Université Bretagne Loire

Thèse de Doctorat

Présentée en vue de l'obtention du

Grade de Docteur de l'École des Mines de Nantes

École Doctorale Sciences et technologies de l'information, et mathématiques
Discipline : Informatique et applications, section CNU 27
Unité de recherche : Mines Nantes, Inria, Lina
Présenté par : **Ronan-Alexandre Cherrueau**
Soutenue le 18 novembre 2016
Thèse n° 2016 EMNA 0233

Un langage de composition des techniques de sécurité pour préserver la vie privée dans le nuage

M. Claude Jard, Professeur des universités, Université de Nantes, Président
M. Wolfgang De Meuter, Full Professor, Vrije Universiteit Brussel, Rapporteur
M. Ludovic Mé, Professeur, CentraleSupélec, Rapporteur
M. Yves Roudier, Professeur des universités, Université de Nice, Examinateur
M. Anderson Santana de Oliveira, Maitre de recherche, SAP, Examinateur
M. Mario Südholt, Professeur, Mines Nantes, Directeur de thèse

Résumé

Grâce à l'informatique en nuage, les services de l'information occupent un rôle prépondérant dans la société moderne (clients pour les courriers électroniques, réseaux sociaux, services de stockage). Un rôle qui exige d'être intransigeant avec la sécurité de l'information puisque ces services génèrent et sauvegardent des données qui constituent la vie privée numérique des utilisateurs.

Un service du nuage peut employer des techniques de sécurités pour assurer la sécurité de l'information. Ces techniques protègent une donnée personnelle en la rendant inintelligible pour toutes personnes autres que l'utilisateur du service. En contrepartie, certaines fonctionnalités ne peuvent plus être implémentées. Par exemple, la technique du chiffrement symétrique rend les données inintelligibles, mais empêche le calcul sur ces données.

Cette thèse avance qu'un service du nuage doit *composer* les techniques pour assurer la sécurité de l'information sans perdre de fonctionnalités. Elle se base sur l'étude de la composition de trois techniques qui sont le chiffrement, la fragmentation verticale et les calculs côté client. Cette étude montre que la composition sécurise sans perdre de fonctionnalités, mais complexifie l'écriture du service. La thèse propose alors un nouveau langage pour l'écriture de services du nuage qui assurent la sécurité des données personnelles par compositions des techniques de sécurité. Ce langage est muni de lois algébriques pour dériver, systématiquement, un service local sans protection vers son équivalent sécurisé du nuage. Le langage est implémenté en Idris et profite de son système de type expressif pour vérifier la composition correcte des techniques de cryptographie. Dans le même temps, un encodage traduit le langage en ProVerif, un vérificateur de modèle pour l'analyse automatique des propriétés de sécurité sur les protocoles cryptographiques. Cette traduction vérifie alors la sécurité des données personnelles dans le service.

Mots clefs : Vie privée, confidentialité, informatique en nuage, chiffrement, fragmentation verticale, calculs côté client, langage de programmation, lois algébriques, vérification automatique, Idris, ProVerif.

Abstract

In the context of cloud computing, information services are ever-present (web-based email client, social networks, file hosting). Such services yield personal data that constitutes the privacy of the client and therefore, should be uncompromising on information privacy.

A cloud service can use security techniques to ensure information privacy. These techniques protect privacy by converting the client's personal data into unintelligible text. But they can also cause the loss of some functionalities of the service. For instance, a symmetric-key cipher protects privacy by converting readable personal data into unreadable one. However, this causes the loss of computational functionalities on this data.

This thesis claims that a cloud service has to *compose* security techniques to ensure information privacy without the loss of functionalities. This claim is based on the study of the composition of three techniques : symmetric cipher, vertical data fragmentation and client-side computation. This study shows that the composition makes the service privacy preserving, but makes its formulation overwhelming. In response, the thesis offers a new language for the writing of cloud services that enforces information privacy using the composition of security techniques. This language comes with a set of algebraic laws to systematically transform a local service without protection into its cloud equivalent protected by composition. An Idris implementation harnesses the Idris expressive type system to ensure the correct composition of security techniques. Furthermore, an encoding translates the language into ProVerif, a model checker for automated reasoning about the security properties found in cryptographic protocols. This translation checks that the service preserves the privacy of its client.

Keywords : Privacy, information privacy, cloud computing, encryption, data fragmentation, client-side computation, programming language, Idris, algebraic laws, automatique vérification, ProVerif.

Publications

Plusieurs idées et figures présentées dans cette thèse ont fait l'objet des publications suivantes :

A Language for the Composition of Privacy-Enforcement Techniques Ronan-Alexandre Cherrueau, Rémi Douence et Mario Südholt dans *RATSP - The 2015 IEEE International Symposium on Recent Advances of Trust, Security and Privacy in Computing and Communications, pages 1037–1044, Helsinki, Finland, August 2015.* IEEE.

Enforcing Expressive Accountability Policies Ronan-Alexandre Cherrueau et Mario Südholt dans *WETICE - IEEE International Conference on Enabling Technologies : Infrastructure for Collaborative Enterprises, pages 333–338, Parma, Italie, Jun 2014.* IEEE Computer Society.

Adapting Workflows Using Generic Schemas : Application to the Security of Business Processes Ronan-Alexandre Cherrueau, Mario Südholt et Omar Chebaro dans *CloudCom - 5th IEEE International Conference on Cloud Computing Technology and Science, pages 519–524, Bristol, Royaume-Uni, Decembre 2013.* IEEE Computer Society.

Reference Monitors for Security and Interoperability in OAuth 2.0 Ronan-Alexandre Cherrueau, Rémi Douence, Jean-Claude Royer, Mario Südholt, Anderson Santana de Oliveira, Yves Roudier et Matteo Dell'Amico dans *Data Privacy Management and Autonomous Spontaneous Security - 6th International Workshop, SETOP 2013, pages 235–249, Egham, UK, September 2013.* Springer.

Flexible Aspect-Based Service Adaptation for Accountability Properties in the Cloud Ronan-Alexandre Cherrueau, Omar Chebaro et Mario Südholt dans *Proceedings of the 4th International Workshop on Variability and Composition (VariComp'13), pages 13–18, Fukuoka, Japan, March 2013.* ACM.

Remerciements

> *"Il y a un milliard d'années, il n'y avait pas un seul homme sur Terre, il n'y avait qu'un poisson, pauvre chose visqueuse pourvue d'écailles, de branchies et de petits yeux tout ronds. Il vivait dans l'océan, et l'océan était comme une prison, et l'air formait comme un toit au-dessus de sa geôle. Personne ne pouvait traverser le toit. On mourait si on le traversait, disait-on. Mais il y eut un poisson qui traversa, et il mourut. Et il y en eut un autre, et il le traversa, et il mourut. Mais il y eut un troisième poisson, et il le traversa, et ce fut comme si son cerveau était en feu, ses branchies en flammes. et l'air l'étouffait, et le soleil était une torche dans ses yeux, et il resta gisant dans la boue, attendant la mort, mais il ne mourut pas. Il rampa sur la plage rentra dans l'eau et dit : « Dites donc, il y a un tout autre monde, là-haut ! » Et il y retourna, et il vécut encore, disons deux jours, et puis il mourut."*
>
> *La Tour de verre* (p.95, poche) — Robert Silverberg

En 2012, Hervé Grall m'a initié à la théorie des langages de programmation à travers son stage de master. Grâce à lui, j'allais découvrir le monde de la recherche informatique et le plaisir d'approfondir ma compréhension des langages de programmation. Cette expérience a décidé de mon avenir pour les années à suivre : faire une thèse en informatique dans les langages de programmation. Et faire une thèse n'est pas quelque chose que l'on peut faire tout seul. Cela demande un soutien scientifique, intellectuel et émotionnel. C'est pourquoi je profite de cette page pour remercier les personnes qui m'ont apporté ce soutien.

— Mario Südholt, mon directeur de thèse. Je le remercie de m'avoir accueilli dans son équipe et conseillé durant toutes ces années pour faire de moi un chercheur en informatique. De la même manière, je remercie plus généralement tous les membres de l'équipe ASCOLA avec une pensée particulière pour ceux avec qui j'ai l'habitude de partager le café : Jacques Noyé, les petits amis du nuage (Alexandre Garnier, Rémy Pottier, Simon Dupont) et les petits amis de Coq (Kevin Quirin et Simon Boulier).
— Hervé Grall et Rémi Douence pour leurs conseils précieux autant théoriques que techniques. Vous m'avez fait découvrir la satisfaction d'une conception fonctionnelle et l'intérêt de considérer sa bibliothèque comme une algèbre. En particulier, Rémi, merci d'avoir pris le temps de partager avec moi tes connaissances quant à l'utilisation des monades pour structurer les programmes fonctionnels. Ceci a radicalement changé, et pour toujours, ma façon de concevoir un programme informatique.
— Florent Marchand de Kerchove, Jonathan Pastor et Walid Benghabrit, co-bureau, mais surtout, amis et compagnons autant dans la recherche (papredi) que dans l'amusement (arcadi). On s'est bien marré tous les

quatre. La fin de thèse a son lot de désolation ; en voilà une ; je vais regretter de ne plus vous avoir dans le bureau. Fini les discussions sur les principes de programmation, les débats sur ce qu'apportent les langages dynamiques par rapport à la vérification statique, le hacking d'emacs, les joutes entre les perso. shoto *vs.* à charge … Heureusement, on peut toujours se mettre sur le pif entre deux parties avec un Ken du live.

— Adrien Bougouin et Brice Nédelec, les copains du Lina (initialement du master ALMA), avec lesquels j'ai partagé de longues conversations sur la ~~misère~~ joie de faire une thèse. Merci de m'avoir écouté à chaque fois que j'ai fait du prosélytisme pour la programmation fonctionnelle. Je ne désespère pas de vous faire, un jour, implémenter le *parser combinator*. Comme ça, vous aussi vous verrez la lumière :)
— Les joyeux membres de l'initiative Discovery (Adrien Lèbre, Anthony Simonet, Dimitri Pertin, Matthieu Simonin) qui m'ont accueillie, pour trois ans d'une nouvelle aventure avec eux, alors que je n'avais pas fini la rédaction de mon manuscrit.
— Wolfgang De Meuter, Ludovic Mé, Claude Jard, Yves Roudier et Anderson Santana de Oliveira, les membres du jury de cette thèse, qui ont pris le temps de lire mon manuscrit pour me faire des retours judicieux.

Bien évidemment, rien de tout cela n'aurait pu être possible sans ma famille et mes amis. Merci Maman, Papa, Ambre et Margaux pour votre soutien pendant toutes ces années.

Enfin, merci à ma compagne Émilie et notre petite fille Aline. Il m'est impossible de trouver les mots pour vous dire à quel point je vous suis reconnaissant pour votre patience et votre amour.

Table des matières

Table des figures & programmes

Introduction

1

Les ressources mutualisées et plus particulièrement l'*informatique en nuage* (cloud computing) se sont très largement imposés ces dernières années pour être, aujourd'hui, un élément incontournable dans la création de nouvelles applications. En informatique, le terme *nuage* est utilisé pour représenter les ressources (code applicatif, base de données, système d'exploitation, plates-formes matérielles…) accessibles depuis n'importe où dans le monde, faisant de ces ressources une commodité pour les développeurs d'applications.

Sur les quarante dernières années, un développeur d'applications devait *investir* dans du matériel, des licences logiciels et former du personnel pour concevoir ses applications. Mais maintenant grâce au nuage, il se contente de *louer* les ressources nécessaires à son application pour profiter des avantages qu'offre cette location.

Par exemple, une application qui sauvegarde des informations doit s'assurer que celles-ci sont sauvegardées de façon pérenne. Une tâche qui est ardue et coûteuse. Ardue, car il faut maintenir l'accès aux serveurs de données et gérer les réplications. Coûteuse, car héberger les serveurs de données requiert une infrastructure physique spéciale. Pire, gérer les réplications requiert un administrateur de base de données. Mais aujourd'hui, grâce à l'informatique en nuage et son système de location, cette tâche d'intendance bas niveaux est assurée par le bailleur. Le bailleur libère ainsi le développeur d'applications qui peut mettre l'accent sur l'innovation et la logique métier lors de la conception de son application.

Concrètement, l'informatique en nuage propose un nouvel environnement pour l'implémentation, la réutilisation, le déploiement et l'exécution de services informatiques. Dans cet environnement, les détails d'un service fourni sont abstraits pour le locataire. Une abstraction qui simplifie l'utilisation, mais qui, inéluctablement, cause *la perte du contrôle des données* qui se retrouvent concédées au bailleur.

1.1 Formulation du problème

En quelques années, les applications qui utilisent et servent le nuage sont devenues légions et de nombreux utilisateurs leur confient chaque jour des données personnelles ; un don d'information qui n'est pas sans risque puisque peu d'applications sont responsabilisées quant à la préservation des données personnelles. Pire encore, les données personnelles constituent une manne pour l'économie, ce qui incite les sociétés informatiques du nuages à collecter autant de données personnelles que possibles.

Une donnée personnelle est définie comme toute information relative à un individu qui permettrait de l'identifier. Le nom et la date de naissance, qui sont disponible à l'état civil, en sont des exemples.

Dans le contexte de l'informatique en nuage, une donnée personnelle revêt des formes plus diverses telles que la sauvegarde des transactions d'un client sur un site marchand, les mots clés dans un moteur de recherche, les échanges de courriels ou l'enregistrement du comportement (clics) d'un internaute. Ces données personnelles se retrouve ainsi numérisées pour être exploitées.

Exploiter ces données personnelles constitue une ressource financière très intéressante, comme en témoigne le chiffre d'affaires des entreprises du nuage qui s'y emploient. En effet, que ce soit par la sauvegarde des transactions marchandes, par la collecte de mots clefs ou par l'enregistrement du comportement, toutes ces techniques permettent l'élaboration d'un profile de l'individu. Un profil qui, chez *Amazon*[1] par exemple, sert à recommander des produits pour inciter l'achat. Tandis que chez *Facebook*[2] et *Google*[3], sert à vendre des annonces publicitaires ciblées.

Facebook et Google illustrent bien la fausse dévotion qui caractérise les sociétés du nuage d'aujourd'hui. C'est sous couvert d'un service personnalisé et souvent gratuit (courriels, agenda, stockage et partage en ligne…) que ces sociétés amoncellent des très grands volumes de données personnelles (liste de contacts, documents privés, photos personnelles…). Provoquant une invasion dans la vie privée de l'individu.

L'aspect moral n'est pas considéré dans cette thèse. La question de la morale sur la collecte et l'utilisation des données personnelles est discutée dans la loi européenne General Data Protection Regulation (GDPR), adopté en Avril 2016 et qui entrera en vigueur en 2018. Cette loi prévoie une protection des données personnelles stricte avec des sanctions très sévères pour tout manquement à cette protection.

Sans considérer l'aspect moral que représente l'exploitation des données personnelles, la collecte, elle, dépossède l'individu de la gestion de ses données. Le respect de la vie privée de l'individu se retrouve délégué à l'application. Or, l'application est souvent dans l'incapacité d'assurer le respect de la vie privée à cause du caractère abstrait et compositionnel du nuage. S'en suit une exploitation peu transparente des données personnelles que l'individu ne peut pas contrôler, ni même connaître la plupart du temps (*ex.*, surcollecte des données, violation de la spécification d'usage, mise à disposition d'un tiers non stipulé ou absence de moyen pour l'individu de supprimer, rectifier et compléter sa donnée – Pearson, 2011).

1.1.1 Perte de contrôle : le contrecoup de l'abstraction et de la composition des services

Dans l'informatique en nuage, les détails d'un service fourni sont abstraits. Cette abstraction profite au concepteur d'applications qui compose facilement son service avec d'autres services de plus bas niveaux. De plus, avec la composition, le concepteur se focalise sur la logique métier et délègue au bailleur la gestion des tâches bas niveaux. En contrepartie, il délègue également la gestion des données de ses utilisateurs. De ce fait, les utilisateurs se voient privés du contrôle qu'ils avaient sur leurs données personnelles et sur le respect de leur vie privée.

L'exemple d'un agenda personnel en ligne expose ce risque (*fig.* 1). À l'instar de Mozilla Lightning[4] ou Google Calendar[5], l'agenda personnel permet, via un calendrier, d'ajouter et modifier un rendez-vous ; d'envoyer un rappel pour

1 amazon.fr
2 facebook.com
3 google.com
4 mozilla.org/en-US/projects/calendar
5 google.com/calendar/about

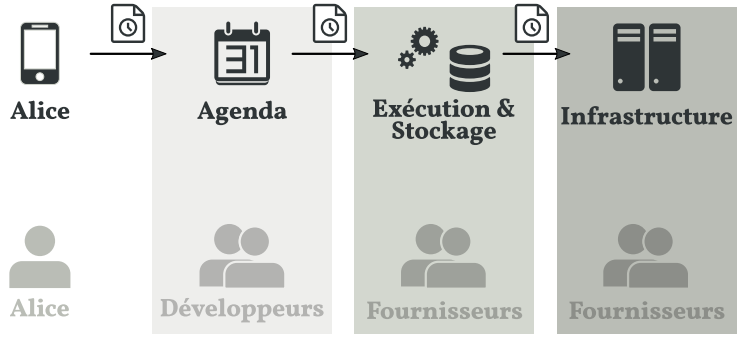


FIGURE 1 – *Ajout par Alice d'un rendez-vous sur son agenda personnel en ligne.*

un futur rendez-vous ; de rechercher un événement *etc.* tout en profitant des avantages du nuage.

Cet exemple considère quatre acteurs : (1) Alice, l'utilisatrice de l'agenda en ligne, (2) l'application agenda en ligne, (3) un service d'exécution de code et de base de données distant et (4) un bailleur de ressources physiques ; les trois derniers acteurs étant exécutés dans le nuage.

Le développeur de l'agenda loue un service d'exécution de code qui déploie et exécute l'application dans le nuage pour la rendre hautement disponible. Il profite également des bases de données distantes pour y stocker les rendez-vous d'Alice de façon simple et pérenne. Dans les mêmes temps, le fournisseur de base de données profite de la très large infrastructure du bailleur des ressources physiques pour déployer son gestionnaire de bases de données et s'adapter au rythme de la demande.

Dans cet exemple, ni Alice ni le développeur de l'agenda n'ont la main, physiquement, sur les rendez-vous qui sont stockés et gérés par le bailleur de base de données. Par conséquent, lorsque Alice demande à supprimer un rendez-vous de l'agenda, la demande est déléguée par l'application agenda au service de base de données qui se doit de faire la suppression.

Une délégation comme celle-ci est naturelle dans l'informatique en nuage, mais a des implications potentiellement graves pour l'utilisateur.

Google Announcement, xkcd.com/1361

Par exemple, il est impossible pour le développeur de l'agenda en ligne de garantir à Alice qu'un rendez-vous a réellement été supprimé. Le bailleur de base de données peut retenir le rendez-vous et simuler sa suppression. Le bailleur de base de données peut lui-même être trompé par le bailleur de ressources physiques si ce dernier fait des copies de son système.

De même, il est impossible pour le développeur de l'agenda de garantir l'usage qui est fait du rendez-vous. Le bailleur peut utiliser cette information dans un dessein autre que celui attendu par Alice (revente d'information à un tiers, passe-droit accordé au gouvernement…).

Plus grave, le bailleur peut cesser son activité, ce qui, *de facto*, provoque la suppression de tous les rendez-vous qui lui ont été confiés. Un tel arrêt entraîne alors une interruption de l'application agenda et une perte sèche de ses rendez-vous pour Alice.

Enfin, l'agenda, la base de données en ligne et l'infrastructure sont des systèmes complexes qui peuvent avoir des dysfonctionnements et des failles de

6 korben.info/windows-10-est-il-un-systeme-dexploitation-diabolique.html
7 indiewebcamp.com/site-deaths
8 framasoft.org

sécurités. Par conséquent, même si le développeur et les bailleurs sont bien intentionnés, Alice ne peut jamais être sûre de l'usage qui pourrait être fait de ses données personnelles (Grimmelmann, 2005).

Parmi les dérives, on peut citer la récente polémique autour de Microsoft et sa politique de confidentialité sur Windows 10[6]. Ou encore la liste[7] élaborée par la communauté IndieWeb qui énumère tous les services majeurs du nuage qui ont cessé de fonctionner ces dix dernières années. Et enfin, le site Web degooglisons-internet.org par Framasoft[8] qui recense de nombreux articles traitant du sujet.

Tous ces usages peuvent êtres menés sans l'approbation de l'utilisateur et beaucoup de faits réels sont l'illustration de dérives abusives. Pour l'utilisateur, c'est un risque de vol d'identité, de perte de données et une destruction de sa vie privée. Pour les entreprises du nuage, c'est une perte de la confiance de ses locataires et une baisse de ses revenus.

La situation actuelle est donc très contestable du point de vue de la préservation des données personnelles. Il faut sécuriser le nuage pour le rendre plus sûr et empêcher les violations au droit à la vie privée. De manière générale, la sécurisation consiste à analyser et comprendre les risques qui peuvent apparaître pour les utilisateurs d'un système. Puis, prendre les moyens nécessaires et appropriés pour limiter ces risques. Et enfin, s'assurer que les moyens pris protègent correctement les utilisateurs du système (Lampson, 2004).

1.1.2 Des cadriciels de sécurisation limités

Historiquement, les techniques de cryptographies (Menezes et collab., 1996) sont les premières à être employées pour protéger les informations des yeux indiscrets. Elles cherchent à protéger un message en assurant confidentialité, authenticité et intégrité. Deux des exemples les plus connus sont le très simple *chiffre de César* (Lyons, 2012a) utilisé par Jules César pour communiquer avec ses généraux et la très stratégique *machine Enigma* (Lyons, 2012b) utilisée par l'armée allemande pendant la Deuxième Guerre mondiale.

Les techniques de cryptographie rendent un message inintelligible à toutes personnes autres que celles autorisées. Rien d'étonnant donc, à voir ces techniques employées dans une approche pour la préservation de la vie privée. Celle-ci se nomme la *protection intégrée de la vie privée* (Privacy-by-Design ou PbD – Cavoukian, 2011 ; Spiekermann, 2012).

L'approche de protection intégrée de la vie privée

Le dessein de la protection intégrée de la vie privée est de concevoir et développer les applications du nuage de telle manière qu'elles satisfassent les principes et les règles du respect de la vie privée. Une application conçue suivant l'approche PbD est respectueuse de la vie privée de ses utilisateurs et par conséquent, préserve leurs données personnelles. Pour se faire, l'approche dispose de techniques dites de *renforcement du respect de la vie privée* (Privacy-Enhancing Technologies), dont les célèbres techniques de cryptographie.

Un exemple de technique de cryptographie est la machine Enigma mentionnée plus tôt. Il s'agit plus exactement d'un *chiffrement symétriques* (Menezes et collab., 1996) qui permet de chiffrer et déchiffrer un message à l'aide d'un mot clef. Mais beaucoup d'autres technologies sont utilisées, comme la *preuve à divulgation nulle de connaissance* (zero-knowledge proof – Goldwasser et collab., 1989 ; Danezis et Livshits, 2011) qui donne la possibilité de traiter de manière privée les données personnelles en effectuant le traitement chez le client. Ou la notion *d'indistinguabilité* (differential privacy – Dwork et collab., 2006 ;

Dwork et Roth, 2014) qui permet d'interroger une base de données statistique sans révéler d'informations sur ses occupants.

Les protections offertes par une technique sont *proactives*. Elles *empêchent* et *préviennent* de l'usage frauduleux d'une donnée personnelle. En contrepartie, elles ne proposent aucune solution si l'usage frauduleux se produit. L'avantage d'une telle stratégie est qu'elle est souvent plus simple et limite donc le nombre de composants du système responsables des informations de protections. Généralement, seul l'utilisateur de l'application à connaissance des données sensibles.

Malheureusement pour les utilisateurs, chaque technique de renforcement du respect de la vie privée s'applique dans un contexte très spécifique. Par exemple, le chiffrement symétrique a pour visée de protéger les données sauvegardées, mais pas calculées, tandis que la notion d'indistinguabilité n'a de sens que pour les bases de données statistiques.

Développer une application substantielle qui préserve les données personnelles de ses utilisateurs oblige donc à composer les techniques. Mais, dans le domaine PbD, les travaux scientifiques effectués depuis dix ans se concentrent sur l'élaboration et le perfectionnement de nouvelles techniques et non sur leur composition. Or, la composition est sujette à de nombreuses erreurs.

Pour s'en convaincre, il suffit d'imaginer une application qui est faite de plusieurs flux d'informations. Dans cette application le développeur rend anonyme certains flux et en chiffre d'autres pour préserver les données personnelles tout en maximisant la quantité d'information utilisable. Mais, en multipliant les techniques, le développeur multiplie la complexité de l'écriture de l'application. Dès lors, il lui est très difficile d'être sûr que chaque flux d'information préserve correctement les données personnelles de ses utilisateurs.

L'approche de responsabilisation

Les techniques de cryptographie, qui empêchent la fuite des données personnelles, ne sont pas toujours suffisantes pour sécuriser le nuage. Les données personnelles peuvent être copiées, agrégées, disséminées et même inférées grâce aux techniques de fouilles de données. Un exemple surprenant est l'article de Narayanan et Shmatikov sur la désanonymisation des profils Netflix grâce à l'entrecoupement de données tiers (2008). C'est pourquoi Weitzner (ancien directeur de la technologie à la Maison Blanche) avec Berners-Lee (considéré comme l'inventeur du *World Wide Web*) et collab. (2008) ont défini que le futur enjeu du nuage était de le *responsabiliser* (to be accountable). C'est-à-dire, avoir un nuage où les utilisateurs, comme les fournisseurs de services seraient dans l'obligation de rendre compte de leurs agissements. Le dessein est de limiter l'usage nocif qui pourrait être fait des données personnelles.

Weitzner et collab. (2008) dans leur article sur le besoin de responsabilisation proposent une autre approche pour sécuriser le nuage. L'usage qui est fait des données personnelles est transparent, ce qui permet de capturer et condamner un emploi abusif. Pour se faire, l'approche se base sur trois composants. Le premier est un registre qui contient tous les évènements relatifs à l'utilisation d'une donnée personnelle. Le second est un langage pour décrire les usages qui peuvent être faits d'une donnée personnelle. Enfin, le troisième

est un outil de raisonnement pour vérifier la bonne application des règles d'usages.

La force de l'approche présentée par Weitzner et collab. est qu'elle capture complètement la notion de responsabilisation du nuage. En effet, Weitzner, tout comme Pearson (2011) et Feigenbaum et collab. (2012) s'accordent à dire que la responsabilisation doit se faire, non seulement au moyen de techniques proactives, comme pour l'approche de protection intégrée de la vie privée. Mais, également au moyen de techniques *réactives* qui *condamnent*, après coup, l'usage frauduleux d'une donnée personnelle, par exemple, en enquêtant sur le fraudeur et en lui demandant une compensation. Dans l'approche de responsabilisation, c'est l'outil de raisonnement qui empêche ou condamne un usage frauduleux grâce au registre d'évènements et aux règles d'usages.

Cependant, le modèle de responsabilisation est biaisé par sa stratégie réactive lorsqu'il s'agit de respecter la confidentialité des données personnelles. Cette stratégie oblige de gérer l'identification des utilisateurs du système pour pouvoir condamner les individus malveillants. Ce point est critiquable, car l'application se retrouve avec des traces d'exécutions où l'utilisateur est clairement identifié. La défense souvent utilisée contre cette critique est de dire que l'application est de bonne foi. Mais être de bonne foi n'est pas suffisant pour la confidentialité. Le fait de retenir des données personnelle rend l'application faillible. C'est le cas de Facebook qui a un très bon mécanisme de responsabilisation entre les utilisateurs, mais est faillible aux demandes du gouvernement.

Pour cette raison, cette thèse se concentre sur l'approche de la protection intégrée de la vie privée et n'étudie pas l'approche de responsabilisation.

1.2 Dessein et périmètre de la thèse

Le défi, dessein et périmètre de l'approche sur la protection intégrée de la vie privée (PbD) sont les suivants.

Défi. Chaque technique de cryptographie a de très bonnes propriétés de contrôle sur les données personnelles. Mais, leur utilisation se fait dans un contexte limité. Par exemple, le chiffrement symétrique est utilisable exclusivement quand la donnée personnelle doit être sauvegardée. Si une application doit faire un calcul sur une donnée personnelle, il faudra employer une solution différente, comme par exemple, le chiffrement homomorphe pour appliquer un calcul sur une donnée chiffrée. Développer une application substantielle, qui crée, lit, sauvegarde et applique des opérations sur une donnée personnelle requiert de composer les techniques de cryptographie.

En informatique la composition consiste à former un programme complexe en combinant des programmes plus simples. Le résultat de la composition doit former un tout cohérent, mais ne doit pas altérer les parties plus simples. Ceci par volonté d'avoir un code clair dont la correction est facile à établir.

Mais, une technique de cryptographie représente un morceau de code compliqué et combiner les techniques complexifie encore plus l'application. Donc il est difficile de se convaincre que le code est exact.

Dessein. Le dessein de cette thèse est d'offrir au développeur d'application PbD un langage pour l'écriture de service du nuage qui assure la sécurité des données personnelles par composition des techniques de cryptographie. Le langage garantit une composition sûre des techniques pour produire une application qui s'exécute sans bogues de composition ni risques pour la confidentialité des données personnelles.

Périmètre. Pour le modèle de la protection intégrée de la vie privée, cette thèse se focalise sur un sous-ensemble des techniques respectueuses de la vie privée. À savoir, le chiffrement, la fragmentation et les calculs côté client. Cette limitation est due à la difficulté de faire une liste exhaustive de toutes les techniques. De plus, toutes les techniques ne sont pas intéressantes à composer. De ce fait, la technique de l'indistinguabilité par exemple qui est le fer de lance pour la préservation des données personnelles n'est pas envisagée.

Notons que la technique de l'indistinguabilité essuie des critiques ces derniers temps. Les implémentations oscillent entre des versions trop strictes qui retournent des résultats trop anonymes pour être pertinent et des versions trop laxistes qui ne protègent pas correctement la vie privée de leurs occupants (Bambauer et collab., 2013).

Le langage de composition vise le développement d'applications qui sont déployées au niveau SaaS et qui utilisent des bases de données au niveau PaaS. De même, le langage se limite aux applications qui n'ont pas d'interactions avec plusieurs utilisateurs (*ex.*, Alice partage un rendez-vous avec Bob).

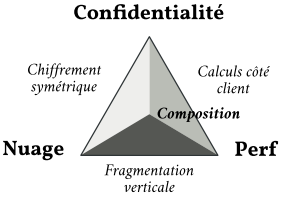
Enfin, l'objectif du langage est de garantir la *confidentialité* des données personnelles. Les propriétés d'intégrité et de disponibilité ne sont pas étudiées ici. Il faut toutefois faire remarquer que ceci n'est pas une limite dû au langage.

1.3 Vue d'ensemble de la thèse

La thèse est organisée comme suite.

Chapitre 2. Ce chapitre présente les connaissances nécessaires au reste de ce document. Il définit ce qu'est l'informatique en nuage, ses abstractions, sa manière de manipuler les données, ses avantages et ses inconvénients. Il définit également sous quelles conditions la préservation des données personnelles est violée dans un système informatique. Puis, il présente le principe d'un système informatique sécurisé avec ses trois piliers : politique, mécanisme et assurance. Le pilier politique, qui spécifie les règles d'usages, est instancié pour préserver les données personnelles dans le nuage. La suite du chapitre analyse les mécanismes et assurances offertes par l'approche de protection intégrée de la vie privée pour appliquer cette politique.

Chapitre 3. Ce chapitre présente en détail le cas de l'agenda personnel en ligne. Il étudie l'implémentation de deux requêtes de l'application agenda lorsque le développeur suit l'approche PbD. Pour ce faire, ces deux requêtes sont implémentées tour à tour par les trois techniques considérées dans cette thèse. Les implémentations sont évaluées sur trois critères (confidentialité, utilisation du nuage et performance) pour montrer leur limite. Puis les mêmes requêtes sont implémentées par composition des trois techniques. Les expérimentations montrent que l'application agenda ne souffre plus des limites précédentes, mais elle est plus difficile à implémenter et donc prompte à plus de bogues. Ce chapitre propose alors une nouvelle approche nommée "approche du *nuage confidentiel*" et fait la liste de ses besoins. Les trois chapitres suivants implémentent ces besoins.



Approche du nuage confidentiel : une application PbD doit, (1) être sécurisée, (2) exécutée dans le nuage, (3) avoir des performances sacrifiées au profit de la confidentialité et du nuage, (4) voir sa simplicité sacrifiée au profit de la performance.

Chapitre 4. Ce chapitre présente un nouveau langage nommé C2QL (*Cryptographic Compositions for Query Language*) pour le développeur d'application PbD. Ce langage est accompagné de lois algébriques pour composer les fonctions de cryptographie. À partir de ces lois, une méthodologie simple est élaborée pour que le développeur dérive, systématiquement, une application locale sans protection vers son équivalent qui suit l'approche du *nuage confidentiel*. Dans le même temps, une traduction de C2QL en π-calcul montre comment l'application est distribuée sur le nuage.

Chapitre 5. Ce chapitre présente la vérification automatiquement de la confidentialité des données personnelles sur une requête C2QL. Pour ce faire, ce chapitre utilise l'outil ProVerif, un vérificateur de modèle pour l'analyse automatique des propriétés de sécurité sur les protocoles de cryptographie.

Chapitre 6. Ce chapitre présente une implémentation du langage C2QL en tant que langage dédié embarqué dans Idris, un langage de programmation fonctionnel avec des types dépendants. Ces types sont utilisés pour prévenir du risque d'une mauvaise composition des fonctions de C2QL. Ils préviennent des quatre classes d'erreurs : erreurs de manipulation des n-uplets, erreurs de spécification de l'environnement, erreurs de composition des techniques de cryptographie et erreurs de composition des requêtes. L'implémentation produit un terme C2QL correct, mais ne propose pas de compilation vers une application concrète par manque de temps. Toutefois, le code implémente une traduction vers un terme π-calcul qui est régulièrement utilisé pour modéliser les applications du nuage. Cette thèse considère que la traduction suffit à montrer que le langage C2QL est pragmatique et peut être utilisé pour implémenter des applications PbD du nuage concrètes.

Finalement, cette thèse conclut avec le chapitre 7 qui résume les contributions et lève d'autres questions de recherches.

Quand les données personnelles montent dans le nuage

2

Ce chapitre fournit les connaissances nécessaires au reste de ce document. Dans un premier temps, il reprend et étaye les concepts énoncés dans l'introduction tels que l'informatique en nuage (§2.1) et la vie privée pour les données personnelles hébergées dans le nuage (§2.2).

Puis, ce chapitre présente le principe d'un système sécurisé (§2.3). Il définit les piliers nécessaires à l'élaboration d'un tel système (§2.3.1) et les instancie pour la préservation des données personnelles dans le nuage (§2.3.2). Cette section montre que le respect de la vie privée dans un système informatique, et plus particulièrement, la préservation des données personnelles hébergées dans le nuage a les mêmes problématiques qu'un système informatique sécurisé.

Une étude est ensuite conduite sur l'approche de protection intégrée de la vie privée (Cavoukian, 2011 – §2.4) qui est l'approche considérée dans cette thèse pour la préservation des données personnelles dans le nuage. L'étude porte sur trois techniques de cryptographie qui sont utilisés dans cette approche. Ces trois techniques sont le chiffrement, la fragmentation verticale et les calculs côté client. Elle montre comment ces techniques doivent être utilisées pour réaliser une application du nuage qui préserve les données personnelles. La dernière section (§2.5) conclut sur les manques de ces techniques et esquisse l'hypothèse faite par cette thèse. À savoir : il faut composer les techniques de cryptographie dans l'approche de protection intégrée de la vie privée pour concevoir une application du nuage qui préserve les données personnelles.

2.1 L'informatique en nuage

L'informatique en nuage se définit comme "un système parallèle et distribué, fait d'un ensemble d'unité de calculs interconnectés, fournis à la demande et présentés sous la forme d'une seule ressource unifiée" (Buyya, 2009). En d'autres termes, l'informatique en nuage représente une ressource telle qu'une application, une base de données ou un système d'exploitation. Cette ressource est fournie à la demande grâce à un système de location et est accessible depuis n'importe où dans le monde, principalement par Internet. La gestion de la ressource est faite par le bailleur dont l'activité se focalise sur l'apport de la ressource. Ceci lui permet d'avoir une infrastructure plus compliquée avec des avantages très intéressants pour les sociétés informatiques.

Parmi les avantages offerts par le bailleur, deux sont intrinsèques à l'informatique en nuage (Mell et Grance, 2011). Le premier est la *disponibilité* : un

locataire doit, à tout moment, être capable de louer une ressource de manière automatique, *c.-à-d.*, sans requérir à une intervention humaine avec le bailleur. Le second est l'*élasticité* : le bailleur doit pouvoir subvenir et s'adapter en fonctions des besoins requis par le locataire.

En 1961 lors de la célébration du centenaire du MIT, John McCarthy (créateur du langage Lisp (1965) et fondateur de l'intelligence artificielle) fut le premier à faire l'analogie entre les ressources d'un ordinateur et les commodités comme l'eau et l'électricité.

Une analogie peut être faite avec le courant électrique. En France, un foyer souscrit à un fournisseur d'électricité (*ex.*, EDF). Le fournisseur gère une infrastructure très complexe, mais très efficace pour offrir de l'électricité. En outre, cette infrastructure et les compétences pour la gérer sont invisibles pour le foyer. Le contact entre les habitants du foyer et l'électricité se fait par une prise de courant, soit, une interface très simple à utiliser (*disponibilité*). EDF fournit en continu de l'électricité aux Français, même en hiver quand la consommation y est la plus importante (*élasticité*).

Le nuage en tant que service

Aujourd'hui, l'informatique en nuage suscite un grand intérêt auprès des entreprises informatiques. Pour preuve, des géants tels que Amazon[1], Google[2] et Microsoft[3] proposent, moyennant rétribution, des infrastructures pour accueillir et fournir des ressources hébergées dans le nuage. À l'instar d'EDF avec son courant électrique, ces infrastructures sont très simples d'utilisation. Elles proposent un nouvel environnement pour le déploiement de nouveaux services en fournissant à la demande, des ressources elles-mêmes présentes sous la forme de services. Ces infrastructures sont classées en trois catégories de services qui sont le SaaS, le PaaS et le IaaS.

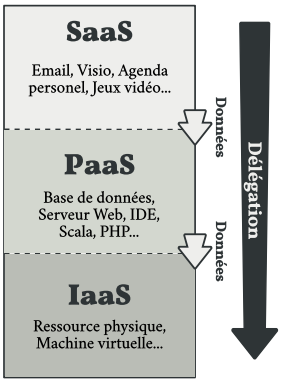
***Logiciel en tant que service* (Software as a Service -- SaaS).** Permet à l'utilisateur de *disposer* d'une application qui profite des avantages du nuage. On y retrouve des applications telles que la gestion de courriels, la lecture de flux RSS, la visioconférence ou l'agenda personnel (décrit en introduction §1.1.1), *etc*. Les applications SaaS sont couramment accessibles depuis un navigateur Web. Elles font donc preuve d'une grande disponibilité. Dans cette infrastructure, l'utilisateur n'a pas de contrôle sur les infrastructures sous-jacentes telles que les mécanismes de sauvegarde des données, la couche réseau ou le système d'exploitation. Google mails, GitLab et Feedly sont des exemples de SaaS.

***Plateforme en tant que service* (Platform as a Serice -- PaaS).** Permet à l'utilisateur de *déployer* dans le nuage une application qui profite des outils nuagiques mis en place pour le bailleur. Les outils peuvent être un système d'exploitation, un environnement d'exécution, un serveur Web ou une base de données (comme celle vue en introduction §1.1.1), *etc*. Les services PaaS offrent une très grande élasticité. Ils peuvent s'adapter automatiquement à la demande. Dans ces services, l'utilisateur n'a pas le contrôle sur les infrastructures sous-jacentes telles que la couche réseau ou la gestion de la mémoire. Heroku, OpenShift et Microsoft SQL Azure sont des exemples de PaaS.

1 aws.amazon.com/ec2
2 cloud.google.com/appengine
3 azure.microsoft.com

***Infrastructure en tant que service* (Infrastrcuture as a Service -- IaaS).** Permet à l'utilisateur d'*accéder* à des ressources de calculs fondamentales tels que des machines physiques, des machines virtuelles, des répartisseurs de charge ou du stockage d'objets, *etc*. En général, l'utilisateur à un contrôle complet sur la ressource, mais ce contrôle ne permet pas la gestion de l'infrastructure réseau. Amazon EC2/S3, OpenStack et Rackspace sont des exemples de IaaS.

Dans leur ontologie sur l'informatique en nuage, Youseff et collab. (2008) structurent les trois catégories : SaaS, PaaS et IaaS sous la forme d'une pile, à l'instar du modèle OSI (Zimmermann, 1980). La pile va des applications (SaaS) aux infrastructures physiques (IaaS). Elle symbolise une architecture en couche où les services de plus haut niveau peuvent utiliser des services d'un niveau inférieur. Ainsi, un fournisseur de SaaS peut exploiter un service de type PaaS, qui peut lui-même se servir de IaaS.



Pile du nuage selon Youseff et collab. (2008).

Ces trois couches d'abstractions facilitent le développement d'applications, mais causent la perte du contrôle des données. Un très simple cas d'utilisation permet de comprendre cette perte de contrôle : si un utilisateur confie des informations personnelles à une application (SaaS), que cette application est déployée sur le nuage (PaaS) et qu'elle est exécutée par des machines virtuelles (IaaS). Alors, l'utilisateur à un contrôle sur l'application SaaS mais n'en a pas sur les services PaaS et IaaS.

De même, l'utilisation des réseaux publics (*c.-à-d.*, Internet) pour accéder aux services du nuage entraîne des risques pour les données personnelles. Si la connexion n'est pas chiffrée, avec le protocole `https` par exemple (Fielding et Reschke, 2014), les participants reliés au même point d'accès que la cliente peuvent lire les données personnelles qu'elle transmet.

2.2 La vie privée dans un système informatique

Définir la vie privée n'est pas une chose aisée tant la notion de vie privée est large et suscite des opinions divergentes. Dans les grandes lignes, la vie privée représente ce qui est d'ordre strictement personnel et qui ne concerne pas les autres. Le droit à la vie privée est reconnu comme étant indispensable pour l'émancipation personnelle. Préserver la vie privée permet l'autonomie et laisse l'individu libre dans ses prises de décision au sein d'un groupe privé. En d'autres termes, manquer au droit à la vie privée produit une société sans pluralité, voir conformiste.

Le roman 1984 de George Orwell illustre très bien ce propos.

Cette définition générale de la vie privée explique *pourquoi* il est important de faire respecter le droit à la vie privée. En revanche, elle n'explique pas *quand* une violation au droit à la vie privée intervient. D'après Klitou (2014), le *quand* est spécifique au domaine d'application de la vie privée. Cette section, et plus généralement cette thèse, s'intéresse au domaine des données personnelles hébergées dans un système informatique.

2.2.1 Le droit à la vie privée : la préservation des données personnelles

Avec l'informatique en nuage arrive une augmentation des technologies qui utilisent les données personnelles des individus. Cette augmentation est principalement due à la forte valeur marchande que représentent les données personnelles. Elle engendre, *de facto* un nouveau domaine d'application pour la vie privée. Ce domaine se concentre sur la *préservation des données personnelles* (informational pricacy – Klitou, 2014) hébergées dans un système informatique.

De manière générale, une donnée personnelle est définie comme toute information relative à un individu qui permettrait de l'identifier. En particulier, par l'emploi d'un numéro d'identification (*ex.*, son numéro de sécurité sociale), ou par un ou plusieurs facteurs physiques, culturels et sociaux (*ex.*, son nom, sa date de naissance, ses empreintes digitales, *etc.*). Mais aussi, par des actions quotidiennes telles que les trajets en voiture et le régime alimentaire. Appliquée à l'informatique en nuage, une donnée personnelle peut revêtir des formes encore plus diverses. On donnera en exemples la sauvegarde des transactions d'un client sur un site marchand, la collecte par un tiers de mots clefs dans des courriels et discussions privées, ou l'enregistrement du comportement (clics) d'un internaute. L'ensemble de ces données personnelles forme l'identité, en partie privée, de l'individu.

On sait aujourd'hui que Facebook sauvegarde à chaque nouvelle connexion de l'un de ses utilisateurs, son adresse IP, l'emplacement de chacun de ses clics, son temps passé par page, etc. Ceci pour créer un profil très précis (Assange et collab., 2012).

Bien que souvent très pratique, héberger ses données dans un système informatique dépossède l'utilisateur de leur gestion. La préservation des données personnelles est alors délégué à l'application. Malheureusement, dans un monde comme le nôtre, où toutes les données personnelles sont conservées (pour toujours) et analysées, l'utilisateur devient otage de la moindre action qu'il fait.

Les applications qui contrôlent ces données personnelles ont un pouvoir sur le propriétaire des données. Qu'elles l'exercent ou non, les applications des systèmes informatiques traitent ces données avec beaucoup de négligences (*cf.* exemples de l'introduction §1.1.1). Les systèmes informatiques violent par la même occasion le droit de l'utilisateur à préserver ses données personnelles.

2.2.2 Violer la préservation des données personnelles

La violation au droit à la vie privée est divisible en quatre grandes catégories (Solove, 2006) : la *collecte* de données personnelles, le *traitement* de données personnelles, la *diffusion* de données personnelles et l'*interférence* sur les données personnelles.

Concrètement, pour la préservation des données personnelles dans un système informatique, la violation prend la forme de :

— L'hébergement des données personnelles d'un individu sans son consentement, sa connaissance ou notification quelconque (*collecte*).
— L'analyse des données personnelles d'un individu sans son consentement, sa connaissance ou notification quelconque (*traitement*).
— La mise à disposition d'un tiers non stipulé (*diffusion*).

— L'absence de moyen pour l'individu de supprimer, rectifier et compléter sa donnée (*interférence*).

Pour qu'un système informatique préserve les données de ses utilisateurs et donc, respecte leur droit à la vie privée, il doit empêcher ces quatre formes de violation. Une tâche qui est du domaine de la sécurisation du système informatique.

2.3 Sécuriser le nuage pour préserver les données personnelles

Pour rendre plus sûr le nuage et empêcher les violations, il faut le sécuriser. De manière générale, la sécurisation consiste à analyser et comprendre les risques qui peuvent apparaître pour les utilisateurs d'un système. Puis, prendre les moyens nécessaires et appropriés pour limiter ces risques. Et enfin, s'assurer que les moyens pris protègent correctement les utilisateurs du système. Ces trois phases, indispensables à un système sécurisé, sont regroupées sous trois piliers qui sont la *politique*, le *mécanisme* et l'*assurance*.

Cette section décrit d'abord ces trois piliers. Puis elle montre que la préservation des données personnelles hébergées dans le nuage est un cas d'utilisation de ces trois piliers.

2.3.1 Politique, mécanisme et assurance : les piliers d'un système sécurisé

Trois piliers sont nécessaires à un système informatique sécurisé (Lampson, 2004). Le premier est le pilier *politique*. Il spécifie les attentes quant à la sécurisation du système. Le second est le pilier *mécanisme*. Il implémente les politiques et fait en sorte qu'elles soient respectées sur le système. Le troisième est le pilier *assurance*. Il garantit que le système est sécurisé.

Butler W. Lampson (2004) fait un parallèle intéressant entre les piliers de la sécurisation (politique, mécanisme, assurance) et les piliers du génie logiciel (spécification, implémentation, correction) et précise, avec amusement, qu'il est habituel dans le domaine de la sécurité de donner de nouveaux noms à des concepts qui nous sont familiers.

Pour comprendre ce qu'est un système sécurisé, il convient d'étudier chacun de ses piliers.

Politique : le pilier pour la spécification

Une politique *décrit*, à la manière d'une loi en droit, une règle à laquelle le système informatique et ses utilisateurs doivent se soumettre sous peine de sanctions. Elle permet aux utilisateurs du système de spécifier leurs besoins de sécurisation pour protéger leurs ressources d'un individu malicieux. En revanche, la politique ne décrit pas comment imposer la règle sur le système.

Toutes les politiques de sécurité s'expriment soit comme une *permission*, soit comme une *interdiction*. Par exemple, la politique qui limite, à un groupe d'administrateurs, la lecture d'une ressource, s'exprime : "Seuls les individus du groupe administrateur *ont le droit* de lire la ressource" à la manière d'une permission, ou "Tous les individus autres que ceux du groupe administrateur *n'ont pas le droit* de lire la ressource" à la manière d'une interdiction.

Lorsque les gens parlent de sécurisation, ils s'expriment généralement avec l'interdiction, car elle fait apparaître l'individu malicieux. Dans la pratique, ce-

pendant, les experts en sécurités préfèrent la forme qui ajoute des permissions, car elle est plus sûre. Dans cette forme, tous les usages sont interdis par défaut et la permission est là pour authoriser un usage. Par conséquent, si une politique vient à manquer ou n'est pas assez précise, le système sera plus contraint. À l'inverse, avec la forme d'interdiction, tous les usages sont permis sur une ressource et la politique restreint cette permission. Du coup, avec une politique manquante ou pas assez précise, le système sera moins contraint et attribuera plus de permission que souhaité aux utilisateurs.

Mécanisme : le pilier pour l'implémentation

Alors que le pilier politique *décrit*, à la manière d'une loi, les règles auxquelles doit se soumettre le système informatique. Le pilier mécanisme *fait respecter*, à la manière d'une police, ces règles sur le système. L'intention est de protéger les utilisateurs contre les vulnérabilités du système.

Dans un système informatique, il existe trois formes de vulnérabilités (Lampson, 2004) :

Le terme pirate informatique employé ici n'est pas correct. Malheureusement pour nous, la langue française ne fait pas la distinction entre le cracker (un pirate informatique mal intentionné) et le hacker (un pirate informatique qui aime avoir une compréhension approfondie du fonctionnement interne d'un système).

Le *pirate informatique* (cracker) est un individu qui cherche à accéder au système sans en avoir l'autorisation (Malkin, 1996). Le pirate informatique use de nombreux outils pour arriver à ses fins : cheval de Troie, virus mais aussi exploit de programmes bogués (*cf.* vulnérabilité suivante).

Le programme bogué est un programme informatique utilisé dans le système dont l'implémentation est souvent complexe ou peu rigoureuse. Ceci entraîne des anomalies de fonctionnement qui sont ensuite utilisées par les pirates ou agents malicieux. Ces bogues sont à l'origine d'attaques telles que la fuite d'information ou l'appel d'une *interface système* (shell) grâce au célèbre "buffer overflow" (Aleph1, 1996) par exemple.

L'agent malicieux est un utilisateur ou un programme dont les intentions sont malveillantes. Il profite de sa position dans le système et de la crédibilité des autres agents pour outrepasser les politiques.

Pour se défendre face à ces vulnérabilités et faire respecter les politiques, un mécanisme doit adopter, au moins, l'une des deux stratégies de défense suivantes :

Une stratégie proactive qui prévient la violation d'une politique. Elle *empêche* les pirates informatiques et agents malicieux d'interférer avec le système. C'est un mécanisme de défense qui ne laisse pas une violation se produire, mais qui, le cas échéant, ne propose aucune solution pour résoudre la violation. La stratégie est souvent difficile à mettre en place car l'implémentation du mécanisme doit être correcte.

Une stratégie réactive qui laisse l'individu malicieux violer la politique. Un audit est régulièrement fait pour vérifier si une violation a eu lieu. Si oui, une enquête est menée pour condamner l'individu malicieux. Cette stratégie de défense équivaut à notre système policier. Elle *condamne* les pirates informatiques et agents malicieux qui interfèrent avec le système. Elle tente également de *remédier* à la violation par des corrections.

Assurance : le pilier pour la confiance

Implémenter un mécanisme de défense de politiques est important. Mais, à quoi sert ce mécanisme s'il ne défend pas *correctement* le système de ses vulnérabilités ? Le résultat est un système poreux qui laisse pénétrer les individus malicieux et qui permet la violation des politiques. C'est pourquoi, dans un système sécurisé, il est important d'avoir l'assurance que le mécanisme de défense est correct. Ainsi, les utilisateurs peuvent avoir confiance dans le système.

***Trusted* vs. *Trustworthy*.** Cette assurance porte sur tous les composants responsables de la sécurisation du système. Il faut distinguer plusieurs niveaux d'assurances (Hoffman et collab., 2006).

Le plus faible niveau *fait confiance aux composants responsables de la sécurisation*. En d'autres termes, le système fait l'hypothèse que les composants de sécurisation se comportent correctement. Ce niveau peut être vu comme pas d'assurance du tout.

À l'opposé, le plus fort niveau *requiert que tous les composants responsables de la sécurisation soient corrects*. Ceci se fait en deux phases distinctes. La première phase s'assure que la spécification de chaque composant respecte la politique. La seconde phase garantit que l'implémentation du composant respecte sa spécification. Ce niveau correspond à une vérification formelle des composants de sécurisation.

Le niveau le plus fort est toujours préférable. Toutefois, le développeur n'a pas d'autres choix que de faire confiance aux composants de sécurisations lorsqu'ils sont trop complexes (*ex.*, un composant qui empêche la seconde dissémination des données).

2.3.2 Application à la préservation des données personnelles dans le nuage

Il est important de ne pas confondre la préservation des données personnelles (information privacy) dont un cadre est donné ci-après. Et la sécurité des données (data security) un domaine plus vaste qui considère, entre autres, la pérennité des données.

Préserver les données personnelles hébergées dans le nuage est un cas d'utilisation de la sécurisation. Pour s'en rendre compte, il suffit de reprendre les éléments mis en exergues par les sections précédentes. Puis d'extraire une politique générale pour la préservation des données personnelles.

Tout d'abord, la section sur le nuage (§2.1) donne les caractéristiques de celui-ci, soit un système informatique fait de plusieurs agents qui communiquent entre eux. Il existe quatre formes d'agents : (1) la cliente qui est la propriétaire des données (2) l'application du nuage qui est proposée en tant que service SaaS (3) les fournisseurs d'outils nuagiques au niveau PaaS et (4) les fournisseurs d'infrastructures au niveau IaaS. Par conséquent, une première instanciation de la politique générale pour la préservation des données personnelles peut être *l'obligation pour n'importe lequel des agents SaaS, PaaS et IaaS de préserver la vie privée de la cliente*.

Ensuite, la section sur la vie privée (§2.2) montre que, dans un système informatique, les données personnelles doivent ce protéger contre quatre formes de violations : collections, traitement, diffusion et interférence. Par conséquent, la politique générale pour la préservation des données personnelles se raffine

en *l'obligation pour n'importe lequel des agents SaaS, PaaS et IaaS de non collecter, non traiter, non diffuser et non interférer sur les données personnelles de la cliente*.

L'exemple de l'agenda personnel en ligne (décrit rapidement lors de l'introduction – *cf.* §1.1.1, *fig.* 1) justifie le besoin d'une telle politique générale. Cette exemple témoigne de deux réalités.

La première, positive, montre que les développeurs peuvent facilement réaliser des applications avec des propriétés très intéressantes, simplement en composant les couches du nuage.

La seconde, négative, montre comment le nuage provoque inévitablement la délégation des données personnelles de ses utilisateurs (ici, un rendez-vous pris par Alice) aux autres acteurs du nuage (les développeurs et fournisseurs). Dès lors, rien n'empêche ces acteurs s'ils sont malicieux, de violer le droit de l'utilisateur à préserver ses données personnelles. Soit la politique finale :

> *"L'obligation pour n'importe lequel des agents SaaS, PaaS et IaaS de non collecter, non traiter, non diffuser et non interférer sur les données personnelles d'Alice."*

Par conséquent, et en accord avec les piliers de la sécurisation (§2.3.1), il faut mettre en place des mécanismes qui fassent respecter cette politique générale. Puis, il faut avoir l'assurance que ces mécanismes soient corrects. La suite fait l'état de l'art des mécanismes les plus employés dans l'approche de protection intégrée de la vie privée (Cavoukian, 2011) et explicite qu'elles sont les assurances apportés pas ces mécanismes.

2.4 Sécuriser par l'approche de protection intégrée de la vie privée

Le dessein de l'approche de *protection intégrée de la vie privée* (Privacy-by-Design ou PbD – Cavoukian, 2011 ; Spiekermann, 2012) est d'*empêcher* les violations au droit à la vie privée, dans un système. L'approche PbD est proposée comme une méthodologie qui doit être suivie par les développeurs de logiciels. Elle exige des développeurs qu'ils considèrent le respect de la vie privée de leurs clients lors de la conception du logiciel.

L'approche se veut proactive (§2.3.1) pour éviter toute violation d'une politique. Pour se faire, elle dispose de techniques dites de *renforcement du respect de la vie privée* (Privacy-Enhancing Technologies), dont les bien connues techniques de cryptographie.

L'exemple de la technique de cryptographie la plus répandue est le *chiffrement symétriques* (Menezes et collab., 1996) qui permet de chiffrer et déchiffrer un message à l'aide d'un mot clef. Mais, dans l'approche PbD, beaucoup d'autres techniques sont utilisées, comprenant, le *chiffrement homomorphe* (homomorphic encryption – Gentry, 2009) qui permet d'effectuer des calculs sur une donnée chiffrée, sans la déchiffrer. La *preuve à divulgation nulle de connaissance* (zero-knowledge proof – Goldwasser et collab., 1989 ; Danezis et Livshits, 2011) qui donne la possibilité de traiter de manière privée les données personnelles collectées par des capteurs intelligents. La fragmentation de données (Aggarwal et collab., 2005 ; di Vimercati et collab., 2013) qui supporte,

sans chiffrement, la sauvegarde des données personnelles sur le nuage de manière privée. La notion *d'indistinguabilité* (differential privacy – Dwork et collab., 2006 ; Dwork et Roth, 2014) qui permet d'interroger une base de données statistique sans révéler d'informations sur ses occupants. Le protocole de *calculs sécurisés entre tiers* (secure multi-party computation – Yao, 1982 ; Kerschbaum, 2009) qui rend possible et de manière privée les calculs sur des données personnelles. Le filtre de Bloom (1970) pour tester l'appartenance d'un élément dans un ensemble et utilisé par Google Chrome (Erlingsson et collab., 2014) pour collecter des statistiques de l'utilisateur, *etc*.

Concrètement, l'approche PbD repose sur ces techniques pour protéger la vie privée des clients en supprimant ou en rendant inintelligible les données personnelles à autre que qui de droit.

Cette section définit ce qu'est l'approche de protection intégrée de la vie privée grâce à un exemple simple d'une application de dialogue privée en ligne. Elle analyse ensuite trois techniques régulièrement utilisées dans l'approche PbD qui sont les calculs côté client, le chiffrement et la fragmentation verticale. Par cette analyse, la section conclut sur les limites actuelles de l'approche.

2.4.1 Concevoir une application avec l'approche de protection intégrée de la vie privée

Dans l'approche de protection intégrée de la vie privée (PbD), la préservation des données personnelles est l'affaire du développeur. En d'autres termes, c'est au développeur de logiciels de concevoir une application qui est respectueuse des données personnelles de ses clients. De ce fait, le développeur *ne considère pas son application comme étant de confiance*, au contraire ! Ceci le force à garantir la correction de son application par rapport aux vulnérabilités du nuage.

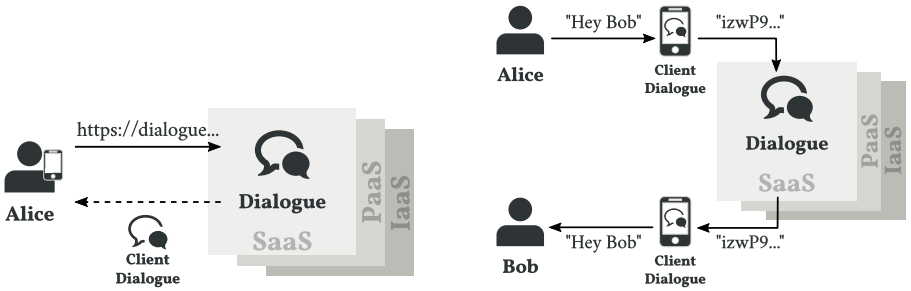
Pour se faire, en plus du panel de techniques mis à sa disposition, le développeur doit répondre aux deux questions suivantes (Van Blarkom et collab., 2003) :

1. Quelle est la donnée personnelle dans l'application ? Ceci dans le dessein d'empêcher qu'elle soit collectée/traitée/diffusée/interférée par un tiers.
2. Quelle technique de cryptographie peut protéger la donnée personnelle sans que l'application perde en fonctionnalité ? Ceci dans le dessein d'employer cette technique durant la conception de l'application et ainsi empêcher l'application et les participants malveillants de collecter/traiter/-diffuser/interférer sur la donnée personnelle.

Qu'en est-il pour une application de dialogue privée en ligne ? Du point de vue des fonctionnalités, l'application permet l'échange, instantané et de manière privée, de messages entre plusieurs personnes par le réseau Internet. Elle est composée d'une application cliente et d'un serveur. L'application cliente envoie et reçoit les messages. Elle est fournie en tant que SaaS pour disposer de la disponibilité. L'application serveur, quant à elle, distribue les messages entre les participants d'une discussion. Elle est exécutée par le PaaS pour profiter de l'élasticité.

Cet exemple se base sur une application concrète et open source nommée Cryptocat[4]. Le principe est le même pour la célèbre application Telegram messenger[5].

4 crypto.cat
5 telegram.org



(a) *Récupération de l'application cliente par Alice.*

(b) *Envoi d'un message privé d'Alice vers Bob.*

FIGURE 2 – *Application de dialogue privé en ligne, sécurisée par l'approche PbD.*

La première question de la méthodologie PbD demande au développeur de l'application d'identifier la donnée personnelle. Ici, il s'agit du message à transmettre.

Fort de cette réponse, le développeur de l'application de dialogue passe ensuite à la deuxième question. Il sait, grâce aux constatations faites dans la section sur la sécurisation du nuage (§2.3.2), que le message à transmettre doit être protégé de tous agents autres que les participants du dialogue. En l'occurrence, l'application serveur, les couches SaaS/PaaS/IaaS et les pirates informatiques. Le développeur de l'application cherche donc une technique qui empêche la collecte/traite/diffusion/interférence d'un message par ces agents. Il trouve comme solution le protocole de *messagerie confidentielle* (Off-the-Recode Messaging ou OTR – Borisov et collab., 2004) qui offre un chiffrement adapté pour les messages de dialogue en ligne.

Dans le protocole OTR, un message est chiffré côté client. Une particularité que le développeur doit prendre en compte lors de la conception de l'application. Il conçoit donc la partie cliente, ou tout du moins la partie chiffrement/-déchiffrement du message, avec JavaScript. JavaScript est un langage de programmation dont un programme est aisément hébergé sur le nuage, mais dont l'exécution s'effectue chez le client (MDN, 2015). Ainsi, l'application continue à profiter de la disponibilité du SaaS tout en permettant le chiffrement/déchiffrement du message sur le terminal du client.

La figure 2 représente deux cas d'interaction avec l'application de dialogue privé. Tout d'abord, la figure 2a montre l'établissement de la connexion entre Alice et l'application. Alice se connecte à l'application via une adresse URL et reçoit en réponse le programme de l'application cliente. L'application cliente est dorénavant directement exécutée sur le terminal d'Alice sans passer par le nuage.

Dans la figure 2b, Alice utilise, par la suite, l'application pour dialoguer de manière privée avec Bob. Alice rédige le message "Hey Bob" qui est chiffré par l'application cliente grâce au protocole OTR et est transmis sur le réseau à l'application serveur. Le message se retrouve donc dans le nuage. Heureusement, aucun acteur autre qu'Alice et Bob ne peuvent lire le message puisqu'il est chiffré. Le serveur distribue ensuite le message chiffré jusqu'au terminal de Bob. Celui-ci utilise l'application cliente pour déchiffrer le message et afficher "Hey Bob" sur son écran.

Cet exemple montre que l'approche PbD permet de concevoir une application qui est respectueuse des données personnelles de ses clients tout en pro-

fitant des avantages du nuage. Toutefois, cet exemple est possible car le protocole OTR est spécialement conçu pour ce type d'application. En revanche, qu'en est-il pour l'agenda personnel en ligne ? Est-ce qu'il existe une unique technique pour implémenter l'application ?

De même, cet exemple montre que choisir une technique n'est pas suffisant. Il faut, en plus, correctement employer la technique sous peine de mal faire appliquer la politique générale pour la préservation des données personnelles. Ici, une erreur de conception où l'application serait entièrement écrite dans un langage exécuté côté serveur (comme en PHP par exemple) remettrait en cause la préservation des données personnelles. Et ceux-ci, même si le protocole OTR était utilisé, car le message personnel transiterait en claire sur le réseau avant d'être chiffré par l'application. Cette vérification correspond au pilier assurance (§2.3.1) des piliers d'un système sécurisé.

Le langage de programmation PHP (Lerdorf et collab., 2006).

2.4.2 Les contraintes de confidentialités

Répondre à la question "Quelle est la donnée personnelle dans l'application ?" est l'une des deux étapes primordiales à la réalisation d'une application sécurisée par l'approche PbD. Cependant, la réponse n'est pas toujours aussi simple que dans l'application de dialogue en ligne. Dans cet exemple, l'application propose une seule fonctionnalité qui est d'échanger un message privé à travers le nuage. C'est donc naturelle pour le développeur de l'application de désigner le message comme donnée personnelle. En revanche, qu'en est-il pour l'agenda personnel en ligne (*cf.* §1.1.1) ? Est-ce que, parce que cette application héberge des rendez-vous dans le nuage, le développeur doit désigner un rendez-vous, dans son ensemble, comme une donnée personnelle ?

La suite stipule que le développeur de l'application sauvegarde les rendez-vous d'Alice dans la base de données sous la forme d'un triplet "date, nom, adresse" qui représente la date du rendez-vous, le nom du contact et le lieu du rendez-vous. Dans cette représentation, le nom est évidemment une information sensible, puisqu'un individu malveillant qui aurait cette information devinerait les contacts d'Alice. En revanche, la date et l'adresse en eux-mêmes ne représentent pas une information sensible. Un individu malveillant qui verrait juste les dates ne violerait pas la vie privée d'Alice. Idem pour les adresses. Par contre, un individu malveillant en possession de ces deux informations serait capable de localiser précisément Alice.

Le développeur de l'application spécifie donc deux contraintes sur un rendez-vous : $\{nom\}$ et $\{date, adresse\}$. On parle de contraintes de confidentialités (Aggarwal et collab., 2005). Elles spécifient quelles sont les données personnelles dans l'application.

Spécification d'une contrainte de confidentialité

Pour spécifier les contraintes de confidentialités sur une donnée personnelle, le plus simple est de représenter la donnée comme un n-uplet de *valeurs d'attributs* (di Vimercati et collab., 2013). Puis, d'exprimer les contraintes sur les noms des *attributs*.

FIGURE 3 – *Exemple d'un rendez-vous pris par Alice.*

```
{
 "date":
   new Date('2015-11-16T09:15:00
    '),
 "nom": "Bob",
 "adresse": Bureau
}
```

(a) *Rendez-vous d'Alice défini au format JSON (Bray, 2014) pour le transmettre sur le réseau.*

date	nom	adresse
16/11/2015 à 9h15	Bob	Bureau
16/11/2015 à 12h30	Claire	Parc
19/11/2015 à 9h15	Bob	Bureau

(b) *Rendez-vous d'Alice défini avec un schéma relationnel pour les sauvegarder dans une base de données.*

Un *attribut* est un nom et potentiellement un type. Une *valeur d'attribut* est un nom d'attribut associé à une valeur. Si l'attribut est accompagné d'un type, alors la valeur est de ce type. Enfin, un n-uplet est fait de plusieurs valeurs d'attributs dans lequel tous les éléments ont un nom différent.

Avec cette représentation, le développeur n'a plus qu'à utiliser les noms d'attributs du n-uplet pour spécifier les contraintes de confidentialités. Ceci offre un cadre général applicable à n'importe quelle donnée non récursive.

Par exemple, un rendez-vous pris par Alice le 16 novembre 2015 à 9 heures 15, avec Bob, au bureau, est représenté par le 3-uplets de valeurs d'attributs :

$$(11/16/2015 \text{ à } 9\text{h}15, \text{“}Bob\text{”}, Bureau)$$

ainsi que par ses fonctions d'accès qui définissent le nom et le type des attributs :

$$\begin{aligned} date &: (Date, Texte, Lieu) \to Date \\ nom &: (Date, Texte, Lieu) \to Texte \\ adresse &: (Date, Texte, Lieu) \to Lieu \end{aligned}$$

Grâce à ces deux informations, le développeur construit le 3-uplet d'attributs :

$$(date, nom, adresse)$$

et spécifie la contrainte $\{nom\}$ pour qu'un individu malveillant n'infère pas le nom des contacts d'Alice. Puis, spécifie la contrainte $\{date, adresse\}$ pour qu'un individu malveillant ne localise pas Alice.

Cette spécification a du sens, quel que soit le format des données considéré. Que ce soit pour un rendez-vous transmis sur le réseau (*code* 3a). Ou pour un rendez-vous sauvegardé dans une base de données (*fig.* 3b).

Il est important de remarquer qu'il existe deux types de contrainte de confidentialité. La première est la *contrainte singleton*. Elle contient un seul attribut (*ex.*, $\{nom\}$). Ceci signifie que la valeur de son attribut est privée. La seconde est la *contrainte associative*. Elle contient une association d'attributs (*ex.*, $\{date, adresse\}$). Ceci signifie que c'est l'association des valeurs qui est privée. C'est une distinction importante, car le développeur peut envisager différentes techniques de cryptographie en fonction de la contrainte.

2.4.3 Les techniques de cryptographie

Une contrainte de confidentialité modélise *l'aspect personnel d'une donnée sur le plan de sa structure*. En revanche, elle ne modélise pas le *type* de vie privée qui est nécessaire à cette donnée.

Le type de vie privée dépend des besoins de l'application. Ainsi, un site marchand se contente généralement de rendre les transactions commerciales inintelligibles. En revanche, une application d'échange de messages privés en ligne requière, en plus des messages inintelligibles, une propriété de *confidentialité persistante* (forward secrecy – Diffie et collab., 1992) pour ne jamais compromettre la confidentialité des messages passés.

Dans cette thèse nous nous limitons à la propriété de la confidentialité comme type de vie privée.

Satisfaire les contraintes de confidentialités en fonction du type de vie privée est le dessein des techniques de cryptographie. La littérature scientifique foisonne de ces techniques et l'introduction de cette section en énumère les plus importantes. La suite décrit trois techniques pour préserver la confidentialité des données personnelles. Ces techniques sont le calcul côté client, le chiffrement et la fragmentation. Chaque description se veut non technique et présente les caractéristiques en terme de types de sécurités assurés, de performances et de fonctionnalités. Ce sont également les trois techniques considérées dans cette thèse.

2.4.3.1 Le calcul côté client

Le calcul côté client permet à une application du nuage (*ex.*, l'agenda) de réaliser un calcul qui implique des données personnelles, sans violer le droit du client à préserver ses données personnelles. Pour se faire, le client (*ex.*, Alice) conserve ses données personnelles sur son terminal. Elle utilise ensuite son terminal pour réaliser le calcul sensible et retourner le résultat du calcul à l'application du nuage. Par conséquent, seul le résultat du calcul est connu par l'application tandis que les données personnelles sont préservées.

La confidentialité des données personnelles est préservée à condition que le résultat ne permette pas d'inférer les données personnelles.

L'approche du calcul côté client constitue une technique fondamentale pour le *compteurs communicants* (smart meters). Un compteur communicant est un compteur qui mesure en temps réel, et de manière détaillée la consommation électrique d'un foyer. L'idée du calcul côté client est de conserver chez le client les données mesurées puisqu'elles permettent d'inférer son mode de vie. Puis, de laisser le client calculer le montant de sa facture et retourner le résultat à la compagnie d'électricité.

Transmettre au client la fonction qui réalise le calcul est assez simple. Par exemple, le cadriciel Akka[6], qui permet d'écrire des applications concurrentes en Scala, sérialise et transmet sur le réseau une fonction avec son contexte d'exécution grâce à la bibliothèque Kryo[7].

Le langage de programmation Scala (Odersky et collab., 2008).

En revanche, lorsque le client retourne une réponse (ici, le montant de sa facture), l'application est face à un problème d'*intégrité* (Danezis et Livshits, 2011). Le client peut "mentir" sur le résultat du calcul. Il faut donc être capable de vérifier l'intégrité du résultat sans transmettre les données personnelles.

6 akka.io
7 github.com/EsotericSoftware/kryo

La preuve à divulgation nulle de connaissance

Une technique de cryptographie appelée *preuve à divulgation nulle de connaissance* (zero-knowledge proof – Goldwasser et collab., 1989) permet à un client de prouver à l'application l'intégrité de son calcul sans révéler ses données personnelles. Cette technique a été reprise par Danezis et Livshits (2011), pour l'appliquer au nuage.

Intuitivement, l'approche est similaire à celle de la très connue *signature numérique* (digital signature), offerte par un chiffrement asymétrique (Menezes et collab., 1996). Pour rappel, avec la signature numérique, un utilisateur peut signer son message et garantir que son contenu n'a subi aucune altération (*intégrité*). C'est une approche qui repose sur deux clefs (une privée, secrète et une publique, diffusée) conjointement avec un algorithme de signature et un algorithme de vérification de signature. Concrètement :

1. L'utilisateur calcule la signature (s) de son message (m) grâce à sa clef privée (sk) et l'algorithme de signature ($sign$).

$$s = sign(sk, m)$$

2. Une personne qui veut vérifier l'intégrité du message (m) utilise la signature (s), la clef publique du signataire (pk) et l'algorithme de vérification ($verify$).

$$verify(pk, m, s) \tag{1}$$

L'algorithme de vérification (*eq.* 1) s'assure de l'intégrité du message sans faire appel à la clef privée. En d'autres termes, sans faire appel aux données personnelles de l'utilisateur.

L'approche est identique pour la preuve à divulgation nulle de connaissance. Dans un premier temps le client génère une preuve de l'intégrité de son calcul. Puis, l'application du nuage vérifie cette preuve, sans avoir à lire les données personnelles :

1. Le client applique la fonction (f) pour faire le calcul sur une donnée personnelle (d). Puis, il utilise un algorithme de génération de preuve (zkp) pour générer une preuve (p) de l'intégrité de son calcul. L'algorithme prend en argument la fonction (f), le résultat du calcul ($f(d)$), la donnée chiffrée ($enc(d)$) et une valeur de gage ($commit(d)$).

La valeur de gage sert à générer une preuve "sans interaction", c'est-à-dire que l'algorithme de génération ne requiert pas d'interagir avec l'application, ce qui est préférable pour le nuage.

$$x = f(d)$$
$$p = zkp(f, enc(d), commit(d), x)$$

2. Pour vérifier l'intégrité du calcul ($f(d)$), l'application du nuage utilise la preuve (p) fournie par le client, la fonction (f), le résultat du calcul (x), la donnée chiffrée ($enc(d)$), la valeur de gage ($commit(d)$) et un algorithme de vérification ($verify$).

$$verify(p, f, x, enc(d), commit(d)) \tag{2}$$

À la manière de la vérification d'une signature digitale (*eq.* 1), la vérification d'une preuve (*eq.* 2) assure à l'application du nuage de l'intégrité du calcul, sans

faire appel aux données personnelles du client. Seule une forme inintelligible des données est requise ($enc(d)$ et $commit(d)$).

La littérature offre de nombreuses implémentations efficaces des preuves à divulgation nulle de connaissance sans interaction. Elles prouvent : l'intégrité de l'égalité entre plusieurs valeurs (Schnorr, 1991); l'intégrité de l'application d'opérations élémentaires (Camenisch et Michels, 1999; Camenisch et collab., 2008); l'intégrité d'un prédicat fait de AND, OR et NOT appliqué sur une valeur (Brands, 1997), *etc*. Chaque implémentation propose ses propres algorithmes de génération de preuve et de vérification. Mais, pour toutes ces implémentations, la valeur de gage peut toujours être celle calculée à partir du schéma de Pedersen (1991). Par conséquent, l'application du nuage peut retenir cette information pour plus d'efficacité (Danezis et Livshits, 2011).

Performances des preuves à divulgation nulle de connaissance sans interaction et discussion

Fournet et collab. (2013) proposent le langage de requête *ZQL* pour réaliser des calcules côté client tout en vérifiant l'intégrité des résultats. Le langage possède un compilateur qui traduit une expression *ZQL* et un protocole de preuve à divulgation nulle de connaissance sans interaction (implémenté en F#). L'article propose trois requêtes différentes qui exploitent soit des opérations simples (soustraction et multiplication), soit des recherches dans un n-uplet, ou soit des calculs d'agrégation de valeurs. Pour chaque requête, l'article donne la taille de la preuve ainsi que les temps de génération et de vérification de cette preuve. La taille de la preuve est de l'ordre du kilo-octet. Elle est linéaire en fonction de la taille du calcul. Les temps de génération et de vérifications prennent plusieurs secondes.

Le langage de programmation F# (Syme et collab., 2012).

Toutefois, une preuve est nécessaire que lorsque l'intégrité du calcul est en jeux. C'est-à-dire, *que lorsque l'application a besoin de réutiliser le résultat du calcul effectué par le client*. En revanche, si l'application n'a pas besoin du résultat, alors le client n'a pas besoin de générer de preuve à divulgation nulle de connaissance.

2.4.3.2 Le chiffrement

Le chiffrement code une donnée avant son externalisation de telle manière que seules les personnes qui y sont autorisées puissent la lire. Le chiffrement se fait au moyen d'une clef, également requise lors du déchiffrement (opération inverse). Il existe plusieurs techniques de chiffrement (asymétrique, fondé sur l'identité, par attributs…). Cette section se concentre sur les chiffrements *symétrique* et *homomorphe*.

Lorsqu'un attaquant parvient à trouver le message en clair sans posséder la clef de déchiffrement, on parle alors de décryptage.

Le chiffrement symétrique

Le chiffrement symétrique (Menezes et collab., 1996) est un chiffrement qui utilise la même clef pour chiffrer et déchiffrer une donnée. Le fait de compromettre la clef casse la sécurité du chiffrement et par conséquent la protection de la donnée. C'est pourquoi la sécurité du chiffrement réside dans la capacité ou non à compromettre la clef.

Code 4 – *Temps de chiffrement/déchiffrement d'un fichier de 1 Go avec l'algorithme AES et une clef de 256 bits.*

```
[rcherr@nixos:tmp]$ time openssl aes-256-ecb -e -pass pass:passwd\
> -in nixos-16.03-linux.iso -out file.aes

real  0m0.804s
user  0m0.431s
sys   0m0.330s

[rcherr@nixos:tmp]$ time openssl aes-256-ecb -d -pass pass:passwd\
> -in file.aes -out nixos-16.03-linux.iso

real  0m0.857s
user  0m0.429s
sys   0m0.422s
```

Une clef de 512 bits est probablement sûre face à n'importe quelles technologies Alien !

À titre d'exemple, l'*Institut national des normes et de la technologie* (National Institute of Standards and Technology – NIST) a choisi l'algorithme AES (Daemen et Rijmen, 1999) comme standard de chiffrement symétrique. À ce jour, le seul moyen de compromettre une clef, hormis le fait de voler celle-ci, est d'appliquer une attaque par force brute qui teste tous les candidats possibles et inimaginables jusqu'à obtenir le résultat correct. La solution est donc de choisir une clef suffisamment grande telle qu'il n'y ait pas assez d'énergie dans l'univers pour tester tous les candidats. Le NIST estime qu'avec la technologie actuelle, une clef d'une taille de 128 bits est sûre. La communauté de la sécurité s'accorde à dire qu'une clef de 256 bits est sûre par rapport à n'importe quelles technologies futures (*c.-à-d.*, augmentation de la puissance de calculs et ordinateurs quantiques).

Le chiffrement/déchiffrement AES est également rapide. Par exemple, il faut moins d'une seconde au Pentium 5 et 8 Go de RAM utilisé pour écrire cette thèse pour chiffrer/déchiffrer un fichier de 1 Go (*code* 4).

La littérature propose d'autres algorithmes comme Twofish (Schneier et collab., 1999) et A5 (Golić, 1997). Ces algorithmes sont répartis en deux types de chiffrement symétrique. Le chiffrement par bloc (AES, Twofish) et le chiffrement par flot (A5). Puisque cette thèse s'intéresse plus aux fonctionnalités qu'à la technicité, elle ne fait pas la différence entre ces deux types de chiffrement symétriques. Il est toutefois bon de noter que lorsque la taille de la donnée à chiffrer est connue, comme pour un message, un fichier ou le protocole HTTP, c'est le chiffrement symétrique par bloc qui est préféré. À l'inverse, quand le volume de la donnée n'est pas fixé, comme pour une communication mobile (GSM), c'est le chiffrement symétrique par flot qui est employé.

Historiquement, le chiffrement symétrique est la première approche à être adoptée pour protéger les données personnelles dans une base de données du nuage (Hacigümüs et collab., 2002). Dans cette approche, le client chiffre ses données avant de les héberger dans le nuage, à l'instar des messages dans l'application de dialogue privé en ligne (§2.4.1). Malheureusement, dans le cas des bases de données du nuage, le protocole qui s'en suit pour exécuter une requête est trop coûteux.

Une requête est exécutée en rapatriant les données nécessaires, dans leur forme chiffrée, depuis la base de données vers le client. Le client déchiffre ensuite les données et enfin, exécute la requête de son côté.

Avec cette approche, une requête qui exige un très grand nombre de n-uplets, rapatrie un très gros volume de données chez le client. Ce qui requiert de solliciter trop fortement le réseau pour être envisageable. Déchiffrer chaque n-uplet et faire la requête est également une charge de calcul très importante pour le client.

Ceci fait du chiffrement symétrique une solution très inefficace pour réaliser des calculs, malgré le fait qu'elle soit très sécurisante et très utile pour préserver les données personnelles du point de vue du stockage. La littérature propose une autre forme de chiffrement, nommé le chiffrement homomorphe, pour répondre au problème du calcul.

Le chiffrement homomorphe

Le chiffrement homomorphe (Gentry, 2010) permet de faire un calcul simple, directement sur la donnée chiffrée. Avec ce chiffrement, dans un premier temps, le client héberge sa donnée personnelle dans le nuage de manière chiffrée. La donnée personnelle est inintelligible, ce qui empêche les participants malveillants de collecter/traiter/diffuser celle-ci. Toutefois, grâce au chiffrement homomorphe, le client peut autoriser le nuage à appliquer des opérations sur la donnée chiffrée. En d'autres termes, le nuage fait des calculs sur la donnée tout en respectant la préservation des données personnelles de son client.

Théoriquement, un chiffrement homomorphe dit *total* (Gentry, 2010) permet d'appliquer n'importe quelle opération sur une donnée chiffrée, sans la déchiffrer. Pour que le chiffrement soit total, le schéma (*c.-à-d.*, l'algorithme) décompose le calcul en un circuit logique et implémente les portes sous la forme d'opérations "plus" et "multiplier", ramenant ainsi toutes les opérations à une suite logique de ces deux là. Malheureusement, les implémentations du chiffrement homomorphe total sont, encore aujourd'hui, trop coûteuses. C'est le cas du schéma de Gentry (2009) qui permet d'appliquer les opérations homomorphes seulement un nombre limité de fois. Dépasser un certain seuil, l'information chiffrée contient tellement de bruit qu'elle en devient indéchiffrable !

Gentry a montré que ce bruit pouvait être réduit par une procédure de rafraîchissement de l'information chiffrée, permettant ainsi d'appliquer les opérations homomorphes un nombre illimité de fois. Gentry et Halevi (2011) proposent une implémentation de ce schéma. Cependant, la clef publique fait une taille de 2.3 Go. La procédure de rafraîchissement prend 30 minutes et doit être appliquée sur le texte chiffré toutes les deux à trois opérations.

D'autres schémas homomorphes totaux existent tels que le schéma de van Dijk et collab. (2010) applicable sur les entiers. Mais, la taille de la clef publique (10 Mo) ainsi que le temps d'application de la procédure de rafraîchissement (11 minutes) sont encore très longs. Trop longs pour être raisonnablement considérés dans les applications du nuage.

Ces chiffres proviennent d'une optimisation faite par Coron et collab. (2012).

des chiffrements homomorphes raisonnablement efficaces existent pour un *sous ensemble d'opérations*. L'algorithme Paillier (1999) qui supporte uniquement l'addition et l'algorithme ElGamal (1985) qui supporte uniquement la multiplication en sont des exemples.

Au même titre, un chiffrement déterministe (Schneier, 1996), comme l'algorithme AES vue précédemment, est considérée comme un chiffrement ho-

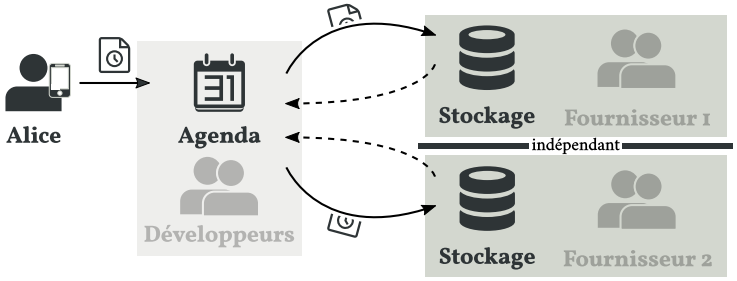


FIGURE 5 – *Principe de la fragmentation verticale.*

momorphe. Celui-ci vérifie l'égalité pour deux données chiffrées avec la même clef.

2.4.3.3 La fragmentation verticale

La fragmentation verticale (Aggarwal et collab., 2005) divise une donnée en fragments inintelligibles, de telle manière que seules les personnes qui y sont autorisées puissent la recomposer. La fragmentation protège une association d'attributs (§2.4.2) considérée comme une donnée personnelle. Elle fragmente l'association et héberge chaque fragment, devenu inintelligible, chez différents fournisseurs du nuage. Cependant, pour que le résultat soit sûre, la fragmentation verticale présuppose que les fournisseurs ne se connaissent pas et ne communiquent pas entre eux.

La fragmentation verticale est une bonne alternative au chiffrement lorsqu'une contrainte de confidentialité est faite d'une association d'attributs. En cassant l'association, la fragmentation protège la contrainte associative, mais laisse l'information dans chaque fragment lisible et traitable.

Aggarwal et collab. (2005) sont les premiers à proposer une approche qui réalise la fragmentation verticale d'une donnée. Dans celle-ci (*cf. fig.* 5), le développeur de l'application fragmente le schéma relationnel d'une base de données (*ex.*, la table des rendez-vous, *fig.* 3b) sur ses contraintes de confidentialités associatives (*ex.*, $\{date, adresse\}$). La fragmentation produit au minimum deux (*généralisable*) schémas relationnels dont les attributs ne violent pas les contraintes associatives. L'idée ensuite est que deux fournisseurs de base de données *indépendants* implémentent les schémas relationnels résultants. Ainsi, le développeur d'application héberge, *en clair*, les fragments des données personnelles chez les différents fournisseurs, mais *préserve* les contraintes associatives des yeux des fournisseurs.

Pour exécuter une requête, le développeur fragmente celle-ci et transmet les sous-requêtes aux fournisseurs appropriés. Il recompose ensuite les résultats de son côté dans le dessein de protéger les contraintes associatives.

Enfin, pour que la protection soit totale, Aggarwal et collab. proposent de protéger les contraintes singleton (*ex.*, $\{nom\}$) en chiffrant dans le premier fragment la valeur de l'attribut en question. Puis, en insérant dans le second fragment la clef de déchiffrement.

Exemple avec la table des rendez-vous

L'application d'agenda personnel en ligne héberge les rendez-vous d'Alice dans une base de données du nuage. La base de données a le schéma relationnel

$date$	$nom_{\text{chiffré}}$	id
16/11/2015	gTR7Y0kmcUCCK8f97EaLBQ==	1
16/11/2015	3t77prme/XGoTwTVRRdtnQ==	2
19/11/2015	ZjbBkJEmjB6vDawjv53iRA==	3

(a) *Premier fragment. Pour le chiffrement, nous utilisons l'algorithme AES 128 bits. Le texte est obtenu par un codage base64.*

nom_{clef}	$adresse$	id
key1	Bureau	1
key2	Parc	2
key3	Bureau	3

(b) *Second fragment. Les clefs (cf. colonne nom_{clef}) permettent de retrouver les valeurs initiales "Bob" et "Claire".*

FIGURE 6 – *Protection des rendez-vous d'Alice par la fragmentation verticale.*

en.wikipedia.org/wiki/Base64

$(date, nom, adresse)$ et deux contraintes de confidentialités ($\{date, adresse\}$ et $\{nom\}$). La première, associative, dit qu'un individu malveillant ne peut pas localiser Alice. La seconde, singleton, dit qu'un individu malveillant n'infère pas le nom des contacts d'Alice.

Dans un premier temps, l'approche d'Aggarwal et collab. fragmente le schéma relationnel sur les contraintes associatives. La fragmentation produit deux sous-schémas relationnels de sorte que tous les attributs membres de la contrainte associative n'apparaissent pas dans la liste des attributs d'un des schémas. Pour l'agenda en ligne, le développeur découpe en un premier fragment $(date)$ et un second $(adresse)$. Ainsi, un fournisseur de base de données ne peut pas reconstruire la contrainte $\{date, adresse\}$ en introspectant les valeurs de son fragment.

Dans un second temps, l'approche protège les contraintes singletons. La valeur de l'attribut de la contrainte singleton est chiffrée dans le premier fragment. Sa clef de déchiffrement est stockée dans le second fragment. Ceci produit les schémas relationnels $(date, nom_{\text{chiffré}})$ et $(nom_{\text{clef}}, adresse)$. Comme pour la première étape, un fournisseur ne peut pas inférer les noms des contacts d'Alice.

Enfin, pour que le développeur puisse rassembler les morceaux des sous-requêtes, chaque ligne dans la base est accompagnée d'un identifiant. Le résultat de la fragmentation est donné dans la figure 6.

Calcul d'une requête distribuée

Pour calculer une requête sur une base de données fragmentée, Aggarwal et collab. réutilisent les travaux sur les bases de données distribuées. Une base de données distribuée est définie comme "un ensemble de plusieurs bases logiquement corrélées et distribuées sur le réseau. Un système de gestion de bases de données distribuées (SGBDD) est, quant à lui, définit comme l'application qui permet la gestion des bases distribuées et rend la distribution transparente pour l'utilisateur" (Özsu et Valduriez, 2011).

Relativement à la fragmentation verticale, les bases de données distribuées sont les fragments. Le SGBDD, quant à lui, fait partie de l'application SaaS. Dans l'idée, le développeur d'applications écrit sa requête comme si la base était centralisée et le SGBDD réécrit la requête pour la distribuer sur les fragments. La réécriture utilise l'information des schémas de fragmentation et les lois algébriques sur l'algèbre relationnelle (Ullman, 1982 ; Özsu et Valduriez, 2011).

L'algèbre relationnelle. L'algèbre relationnelle est un langage abstrait pour manipuler des n-uplets dans une relation (*c.-à-d.*, une table). Dans l'algèbre relationnelle, le type de la relation est défini par son schéma relationnel. Celui-ci est fait d'une liste d'attributs qui spécifient l'arité, le nom et potentiellement, le type des n-uplets. En particulier, l'algèbre relationnelle présuppose que le schéma est une liste non vide et finit. De même, l'ordre des attributs dans le schéma est important.

Connaître le type d'une relation nous est utile pour le chapitre 4 et 6.

La suite présente les principales fonctions de l'algèbre relationnelle avec une définition proche de celle d'Ullman (1982). Elle fait toutefois, apparaître en plus, le type de la relation après application de la fonction. Les fonctions d'insertion, de suppression et de modification d'un n-uplets ne sont pas spécifiées car, comme le dit Ullman :

Ullman, 1982, p.151.

> *"The nonquery aspects of a query langage are often straighforward, being concerned with the insertion, deletion and modification of tuples"*

La projection (notée $\pi_{a_i,\dots}$ et équivalent au "`SELECT ai, ...`" de SQL) conserve un sous-ensemble des colonnes d'une relation. Appliquer la fonction π_δ sur la relation R de type Δ (avec $\delta \subseteq \Delta$), produit une nouvelle relation R' de type δ. Pour construire les n-uplets de la relation R', la projection prend tous les n-uplets de le relation R et conserve les valeurs des attributs contenus dans δ. Par exemple, appliquer $\pi_{date,nom}$ sur la table des rendez-vous (*fig.* 3b) conserve uniquement les dates et les noms. Le type de la nouvelle relation est donc $(date, nom)$.

Pour raccourcir la syntaxe, on utilisera la notation $\pi_{date,nom}$, plutôt que $\pi_{(date,nom)}$.

La sélection (notée $\sigma_{pa_i \wedge \dots}$ et équivalent au "`WHERE p(ai) AND ...`" de SQL avec `p` un prédicat sur la valeur de l'attribut `ai`) filtre les n-uplets sur un prédicat. Le filtrage élimine des n-uplets mais ne change pas le type de la relation. Ainsi, appliquer la fonction σ_{p_δ} sur la relation R de type Δ, produit une nouvelle relation R' de type Δ. Par exemple, la sélection qui garde les rendez-vous d'Alice de la semaine prochaine au bureau, se note :

$$\sigma_{(date - aujourdhui) \in [0..7] \wedge adresse = \text{`}Bureau\text{'}}$$

Le produit cartésien (noté $\times$) combine les n-uplets d'une première relation avec les m-uplets d'une seconde relation pour générer une nouvelle relation constituées des $(n+m)$-uplets. En considérant que les types de la première et de la seconde relation sont, respectivement, Δ et Δ', alors le type de la relation après application du produit cartésien est $\Delta +\!\!+ \Delta'$. Lorsqu'il y a un conflit sur les noms des attributs ainsi réunis, ils sont préfixés par le nom de la relation. Par exemple, le type du produit cartésien $G \times D$, avec G la relation qui contient le fragment gauche des rendez-vous d'Alice (*fig.* 6a) et D la relation du fragment droit (*fig.* 6b) est le schéma :

L'opérateur $+\!\!+$ est la concaténation de deux listes.

$$(date, nom_{\text{chiffré}}, G.id, nom_{\text{clef}}, adresse, D.id)$$

La jointure naturelle (notée $\bowtie$ et équivalant au `JOIN` de SQL) combine les n-uplets d'une première relation avec les m-uplets d'une seconde relation, dont les valeurs des attributs commun sont égales. En considérant que les types de la première et de la seconde relation sont, respectivement, Δ et Δ',

alors le type de la relation après jointure est $nub(\Delta +\!\!+ \Delta')$. La jointure naturelle s'encode comme la composition du produit cartésien avec une sélection sur les attributs commun (le prédicat est le test d'égalité) et une projection sur les attributs de $nub(\Delta +\!\!+ \Delta')$. Par exemple, le type de la jointure naturelle $G \bowtie D$ est le schéma :

La fonction nub[8] supprime les doublons dans une liste.

En fonction des besoin on optera pour la notation infixe $G \bowtie D$ ou préfixe $\bowtie (G, D)$.

$$(date, nom_{\text{chiffré}}, nom_{\text{clef}}, adresse, id)$$

En plus des quatre fonctions ci-dessus définies par Ullman, des fonctions d'agrégation sont couramment ajoutées pour exprimer des requêtes plus intéressantes. Dans cette thèse, seule l'agrégation par dénombrement (noté $count_{a_i,\ldots}$ et équivalent au `COUNT(*)/GROUP BY ai,...` de SQL) est considérée. Dans un premier temps, l'agrégation range les n-uplets dans des groupes. La distribution d'un n-uplet dans un groupe plutôt que dans un autre se fait en testant l'égalité de certains attributs. Puis, l'agrégation compte le nombre de n-uplets par groupe, réduisant chaque groupe à un n-uplet. Formellement, appliquer la fonction $count_\delta$ sur une relation de type Δ, groupe les n-uplets sur les attributs de δ et compte le nombre de n-uplets pour former une nouvelle relation de type $\delta +\!\!+ (count)$. Le dernier attribut contient le nombre de lignes comptées dans le groupe. Par exemple, la fonction $count_{date}$ compte le nombre de rendez-vous par jour. Le type de la relation est $(date, count)$.

$$Q ::= Q \circ Q \mid \pi_\delta \mid \sigma_{p_\delta} \mid (\times) \mid (\bowtie) \mid count_\delta$$

FIGURE 7 – *La syntaxe de l'algèbre relationnelle.*

Les fonctions se composent ($\circ$) pour former une requête plus complexe, selon la grammaire de la figure 7. Une séquence de composition de fonctions se lit de droite à gauche. Par exemple, la requête qui retourne les *dates et contacts des rendez-vous d'Alice de la semaine prochaine au bureau* s'écrit :

Nous utilisons l'opérateur de composition de fonction ($\circ$) pour représenter les requêtes, alors que la communauté des bases de données utilise communément les arbres d'opérations. Mais, nous préférons la notation par composition par souci de cohérence avec les contributions de cette thèse.

$$\pi_{date,nom} \circ \sigma_{\substack{(date-aujourdhui)\in[0..7] \\ \wedge adresse=\text{`}Bureau\text{'}}} \quad rendezvous$$

Et elle se lit : sur la relation des rendez-vous (*c.-à-d.,* $(date, nom, adresse)$) appliquer la sélection (σ) des rendez-vous effectués la semaine prochaine au bureau, puis projeter (π) pour obtenir les valeurs des dates et des contacts. L'équivalent SQL est donnée dans le code 8.

```
SELECT date, nom FROM rendezvous
WHERE (date - aujourdhui) in [0..7]
AND adresse='Bureau'
```

Code 8 – *Requête qui retourne les dates et contacts des rendez-vous d'Alice de la semaine prochaine au bureau dans un langage à la SQL.*

Lois algébriques de l'algèbre relationnelle. Une loi algébrique est une règle de transformation qui, si elle est suivie, garantit au système la correction du résultat de la transformation. Une loi algébrique de l'algèbre relationnelle est donc une règle qui, si elle est appliquée sur une requête relationnelle, garantit que la requête transformée pourra être exécutée et retournera un résultat équivalent. Deux de ces lois sont les lois de cascades sur les fonctions

8 hackage.haskell.org/package/base-4.8.2.0/docs/Data-List.html#v:nub

unaires (Ullman, 1982). Ici, les lois sont exprimées sur une relation qui a le schéma $\Delta = (a_1, \ldots, a_i, \ldots, a_n)$:

$$\pi_{a_1} \circ \pi_{a_2} \circ \cdots \circ \pi_n \equiv \pi_{a_1,a_2,\ldots,a_n} \tag{3}$$

$$\sigma_{pa_1} \circ \sigma_{pa_2} \circ \cdots \circ \sigma_{pa_n} \equiv \sigma_{pa_1 \wedge pa_2 \wedge \cdots \wedge pa_n} \tag{4}$$

Pour la projection (π), la loi dit que plusieurs projections successives sur une même relation peuvent être groupées. Inversement, une projection sur plusieurs attributs peut être séparée en plusieurs projections successives. Pour la sélection (σ) la loi dit que plusieurs sélections successives sur une même relation peuvent êtres groupées en une seule sélection par la conjonction des prédicats. Inversement, une sélection en forme conjonctive peut être séparée en plusieurs sélections successives.

La loi de commutativité suivante (Ullman, 1982), dit que la projection et la sélection, sur une même relation, commutent :

$$\pi_{a_1,\ldots,a_n} \circ \sigma_{pa_i} \equiv \sigma_{pa_i} \circ \pi_{a_1,\ldots,a_n} \tag{5}$$

Ceci permet de transformer une requête en appliquant, selon les besoins, d'abord la projection ou la sélection.

Deux autres lois régulièrement utilisées pour transformer une requête sont les lois de commutativité entre la jointure naturelle ($\bowtie$) et la projection/sélection (Özsu et Valduriez, 2011). La jointure s'exprime ici sur deux relations qui ont, respectivement, le type Δ et Δ'.

$$\pi_{\delta +\!\!+ \delta'} \circ \bowtie \equiv \bowtie_{\alpha} (\pi_{\delta}, \pi_{\delta'}) \qquad \text{avec } \alpha \in \delta \text{ et } \delta', \delta \subseteq \Delta, \delta' \subseteq \Delta' \tag{6}$$

$$\sigma_{p\alpha \wedge q\beta} \circ \bowtie \equiv \bowtie (\sigma_{p\alpha}, \sigma_{q\beta}) \qquad \text{avec } \alpha \in \Delta, \beta \in \Delta' \tag{7}$$

Nous utilisons la fonction ($\bowtie$) avec sa notation préfixe.

La paire à droite de l'équivalence représente les deux relations (ou les deux fragments), c'est pourquoi elle est toujours devant la fonction de jointure et qu'elle disparaît après application de la jointure. L'équation 6 déclare qu'une projection commute avec la jointure si la projection est distribuée correctement sur chaque relation. De plus, la projection ne doit pas supprimer les attributs communs nécessaires à la jointure naturelle. L'équation 7 spécifie qu'une sélection commute avec la jointure en séparant le prédicat.

Dans le cas des bases de données distribuées, commuter la projection / sélection avec la jointure est une très bonne optimisation. Cela permet de faire plus de calculs de manière concurrente sur chaque fragment.

Applications des lois algébriques. La suite utilise ces lois pour transformer la requête qui retourne les *dates et contacts des rendez-vous d'Alice de la semaine prochaine au bureau*, de sa version locale vers sa version distribuée. Le schéma de la relation (à droite de la requête) est présent à titre indicatif. Il aide à mieux comprendre l'évolution de la requête.

$$\begin{array}{c} \pi_{date,nom} \\ \uparrow \\ \sigma_{(date-aujourdhui)\in[0..7] \atop \wedge adresse='Bureau'} \\ \uparrow \\ (date, nom, adresse) \end{array}$$

La même requête avec la représentation par un arbre d'opérations, plus commune à la communauté des bases de données.

$$\pi_{date,nom} \circ \sigma_{(date-aujourdhui)\in[0..7] \atop \wedge adresse='Bureau'} \quad (date, nom, adresse)$$

Une vision générale de l'approche proposé par Aggarwal et collab. est de commencer par introduire la fonction de jointure à droite de la requête, pour stipuler l'existence de deux fragments. Puis, de pousser à l'aide des lois de Özsu

et Valdurier, 2011, les fonctions de la requête locale vers l'intérieure de la jointure. Ceci pour distribuer la requête et optimiser/maximiser la quantité de calcul fait sur chaque fragment. Introduire la jointure produit la requête suivante.

$$\pi_{date,nom} \circ \sigma_{\substack{(date-aujourdhui)\in[0..7] \\ \wedge adresse='Bureau'}} \circ \bowtie \begin{array}{l}((date, nom_{\text{chiffré}}, id), \\ (nom_{\text{clef}}, adresse, id))\end{array}$$

Comme le mentionnent Aggarwal et collab., la jointure, telle qu'elle est défini précédemment, ne déchiffre pas les paires d'attributs chiffrées ($nom_{\text{chiffré}}$, nom_{clef}). Ils proposent de modifier la sémantique de la jointure pour qu'elle déchiffre les paires chiffrées. Ainsi, le reste de la requête peux s'appliquer.

Concrètement, la jointure s'effectue chez le développeur de l'application (ici, l'agenda) pour empêcher les fournisseurs, qui gèrent les fragments, de reconstruire les contraintes de confidentialités associatives (ici, $\{date, adresse\}$). Néanmoins, l'agenda doit rapatrier chez lui l'entièreté des données des deux fragments pour faire la jointure et exécuter le reste de la requête.

Pour optimiser la requête, Aggarwal et collab. disent utiliser ensuite les lois de commutativités sur la jointure. Ceci fonctionne très bien pour la sélection, grâce à l'équation 7.

$$\pi_{date,nom} \circ \bowtie \begin{array}{ll}(\sigma_{(date-aujourdhui)\in[0..7]} & (date, nom_{\text{chiffré}}, id), \\ \sigma_{adresse='Bureau'} & (nom_{\text{clef}}, adresse, id))\end{array}$$

En revanche, il n'est pas possible d'utiliser l'équation 6 qui commute la projection avec la jointure, car la projection $\pi_{date,nom}$ ne prend pas en compte les identifiants. De même, que signifie la commutativité de la projection sur les noms, sachant que ceux-ci sont chiffrés ? Malheureusement, Aggarwal et collab. ne formalisent pas cette spécificité dans leurs travaux sur la fragmentation verticale (2005). Il en va de même pour les travaux plus récents (di Vimercati et collab., 2013).

L'optimisation de la requête n'est pas bloqué pour autant. Mais, il faut utiliser l'algorithme fournit par di Vimercati et collab. (2013), plutôt que les lois.

$$\pi_{date,nom} \circ \bowtie \begin{array}{ll}(\pi_{date,nom_{\text{chiffré}},id} \circ \sigma_{(date-aujourdhui)\in[0..7]} & (date, nom_{\text{chiffré}}, id), \\ \pi_{nom_{\text{clef}},id} \circ \sigma_{adresse='Bureau'} & (nom_{\text{clef}}, adresse, id))\end{array}$$

Il n'y a pas de problème à utiliser un algorithme. Seulement, un algorithme ne spécifie pas, à l'inverse des lois, le comportement d'une fonction par rapport à un autre. Ce qui limite le raisonnement. Par exemple, l'algorithme de di Vimercati et collab. propose une unique solution (une transformation), alors que plusieurs sont envisageables.

Une meilleur approche aurait été de proposer un nouvelle fonction pour la fragmentation. Puis, d'étendre les lois lois proposées par Ullman (1982) ou Özsu et Valduriez (2011) pour spécifier le comportement de la fragmentation. Ce type d'approche est également plus résilient à l'extension de l'algèbre relationnelle (si on voulait introduire plusieurs fonctions de cryptographie, par exemple).

Évaluation d'une requête distribuée

L'évaluation d'une requête distribuée peut suivre deux stratégies selon que les sous-requêtes sont évaluées en séquence ou de manière concurrente (di Vimercati et collab., 2013).

Dans la stratégie séquentielle, un premier fournisseur de base de données évalue sa sous-requête et retourne le résultat au SGBDD. Le SGBDD utilise ensuite les identifiants de la première sous-requête pour rajouter un prédicat dans la seconde sous-requête. Ce prédicat limite les résultats retournés par la seconde sous-requête aux n-uplets qui respectent les conditions de la première sous-requête. Ceci réduit la quantité d'information transmise sur le réseau.

Dans la stratégie concurrente, les fournisseurs de base de données évaluent de manière concurrente les sous-requêtes. Le résultat de chaque sous-requête est retourné au SGBDD qui fait la jointure sur les identifiants. Avec cette stratégie, le temps global de l'évaluation est potentiellement réduit du fait de l'évaluation concurrente. En revanche, elle est plus coûteuse que la stratégie séquentielle sur la quantité d'information transmise sur le réseau, puisque chaque sous-requête va potentiellement retourner plus de n-uplets que nécessaire.

Une fragmentation optimale ?

Quel que soit un n-uplet d'attributs et son ensemble de contraintes de confidentialités, plusieurs fragmentations sont envisageables. Par exemple, si un rendez-vous avait que sa contrainte associative comme contrainte de confidentialité (et que la contrainte $\{nom\}$ n'était pas considérée). Alors le n-uplet admettrait deux schémas de fragmentation. Soit $\{(date, id), (nom, adresse, id)\}$, ou $\{(date, nom, id), (adresse, id)\}$.

Calculer le schéma optimal, tel que le résultat de la fragmentation exécute un maximum de calcul sur le nuage, est un problème NP-difficile[9] (Aggarwal et collab., 2005). Aggarwal et collab. proposent une heuristique pour calculer une solution acceptable, mais le calcul d'un schéma de fragmentation optimal n'entre pas dans les préoccupations de cette thèse.

Problème de la fragmentation verticale

En plus de l'absence de lois pour raisonner sur la fragmentation, la fragmentation verticale souffre d'un deuxième problème. Elle ne protège pas les données par rapport à l'application (ici, l'agenda) qui est toujours considérée comme digne de confiance, et ce, quelleque soit l'implémentation envisagée (Aggarwal et collab., 2005 ; Ciriani et collab., 2010 ; Biskup et collab., 2011 ; di Vimercati et collab., 2013).

Cette assertion est contradictoire avec l'approche PbD où le développeur se doit de concevoir une application qui est respectueuse des données personnelles de ses clients. Cela implique de préserver les données personnelles de tous les acteurs autres que le propriétaire des données, *y compris de l'application elle même* (§2.3.2). Le chapitre suivant (*chap.* 3) montre qu'une modification de

9 fr.wikipedia.org/wiki/Problème_NP-complet

l'endroit où s'applique la jointure élimine l'assertion et fait de la fragmentation verticale une technique très utile.

2.5 Les limites de l'approche de protection intégrée de la vie privée

Aujourd'hui, l'approche de protection intégrée de la vie privée (PbD) fait face à deux problèmes.

Le premier est dû aux techniques de cryptographie. Comme l'analyse le montre (§2.4.3), chaque technique est un mécanisme très sûr pour protéger *un aspect* du nuage. Mais, aucune n'est la panacée. Ainsi, le calcul côté client permet de faire n'importe quel calcul de manière privée, mais perd tous les avantages du nuage vu que le calcul est fait chez le client. Le chiffrement symétrique est très performant pour sécuriser une donnée stockée, mais empêche tous calculs. Le chiffrement homomorphe total n'est pas la solution, car trop coûteux. Le chiffrement homomorphe partiel est envisageable, mais se limite à seul type d'opération (*ex.*, le test d'égalité). Enfin, La fragmentation corrige cette limitation, mais s'applique que sur les contraintes associatives et l'application doit être de confiance ce qui est contraire à l'approche PbD.

Du fait de chacune de ces limitations, une application complexe telle que l'agenda personnel en ligne (*cf.* §1.1.1, *fig.* 1) ne peut pas être implémenté avec l'approche PbD. Une implémentation basée uniquement sur le calcul côté client n'aurait pas plus d'intérêt qu'une application locale. Une implémentation basée uniquement sur le chiffrement symétrique obligerait à rapatrier entièrement la base de données chez le client à chaque requête. Une implémentation basée uniquement sur le chiffrement homomorphe demanderait des temps d'exécution trop longs (à cause de la procédure de rafraîchissement). Enfin, Une implémentation basée uniquement sur la fragmentation verticale requerrait de faire confiance à l'application SaaS agenda. Ces affirmations sont vérifiés dans le chapitre suivant (*chap.* 3) par un travail d'implémentation.

Le second problème vient de la littérature scientifique qui se focalise exclusivement sur les techniques (pilier mécanisme d'un système sécurisé – §2.3.1). Mais ceci n'est pas suffisant. La section sur les piliers d'un système sécurisé explique que le pilier mécanisme doit être vérifier par le pilier assurance (§2.3.1). Or, même si les techniques présentées précédemment sont des mécanismes très sûres, chaque technique exige de respecter un protocole compliqué pour être utilisées correctement. Ceci rend les applications sujettes aux vulnérabilités sur les bogues (§2.3.1). Il faut donc s'assurer que l'application utilise correctement les techniques et que les techniques protègent correctement les contraintes de confidentialités.

Le cas de l'agenda personnel en ligne 3

Le chapitre précédent présente l'approches de la protection intégrée de la vie privée (PbD) pour préserver les données personnelles hébergées dans le nuage. Cette approche repose sur l'utilisation de techniques de cryptographie. Son dessein est de concevoir et développer les applications du nuage de telle manière qu'elles satisfassent les principes et les règles du respect de la vie privée.

Le chapitre précédent soutient que les techniques de cryptographie sont limitées dans leur utilisation et suggère de composer les techniques pour palier à cette limitation. Le dessein de ce chapitre est de vérifier cette suggestion. Pour ce faire, le cas du développement de l'agenda personnel en ligne par l'approche PbD est étudié et sert de référence pour le reste de cette thèse. Les techniques de cryptographie définies précédemment sont utilisées pour implémenter l'agenda personnel dans une infrastructure réelle du nuage.

Dans un premier temps, les implémentations servent à mener des expérimentations. Elles permettent de comparer les techniques en fonction de trois critères qui sont la confidentialité des données personnelles, l'utilisation du nuage et la performance en temps d'exécution. Les résultats de ces expérimentations montrent que les techniques de cryptographie sont toutes limitées dans leur utilisation (§3.1).

Dans un second temps, cette thèse vérifie l'hypothèse que composer toutes les techniques permet de pallier aux limites de chacune. À nouveau, une implémentation est réalisée et des expérimentations sont faites en fonction des critères de la confidentialité, du nuage et des performances. Les expérimentations montrent que l'application agenda ne souffre plus des limites précédentes grâce à la composition. En revanche, la composition produit une application plus difficile à implémenter et donc prompte à plus de bogues (§3.2).

Ensuite, ce chapitre propose une nouvelle approche pour la conception et le développement d'une application PbD. Celle-ci se nomme l'"approche du *nuage confidentiel*" et stipule quelles sont les caractéristiques d'une bonne application PbD pour le nuage (§3.3). À partir de cette approche, quelques travaux connexes qui ont, eux aussi, choisi l'axe de la composition des techniques de cryptographie sont évalués (§3.4).

Enfin, l'approche imaginée par cette thèse pour réaliser l'approche du *nuage confidentiel* est décrite en conclusion et donne la structure du reste de cette thèse.

3.1 L'emploi des techniques de cryptographie

Cette section reprend les trois techniques de cryptographie vues au chapitre précédent, à savoir : les calculs côté client, le chiffrement symétrique et la fragmentation verticale. Chaque technique est utilisée pour implémenter l'agenda personnel en ligne. Pour chaque implémentation, deux opérations sont étudiées.

La première opération se nomme `[adresse]`. Elle retourne la liste des adresses et contacts visités par Alice (dont le nom commence par la lettre C) pour, par exemple, ajouter un service de cartographie. La seconde opération se nomme `#rendezvous`. Elle compte le nombre de rendez-vous pris avec Bob au bureau la semaine prochaine pour, par exemple, effectuer des statistiques.

À chaque fois, une évaluation est faite pour comparer les techniques entre elles. Cette évaluation se base sur trois critères définis à la suite des constatations faites dans le chapitre précédent (*cf.* §2.5) :

1. La confidentialité. Est-ce que l'agenda protège les données personnelles d'Alice ? Est-ce que les contraintes de confidentialités $\{nom\}$ et $\{date, adresse\}$ sont préservées des actions autres que celles d'Alice ?
2. L'utilisation du nuage. Est-ce que l'agenda est un service du nuage (SaaS) ? Est-ce que la base de données est un service du nuage (PaaS) ? Est-ce que l'application profite des avantages du nuage pour que le développeur puisse se concentrer sur l'innovation ?
3. La performance. Quel est le temps d'exécution des opérations ? Quelle est la quantité d'information transportée sur le réseau ?

Le dessein est de montrer que les techniques actuelles sont toutes limitées dans leur utilisation au regard de ces trois critères. De ce fait, l'emploi d'une seule technique est, bien souvent, insuffisant lorsque l'application est complexe et possède plusieurs contraintes de confidentialités.

3.1.1 Plateforme pour l'évaluation

L'évaluation est faite dans une infrastructure réelle. L'application agenda est hébergée en tant que SaaS grâce à la plateforme d'exécution PaaS Heroku[1]. Si l'implémentation requiert une base de données dans le nuage, alors celle-ci est une base de données PostgreSQL[2], proposée en PaaS grâce au IaaS Amazon Web Services[3]. Pour l'implémentation par fragmentation verticale, la deuxième base de données PostgreSQL est proposée en PaaS grâce au IaaS Google Compute Engine[4]. Ainsi, les deux bases de données sont indépendantes.

L'agenda est implémenté dans le langage JavaScript (MDN, 2015). De ce fait, le même langage est utilisé pour le code exécuté dans le nuage et chez le client. Les calculs réalisés dans le nuage sont faits grâce à l'exécutant Node.js en version 5.10.1. Les calculs réalisés chez Alice sont exécutés par le navigateur Firefox en version 48. L'ordinateur où est installé Firefox est le Pentium

Node.js[5] permet l'exécution, côté serveur, d'applications Web écrites en JavaScript.

1 heroku.com
2 postgresql.org
3 aws.amazon.com
4 cloud.google.com/compute
5 nodejs.org

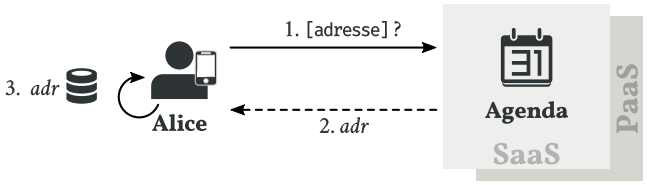


FIGURE 9 – *L'opération* `[adresse]` *sur l'agenda personnel protégé par la technique des calculs côté client.*

5 avec 8 Go de RAM utilisé pour écrire cette thèse. Enfin, à chaque fois qu'une technique considère des données chiffrées, l'implémentation utilise le schéma AES-CBC (Daemen et Rijmen, 1999) avec une clef de 256 bits.

L'évaluation mesure le temps d'exécution en secondes des deux opérations `[adresse]` et `#rendezvous`. Chaque opération est appliquée sur une base de données contenant 2 500 rendez-vous sous la forme d'un 3-uplet $(date, nom, adresse)$. La taille de 2 500 rendez-vous n'est pas arbitraire. Ceci est le nombre maximum, permis par l'API Google Calendar[6], de rendez-vous qui peuvent être retournés en une seule opération. Le choix des opérations `[adresse]` et `#rendezvous` n'est pas arbitraire non plus. Ce sont deux opérations qui impliquent d'utiliser tous les opérateurs de l'algèbre relationnelle.

L'évaluation est faite grâce à la bibliothèque Benchmark.js[7]. Le code de l'application et de l'évaluation sont disponibles sur le répertoire Git de cette thèse[8]. Le lecteur peut exécuter l'évaluation en local pour vérifier que l'implémentation fonctionne. En revanche, pour des raisons de coût, le lecteur devra fournir ses propres instances de SaaS, PaaS et IaaS pour exécuter l'évaluation sur le nuage.

3.1.2 Emploi du calcul côté client

Dans le calcul côté client (*cf.* §2.4.3.1), Alice conserve ses rendez-vous sur son terminal. Elle utilise ensuite ce même terminal pour réaliser les calculs de l'agenda. Ainsi, aucun de ses rendez-vous ne peut être vu par un acteur du nuage autre qu'elle-même.

La figure 9 représente les interactions entre Alice et l'agenda, en suivant la technique du calcul côté client. Ici, la figure modélise l'opération `[adresse]` qui retourne la liste des adresses et des contacts visités par Alice. Les interactions sont les mêmes pour l'opération `#rendezvous`.

Utilisation du nuage. Concrètement, le développeur utilise un serveur Web hébergé en tant que PaaS pour exécuter l'agenda. L'application agenda est alors déployée dans le nuage en tant que service SaaS et, à ce titre, devient accessible depuis n'importe où dans le monde (propriété de disponibilité). En revanche, puisque l'agenda est implémenté avec la technique du calcul côté client, Alice conserve ses rendez-vous sur son terminal. De ce fait, l'application se prive d'une base de données PaaS qui offre élasticité et réplication aux rendez-vous. Il est donc clair que cet agenda personnel en ligne ne profite pas pleinement des faveurs du nuage.

6 developers.google.com/apis-explorer/#s/calendar/v3/calendar.events.list
7 benchmarkjs.com
8 Les sources des expérimentations sont disponibles à l'adresse github.com/rcherrueau/C2QL/tree/master/experimentations

Dans cet exemple, il n'est pas nécessaire de calculer une preuve à divulgation nulle de connaissance (cf. §2.4.3.1) car aucun résultat n'est retourné à l'agenda.

Confidentialité. Qu'en est-il de la confidentialité ? Pour répondre à cette question, il faut regarder les interactions de la figure 9. D'abord, Alice demande à exécuter la requête `[adresse]` (1). En accord avec la technique du calcul côté client, l'agenda répond alors avec le calcul à effectuer, symbolisé, ici, par la fonction adr (2). Finalement, Alice utilise la fonction adr sur sa base locale de rendez-vous pour calculer la liste des adresses et des contactes (3).

Alice est donc la seule à posséder ses données personnelles. De plus, tous les calculs de l'agenda sont faits chez elle. Par conséquent, aucun acteur du nuage hormis Alice, ne peut reconstruire l'une des deux contraintes de confidentialité $\{nom\}$ et $\{date, adresse\}$. Ceci fait du calcul côté client une technique excellente du point de vue de la protection des données personnelles.

Performance. L'agenda avec calculs côté client a été implémenté en JavaScript pour tester les performances. La partie serveur est hébergée en tant que SaaS grâce à l'exécutant Node.js proposé par le PaaS Heroku. Elle fournit à Alice le téléchargement des calculs (spécifiquement, `[adresse]` et `#rendezvous`) sur son terminal personnel (ici, Firefox version 48). Ainsi, Alice applique elle-même les opérations sans dévoiler ses données personnelles.

Les temps d'exécutions des deux opérations, sur un échantillon de 2 500 rendez-vous, sont très bons. L'opération `[adresse]` s'effectue en 0.26 ms, tandis que l'opération `#rendezvous` s'effectue en 140.58 ms. La différence de temps entre les deux opérations s'explique par la complexité de la deuxième opération par rapport à la première. Notamment, la sélection dans la deuxième opération opère sur tous les champs de la base de données avec, en plus, un calcul sur la date qui teste son appartenance à des bornes.

3.1.3 Emploi du chiffrement symétrique

Avec le chiffrement symétrique (*cf.* §2.4.3.2), Alice sauvegarde ses rendez-vous dans le nuage de manière inintelligible. Elle doit néanmoins rapatrier tous ses 3-uplets et les déchiffrer pour ensuite réaliser les calculs de l'agenda.

La figure 10 représente les interactions entre Alice, l'agenda et la base de données en suivant la technique du chiffrement symétrique. La figure modélise l'opération `[adresse]`, mais les interactions sont similaires pour l'opération `#rendezvous`.

Utilisation du nuage. Comme pour l'implémentation précédente avec les calculs côté client, l'application agenda est déployée dans le nuage en tant que service SaaS et profite de ses avantages. La différence est sur la sauvegarde des rendez-vous. Ici, l'application utilise une base de données PaaS qui offre des réplications et préserve le développeur des besoins d'une infrastructure spécifique. Par conséquent, l'application agenda tire pleinement parti des bienfaits du nuage.

Confidentialité. Le contrat de confidentialité est respecté. Chaque rendez-vous est sauvegardé dans la base de données de manière chiffrée. Par conséquent, au cours des différentes interactions, tous les rendez-vous qui transitent sur le réseau sont inintelligibles. De ce fait, ni le PaaS de base de données, ni le PaaS d'exécution, ni le SaaS agenda sont capables de reconstruire les contraintes de confidentialités $\{nom\}$ et $\{date, adresse\}$. À l'instar du calcul

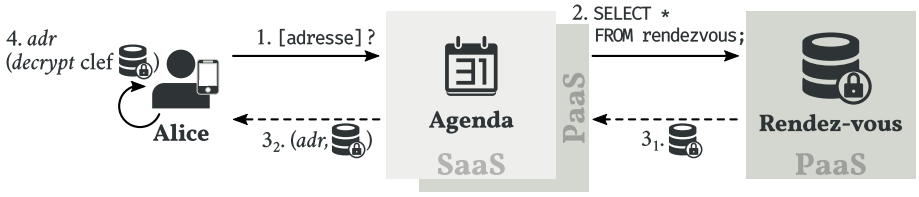


FIGURE 10 – *L'opération* `[adresse]` *sur l'agenda personnel protégé par la technique du chiffrement symétrique.*

côté client, la technique du chiffrement symétrique est excellente du point de vue de la protection des données personnelles.

Performance. Le "hic" est sur la performance ! La figure 10 explique pourquoi : pour pouvoir calculer l'opération `[adresse]`, Alice doit d'abord rapatrier sa base de données, soit 2 500 rendez-vous depuis le IaaS Amazon, vers le PaaS Heroku (3_1), puis vers son terminal (3_2). Alice doit ensuite déchiffrer tous ses rendez-vous et appliquer la fonction *adr* pour obtenir le résultat de l'opération `[adresse]` (4). *De facto*, le temps d'exécution est fortement impacté.

Firefox 48 met 1.81 sec pour calculer l'opération `[adresse]` et 2.19 sec pour l'opération `#rendezvous`. Deux raisons à ces temps. Premièrement, la quantité d'information transportée sur le réseau est 2 500 fois plus importante que dans l'implémentation côté client. Deuxièmement, l'étape (4) utilise la jeune API *Web Cryptography*[9] pour déchiffrer les rendez-vous et cette API n'est pas encore bien optimisée sur le navigateur.

3.1.4 Emploi de la fragmentation verticale

Dans la fragmentation verticale (*cf.* §2.4.3.3), l'agenda divise les rendez-vous d'Alice sur ses contraintes de confidentialités. Il forme ainsi des fragments inintelligibles qui sont hébergés chez deux fournisseurs de base de données indépendants. L'agenda requête ensuite les deux fournisseurs pour réaliser les opérations de l'application. Dans cette technique, le calcul de la requête distribué se fait automatiquement en suivant l'approche décrite par di Vimercati et collab. (2013 ; initié par Aggarwal et collab., 2005 – *cf.* §2.4.3.3).

La figure 11 représente les interactions entre Alice, l'agenda et les deux bases de données. Elle modélise l'opération `#rendezvous` qui compte le nombre de rendez-vous pris avec Bob au bureau la semaine prochaine. Les interactions sont dérivées automatiquement à partir de l'opération `#rendezvous` dans sa version centralisée (*code* 12). Ces interactions sont les mêmes pour l'opération `[adresse]`.

Utilisation du nuage. Comme les implémentations avec calcul côté client et chiffrement symétrique, l'application agenda est déployée dans le nuage en tant que service SaaS. Elle profite donc de ses avantages. La particularité ici est que les rendez-vous sont fragmentés et distribués sur deux bases de données PaaS indépendantes.

Concrètement (*cf. fig.* 6), la base de données de gauche contient les dates et les noms chiffrés. Elle est déployée sur le IaaS de Google Compute Engine. La base de données de droite contient les adresses et les clefs de déchiffrement des noms. Elle est déployée sur le IaaS d'Amazon Web Services. Cette distribution n'a pas d'incidence sur le critère d'utilisation du nuage puisque tous les calculs

9 w3.org/TR/WebCryptoAPI

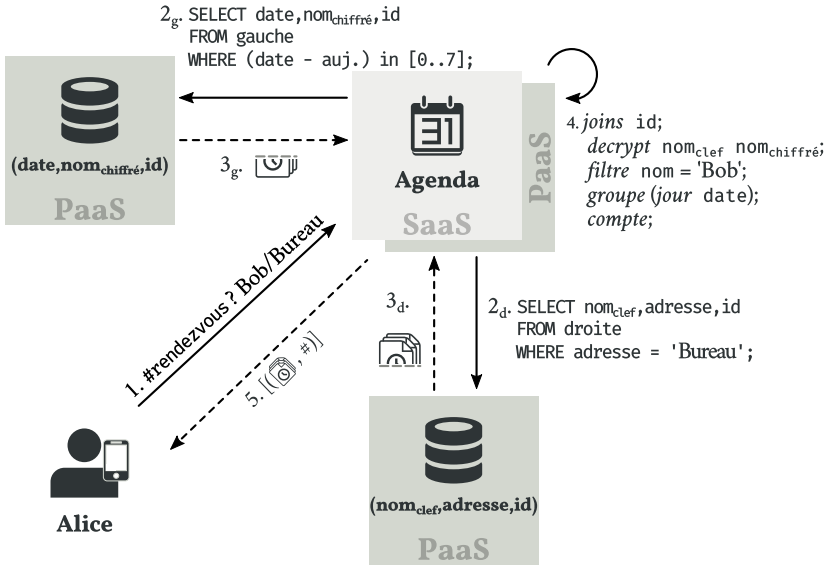


FIGURE 11 – *L'opération* `#rendezvous` *sur l'agenda personnel protégé par la technique de la fragmentation verticale.*

continuent de se faire dans le nuage. De ce fait, la fragmentation est un bon candidat du point de vue du critère d'utilisation du nuage.

Confidentialité. Contrairement aux deux versions précédentes, l'implémentation par fragmentation verticale ne protège pas les 3-uplets de toute la pile du nuage. Certes, en divisant les rendez-vous sur leur contrainte associative $\{date, adresse\}$ et en chiffrant les noms à cause de la contrainte singleton $\{nom\}$, aucune des deux bases de données PaaS ne peut violer une contrainte de confidentialité. Mais, ce n'est pas le cas de l'application agenda et de son PaaS d'exécution.

Par exemple, pour implémenter l'opération `#rendezvous` (*fig.* 11), le calcul de la requête distribuée dit que l'agenda requête de manière concurrente la base de données de gauche avec une sélection sur les dates (2_g, 3_g) et la base de données de droite avec une sélection sur les adresses (2_d, 3_d). À ce stade, il n'y a pas encore de violation des contraintes de confidentialités. En revanche, l'étape (4) requiert que l'agenda joigne les résultats des deux bases. Puis, déchiffre les noms pour sélectionner uniquement les rendez-vous avec Bob. Et enfin, compte le nombre de lignes par jour. Ici, la jointure reconstruit la contrainte associative $\{date, adresse\}$ et le déchiffrement rend lisible la contrainte singleton $\{nom\}$. Par conséquent, à la fois le SaaS agenda et son PaaS d'exécution ne respectent pas la vie privée d'Alice.

Performance. Les temps d'exécutions sont environ deux fois plus rapides que pour l'implémentation par chiffrement symétrique (avec 0.91 sec pour l'opération `[adresse]` et 1.21 sec pour l'opération `#rendezvous`). Les opérations qui n'impliquent pas de déchiffrement s'exécutent plus rapidement (avec

Code 12 – *Requête* `#rendezvous` *pour un agenda centralisé dans un langage à la SQL.*

```
SELECT (jour date), COUNT(*) FROM rendezvous
WHERE nom = 'Bob'
AND adresse = 'Bureau'
AND (date - aujourdhui) in [0..7]
GROUP BY (jour date);
```

un temps moyen de 0,27 sec.). Ces résultats font que l'implémentation par fragmentation verticale est envisageable du point de vue des performances.

3.1.5 Conclusion sur l'emploi des techniques de cryptographie

La figure 13 clarifie l'état de l'art. Chaque sommet du triangle représente un critère exigé par les applications PbD. À savoir, la confidentialité, l'utilisation du nuage et la performance en temps d'exécution. Sur les côtés, sont indiqués en italique les techniques qui soutiennent, dans un certain contexte, les critères des sommets associés.

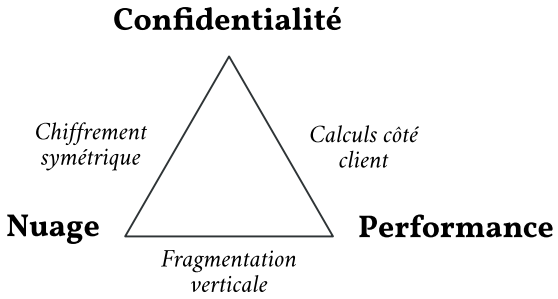


FIGURE 13 – *Les techniques de cryptographie en fonction des critères exigés par une application PbD.*

Au centre du triangle, devrait se tenir les techniques qui satisfont, dans tous les contextes, les trois critères exigés par une application PbD. Malheureusement, l'intérieur du triangle est vide, car au moment de la rédaction de cette thèse, cette approche n'existe pas. La section suivante (§3.2) fait l'hypothèse et teste l'idée que la *composition* des techniques de cryptographie permet de remplir cet espace vide.

3.2 Composer les techniques de cryptographie

La recherche scientifique dans le domaine de la protection intégrée de la vie privée (PbD) s'attelle à trouver la technique parfaite qui remplirait les trois critères : confidentialité, nuage et performance. Cette thèse s'attaque au problème d'une manière différente en proposant de composer toutes les techniques existantes en fonction du contexte.

Par exemple, si l'application a besoin de sauvegarder de l'information personnelle, mais n'a pas besoin de faire de calculs dessus, alors le chiffrement symétrique convient très bien. En revanche, pour une application plus compliquée, faite de plusieurs contraintes de confidentialités, un chiffrement homomorphe partiel, adapté au contexte de l'application, pourrait préserver efficacement les contraintes singletons. Tandis que la technique de la fragmentation verticale pourrait préserver les contraintes associatives tout en conservant un maximum d'informations exploitables. Ceci, à condition qu'aucun acteur du nuage ne puisse accéder aux différents fragments.

Dans la suite, l'idée de composer les techniques est étudiée. L'étude se fait sur l'agenda personnel en ligne avec l'implémentation des opérations `[adresse]` et `#rendezvous` en composant les techniques de calculs côté client, de chiffrement et de fragmentation verticale.

Comme pour la section précédente l'étude teste l'implémentation en fonction des trois critères : confidentialité, nuage et performance. Le code de l'application et de l'évaluation sont disponibles sur le répertoire Git de cette thèse[8]. L'étude révèle que la composition permet de satisfaire les trois critères. Mais, ceci se fait au détriment de la simplicité d'implémentation de l'application. Du coup, les risques de produire une application boguée qui ne protège pas correctement la vie privée sont plus importants.

3.2.1 Composer les techniques pour protéger les contraintes de confidentialités

Pour rappel, le développeur de l'agenda personnel en ligne adopte deux contraintes de confidentialités (*cf.* §2.4.2). La première, notée $\{nom\}$ est une contrainte singleton. Elle empêche un individu malicieux d'obtenir les noms des contacts d'Alice. La seconde, notée $\{date, adresse\}$ est une contrainte associative. Elle empêche un individu malicieux de localiser Alice en associant une date à une adresse.

Avec la composition et en suivant les principes du PbD, le développeur choisit la meilleure technique applicable pour protéger chaque contrainte.

Protéger la contrainte $\{nom\}$

Les attributs de base de données qui sont confidentiels en eux-mêmes, tel que le nom, peuvent être protégés par du chiffrement. Cependant, une requête qui implique un nom (comme `#rendezvous` par exemple) n'est plus exécutable directement sans déchiffrement. En effet, comme le montre la section précédente (*cf.* §3.1.3), l'agenda doit renvoyer à Alice les 3-uplets chiffrés, pour ensuite permettre à Alice de déchiffrer les noms et enfin exécuter la requête sur son terminal. Une approche trop coûteuse pour être applicable.

La solution alternative est d'utiliser un chiffrement homomorphe avec lequel la requête peut être exécutée directement sur la donnée chiffrée. Les temps d'exécutions d'un chiffrement homomorphe total font que cette solution n'est pas envisageable (*cf.* §2.4.3.2). En revanche, le chapitre précédent fait mention de chiffrements homomorphes partiels qui sont très efficaces sur quelques simples opérations. C'est le cas du chiffrement symétrique déterministe AES qui permet de vérifier efficacement l'égalité de valeurs en comparant l'information chiffrée. Ça tombe bien, la requête `#rendezvous` utilise justement l'égalité pour les noms !

D'un point de vue technique, l'implémentation chiffre les noms avec un chiffrement symétrique AES-CBC (Daemen et Rijmen, 1999). Le schéma AES-CBC n'est pas déterministe, mais probabiliste. Il est paramétré par un vecteur d'initialisation que le développeur fait varier à chaque nouvelle donnée chiffrée. Ainsi, chaque chiffrement produit une valeur unique quand bien même le chiffrement chiffre la même donnée avec la même clef. Cette spécificité est problématique pour l'agenda, car elle casse le support de l'égalité. Pour retrouver l'égalité, la valeur du vecteur d'initialisation est fixée dans l'implémentation et est la même pour tous les noms. Une autre solution aurait été d'utiliser le schéma AES-ECB (Daemen et Rijmen, 1999) qui n'a pas de vecteur d'initia-

lisation. Mais, l'API WebCrypto utilisée côté client ne fournit pas d'implémentation. Enfin, le chiffrement utilise une clef de 256 bits et seule Alice connaît la clef.

Protéger la contrainte $\{date, adresse\}$

La fragmentation verticale sépare l'information en fragments non réunissables, de telle manière que seuls les agents autorisés puissent recomposer l'information originale. C'est pourquoi, les attributs de base de données dont l'association est confidentielle, tels que la $date$ avec l'$adresse$, peuvent êtres protégés par la fragmentation verticale.

Le développeur de l'agenda coupe la base de données en deux fragments. Mais, à la différence de la fragmentation verticale précédente (*cf.* §2.4.3.3), ici, le premier fragment contient juste les dates. Le second fragment contient les adresses plus les noms chiffrés avec le schéma AES-CBC. De cette façon, l'agenda ne peut jamais déchiffrer les noms, car il n'est pas en possession de la clef. Enfin, un identifiant est ajouté aux n-uplets de chaque fragment pour pouvoir recomposer les 3-uplets de départ.

Ceci produit deux schémas relationnels $(date, id)$ et $(\texttt{AES}\ nom, adresse, id)$. Dans l'implémentation, la relation $(date, id)$ est sur la base de données PaaS de gauche, hébergée par le IaaS Google Compute Engine. La relation $(\texttt{AES}\ nom, adresse, id)$ est sur la base de données PaaS de droite, hébergée par le IaaS Amazon Web Services.

Du point de vue de la méthodologie, le développeur utilise la technique de la fragmentation verticale seulement pour protéger les contraintes associatives (ici, $\{date, adresse\}$). Il s'en remet à ses choix précédents pour les autres contraintes et compose ainsi les techniques.

Dans le cas de l'agenda personnel en ligne, la composition considère également les calculs côté client pour ne pas faire les calculs sensibles sur l'agenda (à l'inverses de la technique de la fragmentation verticale initiale). L'idée est de faire un maximum de calculs sur les acteurs du nuage quand cela ne pose pas de problèmes de confidentialités. Puis, de finir les calculs chez le client s'il y a un risque pour les données personnelles.

3.2.2 Les opérations `#rendezvous` et `[adresse]` par composition des techniques

Cette section montre comment implémenter les opérations `#rendezvous` et `[adresse]` par composition des techniques de cryptographie. Les implémentations sont évaluées en fonction des mêmes critères qu'avant : confidentialité, nuage et performance. Puis, elles sont comparées aux évaluations de la section précédente (*cf.* §3.1) pour valider l'approche par composition.

Utilisation du nuage. En se basant sur la configuration qui consiste à produire deux relations $(date, id)$ et $(\texttt{AES}\ nom, adresse, id)$, l'agenda personnel en ligne exploite entièrement le nuage. Les fragments de rendez-vous sont distribués sur des bases de données PaaS. Ainsi, le développeur de l'agenda profite des avantages de réplication offerts par le bailleur. Pareillement pour

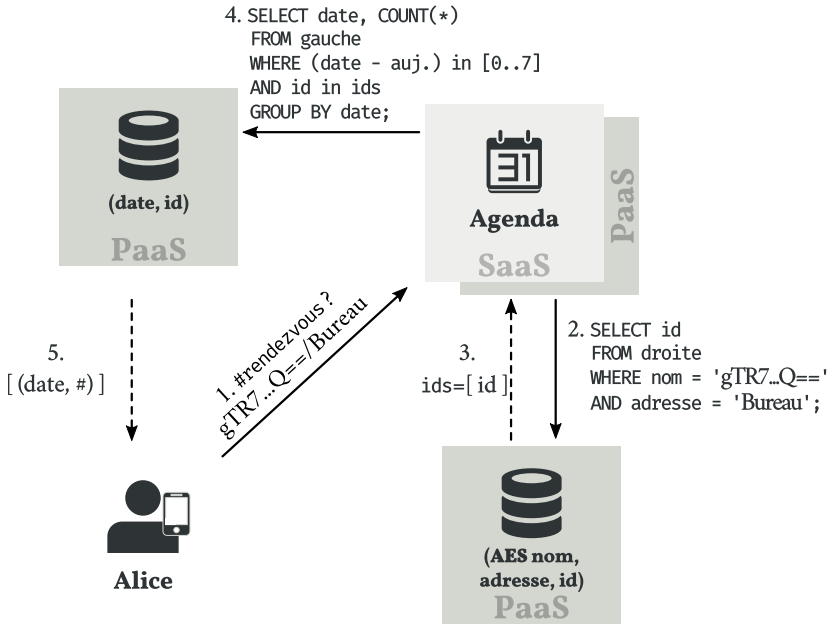


FIGURE 14 – *L'opération* `#rendezvous` *sur l'agenda personnel protégé par la composition de techniques de cryptographie.*

le code de l'agenda qui est fourni en tant que SaaS grâce à un exécutant PaaS. Les interactions pour l'opération `#rendezvous` (*fig.* 14) rendent compte de cette architecture qui tire parti du nuage.

Confidentialité. L'opération `#rendezvous` compte le nombre de rendez-vous pris avec Bob au bureau la semaine prochaine. Pour ce faire, Alice émet d'abord une requête pour l'opération `#rendezvous` (étape 1). Elle fournit en arguments le nom du contact (*c.-à-d.*, Bob en chiffré) et le lieu du rendez-vous (*c.-à-d.*, le bureau). Le nom du contact est chiffré avec un schéma AES-CBC 256 qui supporte le teste d'égalité. Ainsi, lorsque l'opération est distribuée et que la sélection, par l'égalité sur les noms, est faite (2), elle s'applique directement sur la base de données de droite, sans que ni la base ni l'agenda ne violent la contrainte $\{nom\}$. En réponse, la base de données de droite retourne à l'agenda la liste des identifiants des rendez-vous pris avec Bob au bureau (3). L'agenda doit maintenant sélectionner et compter les rendez-vous de la semaine prochaine. Ceci se fait en continuant la requête sur la base de gauche (4) et en limitant les résultats à ceux obtenus dans la requête précédente (*c.-à-d.*, `AND id in ids`). Enfin, le résultat final est retourné à Alice (5).

L'emploi d'une stratégie séquentielle avec un retour direct à Alice sans repasser par l'agenda (étapes 2, 3, 4 et 5) n'est pas anodin. Il permet de préserver la contrainte associative $\{date, adresse\}$. En effet, l'agenda sait grâce aux étapes 2 et 3 que les dates retournées à l'étape 5 sont en relation avec le lieu "Bureau". Ainsi, si une stratégie concurrente était utilisée (comme dans l'exemple initial de la fragmentation verticale, §3.1.4) et que le résultat n'était pas directement retourné à Alice, alors, l'agenda aurait accès aux dates et pourrait localiser Alice. Ici, l'étape 5 est donc primordiale pour que l'agenda soit une application PbD.

Il existe également des opérations où tous les calculs ne peuvent pas être faits dans le nuage sans compromettre la confidentialité des données personnelles. C'est le cas de l'opération `[adresse]` (*fig.* 15) qui a besoin de déchiffrer les noms pour trouver les contacts qui commencent par la lettre C.

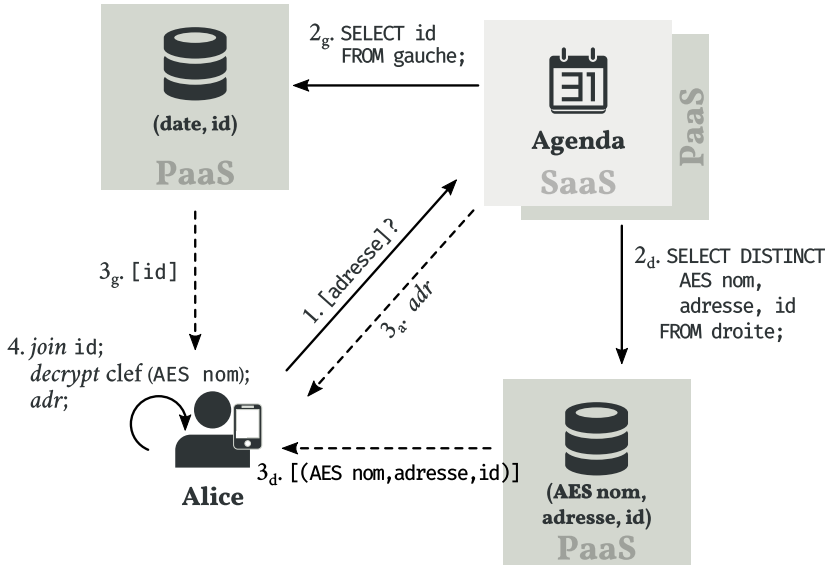


FIGURE 15 – *L'opération* `[adresse]` *sur l'agenda personnel protégé par la composition des techniques de cryptographie.*

Si le déchiffrement des noms est fait sur la base de données de droite, alors la base de données viole la contrainte $\{nom\}$. Il en est de même pour l'agenda. Par conséquent, seule Alice peut déchiffrer les noms grâce à son terminal. En outre, ce calcul côté client n'engendre pas de création d'une preuve à divulgation nulle de connaissance (*cf.* §2.4.3.1), car l'agenda n'a pas besoin du résultat.

Pour résumer, l'idée de la composition consiste à faire un maximum de calculs dans le nuage pour profiter des avantages de celui-ci. En revanche, l'utilisation du nuage peut-être sacrifiée si la confidentialité des données personnelles est en jeux.

Performance. Les implémentations des opérations `[adresse]` et `#rendezvous` sont exécutées par le même système que dans la section précédente (*cf.* §3.1.1).

Les temps d'exécutions pour les deux opérations, sur un échantillon de 2 500 rendez-vous, sont excellents. L'opération `[adresse]` s'effectue en 131.38 ms et l'opération `#rendezvous` s'effectue en 236.09 ms. Des temps qui sont obtenus grâce à des implémentations ingénieuses qui exploitent la composition.

Par exemple, l'opération `[adresse]` qui implique de déchiffrer des noms côté client a un bien meilleur temps d'exécution que l'implémentation par chiffrement symétrique (pour rappel, 1.81 sec). Ceci s'explique par le fait que le terminal d'Alice mémoïse les noms déjà déchiffrés pour payer qu'une seule fois, par nom distinct, le coup de déchiffrement. Une optimisation possible grâce à la propriété homomorphe d'égalité sur les noms chiffrés. Pareillement pour l'opération `#rendezvous` qui a un bien meilleur temps que l'implémentation par fragmentation verticale (pour rappel, 1.21 sec). Encore une fois, la propriété homomorphe d'égalité permet de réaliser la sélection des noms directement sur la base de données (étape 2, *fig.* 14), ce qui est très efficace, car une base de données est conçue pour faire des sélections. Alors que l'implémentation par fragmentation doit déchiffrer les noms et faire la sélection sur l'agenda (étape 4, *fig.* 11).

Le diagramme en bâtons de la figure 16 reprend les temps d'exécutions de la section précédente (*cf.* §3.1). Il y figure les temps pour les calculs côté client en rouge, le chiffrement symétrique en orange et la fragmentation ver-

ticale en bleu. Sur ce diagramme sont aussi présents, en gris, les temps d'exécutions pour une implémentation entièrement exécutée dans le nuage, mais sans confidentialité (à la manière de Google Calendar) et, en violet, les temps de l'implémentation par composition.

Les bâtons gris servent de références pour estimer si une technique remplit le critère de performance ou non. Le diagramme montre que les temps d'exécution pour la composition sont bien meilleurs que ceux de l'implémentation par chiffrement symétrique et ceux de l'implémentation par fragmentation verticale. En faite, c'est temps sont presque équivalents à ceux de l'application exécutée dans le nuage sans confidentialité. Enfin, l'implémentation par composition n'est pas aussi performante que l'implémentation avec calculs côté client. Mais, celle-ci a l'avantage de profiter pleinement du nuage.

Ces résultats montrent qu'avec la composition, les performances d'exécutions sont bonnes. En revanche, ceci se fait au détriment de la simplicité d'écriture de l'application.

3.2.3 La complexité de l'implémentation : le challenge de la composition

Les expérimentations de la section précédente révèlent deux phénomènes suite à la composition. Le premier, positif, est que la composition des techniques de cryptographie permet d'atteindre les trois critères : confidentialité, utilisation du nuage et performance. Le deuxième, négatif, est que la composition des techniques se fait au détriment de la simplicité d'implémentation de l'application.

Pour distribuer la requête, tout d'abord, car le développeur ne peut ni utiliser l'algorithme de di Vimercati et collab. (2013) qui requiert un agenda de confiance, ni les lois algébriques de Özsu et Valduriez (2011) qui n'intègrent pas de fonctions pour la fragmentation, le chiffrement et les calculs côté client (*cf.* §2.4.3.3). Pourtant une bonne distribution augmente les performances. Par exemple, l'étape (4) de l'opération `#rendezvous` (*fig.* 14) optimise en comptant le nombre de rendez-vous sur le fragment de gauche. Une optimisation rendue possible parce que le développeur a choisi une stratégie séquentielle pour joindre les fragments. La stratégie séquentielle fait que les groupes de rendez-vous constitués à l'étape (4) portent sur les rendez-vous effectués avec Bob au bureau (calculé à l'étape 2). Par conséquent, la fonction de dénombrement compte le bon nombre de rendez-vous. En revanche, avec une stratégie

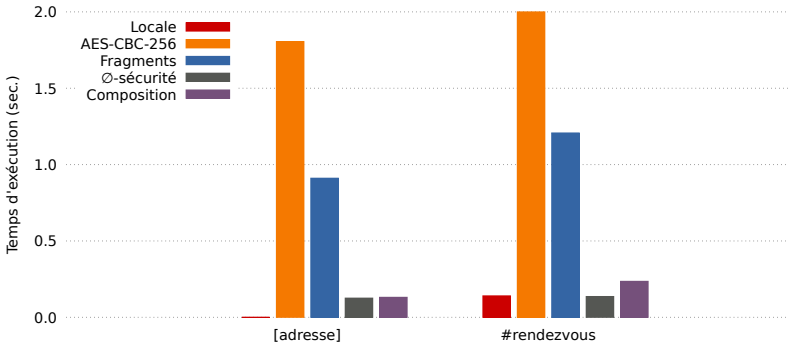


FIGURE 16 – *Temps d'exécution (en seconde) des opérations* `[adresse]` *et* `#rendezvous` *en fonction de la technique de cryptographie.*

concurrente, le compte serait faux, car les groupes formées sur le fragment de gauche ne se limiteraient pas aux rendez-vous effectués avec Bob au bureau. Pour que le compte soit bon, il faudrait défragmenter et compter le nombre de rendez-vous chez Alice. Ceci montre qu'il est difficile de distribuer une requête.

Un autre exemple d'erreur qui peut facilement survenir est lors de l'utilisation d'un chiffrement. Si le développeur avait utilisé un chiffrement probabiliste (*ex.*, ElGamal 1985) pour les opérations `[adresse]` et `#rendezvous`, alors la propriété homomorphe d'égalité sur les noms chiffrés ne serait pas vraie. Le développeur se retrouverait avec une implémentation qui n'a pas de sens.

Enfin, l'opération `#rendezvous` est sûre du point de vue de la vie privée uniquement grâce à l'étape (5) qui empêche l'agenda de voir les dates. Cette interaction entre Alice et la base de données n'existe pas dans la technique initiale de la fragmentation verticale, car l'agenda est considéré comme un agent de confiance. Mais, avec l'approche PbD, l'agenda n'est plus un agent de confiance. Une contrainte qui complique le travail du développeur.

En d'autre termes, dans son effort d'optimisation, le développeur doit toujours s'assurer de la protection des données privées. Composer les techniques de cryptographie introduit de la complexité lors de la conception d'une application PbD. Ceci mène à une nouvelle façon de concevoir les applications, symbolisée dans cette thèse par l'approche du *nuage confidentiel*.

3.3 L'approche du *nuage confidentiel*

Cette thèse détermine quatre caractéristiques nécessaires au bon développement d'une application PbD. Ces quatre caractéristiques sont réunies sous le nom de "approche du *nuage confidentiel*" :

Confidentialité. Une application PbD doit protéger les données personnelles de ses utilisateurs. Une application qui ne respecte pas la confidentialité n'est pas permise.

> *"Il est vain d'essayer de développer une application PbD si la confidentialité des données personnelles de l'utilisateur n'est pas préservée."*

Nuage. L'application s'exécute au maximum dans le nuage pour tirer parti des avantages de celui-ci (disponibilité, élasticité… *cf.* §2.1). Être une application nuagique est plus important que tout le reste, excepté la confidentialité.

> *"Utiliser le nuage libère le développeur des tâches d'intendances bas niveaux. Ceci lui permet de se concentrer sur l'innovation et la logique métier lors de la conception de l'application."*

Performance. La vitesse d'exécution peut être sacrifiée au profit de la confidentialité et de l'utilisation du nuage.

> *"Une application exécutée localement est toujours sûre et a de bonnes performances d'exécution, mais elle ne profite pas des avantages du nuage essentiels aux développeurs d'applications."*

Simplicité. La simplicité peut être sacrifiée au profit de la performance.

> *"Le plus rapidement l'application PbD s'exécute, le mieux c'est. Il n'y a pas de soucis à complexifier le travail du développeur si cela améliore les performances."*

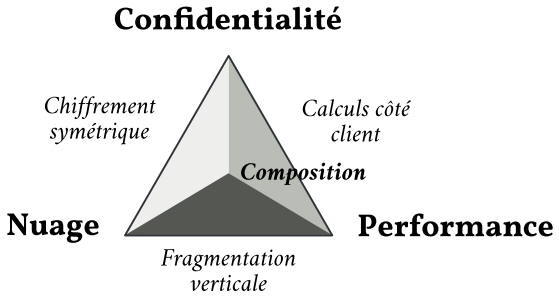


FIGURE 17 – *Approche du nuage confidentiel : une application PbD doit, (1) être sécurisée, (2) au plus exécutée dans le nuage, (3) avoir des performances sacrifiées au profit de la confidentialité et du nuage, (4) voir sa simplicité sacrifiée au profit de la performance.*

L'approche peut-être vue comme une pyramide (*fig.* 17). Être à la base de la pyramide est simple. Y figurent les techniques de calculs côté client, chiffrement symétrique et fragmentation verticale (à la manière de la figure 13). Mais ces techniques ne soutiennent que deux des trois critères confidentialité, nuage et performance.

Pour atteindre les trois critères, le développeur doit monter au sommet de la pyramide et composer les techniques. Ceci se fait au détriment de la simplicité ce qui augmente les risques de produire une application boguée qui ne protège pas correctement la vie privée de ses utilisateurs. Il faut donc un outil pour aider le développeur à suivre cette approche.

3.4 Travaux sur la composition

Comme l'a montré le chapitre sur le contexte, la littérature scientifique se concentre sur la découverte et l'amélioration des techniques de cryptographie (*cf.* §2.4, *chap.* 2). Il existe toutefois quelques initiatives/outils en faveur de la composition.

Le travail le plus remarquable est CryptDB (Popa et collab., 2011), un système de bases de données chiffrées. CryptDB supporte l'exécution de requêtes SQL sur des données chiffrées et se veut aussi compétitif en temps d'exécution que les bases de données commerciales. Pour ce faire, CryptDB repose sur la composition de plusieurs schémas de chiffrements. Certains ont été discutés dans la section sur les techniques de cryptographie (*cf.* §2.4.3.2) : le chiffrement probabiliste, le chiffrement déterministe ainsi que les chiffrements homomorphes partiels Paillier et ElGamal. Les autres schémas disponibles sont le chiffrement à préservation d'ordre (Boldyreva et collab., 2011) et le schéma de recherche (Song et collab., 2000 ; Raykova et collab., 2009). Le premier maintient la relation d'ordre entre deux données chiffrées pour exécuter des requêtes SQL de la forme `ORDER BY`, `MIN`, `MAX` et `SORT`. Le second support la reconnaissance de motifs simples sur du texte chiffré pour exécuter des requêtes SQL de la forme `LIKE`.

L'originalité de CryptDB est de chiffrer les valeurs de la base avec tous ces schémas de chiffrements. Puis, de dynamiquement viser la valeur chiffrée avec le bon schéma de chiffrement en fonction de la requête à exécuter. Par exemple,

une requête qui teste l'égalité visera la valeur chiffrée avec le schéma déterministe. Tandis qu'une requête qui teste l'appartenance d'un motif visera la valeur chiffrée avec le schéma de recherche.

Le surcoût en taille mémoire n'est pas trop important grâce à une gestion intelligente des valeurs chiffrées (entre une et quatre fois plus larges en fonction des applications). Par exemple, il est inutile de chiffrer un entier naturel avec un schéma de recherche. En effet, la recherche de motif sur un entier n'a pas de sens. De même, il est inutile de chiffrer une chaîne de caractère avec le schéma de ElGamal. L'opération de multiplication, offerte par ElGama, n'a pas de sens sur une chaîne de caractère.

Du point de vue de l'architecture, CryptDB se décompose en deux composants principaux. Le premier est un gestionnaire de bases de données (SGBD) qui gère les données chiffrées. Il est surtout responsable de sélectionner la bonne représentation d'une valeur en fonction de la requête à appliquer (*ex.*, sélectionner la valeur chiffrée par un schéma de recherche quand la requête contient un `LIKE`). Le SGBD manipule uniquement des données chiffrées dont il ne possède pas les clefs de déchiffrement. Par conséquent, ni le SGBD ni le serveur qui héberge les bases ne peuvent violer la confidentialité des données de l'utilisateur. Dans le cas de l'application agenda, et plus généralement pour les applications du nuage, cela signifie que la base de données PaaS respecte la politique générale pour la préservation des données personnelles.

Le second composant est un *intermédiaire* (proxy) entre les données confidentielles fournies par l'application et le SGBD. L'intermédiaire à deux fonctions. Il intercepte les requêtes SQL faites par l'application et les réécrit en chiffrant les constantes avec le schéma de chiffrement le plus adapté, avant de transmettre la requête au SGBD. Il déchiffre le résultat de l'exécution d'une requête par le SGBD et transmet le résultat lisible à l'application.

L'intermédiaire est une passerelle entre le monde lisible et le monde chiffré. Par conséquent, une application peut migrer vers une solution où les données sont chiffrées simplement en redirigeant ses requêtes SQL vers l'intermédiaire. Mais, ceci rend CryptDB inutilisable pour les applications PbD du nuage. En effet, à cause de cet intermédiaire, l'application accède aux données de manière lisibles. Dans le cas de l'agenda personnel en ligne, cela signifie que l'application SaaS à un accès lisible aux données d'Alice. Un accès qui est contraire à la politique générale pour la préservation des données personnelles.

Une critique similaire peut être formulée pour les systèmes TrustedDB (Bajaj et Sion, 2011), Cipherbase (Arasu et collab., 2015) et ZeroDB (Egorov et Wilkison, 2016), d'autres implémentations d'un système de bases de données chiffrées.

Un travail radicalement différent, mais tout aussi intéressant est celui d'Antignac et Le Métayer (2015). Les auteurs font le même constat que celui émis dans ce chapitre, à savoir que toutes les techniques de cryptographie fournissent des garanties différentes et sont utilisables dans des contextes différents. Par conséquent, il est difficile pour un développeur de trouver la combinaison de techniques la plus appropriée aux besoins de son application. De surcroît, il est plus difficile pour ce même développeur de savoir si la combinaison des techniques protège l'application.

Un langage d'architecture permet de décrire un système. UML (Booch et collab., 2005) est un exemple de langage d'architecture.

Pour répondre à ce problème Antignac et Le Métayer propose un langage d'architecture pour les applications PbD. Dans un programme écrit avec ce langage apparaissent les besoins de l'application (performance, niveau de confidentialité, *etc.*) et les techniques de cryptographie à disposition du développeur. Ce programme peut ensuite être vérifié grâce à un outil formel qui repose sur une variante de la logique épistémique. L'outil est en mesure de dire si l'architecture choisie satisfait les besoins de l'application ou non.

Eclipse Modeling Framework[10](EMF) est un exemple d'outil qui génère du code Java à partir d'un diagramme UML.

Le choix d'un langage d'architecture pose néanmoins le problème de ne pas être un langage opérationnel. Par conséquent, un programme écrit dans ce langage ne peut pas être exécuté. Bien évidemment, il est possible de produire du code à partir d'un programme écrit dans un langage d'architecture. Ce code servant ensuite d'ébauche pour l'application opérationnelle. Mais ce travail n'est pas soutenu par l'outil d'Antignac et Le Métayer.

Également important, l'outil fournit une vérification semi-automatique. Par conséquent, l'outil *essaye* de montrer que l'architecture satisfait les besoins de l'application, mais peut ne pas y arriver. Le cas échéant, le développeur doit prouver manuellement que son architecture est correcte s'il en à la conviction. La preuve manuelle est conduite dans Why3 (Filliâtre et Paskevich, 2013), une interface vers plusieurs systèmes de preuves (Alt-Ergo, Coq, Z3, *etc.*[11]). Par conséquent, le développeur doit avoir une forte connaissance dans les systèmes formels. Ceci représente une limitation pour l'adoption de l'outil d'Antignac et Le Métayer.

3.5 Conclusion et perspective sur l'approche

La plupart des travaux sur la protection intégrée de la vie privée (PbD) concernent une technique de cryptographie, plutôt qu'un langage compositionnel général. Ce choix, facilite l'écriture de l'application, mais limite son expressivité, puisqu'elle est obligée de faire avec les limites de la technique. Assurément, l'application en pâtit et se retrouve à délaisser l'un des trois critères d'une bonne application PbD : confidentialité, utilisation du nuage ou performance.

Les expérimentations réalisées dans ce chapitre montrent que la composition des techniques de cryptographie permet d'atteindre les trois critères. Cependant, composer complexifie l'écriture de l'application et des erreurs peuvent survenir. Cet aspect du développement est résumé par l'approche du *nuage confidentiel*. Par conséquent, un développeur qui souhaiterait suivre l'approche du *nuage confidentiel* aura besoin d'outils qui l'aident et lui assurent de produire une application correcte.

Le reste de la thèse élabore un outil pour qu'un développeur d'application PbD du nuage puisse suivre l'approche du *nuage confidentiel*. Cet outil se veut à la croisée des travaux présentés à la section précédente. Soit un outil avec une intention opérationnelle pour que le développeur puisse directement programmer son application PbD, à la manière de CryptDB. Mais, aussi avec un cadre formel pour que le développeur ait la garantie que son application com-

10 eclipse.org/modeling/emf

11 La liste exhaustive des systèmes de preuves supportés par Why3 est disponible à l'adresse why3.lri.fr/#provers.

pose correctement les techniques de cryptographie et que l'application résultante ne viole pas la confidentialité d'Alice, à la manière d'Antignac et Le Métayer.

Cet outil est un *langage dédié* (Domain Specific Language – DSL) fonctionnel qui étend l'algèbre relationnelle (*cf.* §2.4.3.3) avec de nouvelles fonctions pour les techniques de cryptographie. L'avantage d'un DSL est qu'il rend le programme plus facile à écrire, raisonner et modifier que son équivalent écrit dans un langage plus général (Hudak, 1998).

Dans ce DSL, nommé C2QL (*Cryptographic Compositions for Query Language*), le développeur programme une fonctionnalité de l'application au moyen d'une requête SQL où apparaissent les techniques de cryptographie à employer. Cette requête SQL abstrait les acteurs du nuage. C'est-à-dire que la requête absorbe et fait disparaître les notions de client, application SaaS et base de données PaaS pour que le développeur se concentre sur l'aspect manipulation des données. Une abstraction primordiale pour raisonner sur la confidentialité de celles-ci.

Dans le même temps, des lois algébriques, à la manière d'Ullman (*cf.* §2.4.3.3 – 1982), spécifient comment les fonctions de cryptographie interagissent avec l'algèbre relationnelle. L'intérêt des lois est double. Premièrement, elles aident le développeur à optimiser les requêtes pour qu'un maximum de calcul soit fait dans le nuage tout en préservant la confidentialité des données personnelles. Deuxièmement, les lois offrent un cadre formel pour raisonner sur la composition des techniques de cryptographie. Ce deuxième point facilite l'ajout de nouvelles techniques.

Concrètement, le reste de la thèse est organisée comme suite. Le chapitre 4 présente le langage et ses lois. Le chapitre 5 encode le langage C2QL dans ProVerif (Blanchet et collab., 2014), un *vérificateur de modèles* (model checker) pour les propriétés de confidentialité dans les protocoles de cryptographies. Cet encodage contrôle qu'une application PbD vérifie la confidentialité des données personnelles. Ceci pour garantir au développeur que le choix des techniques de cryptographie est le bon. Enfin, le chapitre 6 implémente le langage C2QL, à la manière d'un *langage dédié embarqué* (Embedded Domain-Specic Language – EDSL) dans Idris (Brady, 2013). Idris est un langage de programmation fonctionnel avec des types dépendants. L'utilisation des types dépendants permet de garantir, par construction, que la composition des techniques de cryptographie a du sens du point de vue de l'exécution.

4 *Le langage C2QL pour composer les techniques de cryptographie*

Le défi de cette thèse est de permettre l'écriture correcte d'applications du nuage qui préservent la vie privée par composition des techniques de cryptographie. Pour ce faire, ce chapitre présente le langage C2QL (*Cryptographic Compositions for Query Language*), un langage sans-tiers qui étend l'algèbre relationnelle avec des techniques de cryptographie. Le langage aide le développeur à composer les techniques pour protéger une requête exécutée sur le nuage. La distribution de la requête sur les différents acteurs du nuage (Alice, l'application SaaS, les bases de données PaaS) est implicite. Elle est automatiquement dérivée à partir des choix de techniques de cryptographie et garantit que seule Alice peut défaire une cryptographie. Par conséquent, le développeur est assuré qu'une donnée qui est protégée dans le nuage le restera, quelleque soit la requête.

Le dessein du langage C2QL est d'aider le développeur à suivre l'approche du *nuage confidentiel*. Pour ce faire, le langage propose des lois algébriques. Ces lois aident le développeur à raisonner sur les techniques de cryptographie. Par exemple, les lois spécifient quelles sont les conditions nécessaires pour qu'une fonction de l'algèbre relationnelle (projection, sélection, agrégation, *etc.*) s'applique sur une donnée rendue inintelligible par une technique de cryptographie. En d'autres termes, pour le développeur qui écrit une requête, suivre systématiquement ces lois maximise le nombre de fonctions qui sont exécutées sur les données inintelligibles. Et puisque les données inintelligibles peuvent être hébergées dans le nuage sans risque pour la confidentialité, les lois permettent de maximiser le nombre de fonctions exécutées dans le nuage.

Dans la suite, la section 4.1 motive le choix d'étendre l'algèbre relationnelle, présente la syntaxe du langage C2QL et montre comment représenter la requête `[adresse]`. La section 4.2 donne les lois algébriques du langage. Celles-ci spécifient le comportement du langage et dirigent la réécriture d'une requête pour maximiser la quantité de calcul fait dans le nuage sans compromettre la confidentialité. La section 4.3 utilise les lois pour dériver la requête `#rendezvous` de sa version locale vers sa version qui compose les techniques de cryptographie. La section 4.4 spécifie comment une requête C2QL est distribuée sur les acteurs du nuage. Enfin, la section 4.5 définit les limites des lois et présente les défis à relever dans les prochains chapitres.

4.1 Définition du langage C2QL

Cette section présente le langage fonctionnel abstrait C2QL (*Cryptographic Compositions for Query Language*) pour aider le développeur à composer des

techniques de cryptographie lors de l'écriture d'une requête PbD. Le langage étend l'algèbre relationnelle (Ullman, 1982) avec des fonctions de cryptographie. L'avantage est double. Premièrement, les fonctions permettent au développeur de définir une configuration qui spécifie comment les données sont protégées dans la requête. Deuxièmement, les fonctions explicitent le moment où la donnée est *confidentielle* (*c.-à-d.*, lisible), du moment où elle est *inintelligible* (*c.-à-d.*, protégée). En ayant à l'esprit que, pour pallier aux violations de confidentialité, une donnée confidentielle doit être traitée chez la cliente, tandis qu'une donnée inintelligible peut-être traitée dans le nuage. Alors, le langage n'a pas besoin de faire apparaître les notions de cliente, application SaaS et base de données PaaS. Ces informations sont déductibles depuis la position d'application des fonctions de cryptographie.

4.1.1 Pourquoi étendre l'algèbre relationnelle ?

Le dessein de cette thèse est de proposer un outil d'aide à l'écriture d'applications PbD pour les développeurs qui souhaiteraient suivre l'approche du *nuage confidentiel*. Cet outil se présente sous la forme d'un langage dédié, nommé C2QL, pour manipuler les données du nuage et composer les techniques de cryptographie.

Un langage dédié capture précisément la sémantique du domaine d'application. Un programme écrit avec ce langage est plus facile à modifier et il est plus facile de raisonner sur ses propriétés que sur son homologue écrit dans un langage général (Hudak, 1998). Le langage dédié C2QL a deux besoins. Le premier est d'être *agnostique* de la manipulation des données. La requête peut alors être exécutée par les différents acteurs du nuage. Le second est de permettre un *raisonnement équationnel*. Une propriété très pratique pour raisonner, optimiser et vérifier un programme.

Un langage agnostique pour manipuler des données

Un *langage agnostique* d'une implémentation est un langage qui fournit une interface commune et dont la sémantique est applicable par un langage arbitraire via une *liaison* (binding). L'intérêt pour C2QL est qu'une requête va être automatiquement distribuée sur les différents acteurs d'un système PbD à partir des choix de techniques de cryptographie. Par conséquent, les fonctions C2QL responsables de la manipulation des données doivent avoir du sens, qu'elles soient évaluées chez Alice, sur l'application SaaS ou sur les bases de données PaaS.

En ces critères, l'algèbre relationnelle est un bon point de départ pour le langage C2QL. Le dessein de l'algèbre relationnelle est de manipuler les données. Par ailleurs, c'est un langage abstrait avec des implémentations dans plusieurs systèmes. L'implémentation la plus connue est, très certainement, le langage SQL pour les bases de données. Mais, n'importe quelle bonne bibliothèque de liste fonctionnelle[1, 2, 3, 4] propose des fonctions comme `map`, `filter`, `sequence`

1 Pour Haskell, hackage.haskell.org/package/base-4.8.2.0/docs/Data-List.html
2 Pour Scala, scala-lang.org/api/current/index.html#scala.collection.immutable.List
3 Pour Java, docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
4 Pour JavaScript, lodash.com

et `fold` qui sont les équivalents de la projection (π), la sélection (σ), du produit cartésien ($\times$) et des fonctions d'agrégations.

Un langage avec un raisonnement équationnel

Dans la programmation fonctionnelle, un *raisonnement équationnel* est une méthodologie pour raisonner sur l'égalité des programmes en se basant sur des lois algébriques (Schrijvers et Demoen, 2008). Une loi algébrique spécifie l'égalité sémantique entre deux *modèles de programme* (code patterns). Elle rend ainsi possible la transformation d'un programme vers un autre sémantiquement équivalent. Par ailleurs, l'ensemble de ces lois spécifie la nature du langage. C'est-à-dire qu'elles définissent comment celui-ci se comporte. Par conséquent, un développeur qui connaît ces lois n'a pas besoin de comprendre comment fonctionne l'interpréteur du langage pour comprendre ce que fait le programme. L'intérêt pour C2QL est multiple.

Premièrement, il est possible de définir des lois qui spécifient comment les techniques de cryptographie interagissent entre elles. Ainsi, le développeur à l'assurance de composer correctement ces techniques en suivant leur lois.

Deuxièmement, les lois vont permettre d'optimiser la requête. L'optimisation vise à ce que la requête soit au maximum évaluée par l'application SaaS et les bases de données PaaS, sans compromettre la confidentialité des données personnelles. Ce mécanisme pourrait être un algorithme de transformation, à la manière des travaux sur la fragmentation verticale (Aggarwal et collab., 2005 ; di Vimercati et collab., 2013). Seulement, composer les techniques de cryptographie implique, non pas une, mais plusieurs transformations. Elles sont toutes sécurisées, mais ont des optimisations différentes entre la quantité d'information calculée dans le nuage et la performance d'exécution. Prendre une transformation, plutôt qu'une autre, ne doit pas être une obligation, mais un choix. Un choix dirigé par les besoins de l'application. Cette thèse affirme que le développeur connaît les besoins de son application. Ainsi, au lieu de lui proposer une solution unique, il est préférable de lui fournir des lois pour qu'il conduise l'écriture de l'application vers une optimisation plutôt qu'une autre.

Enfin, l'objectif du langage C2QL est de composer les techniques de cryptographie. Trois techniques sont incluses dans la définition qui suit. Mais, il est indispensable que le langage soit facile à étendre avec plus de techniques et que l'ajout soit relatif aux propriétés de confidentialité définies par C2QL. De nouveau, avoir un langage avec raisonnement équationnel facilite le travail. L'ensemble des lois spécifie le comportement du langage. Par conséquent, le langage et ses lois peuvent être vus comme un cadre formel qui facilite le raisonnement. Dans ce cadre, étendre le langage consiste à donner de nouvelles lois qui spécifient comment une nouvelle technique interagie avec le reste du langage.

Là encore, l'algèbre relationnelle est un bon point de départ pour le langage C2QL. Les travaux d'Ullman (1982) et Özsu (2011) donnent déjà plusieurs lois sur les fonctions de l'algèbre relationnelle (*cf.* §2.4.3.3). Par exemple, certaines de ces lois commutent les fonctions de l'algèbre relationnelle pour réorganiser la requête sans changer sa sémantique.

4.1.2 L'environnement d'exécution d'une requête C2QL

Lorsqu'une requête C2QL est correctement protégée, elle est vouée à s'exécuter dans un nuage constitué de trois acteurs : les bases de données PaaS, l'application SaaS et la cliente. Par exemple, pour les requêtes `[adresse]` et `#rendezvous` les trois acteurs sont les fragments de gauche et droite, l'application agenda et Alice. Mais une requête C2QL ne fait pas apparaître les acteurs.

Ce type de langage de programmation qui permet de développer pour toute la pile du nuage dans un seul programme est dit *sans-tiers* (tierless – Philips et collab., 2014 ; Nelson et collab., 2014). L'avantage d'un langage sans-tiers est qu'il cache tout le code pour coordonner les tiers. Ceci pour que le développeur se concentre sur les aspects importants de son application. Dans le cas de C2QL, ne pas faire apparaître les acteurs du nuage et les interactions entre eux donne un langage qui est orienté sur la manipulation des données. Une orientation qui aide à raisonner sur la confidentialité des données.

Généralement, des annotations dans le programme identifient les différents tiers et une transformation produit le code pour la coordination. Dans le langage C2QL, c'est la position des fonctions de cryptographie qui sert à identifier les tiers. La suite explique, de manière non formelle, comment est faite cette identification. Plus tard, la section 4.4 formalise ceci en donnant la transformation d'une requête C2QL vers le π-calcul.

Identifier les bases de données PaaS. Les bases de données PaaS hébergent les données de la cliente. Ces bases ne sont pas des acteurs de confiances. Or, ces bases sont destinées à héberger des données confidentielles. Par conséquent, l'exécution d'une requête C2QL considère un environnement avec des bases de données PaaS que si les données sont protégées par au moins une fonction de cryptographie.

Identifier l'application SaaS. L'application SaaS joue le rôle d'intermédiaire entre la cliente et les bases de données. Lorsque la cliente demande à réaliser une requête, sa demande passe par une application (*c.-à-d.*, l'agenda) qui chorégraphie l'appel des différentes bases de données. L'application est également responsable de fournir le code de la cliente si cette dernière doit faire des calculs chez lui. L'environnent d'exécution contient systématiquement l'application SaaS. En revanche, l'application n'est pas un acteur de confiance. Par conséquent, seules les données inintelligibles sont autorisées à transiter par cet acteur lors de l'évaluation d'une requête C2QL.

Identifier la cliente. La cliente est le seul acteur de confiance dans ce système. Une requête s'évalue chez la cliente dès qu'il y a un risque pour les données confidentielles. D'ailleurs, une requête C2QL qui ne spécifie pas de fonction de cryptographie considère uniquement la cliente dans son environnent (et l'application SaaS pour récupérer le code). La requête est entièrement évaluée chez cette cliente. C'est le seul moyen de préserver les données confidentielles quand elles ne sont pas protégées.

4.1.3 La syntaxe du langage C2QL

Cette section donne la syntaxe du langage C2QL. Le langage C2QL étend l'algèbre relationnelle avec les techniques de cryptographie étudiées dans cette thèse. Chaque technique se présente sous la forme d'une paire de fonctions qui s'applique sur une relation (table). La première fonction se nomme un *spécificateur*. Elle spécifie comment la relation doit être protégée. Après cette fonction, les n-uplets de la relation sont considérés inintelligibles dans la limite des garanties offertes par la technique de cryptographie. La deuxième fonction se nomme un *destructeur*. Elle détruit la protection. Par conséquent, après cette fonction, les n-uplets de la relation sont considérés comme confidentiels.

Ces spécificateurs/destructeurs de techniques de cryptographie se composent avec les fonctions de l'algèbre relationnelle pour décrire une requête qui manipule des n-uplets du nuage. Dans la requête, une fonction de l'algèbre relationnelle manipule soit des n-uplets inintelligibles, soit des n-uplets confidentiels, en *fonction de sa position* par rapport aux spécificateurs et aux destructeurs de techniques.

Pour rappel (*cf.* §2.4.3.3, *fig.* 7), les fonctions de l'algèbre relationnelle se composent selon la grammaire ci-après. Avec π_δ la projection sur les δ ; σ_{p_δ} la sélection avec un prédicat p sur les δ ; $(\times)$ et $(\bowtie)$ respectivement le produit cartésien et la jointure naturelle de deux relations ; et $count_\delta$ l'agrégation par dénombrement sur les δ.

$$Q ::= Q \circ Q \mid \pi_\delta \mid \sigma_{p_\delta} \mid (\times) \mid (\bowtie) \mid count_\delta$$

Pour former le langage C2QL, les fonctions de l'algèbre relationnelle se composent avec les spécificateurs et destructeurs de techniques de cryptographie, selon la grammaire figure 18. Le sens des spécificateurs et destructeurs est donné dans la suite.

$$Q_c ::= Q \mid Q_c \circ Q_c \mid crypt_{\alpha,\mathsf{c}} \mid decrypt_{\alpha,\mathsf{c}} \mid frag_\delta \mid defrag_\delta\ (Q_c, Q_c)$$

FIGURE 18 – *La syntaxe de C2QL.*

Les spécificateurs de techniques de cryptographie

Un spécificateur de technique de cryptographie est placé en début de requête pour spécifier comment la relation est protégée. Le langage C2QL propose deux spécificateurs, un pour le chiffrement et un pour la fragmentation.

Le spécificateur de chiffrement, noté $crypt_{\alpha,\mathsf{c}}$, spécifie que sur une relation de type Δ, les valeurs de l'attribut α (avec $\alpha \in \Delta$) sont chiffrées avec le schéma de chiffrement c. Conséquemment, les valeurs de l'attribut α sont inintelligibles. Par exemple, appliquer $crypt_{nom,\mathsf{AES}}$ à la relation des rendez-vous spécifie que les contacts sont chiffrés avec un chiffrement AES. Par conséquent, les noms des contacts sont inintelligibles. Toutefois, les contacts pourront toujours être comparés, car le chiffrement AES supporte le test d'égalité.

Le spécificateur de fragmentation, noté $frag_\delta$, spécifie qu'une relation de type Δ est séparée en deux, de telle manière que le premier fragment contienne les valeurs des attributs spécifiés par δ (avec $\delta \subseteq \Delta$) et que le deuxième fragment contienne les valeurs des attributs restants (soit $\Delta \setminus \delta$). De ce fait, les

associations d'attributs sensibles entre les deux fragments sont inintelligibles. De plus, le spécificateur suppose qu'un identifiant indexe les n-uplets dans chaque fragment pour rendre possible la reconstruction des n-uplets d'origine.

La fragmentation se généralise facilement à plus de deux fragements. Ceci est montré dans la section 4.2.6 avec la loi 23.

Appliquer le spécificateur $frag_{\delta}$ sur une relation de type Δ produit une paire de relations de type $(\delta, \Delta \backslash \delta)$. Le type ne fait pas apparaître les identifiants dans chaque fragment, car ceux-ci sont gérés implicitement par le spécificateur. Par exemple, appliquer le spécificateur $frag_{date}$ sur la relation des rendez-vous spécifie que la relation est séparée en deux fragments. Le premier fragment contient les dates et le second les noms et les adresses $((date, nom, adresse) \setminus date)$. Par conséquent, l'association sensible $\{date, adresse\}$ est inintelligible. En outre, les dates et les adresses restent lisibles dans leur fragment respectif, ce qui permet d'appliquer des fonctions de l'algèbre relationnelle sur ces valeurs.

Le terme "spécificateur" n'est pas anodin. Un spécificateur ne modifie pas la relation, mais spécifie dans quel état (au sens informatique du terme) la relation doit être pour que la requête s'évalue. Du point de vue de l'environnement d'exécution (*cf.* §4.1.2), un spécificateur *spécifie l'environnement de départ* d'une requête C2QL.

Les destructeurs de techniques de cryptographie

Un destructeur de technique de cryptographie est le dual du spécificateur. Dans la requête, le destructeur se situe quelque part après son spécificateur. Avant lui, les n-uplets sont inintelligibles grâce à la technique de cryptographie spécifiée par le spécificateur. Après lui, les n-uplets sont confidentiels puisque le destructeur détruit la technique. Ainsi, le destructeur de déchiffrement $decrypt_{\alpha,\mathsf{c}}$ apparaît après le $crypt_{\alpha,\mathsf{c}}$. Il déchiffre les valeurs de l'attribut α qui étaient chiffrées avec le schéma de chiffrement c. De même, le destructeur de fragmentation $defrag_{\delta}$ apparaît après le $frag_{\delta}$. Il joint les deux fragments et reconstitue les n-uplets grâce à leur identifiant.

Du point de vue de l'environnement d'exécution (*cf.* §4.1.2), un destructeur est toujours évalué chez Alice, et avec lui, le reste de la requête. La raison est la suivante : un destructeur détruit la protection introduite par le spécificateur. De ce fait, les n-uplets confidentiels, qui étaient précédemment inintelligibles, sont maintenant lisibles. Dorénavant, le risque pour ces n-uplets est qu'un individu malveillant viole les contraintes de confidentialités. C'est pourquoi seule Alice peut appliquer un destructeur et, qu'à partir de ce moment, les n-uplets ne peuvent plus sortir de chez elle. Le langage garantit ainsi que tous les n-uplets qui transitent sur le nuage sont inintelligibles par rapport à leurs spécificateurs.

Il n'existe pas de fonction explicite pour rapatrier la relation chez la cliente et y effectuer des calculs (à la manière des calculs côté client, *cf.* §2.4.3.1). Cet aspect est impliqué implicitement par l'évaluation d'un destructeur. La technique des calculs côté client est donc présente dans le langage, mais dictée par les destructeurs de techniques de cryptographie.

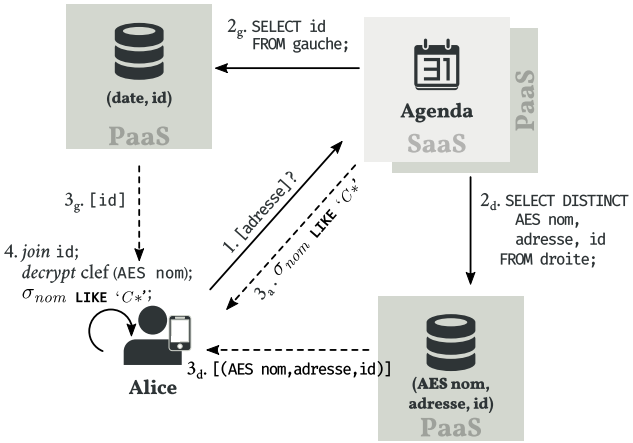
4.1.4 La requête [adresse] en C2QL

Cette section utilise le langage C2QL pour représenter la requête [adresse] vue au chapitre précédent. Ceci montre que le langage a le bon niveau d'abstraction pour programmer les requêtes d'un système PbD avec composition des techniques de cryptographie. En particulier, cette section montre que ne pas faire figurer les tiers aide le développeur à se concentrer sur la manipulation des données.

L'environnement d'exécution pour la requête [adresse] est le même que celui pour la composition des techniques de cryptographie présenté à la section 3.2 du chapitre 3. Premièrement, la relation des rendez-vous est séparée en deux par la technique de fragmentation verticale. Le fragment de gauche contient les dates. Le fragment de droite contient les noms et les adresses. Deux bases de données PaaS, indépendantes l'une de l'autre, hébergent les fragments. De ce fait, le lien entre les dates et les adresses est inintelligible, ce qui protège la contrainte $\{date, adresse\}$. Deuxièmement, un chiffrement symétrique déterministe (*c.-à-d.*, AES) protège les noms dans le fragment de droite. Les noms sont donc inintelligibles, ce qui protège la contrainte $\{nom\}$.

La requête [adresse] retourne la liste des adresses et contacts dont le nom commence par la lettre 'C'. Le chapitre précédent explique comment implémenter cette requête dans un système qui compose les techniques de cryptographie (*cf.* §3.2.2, *fig.* 15). L'explication ici reprend la figure 15 pour illustrer graphiquement l'interprétation de la requête [adresse] en C2QL.

$$\sigma_{nom\ \texttt{LIKE}\ \text{`}C*\text{'}} \circ decrypt_{nom,\texttt{AES}} \circ defrag_{date}\ (\pi_{\emptyset}, \pi_{nom,adresse}) \circ frag_{date} \circ crypt_{nom,\texttt{AES}}\ rendezvous$$



Le langage C2QL est un langage fonctionnel. Une requête C2QL est définie par la composition (∘) des spécificateurs, destructeurs et fonctions de l'algèbre relationnelle. Les fonctions de la requête s'interprètent de droite à gauche (et de bas en haut) sur une relation. Chaque fonction utilise la relation produite par la fonction précédente comme argument. Une exception, toutefois, pour le *frag* qui spécifie que la relation est un couple de relations et son dual le *defrag* qui prend un couple de relations en argument.

Voici la signification cette requête :

— Dans un environnement où les noms de la relation des rendez-vous sont chiffrés avec AES et où la relation est fragmentée sur les dates, par la fonction suivante :

$$\circ\ frag_{date} \circ crypt_{nom,\texttt{AES}} \quad rendezvous$$

— Projette sur le nom et l'adresse du rendez-vous dans le fragment de droite, et projette sur l'ensemble vide dans le fragment de gauche, par la fonction suivante :

$$(\pi_{\emptyset}, \pi_{nom,adresse})$$

Les indices sont gérées implicitement puisque les fonctions sont évaluées sur des fragments.

— Rapatrie de manière concurrente le résultat des deux fragments chez Alice et fais la défragmentation, puis déchiffre les noms des contacts, par la fonction suivante :

$$decrypt_{nom,\texttt{AES}} \circ defrag_{date}$$

— Enfin filtre sur les noms qui commencent par la lettre 'C', par la fonction suivante :

$$\sigma_{nom\ \texttt{LIKE}\ `C*'}$$

Comme précisée à la section précédente, la présence de spécificateur dans la requête présuppose un environnement où la relation est sauvegardée dans une/des bases de données PaaS. Ici, les spécificateurs $frag_{date}$ et $crypt_{nom,\texttt{AES}}$ présupposent que la relation $rendezvous$ est sauvegardée dans deux bases de données PaaS avec les schémas relationnels $(date, id)$ et (`AES` $nom, adresse, id$).

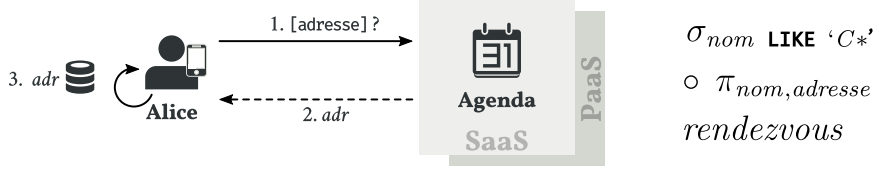
Les fonctions de l'algèbre relationnelle qui viennent après les spécificateurs (*ex.*, $\pi_{nom,adresse}$) s'appliquent toujours sur les bases de données. Il n'y a pas de risque pour la confidentialité des données, car les n-uplets sont rendus inintelligibles avec les techniques de cryptographie.

En revanche, certaines fonctions de l'algèbre relationnelle ne sont pas applicables sur les n-uplets inintelligibles. C'est le cas de $\sigma_{nom\ \texttt{LIKE}\ `C*'}$, dont l'*expression régulière* (regexp) `LIKE` $`C*'$ requiert des noms lisibles. Cette fonction est donc effectuée après le $decrypt_{nom,\texttt{AES}}$.

Cet exemple montre que le langage C2QL permet de représenter les requêtes d'une application PbD par composition des techniques de cryptographie. Le langage se concentre sur la manipulation des données. Dès lors, le développeur peut prédire la quantité d'information calculée dans le nuage d'un seul coup d'œil. Il lui suffit de regarder la position des spécificateurs et destructeurs.

Toutefois, le langage ne dédouane pas entièrement le développeur du coup de réflexion. En effet, lorsqu'il écrit sa requête, le développeur doit se demander si la fonction de l'algèbre relationnelle est applicable sur des n-uplets inintelligibles ou non. Comme avec la fonction $\sigma_{nom\ \texttt{LIKE}\ `C*'}$ par exemple.

Heureusement, le langage permet de représenter des implémentations plus naïves qui sont très faciles à écrire. C'est le cas de la version suivante de la requête `[adresse]` qui est évaluée dans un environnement sans spécificateurs.



La requête ne possède pas de spécificateurs. Par conséquent, l'environnement est considéré comme non sécurisé et la relation des rendez-vous est conservée chez Alice. Cette représentation est facile à écrire pour le développeur. Il n'a pas à se demander si les fonctions de l'algèbre relationnelle sont applicables sur des n-uplets inintelligibles vu que tous les rendez-vous sont sauvegardés de manière lisible chez Alice. En contrepartie, la requête ne profite pas des avantages du nuage. L'interprétation est équivalente à celle où tous les calculs sont faits chez la cliente (*cf.* §3.1.2, *chap.* 3).

La section suivante outille le langage C2QL avec des lois algébriques. Ces lois spécifient sous quelles conditions une fonction de l'algèbre relationnelle s'applique sur des n-uplets inintelligibles. Elles aident le développeur à dériver, systématiquement, à partir de la version naïve la version optimisée de la requête.

4.2 Lois algébriques du langage C2QL

Une loi algébrique est une relation d'égalité pouvant servir de transformation qui, si elle est suivie, garantit au système la correction du résultat de la transformation. Les lois algébriques permettent deux choses. Premièrement, de réécrire le système sans modifier sa sémantique. Deuxièmement, de spécifier de manière *compréhensible* son comportement.

Dans son article sur le *système FP*, Backus (1978) dit, des lois algébriques, qu'elles ne requièrent pas une connaissance profonde de la logique et des mathématiques. Elles sont simples à comprendre. Une simplicité qui aide à prouver facilement et de manière systématique (voir mécanique) le bon comportement du système.

Pour l'algèbre relationnelle, plusieurs lois existent déjà (Ullman, 1982 ; Özsu et Valduriez, 2011). Elles sont données dans la section 2.4.3.3 du chapitre 2. De manière générale, les lois de l'algèbre relationnelle spécifient dans quel ordre une fonction peut s'effectuer par rapport à une autre. C'est pourquoi les gestionnaires de bases de données utilisent ces lois pour optimiser l'évaluation d'une requête.

Cette section augmente les travaux précédents et définit de nouvelles lois qui spécifient comment les opérateurs de l'algèbre relationnelle et les fonctions de cryptographie interagissent entre elles. En particulier, comment elles commutent.

L'idée est d'utiliser ces lois pour reporter les fonctions qui suppriment une protection le plus tard possible dans la requête. Ce qui, intuitivement, signifie qu'il y a plus de calculs faits dans le nuage que chez la cliente. Le tout, sans altérer la confidentialité de la requête.

Avec l'emploi des lois, le développeur d'une application PbD peut partir d'une requête sans protection, comme la requête `#rendezvous`.

$$count_{date} \circ \pi_{date} \circ \sigma_{(date - aujourdhui) \in [0..7] \wedge adresse = \text{`}Bureau\text{'} \wedge nom = \text{`}Bob\text{'}}$$

Et la transformer vers la même requête qui suit l'approche du *nuage confidentiel* en composant les techniques de cryptographie et en évaluant les destructeurs le plus tard possible dans le calcul. Le résultat est la requête `#rendezvous`

suivante qui modélise la version "*nuage confidentiel*" du chapitre 3 (*cf.* §3.2.2, *fig.* 14).

$$defrag_{\,date}\ (count_{\,date} \circ \pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]},\ \pi_{\emptyset} \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge \mathsf{AES} \Rightarrow nom=\text{`}Bob\text{'}}) \circ frag_{\,date} \circ crypt_{nom,\mathsf{AES}}$$

La transformation s'opère de manière systématique et préserve la protection des données privées imposée par l'environnement. Du coup, le développeur n'a plus à réfléchir à comment composer les techniques de façon ingénieuse pour que la requête soit optimisée sans violer la vie privée de ses utilisateurs, comme c'est le cas dans le chapitre précédent (*cf.* §3.2.3). Mais, simplement réfléchir à quelles sont les techniques qu'il veut composer. Puis se laisser guider par les lois pour optimiser.

Les lois de composition sont exprimées à la manière des lois de composition du *système FP* de Backus. Dans la suite, Δ représente le schéma relationnel sur lequel les requêtes s'évaluent. Par exemple dans l'exemple précédent, Δ est le schéma $(date, nom, adresse)$. La lettre α représente un attribut membre du schéma Δ, *ex.,* nom. Les lettres δ et δ' représentent des schémas relationnels inclus dans Δ, *ex.,* $(nom, adresse)$. Le c en police à chasse fixe (comme dans $crypt_{\alpha,\mathsf{c}}$), représente un schéma de chiffrement quelconque. Enfin, id est la fonction identité.

4.2.1 L'équivalence de deux requêtes C2QL

Lorsqu'une loi algébrique est appliquée sur une requête relationnelle, elle garantit que la requête transformée retournera un résultat équivalent. Il est donc important, avant de donner les lois, de définir ce qu'est l'équivalence de deux requêtes C2QL.

Équivalence observationnelle. Premièrement, le langage C2QL préserve la notion d'équivalence de l'algèbre relationnelle. C'est-à-dire, que deux requêtes r et s sont équivalentes (noté, $r \equiv s$), si pour n'importe quelle relation (table) R où rR est un terme valide, alors sR est aussi un terme valide qui produit la même valeur. La notion de *même valeur* signifie que les relations, après application des requêtes r et s, partagent le même schéma relationnel et le même ensemble de valeurs d'attributs, de telle manière qu'il est impossible, dans le système, de distinguer la relation calculée par la première requête, de celle calculée par la deuxième.

Équivalence de confidentialité. La définition précédente spécifie l'équivalence du point de vue du calcul d'un résultat. Mais, ce n'est pas suffisant pour le langage C2QL qui requiert, en plus, une définition d'équivalence du point de vue de la confidentialité. Deux requêtes r et s sont équivalentes, si pour n'importe quelle relation R avec des attributs confidentiels où rR s'applique sur ces attributs, alors sR préserve la confidentialité de ces attributs.

Enfin, la notion d'équivalence ne porte pas sur l'environnement puisque les lois peuvent introduire des spécificateurs.

4.2.2 Lois d'identité

Les lois d'identité servent à introduire une technique de cryptographie dans la requête lorsqu'elle est appliquée de gauche à droite. Elles spécifient que les spécificateurs et destructeurs sont duals. Appliquer un spécificateur puis appliquer son destructeur ne fournit aucune protection.

$$id \equiv decrypt_{\alpha,\mathsf{c}} \circ crypt_{\alpha,\mathsf{c}} \qquad \text{si } \alpha \in \Delta \tag{8}$$

$$id \equiv defrag \circ frag_{\delta} \qquad \text{si } \delta \subseteq \Delta \tag{9}$$

FIGURE 19 – *Lois d'identité.*

Par exemple, l'application de l'équivalence 8 sur la requête `#rendezvous` a pour effet de spécifier un environnement chiffré, puis de déchiffrer les noms de la relation.

$$\begin{aligned} count_{date} \circ \pi_{date} \\ \circ\, \sigma_{(date-aujourdhui)\in[0..7]\wedge adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}} \\ \circ\, decrypt_{nom,\mathsf{AES}} \circ crypt_{nom,\mathsf{AES}} \end{aligned}$$

Du point de vue du comportement de l'application, l'introduction du $crypt$ suggère un environnement avec une base de données PaaS. Alors que la version sans protection (*c.-à-d.*, qui ne fait pas apparaître le $decrypt_{nom,\mathsf{AES}} \circ crypt_{nom,\mathsf{AES}}$) à toutes ses données hébergées chez la cliente. Néanmoins, l'équivalence est respectée entre la version avec et sans protection.

C'est trivial pour l'équivalence observationnelle puisqu'après le $decrypt$ les valeurs des attributs sont de nouveau lisibles comme dans la version sans protection et que le reste de la requête est identique à celle sans protection. Par conséquent, les requêtes avec ou sans protection produisent le même résultat.

Pour l'équivalence de confidentialité, les noms sont confidentiels dans la base de données PaaS par le $crypt$ qui les rend inintelligibles. Les noms restent confidentiels après le $decrypt$ car celui-ci force l'évaluation de la requête chez Alice. Par conséquent, un attaquant ne peut pas lire les noms. Tout comme pour la requête sans protection où un attaquant ne peut pas lire les noms hébergés chez Alice.

Dans l'équivalence 9, la défragmentation devrait être écrite $defrag_{\delta}(id, id)$ pour respecter la grammaire. Mais, la notation sans la paire est préférée lorsque l'information dans les fragments n'est pas utile. Ceci pour gagner en clarté et sauvegarder de l'espace.

4.2.3 Lois de projection

La fonction de projection π_{δ} conserve un sous-ensemble d'attributs δ du schéma relationnel Δ. Les lois de projection déclarent comment une projection commute avec les destructeurs. En particulier, la partie droite de l'équivalence spécifie les conditions pour que la projection s'applique sur des n-uplets inintelligibles. Ceci pour échanger l'ordre d'application de la projection par rapport au destructeur sans changer le résultat final de l'évaluation.

Une requête peut utiliser ces lois pour "pousser" le destructeur. Ainsi, en accord avec les deux premiers principes de l'approche du *nuage confidentiel*

(*cf.* §3.3), une plus grande partie de la requête est évaluée dans le nuage sans affecter la confidentialité de celle-ci.

FIGURE 20 – *Lois de projection.*

$$\pi_\delta \circ decrypt_{\alpha,\mathsf{c}} \equiv decrypt_{\alpha,\mathsf{c}} \circ \pi_\delta \qquad \text{si } \alpha \in \delta \tag{10}$$

$$\pi_\delta \circ decrypt_{\alpha,\mathsf{c}} \equiv \pi_\delta \qquad \text{si } \alpha \notin \delta \tag{11}$$

$$\pi_\delta \circ defrag_{\delta'} \equiv defrag_{\delta'}\ (\pi_{\delta\cap\delta'}, \pi_{\delta\setminus\delta'}) \tag{12}$$

Commutativité sur le chiffrement (*eq.* 10, 11). Pour réduire le nombre d'attributs de Δ à δ, la projection n'a pas besoin de lire leur valeur. Par conséquent, la projection peut s'opérer sur des attributs chiffrés. L'équation 10 spécifie que la projection sur δ et le déchiffrement d'un α commute quelle que soit le chiffrement c considéré.

Une optimisation est possible si l'attribut chiffré n'apparaît pas dans le résultat de la projection ($\alpha \notin \delta$, *eq.* 11). Dans ce cas, le chiffrement peut être enlevé.

Commutativité sur la fragmentation (*eq.* 12). Pour que la projection traverse la défragmentation, elle doit être distribuée sur chaque fragment. Il faut donc séparer la projection en deux, ce qui est possible avec la loi de cascade sur la projection (*eq.* 3, Ullman, 1982). Reste à savoir sur quels attributs projeter dans les fragments de gauche et droite. Pour le savoir, il faut regarder la fonction de défragmentation.

La fonction de défragmentation $defrag_{\delta'}$ dit que le fragment de gauche se compose des attributs contenus δ'. La projection relative est donc $\pi_{\delta\cap\delta'}$ pour que la projection soit réduite aux attributs présents dans le fragment de gauche. Dans le cas du fragment de droite, la projection réduit ses attributs à ceux absents dans le fragment de gauche, soit $\pi_{\delta\setminus\delta'}$. Ce comportement est spécifié par l'équation 12.

Par exemple, l'équation 12 s'instancie de la manière suivante pour la relation des rendez-vous avec une projection sur les dates et les noms, et une fragmentation sur les dates.

$$\pi_{date,nom} \circ defrag_{date} \equiv defrag_{date}\ (\pi_{date}, \pi_{nom})$$

Ainsi instanciée, l'équation dit que défragmenter puis faire une sélection sur les attributs date et nom, est équivalent à faire, une projection des dates dans le fragment de gauche et une projection des noms dans le fragment de droite puis défragmenter.

La gestion des identifiants produits par la fragmentation verticale est implicite. Toutes les fonctions évaluées pendant la fragmentation (*c.-à-d.*, à l'intérieur de la paire) maintiennent implicitement les identifiants et la fonction de défragmentation les détruit implicitement. Cette gestion implicite évite d'introduire une projection supplémentaire qui nettoie les identifiants à la fin de la requête, comme c'est le cas lors du calcul de la requête distribuée dans la fragmentation verticale (*cf.* §2.4.3.3).

Enfin, les projections qui rentrent dans le $defrag$ en partie droite de l'équivalence 12 se composent fonctionnellement avec les précédentes fonctions

déjà présentes dans le $defrag$. Ainsi, une représentation exacte de l'équivalence 12 est la suivante (avec r et s des termes quelconques du langage C2QL).

$$\pi_\delta \circ defrag_{\delta'}\ (r, s) \equiv defrag_{\delta'}\ (\pi_{\delta \cap \delta'} \circ r, \pi_{\delta \setminus \delta'} \circ s)$$

Dans la suite, la première notation sera préférée à la seconde pour gagner en clarté et sauvegarder de l'espace. La règle est toujours la même : lorsqu'un $defrag$ apparaît sans sa paire en partie gauche de l'équivalence, les fonctions membres de cette paire sont composées fonctionnellement ($\circ$) avec les fonctions à l'intérieure du $defrag$ en partie droite de l'équivalence.

4.2.4 Lois de sélection

La fonction de sélection σ_{p_δ} filtre les n-uplets de la relation Δ avec un prédicat p qui porte sur les attributs δ. Tout comme les lois de projection, les lois de sélection spécifient comment une sélection commute avec les destructeurs. Encore une fois, l'idée est de faire apparaître le plus tard possible le destructeur pour qu'un maximum de la requête soit évalué dans le nuage sans affecter sa confidentialité.

$$\sigma_{p_\delta} \circ decrypt_{\alpha,\mathsf{c}} \equiv decrypt_{\alpha,\mathsf{c}} \circ \sigma_{p_\delta} \qquad \text{si } \alpha \notin \delta \tag{13}$$

$$\sigma_{p_\delta} \circ decrypt_{\alpha,\mathsf{c}} \equiv decrypt_{\alpha,\mathsf{c}} \circ \sigma_{\mathsf{c} \Rightarrow p_\delta} \qquad \text{si } \alpha \in \delta \tag{14}$$

$$\sigma_{g_{\delta \cap \delta'} \wedge d_{\delta \setminus \delta'} \wedge p_\delta} \circ defrag_{\delta'} \equiv \sigma_{p_\delta} \circ defrag_{\delta'}\ (\sigma_{g_{\delta \cap \delta'}}, \sigma_{d_{\delta \setminus \delta'}}) \tag{15}$$

FIGURE 21 – *Lois de sélection.*

Commutativité sur le chiffrement (*eq.* 13, 14). Il y a deux cas à considérer lors de l'application d'une sélection sur des n-uplets chiffrés. Le premier (*eq.* 13) stipule que la sélection s'opère normalement quand les valeurs des attributs utilisés par le prédicat ne sont pas chiffrées (si $\alpha \notin \delta$). Le second (*eq.* 14) dit que la sélection doit utiliser un prédicat qui supporte les opérations sur une donnée chiffrée ($\mathsf{c} \Rightarrow p_\delta$) quand les valeurs des attributs utilisés par le prédicat sont chiffrées (si $\alpha \in \delta$). Ceci suppose que le schéma de chiffrement (c) soit homomorphe sur les opérations du prédicat.

Il se peut que la sélection considère à la fois des attributs lisibles et des attributs chiffrés, comme dans $\sigma_{\mathsf{AES} \Rightarrow nom = \text{`}gTR7...Q\text{'} \wedge adress = \text{`}Bureau\text{'}}$. À ce moment, la requête peut utiliser les lois de cascade et de commutativité sur la sélection (*eq.* 4 et *eq.* 5, Ullman, 1982) pour séparer et réordonner le prédicat avant d'utiliser les lois 13 et 14.

Commutativité sur la fragmentation (*eq.* 15 -- 17). Permettre à la sélection de traverser la fragmentation produit potentiellement une importante optimisation. Le prédicat ainsi évalué sur chaque fragment réduit le nombre de n-uplets, ce qui accélère l'évaluation du reste de la requête. Cette réduction limite également la quantité d'information à transporter sur le réseau avant la défragmentation.

Pour que la sélection soit distribuée sur chaque fragment, il faut séparer son prédicat. Si le prédicat p opère sur les attributs contenus dans le schéma δ et que la relation est fragmentée sur le schéma δ', alors celui-ci peut être séparé avec trois fonctions dans une forme conjonctive. Tout d'abord, la fonction g

qui opère uniquement sur les attributs du fragment gauche, soit $g_{\delta\cap\delta'}$. Puis la fonction d qui opère uniquement sur les attributs du fragment droit, soit $d_{\delta\setminus\delta'}$. Et enfin, le prédicat p_δ pour les tests qui requièrent de connaître les valeurs des deux fragments de manière simultanée (*ex.*, un test sur le *hachage* (hash) du nom et d'une date). De ce fait, la sélection σ_{p_δ} est réécrite en $\sigma_{g_{\delta\cap\delta'}\wedge d_{\delta\setminus\delta'}\wedge p_\delta}$. Elle peut alors être distribuée à la manière des lois de commutativité entre la sélection et la jointure (*eq.* 7, Ullman, 1982). Ce comportement est spécifié par l'équation 15.

L'équation 15 peut être optimisée si les attributs considérés par le prédicat de la sélection sont localisés dans un même fragment. Par exemple, si les attributs du prédicat sont inclus dans le fragment de gauche (δ'). Alors, la fonction g filtre les n-uplets de gauche. La fonction d vaut forcément $vrai$ (*c.-à-d.*, le prédicat qui retourne toujours vrai), car elle n'a pas à filtrer des n-uplets du fragment de droite. De même, la fonction p vaut $vrai$, car son filtrage est déjà fait par la fonction g. L'équation 16 spécifie un tel comportement. L'équation 17 spécifie un comportement similaire pour le fragment de droite.

$$\sigma_{g_\delta\wedge vrai\wedge vrai}\circ defrag_{\delta'} \equiv defrag_{\delta'}\ (\sigma_{g_\delta}, id) \qquad \text{avec } \delta\subseteq\delta' \qquad (16)$$

$$\sigma_{vrai\wedge d_\delta\wedge vrai}\circ defrag_{\delta'} \equiv defrag_{\delta'}\ (id, \sigma_{d_\delta}) \qquad \text{avec } \delta\cap\delta'=\emptyset \qquad (17)$$

4.2.5 Lois d'agrégation par dénombrement

La fonction d'agrégation par dénombrement $count_\delta$ compte le nombre de lignes par groupes de δ attributs égaux. Le résultat est une nouvelle relation faite des δ attributs distincts et du nombre de fois où ils apparaissent. Les lois d'agrégation par dénombrement spécifient le comportement du $count$ par rapport aux destructeurs.

FIGURE 22 – *Lois d'agrégation par dénombrement.*

$$count_\delta\circ decrypt_{\alpha,\mathsf{c}} \equiv decrypt_{\alpha,\mathsf{c}}\circ count_{\mathsf{c}\Rightarrow\delta} \qquad \text{si } \alpha\in\delta \qquad (18)$$

$$count_\delta\circ decrypt_{\alpha,\mathsf{c}} \equiv count_\delta \qquad \text{si } \alpha\notin\delta \qquad (19)$$

$$count_\delta\circ defrag_{\delta'}\ (id, \pi_\emptyset) \equiv defrag_{\delta'}\ (count_\delta, \pi_\emptyset) \qquad \text{si } \delta\subseteq\delta' \qquad (20)$$

$$count_\delta\circ defrag_{\delta'}\ (\pi_\emptyset, id) \equiv defrag_{\delta'}\ (\pi_\emptyset, count_\delta) \qquad \text{si } \delta\cap\delta'=\emptyset \qquad (21)$$

Commutativité sur le chiffrement (*eq.* 18, 19). Ces équations spécifient le comportement du $count$ par rapport aux données chiffrées. Pour comprendre ces équations, il faut se rappeler que le $count_\delta$ est la composition de deux fonctions. La fonction de groupe, tout d'abord, qui constitue des groupes en testant l'égalité des n-uplets sur les attributs δ. Suivi de la fonction d'agrégation par dénombrement, qui compte le nombre de lignes dans chaque groupe.

La fonction de groupe a besoin de lire la valeur des attributs δ (ou, tout du moins, de tester leur égalité) pour former les groupes. Du coup, si le groupe porte sur un schéma contenant un attribut chiffré, invariablement le chiffrement employé supporte le test d'égalité. Ce comportement est spécifié par

l'équation 18. L'équation précise également qu'il faut déchiffrer cet attribut après dénombrement.

L'équation suivante (*eq.* 19) stipule que l'opération de chiffrement disparaît quand le $count$ groupe sur des valeurs d'attributs lisibles. L'explication est la suivante : après la formation des groupes, la fonction d'agrégation par dénombrement fusionne les n-uplets de chaque groupe. Pour chaque n-uplet fusionné, sa partie variable (*c.-à-d.*, $\Delta \setminus \delta$) est remplacée par le nombre de n-uplets dans le groupe. L'attribut chiffré se trouvant forcément dans cette partie et ayant été remplacé par un nombre, il n'a plus besoin d'être déchiffré.

Commutativité sur la fragmentation (*eq.* 20, 21). Ces équations sont plus contraignantes que les précédentes. Pour les comprendre, il faut détailler le comportement de la fonction d'agrégation par dénombrement.

Un $count$ doit avoir évalué toutes les sélections qui le précèdent pour être correct. Mais, il est impossible de vérifier cette assertion si le $count$ est dans un fragment et que les fragments sont évalués de manière concurrente.

En effet, une sélection peut avoir été distribuée dans plusieurs fragments avec les lois de sélection. Mais l'évaluation concurrente fait, qu'avant le $count$, seules les sélections propres à son fragment auront été évaluées. Par conséquent, l'équivalence observationnelle ne sera pas vérifiée.

En revanche, l'équivalence observationnelle peut être vérifiée si les fragments sont évalués séquentiellement, en finissant par le fragment qui contient le $count$. Par exemple, si le $count$ se trouve dans le fragment de gauche, le fragment de droite est évalué en premier. Puis, conformément à la stratégie séquentielle (*cf.* §2.4.3.3), le résultat est transmis à l'application SaaS. L'application joue son rôle d'intermédiaire et transmet ensuite les résultats au fragment de gauche pour qu'il rajoute un prédicat sur les identifiants. Ce prédicat limite les résultats retournés par le fragment de gauche aux n-uplets qui respectent les conditions du fragment de droite. Maintenant, l'évaluation du $count$ est correcte.

Cependant, qu'en est-il de l'équivalence de confidentialité ? Par exemple, est-ce que la requête suivante préserve la confidentialité des dates et adresses imposée par la fragmentation ?

$$defrag_{date}\left(count_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]}, \pi_{adresse}\right)$$

Dans cette requête, le fragment de droite retourne des adresses. En suivant le schéma décrit juste avant, il est clair que le fragment de gauche va être en possession de ces adresses. Par conséquent, le fragment de gauche violera la contrainte $\{date, adresse\}$. Du coup, une évaluation séquentielle n'est pas suffisant pour distribuer correctement le $count$.

La seule façon de garantir que le premier fragment évalué ne retourne pas d'informations confidentielles au second fragment est d'obliger celui-ci à supprimer toutes ses valeurs. C'est ce comportement qui est spécifié par les équations 20 et 21. La projection sur l'ensemble vide ($\pi_{\emptyset}$) supprime tous les attributs sauf les identifiants qui sont gérés implicitement.

4.2.6 Lois de composition des cryptographies

Les lois de composition des cryptographies commutent et distribuent les techniques de cryptographie.

FIGURE 23 – *Lois de composition des cryptographies.*

$$f \circ id \equiv id \circ f \equiv f \tag{22}$$

$$defrag_{\delta,\delta'}(id, id, id) \circ frag_{\delta,\delta'} \equiv$$

$$defrag_{\delta}\ (id,\ defrag_{\delta'}\ (id,id) \circ frag_{\delta'}) \circ frag_{\delta} \quad \text{si } \delta' \subseteq (\Delta \setminus \delta) \tag{23}$$

$$frag_{\delta} \circ decrypt_{\alpha,\mathtt{c}} \equiv (decrypt_{\alpha,\mathtt{c}}, id) \circ frag_{\delta} \quad \text{si } \alpha \in \delta \tag{24}$$

$$frag_{\delta} \circ decrypt_{\alpha,\mathtt{c}} \equiv (id, decrypt_{\alpha,\mathtt{c}}) \circ frag_{\delta} \quad \text{si } \alpha \notin \delta \tag{25}$$

$$decrypt_{\alpha,\mathtt{c}} \circ defrag_{\delta} \equiv defrag_{\delta}\ (decrypt_{\alpha,\mathtt{c}}, id) \quad \text{si } \alpha \in \delta \tag{26}$$

$$decrypt_{\alpha,\mathtt{c}} \circ defrag_{\delta} \equiv defrag_{\delta}\ (id, decrypt_{\alpha,\mathtt{c}}) \quad \text{si } \alpha \notin \delta \tag{27}$$

$$frag_{\delta} \circ crypt_{\alpha,\mathtt{c}} \equiv (crypt_{\alpha,\mathtt{c}}, id) \circ frag_{\delta} \quad \text{si } \alpha \in \delta \tag{28}$$

$$frag_{\delta} \circ crypt_{\alpha,\mathtt{c}} \equiv (id, crypt_{\alpha,\mathtt{c}}) \circ frag_{\delta} \quad \text{si } \alpha \notin \delta \tag{29}$$

L'équation 22 déclare que composer une fonction avec l'identité (id) résulte en cette fonction. L'équation rend possible l'introduction et la suppression de la fonction identité dans la requête.

L'équation 23 généralise la fragmentation à plus de deux fragments. La fragmentation à $n+1$ fragments (notée, $frag_{\delta_1,\delta_2,\cdots,\delta_n}$) est du sucre syntaxique pour la composition des *frag* binaires dans le deuxième élément de la paire. La fonction $frag_{\delta_1,\delta_2,\cdots,\delta_n}$ produit $n+1$ fragments dont le type du premier est δ_1, du deuxième $(\Delta \setminus \delta_1) \cap \delta_2$, et ainsi de suite jusqu'au n^e avec le type $(\Delta \setminus \delta_1 \setminus \cdots \setminus \delta_{n-1}) \cap \delta_n$. Le dernier fragment contient les attributs restants.

Les équations 24 et 25 permettent au déchiffrement de rentrer dans la fragmentation. Cette opération n'est valide que si l'attribut considéré par le déchiffrement est localisé dans un fragment. De même, les équations 26 et 27 donnent à la fonction de déchiffrement la possibilité de sortir de la fragmentation.

Enfin, les équations 28 et 29 permettent d'intervertir l'ordre d'application des techniques de chiffrement et de fragmentation. Le chiffrement rentre dans la fragmentation si l'attribut à chiffrer est localisé dans ce fragment.

4.3 Optimiser la requête `#rendezvous` par les lois

Le chapitre 3, qui implémente la requête `#rendezvous` avec composition des techniques de cryptographie, établit que la composition permet d'atteindre les trois critères d'une bonne application PbD. À savoir, confidentialités, utilisation du nuage et performance. En revanche, atteindre ces trois critères se fait en complexifiant, à coup sûr, l'écriture de la requête. Et qui dit plus complexe à écrire, dit plus de risques de bogues et plus de risques de produire une requête qui, *in fine*, ne protège pas correctement les données privées.

À l'inverse, écrire une requête où tous les calculs sont faits chez la cliente est simple à formuler. Cette requête représente un bon point de départ pour le développeur. Toute la difficulté et l'enjeu pour lui sont donc de transformer

cette requête simple, vers une requête équivalente, mais sûre, qui compose ingénieusement les techniques de cryptographie.

Cette section montre comment les lois du langage C2QL aident le développeur à transformer une requête naïve où tous les calculs sont faits chez la cliente, vers son équivalent qui suit l'approche du *nuage confidentiel* (*c.-à-d.*, confidentiel, qui utilise au plus le nuage et performante – *cf. fig.* 17).

Le point de départ est la requête `#rendezvous` sans spécificateurs. L'absence de spécificateurs suggère un environnement non sécurisé. Par conséquent, la relation des rendez-vous est conservée chez Alice et la requête `#rendezvous` est entièrement évaluée chez elle. L'avantage de cette représentation est que tous les rendez-vous sont lisibles. Ceci simplifie l'écriture de la requête, puisque le développeur n'a pas à se demander si les fonctions de l'algèbre relationnelle sont applicables sur des rendez-vous inintelligibles. En revanche, la requête ne profite pas des avantages du nuage.

$$count_{date} \circ \pi_{date} \\ \circ \sigma_{(date-aujourdhui)\in[0..7]\wedge adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}}$$

Le développeur décide d'externaliser la relation des rendez-vous. Il sait que les contraintes de confidentialités $\{date, adresse\}$ et $\{nom\}$ sont protégées en fragmentant sur les dates et en chiffrant les noms. En C2QL, ceci se fait au moyen de spécificateurs. Par exemple, l'équation 9 des lois d'identité introduit le spécificateur de fragmentation. Il est suivi de son destructeur pour que le reste de la requête soit correcte sémantiquement.

$id \equiv defrag_{\delta} \circ frag_{\delta}$

$$\equiv count_{date} \circ \pi_{date} \\ \circ \sigma_{(date-aujourdhui)\in[0..7]\wedge adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}} \\ \circ defrag_{date} \circ frag_{date}$$

De la même manière, l'équation 8 sert à introduire le chiffrement symétrique déterministe (AES) des noms.

$id \equiv decrypt_{\alpha,\mathsf{c}} \circ crypt_{\alpha,\mathsf{c}}$

$$\equiv count_{date} \circ \pi_{date} \\ \circ \sigma_{(date-aujourdhui)\in[0..7]\wedge adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}} \\ \circ defrag_{date} \circ frag_{date} \circ decrypt_{nom,\mathsf{AES}} \circ crypt_{nom,\mathsf{AES}}$$

Désormais, les rendez-vous d'Alice sont inintelligibles. La base de données de gauche héberge le fragment $(date, id)$ et celle de droite, le fragment $(\mathsf{AES}\ nom, adresse, id)$. Cependant, la position des destructeurs très tôt dans la requête oblige à rapatrier l'entièreté des rendez-vous chez Alice pour calculer le reste de la requête.

L'étape suivante est d'optimiser la requête en poussant les fonctions de l'algèbre relationnelle avant les destructeurs $defrag$ et $decrypt$. Plus de fonctions avant les destructeurs impliquent plus de calculs faits sur le nuage.

La première fonction à pousser est la sélection :

$$\sigma_{(date-aujourdhui)\in[0..7]\wedge adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}}$$

Seulement, cette sélection a un prédicat qui porte sur des attributs localisés, à la fois dans le fragment de gauche ($date - aujourdhui \in [0..7]$) et dans

le fragment de droite ($adresse = \text{`}Bureau\text{'} \wedge nom = \text{`}Bob\text{'}$). Or, ni l'état de l'art ni cette thèse ne définissent de lois qui traversent un $defrag$ quand le prédicat de la sélection mêle des attributs présents dans les deux fragments. Il faut d'abord dissocier la sélection. Une opération possible avec la loi de cascade proposée par Ullman (*eq.* 4).

$\sigma_{pa_1} \circ \sigma_{pa_2} \circ \cdots \circ \sigma_{pa_n} \equiv \sigma_{pa_1 \wedge pa_2 \wedge \cdots \wedge pa_n}$

$$\begin{aligned}
&\equiv count_{date} \circ \pi_{date} \\
&\quad \circ \sigma_{(date-aujourdhui)\in[0..7]} \\
&\quad \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}} \\
&\quad \circ defrag_{date} \circ frag_{date} \circ decrypt_{nom,\mathsf{AES}} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}$$

Ceci rend possible l'utilisation des lois de sélection (*eq.* 17) qui poussent la sélection dans le $defrag$. À présent, du point de vue du comportement, la sélection sur les adresses et les noms se fait sur le fragment de droite, plutôt que sur la relation des rendez-vous comme précédemment.

$\sigma_{vrai \wedge d_\delta \wedge vrai} \circ defrag_{\delta'} \equiv defrag_{\delta'}\ (id, \sigma_{d_\delta})$ *avec* $\delta \cap \delta' = \emptyset$

$$\begin{aligned}
&\equiv count_{date} \circ \pi_{date} \\
&\quad \circ \sigma_{(date-aujourdhui)\in[0..7]} \\
&\quad \circ defrag_{date}\ (id, \sigma_{adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}}) \\
&\quad \circ frag_{date} \circ decrypt_{nom,\mathsf{AES}} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}$$

$\sigma_{g_\delta \wedge vrai \wedge vrai} \circ defrag_{\delta'} \equiv defrag_{\delta'}\ (\sigma_{g_\delta}, id)$ *avec* $\delta \subseteq \delta'$

Même raisonnement avec l'équation 16 qui opère sur le fragment de gauche.

$$\begin{aligned}
&\equiv count_{date} \circ \pi_{date} \\
&\quad \circ defrag_{date}\ (\sigma_{(date-aujourdhui)\in[0..7]}, \\
&\qquad\qquad \sigma_{adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}}) \\
&\quad \circ frag_{date} \circ decrypt_{nom,\mathsf{AES}} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}$$

$\pi_\delta \circ defrag_{\delta'} \equiv defrag_{\delta'}\ (\pi_{\delta \cap \delta'}, \pi_{\delta \setminus \delta'})$

La requête utilise ensuite les lois de projection (*eq.* 12) pour faire rentrer la projection dans la fragmentation. Après réécriture, la projection porte sur les dates dans le fragment de gauche et sur le schéma vide dans le fragment de droite.

$$\begin{aligned}
&\equiv count_{date} \circ defrag_{date} \\
&\quad (\pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]}, \\
&\qquad \pi_\emptyset \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}}) \\
&\quad \circ frag_{date} \circ decrypt_{nom,\mathsf{AES}} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}$$

Le $\pi_\emptyset$ indique que la projection sur le fragment de droite ne conserve ni les noms ni les adresses. Ainsi, seuls les identifiants, qui sont gérés implicitement par la fonction de défragmentation, sont conservés.

À ce stade, même si une grande partie des fonctions de l'algèbre relationnelle ont traversé le $defrag$, la requête n'est toujours pas optimisée à cause du $decrypt$. Les lois de composition des protections donnent l'équation 25 qui permet au déchiffrement de pénétrer dans la fragmentation si les attri-

buts considérés par le chiffrement sont localisés dans le fragment de droite. La réécriture suivante utilise cette loi.

$frag_\delta \circ decrypt_{\alpha,\mathsf{c}} \equiv (id, decrypt_{\alpha,\mathsf{c}}) \circ frag_\delta$ *si* $\alpha \notin \delta$

$$
\begin{aligned}
&\equiv count_{date} \circ defrag_{date} \\
&\quad \circ (\pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]}, \\
&\qquad \pi_\emptyset \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge nom=\text{`}Bob\text{'}} \circ \ decrypt_{nom,\mathsf{AES}}) \\
&\quad \circ frag_{date} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}
$$

La requête est ensuite réécrite avec l'équation 14. Elle ajoute une contrainte (souligné en rouge) sur le prédicat des noms. Celle-ci spécifie que les valeurs de l'attribut nom sont chiffrées avec le chiffrement AES.

$\sigma_{p_\delta} \circ decrypt_{\alpha,\mathsf{c}} \equiv decrypt_{\alpha,\mathsf{c}} \circ \sigma_{\mathsf{c}\Rightarrow p_\delta}$ *si* $\alpha \in \delta$

$$
\begin{aligned}
&\equiv count_{date} \circ defrag_{date} \\
&\quad \circ (\pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]}, \\
&\qquad \pi_\emptyset \circ \ decrypt_{nom,\mathsf{AES}} \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge \underline{\mathsf{AES}\Rightarrow} nom=\text{`}Bob\text{'}}) \\
&\quad \circ frag_{date} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}
$$

En fait, la réécriture utilise les équations 4, puis 14, puis 13 et enfin 4 pour que le $decrypt$ *propage sa contrainte uniquement sur l'attribut* nom*, mais ça devenait long à écrire.*

La contrainte est une condition d'application de la loi. Elle impose que le chiffrement supporte le test d'égalité entre le nom et la valeur 'Bob'. Pour l'instant, il appartient au développeur de vérifier cette indication. Mais, le chapitre sur l'implémentation du langage (*chap.* 6) montre comment elle peut être assurée grâce à un système de *classe-type* (type class – Peyton Jones et collab., 1997).

Ensuite, la fonction de déchiffrement commute avec la projection par l'équation 10.

$\pi_\delta \circ decrypt_{\alpha,\mathsf{c}} \equiv decrypt_{\alpha,\mathsf{c}} \circ \pi_\delta$

$$
\begin{aligned}
&\equiv count_{date} \circ defrag_{date} \\
&\quad \circ (\pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]}, \\
&\qquad decrypt_{nom,\mathsf{AES}} \circ \pi_\emptyset \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge \mathsf{AES}\Rightarrow nom=\text{`}Bob\text{'}}) \\
&\quad \circ frag_{date} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}
$$

De même, le déchiffrement utilise l'équation 27 des lois de composition des fonctions de cryptographie pour sortir de la fragmentation.

$decrypt_{\alpha,\mathsf{c}} \circ defrag_{\delta'} \equiv defrag_{\delta'}\ (id, decrypt_{\alpha,\mathsf{c}})$ *si* $\alpha \notin \delta$

$$
\begin{aligned}
&\equiv count_{date} \circ \ decrypt_{nom,\mathsf{AES}} \circ defrag_{date} \\
&\quad \circ (\pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]}, \\
&\qquad \pi_\emptyset \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge \mathsf{AES}\Rightarrow nom=\text{`}Bob\text{'}}) \\
&\quad \circ frag_{date} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}
$$

Le $decrypt$ finit par permuter avec le $count$ par l'équation 19. Cette permutation est possible, car le déchiffrement porte sur l'attribut nom tandis que la fonction d'agrégation par dénombrement conserve l'attribut $date$. Et, puisque l'agrégation retourne un nombre à la place des noms, la fonction de déchiffrement n'est plus nécessaire.

$count_\delta \circ decrypt_{\alpha,\mathsf{c}} \equiv count_\delta$ *si* $\alpha \notin \delta$

$$
\begin{aligned}
&\equiv \ count_{date} \circ defrag_{date} \\
&\quad \circ (\pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]}, \\
&\qquad \pi_\emptyset \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge \mathsf{AES}\Rightarrow nom=\text{`}Bob\text{'}}) \\
&\quad \circ frag_{date} \circ crypt_{nom,\mathsf{AES}}
\end{aligned}
$$

Enfin, la dernière réécriture opère sur le $count_{date}$ et le $defrag_{date}$. D'après les lois sur l'agrégation par dénombrement, le $count$ traverse le $defrag$ quand les attributs du $count$ sont localisables dans un fragment et que l'autre fragment contient une projection sur les identifiants. Ici, le $count$ porte sur l'attribut $date$ qui est localisé dans le fragment de gauche. Le fragment de droite, quant à lui, contient une projection sur les identifiants. Voilà qui rend possible l'utilisation de l'équation 20 pour une dernière optimisation.

$count_\delta \circ defrag_{\delta'}\ (id, \pi_\emptyset) \equiv defrag_{\delta'}\ (count_\delta, \pi_\emptyset)$ *si* $\delta \subseteq \delta'$

$$\equiv defrag_{date}\ (count_{date} \circ \pi_{date} \circ \sigma_{(date - aujourdhui) \in [0..7]},\ \pi_\emptyset \circ \sigma_{adresse = \text{'}Bureau\text{'} \wedge \mathsf{AES} \Rightarrow nom = \text{'}Bob\text{'}}) \circ frag_{date} \circ crypt_{nom,\mathsf{AES}} \quad \blacksquare$$

Le comportement de cette requête ainsi réécrite est équivalant à la requête `#rendezvous` par composition des techniques de cryptographie implémentée dans le chapitre précédent (*cf.* §3.2.2, *fig.* 14).

Cet exemple montre que les lois algébriques du langage C2QL permettent de dériver systématiquement, à partir d'une requête sans spécificateurs, son équivalent sûr et optimisé par composition des techniques de cryptographie.

4.4 Un modèle de distribution d'une requête C2QL

Le π-calcul (Milner, 1999) est un langage pour modéliser et analyser un système concurrent fait de plusieurs agents qui interagissent entre eux. Il est très simple avec seulement deux types d'entités : les processus (ou agents) et les canaux de communications. Mais, ceci n'entrave pas son expressivité. Pour preuve, le λ-calcul s'encode en π-calcul (Milner, 1999), ce qui fait du π-calcul un système Turing-complet.

Les emplois du π-calcul sont divers. Par exemple, il a été utilisé dans l'analyse du standard BPEL (*Business Process Execution Language*) dont l'objectif était l'orchestration des services Web (Lucchi et Mazzara, 2007 ; Abouzaid et Mullins, 2008). En 2001, Abadi et Fournet étendent le π-calcul pour transmettre des termes sur les canaux et ainsi raisonner sur les protocoles de cryptographie. Cette extension, nommée *applied π calculus*, est retenue par Blanchet comme langage pour modéliser les protocoles de cryptographies dans son *vérificateur de modèle* (model checker) ProVerif (2002).

Cette modélisation sera toutefois réutilisée au chapitre 5 pour vérifier automatiquement qu'une requête C2QL préserve ses contraintes de confidentialités.

Cette section concerne l'aspect modélisation offert par le π-calcul. L'aspect analyse n'est pas envisagé ici. L'idée est d'encoder une requête C2QL en π-calcul pour décrire sa distribution sur le nuage. De ce fait, la suite présente la syntaxe du π-calcul et donne sa sémantique informellement (4.4.1). Puis, la transformation d'une requête C2QL en π-calcul (4.4.2) montre comment celle-ci est distribuée sur les acteurs de son environnement.

4.4.1 La syntaxe du π-calcul et sa sémantique informelle

La syntaxe du π-calcul est résumée par la figure 24. La syntaxe présentée ici est celle du π-calcul polyadique qui permet d'envoyer plusieurs noms sur un canal (Milner, 1999). Y figure aussi l'opérateur de test d'égalité sur les noms, un

P	::=	$a(b_1, \ldots, b_n).P$	Réception
	|	$\overline{a}\langle b_1, \ldots, b_n\rangle.P$	Émission
	|	$P_1 \mid P_2$	Composition parallèle
	|	$(\nu a)P$	Restriction
	|	$!P$	Réplication
	|	$[a = b]P$	Garde
	|	0	Processus terminé

FIGURE 24 – *La syntaxe du π-calcul.*

opérateur propice à la modélisation des systèmes qui transmettent des adresses Web (URLs).

Dans ce langage, la modélisation d'un système concurrent répartit l'activité sur plusieurs processus (P). La communication entre ces processus se fait par échange de messages sur les canaux de communications. En particulier, les processus émettent et reçoivent des messages de manière synchrone . C'est-à-dire que le processus émetteur attend que le destinataire ait bien reçu et interprété le message avant d'en envoyer un autre. Le message échangé sur un canal est lui même un canal. Ainsi, le processus d'émission $\overline{echo}\langle m\rangle.p_1$ envoi le canal m sur le canal $echo$, puis, une fois que le destinataire a reçu le message, continue en tant que p_1. À l'inverse, le processus destinataire $echo(k).p_2$ attend de recevoir un message sur le canal $echo$, puis substitue le nom k par la valeur du message dans p_2. Ainsi, les occurrences libres de k dans p_2 se retrouvent liées par la réception dans le processus destinataire. S'il n'existe aucun processus destinataire pour le canal $echo$, le message m est mis en attente jusqu'à ce qu'un destinataire soit prêt à écouter sur le canal $echo$. En revanche, s'il y a plusieurs processus destinataires, l'un d'entre eux est choisi arbitrairement.

La communication synchrone s'oppose à la communication asynchrone où l'émetteur envoie ses messages au destinataire sans attendre de réponses.

Composer ces deux processus parallèlement $\overline{echo}\langle m\rangle.p_1 \mid echo(k).p_2$ suppose une synchronisation sur le canal $echo$ qui entraînera la substitution de k par m dans p_2.

La création d'un nouveau canal se fait avec l'opérateur de restriction ν (prononcé *new*) suivi du nom du canal. Le nom se retrouve lié dans le processus qui suit. Ainsi, le ν dans $(\nu echo)(\overline{echo}\langle m\rangle.p_1 \mid echo(k).p_2)$, modifie la portée lexicale du nom $echo$ de globale, à locale au processus $(\overline{echo}\langle m\rangle.p_1 \mid echo(k).p_2)$. L'intention est qu'aucun autre processus n'interfère avec les communications sur $echo$.

Un processus peut être répliqué autant de fois que nécessaire grâce à l'opérateur (!). Cet opérateur permet, par exemple, de modéliser le service de débogage d'un serveur $!(echo(k).\overline{k}\langle\rangle.0)$ qui notifie, par le canal k, de la bonne réception d'une requête à chaque fois qu'un processus l'interroge. Ainsi, dans le programme suivant, les deux appels au service $echo$ engendrent deux notifications. Ces notifications sont produites de manières concurrentes sur les canaux m et n.

$(\nu echo)($	
$\quad(\nu m)(\overline{echo}\langle m\rangle.m().0) \mid$	Client 1
$\quad(\nu n)(\overline{echo}\langle n\rangle.n().0) \mid$	Client 2
$\quad!(echo(k).\overline{k}\langle\rangle.0))$	Serveur

Ce programme fait également apparaître le processus *nil* (noté, 0) qui représente un processus qui a terminé de s'exécuter.

Enfin, la garde, notée $[a = b]P$, poursuit avec le sous-processus P, si a et b ont un nom identique.

Dans la suite, le langage π-calcul est utilisé pour modéliser la distribution d'une requête C2QL sur le nuage.

4.4.2 Distribuer une requête C2QL avec le π-calcul

Dans la traduction d'une requête C2QL en π-calcul, les acteurs du nuage sont traduits par des processus. Les communications entre ces acteurs modélisent l'émission d'une requête et l'envoi d'une relation après exécution de cette requête.

Le π-calcul est suffisamment expressif pour ça. Mais, les types de données algébriques nécessaires ici, comme les listes (pour représenter les relations), s'encodent par des processus (Milner, 1999), un peu comme les encodages de Church[5] avec le λ-calcul. Ceci complexifie la lecture d'une requête en π-calcul, car il faut distinguer deux types de processus. Ceux qui représentent un acteur du nuage et ceux qui représentent une relation avec les fonctions de l'algèbre relationnelle.

Par conséquent, la suite augmente la syntaxe du π-calcul pour rendre explicites les relations et l'application des fonctions de l'algèbre relationnelle. Lors d'une traduction, ceci permet de distinguer un acteur du nuage modélisé par un processus, d'une relation modélisée par un nom.

La syntaxe d'un π-calcul pour les requêtes C2QL

La syntaxe du π-calcul pour modéliser la distribution d'une requête C2QL est donnée à la figure 25. La lettre Q fait référence à une fonction de C2QL (*fig.* 18), tout comme δ qui indique une liste d'attributs.

FIGURE 25 – *La syntaxe du π-calcul pour une requête C2QL.*

L	$::=$	a, b, c	Nom
	$\mid$	r, s, t	Relation
P	$::=$	$a(L_1, \ldots, L_n).P$	Réception
	$\mid$	$\overline{a}\langle L_1, \ldots, L_n\rangle.P$	Émission
	$\mid$	$P_1 \mid P_2$	Composition parallèle
	$\mid$	$(\nu a)P$	Restriction
	$\mid$	$!P$	Réplication
	$\mid$	$[a = b]P$	Garde
	$\mid$	$(\rho r : \delta)P$	Localisation
	$\mid$	$let\ s = (Q\ L)\ in\ P$	Application C2QL
	$\mid$	0	Processus terminé

5 en.wikipedia.org/wiki/Church_encoding

Les constructions du π-calcul sont préservées, mais la syntaxe ajoute trois spécificités. Premièrement, les processus d'émission et de réception peuvent transmettre et recevoir une relation (table) en plus des noms. Ceci pour modéliser la transmission du résultat du calcul d'une requête (*ex.*, de la base de données à Alice). Deuxièmement, un processus peut définir une nouvelle relation avec l'opérateur ρ. Ceci pour symboliser la présence d'une relation localisée dans le processus P. Le δ après le ρ indique le schéma relationnel (type) de la relation. Ici, le δ est juste une annotation qui fait office de documentation. Il n'a pas pour dessein de permettre une analyse statique. Enfin, l'opérateur let permet d'appliquer les fonctions (Q) de C2QL sur une relation pour calculer une nouvelle relation. Ici, le terme L représente une relation. Le terme L peut être soit un canal, soit une relation, mais le processus bloque son exécution si le L ne se réduit pas à une relation. Le ρ et le let modifient la portée lexicale du nom comme le ν dans le π-calcul polyadique.

L'augmentation de la syntaxe du π-calcul présentée ici s'inspire de l'augmentation proposée par Abadi et Fournet (2001) pour raisonner sur les protocoles de cryptographie.

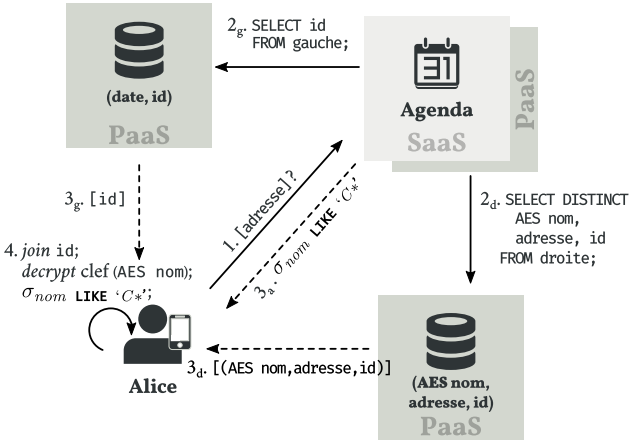
Traduction de la requête `[adresse]`

Grâce à ces trois ajouts, il est possible de traduire une requête C2QL pour rendre compte de sa distribution sur le nuage. Cette section montre la traduction de la requête `[adresse]`

Pour rappel, la distribution d'une requête C2QL est définie par ses spécificateurs et ses destructeurs (*cf.* §4.1.2). La présence de spécificateurs présuppose un environnement où la relation est sauvegardée dans une/des bases de données PaaS, puisque les n-uplets sont protégés par des techniques de cryptographie. La présence de destructeurs impose de rapatrier les n-uplets chez la cliente, puisque les n-uplets ne seront plus protégés.

Avec ces deux commandements, il est assez simple de trouver le programme π-calcul qui modélise la requête "*nuage confidentiel*" `[adresse]` dont la représentation graphique est remise ici pour le lecteur.

$$\sigma_{nom\ \texttt{LIKE}\ \text{`}C*\text{'}} \circ decrypt_{nom,\texttt{AES}} \circ defrag_{date}\ (\pi_{\emptyset}, \pi_{nom,adresse}) \circ frag_{date} \circ crypt_{nom,\texttt{AES}}\ rendezvous$$



La figure montre que l'application agenda SaaS tient un rôle d'intermédiaire entre les acteurs du nuage. Elle est responsable de recevoir la requête d'Alice (étape 1) et d'orchestrer l'appel des bases de données (étapes $2_g/2_d$). L'appel peut se faire avec une stratégie concurrente (comme ici) ou séquentielle (comme dans la requête `#rendezvous`). En π-calcul, l'agenda est traduit par le processus $Agenda$. À la réception de la requête `[adresse]`, l'agenda appelle de manière concurrente les fragments de gauche et droite (symboli-

sés par les canaux db_0 et db_1) pour qu'ils effectuent la requête `[adresse]`. Le deuxième argument dans l'appel spécifie l'envoi de la réponse à Alice.

$$Agenda \equiv app(url).[url = \texttt{"[adresse]"}] \\ (\overline{db_0}\langle url, client\rangle.0 \mid \overline{db_1}\langle url, client\rangle.0)$$

Les fragments de gauche et droite sont traduits par deux processus $Frag_g$ et $Frag_d$. Ces deux processus utilisent l'opérateur ρ pour indiquer la présence d'une relation localisée chez eux. S'ils reçoivent la demande d'effectuer la requête `[adresse]` sur leur canal de réception, alors ils calculent le résultat sur leur relation. Puis, ils transmettent le résultat sur le canal k fourni par l'agenda. Dans la figure, cette modélisation correspond aux réceptions des requêtes, étapes $2_g/2_d$ et à l'émission du résultat, étapes $3_g/3_d$.

$$Frag_g \equiv (\rho rdv_g : (date, id))db_0(url, k).[url = \texttt{"[adresse]"}] \\ let\ s = \pi_{\emptyset}\ rdv_g\ in\ \overline{k}\langle s\rangle.0$$

$$Frag_d \equiv (\rho rdv_d : (\texttt{AES}\ nom, adresse, id))db_1(url, k).[url = \texttt{"[adresse]"}] \\ let\ s = \pi_{nom,adresse}\ rdv_d\ in\ \overline{k}\langle s\rangle.0$$

Le comportement d'Alice est traduit par le processus $Alice$. Alice demande à l'agenda de faire la requête `[adresse]`, puis attend les réponses des deux fragments. Ensuite, Alice applique les destructeurs de cryptographie et filtre sur l'expression régulière "C*". Dans la figure, ceci correspond à l'émission de la requête à l'étape 1, puis à la réception des résultats aux étapes $3_g/3_d$ et enfin, aux calculs faits côté client à l'étape 4.

$$Alice \equiv \overline{app}\langle\texttt{"[adresse]"}\rangle.client(r_1).client(r_2). \\ let\ s = defrag_{date}\ r_1\ r_2\ in \\ let\ t = decrypt_{nom,\texttt{AES}}\ s\ in \\ let\ u = \sigma_{nom\ \texttt{LIKE}\ `C*'}\ t\ in\ 0$$

L'étape 3_a qui consiste à recevoir la fonction de filtrage depuis l'agenda n'a pas besoin d'être représentée en π-calcul puisque la requête C2QL donne toutes les informations pour directement représenter la fonction chez Alice. Dans une application concrète, comme celle de chapitre 3, le code de la cliente est traduit en JavaScript et est hébergé par l'application SaaS. Ainsi, la cliente récupère automatiquement son code lorsqu'il se connecte à l'application avec son navigateur Web. C'est pourquoi il est inutile de représenter l'étape 3_a.

Enfin, la requête `[adresse]` consiste en la mise en parallèle de tous ces processus. Les processus $Agenda$, $Frag_g$ et $Frag_d$ sont répliqués (!) pour symboliser leur rôle de serveur.

$$\texttt{[adresse]} \equiv (\nu app)(\nu db_0)(\nu db_1)(\nu client) \\ !Agenda\ |!Frag_g\ |!Frag_d \mid Alice$$

4.4.3 Traduction automatique d'une requête C2QL en π-calcul

La traduction automatique d'une requête C2QL vers un terme π-calcul est donnée dans la suite. Cette traduction vaut lorsque les fragments sont évalués

avec une stratégie concurrente (*cf.* §2.4.3.3). Mais l'approche est la même pour une évaluation séquentielle. Également important, cette traduction vaut si le terme C2QL représente un programme sémantiquement correcte.

La forme canonique d'une requête C2QL

Avant de traduire automatiquement une requête C2QL en un terme π-calcul, cette requête doit être en forme canonique. Une requête C2QL est en forme canonique quand tous $decrypt$ de chiffrement se retrouvent après les $defrag$. N'importe quelle requête C2QL qui contient une combinaison de $decrypt$ et de $defrag$ peut se ramener à cette forme en appliquant l'équation suivante sur le fragment de gauche. Le même raisonnement s'applique sur le fragment de droite.

$$\begin{aligned} & q_s \circ \cdots \circ q_q \circ defrag_\delta(q_p \circ \cdots \circ q_j \circ decrypt_{a,\mathsf{c}} \circ q_i \circ \cdots \circ q_a, id) \\ & \equiv q_s \circ \cdots \circ q_q \circ q_p \circ \cdots \circ q_j \circ decrypt_{a,\mathsf{c}} \circ defrag_\delta(q_i \circ \cdots \circ q_a, id) \end{aligned}$$

Cette forme est censément plus simple pour la traduction. Grâce à elle, le rapatriement d'une relation d'un fragment vers la cliente est géré uniquement par la lecture d'un $defrag$. Alors que dans sa forme non-canonique, la traduction devrait en plus gérer le fait qu'un $decrypt$ dans un fragment entraîne, lui aussi, le rapatriement de la relation du fragment vers la cliente.

Cette forme canonique ne change pas la sémantique de la requête puisque cette équation s'obtient facilement à partir des lois vues précédemment. Intuitivement, dans la forme non-canonique, la présence des $q_p \circ \cdots \circ q_j$ à l'intérieur du fragment proviennent obligatoirement d'optimisations faites avec des lois sur le $defrag$ (*cf.* section 4.3 sur la méthodologie d'optimisation). Or, s'il y a eu une optimisation, il peut aussi y avoir une désoptimisation. C'est ce qui est fait dans la forme canonique en sortant les $q_p \circ \cdots \circ q_j$ du fragment de gauche. Ensuite, le $decrypt$ peut facilement sortir du $defrag$ grâce aux lois 26 et 27 de composition des cryptographie.

Cette forme ne change pas l'optimisation pour autant. Dans sa forme non-canonique, les fonctions $q_s \circ \cdots \circ q_q \circ q_p \cdots \circ q_j$ sont effectuées chez Alice à cause du $decrypt$. Dans sa forme canonique, les fonctions $q_s \circ \cdots \circ q_q \circ q_p \cdots \circ q_j$ sont effectuées chez Alice à cause du $defrag$.

Une traduction automatique

Dans la suite la traduction est découpée en trois : la traduction $\mathcal{A}$ (*fig.* 26) produit le terme π-calcul pour l'application SaaS, la traduction $\mathcal{C}$ (*fig.* 27) produit le terme pour la cliente et la traduction $\mathcal{D}$ (*fig.* 28) produit le terme pour les bases de données PaaS. À chaque fois, la traduction est écrite par *reconnaissance de motif* (pattern matching) sur les symboles de la grammaire de C2QL et par récursion sur le terme de droite à gauche. Ceci garantit l'automaticité d'une traduction. Le programme π-calcul final est obtenu en appelant toutes les traductions sur une même requête C2QL (*fig.* 29).

Traduction de l'application SaaS (*fig.* 26). Pour rappel, l'application joue un rôle d'intermédiaire entre la cliente et la/les bases de données. La traduction parcourt donc tous les termes de droite à gauche jusqu'à rencontrer

le *premier* destructeur. Ce destructeur indique qu'à un moment donné il y a eu une requête sur une base de données PaaS. Il faut donc générer le terme responsable de l'appel à cette base de données. C'est ce que fait la transformation $\mathcal{A}$. L'indice i dans le contexte sert à construire les identifiants des canaux des bases de données.

FIGURE 26 – *Traduction d'une requête C2QL vers le term π-calcul pour l'application SaaS*

$$\begin{aligned}
\mathcal{A}[\![decrypt]\!]_i &= \overline{db_0}\langle url, client\rangle.0 \\
\mathcal{A}[\![defrag(_, defrag(Q_c, Q'_c) \circ frag)]\!]_i &= \overline{db_i}\langle url, client\rangle.0 \mid \mathcal{A}[\![defrag(Q_c, Q'_c)]\!]_{i+1} \\
\mathcal{A}[\![defrag(_,_)]\!]_i &= \overline{db_i}\langle url, client\rangle.0 \mid \overline{db_{i+1}}\langle url, client\rangle.0 \\
\mathcal{A}[\![Q_{c2} \circ Q_{c1}]\!]_i &= \mathcal{A}[\![Q_{c1}]\!]_i \vee \mathcal{A}[\![Q_{c2}]\!]_i
\end{aligned}$$

Lorsque la requête est en forme canonique, il n'est pas nécessaire d'aller plus loin que le premier destructeur, car le reste de la requête est exécutée chez la cliente. C'est ce que fait le $\vee$ dans la récursion. La sémantique du $\vee$ ici est : "arrête la traduction dès qu'un terme est produit". En outre, la traduction ne fait pas apparaître tous les symboles de la grammaire de C2QL. La raison est que les symboles absents ne contribuent pas à l'élaboration du terme π-calcul de l'application.

Pour avoir le terme π-calcul complet de l'application SaaS, il faut en plus ajouter la réception de la requête de la cliente. Ceci est fait par la fonction App qui prend en paramètre la requête C2QL à traduire (Q_c) et le nom de cette requête (h).

$$App_{Q_c,\mathsf{h}} = app(url).[url = \mathsf{h}]\mathcal{A}[\![Q_c]\!]_0$$

Traduction du code client (*fig.* 27). La traduction du code client ($\mathcal{C}$) passe par un accumulateur (Π) pour construire le terme π-calcul. À chaque étape de la traduction, $\mathcal{C}$ connaît le nom de la relation courante (r), l'endroit courant où s'effectuent les calculs (p) et un indice qui compte le nombre de fragments rencontrés (i).

L'accumulateur Π contient le terme de la cliente en cours de construction. Sa valeur initiale est vide (identifié par la valeur $\perp$). La notation $\Pi[t]$ concatène le terme t à la valeur courante de l'accumulateur. Par exemple, lorsque la valeur courante de Π est $client(k).$, la notation $\Pi[0]$ modifie cette valeur courante en $client(k).0$.

L'argument p indique dans quel processus faire la requête relationnelle courante (Q – *eq.* 30). Il a l'une des valeurs suivantes : $client$, $agenda$, db_i ou $\perp$, avec $\perp$ la valeur initiale de p. Dans la traduction, la lecture d'un spécificateur/destructeur modifie la valeur de p pour indiquer où se font les calculs dorénavant (*eq.* 31 à 37).

Les ϕ, ϕ' et ϕ'' représentent un nom de relation ou de canal frais. Un nom frais est un nom qui n'est pas lié par la portée lexicale du terme. Par exemple, l'équation 33 qui ajoute le déchiffrement de la relation r lie le résultat à la variable ϕ. Par conséquent, le nom ϕ ne doit pas exister dans le terme courant au risque de modifier la sémantique du reste du terme.

À chaque étape, la traduction $\mathcal{C}$ produit un 4-uplets (π, r, p, i) avec π la valeur de l'accumulateur et r, p et i les informations du contexte de $\mathcal{C}$.

$$\mathcal{C}[\![Q]\!]^{\Pi}_{r,p,i} = (\Pi[let\ \phi = Q\ r\ in\ ], \phi, p, i) \qquad \text{if } p \in \{client, \bot\} \tag{30}$$

$$\mathcal{C}[\![crypt]\!]^{\Pi}_{r,\_,i} = (\Pi, r, db_i, i) \tag{31}$$

$$\mathcal{C}[\![frag]\!]^{\Pi}_{r,\_,i} = (\Pi, r, db_i, i) \tag{32}$$

$$\mathcal{C}[\![decrypt_{\alpha,\mathsf{c}}]\!]^{\Pi}_{r,client,i} = (\Pi[let\ \phi = decrypt_{\alpha,\mathsf{c}}\ r\ in\ ], \phi, client, i) \tag{33}$$

$$\mathcal{C}[\![decrypt_{\alpha,\mathsf{c}}]\!]^{\Pi}_{\_,\_,i} = \mathcal{C}[\![decrypt_{\alpha,\mathsf{c}}]\!]^{\Pi[client(\phi).]}_{\phi,client,i} \tag{34}$$

$$\mathcal{C}[\![defrag(\_, defrag(Q_c, Q'_c) \circ frag)]\!]^{\Pi}_{r,\_,0} = \mathcal{C}[\![defrag(Q_c, Q'_c)]\!]^{\Pi[client(\phi).]}_{\phi,client,1} \tag{35}$$

$$\mathcal{C}[\![defrag(\_, defrag(Q_c, Q'_c) \circ frag)]\!]^{\Pi}_{r,\_,i} = \mathcal{C}[\![defrag(Q_c, Q'_c)]\!]^{\Pi[client(\phi).let\ \phi' = defrag(r,\phi)\ in\ ]}_{\phi',client,i+1} \tag{36}$$

$$\mathcal{C}[\![defrag(\_,\_)]\!]^{\Pi}_{\_,\_,i} = (\Pi[client(\phi).client(\phi').let\ \phi'' = defrag(\phi,\phi')\ in\ ], \phi'', client, i+2) \tag{37}$$

$$\mathcal{C}[\![Q_{c2} \circ Q_{c1}]\!]^{\Pi}_{r,p,i} = \texttt{let}\ (\Pi', r', p', i') = \mathcal{C}[\![Q_{c1}]\!]^{\Pi}_{r,p,i}\ \texttt{in}\ \mathcal{C}[\![Q_{c2}]\!]^{\Pi'}_{r',p',i'} \tag{38}$$

FIGURE 27 – *Traduction d'une requête C2QL vers le term π-calcul pour la cliente*

L'équation 38 montre que l'expression C2QL est traduite de droite à gauche. L'accumulateur et le contexte produis par la traduction du symbole le plus à droite sont transmis en argument au reste de l'expression.

La traduction $\mathcal{C}$ produit le terme qui modélise le processus client. Un terme récupère les données depuis les bases de données PaaS et applique les destructeurs s'ils existent. Puis, il applique les fonctions de l'algèbre relationnelle qui sont censées être exécutées chez la cliente. En revanche, la traduction ne produit pas le terme qui consiste à contacter l'application SaaS. Or, la cliente doit contacter l'application pour lui notifier son souhait de faire une requête. Ceci est fait par la fonction $Client$ qui prend en paramètre la requête C2QL à traduire (Q_c), le type de la relation sur laquelle cette requête s'évalue (δ) et le nom de cette requête (h).

$$\begin{aligned} Client_{Q_c,\delta,\mathsf{h}} = &\texttt{let}\ (\pi, \_, p, \_) = \mathcal{C}[\![Q_c]\!]^{\bot}_{r,\bot,0}\ \texttt{in} \\ &\texttt{if}\ p = \bot\ \texttt{then}\ (\rho r : \delta)\pi \\ &\texttt{else}\ \overline{app}\langle \mathsf{h} \rangle.\pi \end{aligned}$$

Il faut distinguer deux cas en regardant la valeur de p à la fin de la traduction. Soit p vaut $\bot$, auquel cas la traduction n'a pas rencontré de spécificateurs. Par conséquent, les données sont hébergées chez la cliente (*cf.* §4.1.2) et il faut créer la relation ρ avec le type δ. Soit p vaut autre chose, auquel cas la traduction a rencontré un spécificateur. Par conséquent, les données sont hébergées dans le nuage et il faut contacter l'application SaaS pour qu'elle joue son travail d'intermédiaire.

Traduction des bases de données PaaS (*fig.* 28). La traduction $\mathcal{D}$ produit une liste où chaque élément est un processus π-calcul qui symbolise

une base de données PaaS. Les paramètres de $\mathcal{D}$ sont identiques à ceux de la traduction $\mathcal{C}$ sauf Π et r qui sont respectivement une liste d'accumulateurs et une liste de relations courantes. La traduction ajoute aussi une liste de δ. Ces trois éléments sont des listes pour répondre au besoin de la traduction qui est de construire une liste de processus.

Le Π, le r et le δ sont indexés par un numéro de fragment. Ainsi, Π_i donne accès à l'accumulateur du processus db_i. De même, r_i donne accès au nom de la relation courante dans le processus db_i. Enfin, δ_i donne accès au type de la relation hébergée par le processus db_i.

Les notations de ces paramètres changent en conséquence. Dans la suite, la notation $\Pi_i[t]$ met à jour l'accumulateur de db_i à la manière de $\Pi[t]$ dans la traduction $\mathcal{C}$. Les notations $r_i[s]$ et $\delta_i[\delta']$ indiquent que le nom de la relation courante et le type de la relation hébergée par db_i valent s et δ', respectivement.

La traduction d'une requêtes C2QL en bases de données PaaS a quelques spécificités. Premièrement, la lecture d'un spécificateur modifie le type de la relation (*eq.* 40 et 41). Ceci pour calculer la forme des attributs dans chaque base de données. Ensuite, la traduction d'un destructeur provoque l'envoi de la relation courante sur le canal k (*eq.* 42 à 44). Le canal k étant le canal de réception qui est transmis par l'application SaaS. L'objectif est que la base de données n'applique pas le destructeur, conformément aux règles de distributions (*cf.* §4.1.2). Pour finir, la traduction du *defrag* provoque la traduction des requêtes qui sont à l'intérieur des fragments (*eq.* 43 et 44).

FIGURE 28 – *Traduction d'une requête C2QL vers le term π-calcul pour les bases de données PaaS*

$$\mathcal{D}[\![Q]\!]^{\Pi}_{r,db_j,i,\delta} = (\Pi_j[let\ \phi = Q\ r_j\ in\ ], r_j[\phi], db_j, i, \delta) \tag{39}$$

$$\mathcal{D}[\![crypt_{\alpha,\text{c}}]\!]^{\Pi}_{r,\_,i,\delta} = (\Pi, r, db_i, i, \delta_i[crypt_{\alpha,\text{c}}]) \tag{40}$$

$$\mathcal{D}[\![frag_\delta]\!]^{\Pi}_{r,\_,i,\delta'} = (\Pi, r, db_i, i, \delta'_i[frag_\delta]) \tag{41}$$

$$\mathcal{D}[\![decrypt]\!]^{\Pi}_{r,db_j,i,\delta} = (\Pi_j[\overline{k}\langle r_j\rangle.], r, client, i, \delta) \tag{42}$$

$$\begin{aligned}&\mathcal{D}[\![defrag(Q_{c3}, defrag(Q_{c2}, Q_{c1}) \circ frag)]\!]^{\Pi}_{r,p,i,\delta} = \\ &\quad \texttt{let}\ (\_,\_,\_,\_,\delta') = \mathcal{D}[\![frag]\!]^{\Pi}_{r,db_{i+1},i+1,\delta}\ \texttt{in} \\ &\quad \texttt{let}\ (\Pi', r', \_, \_, \delta'') = \mathcal{D}[\![Q_{c3}]\!]^{\Pi}_{r,db_i,i,\delta'}\ \texttt{in} \\ &\quad \mathcal{D}[\![defrag(Q_{c2}, Q_{c1})]\!]^{\Pi'_i[\overline{k}\langle r_i\rangle.]}_{r,p,i+1,\delta''}\end{aligned} \tag{43}$$

$$\begin{aligned}&\mathcal{D}[\![defrag(Q_{c2}, Q_{c1})]\!]^{\Pi}_{r,p,i,\delta} = \\ &\quad \texttt{let}\ (\Pi', r', \_, \_, \_) = \mathcal{D}[\![Q_{c2}]\!]^{\Pi}_{r,db_i,i,\delta}\ \texttt{in} \\ &\quad \texttt{let}\ (\Pi'', r'', \_, \_, \_) = \mathcal{D}[\![Q_{c1}]\!]^{\Pi'_i[\overline{k}\langle r'_i\rangle.]}_{r,db_{i+1},i+1,\delta}\ \texttt{in} \\ &\quad (\Pi''_{i+1}[\overline{k}\langle r''_{i+1}\rangle.], r, client, i+1, \delta)\end{aligned} \tag{44}$$

$$\mathcal{D}[\![Q_{c2} \circ Q_{c1}]\!]^{\Pi}_{r,p,i,\delta} = \texttt{let}\ (\Pi', r', p', i', \delta') = \mathcal{D}[\![Q_{c1}]\!]^{\Pi}_{r,p,i,\delta}\ \texttt{in}\ \mathcal{D}[\![Q_{c2}]\!]^{\Pi'}_{r',p',i',\delta'} \tag{45}$$

La traduction $\mathcal{D}$ produit une liste de termes pour modéliser les processus des bases de données PaaS. Un terme applique les fonctions de l'algèbre rela-

tionnelle sur sa relation et transmet le résultat sur son canal k. Il reste à modéliser la création de la relation et la réception du canal k envoyé par l'application SaaS. Ceci est fait par la fonction DB qui prend en paramètre le terme C2QL à traduire (Q_c) et le nom de la requête pour laquelle cette traduction est faite (h). Cette fonction retourne une liste de termes π-calcul qui modélisent les processus des bases de données PaaS.

$$
\begin{aligned}
DB_{Q_c,\mathsf{h}} = & \texttt{let } (\pi, \_, \_, \_, \delta) = \mathcal{D}[\![Q_c]\!]^{[\bot]}_{[r],\bot,0,\delta} \texttt{ in} \\
& \quad \texttt{let } \#db = \texttt{length } \pi \texttt{ in} \\
& \quad [(\rho r : \delta_0)\, db_0(url, k).[url = \mathsf{h}]\pi_0, \\
& \quad \cdots, (\rho r : \delta_{\#db})\, db_{\#db}(url, k).[url = \mathsf{h}]\pi_{\#db}]
\end{aligned}
$$

Programme π-calcul final (*fig.* 29). Le programme π-calcul final consiste en la mise en parallèle de tous les processus.

$$
\begin{aligned}
PiCalcul_{Q_c,\delta,\mathsf{h}} = & \texttt{let } \pi_{client} = Client_{Q_c,\delta,\mathsf{h}} \texttt{ in} \\
& \texttt{let } \pi_{app} = App_{Q_c,\mathsf{h}} \texttt{ in} \\
& \texttt{let } \pi_{db} = DB_{Q_c,\delta,\mathsf{h}} \texttt{ in} \\
& \texttt{let } \#db = \texttt{length } \pi_{db} \texttt{ in} \\
& (\nu\, app)(\nu\, db_0) \cdots (\nu\, db_{\#db})(\nu\, client) \\
& !\pi_{appP} \mid !\pi_{db_0} \mid \cdots \mid !\pi_{db_{\#db}} \mid \pi_{client}
\end{aligned}
$$

FIGURE 29 – *Traduction d'une requête C2QL en π-calcul.*

4.5 Conclusion

Ce chapitre propose le langage abstrait C2QL, un langage pour représenter les requêtes exécutées dans un environnement nuagique protégé par plusieurs techniques de cryptographie. Le langage se base sur l'algèbre relationnelle. Il ajoute des spécificateurs qui spécifient comment l'environnement est protégé et des destructeurs qui indiquent quand détruire une protection.

Le langage est équipé de lois algébriques. Une loi décrit sous quelles conditions une fonction de l'algèbre relationnelle permute avec un destructeur. Cette permutation préserve deux propriétés. La première, nommée *équivalence observationnelle*, assure que le résultat retourné par la requête est équivalent. La seconde, nommée *équivalence de confidentialité*, assure que les garanties apportées par un spécificateur sont préservées. L'intérêt des lois est double.

Premièrement, elles permettent de *dériver systématiquement* une requête suit l'approche du *nuage confidentiel* (*c.-à-d.*, confidentiel, qui utilise au plus le nuage et performante – *cf. fig.* 17), depuis une requête naïve qui est uniquement exécutée chez la cliente. L'avantage pour le développeur est que la requête naïve est très facile à écrire, car elles est sans protection. Alors que la requête "*nuage confidentiel*" est difficile à écrire, car elle utilise la composition de techniques de cryptographie.

Deuxièmement, le langage C2QL et ses lois forment un *cadre formel* pour raisonner sur la composition des techniques de cryptographie dans les applications du nuage. Par conséquent, le langage peut facilement être étendu

avec d'autres techniques de cryptographie. Étendre le langage consiste à ajouter les spécificateurs/destructeurs de la nouvelle technique. Puis, ajouter de nouvelles lois qui expliquent comment ces spécificateurs/destructeurs commutent avec les fonctions déjà présentes dans C2QL tout en préservant l'équivalence.

Dans le même temps, une traduction montre comment *transcrire automatiquement* une requête C2QL vers un programme π-calcul. Le langage n'a pas besoin de faire apparaître les acteurs du nuage et leurs interactions pour modéliser une application nuagique. L'avantage premier est de fournir au développeur un langage qui est orienté sur la manipulation des données. Cette orientation aide le développeur à raisonner sur l'optimisation de sa requête puisqu'il lui suffit de regarder la position des destructeurs pour prédire la quantité d'information calculée dans le nuage.

Le choix du π-calcul présente un autre avantage. Le π-calcul est un langage simple, mais suffisamment expressif pour modéliser les applications du nuage. Par exemple, le programme π-calcul de la requête `[adresse]` à la section 4.4.2 modélise tous les composants, interactions et calculs de l'implémentation concrète faite dans le chapitre 3 (*cf.* §3.2.2). Et même si l'espace pour aller du programme π-calcul au programme concret est grand, celui-ci n'est pas infranchissable. Par conséquent, le langage C2QL est un langage abstrait avec une *vraie vision opérationnelle*.

Une limite aux lois

Vérifier les contraintes de confidentialités. Les lois ne contiennent pas les contraintes de confidentialités. Par conséquent, un développeur peut choisir une configuration qui ne protège pas correctement les requêtes de son application. Par exemple, la requête `#rendezvous` qui serait protégée avec seulement le chiffrement des noms ne protégerait pas la contrainte $\{date, adresse\}$. Le chapitre suivant (*chap.* 5) encode les requêtes C2QL en ProVerif, un *vérificateur de modèle* (model checker) pour l'analyse automatique des propriétés de sécurité sur les protocoles de cryptographie, et vérifie automatiquement qu'une requête préserve ses contraintes de confidentialités.

Appliquer la méthodologie. La deuxième limite réside dans la méthodologie qui dit de partir d'une requête naïve pour obtenir une requête "*nuage confidentiel*". Pour l'instant, rien ne force le développeur à suivre cette méthodologie. Ainsi, le développeur peut écrire la version suivante de `[adresse]` qui n'a pas de sens, car la projection sur les noms n'est pas faite sur le bon fragment.

$$\begin{aligned}\sigma_{nom\ \texttt{LIKE}\ `C*'} \circ\ & decrypt_{nom,\texttt{AES}} \\ & \circ\ defrag_{date}\ (\pi_{nom}, \pi_{adresse}) \\ & \circ\ frag_{date} \circ crypt_{nom,\texttt{AES}}\end{aligned}$$

Il en est de même pour cette autre version qui applique la sélection sur des noms chiffrés, rendant les caractères inintelligibles, alors que l'expression régulière `LIKE` '$C*$' requiert de comparer les caractères.

$$decrypt_{nom,\texttt{AES}} \circ defrag_{date}\ (id,\ \sigma_{AES \Rightarrow nom\ \texttt{LIKE}\ `C*'} \circ \pi_{nom,adresse}) \circ frag_{date} \circ crypt_{nom,\texttt{AES}}$$

Ces erreurs de calcul peuvent, au mieux, stopper l'exécution du programme. Mais, elles peuvent aussi provoquer un bogue qui révèle ou cède l'accès aux données personnelles. Il faut donc détecter le plus tôt possible ces erreurs pour les corriger rapidement.

Le chapitre 6 s'attelle à ce problème et implémente le langage C2QL, à la manière d'un *langage dédié embarqué* (Embedded Domain-Specic Language – EDSL) dans Idris. Cette implémentation profite du système de types dépendants d'Idris pour garantir au développeur que la requête qu'il écrit a du sens du point de vue de l'exécution.

5 *Vérifier la confidentialité pour le langage C2QL*

Le langage C2QL donne au développeur la possibilité de définir une composition de techniques de cryptographie pour protéger une requête exécutée sur le nuage. Le développeur s'aide ensuite des lois pour optimiser la requête protégée pour qu'un maximum de calcul soit fait sur le nuage. Cependant, les lois ne garantissent pas que le choix des techniques de cryptographie protègent correctement les contraintes de confidentialités. En effet, le développeur de l'agenda peut, par exemple, faire le choix de protéger la requête `[adresse]` en fragmentant sur les noms.

$$\sigma_{nom\ \texttt{LIKE}\ `C*'} \circ \pi_{nom,adresse} \circ defrag_{nom} \circ frag_{nom}$$

Il utilise ensuite les lois pour optimiser cette requête. Grâce à la propriété d'équivalence de confidentialités, les fonctions qui se retrouvent à la droite du $defrag_{nom}$ sont maintenant exécutées sur des rendez-vous où le nom est inintelligible. Cependant, le $defrag_{nom}$ n'est pas un bon choix de technique pour protéger les contraintes $\{nom\}$ et $\{date, adresse\}$, car le noms restent lisibles et les dates et adresse sont localisées dans la même base de données. Malheureusement, les lois n'avertissent pas de ce mauvais choix.

Ce chapitre utilise l'outil ProVerif (Blanchet et collab., 2014) pour vérifier systématiquement si une requête C2QL préserve les contraintes de confidentialités. L'intérêt est de donner au développeur la garantie que son choix de techniques de cryptographie est le bon pour protéger l'application.

La section 5.1 présente l'outil ProVerif et montre comment vérifier la confidentialité d'un secret. La section 5.2 discute le choix de l'outil ProVerif et décrit l'approche pour vérifier qu'une requête C2QL préserve ses contraintes de confidentialités. En particulier, cette section dépeint une approche innovant pour représenter une relation (table) et les fonctions de l'algèbre relationnelle en ProVerif. Ensuite, les sections 5.3 à 5.5 décrivent, par l'exemple, comment encoder systématiquement une requête C2QL en ProVerif. La section 5.6 encode les requêtes `[adresse]` et `#rendezvous` pour vérifier que le choix du $crypt_{nom,\texttt{AES}}$ avec $defrag_{date}$ est le bon pour protéger l'application agenda. Enfin, la section 5.7 conclut sur l'utilisation de ProVerif pour vérifier le choix des techniques de cryptographie dans une requête C2QL.

5.1 Le vérificateur de modèles ProVerif

ProVerif (Blanchet et collab., 2014) est un *vérificateur de modèle* (model checker) pour l'analyse automatique des propriétés de sécurité sur les protocoles de cryptographie. Dans ProVerif, les acteurs d'un système sont modélisés avec des processus concurrents qui communiquent par échange de messages sur des canaux. L'outil permet également de modéliser des fonctions de cryptographies et de vérifier que ces fonctions protègent correctement les messages d'un système en s'assurant qu'un attaquant ne puisse pas y accéder.

Formellement, l'outil ProVerif repose sur le modèle de Dolev-Yao (1983), où les messages sont des termes. Une fonction de cryptographie est modélisée par un constructeur qui enveloppe le message avec sa clef de chiffrement. Le résultat produit un nouveau terme ou la fonction apparaît explicitement. Par exemple, un chiffrement symétrique est modélisé par le constructeur `senc` qui produit un nouveau terme fait d'un message (de type `bitstring`) et d'une clef (de type `key`).

```
1 fun senc(bitstring, key): bitstring.
```

Une fonction de cryptographie s'accompagne souvent d'un destructeur qui modélise la découverte d'un message original pourvu que la clef soit fournie. Concrètement, le destructeur est une règle de réécriture qui manipule le terme en *filtrant* (pattern matching) sur ses constructeurs. Par exemple, le constructeur `senc` est accompagné d'un destructeur `sdec` qui réécrit le terme `senc` pour en extraire le message original `m` si la clef `k`, utilisée lors du chiffrement, est connue.

```
2 reduc forall m: bitstring, k: key; sdec(k,senc(m,k)) = m.
```

Dans ce modèle, un attaquant est omnipotent. Il a un contrôle total sur les canaux. Il peut intercepter les messages, en construire de nouveau à partir de ceux interceptés et renvoyer les messages ainsi construits. Par conséquent, si sur un premier canal, le système transmet le terme `senc(m,k)` qui modélise le chiffrement du message `m`. Puis, sur un autre canal, transmet le terme `k` qui modélise la clef. L'attaquant, omnipotent, connaît ces deux termes et est en mesure de retrouver le message original `m`, par la règle de réécriture `sdec`. En ProVerif, vérifier que le terme `m` est confidentiel se fait par la requête suivante.

```
3 query attacker(m).
```

La modélisation des processus et canaux de communications se fait dans un langage proche du π-calcul. C'est un aspect très intéressant pour les requêtes C2QL puisque la transformation, donnée au chapitre précédent (*cf.* §4.4, *fig.* 29), permet d'automatiser la modélisation. Par exemple, le programme suivant modélise trois processus `A`, `B` et `C` avec un canal `toB` pour transmettre un message à `B` et `toC` pour transmettre à `C`.

```
4 free toB, toC: channel.
5 process
6   out(toB, senc(m,k))                (* processus A *)
7   | (in(toB, theCipherM: bitstring); (* processus B *)
8      out(toC, k))
9   | in(toC, theK: key)               (* processus C *)
```

Les opérateurs `in(x,m)` et `out(x,m)` équivalent, respectivement, à l'envoi $\overline{x}\langle m\rangle$ et la réception $x(m)$ d'un message en π-calcul. Dans ce programme, le processus `A` envoie à `B` un message `m` chiffré avec la clef `k` (*l.* 6). En parallèle (`|`), le processus `B` attend le message chiffré (*l.* 7), puis transmet à `C` la clef (*l.* 8). Enfin, le processus `C` attend la clef (*l.* 9). La confidentialité du message `m`, testée par la ligne 3, est compromise dans ce programme. Un attaquant connaît les termes `senc(m,k)` et `k` à la suite de leur envoi ligne 6 et 8. Il peut donc déduire le message `m` avec le destructeur `sdec`.

Une fois modélisée avec le π-calcul, ProVerif transforme les processus en clauses de Horn (Abadi et Blanchet, 2002) pour analyser les propriétés de sécurités. Dans cette transformation, le fait `attacker(m)` symbolise qu'un attaquant peut avoir le message `m`. Dans le même temps, les constructeurs, destructeurs et échanges de messages sont transformés sous forme de règles logiques qui spécifient les implications entre les faits.

Programmer en Prolog (2003)

Le lecteur familier avec la programmation logique (en particulier, Prolog) peut se représenter le constructeur `senc` comme la règle logique qui permet à l'attaquant de construire un message chiffré s'il est en possession d'un message et d'une clef

```
% senc(bitstring, key)
attacker(senc(M,K)) :- attacker(M), attacker(K).
```

le destructeur `sdec` comme la règle qui donne à l'attaquant, la connaissance du message, s'il est en possession de la clef et du message chiffré

```
% forall m,k: sdec(k,senc(m,k)) = m
attacker(M) :- attacker(K), attacker(senc(M,K)).
```

et les échanges sur un canal comme un fait si le message est le premier d'une communication,

```
% out(toB, senc(m,k)) (l.6)
attacker(senc(m,k)) :- true
```

ou comme une règle si le message résulte d'une précédente communication.

```
% in(toB, senc(m,k)) ; out(toC, k) (l.7,8)
attacker(k) :- attacker(senc(m,k))
```

L'encodage de ce petit programme ProVerif vers son équivalent Prolog est fourni en annexe (cf. §A.1.1), ainsi que sur le dépôt de cette thèse[1].

Dans ce programme, savoir si un message `m` est confidentiel revient à tester le prédicat "`?- attacker(m).`". La réponse `true`, qui sera retournée ici, signifie qu'un attaquant peut atteindre le message. Toutefois, la représentation qui est fournie ici est naïve.

En effet, ProVerif a d'abord été développé pour vérifier la propriété de confidentialité (Blanchet, 2001), qui permet de savoir si le message `m` est atteignable par un attaquant. Mais, il a ensuite été étendu avec la vérification de correspondance entre évènements, pour vérifier des propriétés d'authentifications (Blanchet, 2002). Ou encore, l'équivalence observationnelle, pour vérifier des propriétés d'indistinguabilitées (Blanchet et collab., 2008). De plus, l'outil est capable de reconstruire une attaque en donnant la trace d'exécution de l'attaquant.

Enfin, ProVerif est correct, mais pas complet. Ce qui signifie que, si l'outil affirme qu'une propriété est vraie ou fausse, alors cette affirmation est correcte.

1 github.com/rcherrueau/C2QL/tree/master/privacy-checker/horn-transform.org

En revanche, l'outil peut ne pas se positionner sur une propriété et l'analyse peut ne pas se terminer. Cette incomplétude est due à la manière dont ProVerif transforme les processus en clauses de Horn, mais ces cas sont rares (Blanchet et collab., 2014).

5.2 Description de l'approche avec ProVerif

Ce chapitre montre comment encoder systématiquement une requête C2QL en ProVerif. Cet encodage vérifie que les contraintes de confidentialités ne sont pas atteignables par un attaquant dans le modèle de Dolev-Yao (1983). Ceci pour garantir que les techniques de cryptographie choisies par le développeur sont les bonnes.

Les contraintes de confidentialités en Dolev-Yao. Comme énoncé dans la section précédente, ProVerif modélise les processus et canaux du modèle de Dolev-Yao dans un langage proche du π-calcul. Cette modélisation convient parfaitement aux requêtes C2QL qui peuvent également être représentées avec le π-calcul. Reste, néanmoins, à définir comment vérifier la confidentialité d'une contrainte de confidentialité.

Dans le modèle de Dolev-Yao et réciproquement en ProVerif, un attaquant augmente sa connaissance en observant l'information qui transite sur les canaux de communications. Il utilise ensuite cette information pour atteindre le secret de l'application.

Instancié aux applications PbD, les informations qui transitent sur les canaux sont les n-uplets de la base de données. Le secret est modélisé par les contraintes de confidentialités. Dans cette instanciation, l'attaquant utilise les n-uplets qui transitent sur les canaux et s'aide des constructeurs et destructeurs de cryptographie pour atteindre une contrainte de confidentialité. Par exemple, dans l'application agenda, l'attaquant enregistre tous les n-uplets qui transitent sur les canaux. Puis, il applique systématiquement le destructeur de fragmentation (*c.-à-d.*, la défragmentation) sur toutes les combinaisons de n-uplets pour essayer de reconstruire la contrainte $\{date, adresse\}$.

Un encodage naïf de l'application agenda voudrait que la relation des rendez-vous soit modélisée avec tous ses n-uplets. Ce serait néanmoins très fastidieux. Pire, la vérification de la violation, ou non, des contraintes de confidentialités ne serait valide que pour les n-uplets spécifiés. Il faut donc un moyen plus abstrait pour représenter les relations d'une base de données. L'objectif de cette abstraction est qu'il n'y ait pas besoin de spécifier les n-uplets, tout en assurant que la preuve de non-violation soit vraie pour un nombre infini de n-uplets.

Encoder le schéma relationnel. Cette thèse propose une nouvelle approche qui consiste à représenter les relations par leur schéma. Le schéma spécifie les attributs d'une relation et ainsi spécifie le type de cette relation. Il fait foie pour n'importe quels n-uplets, le schéma est donc une bonne abstraction pour raisonner, de manière globale, sur une relation.

En partant de cette idée, l'approche consiste à définir les fonctions de l'algèbre relationnelle, les spécificateurs et les contraintes de confidentialités sur le schéma. Les fonctions de l'algèbre relationnelle et les spécificateurs changent le schéma. Ils définissent quels sont les attributs lisibles ou non. Les contraintes

de confidentialités, quant à elles, indiquent quelles sont les compositions d'attributs qu'il n'est pas sûre de lire. Le tout est encodé dans ProVerif.

Lors de son encodage en ProVerif, la requête C2QL est traduite en plusieurs processus. Cette traduction se fait conformément à la transformation en π-calcul du chapitre 4 (*cf.* §4.4, *fig.* 29). Cependant, en ProVerif les processus appliquent les fonctions de l'algèbre relationnelle sur le schéma, plutôt que sur les n-uplets. L'information transmise sur les canaux de communication est également le schéma.

Dans cet encodage, l'attaquant essaie de reconstruire les contraintes de confidentialités à partir des schémas transmis sur les canaux. Si l'attaquant y arrive, cela signifie que les techniques de cryptographie choisies par le développeur (les spécificateurs) ne protègent pas correctement les contraintes de confidentialités. À l'inverse, si l'attaquant n'y arrive pas, cela signifie que les techniques protègent la requête, quels que soient les n-uplets de la relation considérée.

À propos du modèle de Dolev-Yao. Le lecteur peut se demander si le modèle de Dolev-Yao est bien adapté pour vérifier la confidentialité dans une application PbD. En effet, même si le dessein du modèle de Dolev-Yao est de vérifier la confidentialité des données, celui-ci traque uniquement l'information transmise sur les canaux de communication. Or, une application PbD doit en plus respecter la confidentialité au sein des acteurs hébergés dans le nuage, comme l'agenda SaaS et les bases de données PaaS.

Cette thèse affirme que le modèle est correct si les processus qui modélisent les acteurs du nuage n'appliquent pas de destructeurs de cryptographie. L'intuition est la suivante. Dans une application du nuage, toutes les données sont, au départ, retenues par le client et vont transiter sur les canaux. Par exemple, l'ajout d'un rendez-vous sur l'agenda émane forcement d'Alice et la transmission du rendez-vous se fait sur un canal. De ce fait, il y a deux possibilités. La première est que l'application soit mal protégée. Ainsi, lorsqu'elle transmet une information confidentielle, l'attaquant du modèle de Dolev-Yao va avoir accès à cette information. Il notifiera donc d'un mauvais choix de techniques de cryptographie.

La deuxième possibilité est que l'application soit bien protégée. Si elle est bien protégée, seule de l'information publique ou inintelligible transite sur les canaux. *De facto*, l'attaquant ne notifie pas de violation. Ceci veut également dire que l'information retenue par les processus du nuage est soit publique, soit inintelligible. Dans ce cas, le seul moyen pour un processus du nuage d'avoir accès à une donnée confidentielle est d'appliquer un destructeur de cryptographie.

En interdisant aux processus hébergés dans le nuage d'appliquer un destructeur de cryptographie, l'encodage vérifie que le choix des techniques de cryptographie préserve la confidentialité lors des communications et dans les processus du nuage. Pour rappel, cette doctrine, qui consiste à ne pas appliquer de destructeur sur les acteurs du nuage, est imposée par la traduction d'une requête C2QL en π-calcul (*cf.* §4.4, *fig.* 26 – 28).

5.3 Encodage du schéma et des fonctions de l'algèbre relationnelle

Le schéma relationnel d'une table se compose d'une liste d'attributs. Un attribut est traditionnellement dans une forme lisible spécifié par le constructeur `brut(bitstring)`.

```
1 type attribut.
2 fun brut(bitstring): attribut.
```

L'argument de type `bitstring` spécifie le nom de l'attribut. Pour la relation des rendez-vous, l'argument peut prendre la valeur `d` pour l'attribut des dates, `n` pour l'attribut des noms ou `a` pour l'attribut des adresses. Par exemple, l'encodage ci-après considère une première version du schéma relationnel des rendez-vous qui consiste en une date, un nom et une adresse, tous lisibles.

```
3 letfun leSchema = (brut(d), brut(n), brut(a)).
```

Le terme qui représente le schéma relationnel est encodé avec un 3-uplet. En ProVerif, une fois que l'attaquant est en possession d'un 3-uplet, il a la possibilité d'extraire n'importe quel attribut. L'inverse est également vrai. Si l'attaquant est en possession des trois attributs lisibles, il peut recomposer un 3-uplet.

Ici, le terme qui représente le schéma est défini au moyen d'une macro introduite par `letfun`. Cette macro spécifie qu'au moment de la vérification, les occurrences de `leSchema` sont substituées par le 3-uplet. Cette notation simplifie la lecture du programme ProVerif.

Dans la suite, le programme encode les fonctions de l'algèbre relationnelle avec des règles de réécriture sur le schéma des rendez-vous.

La projection. La projection conserve un sous-ensemble des attributs d'un schéma. Les attributs qui ne font pas partie de ce sous-ensemble n'apparaissent plus dans le schéma et sont remplacés par `unit` (*l.* 4). Une règle produit l'attribut `unit` quand un attribut n'apparaît plus dans le résultat. Ceci signifie que plus aucune information ne peut être inférée de cet attribut.

Par exemple, la projection sur les noms et les adresses (notée, `(n,a)`) conserve l'attribut nom et adresse, mais retourne `unit` pour la date.

```
4 const unit: attribut.
5
6 forall ad: attribut, an: attribut, aa: attribut;
7 proj((n,a), (ad,an,aa)) = (unit,an,aa);
```

Il est important de remarquer que l'encodage utilise des variables (`ad`, `an`, `aa`). Ces variables spécifient que la projection s'applique, quelque soit la forme des attributs : qu'ils soient lisibles (`brut`), inexistants (`unit`), ou chiffrés (`senc`) comme le montrera la section 5.5. Le dessein de cet encodage avec des variables est d'être cohérent avec les équations 10 et 11 des lois algébriques du langage C2QL (*cf.* §4.2.3) qui spécifient que la projection peut s'appliquer sur une donnée chiffrée.

$\pi_\delta \circ decrypt_{\alpha,\mathtt{c}} \overset{10}{\equiv} decrypt_{\alpha,\mathtt{c}} \circ \pi_\delta \; si \; \alpha \in \delta$

$\pi_\delta \circ decrypt_{\alpha,\mathtt{c}} \overset{11}{\equiv} \pi_\delta \; si \; \alpha \notin \delta$

Du point de vue de l'attaquant, cette règle spécifie que les dates ne sont plus lisibles après la projection. Pour s'en convaincre, il suffit de lire la clause de Horn de ce destructeur produite en interne par ProVerif.

```
attacker((unit,N,A)) :- attacker((n,a)), attacker((D,N,A)).
```

La sélection. La sélection modifie une relation en filtrant ses n-uplets sur un prédicat. En revanche, elle ne modifie pas le schéma de cette relation. Par conséquent, la règle de réécriture n'introduit pas de `unit`.

Toutefois, une sélection qui implique par exemple, un prédicat sur `d` (comme *les dates de la semaine prochaine*) requière que les dates soient lisibles (notée, `brut(d)`) pour que le prédicat soit évalué. Les autres attributs peuvent être lisibles ou absents, ce qui est modélisé par l'utilisation des variables `an` et `aa`.

Cette encodage n'autorise pas de sélection homomorphe. Ce point est discuté à la section 5.5.

```
reduc forall an: attribut, aa: attribut;
select(d, (brut(d),an,aa)) = (brut(d),an,aa);
```

L'agrégation par dénombrement. L'agrégation par dénombrement rassemble les n-uplets d'une relation dans des groupes en testant l'égalité des valeurs des attributs spécifiés. Puis, elle compte le nombre de n-uplets par groupe.

Les besoins de la fonction d'agrégation sont les mêmes que pour la sélection. Une agrégation sur les `d` a besoin que les dates soient lisibles (`brute`). Mais, l'agrégation par dénombrement va, en plus, modifier le schéma de la relation en omettant les attributs non spécifiés et en les remplaçant par le nombre de lignes dans le groupe. Cette omission est encodée par l'attribut `unit`.

Le nouveau schéma ne fait pas apparaître la colonne qui contient le nombre de lignes, car cette information n'est pas importante pour vérifier les contraintes de confidentialités.

```
reduc forall an:attribut, aa:attribut;
count(d, (brut(d),an,aa)) = (brut(d),unit,unit).
```

Un cas particulier est celui quand l'agrégation porte sur tous les attributs du schéma (*ex.*, un dénombrement sur les dates, noms et adresses). Ce cas correspond au `COUNT(*)` en langage SQL. Il retourne un seul nombre qui est le cardinal de la relation. Dans ce cas, les trois attributs de la relation rendez-vous ne sont plus lisibles dans le résultat.

```
reduc
count((d,n,a), (brut(d),brut(n),brut(a))) = (unit,unit,unit).
```

La généralisation de cette façon de modéliser les fonctions à n'importe quels schémas arbitraires est directe. L'encodage est systématique. Il donne des règles de réécriture pour tous les sous-ensembles possibles d'attributs d'un schéma relationnel, y compris l'ensemble vide et le schéma lui-même. Ceci produit $\mathcal{P}(\text{schéma})$ règles de réécriture par fonction, soit l'ensemble des parties du schéma. Le nombre peut sembler important, mais il peut être très fortement réduit au nombre de règles de réécriture effectivement utilisées dans l'application. Par exemple, pour le cas de l'agenda personnel en ligne, l'encodage systématique produit 24 règles de réécriture, mais seulement 4 sont utilisées par les requêtes `#rendezvous` et `[adresse]`.

Un premier encodage des requêtes `[adresse]` et `#rendezvous`

Une première version de l'agenda personnel en ligne montre comment encoder les requêtes `[adresse]` et `#rendezvous` en ProVerif. Cette première version ne considère pas de techniques de cryptographie. Elle modélise une application sans protections, à l'instar des agendas personnels hébergés au-

jourd'hui dans le nuage. Par conséquent, le schéma relationnel considéré pour les rendez-vous est celui de la ligne 3 où tous les attributs représentent des valeurs lisibles.

Le système est fait de trois processus : Alice, l'agenda personnel SaaS et la base de données des rendez-vous PaaS. Mais, dans la suite, l'encodage ProVerif ne modélise pas Alice. Ceci est inutile puisque tous les calculs faits chez Alice sont sûrs par définition.

En ProVerif, le mot clef `process` définit le processus principal. L'introduction de ce chapitre montre que c'est sous ce mot que le développeur doit modéliser son application (*l.* 5 du programme ProVerif de la section 5.1). Néanmoins, pour faciliter le développement des *sous-processus*, ceux-ci peuvent être exprimés à l'aide d'une macro introduite par le mot `let`, comme dans la suite.

Par exemple, la requête relationnelle `[adresse]`, qui retourne la liste des adresses et contacts visités par Alice, est modélisée par le sous-processus de la ligne 15 à 17. À la ligne 16, la requête reçoit sur son canal de réception `adressesUrl`, le canal `to`, utilisé à la ligne 17 pour retourner le résultat de la requête.

```
14 free adressesUrl: channel.
15 let Adresses =
16   in (adressesUrl, to: channel);
17   out(to, proj((n,a), leSchema())).
```

Lorsqu'une requête est faite de plusieurs fonctions de l'algèbre relationnelle, les fonctions sont composées fonctionnellement (*l.* 22). Par exemple, calculer la requête `#rendezvous` requiert une sélection, puis une projection et enfin, une agrégation par dénombrement. Cette composition fonctionnelle peut également être décomposée avec des `let` (*l.* 21) exactement comme ce qui est fait dans la traduction en π-calcul (*cf.* §4.4.2, *fig.* 25)

```
18 free rendezvousUrl: channel.
19 let Rendezvous =
20   in(rendezvousUrl, to: channel);
21   let r = select((d,n,a), leSchema()) in
22   out(to, count(d, proj(d, r))).
```

Conformément à l'approche définie dans la section précédente (*cf.* §5.2), les requêtes sont calculées sur `leSchema` relationnel des rendez-vous.

Une fois toutes les requêtes définies, elles sont composées parallèlement (`|`) dans le processus qui modélise la base de données.

```
23 let BD = !Adresses | !Rendezvous.
```

Puisqu'une requête peut être exécutée plusieurs fois, l'opérateur de réplication (`!`) préfixe les sous-processus. Ceci génère autant de copies des requêtes que nécessaire.

Le prochain sous-processus est l'agenda personnel. Les lignes 27 et 28 modélisent respectivement l'invocation des requêtes `[adresse]` et `#rendezvous`. Chaque invocation envoie `toAgenda` comme canal de réponse ce qui signifie que le résultat sera retourné à l'agenda. La ligne 29 modélise la réception du résultat de l'une ou l'autre requête et son transfert à Alice.

```
24 free toAgenda: channel.
25 free toAlice: channel.
```

```
26 let Agenda =
27    !out(adressesUrl, toAgenda) |
28    !out(rendezvousUrl, toAgenda) |
29    !(in(toAgenda, res: bitstring); out(toAlice, res)).
```

Finalement, le processus principal modélise le système complet en composant parallèlement l'agenda et la base de données.

```
30 process BD | Agenda
```

Le lecteur qui serait tenté d'exécuter ce programme ProVerif ne verrait pas de violation, même si cet encodage modélise un agenda sans protection. La raison est simple, le présent encodage ne spécifie pas quelles sont les informations confidentielles dans le système. La section suivante s'attelle à cette tâche.

5.4 Encodage des contraintes de confidentialités

Le dessein de ce chapitre est de vérifier systématiquement que les techniques de cryptographie, utilisées dans une requête C2QL, protègent les données personnelles. Dans le langage C2QL, les données personnelles sont spécifiées par des contraintes de confidentialités (*cf.* §2.4.2). Elles s'expriment sur les attributs d'un schéma relationnel et définissent quelles sont les valeurs d'attributs confidentielles aux yeux d'un attaquant.

Cette section montre comment encoder les contraintes de confidentialités en ProVerif. Plus spécifiquement, l'encodage fournit à l'attaquant des règles qui lui permettent d'atteindre une contrainte à partir d'un schéma relationnel. Ceci faisant, le programme ProVerif peut être exécuté pour s'assurer qu'un attaquant n'atteint pas les contraintes et ainsi prouver que la requête préserve la confidentialité des données personnelles.

L'agenda personnel définit deux contraintes $\{nom\}$ et $\{date, adresse\}$. Ces contraintes sont encodées lignes 31 et 32 par des constantes privées. Ensuite, les lignes 34 et 35 vérifient si ces constantes sont atteignables par un attaquant.

```
31 const cc_n:  bitstring [private].
32 const cc_da: bitstring [private].
33
34 query attacker(cc_n).
35 query attacker(cc_da).
```

Le mot clef `private` exclut le terme de la connaissance de l'attaquant. Dans cet encodage, il est primordial que les contraintes de confidentialités soient privées. Faute de quoi, l'attaquant atteindrait systématiquement les contraintes.

Encore une fois, passer dans le monde des clauses de Horn explique ce comportement. Si les contraintes ne sont pas privées, ProVerif traduit les lignes 31 et 32 par deux faits comme ci-après. Il est alors triviale que ces faits satisfassent les prédicats "`? attacker(cc_n).`" et "`? attacker(cc_da).`" spécifiés aux lignes 34 et 35.

```
attacker(cc_n) :- true.  % const cc_n
attacker(cc_da) :- true. % const cc_da
```

Code 30 – *Résultat de la vérification de la confidentialité dans un agenda sans protection.*

```
[rcherr@nixos:tmp]$ proverif agenda01-nosecurity.pv
-- Query not attacker_bitstring(pc_da)
goal reachable: attacker_bitstring(cc_da)
  ...
The attacker has the message cc_da.
A trace has been found.
RESULT not attacker_bitstring(cc_da) is false.
-- Query not attacker_bitstring(cc_n)
goal reachable: attacker_bitstring(cc_n)
  ...
The attacker has the message cc_n.
A trace has been found.
RESULT not attacker_bitstring(cc_n) is false.
```

En revanche, avec le mot clef `private`, ProVerif ne traduit pas les constantes. Ainsi, le seul moyen pour l'attaquant d'obtenir ces constantes est au moyen des règles de réécriture. La suite profite de ce mécanisme pour définir comment un attaquant viole une contrainte de confidentialité à partir d'un schéma relationnel.

Les règles de réécriture ci-après font le lien entre un schéma relationnel qui fuite des données confidentielles et la violation d'une contrainte de confidentialité. Par exemple, la règle à la ligne 36 spécifie que tous les schémas relationnels où les noms sont lisibles violent la contrainte $\{nom\}$. De même, la règle à la ligne 39 spécifie que tous les schémas relationnels où les dates et adresses sont lisibles violent la contrainte $\{date, adresse\}$.

```
36 reduc forall ad: attribut, aa: attribut;
37 confidentiel_n((ad,brut(n),aa)) = cc_n.
38
39 reduc forall an: attribut;
40 confidentiel_da((brut(d),an,brut(a))) = cc_da.
```

La généralisation des règles de déductions pour toutes les applications est directe. La génération systématique énumère toutes les combinaisons possibles sur un schéma relationnel où un attribut serait lisible (`brut`) ou non. Ceci produit $2^{|\text{schéma}|}$ règles de déductions, mais seules les règles qui violent une contrainte de confidentialité doivent être conservées. Par exemple, pour le cas de l'agenda personnel en ligne, l'encodage systématique produit 8 règles de déductions, mais seulement 2 violent une contrainte de confidentialité.

Exemple de vérification sur l'application agenda

Maintenant que l'encodage spécifie quelles sont les données confidentielles, le programme ProVerif de la section précédente (*cf.* §5.3) peut être vérifié. Le résultat de la vérification sur cet agenda personnel sans protections est donné dans le code 30. Il spécifie que les deux contraintes sont atteignables par un attaquant. Un résultat logique puisque l'application n'utilise pas de techniques de cryptographie.

2 Les sources du fichier "agenda01-nosecurity.pv" sont disponibles à l'adresse github.com/rcherrueau/C2QL/tree/master/privacy-checker/agenda01-nosecurity.pv.

```
[rcherr@nixos:tmp]$ proverif agenda02-local.pv
-- Query not attacker_bitstring(cc_da)
RESULT not attacker_bitstring(cc_da) is true.
-- Query not attacker_bitstring(cc_n)
RESULT not attacker_bitstring(cc_n) is true.
```

Code 31 – *Résultat de la vérification de la confidentialité dans un agenda exécuté localement.*

Les ellipses dans le résultat masquent la trace de l'attaquant. Mais pour la contrainte `cc_n` par exemple, ProVerif précise que l'attaquant récupère les noms lorsque la base de données transmet le résultat de la requête `[adresse]` à l'agenda. Le lecteur peut voir le détail de cette trace en exécutant l'outil ProVerif sur les sources fournies avec cette thèse[2].

Un moyen simple de protéger le système est de déclarer tous les canaux de communication privés. De ce fait, l'attaquant ne peut plus lire les schémas qui transitent entre les processus et ne peut plus atteindre les contraintes de confidentialités (*code* 31)[3].

Intuitivement, cela revient à modéliser et évaluer une application locale. Un aspect qui n'est pas satisfaisant du point de vue de l'approche du *nuage confidentiel* dont l'objectif premier est que l'application s'exécute au maximum dans le nuage. Le langage C2QL répond à ce problème en permettant de composer efficacement les techniques de cryptographie.

5.5 Encodage du chiffrement et de la fragmentation verticale

Cette section montre comment les techniques de cryptographie sont encodées en ProVerif pour pouvoir ensuite vérifier les requêtes C2QL. Jusqu'à maintenant, un attribut est soit lisible (`brut`), soit absent (`unit`). Ce type d'information est un peu limité pour s'attaquer au respect de la vie privée. En réponse, le langage C2QL fournit les spécificateurs $crypt$ et $frag$ qui protègent les valeurs de ces attributs. Comme précédemment, l'encodage de ces techniques porte sur le schéma de la relation pour s'abstraire des n-uplets concrets.

Chiffrement symétrique. Le chiffrement symétrique transforme une information de telle manière que seuls les agents autorisés puissent la lire. Dans le contexte de l'agenda personnel en ligne, le chiffrement peut être utilisé pour protéger la contrainte $\{nom\}$.

L'encodage est similaire à celui donné en introduction de ce chapitre (*cf.* lignes 1 et 2 du programme ProVerif de la section 5.1). La seule différence est que le constructeur, à la ligne 42, produit un attribut (comme `brut`), plutôt qu'un `bitstring`, car l'objectif de ce constructeur est de protéger les attributs du schéma relationnel. Le `bitstring`, en argument du `senc`, a la même fonction que celui en argument de `brut` (*l.* 2) : il spécifie le nom de l'attribut. Intuitivement, cela modélise que toutes les valeurs de cet attribut sont chiffrées symétriquement avec la clef `k`.

```
41 type key.
42 fun senc(bitstring, key): attribut.
```

3 Les source du fichier "agenda02-local.pv" sont disponibles à l'adresse github.com/rcherrueau/C2QL/tree/master/privacy-checker/agenda02-local.pv.

De la même manière, le destructeur à la ligne 43 produit un attribut lisible (`brut`), plutôt qu'un `bitstring`, si la clef de chiffrement `k` est connue.

```
43 reduc forall a:bitstring, k:key; sdec(k,senc(a,k)) = brut(a).
```

$\sigma_{p_\delta} \circ decrypt_{\alpha,\mathtt{c}} \overset{14}{\equiv} decrypt_{\alpha,\mathtt{c}} \circ \sigma_{\mathtt{c} \Rightarrow p_\delta}$ *si* $\alpha \in \delta$

Cet encodage pose néanmoins un problème pour les fonctions de sélection et d'agrégation par dénombrement. En effet, d'après leurs encodages (*cf.* §5.3), ces fonctions sont applicables que si l'attribut, sur lequel porte le prédicat, est lisible. Or, les lois du langage C2QL montrent que la sélection et l'agrégation peuvent se faire sur une donnée chiffrée si ce chiffrement supporte le prédicat (*eq.* 14 et 18). De plus, la section 2.4.3.2 du chapitre 2 spécifie qu'un chiffrement symétrique déterministe, comme AES, supporte le test d'égalité. Il faut donc augmenter les règles de réécriture de la sélection et de l'agrégation pour rendre compte de cette spécificité.

$count_\delta \circ decrypt_{\alpha,\mathtt{c}} \overset{18}{\equiv} decrypt_{\alpha,\mathtt{c}} \circ count_{\mathtt{c} \Rightarrow \delta}$ *si* $\alpha \in \delta$

Les nouvelles règles se comportent exactement comme les précédentes, mais elles permettent aux fonctions de s'évaluer sur une valeur d'attribut chiffrée symétriquement. Après évaluation de la fonction, les valeurs chiffrées restent chiffrées.

```
44 reduc forall an: attribut, aa: attribut;
45 selectAES(d, (senc(d,k),an,aa)) = (senc(d,k),an,aa).
46
47 reduc forall an: attribut, aa: attribut;
48 countAES(d, (senc(d,k),an,aa)) = (senc(d,k),unit,unit).
```

Comme spécifiés dans la section 2.4.3.2 du chapitre 2, les autres schémas homomorphes requièrent l'emploi d'une clef publique. Même si l'exemple de l'agenda n'en a pas l'utilité, la suite montre comment encoder ces fonctions en ProVerif.

L'encodage s'inspire de celui du chiffrement asymétrique (Blanchet et collab., 2014). Le constructeur `henc` (*l.* 51) produit un terme qui enveloppe le nom de l'attribut et une clef publique (de type `pkey`). La clef publique se dérive à partir du constructeur `pk` et d'une clef privée (*l.* 50). Dans cet encodage, le destructeur `hdec` (*l.* 53) spécifie qu'il faut fournir la clef privée `k` qui a servi à générer la clef publique `pk(k)` pour déchiffrer les valeurs d'un attribut.

```
49 type pkey.
50 fun pk(key): pkey.
51 fun henc(bitstring, pkey): attribut.
52 reduc forall a: bitstring, k: key;
53 hdec(k,henc(a,pk(k))) = brut(a).
```

La clef publique a cette spécificité qu'elle peut être publiée sans compromettre la sécurité de l'application.

Les fonctions homomorphes de l'algèbre relationnelle sont ensuite définies en utilisant cette clef publique. Par exemple, une sélection homomorphe dont le prédicat implique de faire un calcul sur une date chiffrée (`henc(d,pk)`) n'a pas besoin de la clef privée qui déchiffre les dates, mais seulement de la clef publique `pk` pour faire le calcul sur les valeurs chiffrées.

```
54 reduc forall an: attribut, aa: attribut, pk: pkey;
55 hselect(d, pk, (henc(d,pk),an,aa)) = (henc(d,pk),an,aa).
```

Le lecteur familier avec le vérificateur ProVerif fera remarquer que cet encodage est inhabituel pour le chiffrement homomorphe et il aura raison. En effet, le comportement d'une opération homomorphe est décrit au moyen d'une équation, par exemple $\diamond\varepsilon(x) = \varepsilon(\diamond x)$, avec ε un constructeur de cryptographie, $\diamondsuit$ une opération unaire et x un message. L'outil ProVerif propose une construction pour vérifier ce type de relation entre des constructeurs (nommé `equation`). Par conséquent, le lecteur pourrait se demander pourquoi le `hselect` n'utilise pas cette construction. La réponse est simple : l'encodage décrit ici n'a pas pour ambition de vérifier la relation `hselect(d, pk (henc(d), an, aa)) = henc(select(d, (d, an, aa)), pk)`. L'ambition de cet encodage est de modéliser dans quelle forme sont les attributs après un `hselect`. Par conséquent, la construction `equation` n'est pas adaptée.

Ce paragraphe répond à une critique du "reviewer #3" lors d'une soumission à SAC'16 :

> *"I have some concerns about the proposed theory for homomorphic encryption, which is encoded as destructors. The gist of homomorphic encryption is in its algebraic properties, and it would be better modelled as* `equation` *in ProVerif."*

Fragmentation verticale. La fragmentation verticale sépare l'information en fragments non réunissables de telle manière que seuls les agents autorisés puissent recomposer l'information originale.

Dans le cas de l'agenda personnel en ligne, la fragmentation a pour dessein de satisfaire la contrainte $\{date, adresse\}$ en séparant le schéma relationnel $(date, nom, adresse)$ en deux fragments : $(date)$ à gauche et $(nom, adresse)$ à droite. Ceci s'encode en redéfinissant le schéma relationnel de la ligne 3 en deux schémas.

```
56 (* Schéma :          date     nom      adresse *)
57 letfun leSchemaG = ( brut(d), unit,    unit     ).
58 letfun leSchemaD = ( unit,    brut(n), brut(a)  ).
```

Le schéma du fragment de gauche (*l.* 57) est défini comme le schéma relationnel dont seules les dates sont lisibles. À l'inverse, le schéma du fragment de droite (*l.* 58) est défini comme le schéma relationnel dont seuls les noms et adresses sont lisibles.

La règle de réécriture `defrag` spécifie que si un attaquant peut avoir les schémas des deux fragments, alors il peut recomposer le schéma original.

```
59 reduc forall ad:attribut, an:attribut, aa:attribut;
60 defrag((ad,unit,unit), (unit,an,aa)) = (ad,an,aa).
```

Maintenant que l'encodage des fonctions de cryptographie est défini, il ne reste plus qu'à réencoder les requêtes `[adresse]` et `#rendezvous` qui suivent l'approche du *nuage confidentiel*. L'outil ProVerif pourra ensuite être exécuté pour vérifier automatiquement si la composition choisie protège les contraintes de confidentialités.

5.6 Encodage de l'agenda personnel en ligne sécurisé par composition

Le schéma relationnel de l'agenda personnel en ligne peut maintenant être protégé. Le dessein est d'assurer que les techniques de cryptographie choisies sont les bonnes en vérifiant qu'un attaquant ne peut pas violer les contraintes de confidentialités. Conformément à ce qui est fait dans les chapitres 3 et 4, le développeur choisit les techniques de fragmentation, chiffrement symétrique et calculs côté client.

La relation des rendez-vous est fragmentée en deux parties. Le fragment de gauche contient les dates. Le fragment de droite contient les noms et les adresses. Ainsi la contrainte $\{date, adresse\}$ est protégée. Ensuite, les noms sont protégés avec un chiffrement symétrique pour satisfaire la contrainte atomique $\{nom\}$.

Ce choix de techniques de cryptographie produit deux schémas (*l.* 63,64). Le premier contient les dates de manière lisibles. Le second contient les noms chiffrés et les adresses lisibles. L'encodage précise que les noms sont chiffrés symétriquement avec la clef secrète d'Alice. D'ailleurs, ces schémas peuvent automatiquement être produits par la traduction en π-calcul puisqu'une relation est annotée par son schéma relationnel (*cf.* §4.4.2, *fig.* 25).

```
61 const aliceKey: key [private].
62
63 letfun leSchemaG = (brut(d), unit, unit).
64 letfun leSchemaD = (unit, senc(n,aliceKey), brut(a)).
```

En utilisant ce choix de techniques et en se basant sur les lois du chapitre précédent, le développeur protège et optimise les requêtes `[adresse]` et `#rendezvous`. Les définitions de ces requêtes dans le langage C2QL sont redonnées ici :

$$\begin{aligned}\texttt{[adresse]} \equiv\ & \sigma_{nom\ \texttt{LIKE}\ \text{`}C*\text{'}} \circ decrypt_{nom,\texttt{AES}} \\ & \circ defrag_{date}\ (\pi_{\emptyset}, \pi_{nom,adresse}) \\ & \circ frag_{date} \circ crypt_{nom,\texttt{AES}}\end{aligned}$$

$$\begin{aligned}\texttt{\#rendezvous} \equiv\ & defrag_{date}(count_{date} \circ \pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]}, \\ & \pi_{\emptyset} \circ \sigma_{adresse=\text{`}Bureau\text{'}\wedge\texttt{AES}\Rightarrow nom=\text{`}Bob\text{'}}) \\ & \circ frag_{date} \circ crypt_{nom,\texttt{AES}}\end{aligned}$$

L'encodage en ProVerif se définit comme suit. Les sous-processus qui modélisent les acteurs du nuage sont introduits avec le mot clef `let` pour faciliter la lecture. Mais, les requêtes pourraient directement être encodées sous le processus principal (`process`) en suivant la transformation en π-calcul donnée au chapitre précédent.

La requête `[adresse]` opère sur les fragments de droite et gauche, modélisés par les sous processus `AdressesD` et `AdressesG`. Pour rappel, la requête opère parallèlement sur ces fragments. Sur le fragment de droite, à la ligne 67, la requête reçoit sur son canal de réception `adressesDUrl`, le canal `to`, utilisé à la ligne 68 pour retourner le résultat de la requête. Il est important de noter que cette requête est évaluée sur `leSchemaD` qui est le schéma relationnel du fragment de droite.

```
65 free adressesDUrl: channel.
66 let AdressesD =
67   in (adressesDUrl, to: channel);
68   out(to, proj((n,a), leSchemaD())).
```

Le principe est le même pour le fragment de gauche.

```
69 let AdressesG =
70   in (adressesGUrl, to: channel);
71   out(to, proj((), leSchemaG())).
```

La requête `#rendezvous` s'applique sur les fragments de droite et gauche, modélisés ici par les sous processus `RendezvousD` et `RendezvousG`. Pour rappel, la requête calcule sur le fragment de droite une liste d'identifiants utilisés, plus tard, dans la requête sur le fragment de gauche (*cf.* §2.4.3.3). La sélection sur les noms et adresses dans le fragment de droite est séparée en deux sélections successives (*l.* 75). En particulier, la première sélection sur les noms utilise la règle de réécriture `selectAES` pour respecter la contrainte `AES` $\Rightarrow$ *nom* de la spécification en C2QL. Le calcul d'identifiants est également modélisé par la projection vide, là encore, conformément à la spécification en C2QL.

Séparer une sélection en plusieurs sélections successives se fait avec les lois de cascades (eq. 4, cf. §2.4.3.3 – Ullman, 1982).

```
72 free rendezvousDUrl: channel.
73 let RendezvousD =
74   in(rendezvousDUrl, to: channel);
75   out(to, proj((), select(a, selectAES(n, leSchemaD())))).
```

Les identifiants sont, plus tard, récupérés par le processus `RendezvousG` du fragment de gauche. Ils sont récupérés sur le canal de réception à la ligne 78.

```
76 free rendezvousGUrl: channel.
77 let RendezvousG =
78   in(rendezvousGUrl, (ids: bitstring, to: channel));
79   out(to, count(d, proj(d, select(d, leSchemaG())))).
```

Une fois toutes les requêtes définies, elles sont composées parallèlement dans les processus qui modélisent les fragments PaaS. Chaque requête est répliquée pour modéliser le fait qu'elle puisse être interrogée un nombre infini de fois.

```
80 let FragD = !AdressesD | !RendezvousD.
81 let FragG = !AdressesG | !RendezvousG.
```

L'agenda SaaS joue le rôle d'intermédiaire entre Alice et les fragments. Toutefois, l'encodage ProVerif ne s'embête pas à modéliser Alice puisque tous les calculs faits chez Alice sont confidentiels. Les lignes 85 et 86 modélisent l'invocation parallèle de la requête `[adresse]`. Conformément à la transformation en π-calcul, l'invocation fournit le canal d'Alice comme canal de retour.

```
82 free toAgenda: channel.
83 free toAlice: channel.
84 let Agenda =
85   !(out(adressesDUrl, toAlice)            (* [adresse]   *)
86     | out(adressesGUrl, toAlice)) |
```

Ensuite, les lignes 87 à 89 modélisent l'invocation de la requête `#rendezvous`. Cette invocation suit une stratégie séquentielle. En premier, l'agenda requête le fragment de droite (*l.* 87) et fournit son canal pour récupérer les identifiants. Il attend alors les identifiants (*l.* 88) et les retransmet au fragment de gauche (*l.* 89) pour qu'il réalise la fin de la requête. En plus des identifiants, l'agenda transmet le canal d'Alice pour la réponse.

```
87   !(out(rendezvousDUrl, toAgenda);        (* #rendezvous *)
88     in(toAgenda, ids: bitstring);
89     out(rendezvousGUrl, (ids, toAlice))).
```

La stratégie séquentielle est imposée par l'équivalence 20 qui permet de faire rentrer le *count* à l'intérieur du *defrag*. La justification est donnée dans le chapitre précédent (*cf.* §4.2.5), mais brièvement, la stratégie séquentielle est

$count_\delta \circ defrag_{\delta'}\ (id, \pi_\emptyset) \equiv defrag_{\delta'}\ (count_\delta, \pi_\emptyset)$ *si* $\delta \subseteq \delta'$

obligatoire pour que la valeur calculée par le *count* soit correcte. Et même si l'encodage ProVerif ne calcule pas de valeur, il est important de respecter les interactions réelles de l'application pour que la vérification soit valable.

Finalement, comme pour la version précédente sans protection, le processus principal compose parallèlement tous les sous processus pour modéliser le système complet.

```
90 process FragD | FragG | Agenda
```

Le lecteur qui souhaiterait avoir une représentation graphique des processus et canaux peut se reporter aux figures 14 et 15 du chapitre 3.

La vérification de manière séparée des requêtes `[adresse]` et `#rendezvous` révèle, comme attendu, qu'un attaquant ne peut pas violer les contraintes de confidentialités. En revanche, vérifier les deux requêtes en même temps montre une violation quelque peu inattendue. En effet, la trace de l'attaquant ProVerif indique que la requête `[adresse]` révèle des adresses. Au même instant, la requête `#rendezvous` révèle des dates. Par conséquent, l'attaquant reconstruit la contrainte $\{date, adresse\}$.

Ce problème de confidentialité s'explique par le fait que l'attaquant dans le modèle de Dolev-Yao est omnipotent, alors que la fragmentation fait l'hypothèse que les fragments sont indépendants. L'application peut quand même être vérifiée confidentielle en utilisant un canal privé pour les réponses faites à Alice. Typiquement, dans une application du nuage un canal privé représente une communication sécurisée par `https` (Fielding et Reschke, 2014).

Les quatre versions de l'encodage de l'agenda, à savoir : sans protection, local, sécurisé par composition et sécurisé par composition avec retour à Alice chiffré, ainsi que les instructions pour vérifier les contraintes de confidentialités, sont disponibles sur le répertoire Git de cette thèse[4].

5.7 Conclusion

Ce chapitre montre comment encoder systématiquement une requête C2QL en ProVerif. Cet encodage vérifie qu'un attaquant, au sens du modèle de Dolev-Yao, ne viole pas les contraintes de confidentialités de l'application. Ainsi, l'encodage indique au développeur s'il a bien choisi ses techniques de cryptographie (spécificateurs).

L'encodage s'obtient à partir de la transformation d'une requête C2QL en π-calcul. Par conséquent, l'encodage peut être obtenu automatiquement grâce à la traduction du chapitre 4. Mais la vraie nouveauté de cet encodage réside dans sa modélisation d'une relation. En effet, l'outil ProVerif a été utilisé à maintes reprises pour vérifier la confidentialité d'un message. La page personnelle de Blanchet donne une liste exhaustive de tous ces articles[5]. Pourtant (étonnement), il n'existe pas d'article pour expliquer comment modéliser la vérification de la confidentialité des attributs d'une relation. Notamment, comment certaines fonctions de l'algèbre relationnelle, comme la projection, participent à rendre une donnée confidentielle.

4 Les sources de l'encodage sont disponibles à l'adresse github.com/rcherrueau/C2QL/tree/master/privacy-checker.

5 prosecco.gforge.inria.fr/personal/bblanche/proverif/proverif-users.html

Ce chapitre propose de modéliser une relation par son schéma relationnel. Le schéma relationnel est une abstraction particulièrement bien adapté pour modéliser la vérification de la confidentialité des attributs d'une relation. Premièrement, avec les constructeurs, il permet d'annoter les attributs pour modéliser comment sont protégées les valeurs de ces attributs (`brut`, `unit`, *etc.*). Deuxièmement, avec les destructeurs, il permet de spécifier le comportement des fonctions de l'algèbre relationnelle pour modéliser comment celles-ci participent à la protection (`proj`, `count`, *etc.*). Enfin, puisque le schéma relationnel abstrait les n-uplets, une vérification correspond à un nombre illimité d'exécutions avec n'importe quelles valeurs de n-uplets concrets.

Il y a toutefois une contrepartie à ne pas représenter les valeurs des n-uplets. Sans les valeurs, l'attaquant de ProVerif peut retourner des faux positifs (*c.-à-d.*, de fausses attaques). Un exemple trivial est celui d'une relation qui ne contient jamais de n-uplets, mais qui est mal protégée par ses spécificateurs. Dans l'encodage présenté ici, la vérification de cette relation va retourner une attaque, alors qu'en réalité aucune attaque ne peut être faite, car il n'y a pas de données.

La fin de ce chapitre montre que le modèle d'attaquant de Dolev-Yao ne prend pas en compte l'hypothèse que les fragments sont indépendants. Néanmoins, même sans cette hypothèse, les contraintes de confidentialités peuvent toujours être protégées si les canaux de réponse à Alice sont sécurisés. Cette information est très intéressante puisqu'elle montre, aux septiques de la fragmentation verticale, que cette technique est sûre dans le modèle de Dolev-Yao.

L'encodage ProVerif assure au développeur la pertinence de ses choix de techniques de cryptographie pour protéger son application. En revanche, l'encodage ne lui assure pas la pertinence de la composition de ses techniques. Par conséquent, un développeur peut par exemple écrire une requête qui teste une expression régulière sur des données chiffrées, ce qui n'a pas de sens. Le chapitre suivant (*chap.* 6) propose une implémentation du langage C2QL. Cette implémentation repose sur un système de types dépendants. Il garantit que la composition des techniques de cryptographie a du sens du point de vue de l'exécution. Il assure également que plusieurs requêtes, avec des choix de techniques de cryptographie différentes, sont compatibles pour pouvoir être exécutées par les mêmes acteurs du nuage.

Implémenter le langage C2QL

6

Avec le langage C2QL, un développeur écrit des requêtes qui s'exécutent sur un nuage sécurisé avec plusieurs techniques de cryptographie. Le langage s'accompagne de lois algébriques qui décrivent sous quelles conditions une fonction de l'algèbre relationnelle s'applique sur des n-uplets inintelligibles. Ces lois servent à définir une méthodologie d'optimisation d'une requête. Elle consiste à partir d'une requête sans spécificateurs (facile à écrire) et à appliquer les lois successivement pour dériver la requête optimisée. Ceci donne au développeur la garantie que sa requête manipule correctement les données. En outre, une transformation en ProVerif garantit que le choix des techniques de cryptographie protège les contraintes de confidentialités. Malgré tout, le langage C2QL reste un langage abstrait.

Ce chapitre propose une implémentation du langage, mais un obstacle vient avec cette implémentation : un développeur peut écrire des requêtes qui n'ont pas de sens. Rien ne force celui-ci à suivre la méthodologie avec les lois. S'ensuit un risque de mauvaise composition des fonctions C2QL qui amènent à des erreurs de programmation. Ces erreurs peuvent, au mieux, stopper l'exécution du programme. Mais, elles peuvent aussi provoquer un bogue qui révèle ou cède l'accès aux données personnelles. Ce chapitre montre qu'il y a quatre classes d'erreurs possibles en C2QL : erreurs de manipulation des n-uplets ; erreurs de spécification de l'environnement ; erreurs d'implémentation de l'environnement ; et erreurs de composition des requêtes.

On redonnera ici l'exemple du "buffer overflow" (Aleph 1, 1996) (cf. §2.3.1)

Par la suite, il est impératif que l'implémentation du langage protège de ces quatre classes d'erreurs de programmation. Dans un langage informatique, un bon outil pour détecter tôt certaines erreurs de programmation est un système de types. Pour citer un célèbre slogan : "*Well typed programs do not go wrong*" (Milner, 1978). C'est pourquoi l'implémentation du langage C2QL utilise un tel système. Les contributions de ce chapitre sont :

— Une implémentation du langage C2QL en tant que *langage dédié embarqué* (Embedded Domain-Specic Language – EDSL) dans Idris (Brady, 2013). Idris est un langage de programmation fonctionnel avec des types dépendants. L'avantage d'un EDSL est de pouvoir exploiter les fonctionnalités du langage hôte comme l'*analyse syntaxique* (parsing) et la génération de code (Hudak, 1998). L'avantage de prendre Idris comme langage hôte est de profiter de son système de types suffisamment expressifs pour prévenir des quatre classes d'erreurs. Ainsi, l'implémentation ne permet d'écrire que des termes C2QL corrects.

— Un terme C2QL produit par l'implémentation est une description d'une requête. Pour que le langage C2QL soit réellement implémenté, il faut écrire un compilateur de ce terme qui produit une application du nuage comme celle utilisée pour les expérimentations au chapitre 3 (*cf.* §3.2). Écrire ce compilateur est une tâche non triviale qui n'est pas faite dans

cette thèse par manque de temps. Toutefois, le code implémente la traduction vers un terme π-calcul, comme vu au chapitre 4 (*cf.* §4.4). Cette thèse considère que cette traduction suffit à montrer qu'un terme C2QL peut être compilé en une application du nuage pour être exécutée (à la manière du chapitre 3) et en un programme ProVerif pour vérifier les contraintes de confidentialité (à la manière chapitre 5).

Le reste de ce chapitre est organisé comme suite. La section 6.1 explicite les quatre classes d'erreurs au moyen de faux programmes écrit dans le langage C2QL. La section 6.2 présente le langage de programmation Idris et son système de types dépendants. La section 6.3 s'inspire d'un article de Oury et Swierstra (2008) pour montrer l'intérêt d'implémenter un langage dans un système à types dépendants. La section 6.4 montre comment implémenter le langage C2QL en Idris. Cette section montre également, par l'exemple, comment le système de type refuse les programmes qui appartiennent à l'une des quatre classes d'erreurs présentées à la section 6.1. La section 6.5 présente l'implémentation de la traduction vers le π-calcul. Enfin, la section 6.6 conclut sur l'implémentation de C2QL.

6.1 Les classes d'erreurs de composition en C2QL

Cette section présente quatre formes d'erreurs qu'un développeur peut faire en utilisant le langage C2QL. Ces erreurs ne sont pas relatives à une mauvaise protection des contraintes de confidentialités, comme vue au chapitre 5. Ces erreurs sont relatives à une mauvaise composition des fonctions de C2QL lorsque le développeur n'utilise pas les lois. Chaque forme d'erreur est illustrée par un faux programme écrit en C2QL. Plus tard, la section sur l'implémentation du langage C2QL (*cf.* §6.4) montrera comment exclure statiquement les programmes qui rentrent dans ces classes d'erreurs.

6.1.1 Erreur d'implémentation de l'environnement

Une erreur d'implémentation de l'environnement survient quand le développeur spécifie une composition de spécificateurs qui ne reflètent pas l'environnement visé.

Un exemple est le faux programme 32 qui décrit une requête `[adresse]`. Ce programme n'a pas présente pas un terme mal formé. Simplement, la description de l'environnement ne reflète pas l'environnement protégé qui est considéré dans cette thèse (*c.-à-d.*, chiffrement des noms et fragmentation sur les dates).

Code 32 – *Erreur d'implémentation de l'environnement protégé.*

$$\sigma_{nom\ \texttt{LIKE}\ `C*'} \circ \pi_{nom,adresse} \circ defrag_{nom} \circ frag_{nom}$$

6.1.2 Erreur de spécification de l'environnement

Une erreur de spécification de l'environnement survient quand l'utilisation des spécificateurs ou leur composition décrit un environnement qui n'a pas de sens. La cause est un non-respect des conditions d'utilisation d'un spécificateur qui spécifient comment introduire celui-ci (*cf.* §4.1.3) ou une mauvaise application des lois de composition qui spécifient comment les spécificateurs se composent (*fig.* 23).

Par exemple, la description du *crypt* spécifie que l'attribut chiffré appartient à la relation. Mais le développeur peut oublier cette description et écrire le faux programme 33 qui chiffre l'attribut *foo* alors que celui-ci est absent de la relation *rendezvous*.

$crypt_{\alpha,\texttt{c}}$ *avec* $\alpha \in \Delta$

$$crypt_{foo,\texttt{AES}} \quad rendezvous$$

Code 33 – *Erreur d'introduction du crypt :* $foo \notin rendezvous$.

De la même manière, les lois introduisent des conditions d'applications qui doivent être suivies par le développeur pour composer correctement les spécificateurs. Par exemple, le développeur utilise la loi de composition 23 lorsqu'il veut faire une fragmentation multiple. Cette loi stipule que la deuxième fragmentation porte sur des attributs présents dans la relation de droite. Sans cette loi, le développeur peut écrire le faux programme 34 qui fragmente la relation de droite sur les dates alors que celles-ci sont à gauche.

$defrag_{\delta}\ (id, defrag_{\delta'}(id, id) \circ frag_{\delta'}) \circ frag_{\delta}$ *si* $\delta' \subseteq (\Delta \setminus \delta)$

$$(id, frag_{date}) \circ frag_{date} \quad rendezvous$$

Code 34 – *Erreur de composition des frag :* $(date) \not\subseteq (rendezvous \setminus (date))$.

6.1.3 Erreur de manipulation des n-uplets

Une erreur de manipulation des n-uplets survient lorsque l'emploi des fonctions de l'algèbre relationnelle n'a pas de sens, soit parce que la fonction se compose mal avec le reste de l'application, soit parce qu'elle n'est pas utilisable dans l'environnement considéré. La cause est un non-respect des conditions d'utilisation de la fonction (*cf.* §2.4.3.3) ou une mauvaise application des lois de projection/sélection/agrégation (*fig.* 20, 21 et 22).

Par exemple, le développeur utilise la loi de projection 12 lorsqu'il veut faire une projection dans un fragment. Cette loi a comme condition que la projection dans le fragment de gauche porte sur les attributs de ce fragment. Mais sans cette loi, le développeur peut écrire la fausse version suivante de `[adresse]` (*code* 35) qui n'a pas de sens, car la projection sur les noms n'est pas faite sur le bon fragment.

$\pi_{\delta} \circ defrag_{\delta'} \equiv defrag_{\delta'}\ (\pi_{\delta \cap \delta'}, \pi_{\delta \setminus \delta'})$

$$\begin{aligned}\sigma_{nom\ \texttt{LIKE}\ \text{`}C*\text{'}} &\circ decrypt_{nom,\texttt{AES}} \\ &\circ defrag_{date}\ (\pi_{nom}, \pi_{adresse}) \\ &\circ frag_{date} \circ crypt_{nom,\texttt{AES}}\end{aligned}$$

Code 35 – *Requête* `[adresse]` *avec une projection sur les noms dans le mauvais fragment.*

De la même manière, le développeur utilise la loi de sélection 14 lorsqu'il veut faire une sélection sur de données chiffrées. Cette loi a comme condition que le prédicat de la sélection supporte les opérations sur une donnée chiffrée. Mais sans cette loi, le développeur peut écrire le faux programme 36 qui

$\sigma_{p_{\delta}} \circ decrypt_{\alpha,\texttt{c}} \equiv decrypt_{\alpha,\texttt{c}} \circ \sigma_{\texttt{c} \Rightarrow p_{\delta}}$ *si* $\alpha \in \delta$

applique la sélection d'égalité sur des noms chiffrés avec ElGamal, alors que ElGamal est un chiffrement probabiliste qui ne supporte pas le test d'égalité.

Code 36 – *Requête* `#rendezvous` *avec un test d'égalité sur des données inintelligibles qui ne supportent pas ce test.*

$$defrag_{date}\ (count_{date} \circ \pi_{date} \circ \sigma_{(date-aujourdhui)\in[0..7]},\ \pi_{\emptyset} \circ \sigma_{adresse=\text{`}Bureau\text{'} \wedge \texttt{ElGamal} \Rightarrow nom=\text{`}Bob\text{'}}) \circ frag_{date} \circ crypt_{nom,\texttt{ElGamal}}$$

6.1.4 Erreur de composition des requêtes

Une erreur de composition des requêtes survient lorsque le développeur choisit deux configurations de techniques de cryptographie différentes pour protéger deux requêtes d'une même application. Dans ce cas, ces deux requêtes ne peuvent pas coexister sur l'application. L'exemple suivant permet de s'en convaincre.

En imaginant que le développeur de l'agenda personnel choisisse la configuration $frag_{date} \circ crypt_{nom,\texttt{AES}}$ pour protéger la requête `#rendezvous`. Puis, qu'il opte pour la configuration $frag_{(date,\mathbf{nom})} \circ crypt_{nom,\texttt{AES}}$ pour protéger la requête `[adresse]`. Ceci obligerait le développeur à reconfigurer dynamiquement l'application en fonction de la requête qui est exécutée en faisant migrer la colonne des noms d'un fragment à l'autre. Évidemment, cette reconfiguration dynamique n'est pas envisageable à cause de son coup d'exécution. C'est pourquoi les configurations doivent être les mêmes, ou du moins compatibles, pour toutes les requêtes d'une application.

6.2 Le langage de programmation Idris et les types dépendants

Le langage de programmation Haskell (Peyton Jones, 2003)

Idris est un langage de programmation fonctionnel avec un système de types dépendants écrit par Brady (2013). Sa syntaxe est proche du langage Haskell, mais son système de types dépendants rend possible l'encodage précis de certains aspects du comportement d'un programme dans le type.

Un type dépendant est un type qui peut dépendre d'une valeur. Cet ajout, par rapport à un système de type classique comme en ML ou en Haskell, permet au développeur de définir des types beaucoup plus précis. De ce fait, les propriétés requises par une application peuvent être capturées par le système de type et vérifiées par le *vérificateur de type* (type checker). Ce qui augmente la confiance du développeur dans l'exactitude de son programme. La suite profite du classique exemple des listes indicées par leur taille pour présenter les types dépendants et la syntaxe du langage Idris.

Type algébrique (ADT). Idris a une syntaxe proche de celle d'Haskell. Définir un nouveau *type algébrique* (algebraic data type – ADT) se fait dans une syntaxe similaire à celle des GADT (Peyton Jones et collab., 2006). Par exemple, les entiers naturels de Peano sont définis de la manière suivante en Idris.

```
data Nat : Type where
  Z : Nat
  S : Nat → Nat
```

Le type `Nat` à deux *constructeurs de donnée* : `Z` pour zéro et `S` pour le successeur. Avec cette définition, le programmeur construit le nombre 1 comme le successeur de zéro (`S Z`), le nombre 2 comme le successeur de 1 (`S (S Z)`) et ainsi de suite. Dans cette définition, `Nat:Type` signifie que `Nat` est un type de base (sans dépendre d'une valeur). Par conséquent, Idris offre le raccourci suivant, plus direct.

```
data Nat = Z | S Nat
```

En outre, un développeur peut utiliser les entiers littéraux (*ex.*, 0, 1, 402) pour écrire les entiers naturels, ce qui simplifie l'écriture des grands nombres.

Fonction totale. Idris est un langage fonctionnel. Le développeur définit une nouvelle fonction par récursion et *reconnaissance de motif* (pattern matching). Par exemple, l'addition de deux entiers est définie comme suit.

```
total
plus : (n, m : Nat) → Nat
plus Z right        = right
plus (S left) right = S (plus left right)
```

La vérification est faite comme suggérée par Turner (1995)

Le mot clef `total` vérifie (statiquement) que la fonction est totale. C'est-à-dire que l'implémentation couvre tous les cas et que les appels récursifs se font sur une valeur plus petite structurellement. C'est indispensable si cette fonction vient à être utilisée dans la définition d'un type. En effet, dans ce cas, le vérificateur de type va devoir faire un calcul et une fonction non totale pourrait mener à une vérification indécidable.

Un fichier Idris qui contient la directive `%default total` vérifie que toutes les fonctions définies au sein de ce fichier sont totales. La suite considère que cette directive est toujours présente. Par conséquent, toutes les fonctions dans le suite sont totales, même si le mot clef `total` est omis.

Constructeur de type paramétré. La définition d'une liste affiche un *constructeur de type* plus intéressant que celui des entiers naturels. Là où l'ADT des entiers naturels est juste un type de base (`Nat:Type`), ici le constructeur du type `List` est paramétré par un type (`a:Type`) pour fournir une définition polymorphe.

```
data List : (a : Type) → Type where
  Nil  : List a
  (::) : a → List a → List a
```

Le constructeur de donnée `Nil` n'a pas de paramètre. Il produit une liste vide. Le constructeur de donnée `(::)` prend deux paramètres : un élément de type `a` et une liste d'éléments de type `a`. Il les combine pour produire une nouvelle liste d'éléments de type `a`. Les parenthèses autour du constructeur `(::)` spécifient d'utiliser celui-ci en position infixe.

Dans le programme suivant, le développeur utilise le *constructeur de type* `List` pour définir le type de la constante `l1` et les *constructeurs de donnée* `Nil` et `(::)` pour définir la valeur de `l1`.

```
l1 : List Nat
l1 = Z :: Z :: Nil
```

Une fonction qui travaille avec une liste polymorphe est aussi paramétrée par un type. Rien de plus logique puisque le constructeur de type de `List` prend un type en argument. Un exemple est la fonction polymorphe `append` ci-après qui concatène deux listes avec des éléments de même type. Le type d'`append` montre qu'elle prend un type `a` en argument pour construire le type de la liste `List a`.

```
append : (a : Type) → List a → List a → List a
append a Nil       right = right
append a (x :: xs) right = x :: (append a xs right)
```

Appliquer la fonction `append` demande d'instancier le paramètre `a`. C'est ce qui est fait dans l'exemple suivant où l'implémentation de la constante `l2` instancie `a` avec le type `Nat` pour appliquer la fonction `append`.

```
l2 : List Nat
l2 = append Nat l1 l1
```

Mais, instancier la variable de type est laborieux, surtout quand, comme ici, la valeur peut facilement être déduite du contexte (*c.-à-d.*, du type de `l1`). Pour éviter cela, Idris permet de définir certains arguments comme implicites en les entourant d'accolades.

```
append : {a : Type} → List a → List a → List a
append Nil       right = right
append (x :: xs) right = x :: (append xs right)
```

Dans ce cas, Idris instancie automatiquement ces arguments avec l'aide du contexte. Maintenant, l'implémentation de `l2` n'a plus à instancier le paramètre `a` avec le type `Nat`.

```
l2 : List Nat
l2 = append l1 l1
```

Type dépendant. Cette façon de paramétrer les fonctions par un type est similaire à ce qu'il est déjà possible de faire en Haskell avec le polymorphisme paramétré. Ou même en Java avec la généricité. Mais, un type dépendant peut dépendre d'une valeur. L'exemple classique est celui du type `Vect` qui représente une liste avec sa taille.

```
data Vect : Nat → (a : Type) → Type where
  Nil  : Vect Z a
  (::) : {n : Nat} → a → Vect n a → Vect (S n) a
```

Le constructeur de type de `Vect` ressemble à celui de `List`, mais il est paramétré par un entier naturel. Cet entier naturel est là pour spécifier la taille de la liste et les constructeurs de donnée implémentent cette spécification. Le constructeur `Nil` produit une liste vide et son type est une liste de taille zéro (`Z`). `(::)` combine un élément de type `a` avec une liste de `n` éléments de type `a`. Son type est une liste de taille `n` + 1 (soit, `S n`).

Le développeur n'a pas d'autre choix que d'utiliser les constructeurs de donnée de `Vect` pour construire une liste indicée par la taille. Or, ces constructeurs sont définis de sorte que leur type *reflète toujours le nombre correct d'éléments* dans la liste. Par conséquent, le développeur ne peut pas définir la constante `l3` comme la liste de taille 1 en la construisant avec `Nil`, car le type de `Nil` dé-

finit une liste de taille zéro. C'est pourquoi il est souvent dit que la programmation par types dépendants produit des *programmes corrects par construction* (correct-by-construction programs – Peter, 2002).

```
l3 : Vect (S Z) Nat
l3 = Nil -- ill-typed: Vect Z Nat ≠ Vect (S Z) Nat
```

Représenter la taille de la liste dans le type donne une définition précise de la fonction. Par exemple, le type du résultat de la fonction `append` est défini comme une liste dont la taille correspond à la taille de la première et de la seconde liste.

```
append : {a : Type} → {m, n : Nat} →
         Vect m a → Vect n a → Vect (plus m n) a
append Nil       ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

Ceci permet au développeur d'avoir plus confiance dans son implémentation. Par exemple, cette spécification refuse la mauvaise implémentation suivante, car, trivialement, la taille de `(x :: xs)` plus `ys` n'est pas équivalent à la taille de `ys`.

```
append (x :: xs) ys = ys -- ill-typed: n ≠ (plus (S m) n)
```

La suite montre comment un EDSL peut profiter des propriétés du langage hôte Idris, comme la vérification décidable des types ou leur réduction, pour capturer les besoins en vérification de l'EDSL.

6.3 Implémenter l'algèbre relationnelle en Idris

Le langage de programmation Agda (Stump, 2014)

L'article *The Power of Pi*, par Oury et Swierstra (2008), implémente plusieurs DSL en les embarquant dans Agda, un autre langage de programmation avec des types dépendants. Le dessein de l'article est de montrer qu'un langage avec des types dépendants est un hôte idéal pour embarquer un système de type dédié. Parmi les DSL proposés, l'un d'entre eux est l'algèbre relationnelle. La suite redonne l'implémentation de Oury et Swierstra et l'adapte à Idris. Cette implémentation sert de base pour l'implémentation du langage C2QL.

6.3.1 Typer avec le schéma relationnel

Comme spécifié à plusieurs reprises dans ce document, le type d'une relation dans l'algèbre relationnelle est défini par son schéma relationnel. Le chapitre 2 montre que le comportement d'une fonction de l'algèbre relationnelle peut, en partie, être spécifié par le schéma (*cf.* §2.4.3.3). De même, les lois de C2QL au chapitre 4 utilisent le schéma pour spécifier sous quelles conditions une fonction est applicable sur des n-uplets inintelligibles (*cf.* §4.2). Il est donc naturel de représenter le schéma comme un type pour vérifier statiquement de la bonne forme d'une requête C2QL.

Le schéma relationnel est fait d'une liste d'attributs qui spécifient l'arité, le nom et le type des n-uplets. Les implémentations concrètes de l'algèbre relationnelle, comme PostgreSQL et MySQL, offrent une collection dédiée et

1 Les types de données en PostgreSQL postgresql.org/docs/9.3/static/datatype.html
2 Les types de données en MySQL dev.mysql.com/doc/refman/5.7/en/data-types.html

finie de types pour les éléments d'un n-uplet[1, 2]. Représenter une collection de types avec un nombre fini d'habitants se fait par un univers dans la théorie des types.

```
1 data Ty = NAT | BOOL | TEXT | DATE
```

L'implémentation propose une collection de quatre habitants pour le type d'un élément (`Ty`). La collection n'est pas exhaustive comparée à ce que propose, par exemple, PostgreSQL[1]. Elle est volontairement simplifiée pour le bien de la présentation, mais ne représente pas une limitation.

Les types sont traduits en type Idris avec la fonction de décodage `interpTy`.

```
2 interpTy : Ty → Type
3 interpTy NAT  = Nat
4 interpTy BOOL = Bool
5 interpTy TEXT = String
6 interpTy DATE = String
```

C'est le type `Ty` et la fonction de décodage `interpTy` qui forment l'univers. La suite profite de cet univers pour définir ce qu'est un attribut et un schéma.

Le type attribut est une paire. Le premier élément de la paire spécifie sous quel nom un élément est identifié. Le second élément représente le type de l'élément. Enfin, le type schéma est une liste d'attribut.

```
7 Attribute: Type
8 Attribute = (String, Ty)
9
10 Schema : Type
11 Schema = List Attribute
```

Avec ces deux types, le développeur peut spécifier le type du schéma relationnel des rendez-vous. Par exemple, l'attribut `A` aux lignes 12 et 13 identifie les adresses dans le schéma $(date, nom, adresse)$. Il spécifie que les adresses sont identifiées par le nom `"Adresse"` et que leur type est du `TEXT`. La spécification est identique pour les dates (`D`) et les noms (`N`). Le schéma des rendez-vous est une liste des attributs `D`, `N` et `A`.

```
12 A : Attribute
13 A = ("Adresse", TEXT)
14
15 D : Attribute
16 D = ("Date", DATE)
17
18 N : Attribute
19 N = ("Nom", TEXT)
20
21 RendezVous : Schema
22 RendezVous = [D,N,A] -- sucre syntaxique pour (D::N::A::Nil)
```

Dans la suite, un schéma type les fonctions de l'algèbre relationnelle pour spécifier leur comportement sur les valeurs d'attributs. Ensuite, le vérificateur de types utilise cette information pour vérifier, statiquement, la composition correcte des fonctions de l'algèbre relationnelle.

```
data Query : Schema → Type where
  π : (δ: Schema) → Query Δ → Query δ
  σ : Pred Δ BOOL → Query Δ → Query Δ
  count : (δ : Schema) → Query Δ → {auto p : So (includes δ Δ)} →
          Query (δ ++ [("Count", NAT)])
  (×) : Query Δ → Query Δ′ → Query (Δ ++ Δ′)
  unit : Handler Δ →  Query Δ
```

Code 37 – *L'ADT* `Query` *pour l'EDSL de l'algèbre relationnelle.*

6.3.2 L'EDSL pour l'algèbre relationnelle : l'ADT `Query`

Le langage de l'algèbre relationnelle est implémenté en Idris par l'ADT `Query` (*code* 37). Cette ADT modélise la grammaire de l'algèbre relationnelle (*fig.* 7, *cf.* §2.4.3.3) pour générer l'*arbre syntaxique abstrait* (abstract syntax tree – AST) d'une requête. Chaque constructeur de donnée de l'ADT représente une fonction de l'algèbre relationnelle (*c.-à-d.*, π_δ, σ_{p_δ} …). Un constructeur est paramétré par ses arguments (*ex.*, δ pour la projection π) et par le reste de la requête `Query`. De ce fait, un constructeur sert à produire un nouveau nœud dans l'arbre syntaxique abstrait d'une requête.

L'idée de l'EDSL est de profiter des types dépendants pour limiter la construction des AST à ceux qui *font sens* et rejeter, statiquement, les AST des requêtes mal formées. Il est ensuite possible de générer une requête SQL à partir de cette AST pour interroger la base de données avec comme garantie que la requête est correcte sémantiquement. La suite détaille chacun des constructeurs de donnée de `Query` et montre comment les programmes mal formés sont rejetés par le vérificateur de types.

Typer avec le schéma relationnel. L'ADT `Query` (*l.* 23) est typé par un `Schema` de telle manière qu'une requête avec le type `Query` Δ représente une requête qui retourne une relation de type Δ. Par conséquent, le vérificateur de type connaît statiquement la forme de la relation retournée pour n'importe quelle requête `Query`.

```
23 data Query : Schema → Type where
```

Connaître statiquement le schéma de relation est très intéressant. Ceci permet de vérifier statiquement qu'une fonction de l'algèbre relationnelle se compose correctement avec une requête pour l'étendre. L'objectif étant de couvrir, en partie, la première classe d'erreur qui porte sur une mauvaise manipulation des n-uplets en refusant les mauvaises compositions des fonctions de l'algèbre relationnelle.

Concrètement, le système de type vérifie si composer une fonction de l'algèbre relationnelle avec une requête existante est possible. Pour ce faire, il s'assure que les conditions d'application de la fonction sur la requête sont respectés. Les conditions d'applications sont celles données par Ullman (1982) qui s'expriment sur le schéma de la relation (*cf.* §2.4.3.3).

La projection. La spécification de la projection dit qu'appliquer la fonction π_δ sur la relation R de type Δ (avec $\delta \subseteq \Delta$) produit une nouvelle relation R' de type δ (*cf.* §2.4.3.3). Une première proposition pour le constructeur de donnée de π est la suivante.

```
π : (δ: Schema) → Query Δ → Query δ
```

Ici, le constructeur π prend deux arguments. Le premier est le schéma qui définit la projection. Le second est une requête qui retourne une relation de type Δ avec laquelle π se compose. Le résultat est typé comme une requête qui retourne des δ.

La définition est donc très proche de la spécification, mais les types dépendants peuvent faire encore mieux en garantissant, statiquement, que la contrainte $\delta \subseteq \Delta$ est vraie.

```
24  π : (δ: Schema) → Query Δ → {auto p: So (includes δ Δ)} → Query δ
```

L'argument `p` agit comme une preuve de $\delta \subseteq \Delta$. Autrement dit, un développeur a le droit d'utiliser le constructeur π que s'il a de quoi construire la preuve $\delta \subseteq \Delta$.

So.idr[3] :

```
data So : Bool → Type
  where
  Oh : So True
```

Le type de la preuve `p` est `So (includes δ Δ)` et le seul constructeur de donnée pour `So` est `Oh`. Par conséquent, le développeur n'a pas d'autres choix que de fixer la valeur de `p` à `Oh`. Or, le type de `Oh` est `So True`. De ce fait, le vérificateur de type évalue `includes δ Δ` pour l'unifier avec `True` et le programme est refusé par le vérificateur de type quand l'évaluation d'`includes δ Δ` ne se réduit pas à `True`. Ceci implique, tout de même, que les valeurs de δ et Δ soient connues statiquement par le vérificateur de type.

Si les valeurs ne sont pas connues, le développeur doit se débrouiller. Il peut, par exemple, encapsuler son calcul dans la monade `Either` et utiliser dynamiquement la fonction `choose` pour récupérer la preuve que δ est inclus dans Δ, ou gérer l'erreur le cas contraire. Un exemple d'utilisation du `choose` est donné en annexe (*cf.* §A.2.1).

```
choose : (b : Bool) → Either (So b) (So (not b))
```

Grâce à ce genre de construction, la programmation avec les types dépendants n'est pas limitée à un environnement clos. Seulement, le développeur doit gérer explicitement les cas d'erreurs.

Enfin, il est bon de noter que dans la pratique, le développeur ne passe pas explicitement le constructeur de donnée `Oh` à l'argument `p`, puisque `p` est implicite et que le mot clef `auto` appelle une tactique qui passe automatiquement le `Oh`.

La sélection. La spécification de la sélection dit qu'appliquer la fonction σ_{p_δ} sur la relation R de type Δ produit une nouvelle relation R' de type Δ (*cf.* §2.4.3.3).

```
25  σ : Pred Δ BOOL → Query Δ → Query Δ
```

Le constructeur `σ` produit un terme `Query` qui respecte cette spécification. Il est paramétré par un prédicat `Pred` (*code* 38) qui est lui même un ADT. Il suit la grammaire d'une clause `WHERE` dans la syntaxe SQL[4]. Comme pour l'univers de type, l'ADT des prédicats n'est pas exhaustif. Seuls les prédicats nécessaires à l'agenda personnel sont donnés ici pour plus de clarté.

Un prédicat intéressant est l'opérateur `(==)` qui définit un test d'égalité entre la valeur d'un attribut et un argument, par exemple `N == "Bob"`. L'attribut est spécifié par la variable `a` et la valeur par la variable `v`. Cette définition apporte deux garanties.

3 github.com/idris-lang/Idris-dev/blob/59577e6c1df4587e2a44e46bee4041dea63d0927/libs/-base/Data/So.idr#L15

4 postgresql.org/docs/current/static/queries-table-expressions.html#QUERIES-WHERE

```
data Pred : Schema → Ty → Type where
  (&&)     : Pred Δ BOOL → Pred Δ BOOL → Pred Δ BOOL
  (||)     : Pred Δ BOOL → Pred Δ BOOL → Pred Δ BOOL
  (==)     : (a : Attribute) → (v : interpTy (snd a)) →
             {auto p : Elem a Δ} →
             Pred Δ BOOL
  like     : (a : Attribute) → String →
             {auto p1 : TEXT = (snd a)} → {auto p2 : Elem a Δ} →
             Pred Δ BOOL
  nextWeek : (a : Attribute) →
             {auto p1 : DATE = (snd a)} → {auto p2 : Elem a Δ} →
             Pred Δ BOOL
```

Code 38 – *L'ADT des prédicats pour l'EDSL de l'algèbre relationnelle.*

Premièrement, la preuve `p` oblige l'attribut `a` à être un élément de Δ. Ainsi, le développeur peut écrire le prédicat `N == "Bob"` que si la relation Δ contient un attribut `("Nom", TEXT)`. Sinon, le vérificateur de types indique que le terme est mal formé.

Deuxièmement, la variable `v` est typée comme `interpTy (snd a)` plutôt que `snd a`, soit le type Idris associé au type de l'attribut `a` (*cf.* l.5). Grâce à cela, le développeur fournit, pour `v`, une valeur d'Idris, par exemple `"Bob" : String`, plutôt que d'encapsuler la valeur dans un constructeur de donnée, par exemple `TEXT("Bob")` qui a le type `TEXT`. Cette construction peut être vue comme une *coercition automatique et sure* entre les types de SQL et les types d'Idris. La coercition est dite sure, car le développeur ne peut pas écrire le prédicat `N == (S Z)`, par exemple. La fonction `interpTy` ne fournit pas de coercition entre le type `TEXT` de SQL et le type `Nat` d'Idris (*l.* 2 à 6).

Les autres prédicats sont de mêmes natures. À noter que le prédicat `like` qui teste une expression régulière n'a du sens que pour les chaînes de caractères. Ceci est garanti par la preuve `p1`. Idem, pour le prédicat `nextWeek` qui n'a du sens que pour les dates.

L'agrégation par dénombrement. La spécification de l'agrégation par dénombrement dit qu'appliquer la fonction $count_\delta$ sur une relation de type Δ groupe les n-uplets sur les attributs de δ et compte le nombre de n-uplets pour former une nouvelle relation de type $\delta +\!\!+ (count)$ (*cf.* §2.4.3.3). Ainsi, la définition de `count` est similaire à celle de π. La différence se fait sur le type de la relation produite qui contient en plus la colonne "Count" de type `NAT`.

```
26  count : (δ : Schema) → Query Δ → {auto p : So (includes δ Δ)} →
27          Query (δ ++ [("Count", NAT)])
```

Le produit cartésien et la jointure naturelle. La spécification du produit cartésien dit qu'appliquer la fonction $(\times)$, sur respectivement deux relations de type Δ et Δ', couple tous les n-uplets pour former une nouvelle relation de type $\Delta +\!\!+ \Delta'$ (*cf.* §2.4.3.3).

```
28  (×) : Query Δ → Query Δ' → Query (Δ ++ Δ')
```

Encore une fois, il est triviale que l'implémentation du produit cartésien suit cette spécification.

5 github.com/idris-lang/Idris-dev/blob/caefd9add609af7b242c8280bfaf9721b8eefb80/libs/prelude/Prelude/List.idr#L584

La spécification de la jointure naturelle dit qu'appliquer la fonction $(\bowtie)$ sur deux relations de type, respectivement, Δ et Δ', couple les n-uplets de la première relation avec les m-uplets de la seconde sur les attributs communs, pour former une nouvelle relation de type $nub(\Delta +\!\!+ \Delta')$.

Pour rappel, la fonction nub[5] supprime les doublons dans une liste.

```
njoin : Query Δ → Query Δ′ → {auto p : So (hasAny Δ Δ′)} →
        Query (nub (Δ ++ Δ′))
```

Dans cette implémentation, la preuve `p` garantie que les deux relations ont, au moins, un attribut en commun.

La connexion avec le monde réel. L'ADT `Query` contient maintenant toutes les fonctions de l'algèbre relationnelle. De plus, typer une `Query` sur le schéma relationnel fait que le vérificateur de type n'accepte, statiquement, que les requêtes bien composées. Mais, l'ADT n'a pas de point d'entrée. Tous les constructeurs de données se composent avec une requête existante. Ce point d'entrée est le constructeur `unit`.

```
unit : Handler Δ → Query Δ
```

Le constructeur `unit` prend un `Handler` avec le schéma Δ en paramètre et produit une requête avec le même schéma.

La donnée de type `Handler` contient toutes les informations pour se connecter à la base de données et accéder à la relation de type Δ. Par conséquent, lorsque le développeur définit une requête et produit un AST, cet AST contient forcément un nœud `unit` avec les informations pour requêter la base de données. Reste à savoir comment construire cet `Handler`.

Un `Handler` Δ ne doit être construit que si la relation Δ existe sur la base de données. À cette condition, la requête pourra s'exécuter avec la garantie statique qu'elle est correcte. Mais, savoir si la relation Δ existe dans la base de données requiert d'interroger celle-ci. En Haskell, la manière idiomatique de traiter ce genre d'effet de bord est d'utiliser la monade `IO`. Le principe est le même en Idris.

```
connect : (dbUrl : String) → (relName : String) → (Δ : Schema) →
          IO (Either DBError (Handler Δ))
```

La fonction `connect` se connecte à la base de données à l'adresse `dbUrl` et interroge celle-ci pour vérifier que la relation `relName` existe et que le schéma de cette relation est le même que celui spécifié par Δ. La fonction retourne une action `IO` qui contient un `Handler` Δ ou une erreur. Le risque d'erreur est dû au fait d'être dans un environnement ouvert qui peut se comporter de manière imprévue (*ex.*, la base de données est indisponible). Toutefois, c'est la seule défaillance du système de type. Notamment, si la connexion se fait, alors le système de type garantit que toutes les requêtes qui utilisent ce `Handler` manipulent correctement les n-uplets.

Un exemple de programme principal est donné en annexe (*cf.* §A.2.2). Ce programme montre comment la fonction `connect` est utilisée pour s'interfacer avec le monde extérieur et récupérer un `Handler`. Puis, comment à l'aide d'une fonction tierce les requêtes sont traduites en requête SQL pour interroger la base de données et retourner les résultats dans le programme.

Les requêtes relationnelles `[adresse]` et `#rendezvous` avec l'EDSL

L'ADT `Query` représente des requêtes SQL sans protections ce qui est équivalent à une requête locale en C2QL. Pour rappel, la requête `[adresse]` sur une machine locale s'écrit de la manière suivante avec l'algèbre relationnelle.

$$\sigma_{nom \texttt{ LIKE } \text{`}C*\text{'}} \circ \pi_{nom,adresse} \quad rendezvous$$

La traduction dans l'EDSL Idris est presque automatique tant les deux définitions sont proches. Et ce, même si l'EDSL est difficile à implémenter à cause des informations de type. Cette difficulté est invisible pour le développeur qui utilise l'EDSL, ce qui montre l'intérêt de celui-ci.

```
qadr : Handler RendezVous → Query [N,A]
qadr rendezvous =
     σ (N `like` "C*") $ π [N,A] $ unit rendezvous      (ra-qaddr)
```

Code 39 – *Requête* `[adresse]` *avec l'ADT* `Query` *– En Idris, comme en Haskell, mettre des accents graves (backquotes `) autour d'une fonction binaire permet d'utiliser celle-ci comme un opérateur infixe.*

Le seul ajout que le développeur doit faire est de spécifier, dans le type, le schéma en entrée et en sortie. Pour la requête `[adresse]`, le schéma en entrée est `RendezVous` précédemment défini aux lignes 12 à 22. Le schéma en sortie est `[N,A]`, car la requête retourne la liste des contacts et adresses visités par Alice.

Le fait de spécifier, dans le type, le schéma en entrée et en sortie s'apparente à une approche *descendante* (top-down). Dans cette approche, le type, facile à écrire, déclare les intentions du développeur. Ici, avoir une requête sur les rendez-vous qui produit une liste de contacts et d'adresses. Ensuite, le développeur peut écrire la requête et le vérificateur de type garantit qu'elle est conforme à ses intentions.

Dans l'implémentation, inclure un attribut `Foo` dans la projection (*c.-à-d.*, `π [Foo,N,A]`) produit une erreur à la vérification des types, car `Foo` n'est pas un membre du schéma `RendezVous`. Idem, si le développeur essaye de comparer les noms avec un entier naturel (`N `like` Z`). Ou encore, si le développeur essaye de faire une agrégation sur les dates après la projection alors que les dates ne sont plus disponibles. Par conséquent, il est impossible de produire un terme `Query` qui ne représente pas une manipulation correcte des n-uplets d'une relation donnée.

La traduction de la requête `#rendezvous` qui compte le nombre de rendez-vous pris avec Bob au bureau la semaine prochaine est tout aussi facile. Et encore une fois, le système de type garantit que cette requête manipule correctement les n-uplets de la relation des rendez-vous.

$$count_{date} \circ \pi_{date} \circ \sigma_{\substack{(date-aujourdhui)\in[0..7] \\ \wedge adresse=\text{`}Bureau\text{'} \wedge nom=\text{`}Bob\text{'}}} \quad rendezvous$$

```
qrdv : Handler RendezVous → Query [D,("Count", NAT)]
qrdv rendezvous =
     count [D] $ π [D] $
                 σ (nextWeek D
                    && A == "Bureau"
                    && N == "Bob") $ unit rendezvous
```

Code 40 – *Requête* `#rendezvous` *avec l'ADT* `Query`.

6 Les sources sont disponibles à l'adresse

Les sources de l'EDSL ainsi que les exemples sont disponibles sur le répertoire Git de cette thèse[6]. Le lecteur peut s'amuser à modifier les requêtes pour voir comment elles sont refusées par le vérificateur de type. Une bribe de programme montre comment exécuter une requête quand le schéma relationnel n'est pas connu statiquement (comme dans l'annexe A.2.1). Enfin, les sources montrent également comment récupérer un objet `Handler` et comment récupérer le résultat de l'exécution d'une requête (comme dans l'annexe A.2.2).

Cette implémentation évite les erreurs de manipulation des n-uplets dues à une mauvaise composition des fonctions de l'algèbre relationnelle. Mais, cette implémentation n'évite pas les erreurs de manipulation dues à une mauvaise composition avec les techniques de cryptographie. De même, cette implémentation n'évite pas les erreurs qui surviennent dans les trois autres classes (*cf.* §6.1).

La suite montre comment s'inspirer de cette implémentation pour implémenter l'EDSL de C2QL et éviter les erreurs de toutes les classes.

6.4 Implémenter le langage C2QL en Idris

Le langage C2QL se base sur l'algèbre relationnelle et ajoute deux types de fonction qui sont les spécificateurs et destructeurs de techniques de cryptographie (*cf.* §4.1.3). Un spécificateur spécifie comment une relation est protégée sur son environnement d'exécution. Tandis qu'un destructeur défait une technique de cryptographie pour rendre lisible les n-uplets précédemment inintelligibles. Par conséquent, une façon naïve d'implémenter le langage C2QL consiste à ajouter les spécificateurs et destructeurs à l'ADT `Query`. Le type de la requête est donc le même que pour `Query` et les vérifications du système de type s'étendent par définition aux compositions des fonctions de l'algèbre relationnelle avec les spécificateurs/destructeurs. L'avantage ici est de couvrir entièrement la première classe d'erreur, là où la couverture n'était que partielle avec l'ADT `Query`.

Mais cette implémentation est naïve, car le type ne donne pas la forme de l'environnement. Or, il est important de connaître statiquement l'environnement pour vérifier que deux requêtes d'une même application s'exécutent sur un environnement équivalent (*cf.* §6.1.4). Par conséquent, l'implémentation ne peut pas se contenter d'étendre l'ADT `Query`. La suite montre comment typer avec l'environnement, puis comment mettre l'environnement dans le type de l'implémentation de C2QL pour refuser les faux programmes qui appartiennent aux quatre classes d'erreurs de la section 6.1.

6.4.1 Typer avec l'environnement

L'implémentation du langage C2QL type une requête sur son schéma et son environnement d'exécution. L'objectif est de garantir statiquement que toutes les requêtes qui composent une application PbD s'exécutent sur le même environnement. Pour ce faire, il faut introduire un nouveau type `Env` qui spécifie comment est protégée une relation par rapport à son environnement d'exécution. L'implémentation (l.35) montre que c'est un alias pour un vecteur de schémas.

```
34 Env : Nat → Type
35 Env n = Vect (S n) Schema
```

Le vecteur décrit comment la relation est fragmentée. Un environnement de taille `0` a un vecteur de taille `1`, ce qui signifie que la relation est contenue dans une seule base de données ; un environnement de taille `1` a un vecteur de taille `2`, ce qui signifie que la relation est fragmentée verticalement en deux ; et ainsi de suite. Il est impossible de construire un environnement avec un vecteur de taille `0`. Par conséquent, un environnement contient toujours au moins une relation sur une base. Les éléments du vecteur sont des `Schema`. Ils indiquent le type de la relation pour le fragment en question.

Par exemple, l'environnement où la relation des rendez-vous est fragmentée sur les dates est spécifié par la constante `SafeRendezVousEnv`.

```
Id : Attribute
Id = ("Id", NAT)

SafeRendezVousEnv : Env 1
SafeRendezVousEnv = [[D,Id], [N,A,Id]]
```

L'environnement est de taille `1`, car il y a deux fragments. Les constantes `D`, `N` et `A` sont les attributs pour les dates, noms et adresses (l.12-19). La constante `Id` désigne l'attribut qui contient les identifiants pour défragmenter les fragments.

Dans cet exemple, il manque le chiffrement des noms pour que l'environnement `SafeRendezVousEnv` soit vraiment protégé comme décrit dans l'agenda personnel en ligne avec composition des techniques de cryptographie. Pour se faire, il faut introduire un nouvel univers de types pour le chiffrement.

```
36 data CryptTy = AES -- | RSA | ElGamal | ...
37
38 interpCryptTy : CryptTy → Type → Type
39 interpCryptTy AES t = Stream Bits8
```

Cet univers décrit quelles sont les techniques de chiffrement offertes par le langage. Seul le chiffrement AES est requis pour l'agenda personnel, mais rien n'empêche d'ajouter d'autres techniques comme RSA (1978), ElGamal (1985), Paillier (1999), *etc*.

La suite consiste à ajouter un nouvel habitant `CRYPT` à la famille de types `Ty` utilisé précédemment pour typer les attributs (*cf.* §6.3, *l.* 1). Le dessein est d'indiquer quelles sont les valeurs d'attributs qui sont chiffrées dans une relation. Pour ce faire, le type `CRYPT` est paramétré par deux informations. La première est la technique de chiffrement qui spécifie comment les valeurs sont chiffrées dans la relation (`CryptTy`). La seconde est le type des valeurs chiffrées (`Ty`).

```
data Ty = NAT | BOOL | ... | CRYPT CryptTy Ty

interpTy : Ty → Type
interpTy NAT         = Nat
...
interpTy (CRYPT c t) = interpCryptTy c (interpTy t)
```

Désormais, il est possible de définir un nouvel attribut `N_AES` qui spécifie que les noms des contacts sont chiffrés avec un chiffrement AES.

```
40 N_AES : Attribute
41 N_AES = ("Nom", CRYPT AES TEXT)
```

L'environnement `SafeRendezVousEnv` peut ainsi être représentatif de l'environnement d'exécution de l'agenda personnel en ligne protégé par chiffrement symétrique et fragmentation verticale.

```
SafeRendezVousEnv : Env 1
SafeRendezVousEnv = [[D,Id], [N_AES,A,Id]]
```

La suite intègre l'environnement dans l'implémentation du langage C2QL en Idris et montre comment cette implémentation refuse les fausses requêtes qui rentrent dans l'une des quatre classes d'erreurs présentées à la section 6.1.

6.4.2 L'EDSL pour C2QL : l'ADT `Privy`

Le langage C2QL est implémenté en Idris par l'ADT `Privy` (*code* 41). `Privy` est indexé par un schéma relationnel qui spécifie la forme des n-uplets retournés par la requête, comme pour l'ADT `Query`. Il est en plus indexé par le type `Env` qui spécifie sur quel environnement la requête est exécutable.

L'ADT `Privy` modélise avec une forme un peu particulière la grammaire de C2QL (*cf.* §4.1.3, *fig.* 18) pour générer l'arbre syntaxique abstrait (AST) d'une requête protégée par composition des techniques de cryptographie. Cette forme, nommée forme monadique, est la manière idiomatique de manipuler un environnement dans un langage fonctionnel pur. Elle permet de définir les constructeurs de donnée `crypt` et `frag` qui manipulent l'environnement sans avoir à passer explicitement l'environnement à tous les constructeurs de donnée.

La raison est que la gestion d'un état, comme l'environnement, est toujours explicite dans un langage fonctionnel pur puisqu'il n'y a pas d'effets de bord. Par conséquent, il n'est pas possible de modifier l'état en place. L'état doit être passé et retourné par les fonctions qui utilisent et modifient l'état. Pour `Privy`, ceci signifie de passer l'environnement courant comme argument à tous les constructeurs de donnée et récupérer l'environnement en sortie en plus de l'ADT `Privy`. Mais, passer systématiquement l'environnement est pénible et répétitif. Ça complexifie inutilement l'écriture et la lecture d'une requête. Heureusement, Wadler (1992) a montré que ce genre de construction

github.com/rcherrueau/C2QL/tree/master/composition-checker/RelAlgebra.idr.

Code 41 – *L'ADT* `Privy` *pour l'EDSL de C2QL.*

```
data Privy : Env n → Env m → Schema → Type where
  crypt  : (a : Attribute) → (c : CryptTy) →
           {auto p : So (a `isInEnv` env)} →
           Privy env (cryptEnv c a env) ⊥
  frag   : (δ : Schema) → {auto p : So (includes δ (last env))} →
           Privy env (fragEnv δ env) ⊥
  query  : (fId: Fin (S n)) → (q: Query (index fId env) → Query δ) →
           Privy env env δ
  (>>=)  : Privy env0 env1 δ → (Query δ → Privy env1 env2 δ') →
           Privy env0 env2 δ'
  return : Query δ → Privy env env δ
```

peut être structuré grâce à une forme monadique. La forme monadique requiert de définir deux fonctions supplémentaires, le *bind* (`>>=`) et le `return`. Grâce à ça, le développeur n'a pas à passer l'environnement courant en argument des constructeurs. Le passage est géré par la monade, ce qui simplifie l'écriture d'un terme `Privy`.

Le programme 42 montre comment implémenter la requête `[adresse]` qui suit l'approche du *nuage confidentiel* avec l'ADT `Privy`. Même si de prime à bord ce programme peut sembler très différent de son équivalent écrit dans le langage C2QL, il n'en est rien. La suite de cette section donne une interprétation intuitive de ce programme.

Code 42 – *Requête* `[adresse]` *avec l'ADT* `Privy`.

```
42 RendezVousEnv : Env 0
43 RendezVousEnv = [[D,N,A]]
44
45 SafeRendezVousEnv : Env 1
46 SafeRendezVousEnv = [[D,Id], [N_AES,A,Id]]
47
48 qadr : Privy RendezVousEnv SafeRendezVousEnv [N,A]
49 qadr =
50    -- description de l'environnement :
51    crypt N AES                                  >>= \_ =>
52    frag [D]                                     >>= \_ =>
53    -- description de la requête :
54    query 0 (\f0 => π [ ] f0)                    >>= \r0 =>
55    query 1 (\f1 => π [N,A] f1)                  >>= \r1 =>
56    return ( σ (N `like` "C*") $ decrypt N (aes "the-key")
57                               $ defrag r0 r1 )
```

Interprétation intuitive d'une requête `Privy`

Le programme 42 implémente la requête `[adresse]` avec composition des techniques de cryptographie. Dans ce programme, l'environnement `RendezVousEnv` (l.42) modélise l'environnement de la relation des rendez-vous sans protection. L'environnement `SafeRendezVousEnv` (l.45) modélise celui avec chiffrement des noms et fragmentation sur les dates.

Interpréter le type d'un terme `Privy`. Le type de la requête `qadr` (l.48) stipule qu'elle produit un terme `Privy`. Les trois arguments dans le constructeur de type de `Privy` définissent une spécification/un plan. Cette spécification indique comment le terme doit être implémenté avec deux faits :

1. Les deux premiers arguments (`RendezVousEnv` et `SafeRendezVousEnv`) spécifient que le terme doit décrire quelles sont les techniques de cryptographie à employer et comment elles se composent pour passer d'un environnement `RendezVousEnv` où la relation des rendez-vous n'est pas protégée, à un environnement `SafeRendezVousEnv` où les noms sont chiffrés par un chiffrement AES et où la relation est fragmentée sur les dates.
2. Le troisième argument (`[N,A]`) spécifie que le terme doit décrire comment obtenir des n-uplets avec le schéma relationnel `[N,A]`, sur l'environnement `SafeRendezVousEnv`.

Dans l'implémentation le développeur utilise les constructeurs de données de l'ADT `Privy` (*code* 41) pour produire un terme. Ce terme décrit comment atteindre les deux faits imposés par le type. Ainsi, le terme est correct par construction par rapport à la spécification du développeur.

Interpréter l'implémentation d'un terme `Privy`. L'implémentation de la requête `qadr` (l.51 à 57) est découpée en deux parties. La première partie, lignes 51 et 52, décrit comment atteindre le premier fait. La seconde, ligne 55 et 57, décrit comment atteindre le deuxième fait.

La première partie compose les spécificateurs du langage C2QL pour décrire comment protéger l'environnement. Elle se lit : ligne 51, sur l'environnement courant (*c.-à-d.*, la relation des rendez-vous sans protection), spécifie que les éléments de l'attribut `N` sont chiffrés avec le chiffrement symétrique déterministe AES ; ligne 52, sur l'environnement courant, spécifie que la relation est fragmentée sur l'attribut `D`

La deuxième partie compose les termes de l'ADT `Query` pour décrire une requête qui manipule des n-uplets sur l'environnement courant. Elle se lit : ligne 54, construit un terme `Query` qui fait la projection sur les identifiants dans le premier fragment, puis affecte ce terme à la variable `r0` ; ligne 55, construit un terme `Query` qui fait la projection sur les noms et les adresses dans le deuxième fragment, puis affecte ce terme à la variable `r1` ; lignes 56 et 57, retourne le terme `Query` qui défragmente `r0` et `r1`, puis déchiffre les noms et enfin, filtre pour avoir ceux qui commencent par la lettre C.

La notation `do`. Malgré la notation monadique, une requête `Privy` est très proche de sa définition dans le langage abstrait C2QL. D'ailleurs, Idris offre, avec la notation `do`, du sucre syntaxique qui embellit l'écriture et la lecture de la requête `qadr`.

```
qadr : Privy RendezVousEnv SafeRendezVousEnv [N,A]
qadr = do
   -- description de l'environnement :
   crypt N AES
   frag [D]
   -- description de la requête :
   r0 ← query 0 (\f0 => π [ ] f0)
   r1 ← query 1 (\f1 => π [N,A] f1)
   return ( σ (N `like` "C*") $ decrypt N (aes "the-key")
                              $ defrag r0 r1 )
```

Cette notation donne un aspect impératif au langage C2QL. L'environnement est un état implicite. Appliquer un spécificateur accède et modifie cet état. Calculer une requête accède à l'état pour connaître la forme de la relation. Le *bind* (←) est équivalent à une affectation. Enfin, le `return` fait office de `return!` Dans la suite, la notation `do` sera préférée.

Le schéma $\perp$ (bottom). Dans le type d'un terme `Privy`, le schéma relationnel (dernier argument) spécifie la forme des n-uplets calculés par la requête sur l'environnement courant. Mais certaines requêtes `Privy` ne calculent pas de n-uplets. C'est le cas d'une requête seulement composée de spécificateurs, comme la requête `qSafeEnv` qui décrit comment avoir un environnement protégé.

```
qSafeEnv : Privy RendezVousEnv SafeRendezVousEnv ⊥
```

```
qSafeEnv = do
   -- description de l'environnement :
   crypt N AES
   frag [D]
```

Dans ces requêtes, la valeur du schéma relationnel est $\perp$. Une requête typée avec le schéma $\perp$ est une requête qui ne renvoie rien à la manière du `unit` en ML ou `()` en Haskell.

Les sections suivantes détaillent les constructeurs de donnée de `Privy` et montrent comment le vérificateur de types exclut les programmes des quatre classes d'erreurs de la section 6.1.

6.4.3 Exclure les erreurs d'implémentation de l'environnement

cf. §6.1.1

> *"Une erreur d'implémentation de l'environnement survient quand le développeur spécifie des spécificateurs qui ne reflètent pas l'environnement visé."*

L'ADT `Privy` est indexé par deux environnements. Le premier (`env0`) indique la forme de l'environnement avant application d'un spécificateur. Le second (`env1`) indique la forme de l'environnement après application du spécificateur.

```
58 data Privy : (env0 : Env n) → (env1 : Env m) → Schema → Type where
```

Cette façon de paramétrer une forme monadique par un état de départ et un état d'arrivée vient d'Atkey (2009). Ici, elle spécifie que le terme doit fournir une composition de techniques de cryptographie pour passer de l'environnement `env0` à `env1`, sans pour autant spécifier comment. L'objectif dans l'ADT `Privy` est de proposer au développeur une approche *descendante* (top-down) pour empêcher l'écriture de faux programmes qui rentrent dans la *classe des erreurs d'implémentation de l'environnement*.

Dans cette approche, le développeur spécifie d'abord le type de l'environnement de départ et d'arrivée. Ceci est très simple pour lui. La plupart du temps, le type de l'environnement de départ est trivial. Il correspond à la relation sans protection (*ex.*, `[[D,N,A]]`). Quant au type de l'environnement d'arrivée, c'est celui dicté par les contraintes de confidentialités (*ex.*, `[[D,Id], [N_AES,A,Id]]`). La définition est simple, par conséquent, le développeur ne fait que rarement des erreurs ici.

La seule erreur possible dans le type est un environnement qui ne protège pas les contraires de confidentialités, mais ceci est vérifié par la traduction en ProVerif.

Le développeur utilise ainsi les types des environnements pour déclarer simplement ses intentions. Ensuite, le développeur peut s'aider de ces types pour trouver une composition de spécificateurs qui dérive l'environnement d'arrivée à partir de l'environnement de départ. Il doit alors se soucier des détails d'implémentation des techniques de cryptographie (*ex.*, le chiffrement AES se fait avec une clef qui doit être spécifiée, organiser la fragmentation multiple pour que l'opération s'applique sur la relation la plus à droite). Cette composition est difficile à écrire d'où les risques d'erreurs d'implémentation de l'environnement. Mais grâce à l'approche descendante et le principe d'implémentation *correct par construction*, la composition de spécificateurs implé-

mente obligatoirement l'environnement visé. Par conséquent, le développeur ne peut pas écrire de faux programmes qui rentrent dans la *classe des erreurs d'implémentation de l'environnement* (*cf.* §6.1.1).

6.4.4 Exclure les erreurs de spécification de l'environnement

cf. §6.1.2

> *"Une erreur de spécification de l'environnement survient quand l'utilisation des spécificateurs ou leur composition décrit un environnement qui n'a pas de sens. La cause est un non-respect des conditions d'utilisation d'un spécificateur qui spécifient comment introduire celui-ci ou une mauvaise application des lois de composition qui spécifient comment les spécificateurs se composent."*

Les spécificateurs du langage C2QL sont définis dans l'ADT `Privy` comme des constructeurs de donnée qui modifient l'environnement. Les constructeurs sont paramétrés pour empêcher une mauvaise utilisation d'un spécificateur et ainsi exclure statiquement les programmes de la *classe des erreurs de spécification de l'environnement*. En particulier, ces paramètres vérifient l'emploi correct du spécificateur comme défini au chapitre 4. Soit le respect des conditions d'utilisation d'un spécificateur (*cf.* §4.1.3) et une composition correcte de ces spécificateurs (*fig.* 23).

Le spécificateur de chiffrement. En C2QL, le spécificateur de chiffrement $crypt_{\alpha,\mathtt{c}}$ spécifie que sur une relation de type Δ, les valeurs de l'attribut α (avec $\alpha \in \Delta$) sont chiffrées avec le schéma de chiffrement `c` (*cf.* §4.1.3). La fonction $crypt_{a,\mathtt{c}}$ est traduite par le constructeur de donnée `crypt` dans l'ADT `Privy`. Le constructeur produit un nouveau terme de type `Privy` et modifie l'environnement courant pour spécifier que les éléments de l'attribut `a` sont chiffrés avec la technique de chiffrement `c`. La modification de l'environnement dans le type se fait par la fonction `replaceInEnv` qui change l'occurrence de `a` par son équivalent chiffré par `c`.

```
59 crypt : (a : Attribute) → (c : CryptTy) →
60         {auto p : So (a `isInEnv` env)} →
61         Privy env (replaceInEnv a (fst a, CRYPT c (snd a)) env) ⊥
```

Par exemple, le terme `specCryptN` spécifie que les noms sont protégés par le chiffrement AES dans la relation des rendez-vous .

Pour rappel, `RendezVousEnv=[[D,N,A]]` `N=("Nom",TEXT)` *et* `N_AES=("Nom",CRYPT AES TEXT)`

```
specCryptN : Privy RendezVousEnv [[D,N_AES,A]] ⊥
specCryptN = crypt N AES
```

La preuve `p` garantit la condition $\alpha \in \Delta$ en vérifiant que l'attribut `a` est présent dans un des fragments de l'environnement. Par conséquent, le faux programme 33 (*cf.* §6.1.2) est refusé par le vérificateur de type, car il est impossible de construire la preuve `p` que l'attribut "`Foo`" existe dans l'environnement.

$$crypt_{foo,\mathtt{AES}} \quad rendezvous$$

```
falseSpecCrypt : Privy RendezVousEnv _ ⊥ -- ill-typed:
falseSpecCrypt = crypt ("Foo",TEXT) AES   -- "Foo" ∉ RendezVousEnv
```

Le spécificateur de fragmentation verticale. En C2QL, le spécificateur de fragmentation $frag_\delta$ spécifie qu'une relation de type Δ est séparée en deux, de telle manière que le premier fragment contienne les valeurs des attributs spécifiés par δ (avec $\delta \subseteq \Delta$) et que le deuxième fragment contienne les valeurs des attributs restants (*cf.* §4.1.3). La loi 23 (*cf.* §4.2.6) généralise également la fragmentation à plus de deux fragments en spécifiant que les fragmentations se font toujours dans le fragment le plus à droite. La fonction $frag_\delta$ est traduite par le constructeur de donnée `frag` dans l'ADT `Privy`. Le constructeur produit un nouveau terme de type `Privy` et modifie l'environnement courant pour y ajouter un fragment. L'ajout du nouveau fragment dans l'environnement se fait par la fonction `fragEnv` qui fragmente le fragment le plus à droite et produit un nouvel environnement de taille $n + 1$. La définition de la fonction `fragEnv` est donnée en annexe (*cf.* §A.2.3).

$defrag_\delta\ (id, defrag_{\delta'}(id, id) \circ frag_{\delta'}) \circ frag_\delta$ *si* $\delta' \subseteq (\Delta \setminus \delta)$

```
62 frag : (δ : Schema) → {auto p : So (includes δ (last env))} →
63        Privy env (fragEnv δ env) ⊥
64 replace (last env)
```

Par exemple, le terme `specFragD` spécifie de fragmenter sur les dates à partir de la relation des rendez-vous où les noms sont protégés.

```
specFragD : Privy [[D,N_AES,A]] SafeRendezVousEnv ⊥
specFragD = frag [D]
```

La séparation en deux du dernier fragment se fait par rapport à δ. La preuve `p` assure que δ est inclus dans le dernier fragment. Par conséquent, le faux programme 34 (*cf.* §6.1.2) est refusé par le vérificateur de type, car il est impossible à la deuxième fragmentation de construire la preuve `p` que la relation `[D]` est incluse dans la relation `[N,A,Id]`.

$$(id, frag_{date}) \circ frag_{date} \quad rendezvous$$

```
falseSpecFragDA : Privy SafeRendezVousEnv _ ⊥ -- ill-typed:
falseSpecFragDA = frag [D]                    -- [D] ⊈ [N_AES,A]
```

Composer les spécificateurs. Composer les spécifications `specCryptN` et `specFragD` décrit comment passer de l'environnement `RendezVousEnv` à `SafeRendezVousEnv` à la manière de la requête C2QL suivante.

$$frag_{date} \circ crypt_{nom,\texttt{AES}} \quad rendezvous$$

```
specSafeRendezVous : Privy RendezVousEnv SafeRendezVousEnv ⊥
specSafeRendezVous = do specCryptN; specFragD
```

Ainsi, `specSafeRendezVous` implémente l'environnement `SafeRendezVousEnv` par composition des spécificateurs. Le vérificateur de type utilise le type de l'environnement courant pour garantir le respect des conditions d'application des spécificateurs et leur composition. Grâce à ça, il est impossible de composer un spécificateur avec une requête où il ne peut pas s'appliquer. Il est donc impossible de produire un environnement qui n'a pas de sens. Par conséquent, le développeur ne peut pas écrire de faux programmes qui rentrent dans la *classe des erreurs de spécification de l'environnement* (*cf.* §6.1.2).

6.4.5 Exclure les erreurs de manipulation des n-uplets

cf. §6.1.3

> *"Une erreur de manipulation des n-uplets survient lorsque l'emploi des fonctions de l'algèbre relationnelle n'a pas de sens, soit parce que la fonction se compose mal avec le reste de l'application, soit parce qu'elle n'est pas utilisable dans l'environnement considéré. Cette fois, la cause est un non-respect des conditions d'utilisation de la fonction ou une mauvaise application des lois de projection/sélection/agrégation."*

Un terme `Privy` est fait de deux parties. Une partie description de l'environnement et une partie description de la requête pour manipuler les n-uplets. Jusqu'à présent, seule la partie description de l'environnement a été vue. La suite montre comment décrire une requête et comment il est impossible de décrire une requête de la classe d'erreur de manipulation.

Le constructeur de donnée `query`. Dans l'ADT `Privy`, le constructeur de donnée `query` donne accès à une relation pour faire une requête relationnelle.

```
65 query  : {env : Env n} →
66          (fId : Fin (S n)) → (q : Query (index fId env) → Query δ) →
67          Privy env env δ
```

Par exemple, le terme suivant décrit une projection sur les dates dans le premier fragment de l'environnement `SafeRendezVousEnv`.

```
qDate : Privy SafeRendezVousEnv SafeRendezVousEnv [D]
qDate = query 0 (\q => π [D] q)
```

Le premier argument du constructeur `query` (`fId`) est un index pour cibler le fragment dans l'environnement. L'index `0` accède au fragment le plus à gauche ; l'index `1` au fragment suivant ; et ainsi de suite jusqu'au fragment le plus à droite avec l'index `n`. Le type de l'index (`Fin`) assure de ne pas pouvoir fournir un index plus grand que le nombre de fragments dans l'environnement. Ainsi, la requête suivante est refusée par le vérificateur de type, car le développeur ne peut pas définir de requête sur le 3^{e} fragment alors que l'environnement `SafeRendezVousEnv` n'en possède que deux.

```
qFalseFragAccess : Privy SafeRendezVousEnv SafeRendezVousEnv _
qFalseFragAccess = query 2 id -- ill-typed: 2 ≯ 2
```

Le deuxième argument du constructeur `query` (`q`) est la requête relationnelle. Le développeur décrit la requête grâce à l'ADT `Query` vu à la section précédente (*cf.* §6.3.2, *code* 37). La requête est définie sur le schéma `(index fId env)`, c'est-à-dire le schéma du fragment numéro `fId`. Cette requête décrit un calcul qui produit un schéma δ. C'est ce schéma qui sert à typer `Privy`.

Requêter sur les fragments. L'avantage de réutiliser l'ADT `Query` est de profiter du système de type mis en place précédemment. Pour rappel, ce système de type garantit qu'une fonction de l'algèbre relationnelle est applicable sur la relation courante en se basant sur son schéma. Transposé dans l'ADT `Privy`, ce système garantit qu'une fonction de l'algèbre relationnelle est applicable sur la relation du fragment numéro `fId` de l'environnement. Cette garantie est la même que celles offertes par les lois qui impliquent un *defrag*.

En effet, la partie optimisation des lois (partie droite d'une équivalence) transforme la fonction de l'algèbre relationnelle pour garantir que cette fonction soit applicable sur la relation du fragment.

Par conséquent, le faux programme 35 (*cf.* §6.1.3) est refusé par le vérificateur de type, car les noms ne sont pas présents dans le premier fragment.

$$defrag_{\,date}\ (\pi_{nom}, \pi_{adresse}) \circ frag_{\,date} \circ crypt_{nom,\texttt{AES}}$$

```
qFalseΠ : Privy SafeRendezVousEnv SafeRendezVousEnv _
qFalseΠ = query 0 (\q => π [N] q) --ill-typed : [N] ⊈ [D,Id]
```

Deux optimisations qui impliquent un *defrag* ne sont pas vérifiées par le système de type. Ce sont celles décrites par les équations 20 et 21 (*cf.* §4.2.5, *fig.* 22). Elles ne sont pas vérifiées, car l'information du schéma ne suffit pas. Il faut en plus regarder comment est formé le terme. La manière de les vérifier statiquement est expliquée un peu plus tard.

$count_\delta \circ defrag_{\delta'}\ (id, \pi_\emptyset) \overset{20}{\equiv} defrag_{\delta'}\ (count_\delta, \pi_\emptyset)$ *si* $\delta \subseteq \delta'$

$count_\delta \circ defrag_{\delta'}\ (\pi_\emptyset, id) \overset{21}{\equiv} defrag_{\delta'}\ (\pi_\emptyset, count_\delta)$ *si* $\delta \cap \delta' = \emptyset$

Requêter avec le chiffrement. Actuellement, le système de type de l'ADT `Query` n'assure pas la contrainte imposée par un chiffrement $crypt_{a,\texttt{c}}$ lors d'une sélection ou d'une agrégation par dénombrement (*cf.*, *eq.* 14 et *eq.* 18, *fig.* 21 et 22). Les lois spécifient qu'une sélection (respectivement, une agrégation) s'applique sur des n-uplets chiffrés à condition de pouvoir réaliser le test du prédicat (respectivement, l'égalité) sur ces n-uplets chiffrés.

$\sigma_{p_\delta} \circ decrypt_{\alpha,\texttt{c}} \overset{14}{\equiv} decrypt_{\alpha,\texttt{c}} \circ \sigma_{\texttt{c} \Rightarrow p_\delta}$ *si* $\alpha \in \delta$

De ce fait, tester l'égalité sur des noms chiffrés avec AES fait sens.

$$decrypt_{nom,\texttt{AES}} \circ \sigma_{AES \Rightarrow nom = \text{`}gTR7Y0kmcUCCK8f97EaLBQ==\text{'}} \circ crypt_{nom,\texttt{AES}}$$

Le test fait sens, car AES est un chiffrement symétrique déterministe. Par conséquent, il vérifie la propriété d'égalité entre deux valeurs chiffrées. À l'inverse, le même test avec un chiffrement probabiliste (*ex.*, ElGamal 1985) n'a pas de sens. Il faut donc interdire ce terme dans le langage `Privy`.

$count_\delta \circ decrypt_{\alpha,\texttt{c}} \overset{18}{\equiv} decrypt_{\alpha,\texttt{c}} \circ count_{\texttt{c} \Rightarrow \delta}$ *si* $\alpha \in \delta$

Pour le moment, le système de type dans L'ADT des prédicats (*code* 38) est trop général sur le test d'égalité (`==`). La seule contrainte pour la valeur donnée en argument (`v`) est qu'elle soit du même type que les valeurs de l'attribut.

```
(==) : (a : Attribute) → (v : interpTy (snd a)) →
       {auto p : Elem a Δ} → Pred Δ
```

Par conséquent, si le développeur donne en argument une donnée chiffrée de manière probabiliste avec ElGamal et que les valeurs de l'attribut nom sont chiffrées avec ElGamal, alors le terme est valide ce qui est faux. Il faut une notion comme : *le prédicat est possible que si les valeurs de l'attribut* (`a`) *supportent le test d'égalité*.

Cette notion se nomme le polymorphisme *ad-hoc*. Le polymorphisme *ad-hoc* concerne une fonction polymorphe qui se comporte différemment en fonction du type de ses arguments. Le langage Idris, comme Haskell, fait du polymorphisme *ad-hoc* avec une *classe-type* (type class – Peyton Jones et collab., 1997). Une fonction polymorphe devient *ad-hoc* en ajoutant une contrainte sur le type des arguments de la fonction. Un argument doit alors décrire comment se comporter par rapport à cette contrainte pour être passé à la fonction.

L'implémentation fournit la classe-type `Equal` qui ajoute une contrainte pour un type `Ty` (le type des attributs, l.1). Cette contrainte exige que les valeurs du type `Ty` supportent le test d'égalité. Le test d'égalité (`==`) dans l'ADT des prédicats est redéfini pour prendre en compte cette contrainte.

Pour les connaisseurs d'Haskell, la classe `Equal` *ne doit pas être confondu avec* `Eq`.

```
--     classe-type ⇒ définition précédente de (==)
(==) : Equal (snd a) ⇒ (a : Attribute) → (v : interpTy (snd a)) →
       {auto p : Elem a Δ} → Pred Δ BOOL
```

Maintenant le développeur peut construire un terme (==) que si le type de l'attribut (a) décrit qu'il satisfait la contrainte `Equal`. Par exemple, l'implémentation de `Equal` pour le type `CRYPT` décrit que le test d'égalité ne fait sens que pour le chiffrement `AES`.

```
implementation Equal (CRYPT AES _)
```

Quantifier universellement sur tous les chiffrements `(CRYPT c _)` serait une erreur. Seuls les chiffrements déterministes vérifient le test d'égalité.

Par conséquent, le faux programme 36 (*cf.* §6.1.3) est refusé par le vérificateur de type, car il n'existe pas d'implémentation d'`Equal` pour le chiffrement ElGamal.

$$\sigma_{\texttt{ElGamal} \Rightarrow nom = \text{`}Bob\text{'}} \circ crypt_{nom,\texttt{ElGamal}} \quad rendezvous$$

```
qFalseΣ : Privy [[D,N,A]] [[D,N_ElGamal,A]] _
qFalseΣ = do
  crypt ElGamal N
  --ill-typed: Can't find implementation for Equal (CRYPT ElGamal _)
  query 0 (\q => σ (N == "ZjbBkJEmjB6vDawjv53iRA==") q)
```

Alors qu'un terme qui fait une sélection avec un test sur les noms chiffrés par AES est un terme valide.

```
qGoodΣ : Privy [[D,N,A]] [[D,N_AES,A]] _
qGoodΣ = do
  crypt AES N
  query 0 (\q => σ (N == "gTR7Y0kmcUCCK8f97EaLBQ==") q)
```

Le code donne également des implémentations de la contrainte `Equal` pour tous les types de base. Ainsi, la contrainte `Equal (snd a)` est satisfaite lorsque le développeur passe, par exemple, un attribut de type `TEXT` au constructeur (==).

```
implementation Equal NAT
implementation Equal BOOL
implementation Equal TEXT
implementation Equal DATE
```

La fonction de défragmentation. Le destructeur de défragmentation joint deux fragments et reconstitue les n-uplets d'origine grâce à leur identifiant. Après lui, les identifiants sont supprimés de la relation.

Le destructeur de défragmentation est ajouté à l'ADT `Query` de la manière suivante.

$$count_\delta \circ defrag_{\delta'}\ (id, \pi_\emptyset) \overset{20}{\equiv} defrag_{\delta'}\ (count_\delta, \pi_\emptyset) \quad si\ \delta \subseteq \delta'$$

```
defrag : (q1 : Query δ) → (q2 : Query δ') →
         {auto p : So (countLaw q1 q2)} →
         Query (delete Id (nub (δ ++ δ')))
```

$$count_\delta \circ defrag_{\delta'}\ (\pi_\emptyset, id) \overset{21}{\equiv} defrag_{\delta'}\ (\pi_\emptyset, count_\delta) \quad si\ \delta \cap \delta' = \emptyset$$

La preuve `p` vérifie les équations 20 et 21. Pour rappel (*cf.* §4.2.5), ces deux équations stipulent que pour que le *count* rentre dans un fragment, le second fragment doit faire une projection qui supprime tous les attributs (sauf les identifiants). C'est une façon simple de garantir que l'application SaaS ne

viole pas les contraintes de confidentialités en cas d'évaluation séquentielle des requêtes dans les fragments. Avec ceci, le système de type est entièrement cohérent avec les optimisations des lois de C2QL. Toutefois, cette preuve n'est pas nécessaire. Le chapitre 5 a montré que les contraintes de confidentialités sont vérifiées par l'encodage en ProVerif.

La fonction de déchiffrement. Le destructeur de chiffrement $decrypt_{a,c}$ déchiffre les valeurs de l'attribut a qui était chiffrées avec le chiffrement `c`. Il est ajouté à l'ADT `Query` de la manière suivante.

```
79 decrypt : (a : Attribute) → Decrypt c → Query Δ →
80           {auto p1: (CRYPT c t) = (snd a)} → {auto p2: Elem a Δ} →
81           Query (replaceOn a (fst a, t) Δ)
```

La preuve `p1` garantit que le déchiffrement s'applique sur un attribut `a` dont les valeurs de type `t` sont chiffrées avec le chiffrement `c`. La preuve `p2` garantit que cet attribut est présent dans la relation courante de type Δ. Le résultat est une nouvelle requête qui produit une relation où les attributs de `a` ne sont plus chiffrés.

Le deuxième argument du constructeur de données `decrypt` est de type `Decrypt c`. Une valeur de type `Decrypt c` est un terme qui contient les informations pour détruire un chiffrement `c`. Pour les besoins de cette thèse, l'ADT `Decrypt c` propose un unique constructeur qui est celui du chiffrement AES. Il prend la clef de chiffrement en argument.

```
82 data Decrypt : CryptTy → Type where
83   aes : Key → Decrypt AES
```

Avec ce constructeur, le développeur stipule la clef de déchiffrement lorsqu'il déchiffre les noms dans la requête `[adresse]`.

```
qadr : Key → Privy SafeRendezVousEnv SafeRendezVousEnv [N,A]
qadr theKey = do
   ...
   return ( σ (N `like` "C*") $ decrypt N (aes theKey) $ ... )
```

L'intérêt de passer par un terme est que tous les destructeurs de chiffrement ne se paramètrent pas de la même manière. Les expérimentations au chapitre 3 donnent un exemple concret. Pour rappel, les expérimentations utilisent un chiffrement AES-CBC, car c'est le seul offert par l'API WebCrypto utilisée côté client. Or, le chiffrement AES-CBC est initialisé avec un vecteur d'initialisation (d'une taille de 16 octets) et ce vecteur est requis au moment du déchiffrement. Ceci peut être représenté dans le langage `Privy` en ajoutant le constructeur qui suit à l'ADT `Derypt`.

```
  aescbc : Key → Vect 16 Bits8 → Decrypt AES
```

Ainsi, le vérificateur de type utilise le type de l'environnement courant et le type de l'ADT `Query` pour garantir le respect des conditions d'application des fonctions de l'algèbre relationnelle et des destructeurs. Grâce à ça, il est impossible de produire une requête qui n'a pas de sens. Par conséquent, le développeur ne peut pas écrire de faux programmes qui rentrent dans la *classe des erreurs de manipulation des n-uplets* (*cf.* §6.1.3).

6.4.6 Exclure les erreurs de composition des requêtes

cf. §6.1.4

> *"Une erreur de composition des requêtes survient lorsque le développeur choisit deux configurations de techniques de cryptographie différentes pour protéger deux requêtes d'une même application. Dans ce cas, ces deux requêtes ne peuvent pas coexister sur l'application."*

Les requêtes d'une même application peuvent être composées grâce à l'ADT `App` (*code* 43) qui spécifie qu'une application PbD est faite d'un ensemble de requêtes `Privy`.

Code 43 – *ADT* `App` *pour composer les requêtes* `Privy` *et définir une application PbD.*

```
84 data App : (env : Env n) → Type where
85   Nil  : App env
86   (::) : Privy _ env _ → App env → App env
```

Avec cette ADT, l'agenda personnel qui suit l'approche du *nuage confidentiel* est défini comme la composition des requêtes `[adresse]` et `#rendezvous` protégées par composition des techniques de cryptographie. Le code 42 a déjà implémenté la requête `[adresse]` en `Privy`. Le code ci-après implémente la requête `#rendezvous` et compose ces requêtes pour former l'application PbD agenda.

```
84 qrdv : Privy SafeRendezVousEnv SafeRendezVousEnv [D,Cnt]
85 qrdv = do
86   r0 ← query 0 (\f0 => count [D] $ π [D] $ σ (nextWeek D) f0)
87   r1 ← query 1 (\f1 => π [ ] $ σ (A == "Bureau" &&
88                                   N == "gTR7Y0kmcUCCK8f97EaLBQ==")
       f1)
89   return ( defrag r0 r1 )
90
91 agenda : App SafeRendezVousEnv
92 agenda = [qadr, qrdv]
```

L'ADT `App` profite de la présence du type de l'environnement dans un terme `Privy` pour garantir que toutes les requêtes s'appliquent sur le même environnement nuagique. Grâce à ça, il est impossible de produire une application dont les requêtes s'exécutent sur des environnements différents. Par conséquent, le développeur ne peut pas écrire de faux programmes qui rentrent dans la *classe des erreurs de composition des requêtes* (*cf.* §6.1.4).

6.5 Traduction en π-calcul

La programmation d'une requête avec l'ADT `Privy` produit un arbre syntaxique abstrait (AST) qui décrit la requête. Mais l'implémentation n'est pas parfaite. Le *desiderata* doit produire deux applications. Une qui décrit les interactions des acteurs en ProVerif pour vérifier que les contraintes de confidentialité sont respectées, comme dans le chapitre 5. Et une qui produit du code concret (par exemple JavaScript) pour s'exécuter sur le nuage comme dans le chapitre 3.

Écrire un compilateur du *desiderata* est une tâche longue qui n'a pas pu être explorée dans cette thèse par manque de temps. La suite présente néanmoins

la traduction d'un programme `Privy` vers un terme π-calcul, comme vu au chapitre 4 (*cf.* §4.4, *fig.* 25). Cette thèse considère que cette traduction suffit à montrer qu'un terme `Privy` peut être compilé en un programme ProVerif ou une application du nuage.

L'idée est de réutiliser la traduction donnée à la fin du chapitre 4. Pour ce faire, la traduction se décompose en deux transformations. Une première transformation prend un terme `Privy` et le traduit vers un terme C2QL. Une deuxième transformation implémente la traduction du chapitre 4 (*fig.* 29) pour obtenir le programme en π-calcul.

Il n'est pas très intéressant de détailler ces transformations. L'implémentation de la première transformation (*c.-à-d.*, `Privy` vers C2QL) est disponible en annexe A.2.5. Une particularité de cette transformation est qu'elle supprime les types dans le terme C2QL, puisque ces types deviennent inutiles une fois que le programme a été vérifié. L'implémentation de la seconde transformation (*c.-à-d.*, C2QL vers π-calcul) est disponible en annexe A.2.6. Cette transformation suit la traduction du chapitre 4 (*fig.* 29). Ces deux traductions sont également disponibles dans le répertoire Git de cette thèse[7] , [8]. Le lecteur est encouragé à exécuter la fonction principale qui traduit le terme `[adresse]` qui suit l'approche du *nuage confidentiel* (*cf. code* 42) vers son équivalent en π-calcul. Le résultat de cette transformation est montré dans le code 44.

```
( vapp )( vfrag_0 )( vfrag_1 )( vclient )
  (!app(url).[url=adresse]
    frag_0<url, client>.0|frag_1<url, client>.0)
| (!( ρr_0 : (Date,Id) )frag_0<url, k>.[url=adresse]
    let r_1 = ( π [ ] r_0 ) in k<r_1>.0)
| (!( ρr_0 : (AES Nom,Adresse,Id) )frag_1<url, k>.[url=adresse]
    let r_1 = ( π [AES Nom,Adresse] r_0 ) in k<r_1>.0)
| app<adresse>.client(r_1).client(r_2).
    let r_3 = ( defrag r_2 r_1 ) in
    let r_4 = ( decrypt Nom r_3 ) in
    let r_5 = ( σ (Nom like C*) r_4 ) in 0
```

Code 44 – *Sortie de la traduction du programme* `Privy [adresse]` *en π-calcul.*

Le lecteur peut voir que ce programme est identique à la traduction donnée dans le chapitre 4 (*cf.* §4.4.2) et remise ici.

$$
\begin{aligned}
\texttt{[adresse]} \equiv\ & (\nu app)(\nu db_0)(\nu db_1)(\nu client) \\
& !Agenda\ |!Frag_g\ |!Frag_d \mid Alice \\
Agenda \equiv\ & app(url).[url = \texttt{"[adresse]"}] \\
& (\overline{db_0}\langle url, client\rangle.0 \mid \overline{db_1}\langle url, client\rangle.0) \\
Frag_g \equiv\ & (\rho rdv_g : (date, id))\, db_0(url, k).[url = \texttt{"[adresse]"}] \\
& let\ s = \pi_\emptyset\ rdv_g\ in\ \overline{k}\langle s\rangle.0
\end{aligned}
$$

7 Les sources de la traduction d'un programme `Privy` vers un programme C2QL sont disponibles à l'adresse github.com/rcherrueau/C2QL/tree/master/composition-checker/C2QL.idr.

8 Les sources de la traduction d'un programme C2QL vers un programme π-calcul sont disponibles à l'adresse github.com/rcherrueau/C2QL/tree/master/composition-checker/Pi.idr.

$$
\begin{aligned}
Frag_d \equiv\ & (\rho rdv_d : (\texttt{AES}\ nom, adresse, id)) \\
& db_1(url, k).[url = \texttt{"[adresse]"}] \\
& let\ s = \pi_{nom,adresse}\ rdv_d\ in\ \overline{k}\langle s\rangle.0 \\
Alice \equiv\ & \overline{app}\langle\texttt{"[adresse]"}\rangle.client(r_1).client(r_2). \\
& let\ s = defrag_{date}\ r_1\ r_2\ in \\
& let\ t = decrypt_{nom,\texttt{AES}}\ s\ in \\
& let\ u = \sigma_{nom\ \texttt{LIKE}\ \text{`}C*\text{'}}\ t\ in\ 0
\end{aligned}
$$

6.6 Conclusion

Ce chapitre montre comment implémenter le langage C2QL en Idris. Le langage est implémenté avec un ADT, nommé `Privy`. L'ADT profite du système de types dépendants d'Idris pour assurer que seul un terme correct puisse être écrit. Ainsi, le développeur peut directement écrire une requête dans sa version optimisée, sans avoir à suivre la méthodologie qui se base sur les lois.

Les avantages de l'implémentation en tant qu'EDSL dans Idris sont multiples. Premièrement, le langage profite de l'*analyse syntaxique* (parsing) d'Idris en représentant la grammaire de C2QL par l'ADT `Privy`. Chaque terme `Privy` est un arbre syntaxique abstrait (AST). Par conséquent, ceci évite d'avoir à écrire un analyseur syntaxique pour le langage. Deuxièmement, le langage profite du système de types dépendants d'Idris pour implémenter le système de type de C2QL. Ceci permet d'avoir un système de type qui est, de fait, décidable. Et évite d'avoir à implémenter ce système de type.

Implémenter le système de type dans l'ADT `Privy` complexifie l'implémentation du langage C2QL. La lecture d'un constructeur de donnée n'est pas toujours très claire à cause des types dépendants et des preuves. Cependant, cette complexité est invisible pour le développeur qui utilise le langage. En outre, les avantages sont certains puisque le développeur ne peut pas écrire de faux programmes qui rentrent dans l'une des quatre classes d'erreurs de la section 6.1.

Enfin, l'implémentation propose une traduction d'une requête `Privy` en un terme π-calcul. Ceci montre que le langage est approprié pour représenter la manipulation de données dans le nuage. Ceci montre également que le langage peut être traduit en un terme ProVerif pour vérifier que les techniques de cryptographie employées assurent les contraintes de confidentialités.

Les sources de l'implémentation sont disponibles sur le répertoire Git de cette thèse[9]. Le lecteur peut récupérer les sources et les compiler avec Idris en version 11.02. Dans les sources, les programmes laissés en commentaire sont des programmes qui sont refusés par le système de type. Le lecteur peut s'amuser à supprimer les commentaires pour voir comment ces programmes sont refusés par le vérificateur de type.

9 Les sources de l'implémentation sont disponibles à l'adresse github.com/rcherrueau/C2QL/tree/master/composition-checker.

Conclusion

7

L'informatique en nuage est une réalité. Les services offerts par celui-ci sont tellement avantageux pour les développeurs et les utilisateurs, qu'en quelques années les applications qui l'utilisent sont devenues légions. Le monde d'aujourd'hui est un monde où nous sommes hyperconnectés, notre téléphone portable, notre ordinateur et bientôt tous les objets de nos foyers seront reliés au nuage. Cette hyperconnectivité est même devenue un besoin, puisque nous nous reposons sur les moteurs de recherche, les sites Web et autres services en ligne pour certaines tâches de notre quotidien. La contrepartie est évidente, chaque jour nous confions des plus en plus de nos données personnelles et chaque jour nous perdons un peu plus le contrôle dessus. Et il serait naïf de croire qu'une loi suffira à arbitrer un système aussi complexe que le nuage.

C'est dans ce contexte que se pose cette thèse. Le défi étant de préserver les données personnelles des utilisateurs dans les applications du nuage. Le chapitre 2 montre que la préservation des données personnelles dans le nuage doit se faire par un système sécurisé. La sécurisation s'obtient en implémentant trois piliers : politique (pour la spécification), mécanisme (pour l'implémentation) et assurance (pour la confiance). Le pilier de la politique peut être instancié de manière générale lorsqu'il s'agit de la préservation des données personnelles pour les applications du nuage. Cette politique est la suivante : "L'obligation pour n'importe lequel des agents SaaS, PaaS et IaaS de non collecter, non traiter, non diffuser et non interférer sur les données personnelles de la cliente". Cette thèse fait le choix d'appliquer cette politique au moyen de l'approche de protection intégrée de la vie privée (PbD). Dans cette approche, le développeur d'application nuagique, bien veillant, souhaite produire une application qui préserve les données personnelles de ces utilisateurs conformément à la politique précédente. Pour ce faire, il emploie lors du développement de son application des techniques de cryptographie qui préservent la confidentialité des données personnelles en les rendant inintelligibles.

Le chapitre 3 montre qu'il est impératif de composer les techniques de cryptographie pour développer une application substantielle. La démonstration est faite par une implémentation concrète sur le nuage d'un agenda personnel en ligne. L'implémentation montre que la composition permet d'atteindre les trois critères d'une bonne application PbD, à savoir : un respect de la confidentialité, une exploitation des avantages du nuage et des performances en temps d'exécution acceptables. Cette thèse désigne l'effort pour atteindre ces trois critères par le nom d'"approche du *nuage confidentiel*" et montre que cet effort se fait au détriment de la simplicité. Un constat tragique pour le pilier de l'assurance qui requiert une application correcte. La suite propose donc un nouveau langage pour l'écriture correcte d'applications du nuage qui suivent l'approche du *nuage confidentiel*.

Le langage se nomme C2QL (*Cryptographic Compositions for Query Language*) et est présenté au chapitre 4. C2QL étend l'algèbre relationnelle avec trois techniques de cryptographie qui sont le chiffrement, la fragmentation verticale et les calculs côté client. Dans ce langage, un développeur programme des applications du nuage en composant les techniques de cryptographie avec les fonctions de l'algèbre relationnelle. Des lois algébriques expriment comment une fonction de l'algèbre relationnelle commute avec les techniques de cryptographie pour s'appliquer sur des données protégées. Cette commutation respecte deux propriétés. La première est l'équivalence observationnelle qui préserve la sémantique des fonctions de l'algèbre relationnelle lorsqu'elles sont appliquées sur des données protégées. La seconde est l'équivalence de confidentialité qui préserve les protections introduites par une technique de cryptographie. Ceci permet d'élaborer une méthodologie pour dériver, systématiquement, une application locale sans protection vers son équivalent qui suit l'approche du *nuage confidentiel*. Cette méthodologie se veut très simple. Elle consiste à "pousser" les destructeurs de cryptographie le plus tard possible dans la requête pour qu'un maximum de calculs se fasse sur des données inintelligibles et donc sur le nuage.

C2QL est également un langage *sans-tiers*, c'est-à-dire que les acteurs du nuage et leurs interactions n'apparaissent pas dans une requête. Ceci fournit un langage orienté sur la manipulation des données. Une orientation qui aide le développeur à raisonner sur l'optimisation de sa requête puisqu'il lui suffit de regarder la position des destructeurs pour prédire la quantité d'information calculée dans le nuage. Une traduction automatique vers le langage π-calcul montre comment obtenir les acteurs et leurs interactions à partir d'une requête C2QL. La traduction évalue tous les destructeurs de cryptographie chez la cliente pour assurer qu'aucune donnée confidentielle ne transite dans le nuage.

Cette assurance n'est toutefois pas suffisante pour garantir la correction d'une application écrite en C2QL, car le langage ne vérifie pas les contraintes de confidentialités de l'application. Le chapitre 5 remédie à ce manquement et vérifie automatiquement la préservation des contraintes de confidentialités avec l'outil ProVerif. Pour ce faire, ce chapitre repose sur la traduction en π-calcul et sur une modélisation innovant d'une relation par son schéma relationnel. Cette modélisation abstrait les n-uplets. Par conséquent, la vérification faite par ProVerif vaut pour un nombre illimité d'exécutions avec n'importe quelles valeurs de n-uplets concrets.

Enfin, le chapitre 6 propose une implémentation du langage C2QL en Idris. Cette implémentation profite des types dépendants d'Idris pour garantir l'écriture correcte d'une requête C2QL au développeur qui ne voudrait pas suivre les lois. L'implémentation en Idris refuse toutes les fausses requêtes qui rentre dans l'une des quatre classes d'erreurs suivantes : erreurs de manipulation des n-uplets ; erreurs de spécification de l'environnement ; erreurs d'implémentation de l'environnement ; et erreurs de composition des requêtes. L'implémentation réalise également la traduction d'une requête C2QL vers un terme π-calcul pour montrer que le langage peut être compilé vers un programme ProVerif et une application concrète du nuage.

Le langage C2QL répond donc au défi posé par cette thèse en permettant l'écriture d'applications du nuage qui préserve les données personnelles. Avec

C2QL, l'application profite au maximum du nuage et préserve correctement les données personnelles. Ce qui n'est pas le cas des autres solutions (*cf.* §3.4). Les solutions comme CryptDB, TrustedDB et Cipherbase ne protègent pas sur toute la pile du nuage. Elles se limitent à la base de données. Or, la politique de préservation des données personnelles dit qu'aucun acteur du nuage n'est de confiance (ni l'infrastructure IaaS, ni les bases de données PaaS, ni l'application SaaS). Le langage d'Antignac et Le Métayer est un langage d'architecture qui ne permet pas l'écriture, mais la spécification des applications PbD. Le langage C2QL se veut plus pragmatique. Il repose sur l'algèbre relationnelle dont la sémantique opérationnelle est très claire.

Le langage C2QL n'est pas limité à la programmation d'un agenda personnel en ligne. Il permet de programmer les applications communes du nuage qui s'exécutent dans un SaaS et sauvegardent leurs données au niveau PaaS. Toutefois, le langage ne permet pas l'écriture d'applications avec des interactions entre les clients (*ex.*, Alice partage un rendez-vous avec Bob). La raison est que ce type d'interaction rend possible la seconde dissémination des données personnelles et qu'il n'existe pas de techniques de cryptographie pour empêcher cela.

Perspectives

Un ensemble complet de techniques de cryptographie. La section 2.4 du chapitre 2 cite une dizaine de techniques de cryptographie et seulement trois sont dans le langage C2QL. Le langage C2QL pourrait donc être étendu avec d'autres techniques de cryptographie. Surtout que le langage et ses lois forment un cadre pour raisonner sur la composition des techniques de cryptographie. Étendre le langage consiste à ajouter les spécificateurs/-destructeurs de la nouvelle technique. Puis, ajouter de nouvelles lois qui expliquent comment ces spécificateurs/destructeurs commutent avec les fonctions déjà présentes dans C2QL tout en préservant les équivalences. Il serait également intéressant de voir quelles applications il serait possible d'écrire avec ces nouvelles techniques.

Vers une stratégie d'optimisation. Les lois de C2QL aident à atteindre l'approche du *nuage confidentiel*, mais les lois ne forment pas une stratégie pour trouver la requête la *plus* optimisée, puisque souvent, plusieurs lois avec des optimisations différentes peuvent s'appliquer. Ce qui est problématique pour implémenter une optimisation automatique, à la manière d'Aggarwal et collab. (2005) par exemple.

Je ne vois pas cet aspect comme une limitation, mais comme une obligation. La composition produit plusieurs optimisations possibles et le choix d'une optimisation, par rapport à une autre, doit être dicté par les besoins de l'application.

Un exemple simple de cette affirmation considère l'optimisation introduite par la loi de sélection 14. Pour rappel, cette loi stipule que la sélection doit utiliser un prédicat p qui supporte les opérations sur une donnée chiffrée (c) quand les valeurs des attributs utilisés par le prédicat sont chiffrées ($\alpha \in \delta$). Mais, il n'est pas toujours intéressant pour le développeur d'appliquer cette loi pour optimiser en faisant passer le déchiffrement après la sélection. Cette

$\sigma_{p_\delta} \circ decrypt_{\alpha,\mathsf{c}} \equiv decrypt_{\alpha,\mathsf{c}} \circ \sigma_{\mathsf{c} \Rightarrow p_\delta}$ *si* $\alpha \in \delta$

optimisation est intéressante que si le chiffrement permet une évaluation efficace du prédicat. C'est le cas avec le chiffrement AES et le test d'égalité. En revanche, les performances seront beaucoup moins bonnes si le chiffrement est un chiffrement homomorphe total (pour rappel, 11 minutes de temps de rafraîchissement avec le schéma de van Dijk – *cf.* §2.4.3.2). Auquel cas le développeur préférera sûrement ne pas appliquer l'optimisation, mais rapatrier les données chez la cliente. En particulier si le volume de données à rapatrier est petit.

Une stratégie d'optimisation serait possible, mais il faudrait pour cela définir des classes de besoins pour une application. Ainsi, avec une requête arbitraire et un ensemble de besoins, il serait possible de dériver une unique optimisation.

Compiler les terms C2QL. Dans l'implémentation au chapitre 6, un terme C2QL est traduit en un terme π-calcul grâce à la traduction du chapitre 4 (*cf.* §A.2.6). Ceci montre que le langage C2QL est adapté pour une traduction vers ProVerif et vers une application concrète. Mais il serait encore mieux d'implémenter un compilateur vers ces programmes. La traduction en ProVerif est directe et est très simple car ProVerif repose également sur le π-calcul. Elle n'a pas été faite par manque de temps. En revanche, je pense que c'est plus compliqué pour la traduction vers l'application concrète. Dans cette traduction, il y a des informations supplémentaires à gérer comme la localisation des bases de données, la gestion des clefs de chiffrement, *etc*.

Personnellement, j'espère que cette thèse aura montré qu'il était possible de programmer une application du nuage qui est respectueuse des données personnelles de ses utilisateurs. Préserver la vie privée est indispensable dans un monde comme le nôtre, où toutes les données personnelles sont conservées (pour toujours) et où l'on ne sait pas de quoi sera fait le lendemain.

Annexes de la thèse

A

A.1 Annexe du chapitre 5

A.1.1 Transformation d'un programme ProVerif en clauses de Horn

Note : this encoding is a naive one. It lets the reader get a feeling on how ProVerif tries to reach a secret. The description of the real transformation is given in the *Formal Models and Techniques for Analyzing Security Protocols* book (Blanchet, 2011).

Given the following ProVerif program.

```
1  type key.
2  fun senc(bitstring, key): bitstring.
3  reduc forall m: bitstring, k: key; sdec(k,senc(m,k)) = m.
4
5  const theSecret: bitstring [private].
6  const theKey: key [private].
7  const aNonPrivateFact.
8
9  query attacker(theSecret).
10
11 free toB, toC: channel.
12 process
13   out(toB, senc(theSecret, theKey))       (* process A *)
14   | (in(toB, theCipherSecret: bitstring); (* process B *)
15      out(toC, theKey))
16   | in(toC, theKey: key)                  (* process C *)
```

The ProVerif transformation takes three steps. The first step encodes all facts and rules that don't involve variables. This is the case for non private `const` such as line 7.

```
attacker(aNonPrivateFact) :- true.
```

And also for messages communications. In the transformation of a message, all its causal past should appear in the right part of the rule. This encoding specifies that an attacker knows the fact sending over the channel, only if the causal past have been known. For the message sent at line 13, there is no causal past, so the encoding is straightforward.

```
attacker(senc(theSecret, theKey)) :- true.
```

In contrast, the message sent at line 15 has the message from 13 in its causal past. So the rule should be the following.

```
attacker(theKey) :- attacker(senc(theSecret, theKey)).
```

The next step is the encoding of constructors. The transformation puts arguments at the right side of the rule and the produced term at left side. Intuitively, this means that an attacker can construct the term if he knows enough information. In this rule, arguments should appears as horn variables so that the rule matches for any terms. The following transforms the `senc` constructor (l.2).

```
attacker(senc(M,K)) :- attackert(M), attacker(K).
```

The final step is the encoding of destructors. The transformation is equivalent to constructors. Variables in the destructor becomes variables in Horn clause. Arguments of the destructor appear in the right side of the rule. The following transforms the `sdec` destructor (l.3).

```
attacker(M) :- attacker(K), attacker(senc(M,K)).
```

Finally, the `query attacker(theSecret)` that tries to reach the secret (l.9) is modeled has a predicate test.

```
?- attacker(theSecret).
```

A.2 Annexe du chapitre 6

A.2.1 Gérer l'absence de preuve `So` dans un programme Idris

```
||| Perform a case analysis on a Boolean, providing clients with
||| a `So` proof. Especialy, this calls the `Data.So.choose`
||| function that returns either a proof or a contradiction and
||| returns the proof into the `Maybe` monad.
mchoose : (b : Bool) → Maybe (So b)
mchoose b = either Just (const Nothing) (choose b)

||| Dynamically applies a projection on addresses `A` over an
||| arbitrary relation of type `s`. This returns the resulted
||| `Query` if the program can compute a proof that
||| `includes [A] s`, or `None` otherwise.
|||
||| @ s The schema of the arbitrary relation
||| @ h The handler that specifies how to connects to a database
|||     with a relation of type `s`
qdynamic : {s : Schema} → (h : Handler s) → Maybe (Query [A])
qdynamic h {s} = do
  -- Get the proof and do the request if any
  prf ← mchoose (includes [A] s)
  return π s' {p=prf} $ (Unit h)
```

A.2.2 Programme principale d'une requête `Query`

```
||| Transforms schema type into a tuple of Idris native types.
|||
||| Example: `sch2Tuple RendezVous` produces
||| `(String, String, String) : Type`.
```

```
sch2Tuple : ( δ : Schema ) → Type
sch2Tuple δ = let initTy = map (interpTy . snd) $ init δ
                  lastTy = (interpTy . snd) $ last δ
              in foldr Pair lastTy initTy

||| Computes the real SQL query by visiting the `Query`. Then
||| executes it on the database. And finally returns the
||| result as a list of tuples.
query : Query Δ → IO (List (sch2tuple Δ))
query = ...

||| Main programs that gets an `Handler` and executes the query.
main : IO ()
main = do
  -- Gets the `Handler` in the `Either` monad
  eh ← connect "pg://user:passwd@url:port" "RendezVous" RendezVous
  case eh of
    -- There is no `Handler`, manage error.
    Left error => putStrLn "Error␣while␣connecting␣to␣the␣database"
    -- There is one `Handler`, executes query.
    Right h =>
      do putStrLn "[adresse]␣result␣is:"
         adresses ← query $ qadr h
         putStrLn (show adresses)
         putStrLn "#rendezvous␣result␣is:"
         nbRdv ← query $ qrdv h
         putStrLn (show nbRdv)
```

A.2.3 La fonction de fragmentation de l'environnement `fragEnv`

```
||| Fragments the environnement in the last fragment.
fragEnv : δ( : Schema) → (env : Env n) → Env (S n)
fragEnv δ env {n} = let Δs  = init env
                        Δ   = last env
                        Δl  = (intersect δ Δ) ++ [Id]
                        Δr  = nub Δ(( \\ δ) ++ [Id])
                        res = Δs ++ Δ[lΔ,r]
                    in rewrite sym (plusTwoSucSuc n) in res
  where
  -- a gentle proof
  plusTwoSucSuc : (n : Nat) → n + 2 = S (S n)
  plusTwoSucSuc Z     = Refl
  plusTwoSucSuc (S k) = let inductHypo = plusTwoSucSuc k
                        in cong inductHypo {f=S}
```

A.2.4 Composer les termes `Privy` -- explication du (>>=)

Deux termes `Privy` sont composés par le constructeur de données (>>=) nommée *bind*. Le constructeur prend deux arguments et produit un nouveau terme `Privy`.

```
(>>=) : Privy env0 env1 δ → (Query δ → Privy env1 env2 δ′) →
        Privy env0 env2 δ′
```

Le premier argument est un terme `Privy`. Dans ce terme, la variable `env0` identifie l'environnement de départ et la variable `env1` identifie l'environnement d'arrivée. La variable δ identifie le schéma relationnel des n-uplets calculés par la requête.

Le deuxième argument est une fonction qui spécifie comment construire un nouveau terme à partir de la requête calculés par le première argument. Le nouveau terme doit avoir `env1` comme environnement de départ. Il peut modifier cette environnement donc il a `env2` comme environnement d'arrivée. Le schéma relationnel δ' indique le type des nom calculé par la requête sur l'environnement `env2`.

Le résultat de ses deux arguments est un nouveau terme `Privy` qui décrit comment calculer des n-uplets dont le schéma relationnel est δ'. Cette requête est exécutable sur un environnement de forme `env2`. Et la requête spécifie comment composer les techniques de cryptographie pour passer d'un environnement `env0` à un environnement `env2`.

Par exemple, les deux spécifications `specCryptN` et `specFragD` sont composées de la manière suivante pour spécifier comment protéger la relation des rendez-vous.

```
specSafeRendezVous : Privy RendezVousEnv SafeRendezVousEnv ⊥
specSafeRendezVous = specCryptN >>= \_ => specFragD
```

Ou, encore mieux avec la notation `do`.

```
specSafeRendezVous = do specCryptN; specFragD
```

A.2.5 Traduction d'un programme `Privy` vers C2QL en Idris

```
||| Translates a Privy term into a C2QL term.
|||
||| The translation is partial since a Privy term is more powerful
||| than a C2QL one. Rules to follow to ensure that the
    transformation
||| goes well is:
||| - Do only one query per fragment and per term.
||| - Use a variable only once (similar to a linear type system).
toC2QL : Privy env env' Δ → UID C2QL
toC2QL p {env} {env'} = (do
   init ← envToC2QL env
   incrAfter fragSize -- Firsts ids of UID are reserved for
                      -- fragments. The rest is for variables.
   privyToC2QL p init) where


  ||| Returns the number of fragments in the Privy term
  fragSize : Nat
  fragSize = length env'

  ||| Translates an environment into a C2QL term.
  |||
  ||| Generates the intial structure for the C2QL term. There is two
  ||| cases:
  ||| - It starts from an unfragmented environment, then the most
```

```
|||   right term is the relation with schema δ.
||| - It starts from a fragmented environment, then the most right
|||   term is the combination of defrag/frag over the relation.
|||   The relation is the concatenation of all attributes in all
|||   fragmentes.
||| This also introduce crypt term if the environment includes
||| encrypted attributes.
envToC2QL : Env n → UID C2QL
envToC2QL δ[] = pure $ Rel δ
envToC2QL δs@(_ :: _ :: _)  =
    let rel = Rel δ[ | δ ← concat δs, δ /= Id]
    in envToC2QL' δs rel where
  envToC2QL' : Env (S n) → C2QL → UID C2QL
  envToC2QL' δ[, δ']               c = do
    id ← freshId
    pure $ Defrag (Hole_ id, Hole_ (S id)) $ c
  envToC2QL' δs@(w :: x :: y :: z) c = do
    id ← freshId
    let env = tail δs
    let c'  = Hole_ (id + fragSize)
    rec ← envToC2QL' env c'
    pure $ Defrag (Hole_ id, rec) c

||| Returns True if the current C2QL term contains, at least, one
||| Defrag.
hasDefrag : C2QL → Bool
hasDefrag (Project x y)   = hasDefrag y
hasDefrag (Product x y)   = hasDefrag x || hasDefrag y
hasDefrag (NJoin x y)     = hasDefrag x || hasDefrag y
hasDefrag (Count x y)     = hasDefrag y
hasDefrag (Select x y)    = hasDefrag y
hasDefrag (Crypt x y z)   = hasDefrag z
hasDefrag (Decrypt x y z) = hasDefrag z
hasDefrag (Frag x y)      = hasDefrag y
hasDefrag (Defrag x y)    = True
hasDefrag (Rel x)         = False
hasDefrag (Hole_ x)       = False

||| Translates a Privy predicate into a C2QL predicate.
predToC2QL : Pred e e' → C2QLPred
predToC2QL (AND x y)    = AND (predToC2QL x) (predToC2QL y)
predToC2QL (OR x y)     = OR (predToC2QL x) (predToC2QL y)
predToC2QL (Like a pat) = Like a pat
predToC2QL (NextWeek a) = NextWeek a
predToC2QL (Equal a v)  = Equal a v

||| Translates a Privy Query into a C2QL term.
queryToC2QL : Query Δ → C2QL → C2QL
queryToC2QL (Project δ q) c2ql = Project δ (queryToC2QL q c2ql)
queryToC2QL (Count δ q) c2ql = Count δ (queryToC2QL q c2ql)
queryToC2QL (Select p q) c2ql = Select (predToC2QL p)
                                       (queryToC2QL q c2ql)
queryToC2QL (Decrypt a d {c} q) c2ql =
  Decrypt a c (queryToC2QL q c2ql)
queryToC2QL q c2ql = c2ql
```

```
  ||| Translates a Privy term into a C2QL one.
  |||
  ||| @c2ql The initial C2QL term (in general, the relation `Rel`)
  privyToC2QL : Privy e e' Δ → (c2ql : C2QL) → UID C2QL
  privyToC2QL (Crypt a c) c2ql             = pure $ Crypt a c c2ql
  privyToC2QL (Frag δ') c2ql {e} with (hasDefrag c2ql)
    privyToC2QL (Frag δ') c2ql {e} | False =
      let fragId    = pred $ length e
          holeLeft  = Hole_ fragId
          holeRight = Hole_ (S fragId)
      in pure $ Defrag (holeLeft, holeRight) $ Frag δ' $ c2ql
    privyToC2QL (Frag δ') c2ql {e} | True  =
      let fragId     = pred $ length e
          holeLeft   = Hole_ fragId
          holeRight  = Hole_ (S fragId)
      in do
      let oldFrag = Hole_ !freshId
      let filler  = Defrag (holeLeft, holeRight) $ Frag δ' $ oldFrag
      pure $ c2qlFillHole c2ql holeLeft filler
  privyToC2QL (Query fId {env} kq) c2ql with (hasDefrag c2ql)
    privyToC2QL (Query fId {env} kq) c2ql | False =
      let query = kq $ Var_ (index fId env) in
      pure $ queryToC2QL query c2ql
    privyToC2QL (Query fId {env} kq) c2ql | True  =
      let query   = kq $ Var_ (index fId env)
          hole    = Hole_ (finToNat fId)
          oldFrag = Hole_ !freshId
          filler  = queryToC2QL query oldFrag
      in pure $ c2qlFillHole c2ql hole filler
  privyToC2QL ((>>=) p δ{} kp) c2ql        = do
    c2ql' ← privyToC2QL p c2ql
    let p' = kp $ Var_ δ
    privyToC2QL p' c2ql'
  privyToC2QL (Return p) c2ql              =
    pure $ queryToC2QL p c2ql
```

A.2.6 Traduction d'un programme C2QL vers π-calcul en Idris

```
||| Translates a C2QL terms into a Pi term with a parallele
||| strategy.
|||
||| @ h Name of the request.
||| @ q C2QL term to translate.
toPi : (h : String) → (q : C2QL) → Pi
toPi h q = let maybeApp  = map Rep (toApp h q)
               frags     = map Rep (toFrag h q)
               client    = toClient h (length frags) q
           in (mkNewNApp maybeApp
              (mkNewNFrags frags
              (NewN (MkN "client")
                (Par (fromMaybe End maybeApp)
                     (parStar frags client))))) where
```

```
||| Makes the application Pi term of the C2QL term.
|||
||| App acts as a proxy between the database and the client.
toApp : (h : String) → C2QL → Maybe Pi
toApp h q = let pi = runPure (toApp' q)
            in pure $
            Rcv (MkN "app") [MkN "url"] (
              Guard (MkN "url") (MkN h) !pi) where
  toApp' : C2QL → UID (Maybe Pi)
  toApp' (Project _ q)       = toApp' q
  toApp' (Product q1 q2)     = toApp' q1 <|> toApp' q2
  toApp' (NJoin q1 q2)       = toApp' q1 <|> toApp' q2
  toApp' (Count _ q)         = toApp' q
  toApp' (Select _ q)        = toApp' q
  toApp' (Crypt _ _ q)       = toApp' q
  toApp' (Decrypt a c q)     = do
    maybeDefrag ← toApp' q
    pure (maybeDefrag
          <|> Just (Send (MkN "db") [MkN "url", MkN "client"] End)
  )
  toApp' (Frag _ q)          = toApp' q
  toApp' (Defrag (ql, Defrag x y) _) = do
    let qr = Defrag x y
    idl  ← mkSym "frag_"
    qrPi ← assert_total $ toApp' qr
    pure $ do
      qrPi' ← qrPi
      pure (Par (Send (MkN "fragi") [MkN "url", MkN "client"] End)
   qrPi')
  toApp' (Defrag (ql, qr) _) = do
    idl ← mkSym "frag_"
    idr ← mkSym "frag_"
    pure $
      Just (Par (Send (MkN idl) [MkN "url", MkN "client"] End)
                (Send (MkN idr) [MkN "url", MkN "client"] End))
  toApp' (Rel _)             = pure Nothing
  toApp' (Hole_ _)           = pure Nothing

||| Makes the client Pi term of the C2QL term.
toClient : (h : String) → (nbFrag : Nat) → C2QL → Pi
toClient h nbFrag q =
  let kpi = runPureInit ['At := Bot, 'UIDVar := Z, 'Ctx := ⊥]
                        (toClient' q)
  in kpi End where
  toClient' : C2QL → { [
                'At     ::: STATE Place,
                'UIDVar ::: STATE Nat,
                'Ctx    ::: STATE (List String) ] } Eff (Pi → Pi)
  toClient' (Select p q)        = do
    kpi ← toClient' q
    if List.elem !('At :- get) [Client, Bot]
    then (do
      rel ← Rel <$> 'Ctx :- pop
      id  ← 'UIDVar :- mkSym "r_"
      'Ctx :- push id
```

```
      -- XXX Report bug: (.) produces segfault
      -- pure $ Let (MkN id) (Select (toQPred p) rel) . kpi)
      pure $ \pi => kpi $ Let (MkN id) (Select (toQPred p) rel) pi
  )
    else pure kpi
  toClient' (Project δ q)       = do
    kpi ← toClient' q
    if List.elem !('At :- get) [Client, Bot]
    then (do
      rel ← Rel <$> 'Ctx :- pop
      id  ← 'UIDVar :- mkSym "r_"
      'Ctx :- push id
      pure $ \pi => kpi $ Let (MkN id) (Project δ rel) pi)
    else pure kpi
  toClient' (Count δ q)         = do
    kpi ← toClient' q
    if List.elem !('At :- get) [Client, Bot]
    then (do
      rel ← Rel <$> 'Ctx :- pop
      id  ← 'UIDVar :- mkSym "r_"
      'Ctx :- push id
      pure $ \pi => kpi $ Let (MkN id) (Count δ rel) pi)
    else pure kpi
  toClient' (Rel δ)             = do
    if !('At :- get) == Bot
    then (do
      id  ← 'UIDVar :- mkSym "r_"
      'Ctx :- push id
      pure $ \pi => NewR (MkR id δ) pi)
    else pure id
  -- C2QL destructor
  toClient' (Decrypt a c q)     = do
    kpi ← toClient' q
    if List.elem !('At :- get) [Client, Bot]
    then (do
      rel ← Rel <$> 'Ctx :- pop
      id  ← 'UIDVar :- mkSym "r_"
      'Ctx :- push id
      pure $ \pi => kpi $ Let (MkN id) (Decrypt a c rel) pi)
    else (do
      idRcv  ← 'UIDVar :- mkSym "r_"
      idLet  ← 'UIDVar :- mkSym "r_"
      'Ctx :- push idLet
      'At  :- put Client
      pure $ \pi => kpi $
        Send (MkN "app") [MkN h] (
          Rcv (MkN "client") [MkN idRcv] (
            Let (MkN idLet) (Decrypt a c (Rel idRcv)) pi)))
  toClient' (Defrag _ q) = (do
    kpi ← toClient' q
    let sendAppPi = \pi => kpi (Send  (MkN "app") [MkN h] pi)
    -- Construct all the receive continuation
    rcvPis ← sequence $ List.replicate nbFrag (do
      idRcv ← 'UIDVar :- mkSym "r_"
      'Ctx :- push idRcv
```

```
    pure $ \pi => Rcv (MkN "client") [MkN idRcv] pi)
  -- Then makes all relation Q term using the context
  rels ← sequence $ List.replicate nbFrag (Rel <$> 'Ctx :- pop)
  -- And construct the Defrag pi term from these rels
  idLet  ← 'UIDVar :- mkSym "r_"
  let defLet = \pi => Let (MkN idLet)
                          (foldr Defrag (last rels) (init rels))
 pi
  'Ctx :- push idLet
  'At  :- put Client
  -- Chain all receives together and finish with the let.
  let rcvPi = foldr (\pik,r => \pi => pik $ r pi) defLet rcvPis
  -- Finally, add the sendApp term and construct the final term.
  pure (\pi => kpi $ Send (MkN "app") [MkN h] $ rcvPi pi)) where
    init : List a → List a
    init ⊥        = ⊥
    init [x]      = ⊥
    init (x :: xs) = x :: (init xs)

    last : List Q → Q
    last ⊥        = Rel "error"
    last [x]      = x
    last (x :: xs) = last xs
-- C2QL spec
toClient' (Frag δ q)          = do
  toClient' q
  'At :- put (Frag Z)
  pure id
toClient' (Crypt a c q)       = do
  toClient' q
  'At :- put (Frag Z)
  pure id
toClient' _                   = pure id
```

Bibliographie

Abadi, M. et Blanchet, B. (2002). Analyzing security protocols with secrecy types and logic programs. Dans *Conference Record of POPL 2002 : The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 33–44.

Abadi, M. et Fournet, C. (2001). Mobile values, new names, and secure communication. Dans *Conference Record of POPL 2001 : The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 104–115.

Abouzaid, F. et Mullins, J. (2008). A calculus for generation, verification and refinement of BPEL specifications. *Electr. Notes Theor. Comput. Sci.*, 200(3):43–65.

Aggarwal, G., Bawa, M., Ganesan, P., Garcia-Molina, H., Kenthapadi, K., Motwani, R., Srivastava, U., Thomas, D., et Xu, Y. (2005). Two can keep A secret : A distributed architecture for secure database services. Dans *CIDR*, pages 186–199.

Aleph1 (1996). Smashing The Stack For Fun And Profit. Phrack 49 – phrack.org/issues/49/14.html. Accès : 26/10/2015.

Amey, P. (2002). Correctness by construction : Better can also be cheaper. *CrossTalk : the Journal of Defense Software Engineering*, 2:24–28.

Antignac, T. et Métayer, D. L. (2015). Trust driven strategies for privacy by design. Dans *Trust Management IX - 9th IFIP WG 11.11 International Conference, IFIPTM 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, pages 60–75.

Arasu, A., Eguro, K., Joglekar, M., Kaushik, R., Kossmann, D., et Ramamurthy, R. (2015). Transaction processing on confidential data using cipherbase. Dans *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 435–446.

Assange, J., Appelbaum, J., Muller-Maguhn, A., et Zimmermann, J. (2012). *Cypherpunks : Freedom and the Future of the Internet*. " O'Reilly Media, Inc.".

Atkey, R. (2009). Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376.

Backus, J. W. (1978). Can programming be liberated from the von neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641.

Bajaj, S. et Sion, R. (2011). Trusteddb : a trusted hardware based database with privacy and data confidentiality. Dans *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 205–216.

Bambauer, J., Muralidhar, K., et Sarathy, R. (2013). Fool's gold : an illustrated critique of differential privacy. *Vand. J. Ent. & Tech. L.*, 16:701.

Biskup, J., Preuß, M., et Wiese, L. (2011). On the inference-proofness of database fragmentation satisfying confidentiality constraints. Dans *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings*, pages 246–261.

Blanchet, B. (2001). An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. Dans *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada. IEEE Computer Society.

Blanchet, B. (2002). From Secrecy to Authenticity in Security Protocols. Dans Hermenegildo, M. et Puebla, G., éditeurs, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359, Madrid, Spain. Springer.

Blanchet, B. (2011). Using Horn clauses for analyzing security protocols. Dans Cortier, V. et Kremer, S., éditeurs, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, pages 86–111. IOS Press.

Blanchet, B., Abadi, M., et Fournet, C. (2008). Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51.

Blanchet, B., Smyth, B., et Cheval, V. (2014). Proverif 1.88 : Automatic cryptographic protocol verifier, user manual and tutorial.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Boldyreva, A., Chenette, N., et O'Neill, A. (2011). Order-preserving encryption revisited : Improved security analysis and alternative solutions. Dans *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 578–595.

Booch, G., Rumbaugh, J. E., et Jacobson, I. (2005). *The unified modeling language user guide - covers UML 2.0, Second Edition*. Addison Wesley object technology series. Addison-Wesley.

Borisov, N., Goldberg, I., et Brewer, E. A. (2004). Off-the-record communication, or, why not to use PGP. Dans *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, pages 77–84.

Brady, E. (2013). Idris, a general-purpose dependently typed programming language : Design and implementation. *Journal of Functional Programming*, 23:552–593.

Brands, S. (1997). Rapid demonstration of linear relations connected by boolean operators. Dans *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 318–333.

Bray, T. (2014). The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7129, RFC Editor.

Buyya, R. (2009). Market-oriented cloud computing : Vision, hype, and reality of delivering computing as the 5th utility. Dans *CCGRID*, page 1.

Camenisch, J., Chaabouni, R., et Shelat, A. (2008). Efficient protocols for set membership and range proofs. Dans *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*, pages 234–252.

Camenisch, J. et Michels, M. (1999). Proving in zero-knowledge that a number is the product of two safe primes. Dans *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 107–122.

Cavoukian, A. (2011). Privacy by Design – 7 Foundational Principles. privacybydesign.ca/index.php/about-pbd/translations. Accès : 11/11/2015.

Ciriani, V., di Vimercati, S. D. C., Foresti, S., Jajodia, S., Paraboschi, S., et Samarati, P. (2010). Combining fragmentation and encryption to protect privacy in data storage. *ACM Trans. Inf. Syst. Secur.*, 13(3).

Clocksin, W. et Mellish, C. S. (2003). *Programming in PROLOG*. Springer Science & Business Media.

Coron, J., Naccache, D., et Tibouchi, M. (2012). Public key compression and modulus switching for fully homomorphic encryption over the integers. Dans *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, pages 446–464.

Daemen, J. et Rijmen, V. (1999). AES proposal : Rijndael.

Danezis, G. et Livshits, B. (2011). Towards ensuring client-side computational integrity. Dans *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 125–130.

di Vimercati, S. D. C., Erbacher, R. F., Foresti, S., Jajodia, S., Livraga, G., et Samarati, P. (2013). Encryption and fragmentation for data confidentiality in the cloud. Dans *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, pages 212–243.

Diffie, W., van Oorschot, P. C., et Wiener, M. J. (1992). Authentication and authenticated key exchanges. *Des. Codes Cryptography*, 2(2):107–125.

Dolev, D. et Yao, A. C. (1983). On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2):198–207.

Dwork, C., McSherry, F., Nissim, K., et Smith, A. (2006). Calibrating noise to sensitivity in private data analysis. Dans *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 265–284.

Dwork, C. et Roth, A. (2014). The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407.

Egorov, M. et Wilkison, M. (2016). Zerodb white paper. *CoRR*, abs/1602.07168.

Erlingsson, Ú., Pihur, V., et Korolova, A. (2014). RAPPOR : randomized aggregatable privacy-preserving ordinal response. Dans *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1054–1067.

Feigenbaum, J., Jaggard, A., Wright, R., et Xiao, H. (2012). Systematizing'accountability'in computer science. Rapport technique, YALEU/DCS/TR-1452, Yale University, New Haven CT.

Fielding, R. et Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1) : Authentication. RFC 7230, RFC Editor.

Filliâtre, J.-C. et Paskevich, A. (2013). Why3 – Where Programs Meet Provers. Dans *ESOP'13 22nd European Symposium on Programming*, volume 7792, Rome, Italy. Springer.

Fournet, C., Kohlweiss, M., Danezis, G., et Luo, Z. (2013). ZQL : A compiler for privacy-preserving data processing. Dans *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 163–178.

Gamal, T. E. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472.

Gentry, C. (2009). *A fully homomorphic encryption scheme*. PhD thesis, Stanford University. crypto.stanford.edu/craig.

Gentry, C. (2010). Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105.

Gentry, C. et Halevi, S. (2011). Implementing gentry's fully-homomorphic encryption scheme. Dans *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, pages 129–148.

Goldwasser, S., Micali, S., et Rackoff, C. (1989). The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208.

Golić, J. D. (1997). Cryptanalysis of alleged a5 stream cipher. Dans *Advances in Cryptology—EUROCRYPT'97*, pages 239–255. Springer.

Grimmelmann, J. (2005). Regulation by software. *Yale Law Journal*, pages 1719–1758.

Hacigümüs, H., Iyer, B. R., Li, C., et Mehrotra, S. (2002). Executing SQL over encrypted data in the database-service-provider model. Dans *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 216–227.

Hoffman, L. J., Lawson-Jenkins, K., et Blum, J. (2006). Trust beyond security : An expanded trust model. *Commun. ACM*, 49(7):94–101.

Hudak, P. (1998). Modular domain specific languages and tools. Dans *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society.

Kerschbaum, F. (2009). Adapting privacy-preserving computation to the service provider model. Dans *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009, Vancouver, BC, Canada, August 29-31, 2009*, pages 34–41.

Klitou, D. (2014). Privacy, liberty and security. Dans *Privacy-Invading Technologies and Privacy by Design*, volume 25 of *Information Technology and Law Series*, pages 13–25. T.M.C. Asser Press.

Lampson, B. W. (2004). Computer security in the real world. *IEEE Computer*, 37(6):37–46.

Lerdorf, R., Tatroe, K., et MacIntyre, P. (2006). *Programming Php*. " O'Reilly Media, Inc.".

Lucchi, R. et Mazzara, M. (2007). A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118.

Lyons, J. (2012a). Caesar Cipher. practicalcryptography.com/ciphers/classical-era/caesar. Accès : 11/11/2015.

Lyons, J. (2012b). Enigma Cipher. practicalcryptography.com/ciphers/mechanical-era/enigma. Accès : 11/11/2015.

Malkin, G. (1996). Internet Users' Glossary. RFC 1983, RFC Editor.

McCarthy, J. (1965). *LISP 1.5 programmer's manual*. MIT press.

MDN (2015). What is JavaScript? developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript. Accès : 12/11/2015, "JavaScript (often shortened to JS) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages [...] JavaScript runs on the client side of the web".

Mell, P. et Grance, T. (2011). The nist definition of cloud computing.

Menezes, A., van Oorschot, P. C., et Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press.

Milner, R. (1978). A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375.

Milner, R. (1999). *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

Narayanan, A. et Shmatikov, V. (2008). Robust de-anonymization of large sparse datasetssch. Dans *IEEE Symposium on Security and Privacy*, pages 111–125.

Nelson, T., Ferguson, A. D., Scheer, M. J., et Krishnamurthi, S. (2014). Tierless programming and reasoning for software-defined networks. Dans *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, Seattle, WA. USENIX Association.

Odersky, M., Spoon, L., et Venners, B. (2008). *Programming in Scala*. Artima Inc.

Oury, N. et Swierstra, W. (2008). The power of pi. Dans *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 39–50.

Özsu, M. T. et Valduriez, P. (2011). *Principles of Distributed Database Systems, Third Edition*. Springer.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. Dans *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 223–238.

Pearson, S. (2011). Toward accountability in the cloud. *IEEE Internet Computing*, 15(4):64–69.

Pedersen, T. P. (1991). Non-interactive and information-theoretic secure verifiable secret sharing. Dans *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, pages 129–140.

Peyton Jones, S. (2003). *Haskell 98 language and libraries : the revised report*. Cambridge University Press. Disponible en ligne à l'adresse wiki.haskell.org/Language_and_library_specification.

Peyton Jones, S., Jones, M., et Meijer, E. (1997). Type classes : exploring the design space. Dans *Haskell workshop*, volume 1997.

Peyton Jones, S., Vytiniotis, D., Weirich, S., et Washburn, G. (2006). Simple unification-based type inference for gadts. Dans *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 50–61.

Philips, L., De Roover, C., Van Cutsem, T., et De Meuter, W. (2014). Towards tierless web development without tierless languages. Dans *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 69–81, New York, NY, USA. ACM.

Popa, R. A., Redfield, C. M. S., Zeldovich, N., et Balakrishnan, H. (2011). Cryptdb : protecting confidentiality with encrypted query processing. Dans *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 85–100.

Raykova, M., Vo, B., Bellovin, S. M., et Malkin, T. (2009). Secure anonymous database search. Dans *Proceedings of the first ACM Cloud Computing Security Workshop, CCSW 2009, Chicago, IL, USA, November 13, 2009*, pages 115–126.

Rivest, R. L., Shamir, A., et Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126.

Schneier, B. (1996). *Applied cryptography - protocols, algorithms, and source code in C (2. ed.)*. Wiley.

Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., et Ferguson, N. (1999). *The Twofish Encryption Algorithm : A 128-bit Block Cipher*. John Wiley & Sons, Inc., New York, NY, USA.

Schnorr, C. (1991). Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174.

Schrijvers, T. et Demoen, B. (2008). Equational Reasoning for Prolog.

Solove, D. J. (2006). A taxonomy of privacy. *University of Pennsylvania law review*, pages 477–564.

Song, D. X., Wagner, D., et Perrig, A. (2000). Practical techniques for searches on encrypted data. Dans *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 44–55.

Spiekermann, S. (2012). The challenges of privacy by design. *Commun. ACM*, 55(7):38–40.

Stump, A. (2014). *Verified Functional Programming in Agda*.

Syme, D., Granicz, A., et Cisternino, A. (2012). *Expert F*. Apress.

Turner, D. A. (1995). Elementary strong functional programming. Dans *Functional Programming Languages in Education, First International Symposium, FPLE'95, Nijmegen, The Netherlands, December 4-6, 1995, Proceedings*, pages 1–13.

Ullman, J. D. (1982). *Principles of Database Systems, 2nd Edition*. Computer Science Press.

Van Blarkom, G., Borking, J., et Olk, J. (2003). Handbook of privacy and privacy-enhancing technologies. *Privacy Incorporated Software Agent (PISA) Consortium, The Hague*.

van Dijk, M., Gentry, C., Halevi, S., et Vaikuntanathan, V. (2010). Fully homomorphic encryption over the integers. Dans *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, pages 24–43.

Wadler, P. (1992). The essence of functional programming. Dans *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14.

Weitzner, D. J., Abelson, H., Berners-Lee, T., Feigenbaum, J., Hendler, J. A., et Sussman, G. J. (2008). Information accountability. *Commun. ACM*, 51(6):82–87.

Yao, A. C. (1982). Protocols for secure computations (extended abstract). Dans *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164.

Youseff, L., Butrico, M., et Da Silva, D. (2008). Toward a unified ontology of cloud computing. Dans *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE.

Zimmermann, H. (1980). Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432.

Colophon

Cette thèse a été rédigée avec le mode `org mode` de l'éditeur de texte libre `GNU Emacs`. Ce document a été composé à l'aide de LaTeX. Il utilise la classe de mise en forme libre `classicthesis` développée par André Miede. Pour le texte, le corps est rendu grâce à la police libre Crimson, les titres des sections et paragraphes grâce à la police libre Vollkorn et le texte à chasse fixe grâce à la police libre Fira Mono.

`http://ctan.org/pkg/classicthesis`
`https://fontlibrary.org/en/font/crimson`
`https://fontlibrary.org/en/font/vollkorn`
`http://mozilla.github.io/Fira`

Les schémas ont été réalisés grâce au logiciel libre Inkscape. Les icônes utilisées dans les images proviennent de la bibliothèque libre `IcoMoon-Free`.

`https://inkscape.org`
`https://github.com/Keyamoon/IcoMoon-Free`

Les programmes développés et présentés dans cette thèse sont sous licence GNU GPLv2. Ils sont disponibles sur github.

`https://github.com/rcherrueau/C2QL`

Un langage de composition des techniques de sécurité pour préserver la vie privée dans le nuage.