,1\}^{\gamma(\eta)}$ qu'elle souhaite utiliser, et perd son accès à K une fois qu'elle a appliqué Γ. Comme K est parfaitement aléatoire, Alice et Bob partagent de l'information que Eve n'a pas les moyens de deviner. Pour finir, Eve a un contrôle total du réseau.

Pour le protocole étudié dans cette thèse, une fonction charnière est la fonction d'extension de clef résistante aux intrusions. Soit $\mathcal{Y} \stackrel{\text{def}}{=} \{0,1\}^{\mu(\eta)}$ l'ensemble des clefs que l'on cherche à étendre. Le but d'une telle fonction est de fournir à Alice et Bob une fonction facilement calculable $f : \mathcal{R} \times \mathcal{Y} \to \{0,1\}^{\nu(\eta)}$, avec $\nu(\eta) \gg \mu(\eta)$, telle que connaissant $\Gamma(K)$ et Y Eve n'ait aucune information sur $f(K,Y)$.

Nous supposons que toutes les tailles sont fonctions polynômes d'un paramètre de sécurité η, c'est à dire $\tau(\eta)$ est la longueur de la chaîne de bits partagée K, $\sigma(\eta)$ est la taille de la mémoire de l'adversaire, $\mu(\eta)$ est la taille de la clef à étendre et $\nu(\eta)$ est la taille de la clef obtenue.

Nous définissons la sécurité d'une telle fonction par le jeu d'indiscernabilité 4.2.

L'avantage de l'adversaire est défini ainsi :

$$AVG^{IND}_{f,\mathbb{A}}(\eta) = \left|\Pr\left[b' \stackrel{\$}{\leftarrow} IND^0(\mathbb{A},\eta) : b' = 0\right] - \Pr\left[b' \stackrel{\$}{\leftarrow} IND^1(\mathbb{A},\eta) : b' = 0\right]\right|$$

Dans le monde IND^0, l'adversaire interagit avec la version réelle de la fonction. Dans le monde IND^1, l'adversaire interagit avec la version aléatoire de la fonction.

Définition 4.2 (Sécurité de la fonction f)**.** Soir le paramètre de sécurité η. La fonction f d'expansion de clef est dite sûre pour le jeu 4.2 si quelque soit l'adversaire $\mathbb{A}$, il existe une probabilité ϵ fonction de η telle que

$$AVG^{IND}_{f,\mathbb{A}}(\eta) \leq \epsilon$$

avec ϵ négligeable en η.

Une telle fonction résistante aux intrusions a été présentée par Dziembowski et Maurer dans [DM04].

4.2.2 Protocole

De façon informelle, le protocole prend place en deux temps. Dans un premier temps, Alice et Bob génèrent une (même) clef d'authentification S' pour authentifier dans un second temps l'échange sécurisée d'une clef de session.

Pour générer la clef d'authentification, Alice et Bob utilisent tous deux K, qu'ils divisent en deux blocs R_A et R_B. Ils génèrent chacun un nonce N_A, N_B, qu'ils s'échangent. Ils utilisent alors la fonction d'expansion de clef pour étendre les nonces sur les deux blocs de K : $f(R_A, N_A)$ et $f(R_B, N_B)$. Ils appliquent alors le ou exclusif sur ces résultats, le hachent, et obtiennent la clef d'authentification qu'ils vont utiliser dans la seconde partie du

Monde réel

1. Le challengeur génère une fonction $f : \mathcal{R} \times \mathcal{Y} \to \{0,1\}^n$.
2. L'adversaire peut lancer un nombre de sessions (non concurrentes, c'est-à-dire strictement séquentielles) un nombre polynomial de fois. Une session est définie par les étapes suivantes :
   (a) L'adversaire peut effectuer un nombre polynomial d'opérations.
   (b) L'adversaire peut générer une fonction Γ prenant en paramètre la fonction f et l'envoyer au challengeur. Si l'adversaire n'envoie pas Γ, la session est dite non-corrompue.
   (c) Le cas échéant, le challengeur renvoie le résultat de Γ à l'adversaire.
   (d) L'adversaire envoie une requête à l'oracle de `o_f`.
   (e) Le challengeur renvoie *le résultat de la fonction* f.
   (f) L'adversaire peut effectuer un nombre polynomial d'opérations avant de relancer une session.
3. L'adversaire choisit une session non-corrompue sur laquelle il souhaite être testé.
4. L'adversaire peut effectuer un nombre polynomial d'opération avant de donner sa supposition sur le monde dans lequel il évolue.

Monde idéal

1. Le challengeur génère une fonction $f : \mathcal{R} \times \mathcal{Y} \to \{0,1\}^n$, *mais pour effectuer les calculs de* f *au sein des sessions non-corrompues, le challengeur utilise une fonction dont le résultat est uniformément aléatoire* $f_U : \mathcal{R} \times \mathcal{Y} \to \{0,1\}^n$.
2. L'adversaire peut lancer un nombre de sessions (non concurrentes, c'est-à-dire strictement séquentielles) un nombre polynomial de fois. Une session est définie par les étapes suivantes :
   (a) L'adversaire peut effectuer un nombre polynomial d'opérations.
   (b) L'adversaire peut générer une fonction Γ prenant en paramètre la fonction f et l'envoyer au challengeur. Si l'adversaire n'envoie pas Γ, la session est dite non-corrompue.
   (c) Le cas échéant, le challengeur renvoie le résultat de Γ à l'adversaire.
   (d) L'adversaire envoie une requête à f.
   (e) Si la requête a déjà été faite, il renvoie la réponse donnée précédemment. *Si la session est corrompue, le challengeur utilise la fonction* f. *Sinon, il utilise la fonction* f_U.
   (f) L'adversaire peut effectuer un nombre polynomial d'opérations avant de relancer une session.
3. L'adversaire choisit une session non-corrompue sur laquelle il souhaite être testé.
4. L'adversaire peut effectuer un nombre polynomial d'opération avant de donner sa supposition sur le monde dans lequel il évolue.

Jeu 4.2 – Indiscernabilité de la fonction d'expansion de clef réelle ou aléatoire

Alice	Bob
$K = (R_A, R_B)$	$K = (R_A, R_B)$
$N_A \xleftarrow{\$} \mathcal{N}$, envoie N_A $S \stackrel{\text{def}}{=} f(R_A, N_A) \oplus f(R_B, N_B)$ $S' \stackrel{\text{def}}{=} H(S)$	$N_B \xleftarrow{\$} \mathcal{N}$, envoie N_B $S \stackrel{\text{def}}{=} f(R_A, N_A) \oplus f(R_B, N_B)$ $S' \stackrel{\text{def}}{=} H(S)$
$(pk_A, sk_A) \xleftarrow{\$} \mathcal{K}$, envoie $(pk_A, \text{MAC}_{S'}(A : pk_A))$	
	vérifie MAC, $\kappa_i \xleftarrow{\$} \{0,1\}^k$, envoie $(c \stackrel{\text{def}}{=} \mathcal{E}_{pk_A}(\kappa_i), \text{MAC}_{S'}(B\|\kappa_i))$
vérifie MAC, $\kappa_i \leftarrow \mathcal{D}_{sk_A}(c)$ efface tout sauf K	efface tout sauf K

f est une fonction d'expansion de clef résistante aux intrusions, H est une fonction de hachage.
En cas d'échec d'une de ses vérifications, la partie arrête le protocole et renvoie une erreur.

TABLE 4.1 – Protocole original décrit par [Dzi06]

protocole. Alice est alors en mesure d'envoyer sa clef publique à Bob de façon authentifiée afin qu'il lui envoient la clef de session, dûment chiffrée et authentifiée.

De façon formelle voici le déroulement du protocole :

Pour un paramètre de sécurité fixe η, soit $\mathcal{R} = \{0,1\}^{\alpha(\eta)}$, $\mathcal{N} = \{0,1\}^{\mu(\eta)}$, $K \xleftarrow{\$} \mathcal{R}$, f est une $(\sigma, \alpha, \nu, \mu)$-fonction sûre d'expansion de clef résistante aux intrusions et $H : \{0,1\}^{\nu(\eta)} \to \{0,1\}^{\lambda(\eta)}$ est une fonction de hachage (modélisée comme une fonction oracle aléatoire), MAC utilise une clef de longueur $\lambda(\eta)$, et le schéma de chiffrement asymétrique est un schéma de chiffrement sémantiquement sûr.

Le protocole de Dziembowski est décrit dans le tableau 4.1.

Le protocole procède de cette façon :

1. A tire au hasard N_A dans $\mathcal{N}$ et l'envoie à Bob.
2. B tire au hasard N_B dans $\mathcal{N}$ et l'envoie à Alice.
3. A et B calculent $S \stackrel{\text{def}}{=} f(R_A, N_A) \oplus f(R_B, N_B)$ et la clef d'authentification $S' \stackrel{\text{def}}{=} H(S)$.
4. A génère une paire de clefs publique et privée [5], et envoie la clef publique avec son MAC en utilisant S' (comme clef) et étiqueté avec son identité à B.
5. B vérifie le MAC en utilisant S' (comme clef), si il est correct, alors il génère de façon aléatoire une clef de session $\kappa_i \in \{0,1\}^{\delta(\eta)}$, le chiffre à l'aide de la clef publique de A, l'envoie avec son MAC en utilisant S' (comme clef) (étiqueté encore une fois avec son identité B) à A et renvoie κ_i.
6. A vérifie le MAC, décrypte le message et renvoie κ_i.
7. A et B effacent toutes leurs données internes, à l'exception de K.

Si l'une des vérifications échoue, la partie correspondante abandonne la session.

5. Notez que cette paire de clefs particulière n'est valable que dans cette session.

4.3 Sécurité du protocole

L'idée de la sécurité de ce système est qu'il peut être attaqué à deux endroits : au cours de la première et de la deuxième partie. Si l'adversaire peut briser S' (dans la première partie), alors il peut usurper l'identité d'Alice et/ou de Bob dans la deuxième partie. Le mieux que Eve puisse faire est de lier S_A et S_B ensemble, mais ce lien ne résiste pas au hachage par H. Ainsi soit $S'_A = S'_B$, soit S'_A et S'_B sont indépendants. L'égalité laisse deux options à Eve : corrompre le schéma MAC ou le schéma de chiffrement, ce qui est impossible. Si S'_A et S'_B sont indépendants, alors Eve ne peut pas fausser un MAC sans S'_A ou S'_B.

Comme nous pouvons le voir, la preuve se réduit à la sécurité de f en tant que fonction BSM-sûre, ainsi que celle du schéma MAC et du schéma de chiffrement.

De façon générale, on peut supposer que l'agent d'observation envoyé par Eve récupère seulement des informations sur la longue clef partagée $K = R_A|R_B$ et les élément aléatoires de la session (les deux nonces générés en début de session et la clef privée d'Alice), car ce sont les éléments qui déterminent le reste des actions effectuées lors d'une session.

La sécurité d'un protocole de sécurité repose sur deux choses essentielles :

— les primitives utilisées ont un niveau de sécurité adapté à leurs utilisations,

— le protocole lui-même est conçu de manière à utiliser les primitives correctement, c'est-à-dire sans introduire de failles de sécurité.

La preuve de sécurité que nous avons développée suppose que les primitives utilisées sont elles-mêmes sûres au sens cryptologique du terme. L'objet de cette section est d'expliciter ce que nous entendons par sûr pour chacune de ces primitives. Pour chacune d'entre elles, nous présentons le jeu d'indiscernabilité cryptographique qui correspond à la modélisation en Coq de l'hypothèse de sécurité.

4.4 Hypothèses de sécurité

Les hypothèses de sécurité sont énoncées concernant les primitives de f, de chiffrement et de code d'authentification de messages. Comme la preuve de sécurité est effectuée dans le modèle ROM, il n'y a pas d'hypothèse correspondant à la fonction de hachage.

Les hypothèses de sécurité reposent toutes sur la définition de sécurité des primitives utilisées. De façon générale, nous avons utilisé des hypothèses proches des définitions de sécurité : ces hypothèses sont émises en fonction de la difficulté qu'a un adversaire à distinguer un système de la version idéalisée au cours d'une session test. Cette difficulté s'exprime en fonction d'un paramètre de sécurité, η, qui définit la longueur des clefs, mais qui comprend également le majorant du nombre d'appels à l'oracle autorisés à l'adversaire et de la probabilité de succès de l'adversaire, tout deux considérés comme étant des fonctions du paramètre de sécurité.

Pour modéliser la sécurité d'un schéma cryptographique, nous sommes passés d'une sécurité asymptotique (qui stipule que l'on peut atteindre la sécurité en fonction d'un

paramètre de sécurité suffisamment grand, souvent la longueur de la clef) à une sécurité concrète, qui donne la sécurité atteinte (c'est à dire la probabilité d'une attaque réussie) par le paramètre de sécurité. Cette sécurité concrète nous permet de simplifier la preuve sans perte de généralité en nous approchant un peu plus de la réalité.

En effet, un adversaire n'a intérêt à attaquer un système sécurisé que tant que l'information protégée a de la valeur. Ce que nous cherchons à montrer est que le temps nécessaire à la réussite d'une attaque (représenté ici comme le nombre d'appels aux oracles) est supérieur au temps durant lequel l'information protégée a de la valeur.

Cependant, nous considérons les fonctions de probabilité de réussite comme des fonctions négligeables (du paramètre de sécurité). Elles en ont donc les propriétés. En particulier, leurs combinaisons linéaires restent négligeables. Nous cherchons donc, en définitive, à exprimer la probabilité maximale de la réussite d'une attaque par une combinaison linéaire de la probabilité de réussite d'une attaque contre chacune des primitives utilisées par le protocole.

Le majorant d'appels aux oracles du protocole est fonction du paramètre de sécurité. Les majorants des primitives sont donc calculés en fonction du nombre d'appels du protocole aux primitives : comme le majorant du théorème principal (`bnd`), valable pour le protocole entier, est donné par η, il faut adapter le majorant supérieur de chaque primitive en appliquant au majorant global le nombre d'appels à chaque primitive. Ainsi, la primitive est sûre pour le majorant d'appels (recalculé), alors la sécurité du protocole est assurée.

Nous présentons dans la suite de cette section les différentes hypothèses de sécurité sur lesquelles repose notre preuve. Nous en présentons également la modélisation en Coq, car elle sont compréhensibles dans leur esprit sans connaître CIL et constituent une première approche plus intuitive. Une compréhension plus fine de cette modélisation est donnée dans le chapitre suivant (5) dédié à la présentation et à l'explication de CIL et de sa modélisation en Coq.

4.4.1 Fonction de hachage : modèle de l'oracle aléatoire

Dans notre preuve comme dans beaucoup d'autres preuves du modèle calculatoire, nous nous plaçons dans le modèle de l'oracle aléatoire (ROM). Un oracle aléatoire est une fonction qui à chaque nouvelle requête tire uniformément au hasard une chaîne de bits de la longueur du haché voulue. Par la suite, si une requête est répétée, alors l'oracle renverra la même réponse que précédemment.

Voici la modélisation en Coq que nous avons eue pour cet oracle. La fonction de hachage suit au plus près la définition du modèle d'oracle aléatoire : soit la valeur a déjà été hachée, auquel cas on retrouve le haché dans l'archive (ici `ListH`), sinon on tire une valeur de manière uniforme, que l'on ajoute à l'archive avec la fonction `update_H`.

```
Definition o_H: text * ListH → hash :=
fun m, Lh ⇒
```

```
      if ∃h, (is_in Lh (m, h))
         then (h, Lh)
      else let h := draw_from hash in
           (h, Lh::( m, h))
end.
```

4.4.2 Fonction d'extension de clef

La définition de la sécurité d'une fonction résistante aux intrusions est donnée dans la section 4.2. Elle repose essentiellement sur la probabilité de distinguer la distribution de sortie de la fonction de la distribution uniforme. On raisonne alors sur un jeu de sécurité où l'adversaire doit distinguer la fonction de sa version idéalisée. Cette fonction idéale a été modélisée comme une fonction qui à chaque appel avec des paramètres frais (i.e. n'ayant jamais été utilisés) tire une nouvelle sortie au hasard de la distribution uniforme (de façon analogue à l'oracle aléatoire du ROM). Si les paramètres ont déjà été utilisés, alors on renvoie la même réponse que précédemment.

Nous avons choisi de modéliser la fonction f en la fusionnant avec la longue clef secrète partagée entre les parties honnêtes. Ce choix de modélisation s'explique d'abord par un souhait de simplicité. Cela ne change rien à la validité de la preuve : l'agent d'observation est tout à fait capable de calculer point par point la fonction, et donc de simuler entièrement la véritable clef. De ce fait, la modélisation prends en paramètre les deux nonces des parties honnêtes pour renvoyer directement la clef étendue.

Le système d'oracles est composé :

— d'un oracle de réinitialisation de session `reset−session` qui exécute l'agent d'observation sur l'état du système et si aucune information n'est renvoyée par l'agent d'observation à l'adversaire marque la session comme session test ;

— de l'oracle de f `o_f`, qui prend en paramètre deux nonces et renvoie la clef étendue, cohérent entre ses réponses (à la manière d'un oracle aléatoire simulant une fonction de hachage).

Les deux fonctions prennent en argument d'une part les deux nonces utilisés, et l'état courant du système d'oracles, c'est-à-dire :

— la longue clef secrète `K: nonce*nonce → f_output`,

— la liste des appels effectués `L: ListF`,

— le marqueur indiquant si l'on se trouve dans une session non-corrompue `t: bool`.

Les fonctions retournent alors le résultat de leur action et le nouvel état, à l'issue de l'appel.

La fonction f et sa version uniforme sont modélisées ainsi :

```
Definition o_f: ((nonce*nonce) * ((nonce*nonce → f_output) * ListF * bool))
                → (f_output * ((nonce*nonce → f_output) * ListF * bool)):=
  fun ((na, nb), state) ⇒
    match state with
```

```
      | K, L, t ⇒ (K na nb,
                   (K, (L::( na, nb,K na nb)), t))
    end
end.

Definition o_U: ((nonce*nonce) * ((nonce*nonce → f_output) * ListF * bool))
                → (f_output * ((nonce*nonce → f_output) * ListF * bool)):=
  fun ((na, nb), state) ⇒
    match state with
      | K, L, t ⇒
          if t then
            if  ∃o, (is_in L (na, nb, o)) then
                (o, (K, L, t))
            else
                let o := draw_from f_output in
                  (o, (K, (L::( na, nb, o)), t))
          else (o_f (na, nb, state), (K, L, t))
     end
end.
```

Où `t` est le marqueur d'une session non-corrompue.

Le jeu de sécurité correspondant est décrit dans le jeu 4.2, présenté précédemment. `nonce` est donc de taille μ, et `f_output` de taille ν. (K ayant été fusionnée avec f, α n'est plus pris en considération.) L'agent d'observation n'étant pas encore pris en compte, la taille de retour de l'agent d'observation et de la mémoire de l'adversaire n'est pas discutée ici. Nous le ferons dans la section 6.3, lors de la modélisation du protocole.

Par souci de simplification, lors de la modélisation Coq, l'adversaire choisit les sessions tests en ne récupèrant pas d'information par le biais de l'agent d'observation (c'est-à-dire que la chaîne de bits renvoyée par l'agent d'observation est de taille nulle). Durant ces sessions, l'adversaire doit déterminer si il interagit avec la version idéale (c'est à dire aléatoire) ou la version réelle de la fonction. Au cours des sessions non tests (c'est-à-dire corrompues), l'adversaire interagit systématiquement avec la version réelle de f. Dans CIL, on parle donc d'indiscernabilité entre les deux systèmes, elle est définie dans la section 5.3.2.

En Coq l'hypothèse de sécurité que nous utilisons se note ainsi :

```
Hypothesis indis_f_U: Frame_f
∼((fc_bound_fpi) bnd; eps_f)
Frame_U.
```

où `Frame_f` est le cadre d'exécution de la primitive f (nous aborderons cette notion dans la section 5.3.1 et particulièrement dans la définition 5.6), `Frame_U` est le cadre d'exécution de sa version idéale U, $\sim$ est la notation pour le jugement d'indiscernabilité de deux cadres d'exécution (nous aborderons la définition en CIL de ce jugement dans la section 5.3.2, plus particulièrement dans la définition 5.12) `bnd` est un majorant supérieur d'appels au

protocole, `fc_bound_fpi` est la fonction donnant le nombre d'appels à la primitive f en fonction du nombre d'appels au protocole (ce point est abordé plus en détail dans la section 5.3.3 et particulièrement dans la définition 5.16), et où `eps_f` est un majorant de la probabilité qu'un adversaire a de distinguer la version idéale de la version réelle, variables du paramètre de sécurité.

4.4.3 Fonctions de chiffrement et déchiffrement

La modélisation en Coq s'est portée sur l'indiscernabilité entre la fonction de chiffrement et une fonction idéale, qui ne chiffre pas le message clair, mais une chaîne de bits de même longueur mais composée uniquement de 0. Cette fonction a un comportement idéal dans le sens où aucune information ne peut fuir du chiffré obtenu.

Les chiffrement et déchiffrement reposent sur les fonctions `crypt` et `decrypt` fournies en tant que primitives au système, dont nous ne définissions que la propriété de correction. La propriété de sécurité est donnée par l'hypothèse (donnée plus loin dans cette même section). Le chiffrement se fait à l'aide de la clef fournie. La mise à jour de l'état est faite en rajoutant à la liste des chiffrements faits l'entrée courante. Sans perte de généralité, la fonction de déchiffrement cherche d'abord à savoir si le chiffré est archivé, auquel cas il retourne le clair auquel il est associé, sinon elle déchiffre le message en utilisant la clef privée donnée en paramètre.

C'est un choix cohérent, car il ne change pas le comportement de l'oracle. Les fonctions `crypt` et `decrypt` étant correctes, chaque chiffré appartenant à la liste a forcement été chiffré par l'oracle `o_encr` avec la clef publique correspondant à la clef privée (car c'est le seul oracle à modifier cette liste) et peut donc être déchiffré par `decrypt`.

Ce choix s'explique par un souhait de simplicité dans les démonstrations de bisimilation : en effet, ainsi, les fonctions de déchiffrement pour le système idéalisé et le système originel seront les mêmes.

Les deux fonctions prennent en argument d'une part le message et la clef utilisée, et l'état courant du système d'oracles, c'est-à-dire :

— la liste des appels effectués `L: ListDec`,

— le marqueur indiquant si l'on se trouve dans une session non-corrompue `t: bool`.

Les fonctions retournent alors le résultat de leur action et le nouvel état, à l'issue de l'appel.

La modélisation se fait ainsi pour la version réelle :

```
Definition o_encr : (key*text) * (ListDec*bool)
          → (cipher* (ListDec*bool) ) :=
fun ((pka, mes), (Ld, t))⇒
     let c := crypt pka mes in
       (c,
        (Ld::(( c, pka), mes), t))
end.
```

```
Definition o_decr: (key*cipher) * (ListDec*bool)
          → (session_key* (ListDec*bool) ) :=
fun ((pka, mes), (Ld, t))⇒
    match Ld (c, (pk sk)) with
      | Some v ⇒ (v, (Ld, t))
      | None ⇒ (decrypt sk c, (Ld, t))
    end
end.
```

La fonction de chiffrement est donc modifiée : si c'est une session test, alors on chiffre `nul_string`, sinon on chiffre le message. On enregistre alors le chiffré dans l'état en tant qu'archive , avec la clef et le clair. Lors du déchiffrement, si c'est une session test, alors on retourne le clair associé au chiffré dans la fonction d'archivage. Sinon, on renvoie le déchiffrement.

```
Definition o_encr_id : (key*text) * (ListDec*bool)
          → (cipher * (ListDec*bool)) :=
fun ((pka, mes), (Ld, t))⇒
     let c :=
        if t then crypt pka nul_string
        else crypt pka mes
     in (c, (Ld::(( c, pka), mes)), t)
end.

Definition o_decr_id: (key*cipher) * (ListDec*bool)
          → (session_key * (ListDec*bool)) :=
fun ((pka, mes), Ld, t)⇒
  if t then
        if ∃mes, (is_in (Ld (c, (pk sk), mes))) then
            (mes, Ld, t)
        else (nul_string, Ld, t)
  else ((decrypt sk c), Ld, t)
end.
```

La définition classique de la sécurité d'un système de chiffrement de niveau CCA1 est donné par la définition 3.3. Nous avons basé notre preuve sur un autre jeu de sécurité, que nous appellerons réel ou nul.

Soit η le paramètre de sécurité. Le jeu qui se déroule est décrit par le jeu 4.3.

L'avantage de l'adversaire est défini ainsi :

$$AVG_{S,\mathbb{A}}^{IND_{Ro0}}(\eta) = \left|\Pr\left[b' \xleftarrow{\$} IND_{Ro0}^{0}(\mathbb{A},\eta) : b' = 0\right] - \Pr\left[b' \xleftarrow{\$} IND_{Ro0}^{1}(\mathbb{A},\eta) : b' = 0\right]\right|$$

Définition 4.3 (Sécurité d'un chiffrement CCA1 dans le jeu Réel ou nul)**.** Soit η un paramètre de sécurité. Un système de chiffrement $S = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ où $\mathcal{K}$ est l'algorithme de génération des clefs, $\mathcal{E}$ est l'algorithme de chiffrement et $\mathcal{D}$ est l'algorithme de déchiffrement

Monde nul (0)

1. Le challengeur génère la paire de clefs utilisée $\mathcal{K}(\eta) = (sk, pk)$.
2. Le challengeur fournit la clef publique pk à l'adversaire.
3. L'adversaire exécute un nombre polynomial d'opérations (en η), en ayant accès à l'oracle de chiffrement $\mathcal{E}$.
4. L'adversaire donne ensuite un message clair $m_{\mathbf{1}}$ sur lesquels il souhaite être testé, et de l'information i (de longueur arbitraire) qu'il souhaite utiliser au cours de la seconde phase.
5. *Soit la chaîne de bit* $m_{\mathbf{0}} = 0$. Il calcule le chiffré correspondant $c \stackrel{\text{def}}{=} \mathcal{E}_{pk}(m_{\mathbf{0}})$ et renvoie ensuite (i, c) à l'adversaire.
6. L'adversaire peut effectuer un nombre polynomial d'opération (en η) avant de donner sa supposition sur le monde dans lequel il évolue.

Monde réel (1)

1. Le challengeur génère la paire de clefs utilisée $\mathcal{K}(\eta) = (sk, pk)$.
2. Le challengeur fournit la clef publique pk à l'adversaire.
3. L'adversaire exécute un nombre polynomial d'opérations (en η), en ayant accès à l'oracle de chiffrement $\mathcal{E}$.
4. L'adversaire donne ensuite un message clair m sur lesquels il souhaite être testé, et de l'information i (de longueur arbitraire) qu'il souhaite utiliser au cours de la seconde phase.
5. Le challengeur récupère le texte clair m. Il calcule le chiffré correspondant $c \stackrel{\text{def}}{=} \mathcal{E}_{pk}(m)$ et renvoie ensuite (i, c) à l'adversaire.
6. L'adversaire peut effectuer un nombre polynomial d'opération (en η) avant de donner sa supposition sur le monde dans lequel il évolue.

Jeu 4.3 – Indiscernabilité de chiffrement réél ou nul

est dit sûr dans le jeu Réel ou nul si et seulement si, pour tout adversaire $\mathbb{A}$, l'avantage $AVG_{S,\mathbb{A}}^{IND_{Ro0}}(\eta)$ est négligeable dans le jeu 4.3.

Nous allons maintenant démontrer que les deux jeux de sécurité sont équivalents.

Lemme 4.1. *Un système de chiffrement $S = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ (où $\mathcal{K}$ est l'algorithme de génération des clefs, $\mathcal{E}$ est l'algorithme de chiffrement et $\mathcal{D}$ est l'algorithme de déchiffrement) est sémantiquement sûr si et seulement si il est sûr dans le jeu 4.3.*

Nous démontrons successivement les deux implications. Il est facile de voir qu'un système de chiffrement satisfaisant cette dernière définition satisfait également la définition donnée par le jeu de la section 3.3.

Démonstration. Soit un adversaire $\mathbb{A}_1$ efficace contre le système de chiffrement dans le jeu réel ou nul. À partir de $\mathbb{A}_1$, on construit un adversaire $\mathbb{A}_2$ contre le jeu de sécurité IND-CCA1 défini section 3.3.

Essentiellement, $\mathbb{A}_2$ transfère les messages entre $\mathbb{A}_1$ et le challengeur en remplaçant m_1 par 0, en utilisant son oracle de chiffrement pour simuler celui de $\mathbb{A}_1$.

Ainsi, si $\mathbb{A}_1$ devine correctement le message clair correspondant au défi, alors $\mathbb{A}_2$ devinera également correctement si le chiffré correspond au message clair qu'il a choisi ou non.

L'avantage de $\mathbb{A}_2$ est donc au moins aussi grand que celui de $\mathbb{A}_1$.

La réciproque est légèrement plus ardue.

Soit un adversaire $\mathbb{A}_1$ efficace contre le système de chiffrement selon le jeu IND-CCA1. À partir de $\mathbb{A}_1$, on construit un adversaire $\mathbb{A}_2$ contre le jeu de sécurité réel ou nul.

Pour les premières étapes, $\mathbb{A}_2$ transfère les messages entre $\mathbb{A}_1$ et le challengeur, en utilisant son oracle de chiffrement pour simuler celui de $\mathbb{A}_1$.

$\mathbb{A}_2$ tire un bit b' au sort. Lorsque $\mathbb{A}_2$ reçoit les deux messages (m_0, m_1), il envoie $m_{b'}$ au challengeur, et renvoie le défi du challengeur à $\mathbb{A}_1$.

Soit d la réponse de $\mathbb{A}_1$. Si $b' = d$, alors $\mathbb{A}_2$ renvoie 0, sinon il renvoie 1.

Calculons maintenant l'avantage de $\mathbb{A}_2$. Intuitivement, si $b' = b$, alors $\mathbb{A}_1$ devrait deviner correctement. Dans le cas contraire, alors $\mathbb{A}_1$ a une chance sur deux de deviner correctement.

$$
\begin{aligned}
&AVG^{IND_{ro0}}_{S,\mathbb{A}_2}(\eta) \\
&\quad = \left| \Pr\left[d \xleftarrow{\$} IND^0_{ro0}(\mathbb{A}_2,\eta) : d = 0\right] - \Pr\left[d \xleftarrow{\$} IND^1_{ro0}(\mathbb{A}_2,\eta) : d = 0\right] \right| \\
&\quad = \Big| \Pr\left[d \xleftarrow{\$} IND^0_{ro0}(\mathbb{A}_2,\eta) : d = 0 | b' = 0\right] \Pr\left[b' = 0\right] \\
&\qquad + \Pr\left[d \xleftarrow{\$} IND^0_{ro0}(\mathbb{A}_2,\eta) : d = 0 | b' = 1\right] \Pr\left[b' = 1\right] \\
&\qquad - \Pr\left[d \xleftarrow{\$} IND^1_{ro0}(\mathbb{A}_2,\eta) : d = 0 | b' = 0\right] \Pr\left[b' = 0\right] \\
&\qquad + \Pr\left[d \xleftarrow{\$} IND^1_{ro0}(\mathbb{A}_2,\eta) : d = 0 | b' = 1\right] \Pr\left[b' = 1\right] \Big| \\
&\quad = \Bigg| \frac{\Pr\left[d \xleftarrow{\$} IND^0_{ro0}(\mathbb{A}_2,\eta) : d = 0 | b' = 0\right]}{2} + 0 \\
&\qquad - \frac{\Pr\left[d \xleftarrow{\$} IND^1_{ro0}(\mathbb{A}_2,\eta) : d = 0 | b' = 0\right]}{2} + 0 \Bigg| \\
&\quad = \left| \frac{\Pr\left[d \xleftarrow{\$} IND^0_{CCA1}(\mathbb{A}_1,\eta) : d = 0\right] - \Pr\left[d \xleftarrow{\$} IND^1_{CCA1}(\mathbb{A}_1,\eta) : d = 0\right]}{2} \right| \\
&\quad = \frac{AVG^{IND_{CCA1}}_{S,\mathbb{A}_1}(\eta)}{2}
\end{aligned}
$$

Donc si $\mathbb{A}_1$ est efficace, alors $\mathbb{A}_2$ est également efficace. □

En Coq l'hypothèse de sécurité que nous utilisons se note ainsi :

```
Hypothesis indis_encr_encrid: Frame_encr
~((fc_bound_cpi) bnd); eps_encr)
Frame_encr_id.
```

où `Frame_encr` est le cadre d'exécution de la primitive de chiffrement, `Frame_encr_id` est le cadre d'exécution de sa version idéale, `bnd` est un majorant d'appels au protocole, `fc_bound_cpi` est la fonction donnant le nombre d'appels à la primitive de chiffrement en fonction du nombre d'appels au protocole, et où `eps_encr` est un majorant de la probabilité qu'un adversaire a de distinguer la version idéale de la version réelle du système de chiffrement.

4.4.4 Fonction de MAC

La modélisation en Coq s'est portée sur l'indiscernabilité entre la fonction de MAC et une fonction idéale, qui ne calcule pas le code d'authentification, mais tire une chaîne de bits aléatoire de même longueur et le sauvegarde en mémoire avec le message et la clef utilisée (à la manière d'un oracle aléatoire). Pour vérifier un code, on regarde dans l'archive

si le code est présent, et associé au bon message et à la bonne clef. Cette fonction a un comportement idéal dans le sens où il est impossible de contrefaire un code.

Les deux fonctions prennent en argument d'une part le message et la clef utilisée, et l'état courant du système d'oracles, c'est-à-dire :

— la liste des appels effectués `L: ListMac`,

— le marqueur indiquant si l'on se trouve dans une session non-corrompue `t: bool`.

Les fonctions retournent alors le résultat de leur action et le nouvel état, à l'issue de l'appel.

La modélisation en Coq se fait ainsi :

```
Definition o_gen_mac: (key*text) * (ListMac*bool)
                          → (Mac * (ListMac*bool)) :=
fun (k,mes), (Lm, t)  ⇒
     let mac := MAC_create k mes in
       (mac, ((Lm::( k, mes, mac)), t))
end.

Definition o_ver_mac: (key*Mac*mes) * (ListMac*bool)
                          → bool * (ListMac*bool) :=
fun (k,mac, m), (Lm, t) ⇒
    (orb [6] (MAC_verify s k m mac)
         ((is_in Lm (k, mes, mac)), (Lm, t))
end.
```

Les fonctions de génération et de vérification des codes d'authentification sont modifiées pour atteindre des propriétés idéales. Pour créer un code, si c'est une session test, alors on tire un code au hasard, sinon on génère un code. On sauvegarde cette nouvelle entrée dans l'état. Pour vérifier un code, si c'est une session test, alors on vérifie la présence du code dans l'archive. Sinon, soit le code est dans l'archive, soit il est vérifié par la fonction de vérification.

```
Definition o_gen_mac_id: (key*text) * (ListMac*bool)
                          → (Mac * (ListMac*bool)) :=
fun (k, mes), (Lm, t)  ⇒
        let mac := if t then draw_from Mac
                   else MAC_create k mes
        in (return mac,
            save (Lm::( k, mes, mac)))
end.

Definition o_ver_mac_id: (key*Mac*mes) * (ListMac*bool)
                          → bool * (ListMac*bool) :=
fun (k, mac, m) , (Lm, t) ⇒
     let ret := if t then (is_in Lm (k, m, mac))
```

6. La fonction du 'ou' booléen.

```
14                else orb (is_in Lm (k, m, mac)) (MAC_verify k m mac)
    in return ret
end.
```

Ces fonctions ont été encapsulées pour être appelées directement avec la clef d'Alice ou de Bob, afin de rendre aussi sûre que possible leur utilisation.

Nous donnons maintenant le jeu d'indiscernabilité correspondant 4.4.

Monde idéal

1. Le challengeur génère la clef utilisée $k = \mathcal{K}(\eta)$.
2. *Le challengeur sauvegarde toutes les demandes de l'adversaire et les réponses de l'oracle, et V se base sur cette liste pour vérifier qu'un code d'authentification a bien été généré par l'oracle.*
3. L'adversaire peut opérer un nombre polynomial d'opérations en η, en ayant un accès oracle aux algorithmes de génération G et de vérification V de MAC.
4. L'adversaire doit retourner un couple (x, m) où x est le code d'authentification de m.
5. Le challengeur vérifie que m n'a jamais été l'objet d'une demande de génération de code par $\mathbb{A}$(auquel cas l'adversaire a perdu). Il vérifie si il existe un triplet (k, x, m) dans la liste des requêtes effectuées (ce qui est impossible).

Monde réel

1. Le challengeur génère la clef utilisée $k = \mathcal{K}(\eta)$.
2. L'adversaire peut opérer un nombre polynomial d'opérations en η, en ayant un accès oracle aux algorithmes de génération G et de vérification V de MAC.
3. L'adversaire doit retourner un couple (x, m) où x est le code d'authentification de m.
4. Le challengeur vérifie que m n'a jamais été l'objet d'une demande de génération de code par $\mathbb{A}$ (auquel cas l'adversaire a perdu). Il vérifie que $V(k, x, m) = \mathsf{true}$.

Jeu 4.4 – Indiscernabilité d'authentification

Il est clair que l'avantage associé à ce jeu d'indiscernabilité est le même que celui donné dans la section 3.2.2, puisque la seule façon de différencier le système de MAC par rapport au système idéalisé est de contrefaire un MAC.

$$AVG^{ex-forge}_{S,\mathbb{A}}(\eta) \stackrel{\text{def}}{=} \Pr\left[(x, m) \leftarrow \mathbb{A}^{(G,V)} : V(k, x, m) = \mathsf{true}\right]$$

En Coq l'hypothèse de sécurité que nous utilisons se note ainsi :

```
Hypothesis indis_mac_macid: Frame_mac_id
2 ~((fc_bound_mpi bnd); eps_mac)
Frame_mac.
```

où `Frame_mac` est le cadre d'exécution de la primitive de MAC, `Frame_mac_id` est le cadre d'exécution de sa version idéale, `bnd` est un majorant d'appels au protocole, `fc_bound_mpi` est la fonction donnant le nombre d'appels à la primitive de MAC en fonction du nombre d'appels au protocole,et où `eps_mac` est un majorant de la probabilité qu'un adversaire a de distinguer la version idéale de la version réelle du système de chiffrement.

4.5 Théorème

Le théorème repose essentiellement sur le jeu d'indiscernabilité entre le protocole et sa version idéale, où les primitives employées sont remplacées par leur versions idéalisées. Ainsi, comme le protocole idéal est sûr, si le protocole originel est indiscernable du protocole idéalisé, il est lui-même sûr.

Le jeu 4.5 décrit l'indiscernabilité du protocole contre sa version idéale.

L'avantage de l'adversaire est la probabilité (moins $\frac{1}{2}$) que l'adversaire a de deviner correctement dans quel monde il évolue. La sécurité du protocole est démontrée si l'avantage est négligeable.

Nos travaux démontrent que cet avantage est simplement la somme des probabilités de réussite de l'adversaire au cours des jeux d'indiscernabilité des hypothèses de la section 4.4. Si ces probabilités sont négligeables, alors l'avantage de l'adversaire est également négligeable et le protocole est sûr.

L'énoncé du théorème est simple : le cadre d'exécution du protocole de Dziembowski doit être indiscernable à `eps` près du cadre d'exécution de la version idéalisée du protocole, dans la limite `bnd` du nombre d'appels aux oracles.

```
Definition eps := Uplus eps_f (Uplus eps_mac eps_encr).

Theorem dziembowski_secure :
Pi.Frotocol (*pi*)
~(bnd; eps)
Pi_id.Frotocol. (*pi_id*)
```

où `Pi.Frotocol` est le cadre d'exécution du protocole, `Pi_id.Frotocol` est le cadre d'exécution de sa version idéale, `bnd` est un majorant d'appels au protocole et `eps` est la somme des majorants de succès de l'adversaire définies dans les paramètres de sécurité.

Nous montrons dans le chapitre 6 la modélisation de la preuve de ce théorème.

Maintenant que les objectifs de la preuve sont définis, nous allons introduire la modélisation de la logique de CIL qui nous permettra de mener à bien cette preuve.

Monde réel

1. Le challengeur génère aléatoirement la longue clef commune K d'Alice et Bob.
2. L'adversaire peut lancer un nombre de sessions (non concurrentes, c'est-à-dire strictement séquentielles) un nombre polynomial de fois. Une session est définie par les étapes suivantes :
   (a) L'adversaire peut effectuer un nombre polynomial d'opérations.
   (b) L'adversaire peut générer une fonction v prenant en paramètre la fonction f et les éléments aléatoires de la session (nonces et clefs privées) et l'envoyer au challengeur. Si l'adversaire n'envoie pas v, on dit que la session est non-corrompue.
   (c) Le cas échéant, le challengeur renvoie le résultat de v à l'adversaire.
   (d) L'adversaire interagit avec les oracles, dans l'ordre qu'il souhaite (limité par la cohérence dans l'ordre d'appel des oracles d'Alice et de Bob).
   (e) L'adversaire peut effectuer un nombre polynomial d'opérations avant de relancer une session.
3. L'adversaire choisit une session non-corrompue sur laquelle il souhaite être testé.
4. L'adversaire peut effectuer un nombre polynomial d'opération avant de donner sa supposition sur le monde dans lequel il évolue.

Monde idéal

1. Le challengeur génère aléatoirement la longue clef commune K d'Alice et Bob.
2. L'adversaire peut lancer un nombre de sessions (non concurrentes, c'est-à-dire strictement séquentielles) un nombre polynomial de fois. Une session est définie par les étapes suivantes :
   (a) L'adversaire peut effectuer un nombre polynomial d'opérations.
   (b) L'adversaire peut générer une fonction v prenant en paramètre la fonction f et les éléments aléatoires de la session (nonces et clefs privées) et l'envoyer au challengeur. Si l'adversaire n'envoie pas v, on dit que la session est non-corrompue.
   (c) Le cas échéant, le challengeur renvoie le résultat de v à l'adversaire.
   (d) L'adversaire interagit avec les oracles, dans l'ordre qu'il souhaite (limité par la cohérence dans l'ordre d'appel des oracles d'Alice et de Bob). *Si la session est non-corrompue, le challengeur utilisera les fonctions idéales des oracles. Sinon, il utilisera les fonctions réelles.*
   (e) L'adversaire peut effectuer un nombre polynomial d'opérations avant de relancer une session.
3. L'adversaire choisit une session non-corrompue sur laquelle il souhaite être testé.
4. L'adversaire peut effectuer un nombre polynomial d'opération avant de donner sa supposition sur le monde dans lequel il évolue.

Jeu 4.5 – Indiscernabilité entre les versions idéale et réelle du protocole

Chapitre 5

La Logique CIL en Coq

Sommaire

La difficulté principale des preuves cryptographiques est qu'elles sont faites à la main, qu'elles sont souvent difficiles à vérifier, et qu'on leur fait confiance jusqu'à ce que quelqu'un prouve qu'elles sont fausses, comme cela arrive quelquefois. En réponse à ces remarques, Halevi conseille dans [Hal05] de concevoir des outils partiellement automatisés de vérification pour les preuves de sécurité. Des logiciels comme CryptoVerif [Bla06] et CertiCrypt [BGZ09] remplissent partiellement les suggestions d'Halevi, en fournissant un langage de modélisation rigoureux pour décrire des systèmes cryptographiques et énoncer leur propriétés, et des outils aidant à vérifier la correction des preuves.

Ces approches, souvent généralistes, ont vérifié avec succès des systèmes emblématiques mais peinent à prendre en compte de nouveaux besoins en matière de preuve cryptographique, comme par exemple l'existence d'attaques par canaux cachés. Le développement de la logique d'indiscernabilité calculatoire (CIL) se base sur Coq pour pouvoir répondre

à ce besoin de flexibilité. Ainsi, cet outil robuste nous permet de vérifier une preuve, tout en en automatisant les parties les plus triviales et en en gardant l'esprit.

La formalisation de CIL en Coq est faite de différentes bibliothèques : on utilise d'ALEA (par C. Paulin et. al., [APM09] qui formalise les distributions de probabilités pour raisonner sur les programmes probabilistes) pour bénéficier des propriétés déjà prouvées en Coq. La seconde bibliothèque intègre la modélisation du modèle d'oracles et des adversaires, ainsi que la modélisation et la preuve de la validité en Coq des règles de CIL.

Les définitions et résultats présentés dans les sections 5.1 et 5.2 sont dues à C. Paulin et al. ([APM09]). Les définitions et résultats présentés dans les sections 5.3 sont dues à Pierre Corbineau.

5.1 Les Probabilités et distributions en Coq par ALEA

Pour représenter les probabilités, Paulin et. al. ont défini axiomatiquement l'ensemble $[0, 1]$. Ce nouvel ensemble, U, repose essentiellement sur la notion d'ordre partiel complet et d'un opérateur sur les bornes (en anglais *least uper bound*, `lub`). On utilise donc les constantes 0 et 1.

5.1.1 La Théorie

Les probabilités sont représentées en Coq par un ensemble d'axiomes définissant l'ensemble $[0, 1]$ représenté par un type `U`. Ces axiomes définissent le comportement de l'ensemble des opérations et constructions dont on dispose :

— l'addition (bornée par 1) `Uplus`,

— la multiplication `Umult`,

— la division `Udiv`,

— l'inversion `Uinv` ($x \mapsto 1 - x$)

— on définit 1 (`U1`) comme étant l'inverse de 0,

— le constructeur de `Unth` $n \mapsto \frac{1}{n+1}$.

On démontre également les propriétés usuelles sur ces opérations (commutativité, symétrie, associativité, distributivité, monotonie...), le fait que la relation inférieur ou égal est totale, et la propriété que l'ensemble est archimédien.

On définit également grâce à ces opérateurs les opérations usuelles comme la soustraction, l'opération `max`, la borne minimale. On définit $n \times x$ et x^n par induction sur n. On définit également l'opération & : $x\&y \stackrel{\text{def}}{=} 1 - ((1 - x) + (1 - y))$ qui permet d'exprimer également l'intersection de probabilité.

Maintenant que les propriétés des probabilités sont définies ainsi que celles de leurs opérations usuelles, l'on peut associer à un élément d'un type une mesure μ correspondant à la probabilité de l'élément : on obtient ainsi une distribution de probabilité.

Pour représenter des variables probabilistes, on utilise des monades, qui permettent de généraliser leur manipulation en s'abstrayant du type de la variable. Soit une variable de

type (β). Une probabilité pour cette variable est définie par une fonction $f : \beta \to [0,1]$ (pour chaque élément $x \in \beta$ sa probabilité est $f(x)$), une distribution sur β est définie par une mesure $\mu : (\beta \to [0,1]) \to [0,1]$.

5.2 Une Histoire de monades

Les monades sont particulièrement utilisées dans les langages de programmation fonctionels pour définir des comportements impératifs (générant par exemple des effets de bords) comme les entrées/sorties et les exceptions. Cela permet d'éviter les effets de bord et de rester dans un cadre purement fonctionnel. Moggi donne une motivation et un contexte d'application ainsi qu'un cadre formel à l'utilisation des monades dans les langages de programmation [Mog90]. Elle permettent également de s'abstraire des types sur lesquels on raisonne.

Par exemple, `option` est un type de Coq. Nous allons l'utiliser ici pour illustrer ce qu'est une monade. Pour rappel, une monade est un triplet $(M, unit, star)$ (pour `option` en Coq : `option`, `Some`, `option_map`).

— M associe au type t le type $M(t)$ (`option` associe à `ascii` le type `option ascii`).

— L'opérateur *unit* associe à un élément de t un élément correspondant de $M(t)$ (`Some('g')` est donc de type `option ascii`).

— L'opérateur *star* permet d'associer à un type monadique une fonction d'association sur un autre type monadique : il permet d'appliquer une fonction usuelle sur un type monadique (`option_map (ascii nat) nat_of_ascii`).

On représente ici les distributions par une monade sur β (qui est l'ensemble sur lequel on veut représenter une distribution). Nous pouvons ainsi raisonner sur des distributions en général plutôt que sur la distribution sur un type en particulier. Une distribution étant monotone, on requiert cette propriété dans sa définition. Soit deux types munis d'ordre `Oa` et `Ob`, on note `Oa -m> Ob` le type d'une fonction monotone de `Oa` vers `Ob`, tandis que `mon` autorise la construction de fonctions monotones.

```
Definition MF (A:Type) : Type := A → U. (*evenement*)
Definition M (A:Type) := MF A -m> U.    (*distribution*)

Definition unit (A:Type) (x:A) : M A := mon (fun (f:MF A) ⇒ f x).

Definition star : forall (A B:Type), M A → (A → M B) → M B.
```

`MF` représente donc le type des fonctions renvoyant une probabilité, `M` représente la distribution d'une telle probabilité (c'est donc une fonction monotone).

Les distributions sont représentées par une monade : `unit`, représentant la mesure Dirac δ_x au point x, et `star` sont les opérateurs monadiques habituels devant respecter les propriétés usuelles.

5.2.1 Les Distributions

Une distribution est donc cette fonction de mesure μ, associée aux preuves de stabilité : la linéarité, la compatibilité avec les inverses et la continuité. μ est également une fonction monotone.

```
Record distr (A:Type) : Type :=
  {mu : M A;
   mu_stable_inv : stable_inv mu;
   mu_stable_plus : stable_plus mu;
   mu_stable_mult : stable_mult mu;
   mu_continuous : continuous mu}.
```

La probabilité est donc une fonction de type $f : (\beta \to [0,1]) \to [0,1]$, le type $M\beta$, encapsulé dans `distr` avec ses preuves de stabilité. De ce fait, nous avons tous les outils en main pour raisonner sur les distributions.

Par exemple, la distribution `flip` est la distribution uniforme sur les booléens. On définit la fonction `flip` comme fonction de f : c'est la somme pondérée de f appliquée aux deux valeurs booléennes. On peut alors composer la distribution sur les booléens avec une probabilité (f) afin d'obtenir une distribution finale (d'un résultat de calcul par exemple).

```
Definition flip : M bool :=
  mon (fun (f : bool → U) ⇒ [1/2] ∗ (f true) + [1/2] ∗ (f false)).
```

`flip` permet de tirer une valeur aléatoirement entre true et false avec une propriété de $\frac{1}{2}$ chacune.

5.2.2 Les Fonctions usuelles

Le but ultime est de raisonner sur les programmes probabilistes. Pour ce faire, ils sont représentés sous la forme de calculs abstraits : un calcul abstrait est un nombre d'étapes fini pour calculer un résultat. On utilise les monades pour représenter les différentes étapes du calcul. On définit donc quelques fonctions de base.

```
Definition Mlet : forall A B: Type, distr A → (A → distr B) → distr B.
intros A B mA mB.
exists (star (mu mA) (fun x ⇒ (mu (mB x)))).
apply star_stable_inv; auto.
apply star_stable_plus; auto.
apply star_stable_mult; auto.
apply star_continuous; auto.
Defined.
```

`Mlet` est l'outil principal dont on se sert dans les programmes probabilistes pour affecter à des variables des valeurs probabilistes. On montre que la fonction `star` satisfait la définition d'une distribution. Les valeurs déterministes ne posent pas de problèmes parti-

culiers, on se servira de la fonction Dirac correspondante pour construire la distribution correspondante, à l'aide de l'autre opérateur essentiel, `Munit` (dénoté par !).

Les propriétés essentielles pour notre usage sont

— celle de totalité, c'est à dire qu'une suite de calculs probabilistes aboutit à une distribution de 1, ce qui signifie que la terminaison est presque sûre,

— celle de la compatibilité avec l'égalité, ce qui permet de mener à bien les preuves concernant les valeurs de retours de ces systèmes.

```
Lemma Munit_eq_compat : forall A (x y : A), x = y → Munit x == Munit y.
```

ALEA définit aussi des opérateurs usuels tels que `if.. then .. else .. , let .. in .. ` , et `random` sur les entiers (qui fonctionne sur le même principe que `flip`).

5.3 Modélisation de CIL

La logique calculatoire d'indiscernabilité (CIL) présentée dans [BDKL10] a été conçue spécialement pour prouver des propriétés cryptographiques, tout en étant suffisamment générale pour pouvoir s'adapter à la spécificité des preuves cryptographiques (pour de nouveaux types de propriété) et pouvoir grâce à cela produire des preuves plus intuitives aux yeux des cryptographes.

Pierre Corbineau a modélisé en Coq cette logique prouvant ainsi la validité de ses règles. De plus, en utilisant cet assistant de preuve, nous pouvons obtenir des preuves (très) sûres d'une propriété (voir [CH88, CP90, Luo90]), et dans ce cas la preuve repose sur la cohérence de CIL.

Nous présentons ici cette logique et présentons comment nous l'utilisons pour modéliser la possibilité de fuite d'information.

Nous présentons dans cette section les bases de CIL (implémentée par Pierre Corbineau).

5.3.1 Définitions fondamentales

Dans cette section, nous expliquons comment CIL s'appuie sur des définitions fondamentales pour modéliser l'interaction entre un système cryptographique et un adversaire.

CIL est basée sur la notion d'adversaire et de modèle oracle. L'adversaire n'est pas décrit, et nous ne faisons aucune autre hypothèse sur lui que la quantité de ressources qu'il a le droit d'utiliser : ses calculs doivent finir dans un temps et un espace mémoire déterminés par des fonctions polynomiales du paramètre de sécurité. En d'autres termes, il doit être raisonnablement efficace. Il peut être implémenté de façon arbitraire, utiliser n'importe quel algorithme. Il a droit à un accès oracle au système : il peut faire n'importe quelle requête à n'importe quel oracle du système. Le système d'oracles est composé d'une mémoire commune et d'oracles. Quand un oracle est appelé, il lance sa routine sur la mémoire, met à jour cette mémoire et renvoie sa réponse.

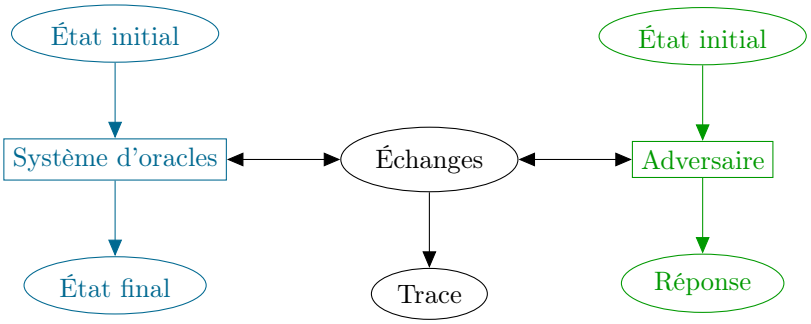


FIGURE 5.1 – Interaction entre un adversaire et un système d'oracles.

Oracle		Adversaire		Effet
Opérateur	Notation	Opérateur	Notation	
`Mlet`	`mlet x:=d in expr`	`Clet`	`clet x:=d in expr`	Assigner une distribution `d` à une variable `x` dans une expression `expr`
`Munit`	`!a`	`Cunit`	`!! a`	Processus renvoyant toujours la valeur `a`
		`Cdraw`	`??d`	Effectuer un tirage dans une distribution
`oracle_call`		`Ccall`		Appeler un oracle

TABLE 5.1 – Combinateurs de la monade `distr` chez l'adversaire

L'intérêt d'utiliser des systèmes d'oracles pour modéliser des schémas de cryptologie est la plasticité de ce modèle : on peut ainsi modéliser les primitives cryptographiques (comme les signatures et le chiffrement) ainsi que les protocoles (en utilisant par exemple un oracle par action).

La modélisation de CIL repose essentiellement sur les échanges entre le système d'oracles et l'adversaire. Le système d'oracles a un oracle pour chaque action du schéma cryptographique qu'il modélise. Cette action peut être pour un protocole une réponse de l'un des acteurs (honnête), ou une action de la primitive (par exemple chiffrer une entrée pour une primitive de chiffrement/déchiffrement). L'adversaire effectue des requêtes au système pour déclencher ces actions.

À la fin de l'interaction, l'adversaire doit :

— soit donner son avis sur le monde dans lequel il a évolué si c'est un jeu d'indiscernabilité,

— soit avoir produit un évènement néfaste pour la sécurité du schéma si c'est un jeu d'existence d'attaque.

Les propriétés s'énoncent sur la trace de ces échanges et la réponse de l'adversaire. Une interaction entre un adversaire et un système d'oracles est représenté dans la figure 5.1.

Nous résumons dans la table 5.1 les opérateurs sur les distributions que nous utilisons pour définir les systèmes d'oracles.

Les définitions suivantes sont une généralisation de [BDKL10]. On commence par définir le système d'oracles, puis l'adversaire et enfin les échanges.

Le Système d'oracles

Le système d'oracles peut être vu comme un ensemble de services interactifs utilisant et modifiant une mémoire commune.

Pour vérifier la sécurité d'un système, on pourra soit étudier les évènements qui rompent la sécurité d'une propriété, soit comparer le schéma à un autre pour lequel on est sûr que la propriété de sécurité est respectée (ou qu'elle ne l'est pas). Il faut donc une notion disant si deux systèmes sont comparables. Deux systèmes d'oracles sont dits compatibles si un adversaire peut agir avec chacun d'eux de la même façon. (On veut généralement montrer qu'un adversaire est capable de distinguer deux systèmes d'oracles compatibles en interagissant avec eux.)

On définit un système d'oracles en deux temps :

1. par leur signature, qui est une interface à usage de l'adversaire ;
2. par leur implémentation, qui comprend la mémoire du système et le fonctionnement des services.

Signature d'un oracle On définit la signature d'un système d'oracles par un ensemble de noms associés à des signatures pour chaque service (appelé un oracle). Ces signatures sont définies pour tous les oracles du système. En outre, on requiert que deux oracles différents du système ne partagent pas un même nom.

On définit donc d'abord l'ensemble des noms d'oracles `oracle_name` avec la preuve de décidabilité des noms d'oracles, c'est-à-dire que l'on requiert que les oracles aient des noms distincts, puis deux fonctions de typage `oracle_input` et `oracle_output` qui associe à un nom d'oracle respectivement le type du paramètre et le type de la réponse de l'oracle. On définit également (par souci de simplicité) des fonctions permettant d'accéder aux types des fonctions des oracles.

Définition 5.1 (Signature d'un système d'oracles)**.** La signature d'un système d'oracles $\mathbb{O}$ est défini par :

1. un ensemble N_o de noms d'oracles ;
2. pour chaque $o \in \mathsf{N}_o$, une signature avec un type d'entrée $\mathsf{In}(o)$ et un type de réponses $\mathsf{Out}(o)$, c'est-à-dire $o : \mathsf{In}(o) \to \mathcal{D}(\mathsf{Out}(o))$

Deux systèmes d'oracles sont dits *compatibles* si ils ont la même signature.

On remarquera qu'un oracle est probabiliste : il renvoie une distribution sur le type de retour.

Ce qui devient en Coq (fichier `Adversary.v`) :

```
Record oracle_signature :=
 mkOS {
```

```
  oracle_name : Type;
  oracle_name_dec:
      forall o1 o2: oracle_name, {o1=o2}+{o1<>o2};
  oracle_input: oracle_name → Type;
  oracle_output: oracle_name → Type }.

Variable os : oracle_signature.
Variable State:Type.

Definition oracle_fun (input:Type) (output:Type) :=
  input * State → distr (output * State).

Definition oracle_functions :=
  forall who, oracle_fun (oracle_input os who) (oracle_output os who).
```

Implémentation d'un oracle L'implémentation de l'oracle peut être vue comme des assignations de distributions successives à partir d'une distribution initiale. La distribution initiale correspond au type d'entrée de la signature de l'oracle, tandis que la distribution finale correspond au type de retour.

Définition 5.2 (Implémentation d'un système d'oracles)**.** Une implémentation d'un système d'oracles $\mathbb{O}$ est constitué :

- d'un état M_o représentant la mémoire du système,
- d'une implémentation qui termine pour chaque oracle o du système, c'est à dire une suite de calculs et d'assignations de distributions satisfaisants la signature de o en incluant la mémoire, c'est-à-dire $o : \mathsf{In}(o) \times \mathsf{M}_o \to \mathcal{D}(\mathsf{Out}(o) \times \mathsf{M}_o)$,
- d'une distribution initiale sur l'état du système $\overline{\mathsf{M}_o}$.

On retrouvera dans (le fichier `Frame.v`) Coq :

```
Record oracle_implementation :=
  mkOI {
    o_funs :> oracle_functions os State;
    o_init : distr State}.
```

Les Échanges La définition suivante présente la description d'une interaction entre l'adversaire et le système d'oracles.

L'adversaire calcule une requête et le nom du prochain oracle qu'il souhaite interroger, met à jour sa mémoire et attend la réponse de l'oracle. Lorsqu'il reçoit celle-ci, il met à jour sa mémoire en conséquence.

Par abus de langage, quand l'oracle concerné o est sans ambiguïté, on notera par In(resp. Out) le type d'entrée $\mathsf{In}(o)$ (resp. de sortie $\mathsf{Out}(o)$) de l'oracle, et de la même façon on désignera son exécution par son nom.

Définition 5.3 (Echange)**.** Un échange est un quadruplet (s, o, q, a) où $s \in \mathsf{M}_o$, $o \in \mathsf{N}_o$, $q \in \mathsf{In}$ et $a \in \mathsf{Out}$. L'ensemble des échanges est noté Xch. Il dénote une requête q à l'oracle o, sa réponse a avec le système d'oracles dans un état s (au moment de l'appel). L'ensemble des requêtes est noté $\mathsf{Que} \stackrel{\text{def}}{=} \{(o, q) | (s, o, q, a) \in \mathsf{Xch}\}$, l'ensemble des réponses est noté $\mathsf{Ans} \stackrel{\text{def}}{=} \{(o, a) | (s, o, q, a) \in \mathsf{Xch}\}$.

Définition 5.4 (Trace d'exécution)**.** Soit $\mathbb{O}$ un système d'oracles et $\mathbb{A}$ un adversaire.

Une trace partielle est une séquence τ de la forme

$$m_0 \xrightarrow{x_1} m_1 \xrightarrow{x_2} \ldots \xrightarrow{x_k} m_k$$

où $m_0 \ldots m_k \in \mathsf{M}_o$ et $x_1 \ldots x_k \in (\mathsf{N}_o \times \mathsf{Que} \times \mathsf{Ans})$ tels que

$$\Pr[\mathsf{O}_{o_i}(q_i, m_{i-1}) = (a_i, m_i)] > 0$$

pour $i = 1 \ldots k$ and $x_i = (o_i, q_i, a_i)$.

Une trace est une trace partielle τ telle que $m_0 = \overline{\mathsf{M}_o}$ et $x_k = (\_, o_{\mathrm{F}}, \_, \_)$ au cours d'une interaction de $\mathbb{A} \mid \mathbb{O}$.

On définit la trace de l'exécution comme étant la liste des échanges entre l'adversaire et le système d'oracles.

```
Record Exchange : Type := mkTR {
  tr_state : State;
  tr_name : oracle_name os;
  tr_input : oracle_input os tr_name;
  tr_output : oracle_output os tr_name}.

Definition Trace :Type := list Exchange.
```

Cette notion d'échange est de trace est utilisée dans les définitions des oracles et des adversaires.

Appel d'un oracle L'appel d'un oracle o d'un système $\mathbb{O}$ s'effectue de manière probabiliste sur M_o (c'est-à-dire que le résultat est une distribution sur la mémoire du système et la réponse de l'oracle). On ajoute ensuite le nouvel état et le résultat à la trace des échanges.

Définition 5.5 (Appel d'oracle)**.** Un appel d'oracle est défini par une fonction sur un triplet contenant :

— le nom de l'oracle appelé : o,

— la valeur de l'argument : i,

— l'état courant : M_o

qui exécute l'implémentation de l'oracle o sur M_o avec pour paramètre i et renvoyant une distribution sur le type de réponse $\mathsf{Out}(o)$ et sur la mémoire M_o après modification par l'oracle.

```
Record Value (Tr A: Type) := mkV
  { v_obj : A;
    v_state : State;
    v_trace : Tr}.

Definition oracle_call (OF : oracle_functions)
  (name:oracle_name osign) (inp:Value Trace (oracle_input osign name))
      : distr (Value Trace (oracle_output osign name)) :=
 mlet res := OF name (v_obj inp,v_state inp) in
!mkV (fst res) (snd res) (trace_push (v_state inp) name (v_obj inp) (fst
res)  (v_trace inp)).
```

Une fois l'implémentation du système faite, on définit un cadre d'exécution réunissant l'implémentation ainsi que la preuve de totalité du système.

Définition 5.6 (Cadre d'exécution)**.** Le cadre d'exécution d'un système d'oracles est constitué des éléments suivants :

- une implémentation (se terminant),
- la preuve de terminaison de chacune des implémentations des différents oracles du système,
- la preuve que la probabilité de la mémoire initiale du système est 1.

Le système qui en résulte doit finir : chaque exécution (avec n'importe quel adversaire) doit terminer avec une probabilité 1, c'est à dire que la probabilité de la distribution finale est 1.

```
Definition total (X:distr D) := prob X (@fone _) == 1%U.

Record frame :Type := mkF {
    frm_impl :> oracle_implementation;
    frm_funs_total :> forall who x, total (o_funs frm_impl who x);
    frm_init_total :> total (o_init frm_impl)}.
```

Le Processus adverse

Le système d'oracles interagit avec l'adversaire en produisant une interaction. On ne décrit l'adversaire qu'en définissant cette interaction : de cette façon on peut raisonner avec un adversaire arbitraire. On lui permet par ailleurs d'avoir un espace mémoire.

On conçoit ici un adversaire comme une série d'actions. Un adversaire est donc défini pour un état interne et une fonction transitoire probabiliste (ainsi qu'un état initial).

A chaque étape, un adversaire a deux alternatives :

1. rendre sa réponse définitive et arrêter le jeu,
2. faire une requête au système d'oracles.

Définition 5.7 (Adversaire)**.** Un *adversaire* $\mathbb{A}$ (contre un système d'oracles $\mathbb{O}$) est donné par :

— un ensemble M_a de mémoire de l'adversaire,

— un état de la mémoire initial $\overline{\mathsf{M}_a} \in \mathsf{M}_a$,

— un ensemble de réponses possibles Resp,

— des fonctions pour émettre une requête envers un oracle o de $\mathbb{O}$ et mettre à jour sa mémoire ou pour donner sa réponse :

$$\begin{aligned} \mathsf{A} &: \mathsf{M}_a \rightharpoonup \mathcal{D}((o, \mathsf{In}(o)) \times \mathsf{M}_a) \cup \mathsf{Resp} \\ \mathsf{A}_{\downarrow} &: \mathsf{Out}(o) \times \mathsf{M}_a \to \mathcal{D}(\mathsf{M}_a) \end{aligned}$$

où $\mathcal{D}((o, \mathsf{In}(o)) \times \mathsf{M}_a) \cup \mathsf{Resp}$ dénote le fait que la fonction A de l'adversaire peut renvoyer soit une nouvelle requête, soit sa réponse.

La réponse définitive d'un adversaire dépend du jeu de sécurité dans lequel il évolue, cela peut être :

— un booléen pour dire dans quel monde il est si c'est un jeu d'indiscernabilité,

— une contrefaçon pour un MAC si c'est un jeu d'existence de contrefaçon,

— un clair pour un message chiffré,

— une collision pour une fonction de hachage,

— etc.

On définit l'ensemble des actions possibles en Coq de l'adversaire (de type `A`) en fonction de son état de type `Cstate` : une action est soit une nouvelle requête (auquel cas l'adversaire met à jour son état après réception de la réponse de l'oracle et recommence une action), soit sa réponse définitive.

```
Inductive Response Cstate A :=
      Request : forall (who : oracle_name os),
                       oracle_input os who
                       → (oracle_output os who → Cstate)
                       → Response Cstate A
  | Return : A → Response Cstate A.
```

Un adversaire de type `A` est défini par un type de données (le type de son état), une fonction de transition et un état d'origine.

```
Definition run_function state A := state → distr (Response state A).
```

```
Record Adversary (A:Type) :=
  mkC {
    c_state: Type;
    c_init: c_state;
    c_run:> run_function c_state A}.
```

Le système de l'adversaire est bien considéré comme probabiliste : à chaque étape, on obtient une distribution sur sa réponse.

Interaction oracle/adversaire

Au départ, on tire aléatoirement la mémoire initiale du système d'oracles. Puis, l'adversaire soumet sa première requête au système d'oracles, qui calcule la première réponse correspondante et se met à jour en conséquence. L'adversaire reçoit sa réponse, met à jour son état interne (mémoire...), et une autre itération du processus peut avoir lieu. L'interaction produit la trace de ces échanges successifs. Le processus s'arrête lorsque l'adversaire renvoie sa réponse.

On définit séquentiellement une interaction par une suite de requêtes successives d'un adversaire à un système d'oracles se terminant par la réponse définitive de l'adversaire. Une étape de l'interaction est une action de l'adversaire (en fonction de l'état courant dans la trace), qui choisit soit

— de faire une nouvelle requête et de rajouter une itération,

— de retourner un résultat.

Définition 5.8 (Interaction)**.** Une interaction entre un adversaire $\mathbb{A}$ et un système d'oracles $\mathbb{O}$ est une séquence d'actions affectant les états du système d'oracles $s \in \mathsf{M}_o$ et de l'adversaire $s_a \in \mathsf{M}_a$ procédant ainsi :

1. exécution de la fonction A sur s_a de l'adversaire,
2. deux cas se présentent :
   — l'adversaire donne sa réponse r, auquel cas l'interaction s'arrête et r est le résultat de l'interaction,
   — l'adversaire produit une requête (o, q) auprès d'un oracle,
3. exécution de l'oracle sur le paramètre donné par l'adversaire et l'état courant $o(s, q) = a$,
4. exécution de la fonction $\mathsf{A}_{\downarrow}(a, s_a)$ de l'adversaire et déclenchement d'une nouvelle action de l'adversaire (étape 1).

Elle est notée $\mathbb{A} \mid \mathbb{O}$. Par souci de simplification, lorsque l'adversaire donne sa réponse finale, on notera qu'il effectue une requête (o_{F}, r).

Les interactions entre le système d'oracles et l'adversaire sont définies ainsi : on utilise une fonctionnelle qui nous permet de s'abstraire du code de l'adversaire. Cette fonctionnelle prend en paramètre

1. la fonction de l'adversaire qui renvoie une distribution de résultats à partir de l'état et de la requête courante,
2. l'état et la requête courante.

Il calcule la réponse de l'adversaire et lance ensuite au besoin la fonction routine de l'oracle appelé sur ces paramètres et rajoute le résultat à la trace.

```
Variable (Frun: run_function state A).

Definition Result := Value Trace A.

Definition Finteract
  (I: Value Trace state → distr Result) (v: Value Trace state)
     : distr Result :=
   mlet rq := Frun (v_obj v) in
   match rq with
   | Request who what wher ⇒
       mlet res := oracle_call OF who (mkV what (v_state v) (v_trace v)) in
           I (mkV (wher (v_obj res)) (v_state res) (v_trace res))
   | Return a ⇒ Munit (mkV a (v_state v) (v_trace v))
    end.
```

L'interaction entre l'adversaire et le système d'oracles est donc bien considéré comme un système de transitions probabilistes. Le reste de la modélisation d'une interaction est technique et ne présente pas beaucoup d'intérêt ici. Elle est définie à l'aide d'un point fixe (donc définie par récurrence). Cette définition étant très technique en n'apportant pas beaucoup d'intérêt ici, nous n'entrerons pas plus dans les détails.

L'interprétation du temps

Le temps d'exécution tient compte de ce qu'un adversaire a (en général) la nécessité d'être efficace : il se trouve dans une position où il faut rompre le plus rapidement possible un schéma cryptographique. Par exemple pour un message chiffré, on finira par en connaître la teneur en faisant une attaque de force brute sur la clef de déchiffrement. Cela prend du temps, certes, mais on y parviendra. On se positionne donc dans une position où l'on considère une attaque comme étant efficace si elle est plus rapide que l'attaque triviale associée au schéma.

Il est fondamental de quantifier la probabilité d'une attaque réussie sur un système cryptographique par le temps d'exécution de l'adversaire. Intuitivement, plus un adversaire a de temps, plus il a de chances de réussir à casser les propriétés de sécurité du système.

La modélisation en Coq part du principe que le temps de calcul de l'adversaire est polynomial entre chaque requête (en fonction du paramètre de sécurité). Ainsi, on majore le nombre d'accès aux oracles par une matrice d'appels.

Le nombre d'appels d'un adversaire `X` au système est toujours un paramètre de sécurité. Il ne fait pas partie de la définition du système d'oracles, mais c'est une propriété de

l'interaction entre l'adversaire et du nombre d'appels maximal à chaque oracle présent dans la trace.

Définition 5.9 (Majorant d'appels supérieure aux oracles)**.** Soit $\mathbb{O}$ un système d'oracles, ayant comme ensemble de noms d'oracles N_o. Soit $b : \mathsf{N}_o \to \mathbb{N}$ une fonction associant à chaque oracle un nombre entier (le nombre maximal d'appels auxquels l'adversaire a droit durant l'interaction).

Une interaction $\mathbb{A} \mid \mathbb{O}$ respecte un majorant d'appels aux oracles b si et seulement si au cours de l'exécution l'adversaire $\mathbb{A}$ n'effectue pas plus d'appels à o que $b(o)$.

```
Variable call_bound : oracle_name os → nat.

Definition at_most_calls := forall State (OF:oracle_functions os State),
  (forall nam x, total (OF nam x)) →
    forall st, prob (interact X OF st (trace_init _ _) )
    (fun v ⇒ underflow (tally_trace (v_trace v) (Some call_bound))) == 0%U.
```

Pour tout oracle du système, si son exécution est totale, alors le nombre d'appels à chaque oracle (pour toute trace) doit être inférieur au total qui lui est autorisé. Pour vérifier cela, on décrémente un compteur à chaque fois qu'un appel est effectué dans la trace. Si à un moment donné dans la trace, le compteur est épuisé et qu'un appel est effectué, la propriété est violée.

Opérations de l'adversaire

On peut représenter les opérations de l'adversaire (qui sont des distributions sur le type de l'adversaire) comme une représentation monadique qui dispose de plusieurs opérateurs de base dont on évoque la définition dans le paragraphe suivant :

- `Cunit(a)` qui renvoie une distribution Dirac sur `a`,
- `Clet (v, f)` qui renvoie la composition séquentielle de deux processus `v` et `f` : on calcule la distribution `x` de `v` avant de calculer la distribution `f(x)`,
- `Cdraw(d)` retourne un tirage dans la distribution `d`,
- `Ccall(o,x)` appelle l'oracle `o` avec le paramètre `x`.

À l'aide de cette représentation monadique sur les distributions, on peut implémenter les calculs probabilistes séquentiels. Pour une utilisation plus transparente, on retrouve une syntaxe proche de celle d'un oracle.

Pour la définition de `Clet`, on se sert encore de la construction monadique de l'adversaire. On doit composer deux fonctions de l'adversaire, ici exécuter `X` puis `Y` sur la sortie de `X`. Le tirage dans une distribution (`Cdraw`) est assuré par la composition de `Mlet` sur la distribution avant de passer à la suite des calculs ; et l'assignation d'une valeur (`Cunit`) qui retourne une valeur donnée grâce à une distribution Dirac. `Ccall(o,x)` utilise simplement

la fonction `oracle_call` pour récupérer la distribution résultant de l'oracle appelé. [1]

Pour simplifier les preuves sur les interactions, on ajoute des preuves d'égalité de distributions se ramenant aux opérateurs `Munit` et `Mlet` : `Lemma interact_Cdraw`, `interact_Cunit` et `interact_Clet` en les faisant sortir de l'interaction.

```
Lemma interact_Cdraw : forall (d:distr A) st tr,
   interact (Cdraw d) OF st tr == Mlet d (fun a ⇒ Munit (mkV a st tr)).

Lemma interact_Cunit : forall a st tr,
        interact (Cunit a) OF st tr== Munit (mkV a st tr).

Lemma interact_Clet : forall st tr,
  interact (Clet X Y) OF st tr ==
  Mlet (interact X OF st tr) (fun va ⇒ interact (Y (v_obj va))
       OF (v_state va) (v_trace va)).

Lemma interact_Cdraw : forall (d:distr A) st tr,
   interact (Cdraw d) OF st tr == Mlet d (fun a ⇒ Munit (mkV a st tr)).

Lemma interact_Ccall : forall s t ,
    interact Ccall OF s t ==
    oracle_call OF on (mkV inp s t).
```

Les preuves sont essentiellement des réécritures des interactions, et les deux premières s'obtiennent rapidement. Dans le cas de `Clet`, toutefois, il faut prouver l'égalité de distribution avant l'assignation (par minoration et majoration), et également après. Cela s'explique par la présence de calculs inclus dans l'expression `Clet`.

Évènement

En cryptographie, on étudie régulièrement des évènements sur les interactions entre les systèmes cryptographiques et les adversaires. Nous devons donc être capable de raisonner sur ces évènements.

Définition 5.10 (Évènement sur une trace d'exécution)**.** Soit $\mathbb{O}$ un système d'oracles. Un événement E pour $\mathbb{O}$ est un prédicat sur les traces d'$\mathbb{A} \mid \mathbb{O}$, quelque soit l'adversaire $\mathbb{A}$. Il est noté $\mathbb{A} \mid \mathbb{O} : E$.

En Coq, on définit simplement un événement `event` comme une fonction qui associe à un élément une probabilité. `prob` est donc la fonction qui retourne la probabilité d'un événement dans une distribution particulière.

```
Variable D: Type.
```

1. Toutes ces définitions peuvent être retrouvées dans le fichier Comp_Monad.v, avec les preuves de leurs propriétés.

```
3  Definition event := D → U.

   Definition prob (X:distr D) (ev:event) : U := mu X ev.
6

   Definition frame_event :=event (Value Trace Resp).
```

Lorsque l'on considère les preuves cryptographiques, on a souvent besoin de travailler sur des traces d'exécutions, pour voir si un évènement qui rompt une propriété de sécurité est arrivé. Ainsi les principaux opérateurs des évènements comme la conjonction et la disjonction ont également été définis.

Nous sommes maintenant capables de définir une nouvelle catégorie d'événements sur les exécutions, et de quantifier la probabilité qu'ils se produisent. Soit un prédicat P sur $\mathsf{Xch} \times \mathsf{M}_o \times \mathsf{M}_o$. On définit les événements suivants :

F_P **:** il existe un moment de l'exécution où P est vraie,

G_P **:** Pour toute trace d'exécution, P est toujours vérifiée,

$E \mathsf{U} P$ **:** si P est fausse alors E doit être vraie jusqu'à ce que P devienne vraie.

5.3.2 Les Jugements

CIL prend en compte 2 types de jugements :

— une majoration sur la probabilité d'un événement de se produire, comme un succès de l'adversaire,

— une majoration sur la probabilité que deux systèmes d'oracles soient distingués l'un de l'autre.

Généralement, pour prouver la sécurité d'un système d'oracle, il faut montrer que ces majorants sont négligeables.

Informellement, on quantifie le risque qu'un adversaire puisse briser une propriété de sécurité à l'aide d'une fonction ϵ. La fonction ϵ est essentiellement la probabilité que l'adversaire puisse produire un événement donné, ou distinguer deux systèmes d'oracles. Cette fonction dépend du nombre d'appels de l'adversaire au système d'oracle. Intuitivement, plus un adversaire fait d'appels au système, plus il a de chances de forcer un événement ou de distinguer deux systèmes d'oracles. Nous avons donc : $\epsilon : (\mathsf{N}_o \to \mathbb{N}) \to [0, 1]$.

Définition 5.11 (Jugement sur les évènements)**.** Le jugement $\mathbb{O} :_\epsilon E$ est dit valide ($\models \mathbb{O} :_\epsilon E$) si et seulement si pour tout adversaire $\mathbb{A}$,

$$\Pr\left[\mathbb{A} \mid \mathbb{O} : E\right] \leq \epsilon$$

Le jugement d'un évènement permet surtout d'étudier la probabilité d'une attaque réussie contre le système.

Un système est considéré comme sûr au regard de ϵ et d'un nombre maximum d'appels aux oracles si, et seulement si, pour toute attaque contre le système, si le nombre d'ap-

pels aux oracles est contenu dans la matrice `call_bound` alors la probabilité de succès de provoquer l'évènement `ev` est inférieur à ϵ. Ce jugement est traduit par `negligible`.

```
Variable os : oracle_signature.

Record attack:Type := mkA
  {att_run:> Adversary os Answer;
    att_total : total_comp att_run}.

Definition outcome (frm:frame)
    (att:attack) (fev:frame_event os State Answer) :=
 prob (
   mlet init := o_init frm in
     interact att frm init (trace_init _ _)) fev.

Definition negligible (f:frame) (ev:frame_event os State Answer)
     (call_bound:oracle_name os → nat) (epsilon:U)
  := forall (att:attack), at_most_calls att call_bound →
       (outcome f att ev) <= epsilon.
```

Le jugement d'indiscernabilité permet de raisonner sur le comportement des oracles observable par l'adversaire, alors que le jugement de négligeabilité se prononce sur l'apparition d'évènements mauvais pour la sécurité du système.

Définition 5.12 (Jugement d'indiscernabilité)**.** Le jugement d'indiscernabilité $\mathbb{O} \sim_\epsilon \mathbb{O}'$, avec $\mathbb{O}$ et $\mathbb{O}'$ des systèmes d'oracles compatibles, est dit valide ($\models \mathbb{O} \sim_\epsilon \mathbb{O}'$) si est seulement si : pour tout adversaire $\mathbb{A}$

$$|\Pr[\mathbb{A} \mid \mathbb{O} : \mathrm{R} = \mathsf{true}] - \Pr[\mathbb{A} \mid \mathbb{O}' : \mathrm{R} = \mathsf{true}]| \leq \epsilon$$

où $\mathrm{R} = \mathsf{true}$ est une notation raccourcie pour $\mathsf{F}_{\lambda(o,r).o=o_\mathrm{F} \wedge (r=\mathsf{true})}$.

Deux cadres d'exécutions sont indiscernables au regard de ϵ et d'un nombre maximum d'appels aux oracles si, et seulement si, le nombre d'appels aux oracles est contenu dans la matrice `call_bound` alors aucun adversaire n'est capable de les distinguer, c'est à dire qu'aucune attaque ne peut produire un évènement différent avec une probabilité supérieure à ϵ.

```
Variables State State2: Type.

Definition True_ev St : frame_event os St bool:= (fun v ⇒ B2U (v_obj v)).

Definition indistinguishable
 (f1:frame os State)
 (f2:frame os State2) (call_bound:oracle_name os → nat) (epsilon:U):=
 forall (att:attack _ bool),
```

```
9 |  at_most_calls att call_bound →
  |  diff (outcome f1 att (@True_ev _)) (outcome f2 att (@True_ev _)) <= epsilon.
```

C'est la modélisation directe du jeu d'indiscernabilité vu dans la section 3.2.2.

5.3.3 Comportement idéal et réduction

La Bisimulation

Un outil important dans les preuves cryptographiques est de pouvoir modifier un système d'oracles en un autre système équivalent, avec lequel il sera plus facile de raisonner. Les deux systèmes sont en général similaires jusqu'à un certain point : par exemple, un schéma de signature sûr est équivalent à un schéma de signature idéal, sauf que le premier peut être trompé avec une probabilité négligeable.

Nous définissons la notion de bisimulation dépendant d'une condition : deux systèmes d'oracles sont bisimilaires (c'est à dire ont le même comportement), par rapport à une relation reliant leur mémoire, tant qu'une condition est vérifiée. C'est une généralisation de la bisimulation classique : la bisimulation classique est obtenue avec la condition true.

Soient $\mathbb{O}$ et $\mathbb{O}'$ deux systèmes oracles compatibles. On écrit $m \overset{(o,q,a)}{\longrightarrow}_{>0} m'$ si et seulement si $\Pr\left[\mathsf{O}_o(q,m) = (a,m')\right] > 0$.

La définition suivante formalise la relation entre deux systèmes d'oracles bisimilaires : si les changements d'un côté comme de l'autre maintiennent la relation entre les mémoires, alors la condition tient dans l'un des systèmes si et seulement si la condition tient également dans l'autre système. De la même façon, si la condition tient, alors la relation entre les deux mémoires tient toujours.

Définition 5.13 (La bisimulation dépendante des systèmes oracles)**.** Soient $\mathbb{O}$ et $\mathbb{O}'$ deux systèmes oracles compatibles. Soient $\varphi \subseteq \mathsf{Xch} \times \mathsf{M}_o \times \mathsf{M}'_o$ une condition et $\mathcal{R} \subseteq \mathsf{M}_o \times \mathsf{M}'_o$ une relation (on utilisera la notation infixe). $\mathbb{O}$ et $\mathbb{O}'$ sont bisimilaires dépendantes de $\mathcal{R}$,φ, noté $\mathbb{O} \equiv_{\mathcal{R},\varphi} \mathbb{O}'$, si et seulement si $\overline{\mathsf{M}}\ \mathcal{R}\ \overline{\mathsf{M}}'$, et pour tout $m_1 \overset{(o,q,a)}{\longrightarrow}_{>0} m_2$ et $m'_1 \overset{(o,q,a)}{\longrightarrow}_{>0} m'_2$ tels que $m_1\ \mathcal{R}\ m'_1$: $m_1, m_2 \in \mathsf{M}_o, m'_1, m'_2 \in \mathsf{M}'_o$ et $(o,q,a) \in \mathsf{Xch}$ les deux propriétés suivantes sont vérifiées :

La stabilité : si $m_2\ \mathcal{R}\ m'_2$ alors

$$\varphi\left((o,q,a), m_1, m_2\right) \Leftrightarrow \varphi\left((o,q,a), m'_1, m'_2\right)$$

La compatibilité : si $\varphi\left((o,q,a), m_1, m_2\right)$, alors

$$\Pr\left[\mathsf{O}_o\left(q, m_1\right) \in (\mathsf{Ans}, C)\right] = \Pr\left[\mathsf{O}'_o\left(q, m'_1\right) \in (\mathsf{Ans}, C)\right]$$

où C est la classe d'équivalence de m_2 sous $\mathcal{R}$.

Les bisimulations sont très proches des équivalences observatoires et de la logique de Hoare relationnelle. Elles nous permettent de justifier les preuves par simulation.

La règle de bisimulation entre deux systèmes d'oracles dépend de trois paramètres : une condition de validité sur chacun des systèmes et une relation entre les systèmes. Tant que les conditions (`phi` et `phi'`) sont vraies, tant que la relation (`R`) est vérifiée entre les deux systèmes et tant qu'à chaque échange les probabilités des deux évènements restent égales, les deux systèmes sont bisimilaires : chaque évènement a la même probabilité d'arriver au cours des deux interactions (modulo la probabilité des conditions `phi` et `phi'`).

On s'intéresse dans un premier temps à la bisimilarité d'un échange, pour généraliser ensuite à une exécution complète. Le lemme suivant montre que quelque soient les états initiaux, l'oracle appelé, le paramètre d'entrée et les évènements considérés, les distributions des évènements post-échange sont identiques si :

1. la relation entre les deux états initiaux est respectée,
2. pour tous les états post-échanges et tous les résultats les conditions `phi` et `phi'` sont vraies post-échange, [2]
3. tous les évènements sur le résultat ont la même distribution,
4. la probabilité de ces évènements est plus faible que les conditions respectives.

```
Variable s : oracle_functions os State.
Variable s' : oracle_functions os State'.

Variable phi : State → Exchange os State → bool.
Variable phi' : State' → Exchange os State' → bool.
Variable R : State → State' → Prop.


Lemma R_bisim_lemma :
  forall st st' who inp ev ev', R st st' →

  (forall ost ost' outp,
  phi ost (mkTR os st who inp outp) →
  phi' ost' (mkTR os st' who inp outp) →
  R ost ost' → ev (outp,ost) == ev' (outp,ost')) →

  ev <= (fun p ⇒ phi (snd p) (mkTR os st who inp (fst p)):U) →
  ev' <= (fun p ⇒ phi' (snd p) (mkTR os st' who inp (fst p)):U) →
    mu (s who (inp,st)) ev ==
    mu (s' who (inp,st')) ev'.
intros st st' who inp ev ev' HR Hcall Hev Hev'.
apply Ole_antisym;[apply R_sim_le|apply R_sim_ge];try assumption;
intros ost ost' outp Hphi Hphi' HR';apply Ole_refl_eq.
apply Hcall;auto.
symmetry.
```

2. Dans la modélisation Coq, la condition ϕ a été remplacé par une condition sur chacun des états des oracles.

```
apply Hcall;auto.
Qed.
```

Sur une exécution complète, on encapsule les évènements et les conditions sur la trace de l'exécution, à l'aide de `Box`, un point fixe qui vérifie que la condition soit vraie tout au long de la trace. A chaque étape de la trace, on demande que deux évènements aient la même probabilité d'arriver.

```
Lemma bisim_iter_Box: forall att,
  forall (cst : c_state att) (st : State) (st' : State') (tr : Trace os State)
    (tr' : Trace os State'),
  Box phi st tr →
  Box phi' st' tr' →
  R st st' →
  R_trace R tr tr' →
  forall runtime,
mu (iter_ (Finter att s) runtime (mkV cst st tr)) E_n_Box_phi ==
mu (iter_ (Finter att s') runtime (mkV cst st' tr')) E_n_Box_phi'.
```

Une fois cette preuve faite, on généralise le lemme à l'échelle de l'interaction.

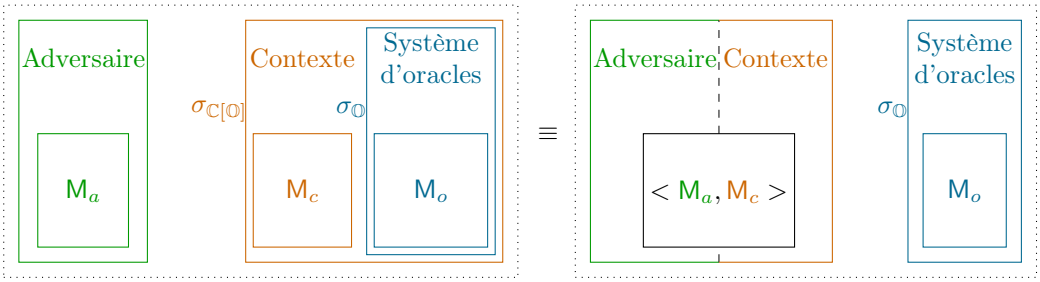
```
Lemma bisim_interact_Box: forall att (cst: c_state att) st st' tr tr',
  Box phi st tr → Box phi' st' tr' → R st st' → R_trace R tr tr' →
  mu (interact_gen att s (mkV cst st tr)) E_n_Box_phi
  == mu (interact_gen att s' (mkV cst st' tr')) E_n_Box_phi'.
```

Contextes et simulations d'oracles

Dans de nombreuses preuves cryptographiques, on permet à l'adversaire de simuler une partie d'un système d'oracles pour se focaliser sur un point du système qui présente un interêt.

Par exemple, si l'on modélise un protocole utilisant un moyen de chiffrement (que l'on n'explicite pas toujours), on peut simuler le reste du système et on utilise un système d'oracles pour modéliser le chiffrement et le déchiffrement. Cela permet de focaliser l'étude du protocole sur le schéma de chiffrement. On reprend alors un outil très utilisé dans les preuves cryptographiques : si il existe un adversaire efficace contre le protocole (ici la simulation composée avec le système d'oracles de chiffrement), alors il existe un adversaire efficace contre le schéma de chiffrement ce qui contredit soit l'hypothèse, soit la preuve de sécurité. Par contradiction, il n'existe donc pas d'adversaires efficaces contre le protocole.

C'est ce que l'on appelle les preuves par réduction. Pour pouvoir utiliser ce concept, la notion de contexte a été développée. Un contexte est un intermédiaire (un interpréteur) entre un système d'oracles et un adversaire : les questions de l'adversaire sont interprétées par le contexte qui calcule de nouvelles requêtes au système d'oracles. De la même façon, le contexte interprète la réponse avant de la renvoyer à l'adversaire.



(a) Composition d'un contexte avec un système d'oracles.

(b) Composition d'un contexte avec un adversaire.

Ces deux figures montrent les différentes compositions possibles entre un système d'oracles $\mathbb{O}$ de signature $\sigma_{\mathbb{O}}$, un contexte $\mathbb{C}$ défini pour la signature de $\mathbb{O}$, et un adversaire compatible avec $\mathbb{C}$.

FIGURE 5.2 – Composition d'un contexte avec un système d'oracle et un adversaire.

Le contexte joue donc le rôle de traducteur entre l'adversaire et le système d'oracles. Un contexte peut être appliqué à un adversaire pour former un nouvel adversaire, ou à un système d'oracles pour former un nouveau système. On capte ainsi l'esprit des preuves par réduction : à partir d'un système d'oracles compliqué, permettre à un contexte de simuler les calculs non cryptographiques du système pour se concentrer sur les primitives cryptographiques.

Un contexte repose essentiellement sur la notion de simulation : la fonction première d'un contexte est de simuler une partie des fonctionnalités d'un système d'oracles. Au final, le contexte pourra être composé soit avec un système d'oracles, soit avec un adversaire. La figure 5.2 représente les compositions possibles entre un contexte et un adversaire et entre un contexte et un système d'oracles.

Une simulation est définie en fonction de deux systèmes d'oracles $\mathbb{O}_{core}$ et $\mathbb{O}_{full}$ de signatures `os_core os_full`. La simulation $\mathbb{C}_{full}$ est de type `os_full` (associé à $\mathbb{O}_{core}$), et doit contenir pour l'adversaire le code entre la signature `os_full` et le système d'oracles $\mathbb{O}_{core}$. Elle est donc un adversaire, comme défini dans la sec. 5.3.1.

Définition 5.14 (Simulation de système d'oracles)**.** Une simulation d'un système d'oracles associée à $\mathbb{O}_{core}$ et $\mathbb{O}_{full}$ est une fonction pouvant faire appel à $\mathbb{O}_{core}$ qui, à un paramètre d'entrée de $\mathbb{O}_{full}$ e et un paramètre propre c, associe un adversaire contre $\mathbb{O}_{core}$.

Dans la plupart des cas, une simulation pourra effectuer un ou deux appels à un système d'oracles.

```
Variables os_full os_core : oracle_signature.

Definition oracle_simulation :=
  forall o_full, (oracle_input os_full o_full * complement)%type →
  Adversary os_core (oracle_output os_full o_full * complement)%type.
```

Ce que l'on veut définir, c'est l'équivalence de la composition d'une simulation et d'un système d'oracles $\mathbb{O}_{core}$ avec un système d'oracles $\mathbb{O}_{full}$ pour pouvoir définir de nouveaux

systèmes d'oracles par composition.

Définition 5.15 (Équivalence)**.** La composition d'une simulation $\mathbb{C}$ et d'un système d'oracles $\mathbb{O}_{core}$ est dite équivalente à un système d'oracle $\mathbb{O}_{full}$ si et seulement si quelles que soient les mémoires de $\mathsf{M}_c \times \mathsf{M}_o^{core}$ se projetant dans M_o^{full}, quelle que soit la trace de l'interaction et quelque soient l'oracle appelé et son paramètre d'entrée, le résultat d'une interaction avec $\mathbb{O}_{full}$ est identique à celui de la composition entre $\mathbb{C}$ et $\mathbb{O}_{core}$.

Ce qui en Coq se définit par :

```
Variable O_core : oracle_functions os_core state_core.
Variable O_full : oracle_functions os_full state_full.
Variable SI :  injection state_core state_full complement.

Definition oracle_simulation_eq sim :
       (forall x : oracle_name os2,
       oracle_input os2 x * complement →
       Adversary os1 (oracle_output os2 x * complement)) →
         Prop:=
  forall st_core sc tr_core o_full inp,
    O_full o_full (inp, SI st_core sc) ==
    mlet v := interact (sim o_full (inp, sc)) O_core st_core tr_core in
     ! (fst (v_obj v), SI (v_state v) (snd (v_obj v))).
```

On note $\mathbb{C}[\mathbb{O}]$ la composition d'une simulation $\mathbb{C}$ et d'un contexte $\mathbb{O}$.

Dans les preuves cryptographiques, on cherche à pouvoir exprimer un système d'oracles comme une composition entre une partie simulée et une primitive, afin de pouvoir définir un nouvel adversaire contre la primitive à l'aide de la simulation. En notant $\mathbb{A} \parallel \mathbb{C}$ la composition d'un adversaire $\mathbb{A}$ et d'une simulation $\mathbb{C}$ et en notant $\mathbb{A} \longleftrightarrow \mathbb{O}$ une interaction entre un adversaire $\mathbb{A}$ et un système d'oracles $\mathbb{O}$, voici la propriété que l'on obtient :

$$\mathbb{A} \parallel \mathbb{C} \longleftrightarrow \mathbb{O} \quad \equiv \quad \mathbb{A} \longleftrightarrow \mathbb{C}[\mathbb{O}]$$

Afin de compléter cette démarche, nous définissons maintenant l'interaction entre l'adversaire et la simulation, sensiblement similaire à celle d'un adversaire et d'un système d'oracles.

Une interaction peut être dans deux états :

— mettre à jour son état et procéder à l'action suivante,

— traiter une requête et procéder à l'action suivante.

```
Variable ctxt :  Adversary os_core A.
Variable simulation : oracle_simulation.
Inductive sim_state : Type :=
  S_main : c_state ctxt → complement → sim_state
| S_call :
```

```
forall sc o inp, (oracle_output os_core o → c_state ctxt) →
  c_state (simulation o (inp,sc)) → sim_state.
```

Pour l'exécution, on applique la simulation (ici `F`) pour obtenir la fonction de l'adversaire. Si l'on se trouve dans l'état de base, `S_main` (le premier état lors de l'exécution d'une simulation), on peut soit :

— retourner une réponse,

— faire une requête au système oracle $\mathbb{O}_{core}$ en passant dans l'état d'appel `S_call`.

Dans l'état d'appel, `S_call`, on suit les étapes suivantes :

1. on effectue la simulation,
2. selon le résultat :
   — on renvoie une réponse en passant par `S_main`,
   — on effectue une autre requête.

L'on définit alors une fonctionnelle `Fsim_run` afin de définir ensuite avec sa version monotone et à l'aide d'un point fixe `Fsim`, l'interaction d'une simulation et d'un adversaire.

```
Definition Fsim_run F (s:sim_state) :=
  match s with
    S_main stc sc ⇒
      mlet resp := c_run ctxt stc in
        match resp with
          Return a ⇒ Munit (Return (a,sc))
        | Request who what wher ⇒
            let sim := simulation who (what,sc) in
              F (S_call sc who what wher (c_init sim))
        end
  | S_call sc o inp kont sto ⇒
     let sim := simulation o (inp,sc) in
      mlet resp := c_run sim sto in
        match resp with
          Return (outp,sc') ⇒
            F (S_main (kont outp) sc')
        | Request who what wher ⇒
            Munit (Request who what
             (fun outp ⇒ S_call sc o inp kont (wher outp)))
        end
  end.
```

Si l'état de simulation demande une requête à un oracle, on procède à cette requête puis on met à jour l'état (de manière très similaire à la fonction `compose_run` présentée dans la section 5.3.1). Si l'état de la simulation est son état de base (`S_main`), alors on initie l'action de la simulation : soit il y a une requête à faire, soit c'est le résultat final que l'on

retourne. Si l'état de la simulation est un état de requête (`S_call`), on calcule la simulation de l'oracle, on calcule la réponse et on met à jour l'état.

Le reste de la définition de la simulation est technique, il s'agit essentiellement d'une preuve de monotonie. Au final, on est capable de définir un adversaire à partir d'une simulation.

```
Definition Restrict_Adversary : complement →
  Adversary os1 (A * complement) :=
    fun sc ⇒ mkC (S_main (c_init c2) sc) (Mfix Fsim).
```

On peut alors définir un cadre d'exécution `F` sur lequel on pourra raisonner. Si l'adversaire est total [3], et si la simulation est totale, alors l'ensemble adversaire/simulation forme lui aussi un système total.

```
Let F state OF (sts:sim_state _ simulation) st (tr_core:Trace os_core state)
tr_full :=
  match sts
    S_main cst sc ⇒
      interact_gen C (sim_O state OF) (mkV cst (st,sc) tr_full)
  | S_call sc who what wher cstsim ⇒
      mlet res := interact_gen (simulation who (what,sc)) OF
           (mkV cstsim st tr_core) in
        interact_gen C (sim_O state OF) (mkV (wher (fst (v_obj res)))
           (v_state res,snd (v_obj res))
           (trace_push (st,sc) who what (fst (v_obj res)) tr_full))
  end.

Lemma total_interact_sim_O_F: forall state OF sts st tr_core tr_full,
 (forall o inp, total (OF o inp)) →
 total (F state OF sts st tr_core tr_full).
```

On introduit deux lemmes facilitant les preuves d'égalité des distributions et la totalité de la composition des adversaires bornés avec le contexte.

```
Lemma  F_iter_mlub: forall state OF (sts:sim_state _ simulation) st
                    (tr_core:Trace os_core state) tr_full,
  F state OF sts st tr_core tr_full ==
  mlub (mlub (F_iter state OF sts st tr_core tr_full)).

Lemma total_Restrict: forall sc_init,
total_comp (Restrict_Adversary C simulation sc_init).
```

Il reste encore, pour compléter cet outil, à définir la nouvelle fonction majorant le nombre d'appels aux oracles.

3. On rappelle que la totalité d'un système de transition signifie également que celui-ci termine avec une probabilité 1.

Définition 5.16 (Matrice de majoration d'appels d'une simulation)**.** Une matrice d'appels pour une simulation S associée à $\mathbb{O}_{core}$ est la matrice associée à l'application linéaire donnant pour chaque oracle de S le nombre maximal d'appels effectués à chaque oracle de $\mathbb{O}_{core}$.

```
Record Matrix : Type := mkMx {
  Mx_fun :> (oracle_name os_full → nat) → (oracle_name os_core → nat);
  Mx_zero :  Mx_fun (@Nfzero _ ) == (@Nfzero _);
  Mx_plus :  forall f g, Mx_fun (Nfplus f g) == Nfplus (Mx_fun f) (Mx_fun g);
  Mx_eq_compat: forall f g, (f == g) → (Mx_fun f) == (Mx_fun g)
  }.

Variable Fmatrix: Matrix.

Lemma Restrict_runtime:
at_most_calls (Restrict_Adversary C simulation sc_init) (Fmatrix C_calls).
```

La modélisation de cette "matrice" est un 4-uplet contenant les propriétés essentielles d'une application linéaire : une fonction associant à chaque oracle de la simulation le nombre d'appels pour chaque oracle du système d'oracles noyau, et les preuves de linéarité de ces fonctions.

L'utilisateur définit une matrice pour lier le nombre d'appels aux oracles du système $\mathbb{C}[\mathbb{O}_{core}]$ et le nombre d'appels au système $\mathbb{O}_{core}$ (`C_calls`). Cette matrice est définie par une fonction pour chaque oracle de $\mathbb{C}$ au nombre maximal d'appels à $\mathbb{O}_{core}$: depuis le nombre d'appels fait à $\mathbb{C}[\mathbb{O}_{core}]$, on calcule le nombre maximal d'appels fait à $\mathbb{O}_{core}$ (suivant le nombre d'appels que chacune des procédures du contexte fait). On demande également que la simulation respecte cette matrice d'appels.

Cette section introduit des règles additionnelles qui peuvent être dérivées des résultats de ce qui précède, et établissent la validité de la logique. Les méthodes pour établir les prémisses externes utilisées dans CIL sont également discutées.

5.3.4 Règles

Nous commençons par présenter les concepts fondateurs et les théorèmes sur lesquels les règles de déduction de CIL sont basées. Nous adaptons les concepts directement de [BDKL10] avec les extensions présentées dans la section 5.5.

Pour modéliser les règles de CIL en Coq, on a recours à une règle universelle : si deux cadres d'exécution ont les mêmes distributions de messages (à distribution d'entrées égales, distribution de valeur de retour égale), ces deux cadres sont indiscernables.

Toutes les règles présentées ci-dessous ont été prouvées en Coq par Pierre Corbineau.

```
Definition dprojResult St Tr A
  (L:  distr (Value St Tr A)) :=
  Mlet L (fun x ⇒ Munit (v_obj x)).
```

```
Definition frame_R_equiv :=
 forall (att:attack os bool),
 (dprojResult (mlet init := o_init s in interact att s init
                          (trace_init os State))) ==
 (dprojResult (mlet init := o_init t in interact att t init
                          (trace_init os State'))).

Definition Rule_univ_statement :=
  frame_R_equiv →
  forall bnd, indistinguishable s t bnd 0.
```

La fonction `dprojResult` permet de projeter le résultat de la trace, afin de pouvoir comparer les vues de différentes interactions. Si ces vues ont la même distribution, alors les deux cadres d'exécution sont indiscernables.

L'Indiscernabilité comme distance

L'indiscernabilité en CIL peut être vue comme une distance : on retrouve en effet la réflexivité, la symétrie et l'inégalité triangulaire. On peut considérer également la propriété de séparation : si deux systèmes sont absolument indiscernables, c'est qu'ils sont parfaitement bisimilaires.

$$\frac{}{\mathbb{O} \sim_0 \mathbb{O}}(\text{REFL})$$

La règle de réflexivité est trivialement modélisée en Coq à l'aide de cette règle.

```
Definition Rule_refl_statement :=
  forall bnd, indistinguishable s s bnd 0.
```

$$\frac{\mathbb{O} \sim_{\epsilon_1} \mathbb{O}' \quad \mathbb{O}' \sim_{\epsilon_2} \mathbb{O}''}{\mathbb{O} \sim_{\epsilon_1+\epsilon_2} \mathbb{O}''}(\text{TRANS})$$

La règle de transitivité est modélisée tout aussi simplement. Où `s` ,`t` et `u` sont des cadres d'exécution, `bnd` une fonction d'appels aux oracles, `epsilon1` (resp. `epsilon2`) le majorant sur la probabilité de distinguer `s` de `t` (resp. `t` de `u`). `epsilon3` est donc un majorant sur la probabilité de distinguer `s` de `u`.

```
Definition Rule_trans_statement :=
  indistinguishable s t bnd epsilon1 →
  indistinguishable t u bnd epsilon2 →
  let epsilon3 := (epsilon1 + epsilon2) in
  indistinguishable s u bnd epsilon3.
```

$$\frac{\mathbb{O} \sim_\epsilon \mathbb{O}'}{\mathbb{O}' \sim_\epsilon \mathbb{O}}(\text{SYM})$$

On retrouve cette propriété en Coq grâce à la symétrie de l'opérateur `diff`.

Règles sur la bisimulation

Les règles suivantes sont des conséquences directes de la section 5.3.3, et impliquent la bisimilarité (dépendante) de systèmes d'oracles. La règle OR signifie qu'à moins que la condition de la bisimulation ne tienne plus, il n'y a aucun moyen de distinguer deux systèmes d'oracles bisimilaires.

Lorsque la bisimulation est parfaite et ne dépend d'aucune condition mais uniquement d'une relation entre les états, on utilise la règle `Rule_simple_or`. Si deux systèmes sont bisimilaires, alors ils sont indiscernables.

$$\frac{\mathbb{O} \equiv_{\mathcal{R},\top} \mathbb{O}'}{\mathbb{O} \sim_0 \mathbb{O}'}(\text{simple OR})$$

Pour utiliser le lemme, il faut prouver que la relation R est vraie à l'initiation et à chaque appel, et que pour chaque paire d'évènements ayant la même distribution, celle-ci est conservée à l'initialisation. C'est en fait un cas particulier de la règle OR où la condition φ serait toujours vraie.

```
  Hypothesis R_init : forall ev ev',
  (forall st st', R st st' → ev st == ev' st') →
  mu (o_init s) ev == mu (o_init s') ev'.

Hypothesis R_call_R :
forall st st' who inp ev ev', R st st' →

 (forall ost ost' outp,
R ost ost' → ev (outp,ost) == ev' (outp,ost')) →
mu (o_funs s who (inp,st)) ev ==
mu (o_funs s' who (inp,st')) ev'.

Variable bnd: oracle_name os → nat.

Let or_statement:= indistinguishable s s' bnd 0.
Theorem Rule_simple_or: or_statement.
```

La règle générale de bisimulation, OR, stipule que si $\neg\varphi$ n'arrive au cours de l'exécution qu'avec une probabilité ϵ, c'est à dire si φ est vrai tout au cours de l'exécution avec une probabilité de $1-\epsilon$, et si les deux systèmes sont bisimilaires par $\mathcal{R}, \varphi$, alors les deux systèmes sont discernables avec une probabilité d'au plus ϵ.

$$\frac{\mathbb{O} :_\epsilon \mathsf{F}_{\neg\varphi} \quad \mathbb{O} \equiv_{\mathcal{R},\varphi} \mathbb{O}'}{\mathbb{O} \sim_\epsilon \mathbb{O}'}(\text{OR})$$

La règle générale se complexifie avec la présence de la condition φ. En somme ce n'est

qu'une condition supplémentaire dans la preuve de la préservation de R lors d'un appel oracle. Il faut cependant aussi montrer que φ ne doit se produire au cours de la trace qu'avec une probabilité négligeable.

```
Hypothesis R_init : forall ev ev',
  (forall st st', R st st' → ev st == ev' st') →
  mu (o_init s) ev == mu (o_init s') ev'.

Hypothesis R_call_R :
   forall st st' who inp ev ev', R st st' →
   (forall ost ost' outp,
     phi ost (mkTR os st who inp outp) →
     phi' ost' (mkTR os st' who inp outp) →
     R ost ost' → ev (outp,ost) == ev' (outp,ost')) →
   ev <= (fun p ⇒ phi (snd p) (mkTR os st who inp (fst p)):U) →
   ev' <= (fun p ⇒ phi' (snd p) (mkTR os st' who inp (fst p)):U) →
mu (o_funs s who (inp,st)) ev ==
mu (o_funs s' who (inp,st')) ev'.

Hypothesis neg_phi :
 negligible (Answer:=bool) s (trace_event (Diamond (not_ phi))) bnd epsilon.

Let or_statement:= indistinguishable s s' bnd epsilon.
Theorem or_: or_statement.
```

L'évènement φ sera en général un mauvais évènement que l'on ne souhaite pas voir arriver : un secret a été révélé, une signature a été contrefaite, etc.

Les règles pour les contextes

CIL rassemble plusieurs règles de composition qui découlent directement de la section 5.3.3.

Au sein d'une preuve Coq, un contexte est défini comme un nouveau cadre d'exécution doté d'une mémoire propre, d'une distribution initiale sur cette mémoire, d'une implémentation de son code d'exécution, des preuves de totalité de l'implémentation ainsi que d'une matrice d'appels, afin de préserver la propriété du nombre d'appels maximum aux oracles. Ce cadre peut être composé avec un cadre d'un système d'oracles.

Définition 5.17 (Contexte)**.** Un *contexte* pour $\mathbb{O}_{core}$ noté $\mathbb{C}$ est une simulation associée à $\mathbb{O}_{core}$ définie par :

— une mémoire M_c propre avec une mémoire initiale $\overline{\mathsf{M}_c}$;

— un ensemble de noms N_o de procédures ;

— pour tout $c \in \mathsf{N}_o$, un domaine de requêtes $\mathsf{In}(c)$, un domaine de réponses $\mathsf{Out}(c)$, et

une implémentation :

$$\mathsf{C}_c : \mathsf{In}(c) \times \mathsf{M}_c \to \mathcal{D}(\mathsf{Out}(c) \times \mathsf{M}_c)$$

pouvant faire appel aux oracles de $\mathbb{O}_{core}$.

— une preuve de totalité pour chaque procédure ;

— une matrice de majoration d'appels aux oracles ;

— une preuve de respect de la majoration d'appels aux oracles.

```
Variable os os2 : oracle_signature.

Record frame_ctx := mk_simple_frame_ctx {
  fc_complement:Type;
  fc_init : distr fc_complement;
  fc_init_total : total fc_init;
  fc_simulation : oracle_simulation os os2 fc_complement;
  fc_total : forall sc who what, total_comp (fc_simulation who (what,sc));
  fc_bound_matrix : Matrix os os2;
  fc_bounded: forall who what sc,
  at_most_calls (fc_simulation who (what,sc))
      (fc_bound_matrix (unit_count os2 who))}.
```

Cette règle énonce le fait qu'aucun contexte ne peut aider à distinguer deux systèmes d'oracles indiscernables.

$$\frac{\mathbb{O} \sim_{\epsilon_o} \mathbb{O}'}{\mathbb{C}[\mathbb{O}] \sim_{\epsilon_c} \mathbb{C}[\mathbb{O}']}(\text{SUB})$$

On rappelle au lecteur que dans cette modélisation de CIL en Coq, ϵ est représenté et par une probabilité `epsilon` et par une borne sur les appels aux oracles `bnd` (pour représenter le temps de calcul de l'adversaire). Ici, `epsilon` sera le même avant et après application de la règle, tandis qu'on applique la matrice d'appels du contexte à `bnd`.

```
Variable SI : injection State State2 (fc_complement ctx).
Variable SI' : injection State' State2' (fc_complement ctx).

Hypothesis fcs : frame_ctx_simulates frm frm2 ctx SI.
Hypothesis fcs' : frame_ctx_simulates frm' frm2' ctx SI'.

Definition Rule_sub_statement:=
indistinguishable frm frm' (fc_bound_matrix ctx bnd) epsilon →
indistinguishable frm2 frm2' bnd epsilon.
```

Ici aussi, la transcription est simple à comprendre : les états `SI` et `SI'` sont les états "étendus" du système $\mathbb{C}[\mathbb{O}_{\text{core}}]$ et l'on a besoin des preuves de simulation du contexte appliqué aux deux systèmes.

5.4 Validité

Pour montrer la sécurité de systèmes cryptographique grâce à CIL, il reste à prouver le rapport exact entre une preuve en CIL et la sécurité asymptotique du système étudié.

Dans un premier temps, nous étudierons le lien entre la propriété étudiée et les hypothèses de sécurités utilisées, avant de conclure sur le rapport entre la sécurité asymptotique d'un système et la preuve en CIL de la sécurité de ce système.

5.4.1 De la propriété à l'hypothèse de sécurité

Une propriété de sécurité est énoncée en fonction d'un paramètre de sécurité η. De celui-ci découle deux paramètres de nos preuves : les ressources utilisées par l'adversaire, c'est-à-dire le nombre d'appels aux oracles auxquels il a droit `bnd`, ainsi que la probabilité qu'il réussisse son attaque ϵ. Ces deux paramètres sont des fonctions polynomiales de η.

Pour que la preuve soit complète, il faut donc justifier :

1. que la propriété de sécurité de chaque primitive implique l'hypothèse de sécurité utilisée (c'est-à-dire que la modification de l'accès aux ressources reste comprise dans la propriété de sécurité),
2. que la combinaison des avantages de chacun des adversaires contre les primitives reste négligeable,
3. que la preuve de la sécurité concrète du protocole implique la sécurité asymptotique du protocole si les primitives sont sûres asymptotiquement.

Les deux premières propriétés sont simples à montrer. La première concerne les contextes et le nombre d'appels effectués. Comme ce nombre est limité (constant), on n'applique qu'une transformation linéaire au majorant d'appels aux oracles, ce qui ne modifie pas la propriété asymptotique du système cryptographique puisque ce majorant reste polynomiale en fonction du paramètre de sécurité.

Lemme 5.1 (De la propriété à l'hypothèse)**.** *Soit $\mathcal{P}$ un schéma cryptographique, η un paramètre de sécurité, L une fonction linéaire et game un jeu de sécurité. Si pour toute machine de Turing en temps polynomial $\mathbb{A}$ et pour toute fonction polynomiale majorante d'accès aux oracles bnd il existe ϵ une fonction négligeable telle que*

$$AVG^{game}_{\mathcal{P},\mathbb{A},bnd}(\eta) \leq \epsilon(\eta)$$

alors il existe ϵ' une fonction négligeable telle que

$$AVG^{game}_{\mathcal{P},\mathbb{A},L\circ bnd}(\eta) \leq \epsilon(\eta)$$

Démonstration. Soient $\mathbb{A}$ un adversaire, et une fonction polynomiale majorante d'accès aux oracles *bnd*. Supposons ϵ une fonction négligeable telle que l'on ait :

$$AVG^{game}_{\mathcal{P},\mathbb{A},bnd}(\eta) \leq \epsilon(\eta)$$

Soit L une fonction linéaire. Alors $L \circ bnd$ est une fonction polynomiale. Donc il existe ϵ' une fonction négligeable telle que

$$AVG^{game}_{\mathcal{P},\mathbb{A},L\circ bnd}(\eta) \leq \epsilon'(\eta)$$

□

Ainsi, nous pouvons reposer les preuves de sécurité sur des bases solides. Un second facteur essentiel est la preuve que la combinaison linéaire de fonctions négligeables est également négligeable.

À partir de maintenant et pour la fin de la section, il est implicite que les adversaires $\mathbb{A}$ et les majorants d'appels *bnd* des lemmes et preuves sont polynômiaux en fonction du paramètre de sécurité. Le second lemme concerne la règle de transitivité des preuves : lors de l'utilisation de la règle de transitivité, on obtient une combinaison linéaire (de façon générale une simple somme) de fonctions négligeables.

Lemme 5.2 (Combinaison linéaire de fonctions négligeables)**.** *Soient $\epsilon_1, \epsilon_2, \cdots, \epsilon_n$ n fonctions négligeables. Alors toute combinaison linéaire de $\epsilon_1, \epsilon_2, \cdots, \epsilon_n$ est négligeable.*

Démonstration. Soient $\epsilon_1, \epsilon_2, \cdots \epsilon_n$ n fonctions négligeables. Soient $a_1, a_2, \cdots a_n \in \mathbb{N}^n$. Soit P un polynôme.

Soit a_{max} le coefficient maximal de $a_1, a_2, \cdots a_n \in \mathbb{N}^n$.

Par définition d'une fonction négligeable, on a :

$$\forall i, \exists X_i, x > X_i \Rightarrow \epsilon_i \leq \frac{1}{na_{max}P(x)}$$

Soit X_{max} le maximum de l'ensemble X_i.

Donc

$$\forall i, x > X_{max} \Rightarrow \epsilon_i \leq \frac{1}{na_{max}P(x)}$$

et

$$\forall i, x > X_{max} \Rightarrow a_i\epsilon_i \leq \frac{a_{max}}{na_{max}P(x)} = \frac{1}{nP(x)}$$

Donc

$$x > X_{max} \Rightarrow \sum_{n}^{i=1} a_i\epsilon_i \leq \frac{n}{nP(x)} = \frac{1}{P(x)}$$

□

5.4.2 CIL et sécurité asymptotique

Le lemme moteur de CIL permet de montrer que CIL permet également de raisonner sur la sécurité asymptotique d'un système. On montre que si les hypothèses de sécurité sont sûres asymptotiquement, et qu'on parvient à exprimer la sécurité du système en fonction

de la sécurité des hypothèses [4] alors le système est sûr de façon asymptotique.

On passe alors d'une sécurité concrète à une sécurité asymptotique. Pour montrer cela, on part des hypothèses de sécurité asymptotique aux hypothèses de sécurité utilisées par CIL. Puis on suppose que l'on a une preuve de sécurité en CIL, c'est à dire que la probabilité pour l'adversaire de gagner le jeu contre le système cryptographique étudié est une combinaison linéaire des probabilités maximales qu'un adversaire a de remporter les jeux de sécurité liés aux hypothèses de sécurité. Il faut enfin montrer que la probabilité obtenue est bien négligeable (en fonction du paramètre de sécurité).

Lemme 5.3 (De la sécurité concrète à la sécurité asymptotique)**.** *Soit un schéma cryptographique $\mathcal{S}$ ayant comme jeu de sécurité game basé sur n primitives cryptographiques $\mathcal{P}_1, \mathcal{P}_2, \cdots \mathcal{P}_n$ ayant pour jeux de sécurité respectivement $game_1, game_2, \cdots, game_n$.*

Soient η un paramètre de sécurité, $\{a_1, a_2, \cdots, a_n\} \in \mathbb{N}^n$, $L_1, L_2, \cdots, L_n$ des applications linéaires. Si

$$\begin{aligned}\forall bnd \in\ & polynomial,\\ & \Big(\forall i \in \{1,2,\cdots,n\}, \forall \mathbb{A}_i \in\ polynomial, \forall \epsilon_i, AVG^{game_i}_{\mathcal{P}_i,\mathbb{A}_i,L_i \circ bnd}(\eta) \leq \epsilon_i(\eta)\Big)\\ & \qquad \Longrightarrow \forall \mathbb{A} \in\ polynomial, AVG^{game}_{\mathcal{S},\mathbb{A},bnd}(\eta) \leq \sum_{n}^{i=1} a_i \epsilon_i(\eta) \quad (5.1)\end{aligned}$$

alors

$$\begin{aligned}\big(\forall i \in \{1,2,\cdots,n\}, \forall \mathbb{A}_i \in\ & polynomial, \forall bnd_i \in\ polynomial, \exists \epsilon_i\ \textit{négligeables t.q.}\\ & AVG^{game_i}_{\mathcal{P}_i,\mathbb{A}_i,bnd_i}(\eta) \leq \epsilon_i(\eta)\big)\\ \Longrightarrow \forall \mathbb{A} \in\ & polynomial, \forall bnd \in\ polynomial, \exists \epsilon\ \textit{négligeable t.q.}\\ & AVG^{game}_{\mathcal{S},\mathbb{A},bnd}(\eta) \leq \epsilon(\eta) \quad (5.2)\end{aligned}$$

Ici, l'implication (5.1) est démontrée par la preuve que l'on a obtenue avec CIL : elle est valable pour tout adversaire, tout majorant d'appels et toute probabilité de réussite de l'adversaire. Les applications L_i sont les matrices de transformation des contextes des preuves CIL. On veut montrer ici que si cette implication est vraie alors la sécurité asymptotique des primitives entraîne la sécurité asymptotique du schéma (5.2).

La preuve découle directement des deux lemmes précédents.

Démonstration. Soit un schéma cryptographique $\mathcal{S}$ ayant comme jeu de sécurité *game* basé sur n primitives cryptographiques $\mathcal{P}_1, \mathcal{P}_2, \cdots \mathcal{P}_n$ ayant pour jeux de sécurité respectifs $game_1, game_2, \cdots, game_n$. Soient η un paramètre de sécurité, $\{a_1, a_2, \cdots, a_n\} \in \mathbb{N}^n$, $L_1, L_2, \cdots, L_n$ des applications linéaires, $\mathbb{A}$ un adversaire polynomial contre $\mathcal{S}$ (dans le jeu *game*) et *bnd* un majorant polynomial d'accès aux oracles.

4. C'est-à-dire que la probabilité d'une attaque réussie contre le système est majorée par une combinaison linéaire des majorants sur les probabilités d'attaque réussie sur chacune des hypothèses.

Supposons pour $\mathbb{A}$ et bnd :

$$\Big(\forall i \in \{1,2,\cdots,n\}, \forall \mathbb{A}_i, \forall \epsilon_i, AVG^{game_i}_{\mathcal{P}_i,\mathbb{A}_i,L_i \circ bnd}(\eta) \leq \epsilon_i(\eta)\Big) \\ \implies AVG^{game}_{\mathcal{S},\mathbb{A},bnd}(\eta) \leq \sum_{n}^{i=1} a_i \epsilon_i(\eta)$$

Supposons encore que

$$\forall i \in \{1,2,\cdots,n\}, \forall \mathbb{A}_i, \forall bnd_i, \exists \epsilon_i \text{ négligeables t.q. } AVG^{game_i}_{\mathcal{P}_i,\mathbb{A}_i,bnd_i}(\eta) \leq \epsilon_i(\eta)$$

On obtient :

$$AVG^{game}_{\mathcal{S},\mathbb{A},bnd}(\eta) \leq \sum_{n}^{i=1} a_i \epsilon_i(\eta)$$

Et comme $\epsilon_1, \epsilon_2, \cdots \epsilon_n$ sont négligeables, $\sum_n^{i=1} a_i \epsilon_i$ est également négligeable.

Donc

$$\forall \mathbb{A}, \forall bnd, \exists \epsilon \text{ négligeable t.q. } AVG^{game}_{\mathcal{S},\mathbb{A},bnd}(\eta) \leq \epsilon(\eta)$$

□

Ces lemmes sont généraux à CIL et montrent la validité de toutes les preuves réalisées avec cette logique en sécurité concrète et asymptotique.

5.5 Agent d'observation

Inclure l'intrusion dans le modèle d'attaque est pertinent lorsque l'objet est connecté à un réseau potentiellement corrompu (par une ou plusieurs entités), et qu'il est utilisé pour traiter une information sensible. Par exemple, un ordinateur raccordé à internet est susceptible d'héberger un agent d'observation pouvant lire des informations personnelles. On cherche donc à prendre ces intrusions en compte dans les preuves.

Dans cette section, nous présentons cette extension conçue lors de ce travail.

La résistance à l'intrusion est prise en compte dans le modèle BSM (Bounded Storage Memory : mémoire de stockage bornée) comme un circuit donné de l'adversaire aux parties, puis exécuté par ces parties sur leur propre mémoire, et dont le résultat (de taille bornée) est renvoyé à l'adversaire à la fin de la session.

L'idée principale pour manipuler la résistance aux intrusions est d'introduire un nouvel argument Γ (en fait un circuit décrivant une fonction) aux autres arguments du système d'oracles. L'implémentation du système devra alors retourner le résultat du circuit Γ.

Il est alors clair que la structure de la preuve restera inchangée. Nous pouvons maintenant supposer que pour une graine aléatoire fraîche, le comportement du système oracle est déterministe. On peut alors considérer Γ comme une fonction de cette graine et de la mémoire du système. Le circuit n'étant pas borné en taille, il peut entièrement simuler

l'adversaire et le système d'oracles. Ainsi, on peut simplement passer Γ à l'oracle d'initialisation de session, et l'exécuter sur la mémoire locale.

Définition 5.18 (Agent d'observation)**.** Un agent d'observation Γ pour $\mathbb{O}$ est une fonction modélisée par un circuit de taille non bornée de type

$$\Gamma : \{0,1\}^s \to \{0,1\}^\gamma$$

où s est la taille de tous les aléas cumulés de $\mathbb{O}$ et où γ dénote la borne supérieure de la taille du résultat du circuit. Γ est envoyée comme paramètre par l'adversaire au système oracle (lors du premier appel d'oracle pour une primitive cryptographique ou à chaque début de session pour un protocole).

L'agent d'observation n'est pas pris en compte dans la modélisation en Coq de CIL, mais au niveau de la définition du système d'oracles avec un nouveau paramètre de sécurité : la taille de la réponse de l'agent d'observation à l'adversaire. De manière générale, on le représente comme une paire dépendante composée d'une taille de retour et d'une fonction sur les aléas du système. Par exemple, pour un système d'oracles avec deux valeurs aléatoires de taille `nr`, on aura :

```
Definition Pcircuit n :=
  bitstring_n nr → bitstring_n nr → bitstring_n n.

Record circuit_gen:= mkCirc {
  circ_size:nat;
  circf:>Pcircuit circ_size
}.
```

On remarque que `circuit_gen` définit une chaîne de bits de longueur arbitraire, choisie par l'utilisateur. Au moment de l'initialisation de la session du système d'oracles, on détermine l'aléa du système, en faisant tous les tirages aléatoires nécessaires à cette session. L'agent d'observation est donc une fonction de ces données aléatoires et sa valeur de retour est calculée au moment de l'initialisation. De cette façon, on modélise le fait que n'importe quel agent d'observation peut être exécuté sur la mémoire du système oracle : comme on ne restreint ni la taille, ni les fonctionnalités de l'agent d'observation, il peut, par exemple, simuler entièrement le système d'oracles.

Chapitre 6

Certification en Coq du protocole

Sommaire

La modélisation du modèle BSM en Coq en utilisant CIL est particulièrement adaptée. Le système d'inférence apporte une preuve de sécurité confortable, tout en étant suffisamment souple pour permettre d'incorporer directement dans les oracles la gestion de la taille de l'information qui fuit vers l'adversaire. On peut donc utiliser en l'état cette logique.

Cette flexibilité ne se retrouve pas dans tous les systèmes. Dans le modèle symbolique, par exemple, on n'a pas réussi à modéliser la fuite partielle d'information. Ceci est dû à la nature des théories équationnelles sur lesquelles on raisonne : la modélisation d'un protocole établit des relations entre les différents objets qui circulent sur le réseau, permettant ou non le déchiffrement de données secrètes. Dans le cas d'une fuite d'information, la mise en équation est contre-intuitive. On peut par contre modéliser la fuite totale d'information (en révélant une clef, par exemple) comme c'est fait par Arapanis et. al. [ACKR13]. Dans

notre cas, on ne cherche pas à prouver un secret éternel, mais une résilience à une fuite d'information, qui pose davantage de problèmes en terme de modélisation.

Nous avons donc choisi de modéliser les agents d'observation envoyés aux parties honnêtes en tant que fonctions des données aléatoires du système d'oracles (en faisant l'hypothèse que cette fonction pouvait simuler toutes les manipulations de l'adversaire). La taille de l'information renvoyée est contrôlée par le système d'oracles, et lorsque la borne est atteinte, on ne permet plus aucune fuite. En modélisant ainsi le BSM, on garde suffisamment de flexibilité pour pouvoir aussi modéliser les autres fuites d'information, comme les attaques par canaux cachés (comme les attaques par consommation électrique). On peut étendre la fonction sur l'état entier du système, mais en considérant le fait que la fonction peut simuler l'état des données déterministes du système, cela complique énormément la modélisation sans nous faire gagner de finesse dans la preuve.

La grande mémoire partagée par les deux parties est modélisée comme une fonction prenant en paramètre deux nonces. De cette façon, on modélise la fonction résistante aux intrusions en faisant l'hypothèse implicite que la mémoire partagée est immense. En permettant à l'adversaire un accès non limité à cette fonction, on s'assure de capturer l'essence du modèle : l'adversaire ne peut retrouver qu'une information partielle et non suffisante de ce qui sert à dériver la clef d'authentification.

On se place également dans le modèle oracle aléatoire (*Random Oracle Model*, dénoté ROM par la suite). Dans ce modèle, on peut faire appel à un oracle aléatoire. Cet oracle, lorsqu'il est appelé sur un paramètre frais (aucune requête avec ce paramètre ne lui a été posée), tire une réponse au hasard (uniformément) dans l'espace des réponses possibles. Ces fonctions sont modélisées ainsi : à chaque requête originale (si le paramètre fourni n'a jamais été traité par la fonction), alors la réponse est tirée uniformément dans le domaine image. Un table des réponses données est maintenue, et sert à pouvoir vérifier si le paramètre fourni est original, et à fournir la réponse déjà donnée si tel est le cas.

6.1 Présentation de la preuve

Pour passer de la preuve papier à une preuve plus formelle, il a fallu en tirer un fil conducteur liant les propriétés des primitives cryptographiques utilisées à la propriété de sécurité de secret perpuel des clefs de session.

Le principe de cette preuve formelle est de montrer l'équivalence du protocole original à un protocole idéal (ayant de fait les propriétés désirées). Le protocole idéal ne fait fuir *aucune* information sur les données privées car aucun des messages échangés n'en contient.

Les versions idéales des fonctions sont ainsi modélisées à l'aide de listes qui permettent de conserver les chiffrements, les codes d'authentification et les hachages demandés afin de garantir un comportement cohérent du système. Ainsi, la création d'un nouveau chiffrement, code ou hachage est réellement aléatoire, et l'on peut retrouver le texte clair associé, vérifier la validité du code ou recalculer le hachage en utilisant ces listes. En particulier, la fonction de hachage est modélisée dans le ROM. Ce qui implique que nous conservons

en mémoire une liste des hachages qui ont été demandés par l'adversaire et le système d'oracles au cours des échanges.

De façon générale, le protocole est modélisé par (ré)action : Alice est modélisée en 3 actions distinctes (envoyer N_A, recevoir N_B et renvoyer sa clef publique authentifiée, recevoir la clef de session chiffrée), Bob est modélisé en 2 actions distinctes (recevoir N_A et renvoyer N_B, recevoir la clef publique et renvoyer chiffrée la clef de session), ainsi qu'un oracle de réinitialisation de session (voir la figure 6.4).

On définit des routines pour la génération de code d'authentification, de chiffrement, déchiffrement et hachage des messages, afin d'éviter les répétitions de codes et d'obtenir une modélisation proche de la programmation (ce qui est facilité par la syntaxe de Coq, qui rend transparentes les actions des oracles).

On étudie la sécurité du protocole sur plusieurs sessions successives (i.e. les sessions en parallèle ne sont pas prises en compte par ce modèle). Dans le système d'oracles il y a donc un oracle initiateur de session chargé d'effacer la mémoire de la session précédente et de tirer les entrées aléatoires de la session.

Pour prendre en compte la résistance aux intrusions (i.e. la résistance aux agents d'observation), nous allons autoriser l'adversaire à envoyer une fonction qui s'exécute sur l'état du système d'oracles en tant que paramètre à l'oracle d'initialisation. Celui-ci l'exécute sur sa mémoire et retourne le résultat à l'adversaire. La taille du résultat est bornée, en revanche la fonction a les limites calculatoires de l'adversaire. Si l'adversaire le souhaite, il peut donc encapsuler son propre algorithme dans la fonction intrusive. La somme de la taille de ces résultats est bornée par la taille de la mémoire de l'adversaire. Nous voulons prouver la résistance aux intrusions héritée de la fonction f.

Ce protocole est essentiellement séquentiel, on le divise (grâce aux contextes) en deux cadres : dans le premier, les deux parties échangent leurs nonces et s'en servent pour calculer une clef d'authentification à court terme. Dans la seconde partie du protocole, on se sert de cette clef pour échanger la clef de session à moyen terme.

6.1.1 Le Principe

Par souci de clarté, nous commençons par détailler la preuve sans considérer l'intrusion.

Nous voulons montrer que le protocole présenté dans la section 4.2 est indiscernable au cours d'une session test d'un protocole idéal qui n'utilise *aucune* donnée secrète au cours de ses échanges.

Le protocole idéal est essentiellement identique à l'original, si ce n'est que pour les sessions tests (celles sans agent d'observation), toutes les données qui transitent sur le réseau et qui dépendent de K (la donnée secrète partagée) sont maintenant tirées uniformément sur l'ensemble des valeurs possibles, les chiffrements sont maintenant des chiffrements de 0 (une chaine de bits mis à la valeur 0) et les codes d'authentifications sont tirés uniformément dans l'espace des codes possibles. Ce protocole idéal est présenté dans la table 6.1 (uniquement pour les sessions de test).

Alice	Bob
$N_A \xleftarrow{\$} \mathcal{N}$, envoie N_A $S'_A \stackrel{\text{def}}{=} f'(N_A, N_B)$	$N_B \xleftarrow{\$} \mathcal{N}$, envoie N_B $S'_B \stackrel{\text{def}}{=} f'(N_A, N_B)$
$(pk_A, sk_A) \xleftarrow{\$} \mathcal{K}$, envoie $(pk_A, \mathrm{MAC}^{\mathrm{id}}_{S'_A}(A : pk_A))$	vérifie $\mathrm{MAC}^{\mathrm{id}}$, $\kappa_i \xleftarrow{\$} \{0,1\}^k$, $c \leftarrow \mathcal{E}_{pk_A}(0)$, envoie $(c, \mathrm{MAC}^{\mathrm{id}}_{S'_B}(B\|c))$
κ_i efface tout sauf K	efface tout sauf K

Avec

$$f'(X,Y) = \begin{cases} \xleftarrow{\$} \{0,1\}^k & \text{si le couple } (X,Y) \text{ est encore inconnu à la fonction,} \\ \text{le précèdent résultat} & \text{sinon.} \end{cases}$$

En cas d'échec d'une de ses vérifications, la partie renvoie un `nul_string`.

TABLE 6.1 – Protocole idéalisé pour les sessions tests

Ainsi, dans le premier échange de messages, les valeurs N_A et N_B n'influencent directement pas la valeur de S'. On construit une fonction f', au fur et à mesure de l'échange, à la distribution uniforme, qui à chaque nouveau couple (N_A, N_B) associe un élément du co-domaine de f tiré au hasard. Si l'adversaire a interféré dans l'échange de nonce, alors la valeur de S' chez chacune des parties sera indépendante. Sinon elle sera "magiquement" identique. Dans la seconde partie, on ne chiffre pas la clef κ_i, mais 0, ce qui préserve κ_i de toute fuite d'information. Si le déchiffrement de ce message donne bien 0, alors la clef de session se retrouve "magiquement" partagée entre les deux parties.

On modélise ces deux protocoles en systèmes d'oracles. Les échanges sont maintenant considérés comme des paramètres d'entrée des oracles et leur réponse. Le système d'oracles est composé d'une série d'oracles comprenant entre autres les implantations de :

— f, la fonction résistante aux intrusions, qui prend en paramètre deux paramètres et renvoie une valeur,

— une fonction génératrice de code d'authentification et la fonction vérificatrice de ces codes,

— un système de chiffrement et de déchiffrement asymétrique, ainsi que la génération des clefs,

— une fonction de hachage.

La structure de la preuve est présentée dans la figure 6.1. La conclusion est située à la racine de l'arbre, les feuilles étant les hypothèses nécessaires à la preuve. Les signes "=" désignent une bisimulation entre les systèmes. Les doubles flèches montrent une implication due à une application d'une règle de CIL : $\Uparrow_{trans}$ montre une application de la règle de transitivité, $\Uparrow_{\mathbb{C}}$ est une application du contexte $\mathbb{C}$.

La preuve repose sur la réécriture des systèmes d'oracles en systèmes de contextes.

$$
\begin{array}{ccccccccccc}
 & & & & \pi & & & \sim & \pi^{id} & & \\
 & & & \preccurlyeq & & & & \Uparrow_{trans} & & \preccurlyeq & \\
 & & \pi_0[\pi_1] & \sim & \pi_0\left[\pi_1^{id}\right] & & & = & \pi_1^{id}[\pi_0] & \sim & \pi_1^{id}\left[\pi_0^{id}\right] \\
 & & & \Uparrow_{\pi_0} & & & & & & \Uparrow_{\pi_1^{id}} & \\
 & & \pi_1 & \sim & \pi_1^{id} & & & & \pi_0 & \sim & \pi_0^{id} \\
 & \preccurlyeq & & \Uparrow_{trans} & & \preccurlyeq & & & \| & & \| \\
\mathcal{E}[\mathrm{MAC}] & \sim & \mathcal{E}\left[\mathrm{MAC}^{id}\right] & = & \mathrm{MAC}^{id}[\mathcal{E}] & \sim & \mathrm{MAC}^{id}\left[\mathcal{E}^{id}\right] & & H[K] & \sim & H\left[K^{id}\right] \\
 & \Uparrow_{\mathcal{E}} & & & & \Uparrow_{\mathrm{MAC}^{id}} & & & & \Uparrow_{H} & \\
\mathrm{MAC} & \sim & \mathrm{MAC}^{id} & & \mathcal{E} & \sim & \mathcal{E}^{id} & & K & \sim & K^{id}
\end{array}
$$

FIGURE 6.1 – Structure de la preuve

Ainsi, on peut concentrer la preuves sur les briques cryptographiques : f, la fonction de hachage, les fonctions de chiffrements et celles d'authentification. On distingue les deux parties du protocole : la première partie de mise au point d'une clef commune d'authentification, la seconde partie d'échange d'une clef de session. La transitivité permet d'obtenir la conclusion.

Nous présentons dans la figure 6.2 la preuve CIL telle que prouvée en Coq. Le symbole $\sim_\epsilon$ note la relation d'indiscernabilité (à ϵ près). Notez que $\sim_0$ est une bisimulation (chacune à une propriété de relation près, non explicitée sur cette figure) dont la preuve est faite. Les noms des règles de CIL utilisées sont notés sur le coté de chaque ligne de l'arbre de preuve.

Le lecteur notera que dans cette preuve, nous n'utilisons que les règles de contexte SUB et de transitivité TRANS. La règle SUB est en soi suffisemment puissante pour prouver des jugements d'indiscernabilité non-triviaux.

La seconde propriété de cette preuve est le grand nombre de systèmes différents gérés. Cela entraîne nécessairement une grande répétition de code. Cependant, ces réécritures sont la plupart du temps inévitables puisque nous devons absolument retrouver aux feuilles les hypothèses de bases (c'est à dire les hypothèses de sécurité des primitives), et d'autre part le découpage du protocole a ses attraits, d'un point de vue pédagogique tout du moins.

On notera également la réécriture de systèmes de contextes associés à des systèmes d'oracles. Bien qu'elles alourdissent la preuve et qu'elles sont l'occasion d'une répétition de code facultative, elle permettent néanmoins de pouvoir garder un système d'oracle lisible lorsque l'on s'approche de la racine de la preuve (ce qui est loin d'être négligeable pour un lecteur humain). Bien que n'étant pas indispensables en soi, ces réécritures permettent de simplifier la lecture des modélisations, ainsi que d'isoler les compositions les unes des autres pour mieux identifier les éventuels bogues et erreurs de modélisation.

Un système d'oracles contient un oracle pour chaque "réaction" d'une des parties. C'est-à-dire qu'Alice sera codée en trois parties :

i) l'envoi de N_A,

ii) la réaction à la réception de N_B par l'envoi de sa clef publique authentifiée,

iii) la réception de sa clef de session.

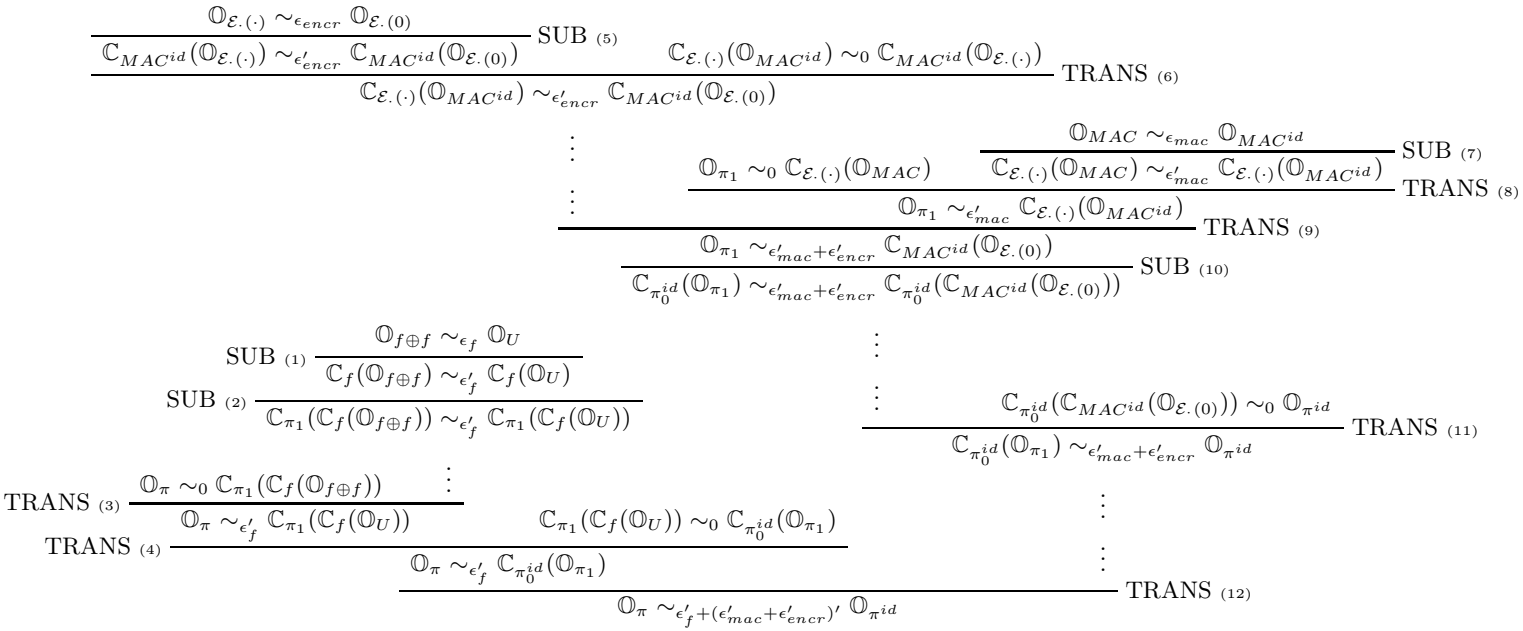


FIGURE 6.2 – Arbre de preuve pour la sécurité du protocole de Dziembowski.

Bob sera codé en deux parties :

i) la réaction à la réception de N_A par l'envoi de N_B,

ii) la réaction à la réception de la clef publique authentifiée d'Alice par l'envoi de la clef de session chiffrée et authentifiée.

A ces cinq oracles s'ajoute l'oracle d'initialisation de session qui a en charge d'effacer les informations de la session précedente et d'effectuer les tirages de la nouvelle session (nouveaux nonces et paire de clefs asymétriques) et d'évaluer l'agent d'observation de l'adversaire.

Pour simuler l'aspect séquentiel du protocole (pour chaque partie), on introduit dans l'état du système d'oracles deux variables `Alice_State` et `Bob_State` contenant la progression de chaque partie. Ainsi, il n'est pas possible pour l'adversaire de modifier l'ordre d'exécution de chacune des parties.

6.1.2 L'architecture de la preuve

Nous avons adopté une division de la preuve par le sens : chaque système d'oracles ou de contexte ainsi que ses preuves courantes (preuve de totalité...) sont contenus dans un seul fichier. Ce choix (certes pratique) a imposé une taille de fichier excessive : il eût fallu (avec le recul) trouver le moyen de morceler ces preuves en plusieurs fichiers.

Malheureusement, il s'avère que cette structure n'est pas triviale à mettre en place : on dépend pour définir un contexte d'un paramètre de sécurité. Si l'on adopte des modules pour définir le contexte, puis que l'on crée de nouveaux modules pour les morceaux de la preuve de totalité (par exemple) attenante (dans l'idéal une preuve de totalité par routine), au moment de regrouper les différents modules, ils sont de natures différentes et ne peuvent servir lors de la preuve de totalité du système.

Cela implique une forte hiérarchie dans l'architecture des modules et donc des fichiers. Celle-ci doit être construite selon un arbre, dont la racine serait la conclusion de la preuve, les feuilles les modules de base modélisant les systèmes de primitives, et les nœuds les modules modélisant les systèmes intermédiaires ainsi que les preuves de bisimilarité et les lemmes intermédiaires. Ainsi, le module de la conclusion défini par des paramètres de sécurité peut appeler les divers modules dont il dépend avec ces paramètres de sécurité, et contenir directement les différentes preuves requises par la conclusion.

Dans la suite de ce chapitre, nous allons vous présenter plusieurs fichiers et preuves. Dans la section 6.3 nous présentons la modélisation du protocole original. Celle de l'idéalisation est décrite dans la section 6.4. La section 6.5 montre la définition d'un contexte, avec ses preuves attenantes. Ensuite, la section 6.6 montre une preuve de bisimulation. Nous rappelons que la section 4.5 explicitait le théorème principal et les hypothèses de sécurité.

La preuve Coq consiste en une douzaine de fichiers, chacun modélisant un système d'oracles, un contexte et ses preuves adjacentes ou une bisimulation. Nous présentons l'architecture de ces fichiers dans la figure 6.3. Par souci de clarté, les fichiers contenant

Type	Définition
nonce	bitstring_n nonce_size
hash	bitstring_n hash_size
MAC	bitstring_n MAC_size
key	bitstring_n key_size
session_key	bitstring_n session_key_size
cipher	bitstring_n cipher_size
f_out	bitstring_n f_output_size
Alice_randoms_gen	mkArnd {Arnd_nonce : nonce; Arnd_skey : key}
Bob_randoms_gen	mkBrnd {Brnd_nonce : nonce; Brnd_skey : key}

TABLE 6.2 – Liste des types utilisés

les définitions, lemmes et tactiques généraux et/ou communs sont omis (Bits.v, Swap.v, Common.v, Generic.v).

6.2 Modélisation des types de base

Certains types sont récurrents dans cette preuve : le type nonce, par exemple, mais également les types de clefs asymétriques, les types de retour des fonctions de hachage et des chiffrements...

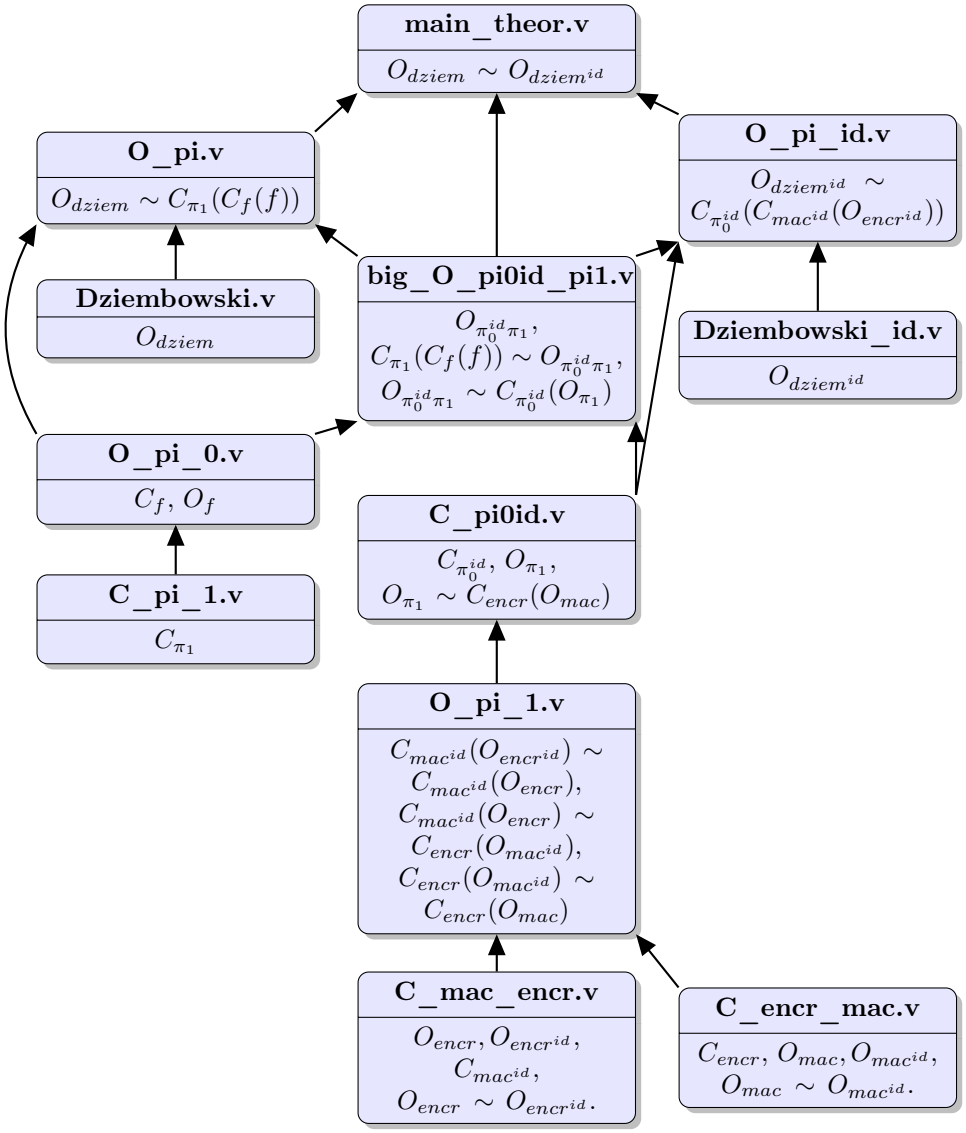
La plupart des types utilisés ici sont modélisés par des chaînes de bits de longueur connue. Un type est donc défini par une taille. On distingue deux chaînes particulières : la chaîne nulle `nul_string` (tous ses bits sont 0) et la chaîne "pleine" `full_string` (tous ses bits sont à 1).

```
Fixpoint bitstring_n (n:nat) : Type :=
  match n with
    |0 ⇒ Datatypes.unit
    | S n' ⇒ (bitstring_n n' * bool)%type
  end.
```

On définit nos différents types comme chaînes de bits de longueurs données, qui seront dorénavant définies dans un module de paramètres de sécurité. Leur seule propriété est que ces chaînes sont de longueur non nulle. La fonction ϵ dépend ainsi de ces paramètres.

On peut maintenant définir l'agent d'observation. Il est modélisé sous la forme d'un triplet dépendant de la taille du retour du circuit, de sa fonction, et de sa preuve montrant que le résultat de l'agent d'observation ne dépend pas de l'algorithme utilisé, mais uniquement des résultats de la fonction passée en paramètre.

```
Definition Pcircuit n := (nonce → nonce → f_out) →
  Alice_randoms_gen → Bob_randoms_gen → bitstring_n n.

Record circuit_gen:= mkCirc {
circ_size:nat;
```



Chaque fichier est représenté par une case dans le diagramme. En entête de cette case, on donne le nom du fichier. Dans la partie basse, on donne la modélisation contenue par le fichier : un système d'oracles (O_{mac}), de contexte (C_{encr}), ou une preuve de bisimilarité ($O_{mac} \sim O_{mac^{id}}$) ainsi que les preuves attenantes. Les fichiers sont représentés suivant les dépendances : les plus bas sont indépendants, alors que les plus hauts dépendent des fichiers auquels ils sont reliés.

FIGURE 6.3 – Architecture des fichiers de la preuve Coq.

```
circf:>Pcircuit circ_size;
circ_ext : forall f f', ( forall x y, f x y = f' x y)
 → forall Arnd Brnd, circf f Arnd Brnd = circf f' Arnd Brnd
}.
```

On remarque que `circuit_gen` définit une chaîne de bits de longueur arbitraire, choisie par l'utilisateur. On retrouve deux types inductifs, `Alice_state` et `Bob_state`, qui prennent en paramètre le nonce reçu au cours de la première partie du protocole.

```
Inductive Alice_state_gen :=
  Alice_start : nonce → Alice_state_gen
| Alice_middle : nonce → Alice_state_gen
| Alice_complete : nonce → Alice_state_gen.

Inductive Bob_state_gen :=
  Bob_start : nonce → Bob_state_gen
| Bob_middle : nonce → Bob_state_gen
| Bob_complete : nonce → Bob_state_gen.
```

On définit également les types des fonctions de MAC, ansi que celles de chiffrement. La clef publique est définie comme fonction de la clef privée. Notez que l'on ne peut chiffrer que des clefs de session, mais comme le protocole ne considère que ce type de chiffrés, on peut ici effectuer cette approximation. Une façon de généraliser cette modélisation aurait été de définir un type clair, et de considérer comme chiffrable tout type de taille inférieure (en rajoutant des bits nuls pour atteindre la longueur voulue) [1].

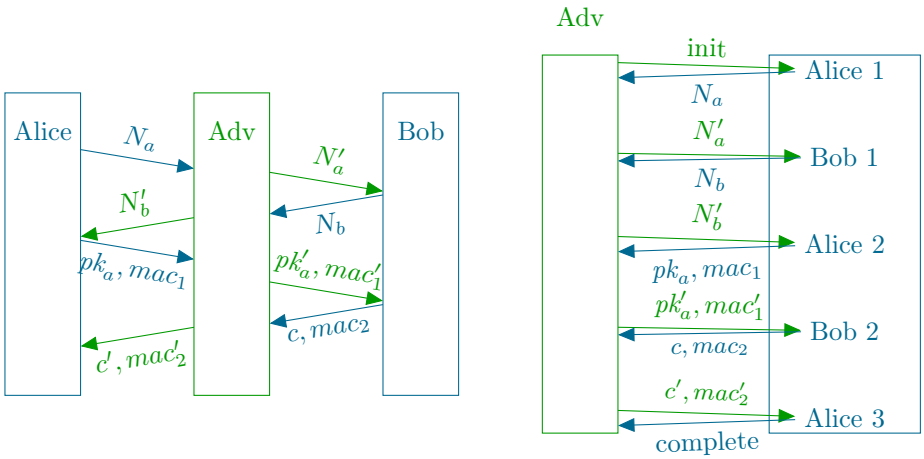
```
Variable Rand: distr (nonce → nonce → f_out).

Variable MAC_create: forall obj_size, hash → bitstring_n obj_size → distr MAC.
Variable MAC_verify: forall obj_size, hash → bitstring_n obj_size → MAC → bool.

Variable crypt: key → session_key → distr cipher.
Variable decrypt: key → cipher → session_key.
Variable generate_key : distr key.
Variable pk : key → key.
```

On demande aussi les preuves que les primitives cryptographiques utilisées soient correctes (un MAC légitime) et totales (en particulier on n'accepte pas de sous-distribution).

6.3 Modélisation du protocole

Pour modéliser un protocole, il faut d'abord passer de la vision classique des cryptographes d'un échange de messages en présence d'un adversaire à celle d'un système d'oracles

1. Ce que l'adversaire peut faire lui-même.



La figure de gauche représente le protocole où l'adversaire est le réseau, en tant qu'échange de messages entre parties. La figure de droite représente le protocole en tant que système d'oracles interagissant avec un adversaire.

FIGURE 6.4 – Du protocole au système d'oracles

Nom de l'oracle	Type du paramètre	Type du résultat
`reset−session`	`circuit`	`sigT bitstring_n`
`Alice_nonce_sender`	`Datatypes.unit`	`nonce`
`Bob_nonce_sender`	`nonce`	`nonce`
`Alice_MAC_key_sender`	`nonce`	`key* MAC`
`Bob_sessionkey_sender`	`key*MAC`	`cipher*MAC`
`Alice_session_key_receiver`	`cipher*MAC`	`bool`

(`sigT bitstring_n` correspond à une chaîne de caractère de taille arbitraire.)

TABLE 6.3 – Signature du système d'oracles représentant le protocole Dziembowski.

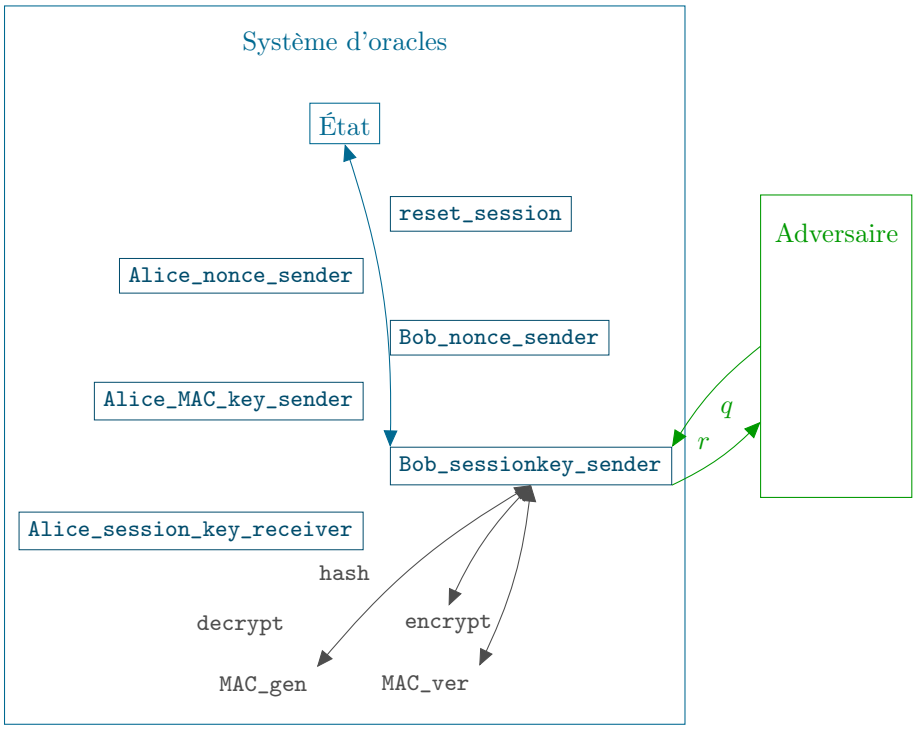
interrogé par un adversaire. Ces deux visions différentes sont décrites par la figure 6.4, elles montrent comment on traduit en termes d'oracles chaque action du protocole. La première figure représente les deux parties du protocole qui s'échangent des messages via un adversaire (pour représenter le fait que l'adversaire a tout pouvoir sur le réseau). La seconde représente un adversaire interagissant avec un système d'oracles.

La première chose à modéliser est la signature des fonctions (qui sera partagée avec d'autres systèmes), la mémoire du système d'oracles, que l'on appelle désormais état du système, et enfin les oracles.

La modélisation du système d'oracle en Coq est donnée par la figure 6.5.

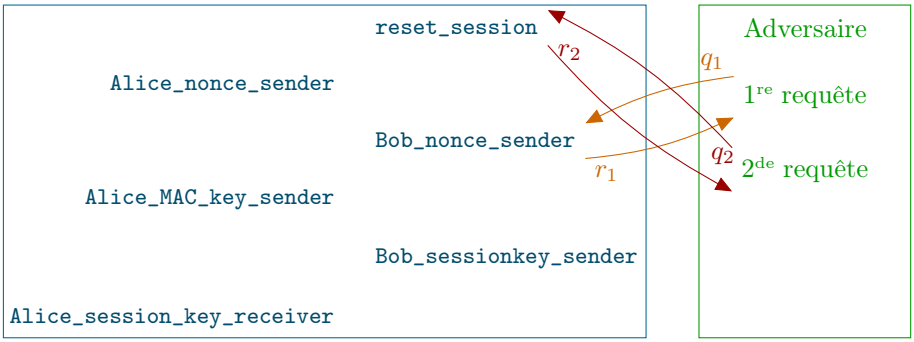
Nous définissons donc dans un premier temps les noms, puis les signatures des oracles du système. Cette signature est décrite dans le tableau 6.3.

Les oracles prennent également en paramètre l'état du système, et retournent cet état (en plus de leur résultat), possiblement modifié, à la fin de leur exécution. Nous noterons par `return r` le résultat `r` renvoyé par la fonction et `save s` les variables modifiées `s` du nouvel état par l'oracle (dans le cas où l'oracle ne modifie pas l'état, on ne notera tout

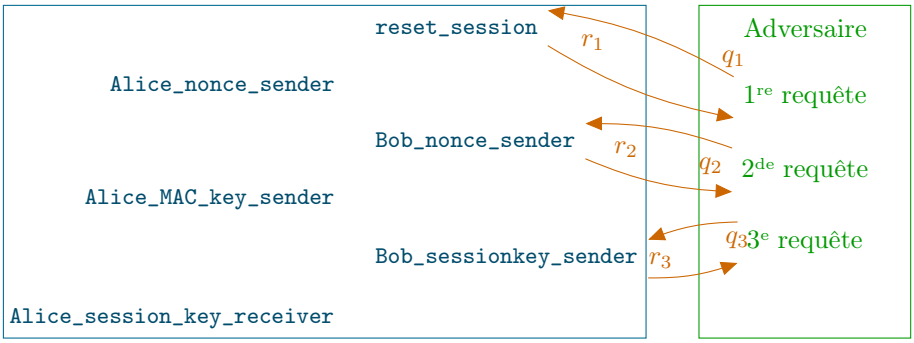


Les oracles accessibles par l'adversaire sont représentés en bleu, les routines utilisées par les oracles en gris. Le système donne accès à un état commun à ses oracles et routines. Cet état peut servir de canal caché permettant aux oracles de communiquer à l'insu de l'adversaire. (Propriété uniquement utilisée pour modéliser la version idéalisée du protocole).

FIGURE 6.5 – Modélisation du système d'oracles représentant le protocole Dziembowski



Cette session est rejetée car la réinitialisation de la session (effectuée par la requête en rouge foncé) n'est pas le premier appel effectué.



La session acceptée : même si les oracles d'Alice n'ont pas été appelés, du point de vue de Bob la session est valide, et la chronologie des échanges est respectée.

FIGURE 6.6 – Exemple de sessions acceptées ou rejetées par le système d'oracle.

simplement pas l'opération `save`).

L'état du sytème (`State`) est un `record` (c'est-à-dire une structure de données rassemblant plusieurs variables pointées par des noms, possiblement interdépendantes) des différentes variables du système.

Il contient entre autres :

La clef secrète à long terme ici modélisée par une fonction qui à deux nonces associe une chaîne de bits (`long_term_key : nonce → nonce → f_out`).

Les variables de l'avancement des deux parties au cours du protocole

(`Aflags : Alice_state`; `Bflags : Bob_state`),

ainsi qu'une montrant si l'initialisation a été faite (`init : bool`). Les exemples de sessions acceptées ou rejetées sont montrées figure 6.6.

La modélisation du système d'oracle en Coq est donnée par la figure 6.5.

Quatre fonctions d'archivage une pour la fonction de hachage

(`ListH : f_out → option(hash)`),

une pour la fonction de chiffrement

(`List_decrypt : cipher → key → option session_key`),

une pour la fonction d'authentification des messages

(`ListMac : (sigT ( fun s ⇒ (bitstring_n s))%type) → hash → bool`)

(`sigT ( fun s ⇒ (bitstring_n s))%type` correspond à une chaîne de caractère de taille arbitraire `s`.) et la dernière pour la fonction f

(`List_f : nonce → nonce → f_out`).

Une variable de sécurité qui contient la quantité d'information récupérée par l'adversaire lors de l'exécution du protocole (`retreive_size: nat`).

Deux variables de session pour les paramètres de session des deux parties : nonce et clefs (`Arnds : Alice_randoms`; `Brnds : Bob_randoms`).

Deux variables pour les clefs d'authentification de chaque partie

(`Sa : hash`; `Sb : hash`).

On définit en tant que fonctions oracles "cachées" (c'est à dire hors de la signature et donc hors de portée de l'adversaire) les primitives cryptographiques utilisées : le chiffrement, le déchiffrement, la génération et la vérification de code d'authentification, ainsi que le hachage de valeurs. Le fait de les définir en tant que fonctions oracles nous permet d'éviter la répétition de code, de modifier automatiquement l'état du système et d'en maintenir la cohérence. Ces modélisations sont données dans la section 4.4.3 pour les fonctions de chiffrement et déchiffrement, dans la sec.4.4.4 pour les fonctions de code d'authentification et de vérification, et dans la sec. 4.4.1 pour l'oracle aléatoire.

Une fois ces oracles définis, on peut s'attaquer au reste des fonctions. A chaque début de session, on appelle l'oracle `o_reset_session`, qui remet à jour, pour la session, nonces, clefs et autres valeurs éphémères. Les fonctions d'archivage ne sont pas remises à jour, dans l'éventualité où des clefs soient réutilisées. Les valeurs qui ne sont pas déterminées à cet instant sont égales à `nul_string`. D'autre part, cet oracle vérifie que l'adversaire n'a pas atteint la limite de la quantité d'information que l'agent d'observation peut retrouver. Dans le cas contraire, il renvoie une chaîne de bits nuls comme étant le résultat de l'agent d'observation.

```
Definition o_reset_session : oracle_fun State
  (Sp.input_type Sp.reset_session) (Sp.output_type Sp.reset_session):=
fun agent, state ⇒
let  (r, K, _,LH,As,Bs,Ar,Br, Sa, Sb, Ok, Lenc, Lm) := state in
mlet na :=new_string nonce_size in
mlet nb := (new_string nonce_size) in
mlet nsk := (new_string key_size) in
mlet nki := (new_string session_key_size) in
let Arnd := (mkArnd na nsk) in
let Brnd :=  (mkBrnd nb nki) in
let circ := (circf (agent)) in
let (out, r_res) :=
```

```
    if (leb 2 (r + (circ_size (agent))) max_retrieve_size) then
15      ((circ K Arnd Brnd), r + (circ_size (agent)))
    else (nul_string, r) in
    !( return out,
18  (save r_res K true LH (Alice_start nul_string)
        (Bob_start nul_string) Arnd Brnd nul_string nul_string
        true Lenc Lm))
21 .
```

On retrouve à la ligne 14 la verification de l'oracle que l'adversaire ne dépasse pas sa taille maximum : `circ_size` est la propriété contenant la taille du résultat de l'agent d'observation envoyé *cette session* aux machines honnêtes (cette taille pouvant changer à chaque session, suivant le choix de l'adversaire). Il s'agit dans la section 4.1.2 de γ. La taille de la mémoire de l'adversaire est bornée ici par `max_retrieve_size`, qui représente également la taille de l'information que l'agent d'observation peut renvoyer en une fois [3] et qui limite également, de fait, le nombre de sessions corrompues. `nonce_size` est égal à β, et `session_key_size` correspond à δ. (K ayant été fusionnée avec f, α n'est plus pris en considération.) Le nombre de sessions n'est pas borné, aussi χ n'est plus pris en considération.

D'autres fonctions utiles sont celles qui changent une valeur de l'état. Elles sont utilisées de temps en temps, suivant la fréquence de modification du champ en question ou la lisibilité du code obtenu. Par exemple, la fonction suivante change la valeur de la variable `init` dans l'état :

```
Definition change_init (init:bool) (s:State) : State :=
 save (init, i)
3 .
```

La fonction d'oracle `o_Alice_nonce_sender` code la première étape du protocole : Alice lance l'interaction en envoyant son nonce, après avoir vérifié que la réinitialisation avait bien été effectuée (grâce aux valeurs de `Aflags` et `init`). On s'assure que l'oracle ne peut être rappelé en modifiant la valeur de `init`. De manière générale, si l'un des tests échoue, on renvoie `nul_string` sans modifier l'état.

```
Definition o_Alice_nonce_sender : oracle_fun State
  (Sp.input_type Sp.Alice_nonce_sender) (Sp.output_type Sp.Alice_nonce_sender):=
3 fun (_, state) ⇒
 match Aflags state, init state with
  |Alice_start _, true ⇒ !(return Arnd_nonce (Arnds state),
6                          save (init, false))
  |_,             _    ⇒ !return nul_string
 end.
```

2. Test d'infériorité entre deux entiers.
3. L'adversaire peut faire le choix de n'utiliser qu'une seule fois un agent d'observation qui renverrait le maximum d'information possible.

Ensuite, `o_Bob_nonce_sender` code l'étape suivante du protocole : Bob reçoit un nonce, calcule la clef d'authentification et retourne son propre nonce (en modifiant l'état de l'oracle en conséquence). L'utilisation de `o_H` force l'utilisation de l'état modifié, que l'on met à jour. On change l'état de Bob et on sauvegarde la valeur de la clef d'authentification.

```
Definition o_Bob_nonce_sender : oracle_fun State
  (Sp.input_type Sp.Bob_nonce_sender) (Sp.output_type Sp.Bob_nonce_sender) :=
fun (na, state) ⇒ match Bflags state with
  | Bob_start _ ⇒
     let n := Brnd_nonce (Brnds state) in
     let s := (long_term_key state na n) in
     mlet (hash, state2) := o_H (s,state) in
     let (r, K, i, LH, As, _, Ar, Br, Sa, _, Ok, Le, Lm) := state2 in
         !( return n,
           save ((Bstate, (Bob_middle na)), (Sb, hash)))
  | _ ⇒ ! return nul_string
end.
```

L'étape suivante est la réception du nonce de Bob par Alice et l'envoi de sa clef publique authentifiée. On vérifie l'état d'Alice, puis on calcule la clef d'authentification. On génère ensuite le code d'authentification de la clef (avec la fonction d'oracle) en modifiant l'état d'Alice avant de retourner la clef et le code.

```
Definition o_Alice_MAC_key_sender : oracle_fun State
  (Sp.input_type Sp.Alice_MAC_key_sender)
  (Sp.output_type Sp.Alice_MAC_key_sender) :=
fun (nb, state) ⇒ match Aflags (state) with
 | Alice_start _ ⇒
     let n := Arnd_nonce (Arnds (state)) in
     let s := (long_term_key (state) n nb) in
     mlet (hash, state2) := o_H (s,state) in
     let k:= Arnd_skey (state2) in
     mlet (mac, state3) :=
       o_gen_mac_sa (pk k, state2)
     in
        !( return (pk k, mac),
          save ((Astate, Alice_middle nb),(Sa, hash)))
 | _ ⇒ !return (nul_string,nul_string)
end.
```

Bob reçoit la clef publique d'Alice (après vérification de son état) et vérifie le code d'authentification à l'aide de la fonction d'oracle. Ensuite il chiffre la clef de session à l'aide de la clef publique, qui génère le code d'authentification du message. Il retourne ensuite le message et le code, ainsi que l'état modifié du système.

```
Definition o_Bob_sessionkey_sender : oracle_fun State
  (Sp.input_type Sp.Bob_sessionkey_sender)
  (Sp.output_type Sp.Bob_sessionkey_sender) :=
fun (pka, mac, state) ⇒ match Bflags (state) with
 | Bob_middle n ⇒
    let s := (Sb (state)) in
    mlet (ver, state2) := o_ver_mac_sb (mac, pka, state) in
       if ver then
         let ki := Brnd_session_key (Brnds (state2)) in
         mlet (crypt_ki, state3) := o_encr( pka, ki, state2) in
         mlet (mac, state4) :=
          o_gen_mac_sb (crypt_ki, state3)
         in !( return (crypt_ki, mac),
              save (Bstate, Bob_complete n))
       else !return (nul_string,nul_string)
 | _ ⇒    !return (nul_string,nul_string)
end.
```

Enfin, Alice reçoit le chiffre, vérifie son authenticité, et retourne la validité qu'elle accorde à la clef reçue. Comme l'utilisation de la clef ne nous intéresse pas, on se repose entièrement sur la vérification du code.

```
Definition o_Alice_session_key_receiver : oracle_fun State
  (Sp.input_type Sp.Alice_session_key_rec)
  (Sp.output_type Sp.Alice_session_key_rec) :=
fun ((c,m),  state) ⇒ match Aflags (state) with
  | Alice_middle n ⇒
       mlet (ver, state2) := o_ver_mac_sa (m, c, state) in
     !( return ver, save (Astate, Alice_complete n))
  | _ ⇒ !return false
end.
```

Maintenant que les fonctions d'oracles sont définies, il reste à définir le protocole, puis à prouver sa totalité avant de pouvoir le définir en tant que cadre.

On renvoie donc chaque nom d'oracle de la signature à la fonction correspondante, et on définit l'état initial du protocole.

```
Definition Protocol : oracle_implementation Sp.oracle_signature State:=
mkOI
(fun name : Adversary.oracle_name Sp.oracle_signature ⇒
  match name with
 | Sp.Alice_nonce_sender ⇒ o_Alice_nonce_sender
 | Sp.Alice_MAC_key_sender ⇒ o_Alice_MAC_key_sender
 | Sp.Alice_session_key_receiver ⇒ o_Alice_session_key_receiver
 | Sp.Bob_nonce_sender ⇒ o_Bob_nonce_sender
```

```
 | Sp.Bob_sessionkey_sender ⇒ o_Bob_sessionkey_sender
 | Sp.reset_session ⇒ o_reset_session
end)
  (mlet K := Rand in ! mks 0 K false (fun _:f_out ⇒ None )
     (Alice_complete full_string) (Bob_complete full_string)
     (mkArnd nul_string nul_string) (mkBrnd nul_string nul_string)
     nul_string nul_string true (fun n c ⇒ None) (fun n m ⇒ false))
.
```

On ne montre ici qu'une preuve de totalité, car le principe est simple, il suffit de dérouler l'oracle et de montrer que dans tous les cas, l'oracle termine. On utilise la tactique `solve_totality`. Elle retrouve la forme du but de la preuve, applique le lemme de totalité pour une commande simple (`Munit`, `Mlet` et `newstring`) ou résoud la totalité pour une expression conditionnelle, puis s'appelle récursivement dans chaque cas du branchement. Dans les autres cas, elle rend la main à l'utilisateur.

```
Ltac solve_totality := match goal with
  | |- @total _ (! _)
          ⇒ apply total_Munit
  | |- @total _ (Mlet _ _)
          ⇒ apply total_Mlet; [solve_totality|intro;solve_totality]
  | |- @total _ (if _ then _ else _)
          ⇒ apply total_if_expr; [solve_totality|solve_totality]
  | |- @total _ (new_string _)
          ⇒ apply new_string_total
  | |- _  ⇒ idtac
end.

Lemma total_o_H : forall msg st, total (o_H (msg,st)).
intros msg st.
unfold o_H.
destruct st;simpl;solve_totality.
destruct (ListH0 msg) as [h| ];solve_totality.
Qed.
```

Avec les démonstrations de la totalité de tous les oracles du système, on peut alors définir le cadre d'exécution du système qui dépend du protocole que l'on vient de définir.

```
Definition Frotocol : frame Sp.oracle_signature State.
```

6.4 Modélisation du protocole idéalisé

Le protocole idéal est quasiment identique au protocole originel. Les noms et signatures des oracles sont les mêmes. L'état ne diffère qu'avec le champ `t`, présent pour informer si

la session est une session test, plus généralement si un agent d'observation a été envoyé ou non au système. On sauvegarde également la valeur de la clef d'authentification (calculée par Bob) avant hachage, pour pouvoir tester si des attaques ont été tentées. De façon générale, on sauvegarde toutes les extensions de clefs calculées avant hachage, par souci de cohérence pour répondre aux requêtes.

Nous gardons donc en mémoire, en particulier, les valeurs suivantes :

Une variable test déterminant si la session en cours est une session test (`t : bool`).

Quatre fonctions d'archivage une pour la fonction de hachage

(`ListH : f_out → option(hash)`),

une pour la fonction de chiffrement

(`List_decrypt : cipher → key → option session_key`),

une pour la fonction d'authentification des messages

(`ListMac : (sigT ( fun s ⇒ (bitstring_n s))%type) → hash → bool`)

et la dernière pour la fonction f

(`List_f : nonce → nonce → f_out`).

Une variable pour la clef d'authentification non hachée de Bob (`Sb : f_out`).

On redéfinit les fonctions comme f, le chiffrement et la génération de code d'authentification pour que, si la session est une session test, les fonctions idéales soient utilisées. Les modélisations sont données dans les sections 4.4.2, 4.4.3 et 4.4.4.

La fonction de hachage n'est pas modifiée, étant elle-même idéale. La fonction de remise à niveau entre deux sessions n'a changé que pour mettre à jour la valeur de session test : si la longueur de retour de l'agent d'observation est nulle, alors on considère qu'aucun agent d'observation n'a été envoyé au système.

La fonction d'initialisation de session tire au hasard les paramètres frais du protocole, teste si l'agent d'observation est actif ou non, et si oui calcule le résultat de cet agent sur les paramètres aléatoires, avant de sauvegarder l'état modifié.

```
Definition o_reset_session : oracle_fun State
  (Sp.input_type Sp.reset_session) (Sp.output_type Sp.reset_session):=
  fun (agent, state) ⇒
    let r:= retrieve_size state in
    let K:= long_term_key state in
    mlet na :=new_string nonce_size in
    mlet nb := (new_string nonce_size) in
    mlet nsk := (new_string key_size) in
    mlet nki := (new_string session_key_size) in
    let Arnd := (mkArnd na nsk) in
    let Brnd :=  (mkBrnd nb nki) in
    let circ := (circf agent) in
 if (leb (r + (circ_size agent)) max_retrieve_size) then
     !( return (circ K Arnd Brnd),
```

```
      save ((r + (circ_size agent)) (beq_nat [4] 0 (circ_size agent))
        K true (ListH state) (Alice_start nul_string) (Bob_start nul_string)
        Arnd Brnd nul_string nul_string true (List_decrypt state)
        (ListMac state) (List_f state)))
 else
  !( return nul_string,
     save (r true K true (ListH state) (Alice_start nul_string)
        (Bob_start nul_string) Arnd Brnd nul_string nul_string true
        (List_decrypt state) (ListMac state) (List_f state)))
.
```

Pour la fonction `o_Bob_nonce_sender`, la différence se fait si la session en cours est une session test, auquel cas on ne calcule pas la clef d'authentification à partir des nonces échangés, mais on tire au sort une chaine de bits que l'on hache. Cette chaine de bits intermédiaire est ensuite gardée dans l'état.

```
Definition o_Bob_nonce_sender : oracle_fun State
   (Sp.input_type Sp.Bob_nonce_sender) (Sp.output_type Sp.Bob_nonce_sender) :=
fun (na, state) ⇒
match Bstate state, t state with
 | Bob_start _, true ⇒
    let n := Brnd_nonce Br in
    mlet f := new_string f_output_size in
    mlet (sbb, state2) := (o_H (f, state)) in
    !( return n, save ((Bob_state, (Bob_middle (na)), (Sb, sbb))
 | Bob_start _, false ⇒ [5]
    let n := Brnd_nonce Br in
    let sb :=K na n in
    mlet (sbb, state2) := (o_H (sb, state)) in
    !( return n, save (Bstate, (Bob_middle state2)), (Sb, sbb))
 | _, _ ⇒ !(return nul_string)
end.
```

Les autres fonctions (`o_Alice_nonce_sender`, `o_Alice_MAC_key_sender`...) sont les mêmes, mis à part qu'elles utilisent les versions idéalisées des routines cryptographiques, données dans la section 4.4.

6.5 Exemple de la modélisation d'un contexte de la preuve

Dans cette section, nous présentons la transformation du système d'oracle du protocole hybride mélant la première partie du protocole idéalisée (modélisée dans le contexte) avec

4. Test d'égalité entre deux entiers.
5. Ce cas est quasi-identique a la version originale

la seconde partie originelle (modélisée par le sous-système d'oracles). Le principal défi de cette dissection en un système de contexte combiné à un sous-système d'oracles est la gestion de la clef d'authentification. Faut-il la calculer dans le contexte et la faire parvenir comme paramètre au sous-système d'oracles ? Ou faut-il simplement vérifier que l'échange de nonces s'est passé correctement et donner cette information au sous-système ?

6.5.1 Principe du contexte

L'écriture d'un contexte repose sur l'analyse du système initial : quelle est la propriété à simuler, quel est le nouveau jeu de sécurité et comment articuler le contexte et le sous-système d'oracle ? Ici, on cherche à simuler la première partie du protocole, c'est à dire l'échange de nonces.

Cette simulation correspond à une preuve par réduction : on construit, à partir d'un adversaire arbitraire $\mathbb{A}$ contre un système d'oracle original (ici $\mathbb{O}_{\pi_0^{id}\pi_1}$), un adversaire contre une partie plus spécifique du système (ici $\mathbb{O}_{\pi_1}$). Le contexte exécute les calculs du reste du système (ici $\mathbb{C}_{\pi_0^{id}}$), il est un simulateur. Si l'on compose le contexte avec l'adversaire, on obtient un nouvel adversaire $\mathbb{A} \parallel \mathbb{C}_{\pi_0^{id}}$ contre $\mathbb{O}_{\pi_1}$.

Si il existe un adversaire $\mathbb{A}$ efficace contre $\mathbb{O}_{\pi_0^{id}\pi_1}$, alors il existe un adversaire $\mathbb{A} \parallel \mathbb{C}_{\pi_0^{id}}$ tout aussi efficace contre $\mathbb{O}_{\pi_1}$. Par contraposé, si $\mathbb{O}_{\pi_1}$ est sûr (c'est à dire s'il n'existe pas d'adversaire efficace contre $\mathbb{O}_{\pi_1}$), alors $\mathbb{O}_{\pi_0^{id}\pi_1}$ est sûr.

La sécurité remonte donc de l'analyse et des propriétés de sécurité de sous-systèmes d'oracles (des petits blocs, dont la sécurité est plus facile à analyser), pour remonter jusqu'à la sécurité du système global (plus gros et donc plus complexe à analyser). La bisimulation entre la composition du contexte composé au sous-système (ici $\mathbb{C}_{\pi_0^{id}}[\mathbb{O}_{\pi_1}]$) et du système originel (ici $\mathbb{O}_{\pi_0^{id}\pi_1}$) est garante d'une construction saine.

On peut également utiliser les contextes pour isoler une primitive (par exemple un système de chiffrement).

6.5.2 Structure du contexte

La structure du contexte se décide avant tout en distinguant ce que l'on veut simuler de ce que l'on ne veut pas. On répartit les calculs généralement en fonction de leur ordre dans le protocole, suivant une limite dans le temps. On peut encore décider d'isoler tous les appels à une primitive (chiffrement, etc.) et de garder le reste des calculs dans le contexte.

Un Choix crucial

On a tout d'abord essayé de modéliser un contexte vérifiant que l'échange de nonces et plus généralement le calcul de la clef n'était pas perturbé par l'adversaire. Ainsi, le contexte vérifie que tout se passe bien lors de l'échange des nonces, puis passe la main au sous-système pour tirer la clef d'authentification dans une distribution uniforme. Cette vision des choses a deux implications directes :

— la génération de la clef utilisée se fait dans le sous-système,

— les distributions de messages sont indiscernables, certes, mais non identiques, car les clefs d'authentification ne sont plus calculées mais tirées d'une distribution.

L'interaction avec l'adversaire se déroulerait ainsi : le contexte prendrait d'abord en charge l'échange de nonces. Ensuite, deux cas se présentent :

la session est une session test, le contexte renverrait alors un booléen indiquant si l'adversaire a modifié les nonces au cours de l'échange au sous-système,

la session n'est pas une session test, et le contexte calculerait les deux clefs et enverrait au sous-système la comparaison entre ces clefs.

Le sous-système prend la suite des calculs et des échanges (le contexte ne servant plus que d'intermédiaire). Si le booléen est vrai, le contexte tire une seule clef pour Alice et Bob, sinon il tire deux clefs indépendantes (dans la distribution des clefs d'authentification).

Dans le cas du modèle ROM, les distributions des clefs calculées et des clefs générées par les deux modèles sont identiques, mais elles posent de gros problèmes dans la preuve Coq de bisimulation qui suit : il faut alors gérer les cas particuliers des égalités des nonces et des calculs, et pouvoir les comparer au système original. Ces égalités impliquent une complexification importante de la relation entre états.

Comme dans le cas d'une session normale la clef est hachée, l'introduction d'une condition de validité (correspondant à la probabilité où les deux clefs calculées sont identiques même si l'adversaire a interféré dans l'échange de nonce) qui renvoie à l'exclusion du cas où la fonction de hachage aurait produit le même hachage pour deux entrées différentes. De la même façon, il faut prendre en compte la probabilité d'une collision de la fonction f dans la condition de validité.

Cette solution relache donc le majorant final en incluant la probabilité de trouver une collision pour la fonction de hachage et pour la fonction f, et en cela, elle n'est déjà plus idéale. En effet, dans les deux systèmes il existe des possibilités de collisions, mais cela ne peut aider à les distinguer l'un de l'autre, étant donné que ces probabilités sont les mêmes de part et d'autre (on utilise bien les mêmes fonctions de hachage et d'expansion de clef -f- dans les deux systèmes).

Devant ces inconvénients non négligeables, la solution suivante fut adoptée. Le contexte prend en charge l'échange des nonces et la génération de la clef, avant de passer cette clef au sous-système et de ne plus rien faire d'autre que de transférer les messages pendant la seconde partie du protocole. L'avantage de cette solution est que les distributions restent identiques, et que le jeu de sécurité du sous-système qui s'en dégage est très clair : il s'agit pour l'adversaire de pouvoir distinguer si l'on utilise les primitives idéales ou classiques en ayant choisi la clef d'authentification. L'adversaire doit donc réussir une attaque sur les schémas de chiffrements et d'authentification. En adoptant cette solution, le majorant final reste resserré au regard de la propriété de sécurité.

Après ce choix, le découpage se fait naturellement. La signature du contexte est identique à celle du système d'oracles original, et la signature du sous-système d'oracles regroupe uniquement les oracles intervenant lors de la seconde partie du protocole, et

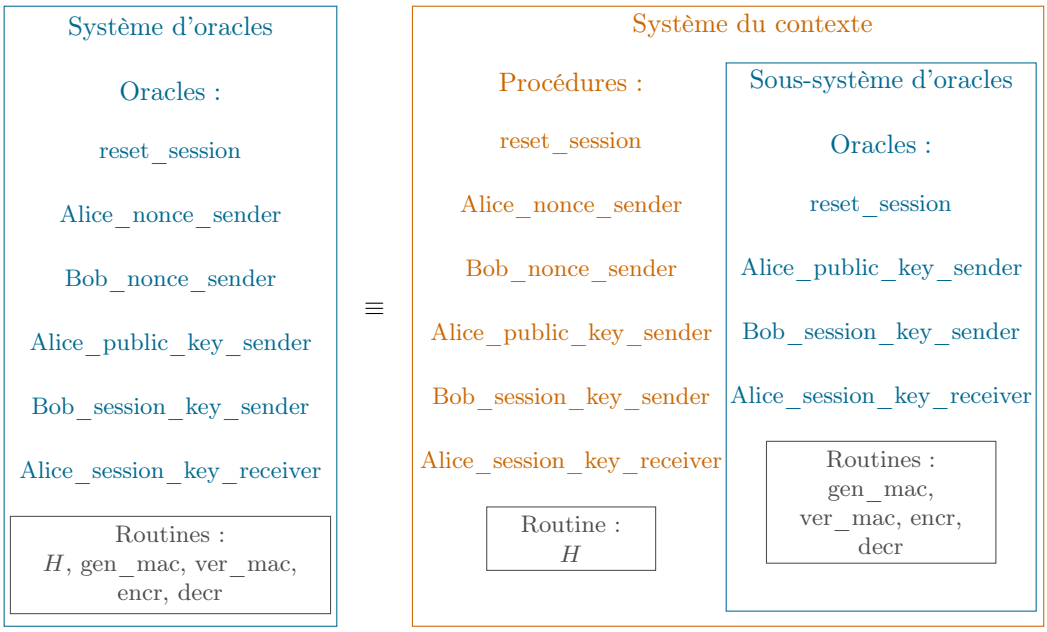


FIGURE 6.7 – Dissection d'un système d'oracles en une composition d'un contexte et d'un sous-système d'oracles.

l'oracle d'initialisation de session. Les routines sont aussi réparties entre contexte et système d'oracles suivant le même procédé. La figure 6.7 représente la dissection du système d'oracles en un sous-système d'oracles combiné à un contexte.

Un autre oracle valant d'être noté est l'oracle d'initialisation, car il gère également l'agent d'observation. La solution choisie ici pour la modélisation est que le contexte fournit à l'oracle toutes les données aléatoires de l'état dont il a la charge. L'oracle calcule ensuite le résultat de l'agent d'observation sur toutes les données aléatoires de la combinaison du contexte et de l'oracle, le renvoie au contexte qui le transfère à l'adversaire. La signature de l'oracle d'initialisation est donc changée, on lui ajoute en paramètres les types des objets dont l'agent d'observation a besoin pour être évalué. Par exemple ici, l'oracle `reset_session` du sous-système $\mathbb{O}_{\pi_1}$ recevra en plus de ses paramètres habituels toutes les variables aléatoire du contexte $\mathbb{C}_{\pi_0}$, c'est à dire les deux nonces n_a, n_b.

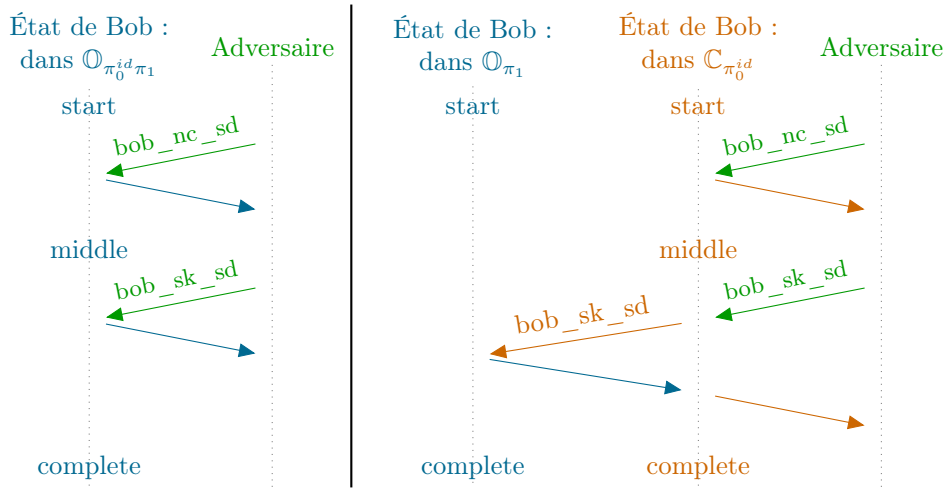
6.5.3 Modélisation des systèmes

La structure du sous-système d'oracles suit celle du contexte car elle vient le compléter. En cela, tout ce qui n'est pas simulé par le contexte doit être exécuté dans le système d'oracles. Le seul réel changement est la signature d'`Alice_public_key_sender`, car c'est le seul oracle à cheval entre les deux parties du protocole. Les signatures des oracles du systèmes sont données dans la table 6.4.

Concernant l'état, toutes les variables de l'état utilisées par le contexte font partie de l'état du contexte, et toutes les variables utilisées par le sous-système font partie de l'état du sous-système. Certaines variables, comme les variables contenant l'état d'Alice et de

Nom de l'oracle	Type du paramètre	Type du résultat
`reset_session`	`circuit*nonce*nonce`	`sigT bitstring_n`
`Alice_MAC_key_sender`	`hash`	`key*MAC`
`Bob_sessionkey_sender`	`key*MAC`	`cipher*MAC`
`Alice_session_key_receiver`	`cipher*MAC`	`bool`

TABLE 6.4 – Signature du sous-système d'oracles représentant la seconde partie du protocole.



Sont représentées les requêtes aux oracles ainsi que leur réponse, étiquetées de l'abréviation du nom de l'oracle concerné (bob_nc_sd pour `Bob_nonce_sender`, bob_sk_sd pour `Bob_session_key_sender`). [6]

FIGURE 6.8 – Changements de la variable d'état de Bob au cours d'une exécution honnête.

Bob, sont conservées en double, car elles ont une utilité et dans le contexte, et dans le sous-système.

Les états de ces variables dédoublées sont dépendants. Toutes les combinaisons d'états dédoublés ne sont pas possibles : si l'état de la variable du contexte est le même que celui du système, et qu'il est sensé être modifié plusieurs fois dont une fois par le contexte seulement, celui du sous-système "saute" généralement un état. Ici, comme la première partie du protocole est exécutée par le contexte, la variable indiquant l'état de Bob dans le sous-système passe directement de `Bob_start` à `Bob_complete`. La figure 6.8 représente les changements de la variable d'état de Bob au cours d'une exécution honnête de l'interaction avec l'adversaire du point de vue de Bob. C'est à dire que l'adversaire interagit avec les oracles de Bob dans cet ordre : `Bob_nonce_sender`, puis `Bob_session_key_sender`.

Ces états sont décrits dans la table 6.5 de la section 6.6, où nous discutons plus en détail de la relation liant ces états entre eux.

L'état est réduit pour ne garder plus que les informations pertinentes pour effectuer les calculs de la première partie du protocole.

On montre dans le code suivant la fonction du contexte la plus affectée par cette

dissection, `Alice_mackey_sender`. La fonction `ctx_pi0_U` intègre le comportement adéquat suivant que la session soit une session test ou non dans le calcul de la clef d'authentification juste avant le hachage de celle-ci.

```
Definition ctx_pi0_Alice_mackey_sender
   (x: (Sp.input_type Sp.Alice_MAC_key_sender*Complement_pi_0)):
      Adversary S1.oracle_signature
        (Sp.output_type Sp.Alice_MAC_key_sender*Complement_pi_0) :=
match (i, cstate) with
  | (_, t false ltk (Alice_start _) Bs Na Nb lh Ok sa sb _ lf)⇒
      clet (u, cstate2) := ctx_pi0_U((Na, i), cstate in
      clet (sa', cstate3) := ctx_pi0_H (u, cstate2) in
           clet outp :=
             Ccall S1.oracle_signature S1.Alice_MAC_key_sender sa'
           in
           !!( return outp,
              save ((Astate, (Alice_middle i)), (Ok,
                   (andb Ok (bitstring_eqb nonce_size i Nb))), (Sa, sa'))
  | e ⇒ !!return (nul_string, nul_string)
end.
```

Il est lisible sur le code que la clef est calculée avant tout appel à la fonction du sous-oracle effectuée à la ligne 10. On remarquera que l'opérateur `Ccall` (qui permet à l'appel d'un oracle) prend en paramètre la signature de l'oracle appelé, et les paramètres à fournir à l'oracle. La réponse de l'oracle est directement transférée à l'adversaire. Le mot-clef `clet` et l'opérateur !! permettent respectivement d'attribuer une valeur à une variable et de retourner une valeur au sein de la monade du contexte. Le reste des opérandes et mots-clefs étant les mêmes que pour la définition d'oracles (notamment pour la définition de sa signature).

Preuves

Une fois la définition du contexte écrite, il faut apporter la preuve qu'il peut se composer avec un système d'oracle ou un adversaire. pour cela, il faut prouver la totalité du système de contexte, définir sa matrice d'appel (combien d'appels à chaque oracle du sous-système ont été faits par le contexte) et montrer la linéarité de cette matrice avec le nombre d'appels faits à chaque fonction du contexte, afin de pouvoir calculer le temps d'exécution de l'adversaire, et de connaître le nombre d'appels aux oracles effectués.

Le lemme montrant que la séquence est totale est celui-ci :

```
Lemma total_seq_pi0 : forall (comp: Complement_pi_0)
     (o : oracle_name Sp.oracle_signature)
     (i : oracle_input Sp.oracle_signature o),
 total_comp (ctx_pi0 o (i, comp)).
```

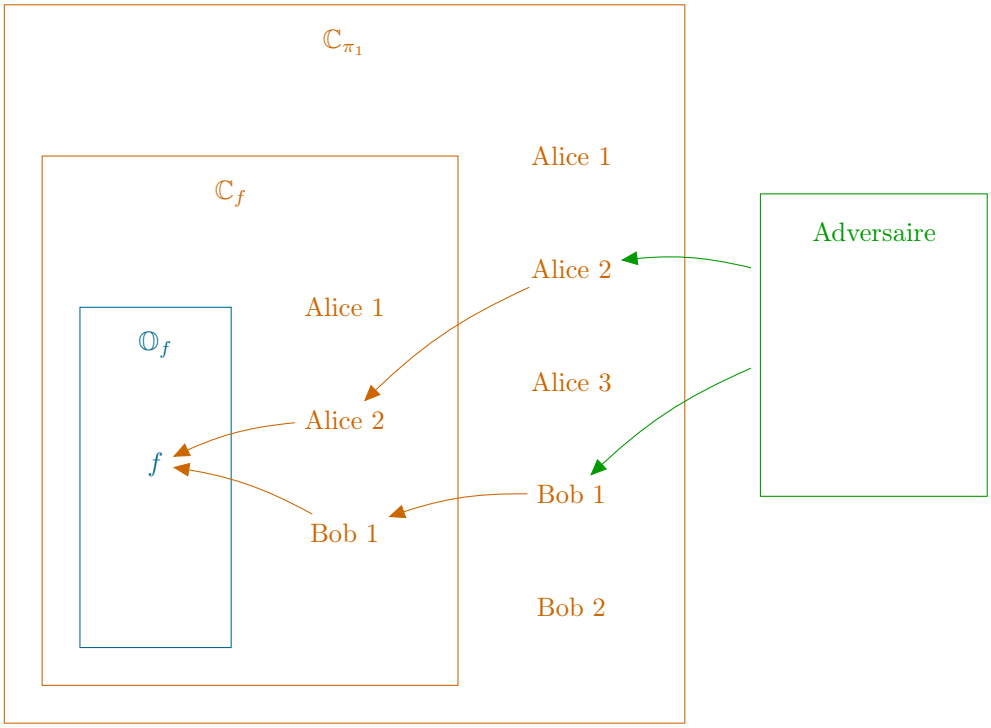
Pour chaque oracle, peu importe les paramètres qui lui sont passés et l'état dans lequel le système se trouve, il doit finir son calcul.

La preuve de totalité du contexte repose essentiellement sur cette tactique :

```
Ltac solve_comp_totality := match goal with
| |- @total_comp ?os _ ([<?who;?what>]) ⇒ apply (@total_Ccall os who what)
| |-  @total_comp _ _ (!! _) ⇒ apply total_Cunit
| |- @total_comp _ _ (?? _) ⇒ apply total_Cdraw;solve_totality
| |- @total_comp _ _ (Clet _ _) ⇒ apply total_Clet;
      [solve_comp_totality
      |intro;solve_comp_totality]
| |- @total_comp _ _ (if _ then _ else _) ⇒ apply total_comp_if_expr;
      [solve_comp_totality
      |solve_comp_totality]
| |- @total _ _  ⇒ solve_totality
| |- _ ⇒ idtac (*default case*)
end.
```

Elle permet d'automatiser, dans le plus grand nombre de cas, la preuve de totalité de l'opération en cours. Par une étude de cas, la tactique gère dans l'ordre :

- l'appel à un oracle (ligne 2), en appelant le lemme qui prouve la totalité de l'appel si l'oracle appelé est lui-même total ;
- le renvoi de valeur (ligne 3), en utilisant le lemme montrant que cette opération est bien totale ;
- le tirage de valeur au sein d'une distribution, en utilisant le lemme montrant que cette opération est bien totale, et en rappelant la tactique (ligne 4) ;
- l'assignation de valeur à une variable, en appliquant d'abord le lemme de la totalité de l'opération (ligne 5), qui produit deux sous-buts :
  - la totalité de l'opération menant à la valeur que doit prendre la variable (on rappelle la tactique pour résoudre ce but - ligne 6),
  - la totalité du reste des opérations, lorsque l'on remplace la variable par sa valeur (on introduit une variable pour remplacer cette valeur, puis on rappelle la tactique pour résoudre ce but - ligne 7) ;
- la totalité des expressions de type si alors sinon, en appliquant d'abord le lemme de la totalité de l'opération (ligne 8), qui produit deux sous-buts dans lesquels on rappelle la tactique pour montrer la totalité des calculs dans le cas où la condition est vraie, et de même lorsque la condition est fausse ;
- la totalité d'une distribution, on est alors dans le déploiement d'un oracle où on appelle la tactique spécifique `solve_totality` (ligne 11) ;
- le cas par défaut où l'on redonne la main à l'utilisateur.



Par souci de clarté, seuls les appels sont représentés. Pour trouver le nombre de fois où l'oracle f (du système d'oracles $\mathbb{O}_f$) est appelé, il faut remonter à la liste des fonctions de contexte qui font appel à lui. Dans le contexte $\mathbb{C}_{\pi_0}$, les fonctions Alice 2 et Bob 1 lancent chacun un appel à f. Pour obtenir un majorant pour l'adversaire, il faut encore remonter les appels au contexte $\mathbb{C}_{\pi_1}$.

FIGURE 6.9 – Calcul d'un majorant d'appels aux oracles

Il reste à l'utilisateur à dérouler le protocole et à détailler chaque cas. En particulier, il faut vérifier la totalité pour chaque valeur possible des expressions évaluées dans les branchements.

Le calcul d'un majorant d'appels aux oracles est montré dans la figure 6.9. C'est une illustration de la matrice d'appel. La matrice des appels de fonction définie avec le système de contexte permet de calculer le nombre d'appels aux oracles. Elle permet de passer du nombre d'appels de $\mathbb{A}$ contre $\mathbb{C}[\mathbb{O}]$ au nombre d'appels de $\mathbb{C} \parallel \mathbb{A}$ contre $\mathbb{O}$.

Construire la matrice d'appels et montrer sa linéarité se fait de façon naturelle, nous ne nous y attarderons donc pas.

Le lemme montrant que le nombre d'appels à l'oracle n'a pas été dépassé est celui-ci :

```
Lemma c_pi_0_id_runtime:
forall (who : oracle_name Sp.oracle_signature)
  (what : oracle_input Sp.oracle_signature who) (sc : Complement_pi_0),
at_most_calls (ctx_pi0 who (what, sc))
  (mkMx S1.oracle_signature Sp.oracle_signature M_pi0 zero_M_pi0
     lin_M_pi0 eg_M_pi0 (unit_count Sp.oracle_signature who)).
```

Sa preuve repose encore une fois essentiellement sur une tactique :

```
Ltac solve_runtime :=
match goal with
| |- @at_most_calls _ _ ([<_;_>]) _ ⇒
  solve [solve_ccall]
| |- @at_most_calls _ _ (!! _) _ ⇒
  apply Cunit_runtime_all
| |- @at_most_calls _ _ (?? _) _⇒
  apply Cdraw_runtime_all
| |- @at_most_calls _ _ (Clet ([<_;_>]) _) _ ⇒
  apply Clet_X_nullY;[solve_runtime|intro;solve_runtime]
| |- @at_most_calls _ _ (Clet _ _ ) _ ⇒
  apply Clet_nullX_Y;[solve_runtime|intro;solve_runtime]
| |- @at_most_calls _ _ (if _ then _ else _) _ ⇒
  apply if_expr_runtime;
   [solve_runtime|solve_runtime]
| |- _ ⇒ idtac(*default case*)
end.
```

On retrouve là encore les mêmes cas que dans la tactique précédente, soit :

- l'appel à un oracle (ligne 4), en appelant la tactique qui permet de compter l'appel ;
- le renvoi de valeur (ligne 6), en utilisant le lemme montrant que cette opération ne change pas le nombre d'appels ;
- le tirage de valeur au sein d'une distribution,en utilisant le lemme montrant que cette opération ne change pas le nombre d'appels (ligne 8) ;
- l'assignation de valeur à une variable,
  - en supposant d'abord qu'un appel à un oracle a lieu au cours de l'assignation mais aucun par la suite, qui est résolu par un lemme dédié (ligne 9),
  - en admettant ensuite qu'aucun appel n'a lieu au moment de l'assignation, mais seulement dans la suite des calculs, où on applique un second lemme (ligne 11) ;
- un branchement conditionnel, où on vérifie dans les deux cas que le nombre d'appels aux oracles est bien en dessous du majorant (ligne 14) ;
- le cas par défaut où l'on redonne la main à l'utilisateur (ligne 16).

De même que précédemment, reste à la charge de l'utilisateur de dérouler le protocole et les appels, ainsi que de décomposer tous les cas des branchements. Il lui restera également à gérer les cas non triviaux que sont les appels à des oracles dans l'assignation d'une variable et le reste des calculs. Il est cependant rare qu'un contexte fasse appel à plusieurs oracles, la conception papier de la logique excluant même explicitement ce cas.

On peut maintenant définir le système du contexte, prêt à être appliqué à un sous-système d'oracles :

```
Definition c_pi_0_id : frame_ctx S1.oracle_signature Sp.oracle_signature :=
mk_simple_frame_ctx C_pi0_distr_init total_C_pi0_init ctx_pi0
```

```
total_seq_pi0 Mx_pi_0 c_pi_0_id_runtime.
```

On obtient ainsi un cadre d'exécution pour un contexte entre la signature principale et la signature de la seconde partie du protocole.

6.6 Exemple de la modélisation d'une bisimulation de la preuve

L'utilité des bisimulations, dans cette preuve, réside dans la preuve que les différentes réécritures du système sont équivalentes les unes aux autres. A chaque réécriture du système en une composition d'un contexte et d'un sous-système, nous devons montrer une bisimulation entre les deux systèmes pour justifier d'un raisonnement correct.

Dans cette section, nous étudions la bisimulation entre $\mathbb{O}_{\pi_0^{id}\pi_1}$ et $\mathbb{C}_{\pi_0^{id}}[\mathbb{O}_{\pi_1}]$. Nous illustrons les similitudes de traces entre les deux systèmes modélisant le protocole hybride avec la fig. 6.10. La figure montre le système d'oracles à gauche, et à droite la combinaison du contexte modélisant la première partie du protocole avec un système d'oracles modélisant la seconde.

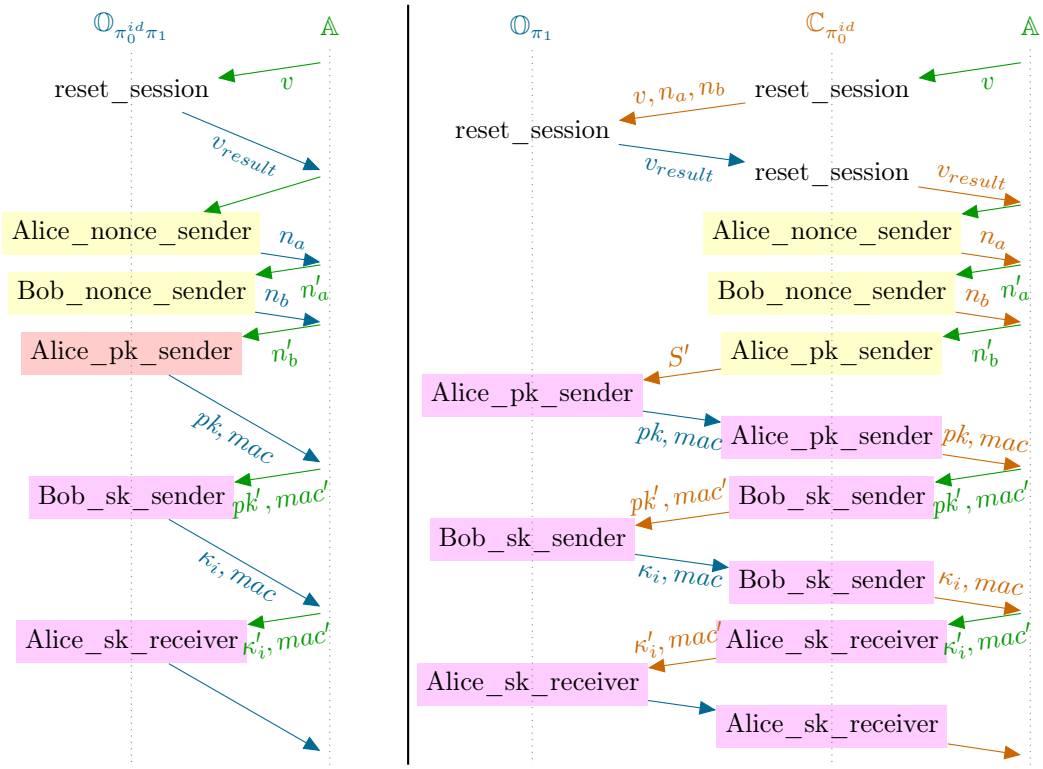
On remarque vite que les signatures des deux systèmes sont identiques. De plus, les distributions de messages échangés entre l'adversaire et les systèmes sont de même nature. On notera également que le système d'oracles n'encapsule que la seconde partie du protocole. Pour l'oracle Alice_pk_sender, qui est mixte car il finalise la première partie du protocole (en recevant le nonce de Bob et en calculant la clef d'authentification) et initie la seconde partie (en renvoyant sa clef publique authentifiée), l'oracle du système $\mathbb{O}_{\pi_1}$ récupère de l'oracle du contexte $\mathbb{C}_{\pi_0^{id}}$ la clef d'authentification utilisée pour générer et vérifier les codes d'authentification.

Nous cherchons à montrer ce lemme :

```
Lemma pi_bisim_cpi0_pi1_to_pi:
 forall bnd:oracle_name Sp.oracle_signature→ nat,
Frame_pi0id_pi1
∼(bnd;0)
App_ctx c_pi_0_id (Protocol_pi1).
```

C'est une bisimulation parfaite : quelque soit le nombre d'appels aux systèmes d'oracles faits par l'adversaire, il ne pourra jamais différencier les deux systèmes.

Une bisimulation repose en premier lieu sur une relation entre états. Cette relation doit être respectée de la distribution initiale des états à leur distribution finale, et entre chaque appel à un oracle. Ici, cette relation est l'égalité entre deux états projetés (sur un état de référence), avec quelques relations en plus permettant de gérer les relations des états d'Alice et de Bob dans le contexte et le sous-système d'oracles. Dans un second temps, si nécessaire, il faut écrire une condition nécéssaire à la bisimulation (plus simplement vérifier qu'aucun évènement dans la trace ne vienne gêner la bisimulation : une requête



Légende : oracle d'initialisation, oracle de π_0 , oracle de π_1 , oracle mixte (à cheval sur π_0 et π_1) , message envoyé par un oracle, un contexte, un adversaire.

FIGURE 6.10 – Comparaison de deux traces de deux systèmes bisimilaires.

inattendue de l'adversaire...). Ici, aucune condition n'est nécessaire, juste une relation entre les états, car la bisimulation est parfaite : ce sont exactement les mêmes opérations qui sont effectuées dans les deux systèmes.

6.6.1 Relations entre états

L'état de référence est l'état du système global de $\mathbb{O}_{\pi_o^{id}\pi_1}$. On construit donc une projection des états combinés de $\mathbb{C}_{\pi_0^{id}}$ et $\mathbb{O}_{\pi_1}$ dans cet état de référence afin de pouvoir les comparer plus facilement dans la relation de la bisimulation. La table 6.5 représente les deux différents états ainsi que la projection des états de la composition contexte/oracle dans un état correspondant au système global.

Les choix dans la projection sont faciles à faire : lorsque le membre concerne la première partie du protocole (comme la fonction d'expansion de clef `Longterm_key`), on garde le membre de l'état du contexte. Dans le cas inverse (par exemple la clef d'authentification `Sa'2`), on garde le membre de l'état du sous-système d'oracles. Dans les cas hybrides (par exemple l'état d'avancement de Bob `Bs` et `Bflag2`), on garde le membre de l'état du contexte. La projection de l'état de $\mathbb{C}_{\pi_0^{id}}[\mathbb{O}_{\pi_1}]$ et l'état de $\mathbb{O}_{\pi_o^{id}\pi_1}$ doivent être égaux, c'est le premier

État de $\mathbb{O}_{\pi_o^{id}\pi_1}$		État de $\mathbb{C}_{\pi_0^{id}}$		État de $\mathbb{O}_{\pi_1}$	
nom	type	nom	type	nom	type
r	nat	r	nat	r	nat
long_term_key	type_f	Longterm_key	type_f		
t	bool	test	bool	t_encr	bool
				t_mac	bool
init	bool	init	bool		
ListH	list_hash	Lh	list_hash		
AstatePi	Alice_state	As	Alice_state	Aflag2	Alice_state
BstatePi	Bob_state	Bs	Bob_state	Bflag2	Bob_state
Arnds	Alice_random	Na	nonce	Ak2	key
Brnds	Bob_random	Nb	nonce	Bk2	session_key
Sa'	hash	sa	hash	Sa'2	hash
Sb'	hash	sb	hash	Sb'2	hash
OkSoFar	bool	Ok	bool	Ok2	bool
List_decrypt	list_cipher			List_decrypt2	list_cipher
ListMac	list_mac			List_Mac2	list_mac
lf	list_f	Lf	list_f		

Légende : les membres des états qui font partie de la projection de la composition contexte/oracle vers l'état du système d'oracles complet. Les booléens font l'objet d'une conjonction (sauf pour le marqueur de la session test qui ne concerne que la première partie du protocole et donc uniquement le contexte). On note par type_f le type $nonce \to nonce \to f\_out$.

TABLE 6.5 – Les états de $\mathbb{O}_{\pi_o^{id}\pi_1}$ et $\mathbb{C}_{\pi_0^{id}}(\mathbb{O}_{\pi_1})$

élément de la relation entre les deux états.

Une fois cette projection effectuée, on définit la relation par l'égalité entre ces deux états. Elle est complétée par toutes les dépendances entre les états du contexte et de son sous-système, principalement par l'état d'avancement de Bob, comme on peut le voir sur la relation ci-après :

```
Record R (s : O_Oidpi1_m.State )
    (s' : C_pi_O_id_m.PI1.State_pi1_id * C_pi_O_id_m.Complement_pi_O):
  Prop :=
    mkR_pi {
      R_state : (proj_c_piO_pi1_to_pi s') = s;
      R_As : forall n,
        (Same_alice_state (O_Oidpi1_m.AstatePi s)
                          (C_pi_O_id_m.PI1.Aflag2 (fst s')));
      R_Bs_start :forall n,
        (Same_bob_state (O_Oidpi1_m.BstatePi s) (Bob_start n)) →
          (Same_bob_state (C_pi_O_id_m.PI1.Bflag2 (fst s')) (Bob_start n));
      R_Bs_mid : forall n,
        (Same_bob_state (O_Oidpi1_m.BstatePi s) (Bob_middle n)) →
          (Same_bob_state (C_pi_O_id_m.PI1.Bflag2 (fst s')) (Bob_start n));
```

```
15      R_Bs_comp : forall n,
          (Same_bob_state (O_Oidpi1_m.BstatePi s) (Bob_complete n)) →
            (Same_bob_state (C_pi_O_id_m.PI1.Bflag2 (fst s')) (Bob_complete n))
18  }.
```

Ces états d'avancement ont une relation entres eux, comme l'explicite la figure 6.5. Ainsi, si ces variables ont le même état dans le contexte et dans le système d'oracles, la variable du sous système ne passe pas par la phase "middle". Ce n'est pas le cas pour la variable d'avancement d'Alice, car sa valeur étant changée par `Alice_MAC_key_sender` et `Alice_session_key_receiver`, sa valeur dans $\mathbb{O}_{\pi_1}$ est identique à celles de $\mathbb{C}_{\pi_0^{id}}$ et $\mathbb{O}_{\pi_o^{id}\pi_1}$. L'oracle `Alice_nonce_sender` change uniquement la valeur du booléen d'initialisation, qui fait uniquement partie du contexte.

L'avantage de définir la relation la plus resserrée possible est de rendre la preuve plus facile par la suite : se servir de la condition de relation pour montrer que les différents branchements conditionnels se comportent de la même façon dans un système et dans l'autre. Il est donc crucial que cette relation soit à la fois la plus simple et la plus complète possible.

6.6.2 Le Maintien de la relation à chaque appel d'oracle

Le cœur de cette preuve est de montrer le maintien, à chaque étape de l'interaction entre l'adversaire et le système, de la relation. C'est à dire qu'à chaque requête (identique pour les deux systèmes), d'une part les deux réponses suivront la même distribution, et d'autre part si leurs états sont modifiés, ils restent en relation l'un avec l'autre.

Nous avons donc prouvé, pour chaque oracle, le maintien de cette relation. Cette fois, la seule tactique écrite simplifie le contexte, en particulier réécrit les fonctions `Clet`, `Ccall`, `Cdraw` et `Cunit` en termes d'un système d'oracles. Le principe de la preuve est de dérouler l'exécution des deux systèmes et de montrer que les égalités aux branchements et au retour sont induites par la relation.

La tactique `simpl_context` permet donc de transformer la composition du contexte et du sous-système en un système d'oracles des lignes 2 à 5, puis la simplification lorsque de mêmes opérations avaient lieu dans chaque système des lignes 6 à 9 et enfin la simplification lors des appels aux oracles à la ligne 10.

```
Ltac simpl_context := repeat (
   rewrite @interact_Clet
3  || rewrite @interact_Cdraw
   || rewrite @interact_Cunit
   || rewrite @interact_Ccall
6  || rewrite @Mlet_assoc
   || rewrite @Mlet_simpl
   || rewrite @Munit_simpl
9  || (apply @mu_eq_compat; [reflexivity|let x := fresh in intro x])
   || (match goal with |- context C[@interact ?os ?ST ?A (Ccall _ ?who ?what) _ _ _]
```

```
    ⇒ rewrite (@interact_Ccall os who what _ _ _ _); unfold oracle_call end)
|| (progress autored)).
```

Les preuves sont donc en général longues et fastidieuses, mais les seuls vrais écueils furent d'une part la preuve de la bisimilarité de `Alice_MAC_key_sender` lors du premier choix de tirer la clef dans le sous-système, qui s'est soldé par une preuve complexe et un majorant insuffisamment resserré.

Le second écueil est l'ordre des tirages des variables de session au cours de l'initialisation de session (avec l'oracle `reset_session`). En effet, dans $\mathbb{O}_{\pi_o^{id}\pi_1}$ les deux nonces sont tirés avant les deux clefs, alors que dans $\mathbb{C}_{\pi_0^{id}}[\mathbb{O}_{\pi_1}]$ le contexte effectue ses tirages en premier, puis le sous-système : les clefs sont ainsi tirées avant les deux nonces.

Il y a deux façons de résoudre cet écueil.

— La première aurait été de mettre en place une autre stratégie pour gérer l'agent d'observation : faire s'évaluer partiellement l'agent d'observation dans le sous-système au cours de l'appel et retourner cette fonction au contexte, qui effectue ensuite ses tirages et finit l'évaluation de l'agent d'observation. L'inconvénient étant, à ce stade, de devoir reprendre toutes les modélisations et preuves pour respecter cette stratégie de façon globale. L'autre inconvénient est que le jeu de sécurité contre $\mathbb{O}_{\pi_1}$ perd de sa lisibilité : le sous-système ne renvoie qu'une évaluation partielle. Même si cela n'a pas d'influence sur la validité de la preuve finale, ce manque de lisibilité peut perturber un usager humain.

— La seconde (notre choix) est de montrer que l'ordre des tirages (et des assignations occasionnées) n'a pas d'importance, du moment que la donnée tirée est discrète. Cette solution a l'avantage de ne pas modifier la modélisation existante, et d'apporter un lemme général sur les distributions pouvant être réutilisé dans un autre contexte.

Ce lemme fait déjà partie de la bibliothèque ALEA, et est formulé ainsi :

```
Lemma is_discrete_swap: forall A B C (d1:distr A) (d2:distr B) (f:A → B → distr C),
  is_discrete d1 →
  (mlet x := d1 in mlet y :=d2 in f x y ) == (mlet y:=d2 in mlet x:=d1 in f x y).
```

Si un tirage est fait dans une distribution discrète, alors on peut effectuer ce tirage plus tard, sans impacter sur la distribution finale. La preuve se base essentiellement sur les propriétés de `munit` (décrites dans la section 5.2.2), et le fait de pouvoir effectuer un tirage dans la distribution discrète indépendamment du reste, jusqu'à son utilisation.

En utilisant cette tactique et ce lemme, nous somme en mesure de démontrer que la relation se maintient bien à chaque appel, quelques soient les paramètres :

```
Lemma R_comp : forall st comp who what ev ev',
  R st comp →
   (forall ost ocomp outp,
    R ost ocomp → ev (outp, ost ) == ev' (outp, ocomp)) →
      mu (o_funs Frame_pi0id_pi1 who (what, st)) ev ==
```

```
6       mu (o_funs (App_ctx c_pi_0_id (Frotocol_pi1)) who (what, comp)) ev'.
```

Et la preuve du lemme de bisimulation découle directement de la règle `Rule_simple_or` appliquée à cette relation, une fois que l'on a montré que les deux distributions initiales étaient elles aussi en relation l'une avec l'autre.

Cette bisimulation est extrêmement forte : il n'y a aucune condition de validité, et leurs distributions sont identiques. Ces deux systèmes sont parfaitement bisimilaires, c'est en cela que la composition d'un contexte et d'un sous-système d'oracles est une réécriture du système d'oracles. Toutes les bisimulations faites au cours de ces travaux sont parfaites, car elles concernent toutes des réécritures d'un système d'oracles en une composition d'un contexte avec un sous-système.

6.6.3 Application de règles

L'énoncé du théorème est simple : le cadre d'exécution du protocole de Dziembowski doit être indiscernable à `eps` près du cadre d'exécution de la version idéalisée du protocole, dans la limite `bnd` du nombre d'appels aux oracles.

La fonction de probabilité maximale elle n'est pas modifiée par les contextes, on utilisera donc directement la somme des `eps` définis dans les paramètres de sécurité.

```
Definition eps := Uplus eps_f (Uplus eps_mac eps_encr).

Theorem dziembowski_secure :
Pi.Frotocol (*pi*)
≈(bnd; eps)
Pi_id.Frotocol. (*pi_id*)
```

La preuve déroule l'arbre présenté dans la figure 6.2. On utilise donc les règles de transitivité pour insérer les cadres d'exécution des systèmes intermédiaires. On utilise ensuite les lemmes d'indiscernabilité prouvés dans les autres modules. Tout au long de cette section, chaque fois que nous utilisons le mot arbre, nous faisons référence à l'arbre de preuve de la fig. 6.2 pour situer l'application d'une règle dans son contexte.

```
rewrite ← Uplus_zero_left. eapply Rule_trans.
instantiate (1:=Frame_Cpi1_pi0).
apply pi_bisim.
```

On commence par se réduire au système $\mathbb{C}_{\pi_1}[\mathbb{C}_f[\mathbb{O}_{f\oplus f}]]$ dont on a prouvé la bisimulation avec $\mathbb{O}_\pi$ dans `pi_bisim`.

```
unfold eps. eapply Rule_trans.
instantiate (1:=App_ctx c_pi_1 (App_ctx c_f Frame_U)).
unfold Frame_Cpi1_pi0. apply Explicit_sub. apply Explicit_sub.
apply indis_f_U.
```

Ensuite on se focalise sur la première partie du protocole, pour montrer l'indiscernabilité avec $\mathbb{C}_{\pi_1}[\mathbb{C}_f[\mathbb{O}_U]]$. En déroulant `eps`, on a conservé la composante `eps_f` dans ce sous but. On utilise donc la règle de sous-routine (`Explicit_sub`) deux fois pour éliminer les contextes, et enfin appliquer l'hypothèse de sécurité de f. La règle de transitivité (l. 1) correspond au nœud 4 de l'arbre. Celles des éliminations de contextes (l. 3) correspondent aux nœuds 2, 3.

```
rewrite ← Uplus_zero_left. eapply Rule_trans.
instantiate (1:=Frame_pi0id_pi1).
apply pi_bisim_cpi1_pi0_to_pi.

rewrite ← Uplus_zero_left. eapply Rule_trans.
instantiate (1:= App_ctx c_pi_0_id (Frotocol_pi1)).
apply pi_bisim_cpi0_pi1_to_pi.
```

On bascule sur le protocole mixte en insérant d'abord $\mathbb{O}_{\pi_0^{id}\pi_1}$, puis appliquant la bisimulation avec $\mathbb{C}_{\pi_1}[\mathbb{C}_f[\mathbb{O}_U]]$. On introduit alors par transitivité $\mathbb{C}_{\pi_0^{id}}[\mathbb{O}_{\pi_1}]$ (en appliquant la bisimulation dans la foulée), ce qui permet de concentrer la preuve sur la seconde partie du protocole. Les règles de transitivité (l. 1, 5) font référence aux nœuds 5, 13 de l'arbre.

```
apply Rule_sym. rewrite ← Uplus_zero_left. eapply Rule_trans.
instantiate (1:=Frame_Cpi0id_pi1). apply PI_id.pi_bisim.
apply Explicit_sub.
```

On introduit le système $\mathbb{C}_{\pi_0^{id}}[\mathbb{C}_{encr}[\mathbb{O}_{mac}]]$ pour pouvoir réduire la preuve aux propriétés du chiffrement et de l'authentification en éliminant le contexte de la première partie. La règle de transitivité (l. 1) correspond au nœud 12 de l'arbre. Celle de l'élimination de contexte (l. 3) correspond au nœud 11.

```
eapply Rule_trans. instantiate (1:= App_ctx c_encr Frame_mac_id).
apply Explicit_sub.
apply indis_mac_macid.
```

On se concentre maintenant sur la dernière branche de la preuve. On introduit $\mathbb{C}_{encr}[\mathbb{O}_{mac}]$ pour pouvoir éliminer le contexte du chiffrement, puis on résoud le sous-but avec l'hypothèse de sécurité de l'authentification. La règle de transitivité (l. 1) correspond au nœud 9 de l'arbre de preuve. Celle de l'élimination de contexte (l. 2) correspond au nœud 8.

```
rewrite← Uplus_zero_right. eapply Rule_trans.
instantiate (1:=(App_ctx c_mac_id Frame_encr_id)).
rewrite← Uplus_zero_left. eapply Rule_trans.
instantiate(1:= App_ctx c_mac_id Frame_encr).
apply pi1_id_bisim.
apply Explicit_sub.
apply indis_encr_encrid.
```

```
Qed.
```

Pour finir, on introduit les systèmes $\mathbb{C}_{mac\ id}[\mathbb{O}_{encr^{id}}]$ et $\mathbb{C}_{mac\ id}[\mathbb{O}_{encr}]$, en appliquant une dernière bisimulation, et la dernière hypothèse de sécurité concernant le chiffrement (après avoir simplifié le contexte de l'authentification). Les règles de transitivité (l. 1, 3) font référence aux nœuds 10, 9 de l'arbre. Celle de l'élimination de contexte (l. 6) correspondent aux nœuds 6.

6.7 Conclusion

Au cours de ce chapitre, nous avons exposé les modélisations clefs de cette preuve, ainsi que les principales difficultés rencontrées. Nous sommes maintenant à même de pouvoir prévenir ce type d'erreur, ainsi que de pouvoir souligner les étapes pouvant être automatisées. Nous avons par ailleurs obtenu une preuve complète et complexe, s'articulant autour d'une dizaine de fichiers.

Chapitre 7

Conclusion

Pour communiquer de façon sécurisée, l'Homme crée des systèmes cryptographiques complexes. De nombreuses applications critiques reposent sur ces systèmes : il est maintenant possible de payer à distance, de faire des opérations boursières, de voter, ou encore de sécuriser des échanges d'informations militaires pendant des manœuvres. Il est aujourd'hui capital de pouvoir avoir confiance en ces systèmes d'information sécurisés. C'est dans ce cadre que nous prouvons formellement la sécurité d'un protocole cryptographique dans un modèle d'attaque innovant.

Les systèmes de sécurité, de par leur utilisation dans des situations critiques, méritent une approche formelle pour leur vérification. Afin de raisonner sur les systèmes cryptographiques, nous nous sommes reposés sur les méthodes formelles, développées pour raisonner sur tous les types de systèmes d'information. Nous nous sommes intéressés au problème de la modélisation et de la vérification de propriété de sécurité en présence d'agents d'observation. Ainsi, au cours de ces travaux, nous montrons commment utiliser une logique spécialisée, CIL, afin de prendre en compte ces intrusions.

CIL est basée sur le modèle calculatoire. Ce modèle considère que tout objet (message ou mémoire des parties) est une chaîne de bits et autorise à l'adversaire toutes les opérations efficaces, c'est à dire s'effectuant en temps polynomial (dans la taille du paramètre de sécurité). Nous montrons la sécurité d'un système par une preuve que l'adversaire a une chance négligeable de rompre la propriété de sécurité du système. Cette propriété de sécurité est définie comme une indiscernabilité entre le système étudié et un système idéal, n'ayant par définition aucune faille.

Cette logique repose sur les probabilités et les distributions des chaînes de bits échangées : pour prouver la sécurité, nous montrons qu'il est impossible de distinguer le système sur lequel nous raisonnons d'un système idéal (qui ne permet aucune fuite d'information) sur la base des messages émis soit par la primitive, soit par le protocole cryptographique. Nous avons explicité les liens entre CIL et les différentes notions de sécurité calculatoire : concrète et asymptotique en montrant que bien que CIL se base sur la sécurité concrète des schémas cryptographiques, on peut l'utiliser également pour raisonner sur la sécurité

asymptotique.

Nous enrichissons cette logique pour prendre en compte les intrusions passives de l'adversaire sur les machines honnêtes, que nous avons considérées comme un nouveau paramètre (facultatif) passé en argument à l'oracle chargé de la ré-initialisation de session. Ainsi, la logique obtenue demeure générale et peut être utilisée pour des preuves dans d'autres modèles de sécurité.

Pour aller au fond des choses, nous élargissons également la modélisation faite en Coq de cette logique. Pour modéliser les intrusions, nous nous situons au plus bas niveau possible, celui du système d'oracles, afin de conserver la flexibilité de CIL. Ce choix s'avère intéressant et pragmatique car il permet à l'utilisateur une plus grande finesse dans ce qu'il souhaite prouver (en particulier s'il ne travaille pas en présence d'intrusion de l'adversaire). Nous modélisons l'agent d'observation comme une fonction des paramètres aléatoires de la session. Grâce à Coq, nous pouvons traiter cette fonction comme un objet, et le passer en argument à l'oracle de ré-initialisation de session. Ce dernier a pour charge de vérifier la taille des informations retournant à l'adversaire.

Cette amélioration de CIL a pour but d'effectuer des preuves dans le modèle calculatoire à taille de stockage de mémoire bornée, conçu pour raisonner sur les fuites d'informations. En voici les hypothèses de sécurité :

— l'adversaire peut envoyer à chaque session un agent d'observation dans chacune des parties honnêtes,

— cet agent d'observation ne peut renvoyer à l'adversaire qu'une quantité limitée d'information sur l'état de la machine honnête,

— les parties honnêtes partagent une grande mémoire commune aléatoire (suffisamment grande pour que l'agent d'observation ne la renvoie pas à l'adversaire),

— les sessions sont non concurrentes (deux sessions ne peuvent pas avoir lieu en même temps),

— le but de l'adversaire est de compromettre les clefs calculées par les parties honnêtes lors d'une session non corrompue.

Le protocole vérifié repose essentiellement sur deux phases : une phase pour calculer une clef d'authentification intermédiaire et une seconde phase pour échanger la clef de session.

La vérification de ce protocole constitue la mise à l'épreuve de cette logique. Au fil de la présentation de la preuve, nous mettons en avant l'importance des choix de modélisation et de l'architecture de la preuve, très similaires à des choix de développement en génie logiciel et leurs possibles conséquences néfastes sur la difficulté de la preuve et le majorant de sécurité (qui s'ajoutent aux délais supplémentaires du développement de la preuve). Nous soulignons ici qu'une preuve en Coq doit être davantage considérée comme un projet logiciel que comme une preuve mathématique. Nous avons résolu le problème que posait la gestion de l'agent d'observation lors des preuves par réduction, c'est à dire l'utilisation des contextes dans la preuve. En outre, nous avons développé plusieurs tactiques automatisant une grande partie des preuves courantes.

7.1 Discussions

La preuve que nous avons menée à bout repose essentiellement sur la manière de modéliser l'agent d'observation. En le réduisant à ce qui fait sa nature, c'est à dire une fonction arbitraire, il nous semble que seule une modélisation d'ordre supérieur pourrait raisonner de façon satisfaisante avec les intrusions. De ce point de vue, il apparaît que les outils basés sur le modèle Dolev-Yao (modèle symbolique) ne peuvent pas, par leur nature même, fournir des preuves de sécurité concernant des propriétés de résistance aux intrusions dans un avenir proche.

Nous appuyons cette réflexion sur la même différence qui existe entre la logique de premier ordre et la logique d'ordre supérieur. Alors que la première abstrait les symboles (comme le modèle de Dolev-Yao), la seconde est capable de raisonner à propos de leurs propriétés (autres que celles gérées par les relations équationnelles) et de leurs résultats. De fait, il nous semble difficile de parvenir à la preuve qu'un protocole est résistant aux intrusions avec les outils basés sur ce modèle (entre autres provérif [Bla01], avispa [ABB+05], scyther [Cre08], AKISS [CCK12]). Cependant, une approche novatrice des modèles symboliques pourrait peut-être permettre de raisonner sur ces propriétés. Bana et Comon-Lundh [BCL14] proposent non plus de raisonner sur ce que l'adversaire est capable de faire, mais sur ce qu'il *ne peut pas faire*. Si cette approche nous semble prometteuse, il parait compliqué de caractériser un modèle tel que le BSM pour cette approche, en particulier imposer une limite sur la quantité d'information que l'adversaire peut récupérer.

À notre connaissance, aucun autre outil basé sur le modèle calculatoire ne prend en compte les intrusions. Cryptoverif [Bla06] est basé sur le π-calcul. Celui-ci permet en théorie d'envoyer des processus à d'autres processus, et cela peut être utilisé pour modéliser les intrusions. Certicrypt [BGZ09] et easycrypt [BGHB11] sont développés en Coq, ce qui leur permettrait de définir un type dépendant pour représenter les agents d'observation, comme nous l'avons fait. Malheureusement, il ne nous apparaît pas clairement comment ces implantations peuvent être faites dans ces outils, et comment cela impacterait le caractère automatique de leurs preuves.

Par ailleurs, bien que ce ne fût pas l'objet de notre travail, nous nous sommes interrogés sur le réalisme du protocole prouvé : il pose la question de l'échange de la mémoire commune de grande taille. Nous pouvons imaginer que le moyen le plus sûr de procéder à cet échange est de le faire par courrier physique, par disque Blue-ray (ou disque dur externe), ces supports permettant l'envoi de 27 gigaoctet (resp. 1 teraoctet) de données, ce qui nous paraît être une masse suffisante d'information pour s'apercevoir d'une fuite. Bien sûr, dans un tel cas, il faut reporter sa confiance sur le messager effectuant la distribution.

7.2 Travaux futurs

Le première remarque sur cette preuve est le temps de compilation : avec une dizaine d'heures pour les bisimulations les plus longues, il serait opportun de trouver des pistes

pour optimiser les preuves, même si nous pensons que le problème vient de la gestion de Coq des réécritures successives inévitables dans ces preuves.

Ensuite, il serait intéressant d'adapter la méthode décrite ici dans d'autres modèles : les preuves de primitives et de protocoles résistant aux fuites d'information physique (ou attaques par canaux cachés), telles que les attaques par différentiels de potentiels et autres, comme définis par [DP08b, MR04, KP10]. La fonction modélisant l'agent d'observation pourrait alors être détournée de son usage premier pour renvoyer les valeurs perdues lors de l'exécution physique du protocole ou de la primitive.

Pour l'utilisation ultérieure de ces travaux, nous soulignons ici que ce travail a demandé une longue et studieuse formation à Coq, et qu'il semble illusoire que des cryptographes se forment à l'utilisation de CIL en l'état. Néanmoins, il nous paraît plausible de parvenir à un outil relativement complet, permettant de générer un squelette de preuve Coq à partir de l'arbre de preuve, des signatures des systèmes et de la modélisation du système que l'on veut prouver (ainsi que de ses routines).

La conception d'un tel outil nous apparaît d'autant plus accessible que les preuves courantes de totalité, de bisimulation et d'appels aux oracles reposent exclusivement sur le code des oracles. Nous sommes confiants qu'à partir de l'arbre de preuve et d'un pseudo code des systèmes d'oracles, il est possible de générer automatiquement une grande partie de la preuve en Coq. Ce serait pour nous l'axe selon lequel il faudrait poursuivre ce travail de recherche.

Une façon d'aborder ce problème serait de définir un langage de description des oracles et des états, et un second pour définir l'arbre de preuve. On pourrait imaginer de modéliser uniquement le système original, et de réserver un mot clef pour distinguer les fonctions routines des fonctions répondant à la signature du système d'oracles.

Les systèmes de contextes seraient générés à partir d'information annexe : la répartition des oracles et routines entre les systèmes. Les oracles (ainsi que les routines) à cheval entre le contexte et le sous-système devront être réécrits par l'utilisateur, afin d'éviter des choix arbitraires et potentiellement mauvais pour la preuve. Malgré tout, l'outil devrait être en mesure de fournir un squelette pour leur définition contenant au moins leur signature.

Les versions idéales des routines pourraient être générées en remplaçant toutes les distributions devant être aléatoires par des distributions uniformes. Les relations entre les paramètres et la sortie de l'oracle pourront être conservées par des listes de l'état (comme nous l'avons fait dans ces travaux pour les fonctions de chiffrement et de MAC).

Le squelette de la preuve ainsi que l'architecture des fichiers pourraient être générés à partir de l'arbre de preuve. Les preuves courantes de totalité, de compte d'appels aux oracles et de bisimulation pourront s'obtenir en déroulant pas à pas la modélisation des oracles.

Par ailleurs, les erreurs de modélisation évoquées dans ce travail ont été coûteuses en temps (plusieurs mois ont été perdus pour la plus sérieuse d'entre elles, alors qu'en changeant de direction la preuve a été résolue en quelques semaines). Cela s'explique par

le remaniement intégral du code à chaque changement de fonction. Le temps d'uniformiser toutes les répétitions de codes était d'autant plus long que la compilation des bisimulations par Coq était complexe. Ces répétitions ont été le fruit d'un choix qui s'expliquait par la difficulté de mettre en place un état minimal dans la signature, dont hériteraient plusieurs états différents. Il est important d'apporter du temps et de la réflexion à la résolution de ce problème.

Il devrait être possible de trouver une solution sur le modèle des classes abstraites de la programmation orientée objet. Il apparaît donc urgent d'adapter la modélisation de CIL, ou du moins des états des systèmes d'oracles, afin de pouvoir coder les fonctions routines (notamment les primitives cryptographiques utilisées) indépendemment des systèmes d'oracles dans lesquels elles sont utilisées. Tout cela afin de réduire au maximum la répétition de code, et de faciliter les corrections d'erreurs.

Une autre piste, probablement plus facile à mettre en place et s'intégrant dans le développement d'un outil, serait de générer à partir de pseudo-code et des différents états ces fonctions routines. Par la suite, il faudra modifier uniquement ce pseudo-code au cours de la conception de la preuve pour modifier l'ensemble des routines des systèmes.

Bibliographie

[ABB+05] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, et al. The avispa tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification*, pages 281–285. Springer, 2005.

[Abr96] Jean-Raymond Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, 1996.

[ACKR13] Myrto Arapinis, Véronique Cortier, Steve Kremer, and Mark Ryan. Practical everlasting privacy. In *Principles of Security and Trust*, pages 21–40. Springer, 2013.

[APM09] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 74(8) :568–589, 2009.

[BCL14] Gergei Bana and Hubert Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 609–620. ACM, 2014.

[BDKL10] G. Barthe, M. Daubignard, B. Kapron, and Y. Lakhnech. Computational indistinguishability logic. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010.

[BGHB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology–CRYPTO 2011*, pages 71–90. Springer, 2011.

[BGZ09] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of POPL'09*, pages 90–101, 2009.

[Bla01] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[Bla06] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, 2006.

[Cam97] Albert J Camilleri. A hybrid approach to verifying liveness in a symmetric multi-processor. In *Theorem Proving in Higher Order Logics*, pages 49–67. Springer, 1997.

[CCK12] Rohit Chadha, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. In *Programming Languages and Systems*, pages 108–127. Springer, 2012.

[CFC59] Haskell B Curry, Robert Feys, and William Craig. Combinatory logic, volume i. 1959.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and computation*, 76(2) :95–120, 1988.

[Com12] CCRA Management Committee. *Common Criteria for Information Technology Security Evaluation*, version 3.1 revision 4 edition, Septembre 2012.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88*, pages 50–66. Springer, 1990.

[Cre08] Cas JF Cremers. The scyther tool : Verification, falsification, and analysis of security protocols. In *Computer Aided Verification*, pages 414–418. Springer, 2008.

[dB70] Nicolaas Govert de Bruijn. The mathematical language automath, its usage, and some of its extensions. In *Symposium on automatic demonstration*, pages 29–61. Springer, 1970.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on information Theory*, 22(6) :644–654, 1976.

[DLL12] S. Devismes, P. Lafourcade, and M. Lévy. *Informatique théorique : Logique et démonstration automatique, Introduction à la logique propositionnelle et à la logique du premier ordre*. Technosup (Paris). Ellipses, 2012.

[DM04] S. Dziembowski and U. Maurer. Optimal randomizer efficiency in the bounded-storage model. *Journal of Cryptology*, 17(1) :5–26, 2004.

[DP08a] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *IEEE 49th Annual IEEE Symposium on Foundations of Computer Science, 2008. FOCS'08*, pages 293–302, 2008.

[DP08b] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 293–302. IEEE, 2008.

[dt14] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2014. Version 8.4, http ://coq.inria.fr.

[DY83] Danny Dolev and Andrew Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2) :198–208, 1983.

[Dzi06] S. Dziembowski. Intrusion-resilience via the bounded-storage model. *Lecture Notes in Computer Science*, 3876 :207, 2006.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *Information Theory, IEEE Transactions on*, 31(4) :469–472, 1985.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1) :176–210, 1935.

[Gir72] Jean-Yves Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. 1972.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption* 1. *Journal of computer and system sciences*, 28(2) :270–299, 1984.

[Hal05] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. `http://eprint.iacr.org/`.

[IR02] G. Itkis and L. Reyzin. SiBIR : Signer-base intrusion-resilient signatures. *Advances in Cryptology—Crypto 2002*, pages 101–116, 2002.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4 : Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[Koc96] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[KP10] E. Kiltz and K. Pietrzak. Leakage resilient elgamal encryption. In *ASIACRYPT*, pages 595–612, 2010.

[lTJC] Enée le Tacticien. *La Poliorcétique*. IV siècle avant J.C.

[Luo90] Zhaohui Luo. *An extended calculus of constructions*. PhD thesis, University of Edinburgh, 1990.

[Mar04] James Margetson. Proving the completeness theorem within isabelle/hol. 2004.

[Mau92] U.M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *Journal of Cryptology*, 5(1) :53–66, 1992.

[Mog90] Eugenio Moggi. *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1990.

[Mon96] J-F. Monin. *Comprendre les Méthodes formelles, panorama et outils logiques*. CTST. Masson, 1996. Préface de G. Huet.

[Mon00] J-F. Monin. *Introduction aux méthodes formelles*. CTST. Hermès, 2000. Edition revue et augmentée de [Mon96].

[MR04] S. Micali and L. Reyzin. Physically observable cryptography. *Theory of Cryptography*, pages 278–296, 2004.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology—EUROCRYPT'99*, pages 223–238. Springer, 1999.

[Pau89] Lawrence C Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3) :363–397, 1989.

[Sha48] CE Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27 :379–423 and 623–656, 1948.

[Sha49] C.E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(1) :656–715, 1949.

[Sin99] Simon Singh. *The Code Book : The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography.* Doubleday, New York, NY, USA, 1st edition, 1999.

[Ste03] Jacques Stern. Why provable security matters ? In *Advances in Cryptology—EUROCRYPT 2003*, pages 449–461. Springer, 2003.

[Wik13] Wikipédia. Histoire de la cryptographie. `http://fr.wikipedia.org/wiki/Histoire_de_la_cryptographie`, 2013. [En ligne, visité le 29 mai 2013].

Index