Conception d'un langage pour les réductions cryptographiques

Ducas Léo, Baudet Mathieu, DCSSI

Mars-Aout 2009

Fiche de synthèse

Le contexte général

Une large branche de la cryptographie s'intéresse aux constructions dont la sécurité est prouvée mathématiquement, à l'aide de réductions. Cependant, certaines erreurs sont difficiles à détecter car les énoncés cryptographiques font intervenir plusieurs algorithmes, avec des interactions parfois complexes.

Quelques projets de recherche se sont récemment intéressés aux méthodes formelles, visant à vérifier, voire automatiser les preuves cryptographiques. *CryptoVerif* [Bla06, BP06] permet de vérifier automatiquement la sécurité des protocoles cryptographiques, utilisant les outils de la réécriture. On trouve aussi des systèmes de preuves certifiés à l'aide d'assistants de preuve, *Coq* [BGJB07], ou *Isabelle/HOL* [BBU08].

Le problème étudié

Jusqu'à présent, les langages utilisés pour énoncer les protocoles et les théorèmes dans ces formalismes sont restés volontairement proches des pseudo-codes utilisés par les cryptologues. Ces langages sont bien adaptés pour exprimer les énoncés cryptographiques usuels et les étapes d'une preuve par réécriture de jeux cryptographiques.

Cependant, de telles approches n'explicitent pas les réductions, ce qui est incontournable pour certains résultats d'impossibilité ou de séparation.

Nous nous sommes au contraire intéressés aux réductions, et donc à la conception d'un formalisme adapté à la description des notions et réductions cryptographiques. Les objectifs principaux étaient que les réductions l'implémentent aussi facilement qu'en pseudo-codes et qu'elles soient lisibles, mais aussi que le langage se prête à des preuves formelles, éventuellement à l'aide d' un assistant de preuve.

La contribution proposée

Après quelques expérimentations en *Ocaml* pour caractériser les éléments nécessaires ou juste utiles pour l'implémentation des réductions cryptographiques, le stage a abouti à la définition d'un langage ayant les caractéristiques recherchées, ainsi qu'à un interpréteur associé. La capacité de ce langage à exprimer les

énoncés cryptographiques est appuyée par trois exemples complets, issus de constructions cryptographiques réalistes.

L'un des exemples nous a amenés à développer une technique originale pour formaliser les propriétés dites d'algébricité des algorithmes, en prenant avantage du polymorphisme du langage, basé sur les résultats de paramétricité de Walder [Wad89].

Les arguments en faveur de sa validité

L'implémentation des exemples de réduction cryptographique nous semble répondre à notre objectif de clarté et de pédagogie, en particulier grâce au typage polymorphe qui aide à comprendre en détail les interactions entre les programmes.

L'explicitation de réductions dans notre langage offrirait déjà des qualités formelles aux preuves cryptographiques, en particulier sans aller jusqu'aux preuves formelles ; on peut grâce à l'interpréteur tester les réductions, en prenant des paramètres de sécurité suffisamment petits pour qu'une attaque exhaustive puisse s'exécuter en un temps raisonnable.

Enfin, la simplicité de la sémantique laisse penser qu'il sera possible de prouver formellement les énoncés, c'est à dire la correction des réductions. En s'imposant un style de programmation modulaire, il est probable que de telles preuves formelles ne soient pas trop fastidieuses à faire dans un assistant de preuve.

Le bilan et les perspectives

Le troisième exemple démontre que notre approche formelle peut s'appliquer aux énoncés complexes de la cryptographie moderne. Il reste cependant des techniques de preuve en cryptographie à explorer, comme la technique de rejeu (Forking Lemma), pour lesquelles il faudra peut-être enrichir le langage. Nous évoquerons de plus la question de la complétude en conclusion.

La suite logique de ces travaux serait de chercher des techniques efficaces et pratiques pour prouver formellement les énoncés construits avec ce langage. Il faudrait en particulier développer un compilateur de notre langage vers un assistant de preuve, et associer aux primitives souvent utilisées dans les réductions des lemmes sur leur sémantique.

Il pourrait en outre être intéressant et fructueux d'utiliser des techniques inspirées de l'analyse statique, notamment pour prouver les propriétés voulues des primitives ayant des effets de bord.

1 Introduction

Les preuves de sécurité de systèmes cryptographiques reposent sur le principe de la réduction algorithmique : on définit la notion de sécurité souhaitée sous forme d'un problème que doit résoudre un attaquant, puis on prouve qu'un attaquant résolvant ce problème pourrait servir d'oracle à un autre algorithme, la réduction, qui serait capable de résoudre un autre problème, dont on est convaincu de la difficulté algorithmique. Pour construire la réduction, on considère l'attaquant comme une "black-box", c'est-à-dire qu'on ne connaît rien

de son fonctionnement interne, car on souhaite que la réduction soit générique, fonctionnant quel que soit l'attaquant pour peu qu'il soit valide.

Cependant, les réductions cryptographiques peuvent avoir un aspect bien différent des réductions classiques de la théorie de la complexité. Les problèmes utilisés en cryptographie ont des énoncés plus complexes, font intervenir plusieurs algorithmes probabilistes selon certaines règles sur leurs interactions. Les réductions cryptographiques doivent alors manipuler plusieurs algorithmes probabilistes et interactifs, en se conformant à de nombreuses contraintes pour que les différents algorithmes protagonistes aient l'impression d'être dans leur environnement naturel.

Le manque de formalisme dans la description d'une réduction peut laisser des ambiguïtés, voire des confusions, et il est arrivé à plusieurs reprises que des preuves de sécurité publiées se soient avérées fausses *a posteriori* – un exemple célèbre étant la preuve initiale du schéma OAEP [BR95] .

Dans ce contexte, nous nous somme donc intéressés à la conception d'un formalisme adapté à la description des notions et réductions cryptographiques. Il existe quelques projets de recherche dans cette direction, visant à vérifier voire automatiser les preuves cryptographiques. *CryptoVerif* [Bla06, BP06] permet de vérifier automatiquement la sécurité des protocoles cryptographiques, utilisant les outils de la réécriture. On trouve aussi des systèmes de preuves certifiés à l'aide d'assistants de preuve, *Coq* [BGJB07], ou *Isabelle/HOL* [BBU08].

Les méthodes formelles utilisées en cryptographie se basent sur une approche par jeu : on change pas à pas le problème de départ jusqu'à arriver à un énoncé trivial. Cette approche, adoptée aujourd'hui par une grande partie des cryptologues, se révèle pratique et efficace. Elle mène par contre à des preuves non constructives, où les réductions ne sont pas explicitées.

Rogaway, dans [Rog06], argumente en faveur d'énoncés constructifs, notamment parce qu'ils permettent de combler le fossé entre la théorie et la pratique en ce qui concerne les fonctions de hachage (avec *vs.* sans clé). Il montre en effet comment formuler les énoncés de manière constructive, mais ne décrit pas explicitement la réduction dans un langage de programmation.

Nous nous sommes donc intéressés à une approche formelle et constructive, consistant à utiliser un langage pour les énoncés cryptographiques, ainsi qu'à donner explicitement les réductions dans ce même langage. Nous ne cherchons pour l'instant pas à traiter la question des preuves de correction des réductions construites, mais nous voulons tout de même que le langage construit permette à terme de faire ces preuves. Bien qu'une telle approche soit peu courante, elle présente plusieurs intérêts pour la formalisation des preuves cryptographiques. Premièrement, elle correspond mieux à l'intuition générale sur les preuves de sécurité, et nous espérons qu'elle soit pédagogiquement plus efficace. Elle permet aussi de raisonner dans un environnement plus simple, sur des systèmes finis : les questions de paramètres de sécurité, de temps de calcul de l'attaquant n'apparaissent pas dans la réduction ni dans sa preuve de correction ; elles peuvent être traitées à part. Enfin, cette approche sépare la partie algorithmique de la partie mathématique, et on peut espérer que la preuve de sécurité soit plus convaincante et plus facile une fois le programme de la réduction explicitement écrit.

Le but du stage a donc été de développer un langage simple et adapté aux énoncés cryptographiques. Nous voulons ce langage simple afin qu'il soit à terme possible de raisonner sur ce langage avec des assistants de preuve (comme *Coq* ou *Isabelle/HOL*). Nous voulons aussi que les programmes cryptographiques y soient faciles à écrire et à lire, et qu'ils permettent d'exprimer les énoncés de cryptographie moderne, comme les méta-réductions ou les techniques de re-jeu.

Les questions posées ont alors été les caractéristiques voulues pour un tel langage. Il a fallu choisir parmi diverses possibilités, en premier lieu le paradigme (λ-calcul ou π-calcul), puis les questions de sémantique (opérationnelle avec appel par valeur, dénotationnelle avec appel par nom), ainsi que les questions de typage. Des difficultés sont apparues pour avoir un langage suffisamment expressif tout en se restreignant aux choix que nous avions faits.

1.1 Résultats

Le stage a abouti à la définition d'un langage ayant les caractéristiques recherchées, ainsi qu'à un interpréteur associé. La capacité de ce langage à exprimer les énoncés cryptographiques est appuyée par trois exemples complets, issus de constructions cryptographiques réalistes. Le premier, hash-then-sign, construction classique, sert aussi d'exemple au propos de Rogaway dans [Rog06]. Le second est la construction de Goldreich Goldwasser et Micali [GGM86], transformant un générateur pseudo-aléatoire en fonction pseudo-aléatoire. Le dernier est un résultat plus complexe, la méta-réduction de Paillier et Vergnaud [PV05], prouvant un résultat de séparation entre la sécurité de certains schémas de signature et le problème du logarithme discret, sur lequel ces schémas sont justement basés. L'implémentation de ces exemples semble répondre à notre objectif de clarté et de pédagogie. Enfin, le troisième exemple nous à amenés à développer une technique originale pour formaliser les propriétés dites d'algébricité des algorithmes, en tirant avantage du polymorphisme du langage, basé sur les résultats de paramétricité de Walder [Wad89].

1.2 Plan

Nous commencerons en section 2 par présenter le langage construit, ses caractéristiques et sa définition formelle. En Section 3 nous donnerons, au travers de l'exemple hash-then-sign comment utiliser notre langage pour les énoncés cryptographiques. La section 4 propose une formalisation originale de la notion d'algébricité d'un algorithme, et fournit les outils nécessaires pour exploiter de telles hypothèses. En conclusion,la section 5 présentera d'autres types de techniques utilisées en cryptographie que l'on peut espérer pouvoir exprimer en utilisant le langage construit, ainsi que quelques pistes pour démontrer formellement la correction des réductions.

L'annexe **??** contient l'implémentation complète des trois exemples,

2 Le langage : λ-calcul monadique et polymorphe

Utiliser le λ-calcul comme base pour notre langage nous a semblé tout d'abord un moyen pratique de modéliser les interactions entre les différents programmes. En effet, l'ordre supérieur nous permet de modéliser naturellement les notions cryptographiques :

- Oracle : requête → réponse à la requête (Ordre 1)
- Attaquant : oracle → attaque sur un protocole (Ordre 2)
- Réduction : attaquant → solution à un problème (Ordre 3)
- Méta-réduction : réduction → solution à un problème (Ordre 4)

Enfin, la théorie associée au λ-calcul en général est bien connue, et il est possible de construire un λ-calcul adapté à des besoins spécifiques.

2.1 Les propriétés désirées

2.1.1 Une sémantique simple

Afin de pouvoir faire efficacement les preuves de correction sur les programmes, il est préférable que la sémantique du langage soit dénotationnelle, c'est à dire construite par induction sur les termes (contrairement à une sémantique opérationnelle, découlant de règles de réduction). Une telle sémantique offrirait, entre autres, la possibilité d'utiliser des raisonnements équationnels.

L'expressivité nécessaire pour la description des énoncés cryptographiques s'est révélée suffisamment faible pour que la sémantique dénotationnelle soit restreinte à **Set** la catégorie des ensembles. Les restrictions notables permettant une sémantique dans **Set** sont la terminaison des programmes, et surtout l'impossibilité de stocker des objets ayant des effets de bord dans les références. Cette dernière propriété évite d'avoir recours à une définition récursive de la sémantique.

Avoir une sémantique dans **Set** présente des intérêts techniques, simplifiant les définitions et les preuves, mais aussi pédagogiques, car plus intuitive, et ne nécessitant pas de connaissance avancée en théorie des langages.

Si l'on souhaite, à terme, prouver formellement la correction des réductions, il sera alors possible de redéfinir notre sémantique dans un assistant de preuve (en *Coq* par exemple).

2.1.2 Effet de bord et types monadiques

Les oracles utilisés en cryptographie ont souvent recours à des effets de bord, pour tirer des jetons aléatoires, ou pour avoir un état interne. Pour ce faire, nous avons inclus dans notre langage un ruban d'aléas, et des références.

Les expériences d'implémentation en *OCaml* des réductions exemples ont mis en évidence le fait qu'il était souvent nécessaire de construire une primitive ayant des effet de bord, puis de déclencher ces effets plus tard. La solution classique est relativement simple, elle consiste à ajouter un argument de type U à la primitive construite, et à mettre les effets de bord "sous le λ". Cependant,

cette solution pollue les programmes et leur type, et pousse le programmeur à l'erreur.

Ces observations nous ont guidés vers un λ-calcul computationnel "à la Moggi",solution que nous avons retenue pour ses nombreux avantages. Premièrement, cela permet d'avoir une sémantique en appel par nom malgré la présence d'effets de bord. Deuxièmement, le déclenchement des effets de bord doit toujours être explicité, ce qui n'est pas forcément agréable pour programmer en général ; mais dans notre cadre, cela s'est avéré au contraire plus simple et plus clair que l'astuce du paragraphe précédent. Enfin, la présence ou l'absence d'effet de bord se répercute dans le type, permettant une classification plus fine des programmes. Il est notamment possible de caractériser par leur type les programmes purement fonctionnels (n'ayant pas d'effet de bord), ce qui offre une formalisation intuitive de certaines définitions cryptographiques.

2.1.3 Typage polymorphe, Types algébriques

Cherchant à trouver un moyen efficace de décrire les notions et les réductions cryptographiques, nous avons souhaité profiter au maximum de l'information donnée par le type des différents programmes. Bien qu'on puisse considérer tous les objets comme des entiers ou des chaînes de bit, nous avons voulu qu'apparaissent dans les types le "domaine intuitif" des objets considérés (est-ce un message ? une clé ? etc.). Le typage polymorphe permet dans les constructions et réductions considérées ces distinctions. De plus, cela permet au programmeur de détecter d'éventuelles erreurs, si le type d'un programme unifie deux types correspondant à des domaines éventuellement différents.

Afin d'offrir un maximum de souplesse (et pour qu'ainsi les programmes et les types puissent être aussi intuitifs que possible), nous voulons aussi avoir des types algébriques. La création de ces types et des constructeurs associés n'apparaît pas dans la syntaxe du langage, nous ajouterons au besoin les constructeurs utiles dans l'ensemble des primitives de base.

2.1.4 Induction primitive

Pour assurer la terminaison des programmes, le langage n'inclut pas d'opérateur général de récursion. Il est cependant nécessaire de pouvoir implémenter des boucles et des inductions. L'induction primitive sur les entiers suffirait pour exprimer toutes les inductions, mais obligerait à de nombreux détours dans les définitions, et rendrait les éventuelles preuves de correction bien plus fastidieuses. Nous avons donc choisi d'offrir dans les primitives du langage l'induction primitive sur tous les types algébriques.

Nous pensons que ce choix se répercutera lors d'éventuelles preuves formelles sur les programmes de notre langage, permettant des preuves par induction sur les objets naturels.

2.2 Définition partielle du langage

La définition complète [Bau09] du langage a été rédigée par Mathieu Baudet, responsable du stage. On donne ici les définitions nécessaires à la compréhension

de la suite du rapport.

2.2.1 syntaxe

Les termes du langages sont définis par la grammaire suivante :

$t, t_1, t_2 \ldots$	$::=$	x	variable
	$\mid$	c	constante prédéfinie (primitive)
	$\mid$	$\lambda x.t$	abstraction
	$\mid$	$t_1\, t_2$	application
	$\mid$	let $x = t_1$ in t_2	définition
	$\mid$	let $x \Leftarrow t_1$ in t_2	séquence de calculs
	$\mid$	val(t)	calcul unitaire

2.2.2 typage

Le typage des termes fait intervenir trois ensembles de type : $\mathcal{T}^{\langle 0\rangle} \subseteq \mathcal{T}^{\langle 1\rangle} \subseteq \mathcal{T}^{\langle 2\rangle}$:

- les types testables $\mathcal{T}^{\langle 0\rangle}$ (types auxquels on peut appliquer la primitive d'égalité polymorphe),
- les types purement fonctionnels (ou purs) $\mathcal{T}^{\langle 1\rangle}$, et
- les types généraux (*i.e.* éventuellement computationnels) $\mathcal{T}^{\langle 2\rangle}$.

Soit $\mathcal{K}$ un ensemble fini de constructeurs κ, ayant chacun une arité $n \in \mathbb{N}$. Soit aussi $\mathcal{X}^{\langle 0\rangle} \subseteq \mathcal{X}^{\langle 1\rangle} \subseteq \mathcal{X}^{\langle 2\rangle}$ trois ensembles dénombrables de variable de type, on définit alors les trois ensembles de types par les grammaires décrites en annexe A.

Un environnement de typage est une séquence finie d'axiomes de typage :

$$\Gamma ::= (x_1 : \sigma_1, \ldots, x_n : \sigma_n) \qquad (n \geq 0).$$

On définit $\mathrm{Dom}(\Gamma) = \{x_1, \ldots, x_n\}$ et $\mathrm{Var}(\Gamma) = \mathrm{Var}(\sigma_1, \ldots, \sigma_n)$. On notera $\Gamma(x) = \sigma$ si $\Gamma = (\Gamma_1, x : \sigma, \Gamma_2)$ et $x \notin \mathrm{Dom}(\Gamma_2)$.

Pour un type τ, la généralisation de τ en un schéma de type dans Γ est définit comme suit. Soit $\tilde{\alpha} = (\alpha_1, \ldots, \alpha_n)$ une séquence arbitraire de variables de type tel que $\mathrm{Var}(\tau) - \mathrm{Var}(\Gamma) = \{\alpha_1, \ldots, \alpha_n\}$. La généralisation de τ dans Γ est alors $\mathrm{Gen}_\Gamma(\tau) = \forall\tilde{\alpha}.\tau$.

L'ensemble des instances d'un schéma de type $\sigma = \forall\tilde{\alpha}.\tau$, noté $\mathrm{Inst}(\sigma)$, est défini comme l'ensemble des types obtenus en substituant les variables de $\tilde{\alpha}$ dans τ (par des types dans l'ensemble correspondant $\mathcal{X}^{\langle 2\rangle}$, $\mathcal{X}^{\langle 1\rangle}$ ou$\mathcal{X}^{\langle 0\rangle}$) :

$$\mathrm{Inst}(\sigma) = \{\, \tau[\alpha_1 \leftarrow \tau_1, \ldots, \tau_1 \leftarrow \alpha_n] \,\} = \{\, \sigma\, \tau_1 \ldots \tau_n \,\}$$

Se donnant enfin un environnement de typage clos Γ_δ pour typer les constantes c, on peut définir les règles de typage, détaillées en table 1.

2.2.3 dénotations

Les détails de la sémantique dénotationnelle sont donnés dans [Bau09].

Cette sémantique est définie par induction sur les jugements de typage. En ce qui concerne les termes de types purs, les dénotations sont définies de manière

$$\frac{\Gamma_\delta(c) = \sigma \quad \tau \in \mathrm{Inst}(\sigma)}{\Gamma \vdash c : \tau} \qquad \frac{\Gamma(x) = \sigma \quad \tau \in \mathrm{Inst}(\sigma)}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma,\, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \mathrm{Gen}_\Gamma(\tau_1) \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : \tau_2}$$

$$\frac{\Gamma \vdash t_1 : T(\tau_1) \quad \Gamma, x : \tau_1 \vdash t_2 : T(\tau_2)}{\Gamma \vdash \text{let } x \Leftarrow t_1 \text{ in } t_2 : T(\tau_2)} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathrm{val}(t) : T(\tau)}$$

TAB. 1 – Règles de typage

classique, et seront notées : $[\![\Gamma \vdash t : \sigma]\!]\xi\ \gamma$ où ξ est un environnement de typage devant associer à chaque variable de type libre de σ un type clos ; et où γ est un environnement associant à chaque axiome de Γ une dénotation.

Les termes de type computationnel ont, eux, pour dénotation des fonctions prenant en argument un état $s \in S$, et retournant un nouvel état et une dénotation : $[\![\Gamma \vdash t : \mathbf{T}\tau]\!]\xi\ \gamma\ s$ est un couple (s', d), où $s' \in \mathbb{S}$ est le nouvel état, et $d \in [\![\tau]\!]$ une dénotation, l'ensemble des états étant

$$\mathbb{S} = \{0,1\}^{\mathbb{N}} \times \left(\sum_{\tau^{\langle 1 \rangle} \text{ clos}} [\![\tau^{\langle 1 \rangle}]\!] \right)^{(\mathbb{N})}$$

2.3 Les résultats sur le langage

Outre la définition de la sémantique, sont présentés dans [Bau09] quelques résultats utiles sur le langage. En premier lieu, la sûreté du typage. Est aussi développée la notion d'équivalence observationnelle, pour pouvoir démontrer les énoncés cryptographiques à l'aide de raisonnement équationnel. Enfin, on construit les relations logiques de Kripke, permettant d'étendre les résultats de Walder [BBU08] concernant les termes polymorphes.

2.4 notations

Définition 2.1 (programmes)**.** On appellera programme de type τ tous termes $P \in t$ typables en τ dans l'environnement vide, c'est à dire tel que $\emptyset \vdash P : \tau$. On notera $(\!|\tau|\!)$ l'ensemble des programmes de type τ (*i.e.* $(\!|\tau|\!) = \{P \in t / \emptyset \vdash P : \tau\}$).

On étend cette définition aux schémas de type : si $\sigma = \forall \tilde{\alpha}.\tau$, $(\!|\sigma|\!) = (\!|\tau|\!)$

Définition 2.2 (Résultat d'exécution)**.** Si P est un programme de type $\mathbf{T}\tau$ pour un type clos τ, $R \in [\![\tau]\!]$ une dénotation, et $r \in \{0,1\}^{\mathbb{N}}$ un ruban d'aléa, on définit la proposition $P \overset{r}{\rightsquigarrow} R$ par

$$\exists s \in \mathbb{S}, [\![P]\!]\ \emptyset\ \emptyset\ (r, \emptyset) = (s, R)$$

De plus, on notera $\Omega = \{0,1\}^{\mathbb{N}}$ l'ensemble des rubans d'aléa.

3 Les énoncés cryptographiques dans notre langage

3.1 Critères cryptographiques

3.1.1 Définition

Pour avoir une formalisation la plus uniforme possible, nous souhaitons utiliser au maximum l'utilisation du langage dans les énoncés. En particulier, même les jeux de sécurités seront définis par des programmes de notre langage plutôt que par une définition descriptive. Ces programmes seront appelés critères.

Un critère, notion inspirée par [Lep05], est un programme qui génère un couple (o, t) après une phase d'initialisation éventuelle,où o désigne les oracles auxquels a droit l'attaquant, et t le test de victoire de l'attaquant (fonction retournant un booléen).

Pour plus de lisibilité on définira tout au long du rapport des alias de type.
alias : (α, β)-criterion $= T(\alpha \times (\beta \to T\,bool))$

Pour chaque critère, on peut définir l'avantage d'un attaquant, à l'aide des programmes en Table 2.

```
1  let apply_crit crit att =
2    let (oracle,test) <= crit in   génération de l'oracle et du test
3    let res <= (att oracle)        exécution de l'attaquant
4    in test res;;                  test du résultat
```

apply_crit : $\forall\, \alpha\, \beta\, .(\alpha, \beta)$-criterion $\to (\alpha \to T\beta) \to T\,bool$

```
1  let no_oracles crit =
2    let (oracle,test) <= crit in
3    val (unit,test);;
```

no_oracles : $\forall \alpha\, \beta\, .(\alpha, \beta)$-criterion $\to (unit, \beta)$-criterion

Tab. 2 – Application de critère et retrait des oracles d'un critère

Définition 3.1 (Avantage)**.** L'avantage d'un attaquant att $\in (\!|\alpha \to \mathbf{T}\beta|\!)$ sur un critère crit de type (α, β)-criterion est défini par

$$adv_{\text{crit}}(\text{att}) = \mathbb{P}_{r\in\Omega}\big[\,\text{apply\_crit crit att} \overset{r}{\rightsquigarrow} \text{true}\,\big] - m_{\text{crit}}$$

où $m_{\text{crit}} = \sup_{\text{att'}\in(\!|\text{U}\to\mathbf{T}\beta|\!)} \mathbb{P}_{r\in\Omega}\big[\,\text{apply\_crit (no\_oracles crit) att'} \overset{r}{\rightsquigarrow} \text{true}\,\big]$

Exemple 3.2 (Définition du critère *existential forgery* pour un schéma de signature)**.** Pour définir le critère, on profite au maximum de la modularité des langages fonctionnels : on utilise quelques fonctions annexes. Vue ainsi, la définition du critère peut sembler un peu verbeuse, mais les fonctions annexes en question sont simples et réutilisables pour d'autres critères ou réductions.

Définition informelle : au début du jeu, on génère un couple (pk, sk) clé-publique/clé-privée selon le programme gen. On donne à l'attaquant la clé publique pk, ainsi qu'un accès à un oracle de signature, qui signe les requêtes avec la clé sk. L'attaquant est éventuellement limité à un certain nombre n d'appels à l'oracle. L'attaquant gagne s'il arrive à générer un couple (m, s) message/signature, tel que la signature soit valide selon pk.

```
let logging f =
  let l <= ref nil in
  val(couple
      (fun x -> let ll<= !l in l:=cons x ll; f x )
      (!l)
     );;
```

logging : $\forall\, \alpha\ \beta.\ (\alpha \to T\beta) \to T\,((\alpha \to T\beta) \times T\,(\alpha\ List))$

```
let limit_calls n f =
  let m <= ref n in
  val(fun x ->
       let m1<= !m in
       if (m1 = 0) then exit
       else begin
         m := (m1-1);
         f x
       end
     );;
```

limit_calls : $\forall\, \alpha\ \beta.\ int \to (\alpha \to T\beta) \to T\,(\alpha \to T\beta)$

TAB. 3 – Enregistrement des appels, et limitation du nombre d'appels

```
let crit_ex_forgery n sign_scheme =
  let (gen,sign,verif) = sign_scheme in              séparation des fonctions
  let (pk,sk)<= gen in                               génération des clés
  let (sign_oracle,log) <= logging (sign sk) in      enregistrement des appels
  let sign_oracle2 <= limit_calls n sign_oracle in   et limitation de l'oracle de signature
  let test signedmessage =                           construction du test
    let (message,signature) = signedmessage in       séparation du message/signature
    let b1 <= verif pk message signature in          vérification de la signature
    let l <= log in                                  récupération du registre de l'oracle
    val (b1 && (not (test_in message l))) in         vérification d'absence dans le registre
  val ((sign_oracle2,pk),test);;
```

crit_ex_forgery : $int \to (\alpha_{pk}, \alpha_{sk}, \alpha_m, \alpha_s)$-signscheme)
$\to ((\alpha_m \to \mathbf{T}\alpha_s) \times \alpha_{pk}\ , \alpha_m \times \alpha_s)$-criterion

TAB. 4 – critère d'usurpation existentielle

Définissons quelques primitives nécessaires pour la suite, en Tables 3.2. Les deux fonctions que nous venons de définir remplissent des rôles simples mais essentielles pour expliciter les algorithmes cryptographiques. Utiliser la modularité de cette façon permet de concilier les impératifs formels (sémantique bien définie) avec les objectifs de concision et d'intuition. De plus, l'approche médullaire pourra permettre de simplifier les preuves de correction dans un assistant de preuve, en associant à ce genre de primitives des lemmes sur leur sémantique et sur la sémantique de certaines de leurs combinaisons.

La définition du critère suivante illustre l'utilisation de ces primitives :

(**alias :** $(\alpha_{pk}, \alpha_{sk}, \alpha_m, \alpha_s)$-signscheme $= T(\alpha_{pk} \times \alpha_{sk}) \times (\alpha_{sk} \to \alpha_m \to \mathbf{T}\alpha_s) \times (\alpha_{pk} \to \alpha_m \to \alpha_s \to T\,bool)$)

3.1.2 Combinaison de critères

Il arrive souvent qu'une preuve de sécurité ne s'appuie non pas sur une mais sur plusieurs hypothèses algorithmiques. La preuve de sécurité contient alors non pas une mais plusieurs réductions (typiquement, les arguments dits hybrides). Dans ces situations, l'énoncé de sécurité prend la forme (supposons

ici qu'il y a seulement deux réductions) : $Adv^S_C(att) \leq Adv_{P_1}(red_1\ att) + Adv_{P_2}(red_2\ att)$

Lorsqu'on travaille avec des critères d'indistinguabilité, on ne peut généralement pas faire mieux. Par contre, si la fonction de test d'un des deux critères ne contient aucun secret, il est possible de savoir sur chaque instance laquelle des deux hypothèses a été mise en défaut. On peut dans ce cas énoncer le résultat sous une forme plus forte. Cela présente de plus l'avantage de ne devoir construire qu'une seule réduction et donc d'éviter d'éventuelles duplications de code.

On définit la combinaison de critère de la manière suivante :

```
1  let combine_crit crit1 crit2 =
2    let (o1,v1) <= crit1 in
3    let (o2,v2) <= crit2 in
4    val (
5     (o1,o2),
6     fun x -> match Sum x with
7       |left y -> v1 y |right y ->v2 y
8       endmatch);;
```

combine_crit : $\forall \alpha\ \beta\ \gamma\ \delta\ .(\alpha, \beta)$-criterion $\rightarrow (\gamma, \delta)$-criterion
$\rightarrow (\alpha \times \gamma, \beta + \delta)$-criterion

TAB. 5 – combinaison de critère

Un attaquant sur une combinaison de deux critères peut à chaque instance "jouer" avec les deux critères (faire appel aux oracles des deux critères), mais à la fin n'en résoudre qu'un des deux (en annonçant lequel). On modélise ici ce choix par un type somme.

Proposition 3.3 (Sécurité d'une combinaison de critères)**.** *Pour tous types* $\tau_\alpha, \tau_\beta, \tau_\gamma, \tau_\delta$*, pour tous programmes* $\mathrm{crit}_1 \in (\!|(\tau_\alpha, \tau_\beta)\text{-}criterion|\!), \mathrm{crit}_2 \in (\!|(\tau_\gamma, \tau_\delta)\text{-}criterion|\!)$, $\mathrm{att} \in (\!|(\tau_\alpha \times \tau_\gamma) \rightarrow (\tau_\beta + \tau_\delta)|\!)$ *on a :*

$$adv_{\mathrm{crit}}(\mathrm{att}) + m_{\mathrm{crit}} \leq adv_{\mathrm{crit}_1}(\mathrm{red\_combine\_left}\ \mathrm{crit}_2\ \mathrm{att}) + m_{\mathrm{crit}_1} + adv_{\mathrm{crit}_2}(\mathrm{red\_combine\_right}\ \mathrm{crit}_1\ \mathrm{att}) + m_{\mathrm{crit}_2}$$

où $\mathrm{crit} = \mathrm{combine\_crit}\ \mathrm{crit}_1\ \mathrm{crit}_2$*, (*red_combine_left *et* red_combine_right *sont définis en annexe, Table 11).*

Exemple 3.4 (Usurpation existentielle ou collision)**.** Dans la preuve de sécurité du schéma hash-then-sign, on s'appuie sur deux hypothèses : la sécurité du schéma de signature sous-jacent, et la résistance aux collisions de la fonction de hachage choisie. Habituellement, pour énoncer une hypothèse de résistance aux collisions, on l'énonce avec une fonction de hachage à clé : l'attaquant doit être capable de construire une collision, étant donnée une clé choisie aléatoirement. Sans clé, il existe toujours un algorithme renvoyant une collision, car mathématiquement, il existe une telle collision, bien qu'on ne la connaisse pas. L'idée de Rogaway, dans [Rog06], est d'utiliser un énoncé constructif, pour affirmer que si l'on connaissait un attaquant sur le protocole, alors on connaîtrait (et non pas "il existerait") une collision.

En utilisant ce paradigme on peut alors énoncer le critère de résistance aux collisions très simplement :

```
let crit_collision hash =
   val(unit,
     (match_Couple
       (fun m1 m2 ->
          let h1 <= hash m1 in
          let h2 <= hash m2 in
          val ((not(m1 = m2)) && (h1 = h2))
       )));;
```

$\forall\ \alpha_m\ \alpha_d.\ (\alpha_m \to T\alpha_d) \to (unit, \alpha_m \times \alpha_m)$-criterion

```
let crit_ex_forgery_or_collision n sc h =
combine_crit (crit_ex_forgery n sc) (crit_collision h);;
```

crit_ex_forgery_or_collision : $\forall \alpha_{pk}\ \alpha_{sk}\ \alpha_d\ \alpha_m\ \alpha_s\ .int \to (\alpha_{pk}, \alpha_{sk}, \alpha_d, \alpha_s)$-signscheme $\to (\alpha_m \to T\alpha_d) \to (((\alpha_m \to \mathbf{T}\alpha_s) \times \alpha_{pk}) \times unit, (\alpha_d \times \alpha_s) + (\alpha_m \times \alpha_m))$-criterion

TAB. 6 – Critère de collision, et combinaison avec le critère d'usurpation

3.2 Construction et réduction

Le schéma hash-then-signse base sur un schéma de signature S de domaine D fini (typiquement $D = \{0,1\}^n$), et une fonction de hachage $h : D' \to D$, ou D' est un ensemble plus grand (idéalement infini, $D' = \{0,1\}^*$). La construction est décrite en Table 3.2 De façon informelle, il s'agit juste d'appliquer la fonction de hachage sur le message avant de le signer, ou avant de vérifier la signature.

On prouve alors la sécurité de ce schéma S' contre les attaques en *existential forgery*, en supposant que S est lui-même résistant à ce type d'attaque, et que la fonction de hachage h est résistante aux collisions. On arrive dans le cas présent à construire une seule réduction (donnée en Table 3.2) qui résout la combinaison des deux critères, plutôt que deux réductions séparées.

La preuve classique du schéma hash-then-sign se traduit alors, à l'aide des programmes définis en Table 3.2, dans notre formalisme par la proposition suivante :

Proposition 3.5 (Correction de la réduction pour hash-then-sign)**.** *Pour tous types $\tau_{sk}, \tau_{pk}, \tau_m, \tau_d, \tau_s$, pour tout schéma de signature* sc $\in [\![(\tau_{pk}, \tau_{sk}, \tau_d, \tau_s)$*-signscheme*$]\!]^{\emptyset}$*, fonction de hachage* h $\in [\![\tau_m \to \mathbf{T}\alpha_s]\!]^{\emptyset}$*, entier* n $\in \mathbb{N}$ *et tout attaquant* att $\in [\![\tau_m \to \mathbf{T}\alpha_s]\!]^{\emptyset}$*, et enfin tout ruban d'aléa $\omega \in \Omega$, on a :*

theorem_premise sc h n att $\overset{\omega,\emptyset}{\rightsquigarrow}$ true $\Rightarrow$ theorem_conclusion sc h n att $\overset{\omega,\emptyset}{\rightsquigarrow}$ true

Notons que dans ce cas particulier, nous avons un énoncé plus fort que nécessaire, en ce qui concerne l'aléa. Cela se traduit en terme d'avantage par l'énoncé suivant (avec les mêmes quantifications) :

$adv_{\text{crit\_ex\_forgery n (hash\_then\_sign sc)}}(\text{att}) \leq adv_{\text{crit\_ex\_forgery\_or\_collision n sc h}}(\text{reduction h att})$

Il faudrait cependant ajouter un autre fait pour affirmer la sécurité du schéma hash-then-sign, concernant les temps d'exécution de la réduction.

```
let hash_then_sign hash sign_scheme =
  let (gen,sign,verif) = sign_scheme in              séparation des fonctions
  ( gen,                                             fonction de génération (identique)
    (fun sk x -> let y<= hash x in sign sk y),       fonction de signature
    (fun pk x s -> let y<= hash x in  verif pk y s) fonction de vérification
  );;
```

hash_then_sign : $\forall \alpha_{pk}\ \alpha_{sk}\ \alpha_d\ \alpha_m\ \alpha_s\ .(\alpha_{pk}, \alpha_{sk}, \alpha_d, \alpha_s)$-signscheme $\to (\alpha_m \to T\alpha_d) \to (\alpha_{pk}, \alpha_{sk}, \alpha_m, \alpha_s)$-signscheme

```
let reduction hash att oracles =
  let ((sign_oracle,pk),dummy) = oracles in           séparation des oracles
  let sign_oracle_h x =                               construction de l'oracle de signature
    let d <= hash x in sign_oracle d in
  let (sign_oracle2,log) <= logging sign_oracle_h in  enregistrement des appels à l'oracle
  let (m,s) <= att (sign_oracle2,pk) in               exécution de l'attaquant
  let d <= hash m in                                  calcul du haché du message
  let l <= log in                            récupération du registre de l'oracle de signature
  let r <= find_antecedant hash d l in       recherche d'un antécédent du haché dans le registre
  match Option r with
    |some m2 -> val(right (m,m2))            s'il y en à un, répondre la collision
    |none -> val(left (d,s))                 sinon, répondre l'usurpation de signature
  endmatch;;
```

reduction : $\forall \alpha_{pk}\ \alpha_{sk}\ \alpha_d\ \alpha_m\ \alpha_s\ .(\alpha_m \to \alpha_d) \to (((\alpha_m \to \mathbf{T}\alpha_s) \times \alpha_{pk}) \to T(\alpha_m \times \alpha_s)) \to (((\alpha_d \to \mathbf{T}\alpha_s) \times \alpha_{pk}) \times unit) \to T((\alpha_d \times \alpha_s) + (\alpha_m \times \alpha_m))$

```
let theorem_premise sc hash n att =
  let sc2 = hash_then_sign hash sc in
  apply_crit (crit_ex_forgery n sc2) att;;
```

theorem_premise : $\forall \alpha_{pk}\ \alpha_{sk}\ \alpha_d\ \alpha_m\ \alpha_s\ .(\alpha_{pk}, \alpha_{sk}, \alpha_d, \alpha_s)$-signscheme $\to (\alpha_m \to \alpha_d) \to int \to (((\alpha_m \to \mathbf{T}\alpha_s) \times \alpha_{pk}) \to T(\alpha_m \times \alpha_s)) \to T\,bool$

```
let theorem_conclusion sc h n att =
  let att2 = reduction h att in
  apply_crit (crit_ex_forgery_or_collision n sc h) att2;;
```

theorem_conclusion : $\forall \alpha_{pk}\ \alpha_{sk}\ \alpha_d\ \alpha_m\ \alpha_s\ .(\alpha_{pk}, \alpha_{sk}, \alpha_d, \alpha_s)$-signscheme $\to (\alpha_m \to \alpha_d) \to int \to (((\alpha_m \to \mathbf{T}\alpha_s) \times \alpha_{pk}) \to T(\alpha_m \times \alpha_s)) \to T\,bool$

TAB. 7 – La construction hash-then-signet la réduction pour sa sécurité

4 Algébricité

Certaines preuves cryptographiques font l'hypothèse que l'attaquant (ou la réduction) n'a qu'un ensemble restreint d'opérations autorisées sur une certaine structure. Ces hypothèses sont utiles à la réduction (ou la méta-réduction), permettant de tracer les opérations faites sur certains objets, et ainsi d'extraire des informations.

C'est le cas par exemples des preuves dans les "modèles génériques" (groupe générique, anneau générique), utilisées en particulier pour prouver l'équivalence entre l'extraction de clé du chiffrement RSA et la factorisation [AM09].

Classiquement, ces modèles d'attaques sont présentés de la façon suivante : l'attaquant n'est pas directement relié au système qu'il attaque, mais à une machine à registres, qui garde les éléments de la structure. L'attaquant peut donner des ordres sur les opérations à effectuer, mais n'a pas d'accès direct à la valeur de ces registres. Ce modèle permet à la réduction d'extraire la séquence des opérations effectuées.

Nous proposons dans cette section une autre façon de formaliser la notion d'algébricité ; la propriété d'algébricité d'un algorithme sera exprimée en termes de typage, et nous montrerons une méthode générique pour extraire l'arbre de construction des éléments de la structure.

4.1 Reformulation de l'algébricité, utilisant le typage

Soit donc une structure, c'est à dire un ensemble E et des fonctions $\mathbf{f_1} \dots \mathbf{f_n}$.

Soit $\tau_G = \tau_G^{\langle 0 \rangle}$ le type testable canonique représentant la structure, et soient $\tau_1 \dots \tau_n$ des types où la variable de type $\alpha = \alpha^{\langle 1 \rangle}$ sert à marquer les entrées/sorties correspondant à des éléments de la structure pour les fonctions $\mathbf{f_1} \dots \mathbf{f_n}$. On se donne de plus des termes $f_i \in (\!|\tau_i[\alpha \leftarrow \tau_G]|\!)$ implémentant les fonctions $\mathbf{f_i}$. On définit api $= (f_1, \dots, f_n)$ ayant pour type $\mathcal{T}_{\text{api}}[\alpha \leftarrow \tau_G]$ ($\mathcal{T}_{\text{api}} = \prod \tau_i$). On supposera que les τ_i sont de la forme $a_i^1 \rightarrow \dots \rightarrow a_i^{n_i} \rightarrow b_i$, avec les a_i^j et les b_i des types simples (c'est à dire un type testable sans référence). Les types ayant cette forme seront appelés par la suite des types d'API.

Définition 4.1 (Programme Algébrique)**.** Soit $\mathcal{T}$ un type, Un programme $P \in (\!|\mathcal{T}[\alpha \leftarrow \tau_G]|\!)$ sera dit $(\mathcal{T}, \mathcal{T}_{\text{api}}, \text{api})$-Algébrique s'il existe un programme $P' \in (\!|\forall \alpha, \mathcal{T}_{\text{api}} \rightarrow \mathcal{T}|\!)$ tel que P' api $\equiv P$.

L'idée derrière cette définition est d'assurer que le programme P n'effectue aucune opération relatives aux éléments du groupe autre que celle de l'API.

Dans la suite, pour simplifier les énoncés nous supposerons que $\text{Var}(\mathcal{T}) \cup \text{Var}(\mathcal{T}_{\text{api}}) \subseteq \{\alpha\}$.

Exemple 4.2 (Modélisation du groupe générique)**.** Dans le modèle du groupe générique, l'attaquant ne peut appliquer que des opérations de groupes sur les éléments du groupe en question et le test d'égalité entre deux éléments : api $= (\text{prod}, \text{exp}, \text{eq})$ et $\mathcal{T}_{\text{api}} = (\alpha \rightarrow \alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{int} \rightarrow \alpha) \times (\alpha \rightarrow \alpha \rightarrow \text{bool})$.

Dans le modèle précédent, l'attaquant n'a même pas accès à la représentation des éléments du groupe qu'il manipule. Ce n'est pas toujours le cas : par exemple, la méta-réduction de Paillier-Vergnaud [PV05] autorise la réduction à utiliser la représentation des éléments, mais en restreignant tout de même la création de nouveaux éléments aux seules opérations de groupes. On prendra alors : api $= (\text{prod}, \text{exp}, \text{repr})$ et $\mathcal{T}_{\text{api}} = (\alpha \rightarrow \alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{int} \rightarrow \alpha) \times (\alpha \rightarrow \tau_G)$.

En plus d'assurer une limitation sur les opérations faites par le programme P sur les éléments de la structure, la propriété d' algébricité nous permet d'extraire l'arbre de construction des éléments "sortis" de P en fonction des éléments "entrés". Pour cela, nous allons nous servir du programme P', avec un type enrichi remplaçant τ_G, et une API truquée.

Définition 4.3 (Construction de l'API truquée)**.** Soit $\mathcal{T}_{\text{api}} = \tau_1 \times \dots \times \tau_n$ un type d'API tel que défini précédemment. On suppose l'existence d'un type

inductif τ_{Arbre} ayant pour constructeurs :

$$\{\text{Base} : \tau_G \to \tau_{Arbre}\} \cup \{F_i : \tau_i[\alpha \leftarrow \tau_{Arbre}]/b_i = \alpha\}$$

On définit la fonction exec $: \tau_{Arbre} \to \tau_G$, par induction primitive :

- Base $x \to x$
- $F_i\ x_i^1\ \ldots\ x_i^{n_i} \to f_i\ y_i^1\ \ldots\ y_i^{n_i}$ avec $y_i^j = \text{exec}\ x_i^j$ si $a_i^j = \alpha$, et $y_i^j = x_i^j$ sinon.

On définit aussi une nouvelle API api_arbre $= (f'_1, \ldots, f'_n)$ de type $\mathcal{T}_{\text{api}}[\tau_{Arbre}]$, par

- $f'_i = F_i$ si $b_i = \alpha$,
- $f'_i\ x_i^1\ \ldots\ x_i^{n_i} = f_i\ y_i^1\ \ldots\ y_i^{n_i}$ (avec les y_i^j définis comme précédemment).

Enfin, pour un type τ, on notera τ° pour $\tau[\alpha \leftarrow \tau_G]$ et τ^* pour $\tau[\alpha \leftarrow \tau_{Arbre}]$.

On cherche ensuite à exprimer de façon formelle l'idée suivante : en donnant à l'algorithme algébrique l'API truquée, si l'on traduit ses entrées $x \in \tau_G$ (correspondant à des éléments de la structure) par les arbres Base x, alors le programme algébrique aura pour sortie des arbres traduisant les mêmes éléments que s'il travaillait avec l'API normale et des entrées non traduites.

Cette idée se formalise à l'aide de relations logiques, plus précisément de relations logiques de Kripke.

On considère la relation suivante,$\mathcal{G}$ entre τ_G et τ_{Arbre}, définie par $(g, a) \in \mathcal{G}^\Delta$ ssi $g = \text{exec}\ a$.

Theorème 4.4. *On considère P un programme typé de $(\!|\mathcal{T}^\circ|\!)$ qui soit $(\mathcal{T}, \mathcal{T}_{api}, \text{api})$-Algébrique, et P' le programme associé, dans $(\!|\forall\alpha, \mathcal{T}_{api} \to \mathcal{T}|\!)$. Alors*

$$(\llbracket\emptyset \vdash P : \mathcal{T}^\circ\ \rrbracket\emptyset\ \emptyset)\ \ \mathcal{R}^\emptyset_{\mathcal{T}}[\alpha \to \mathcal{G}]\ \ \left(\llbracket\emptyset \vdash (P'\ \text{api\_arbre}) : \mathcal{T}^*\ \rrbracket\emptyset\ \emptyset\right)$$

Ce théorème n'est cependant pas suffisamment fort pour assurer que l'algorithme extract soit correct : nous voulons aussi nous assurer que les arbres sortis par $P'' = (P'\,\text{api\_arbre})$ ne contiennent que des bases parmi celles contenues dans les arbres qu'il a reçus. Cette nouvelle propriété s'exprime encore à l'aide de relations logiques. Pour cela on définit d'abord une nouvelle fonction : exec_verif $: \tau_G\text{-List} \to \tau_{Arbre} \to \tau_G\text{-option}$, où exec_verif ℓ est défini par induction primitive :

- *Base* $x \to \text{if}(x \in \ell)\,\text{then}\ \ \textit{some}\ x\ \ \text{else}\ \ \textit{none}$
- $F_i\ x_i^1\ \ldots\ x_i^{n_i} \to f_i^{\text{option}}\ y_i^1\ \ldots\ y_i^{n_i}$ avec $y_i^j = \text{exec\_verif}\ \ \ell\ x_i^j$ si $a_i^j = \alpha$, et $y_i^j = x_i^j$ sinon.

L'idée derrière cette définition est de caractériser les arbres a dont l'ensemble des bases est contenu dans la liste ℓ. Si c'est le cas, on a exec_verif $\ell\ a = \textit{some}(\text{exec}\ a)$, sinon, exec_verif $\ell\ a = \textit{none}$.

Comme précédemment, on définit une relation, paramétrée par une liste $\ell : \mathcal{G}^\ell$ définie entre (τ_G-Option) et τ_{Arbre}, par $(g, a) \in \mathcal{G}^\ell \Leftrightarrow \text{exec\_verif}\ \ \ell\ a = \textit{Some}\ g$.

Theorème 4.5. *On considère P un programme typé de $(\!|\mathcal{T}^\circ|\!)$ qui soit $(\mathcal{T}, \mathcal{T}_{api}, \text{api})$-Algébrique, et P' le programme associé, dans $(\!|\forall\alpha, \mathcal{T}_{api} \to \mathcal{T}|\!)$. Alors, pour tout ℓ on a :*

$$(\llbracket\emptyset \vdash P : \mathcal{T}^\circ\ \rrbracket\emptyset\ \emptyset)\ \ \mathcal{R}^\emptyset_{\mathcal{T}}[\alpha \to \mathcal{G}]\ \ \left(\llbracket\emptyset \vdash (P'\ \text{api\_arbre}) : \mathcal{T}^*\ \rrbracket\emptyset\ \emptyset\right)$$

La preuve du théorème sera donnée en annexe C.1.

4.2 Application : meta-réduction de Paillier-Vergnaud

4.2.1 La version originale

L'article de Paillier et Vergnaud [PV05] démontre que, dans le modèle standard (*i.e.* sans oracle aléatoire), la sécurité d'un certain nombre de schémas de signature basés sur le problème du logarithme discret n'est justement pas équivalente au logarithme discret. Nous nous intéressons ici au premier exemple de l'article : l'impossibilité de prouver la sécurité du schéma de Schnorr. Plus précisément, Paillier et Vergnaud prouvent que s'il existait une réduction capable d'utiliser un attaquant sur le schéma de Schnorr pour résoudre le problème du logarithme discret (noté DL), alors une telle réduction pourrait être utilisée pour résoudre un autre problème supposé difficile : le "one more discret log" (noté n-DL).

Le schéma de signature de Schnorr sera détaillé en annexe D.1.

4.2.2 Hypothèse d'algébricité

La notion d'algébricité est présentée sous deux formes dans l'article de Paillier et Vergnaud. La première, informelle, explique qu'une réduction est algébrique relativement à un groupe τ_G si elle n' applique que des opérations de groupe sur les éléments du groupe. Comme expliqué en introduction de la section 4, dans le formalisme usuel (machine de Turing), il faut se placer dans un modèle spécial pour énoncer proprement ce genre de propriété.

La seconde suppose l'existence d'un algorithme extract associé à la réduction $\mathcal{R}$, capable de décomposer tous les éléments de τ_G sortis de $\mathcal{R}$ sur la base des éléments entrés, c'est-à-dire que si durant son exécution $\mathcal{R}$ a reçu au plus les éléments $g_1 \dots g_n \in \tau_G$, et envoie à un oracle un élément $h \in \tau_G$, alors $\text{extract}(h, [g_1 \dots g_n])$ répond une liste $x_1 \dots x_n$ telle que $h = \prod_{i=1}^{n} g_i^{x_i}$.

L'ensemble défini par la première notion est inclus dans l'ensemble de la seconde, il n'est cependant pas clair que l'inclusion soit stricte. Pour notre approche constructive, nous préférerons donc la première qui a l'avantage de ne pas cacher que la méta-réduction se place dans un modèle particulier. L'algorithme extract est alors construit en utilisant la technique décrite en début de section. Il serait cependant aussi possible dans notre formalisme d'utiliser la seconde, en mettant l'algorithme extract en argument supplémentaire à la méta-réduction. La justification d'une telle hypothèse sur une réduction sera discutée en annexe D.2.

La technique proposée en section 4.1 pose cepedant un problème technique, relatif à l'éfficacité de l'extraction, problème qui sera discuté et résolu en annexe

Pour cette section, le type d'API sera $\mathcal{T}_{\text{api}} = (\alpha \to \alpha \to \alpha) \times (\alpha \to \text{int} \to \alpha) \times (\alpha \to \tau)$, où τ est un type permettant de représenter les éléments du groupe. Les trois fonctions sont respectivement : le produit du groupe, l'exponentiation, et la fonction de représentation.

On utilisera par la suite plutôt l'alias de type suivant :
alias : (α, β)-Group_ api $= \mathcal{T}_{\text{api}} = (\alpha \to \alpha \to \alpha) \times (\alpha \to \text{int} \to \alpha) \times (\alpha \to$

β). De plus, plutôt que de placer l'API en premier argument des différents programmes algébriques, nous l'inclurons dans les oracles, pour rester compatible avec le formalisme général utilisé pour les critères.

4.2.3 Les critères

Les critères que nous allons énoncer ici supposent algébricité de l'algorithme challenger. Cela semble nécessaire pour pouvoir écrire le critère sous la forme d'un programme efficace : en effet, pour le critère sur les réductions d'un attaquant vers le logarithme discret, il faut construire un attaquant sur le schéma de signature. Pour le critère "one more discret log", c'est aussi le cas, puisqu'il faut fournir au challenger un oracle calculant le logarithme discret.

Pour le premier critère, cela ne pose aucun problème, puisque la méta-réduction fait la même hypothèse d' algébricité sur la réduction. Le cas du second critère est plus intéressant, car nous verrons que la méta-réduction est elle-même algébrique, ainsi le programme construit en appliquant la méta-réduction à la réduction est aussi algébrique, nous pourrons donc sans problème lui appliquer le second critère.

Critère de réduction On définit l'alias de type correspondant aux attaquants essayant de forger une signature pour message donné, n'ayant recours qu'à la clé publique (Universal Forgery, Key Only Attack ou *UF-KOA*).
alias : $(\alpha, \alpha_m, \alpha_r)$-UF_ KOA_ att $= (\alpha \times \alpha_m \times \alpha_r) \to \mathbf{T}(\text{int} \times \text{int})$.

Les trois arguments correspondent respectivement à la clé publique, au message et au ruban d'aléa donnés à l'attaquant, qui est alors censé produire une signature valide pour ce message et cette clé publique. Paillier et Vergnaud autorisent la réduction à contrôler le ruban d'aléa de l'attaquant par ce troisième argument ; ce point sera discuté en annexe D.3.

Notons qu'en ne donnant pas accès à un oracle de signature à l'attaquant, on restreint la classe des attaques, c'est-à-dire qu'on *élargit* la classe des réductions concernées par le résultat d'impossibilité.

Le critère permettant d'énoncer la correction d'une réduction de *UF-KOA* vers *DL* nécessite tout d'abord de construire un attaquant *UF-KOA*.

```
1  let pseudo_att_crit q api hash g x =
2    let (prod,expo,repr) = api in        séparation de l'API
3    let (y,m,omega) = x in               séparation des arguments (clé publique, message, ruban)
4    let sk = get_expo g y in             extraction de la clé secrète associée
5    let k <= rand_int q in               tirage d'un exposant
6    let r = expo (extr_base g) k in      !
7    let c = hash (m,repr2 r) in          ! calcul de la signature
8    val(((k+(c*sk)) mod q),c);;          !
```

pseudo_att : $\forall\alpha\ \alpha_g\alpha_m\ \alpha_r$ .int $\to$ (α-Extr, α_g)-Group_ api $\to$ ($\alpha_m \times \alpha_g \to$ int) $\to$ α-Extr $\to$ (α-Extr, α_m, α_r)-UF_ KOA_ att

Tab. 8 – Attaquant *UF-KOA*

On se donne aussi un alias correspondant aux paramètres donnés à une réduction de *UF-KOA* vers *DL*.

alias : $(\alpha, \alpha_g, \alpha_m, \alpha_r)$-red_ inst =
$((\alpha, \alpha_g)$-Group_ api $\times$ $(\alpha, \alpha_m, \alpha_r)$-UF_ KOA_ att$) \times \alpha \times \alpha$.

Le critère peut alors se définir par :

```
1  let crit_red_n_UF_KOA_to_DL q n api hash g =
2    let api2 = extractable_api api in   construction de l'API truquée
3    let (prod2,expo2,repr2) = api2 in   séparation de l'API truquée
4    let g2 = extr_base g in             traduction de l'élément générateur
5    let skey <= rand_int q in           tirage d'un exposant
6    let h = expo2 g2 skey in            calcul du challenge DL
7    let att <= no_recall (              construction de l'attaquant, sans rappel
8               pseudo_att_crit q api2 hash g2) in
9    let att2 <= limit_calls n att in    limitation du nombre d'appels à l'attaquant
10     val(((api2,att2),g2,h), (fun x -> val(x=skey)));;
```

crit_red_n_UF_to_DL : $\forall\alpha\ \alpha_g\ \alpha_m\ \alpha_r$ .int $\rightarrow$ (α, α_g)-Group_ api
$\rightarrow (\alpha_m \times \alpha_g \rightarrow \text{int}) \rightarrow \alpha \rightarrow \mathbf{T}((\alpha\text{-Extr}, \alpha_g, \alpha_m, \alpha_r)\text{-red\_ inst} \times (\text{int} \rightarrow \mathbf{T}\text{bool}))$

TAB. 9 – Critère pour les réductions de *UF-KOA* vers *DL*

On donne à la réduction une API, un attaquant et deux éléments du groupe g et h, et la réduction doit renvoyer un entier x, le logarithme discret de h par rapport à g.

Les seuls éléments g2, h, typés α-Extr, donnés à la réduction, n'ont dans leur décomposition que l'élément g : α. Cela assure, si la réduction est algébrique, que l'attaquant pourra extraire correctement les logarithmes en base g (Table 4.2.3, ligne 4).

critère *n-DL*, ("one more discret-log") Le problème *n-DL* se définit informellement de la façon suivante : étant donné un générateur g du groupe, des éléments $h_1 \dots h_{n+1}$, et enfin un accès limité à n appels à un oracle calculant les logarithmes discrets en base g, calculer les logarithmes de tous les éléments $h_1 \dots h_{n+1}$ en base g.

On définit un nouvel alias, pour les instances du problème *n-DL* :
alias : (α, β)-DL_ inst = (α, β)-Group_ api $\times$ $(\alpha \rightarrow \mathbf{T}\text{int})) \times \alpha \times \alpha$-List).
Le critère peut être implémenté comme suit :

```
1  let crit_n_DL q n api g =
2    let api2 = extractable_api api in     construction d'une API truquée
3    let (prod,expo,repr) = api in         séparation de l'API
4    let (prod2,expo2,repr2) = api2 in     séparation de l'API truquée
5    let g2 = extr_base g in               traduction de l'élément générateur
6    let h_list <= gen_list (let r<= (rand_int q) in val(expo2 g2 r)) (n+1) in
7                                          génération des n+1 challenges
8    let dl_oracle <= limit_calls n (fun h -> val(get_expo g h)) in
9                                          construction de l'oracle DL
10   let verif_dl = verif_2_lists (fun a e -> let (y,yl)=a in ((expo g e) = y)) in
11                                         fonction de vérification des DL
12   val(((api2,dl_oracle),g2,h_list),fun l -> val(verif_dl h_list l));;
```

crit_n_DL : $\forall\alpha\beta$. int $\rightarrow$ int $\rightarrow$ $((\alpha, \beta))$-Group_ api $\rightarrow$ α $\rightarrow$
$\mathbf{T}((\alpha\text{-Extr}, \beta)\text{-DL\_ inst} \times (\text{int-List} \rightarrow \mathbf{T}\text{bool}))$

TAB. 10 – critère *n-DL*

On donne à la réduction une API, un oracle *DL*, un élément g, et une liste l de $n+1$ éléments. La réduction doit renvoyer une liste de $n+1$ entiers :

les logarithmes discrets des élément de la liste l par rapport à g. L'oracle *DL* construit pour le critère *n-DL* (Table 4.2.3, ligne 8) extrait correctement les logarithmes en base g si l'algorithme challenger est algébrique, car g2 et les éléments de h_list n'ont dans leur décomposition que le générateur g.

4.2.4 Méta-réduction et énoncé

Nous pouvons maintenant construire la méta-réduction, prenant en argument une réduction de *UF-KOA* vers *DL*, et les oracles du critère *n-DL*. Le programme de la méta-réduction, ainsi que les programmes aidant à énoncer le résultat de Paillier et Vergnaud sont donnés en annexe D.4.

Le résultat de Paillier et Vergnaud s'énonce alors de manière constructive.

Proposition 4.6 (Séparation entre la sécurité du Schéma de Schnorr et *DL*)**.** *Pour tous types* $\tau, \tau_g, \tau_m, \tau_r$*, Pour tout entier* $\mathrm{n} \in \mathbb{N}$*, pour toute API de groupe (*i.e. *implémentant les lois d'un groupe G)* $\mathrm{api} \in (\!|(\tau, \tau_g)\text{-}Group\_\ api|\!)$*,* $\mathrm{q} \in \mathbb{N}$ *l'ordre du groupe G et* $\mathrm{g} \in (\!|\tau|\!)$ *représentant un générateur du groupe G, pour toute fonction de hachage* $\mathrm{hash} \in (\!|(\tau_m \times \tau_g) \to int|\!)$ *et enfin, pour toute réduction* $\mathrm{red} \in (\!|\forall\alpha.(\alpha, \tau_g, \tau_m, \tau_r)\text{-}red\_\ inst \to \mathbf{T}int|\!)$ *on a :*

$$\mathbb{P}_{r\in\Omega}\left[\,\mathrm{theorem\_premise\ q\ n\ api\ hash\ g\ red} \overset{r}{\rightsquigarrow} \mathrm{true}\,\right] \leq \mathbb{P}_{r\in\Omega}\left[\,\mathrm{theorem\_conclusion\ q\ n\ api\ hash\ g\ red} \overset{r}{\rightsquigarrow} \mathrm{true}\,\right]$$

On notera que dans les deux programmes énonçant la prémisse et la conclusion du théorème, la réduction n'est pas supposée avoir le même type. Mais l'hypothèse d'algébricité nous affirme que le type de la réduction est un schéma de type qui s'instancie correctement dans les deux cas.

5 Conclusion

Ces travaux montrent qu'il est effectivement possible d'exprimer les réductions cryptographiques dans un langage formel, sans s'éloigner de la description intuitive de ces réductions. On pourrait craindre que de telles définitions formelles soient fastidieuses, elles ne sont en fait pas plus longues qu'en utilisant une langue naturelle, et sont assurément sans ambiguïté.

Le typage se révèle être une aide précieuse lors de la rédaction de tels programmes cryptographiques ; et il offre un nouveau point de vue sur les modèles algébriques.

L'implémentation de la réduction accompagnant une preuve informelle apporte aussi un plus à la confiance que l'on peu accorder à un énoncé de sécurité, car il est possible de tester la réduction à l'aide de l'interpréteur, en prenant des paramètres de sécurités suffisamment petits pour qu'une attaque exhaustive s'exécute en temps raisonnable. Un tel test est implémenté pour le premier exemple (fin de l'annexe F.1).

La suite de ces travaux serait de développer un compilateur de notre langage vers un assistant de preuve, et de mettre au point des techniques adaptées aux preuves de correction de ces programmes. Il nous semble intéressant en

particulier de s'inspirer des techniques d'analyses statiques, par exemple en affinant le typage des effets de bords pour obtenir des résultats de commutation.

Il reste aussi quelques pistes que nous n'avons pas eu le temps d'explorer, notamment l'utilisation des types purs pour modéliser les limitations de certains attaquants, ainsi que la formalisation des techniques de re-jeu. Nous détaillerons ces deux pistes, et pour finir nous nous interrogerons sur la complétude de notre approche.

Enfin, notre formalisme benificierait sûrement des notations issus de la programmation objet, pour eviter l'utilisation des n-uplets, qui permettrai par exemple d'écrire `sign_scheme.sign` ou encore `api.prod`, ce qui faciliterai l'implémentation et la lecture des programmes.

5.1 Utilisation des fonctions pures

Jusqu'à présent, dans les notions définies, les attaquants, les réductions et les oracles avaient droit aux effets de bord, pour pouvoir par exemple limiter le nombre d'appels à un oracle, garder un trace des appels fait à un certain oracle, ou encore d'implémenter un programme se comportant comme une fonction aléatoire.

Il y a cependant des cas où nous ne voulons pas autoriser de tels effets de bord, notamment parce qu'ils peuvent servir de canal de communication entre deux programmes.

Exemple 5.1 (Application des fonctions pure : Key Dependant Message)**.** Dans certaines applications, comme le chiffrement de disque dur, il arrive que les données chiffrées dépendent de la clé de chiffrement. Pour prouver la sécurité du chiffrement dans de telles situations, on donne à l'attaquant le choix de la dépendance définissant le message en fonction de la clé, sans pour autant donner la clé à l'attaquant. Dans les modèles classiques, l'attaquant donne donc la fonction, sous la forme d'un terme à interpréter, dans un langage à part.

Notre formalisme permettrait de se passer de l'interprétation ad-hoc de la fonction choisie par l'attaquant, puisque, contrairement au modèle des machines de Turing communiquantes, il offre intrinsèquement une notion de fonction. Nous voulons par contre cette fonction pure, sinon elle pourrait stocker la clé dans une référence partagée avec l'attaquant.

5.2 Technique de re-jeu, ou Forking

La technique dite de re-jeu, ou de forking, consiste à faire revenir l'attaquant à un certain état passé de son exécution, pour ensuite répondre différemment a ses requêtes.

Pour illustrer cette technique, imaginons un jeu vidéo ayant n niveaux, et un joueur capable de réussir chaque niveau avec un probabilité $1/2$. Pour gagner le joueur doit réussir consécutivement les n niveaux, sinon il doit recommencer. Il faudra alors en moyenne un temps 2^n pour gagner.

Cependant, en exécutant le jeu dans une machine virtuelle, le joueur peut gagner bien plus vite (en moyenne $2n$), en sauvegardant l'état de la machine

virtuelle à chaque fois qu'il réussi un niveau, et en chargeant la sauvegarde précédente quand il perd.

Une telle technique s'exprime bien avec les machines de Turing. Une formalisation naturelle en λ-calcul serait d'utiliser des continuations. Cependant, inclure des continuations dans notre langage soulève de sérieux problèmes techniques. Nous proposons en annexe E.

5.3 Complétude de l'approche constructive

L'argument cryptographique précédent,utilisant le re-jeu, n'a pas pu être directement formalisé dans notre langage, même si nous espérons pouvoir étendre notre langage pour le capturer. Cette difficulté soulève la question de la complétude de notre approche : peut-on exprimer et démontrer tous les énoncés cryptographiques de façon constructive ?

Backes et al., dans [BU08] répondent négativement à cette question. A l'aide d'une construction "Zero-knowledge", ils élaborent un protocole sûr sous certaines hypothèses, mais non prouvable par une réduction explicite reposant sur les mêmes hypothèses. Toutefois, ce type de construction ne semble pas être un obstacle à la démarche que nous proposons, pour les constructions cryptographiques habituelles. Il serait de plus intéressant de prouver constructivement la sécurité de ce type de protocole en utilisant des hypothèses plus fortes (*i.e.* un but plus facile), mais considérées comme équivalentes dans l'approche non-constructive.

Références

[AM09] Divesh Aggarwal and Ueli Maurer. Breaking rsa generically is equivalent to factoring. In *EUROCRYPT '09 : Proceedings of the 28th Annual International Conference on Advances in Cryptology*, pages 36–53, Berlin, Heidelberg, 2009. Springer-Verlag.

[Bau09] Mathieu Baudet. Definition of a language for cryptographic reductions (draft). `http ://www.lsv.ens-cachan.fr/~baudet/drafts/crypto_language_definition.pdf`, 2009.

[BBU08] Michael Backes, Matthias Berg, and Dominique Unruh. A formal language for cryptographic pseudocode. In *Proc. 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, volume 5330, pages 353–376, 2008.

[BGJB07] Gilles Barthe, Benjamin Grégoire, Romain Janvier, and Santiago Zanella Béguelin. A framework for language-based cryptographic proofs. In *Proc. 2nd Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*, Oct 2007.

[Bla06] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *Proc. 2006 IEEE Symposium on Security and Privacy*, pages 140–154. IEEE Computer Society Press, May 2006.

[BP06] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology – CRYPTO 2006*, volume 4117, pages 537–554, 2006.

[BR95] MWihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology – EUROCRYPT 1994*, volume 950, pages 92–111, 1995.

[BU08] Michael Backes and Dominique Unruh. Limits of constructive security proofs. In *ASIACRYPT '08 : Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security*, pages 290–307, Berlin, Heidelberg, 2008. Springer-Verlag.

[GGM86] O Goldreich, S Goldwasser, and S Micali. How to construct random functions. *Journal of the ACM*, (33) :792–807, 1986.

[Lep05] Benjamin Leperchey. *Sur la notion d'observation en sémantique*. PhD thesis, Université Paris 7 - Denis Diderot, 2005.

[PV05] Pascal Paillier and Damien Vergnaud. Discrete-log-based signatures may not be equivalent to discrete log. In *ASIACRYPT*, pages 1–20, 2005.

[Rog06] Phillip Rogaway. Formalizing human ignorance. In *Progress in Cryptology – VIETCRYPT 2006*, volume 4341, pages 211–228, 2006.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO '89 : Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 239–252, London, UK, 1990. Springer-Verlag.

[Wad89] Philip Wadler. Theorems for free ! In *Proc. 4th Int. Symposium on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359, 1989.

A Définition des types

$\tau^{\langle 0\rangle}, \tau_1^{\langle 0\rangle}, \ldots$	::=		types testable :$\mathcal{T}^{\langle 0\rangle}$
	|	$\alpha^{\langle 0\rangle}$	variable de type testable ($\alpha^{\langle 0\rangle} \in \mathcal{X}^{\langle 0\rangle}$)
	|	$\kappa(\alpha^{\langle 0\rangle}, \ldots, \alpha^{\langle 0\rangle})$	application de constructeur de type($\kappa \in \mathcal{K}$)
	|	U	type unit
	|	$R(\tau^{\langle 1\rangle})$	type référence

$\tau^{\langle 1\rangle}, \tau_1^{\langle 1\rangle}, \ldots$	::=		types purs : $\mathcal{T}^{\langle 1\rangle}$
	|	$\alpha^{\langle 1\rangle}$	variable de type pur($\alpha^{\langle 1\rangle} \in \mathcal{X}^{\langle 1\rangle}$)
	|	$\tau_1^{\langle 1\rangle} \to \tau_2^{\langle 1\rangle}$	type fonction
	|	$\kappa(\alpha_1^{\langle 1\rangle}, \ldots, \alpha_n^{\langle 1\rangle})$	application de constructeur de type
	|	U	type unit
	|	$R(\tau^{\langle 1\rangle})$	type référence

$\tau^{\langle 2\rangle}, \tau_1^{\langle 2\rangle}, \ldots$	::=		types généraux :$\mathcal{T}^{\langle 2\rangle}$
	|	$\alpha^{\langle 2\rangle}$	variable de type générale($\alpha^{\langle 2\rangle} \in \mathcal{X}^{\langle 2\rangle}$)
	|	$\tau_1^{\langle 2\rangle} \to \tau_2^{\langle 2\rangle}$	type fonction
	|	$\kappa(\alpha_1^{\langle 2\rangle}, \ldots, \alpha_n^{\langle 2\rangle})$	application de constructeur de type
	|	U	type unit
	|	$R(\tau^{\langle 1\rangle})$	type référence
	|	$T(\tau^{\langle 2\rangle})$	type computationnel

B Reductions pour les critères combinés

```
let reduction_combine_left crit_right att oleft =
  let (oright,vright) <= crit_right in
  let r <= att (oleft,oright) in
  match Sum r with
    |left rl -> val rl
    |right rr -> exit
    endmatch;;
```

$\forall\ \alpha\ \beta\ \gamma\ \delta\ \epsilon.\ (\gamma, \delta)\text{-criterion} \to ((\alpha \times \gamma) \to T(\beta + \delta)) \to \alpha \to T\beta$

```
let reduction_combine_right crit_left att oright =
  let (oleft,vleft) <= crit_left in
  let r <= att (oleft,oright) in
  match Sum r with
    |left rl -> exit
    |right rr -> val rr
    endmatch;;
```

$\forall\ \alpha\ \beta\ \gamma\ \delta\ \epsilon.\ (\alpha, \beta)\text{-criterion} \to ((\alpha \times \gamma) \to T(\beta + \delta)) \to \gamma \to T\delta$

Tab. 11 – Réductions pour les critères combinés

C Preuve du théorème 4.5

Theorème C.1. *On considère P un programme typé de $(\!|\mathcal{T}^\circ|\!)$ qui soit $(\mathcal{T}, \mathcal{T}_{api}, \text{api})$-Algébrique, et P' le programme associé, dans $(\!|\forall\alpha, \mathcal{T}_{api} \to \mathcal{T}|\!)$. Alors, pour tout ℓ on a :*

$$(\llbracket \emptyset \vdash P : \mathcal{T}^\circ \rrbracket \emptyset\ \emptyset)\ \ \mathcal{R}^{\emptyset}_{\mathcal{T}}[\alpha \to \mathcal{G}]\ \ (\llbracket \emptyset \vdash (P'\ \text{api\_arbre}) : \mathcal{T}^* \rrbracket \emptyset\ \emptyset)$$

Démonstration. Par définition de l'algébricité, nous avons $P \equiv P'$ api, il suffit donc de montrer que

$$(\llbracket \emptyset \vdash P'\ \text{api} : \mathcal{T}^\circ \rrbracket \emptyset\ \emptyset)\ \ \mathcal{R}^{\emptyset}_{\mathcal{T}}[\alpha \to \mathcal{G}]\ \ (\llbracket \emptyset \vdash (P'\ \text{api\_arbre}) : \mathcal{T}^* \rrbracket \emptyset\ \emptyset)$$

Le *fundamental lemma* dans [Bau09] nous assure que

$$(\llbracket \emptyset \vdash P'\ : \forall\alpha.\mathcal{T}_{\text{api}} \to \mathcal{T} \rrbracket \emptyset\ \emptyset)\ \ \mathcal{R}^{\emptyset}_{\forall\alpha.\mathcal{T}_{\text{api}} \to \mathcal{T}}[\emptyset]\ \ (\llbracket \emptyset \vdash P' : \forall\alpha.\mathcal{T}_{\text{api}} \to \mathcal{T} \rrbracket \emptyset\ \emptyset)$$

En particulier, en utilisant la paramétricité, pour $\xi_1 = (\alpha \to \tau_G)$ et $\xi_2 = (\alpha \to \tau_{Arbre})$ on a :

$$(\llbracket \emptyset \vdash P'\ : \mathcal{T}_{\text{api}} \to \mathcal{T} \rrbracket \xi_1\ \emptyset)\ \ \mathcal{R}^{\emptyset}_{\mathcal{T}_{\text{api}} \to \mathcal{T}}[\alpha \to \mathcal{G}^\ell]\ \ (\llbracket \emptyset \vdash P' : \mathcal{T}_{\text{api}} \to \mathcal{T} \rrbracket \xi_2\ \emptyset)$$

. Reste donc a prouver que

$$\left(\llbracket \emptyset \vdash \text{api} \ : \mathcal{T}^{\circ}_{\text{api}} \rrbracket \emptyset \ \emptyset\right) \ \ \mathcal{R}^{\emptyset}_{\mathcal{T}_{\text{api}}}[\alpha \to \mathcal{G}^{\ell}] \ \ \left(\llbracket \emptyset \vdash \text{api\_arbre} : \mathcal{T}^{*}_{\text{api}} \rrbracket \emptyset \ \emptyset\right)$$

c'est à dire que chacune des fonctions f'_i de l'API truquée api_arbre est en $(\mathcal{R}^{\emptyset}_{\mathcal{T}_i}[\alpha \to \mathcal{G}^{\ell}])$-relation avec f_i. On s'autorise pour la suite de confondre les termes et leur sémantique pour alleger les notations, ce qui ne présente pas de problème car les primitives concernées sont toutes pures.

Pour les indices i tel que $b_i = \alpha$ il faut prouver que si pour tout j tel que $a_i^j = \alpha$, exec_verif $\ell x^j = \text{some}\, z^j$, et que pour tout j tel que $a_i^j \neq \alpha$, $x^j = z^j$; alors exec_verif $\ell(f'_i x^1 \dots x^n) = \text{some}(f_i z^1 \dots z^n)$. Pour de tels i, f'_i est défini comme le constructeur F_i, ainsi

$$\text{exec\_verif}\ \ell\ (f'_i\ x^1 \dots x^n) = \text{exec\_verif}\ \ell\ (F_i\ x^1 \dots x^n) = f^{\text{option}}\ y^1 \dots y^n$$

avec $y^j = \text{exec\_verif}\ \ \ell\ x^j$ si $a_i^j = \alpha$, et $y^j = x^j$ sinon. Les hypothèses sur les x^j nous assure alors que $y^j = \text{some}\, z^j$ si $a_i^j = \alpha$, et que $y^j = z^j$ sinon. On en déduit que $f^{\text{option}}\ y^1 \dots y^n = \text{some}(f_i\ z^1 \dots z^n)$, et donc le résultat voulu.

Pour les autres indices i tel que $b_i \neq \alpha$, il faut montrer, sous la même hypothèse que $(f'_i x^1 \dots x^n) = (f_i z^1 \dots z^n)$, ce qui se vérifie de façon similaire.

□

D Compléments sur la méta-réduction

D.1 Le schéma de signature de Schnorr

Le schéma de signature de Schnorr [Sch90] s'appuie sur un groupe G, d'ordre premier q et sur une fonction de hachage h. Rtant donné g un générateur du groupe, le schéma est défini

- génération des clés : on tire aléatoirement la clé privée $x \in 0 \dots q-1$, la clé publique etant alors $y = g^x$.
- signature du message m avec la clé privée x : On tire aléatoirement $k \in 0 \dots q-1$. On calcul : $r = g^k$, $e = h(m||r)$ et $s = (ke - x) \mod q$. La signature est le couple (e, s)
- verification de la signature (e, s) pour le message m avec la clé publique y : on calcul $r' = g^s \cdot y^e$, et $e' = H(m||r')$. La signature est correcte si $e' = e$.

On peut bien entendu décrire ce schéma de signature dans notre langage (voir Table 12). Pour un typage plus claire on supposera que la fonction de hachage prend un couple d'argument, afin d'éviter de devoir utiliser la concaténation sur les chaînes de bit.

D.2 Justification de l'hypothèse d'algébricité

Dans le cas des preuves de sécurité, on préfère éviter d'avoir à utiliser des hypothèses d'algébricité, car les groupes construits pour la cryptographie peuvent avoir des propriétés spécifiques dont un attaquant pourrait prendre avantage. Ce genre d'hypothèse est cependant acceptable sur les réductions, pour prouver

```
let schnorr_scheme q api h g =
  let (prod,expo,repr) = api in
  let gen =
    let x <= rand_int q in
    let y = expo g x in
    val (x,y) in
  let sign x m =
    let k <= rand_int q in
    let r = expo g k in
    let e = h (m,repr r) in
    let s = (k * e - x) mod q in
    val (e,s) in
  let verif y m sig =
    let (e,s) = sig in
    let r2 = prod (expo g s) (expo y e) in
    let e2 = h (m,repr r2) in
    val (e=e2) in
  (gen,sign,verif);;
```

schnorr_scheme : $\forall\ \alpha\ \beta\ \alpha_m$.int $\rightarrow$ (α, β)-Group_ api $\rightarrow$ $(\alpha_m \times \beta \rightarrow \text{int})$ $\rightarrow \alpha \rightarrow (\alpha, \text{int}, \alpha_m, \text{int} \times \text{int})$-signscheme

TAB. 12 – Schéma de signature de Schnorr

un résultat de séparation. L'article de Paillier et Vergnaud [PV05] justifie l'hypothèse de la façon suivante : *This class of reductions is not overly restrictive (in fact, we do not know any example of a cryptographic reduction which is not algebraic).*

On peut cependant donner un argument plus fort pour justifier l'hypothèse d'algébricité pour une réduction : la correction de la réduction est quantifiée universellement sur tous les groupes vérifiant l'hypothèse algorithmique (ici la difficulté du logarithme discret). Toute implémentation d'une API de groupe définit un groupe, et bien que certains soient isomorphes, la réduction se doit de fonctionner quelle que soit l'implémentation de l'API. Il faut donc dans l'énoncé que la réduction prenne en argument cette API, quelle que soit son 'implémentation, et quelle que soit la représentations des éléments du groupe : d'où le polymorphisme.

D.2.1 Efficacité de l'extraction

La technique proposée en section 4.1 pour l'extraction pose un problème d'efficacité dans le cas des groupes (et dans les cas où la structure algébrique possède un opérateur au moins binaire). En effet, un algorithme appelant n fois la fonction produit de l'API, pourrait construire un arbre de taille exponentielle en n. Plus précisément, sa représentation en mémoire est en fait linéaire en n, car stockée implicitement comme un *DAG* ("Directed Acyclic Graph", c'est à dire un arbre avec partage), mais son temps de parcours est exponentiel. Une solution serait d'implémenter une API qui construit de façon explicite des *DAG*.

Dans ce cas particulier, on peut cependant faire plus simple : plutôt que de garder tout l'arbre de construction, on tient seulement à jour la décomposition des éléments du groupe selon les éléments de base.

On définit l'alias de type suivant : **alias :** α-Extr $= \alpha \times (\alpha \times \text{int})List$.

Si le type α représente un groupe τ_G, un objet $x = (h, [(g_1, x_1), \ldots, (g_n, x_n)])$

de type α-Extr, sera dit correct si $h = \prod_{i=1}^{n} g_i^{x_i}$.

On construit les éléments de base de la façon suivante :

```
let extr_base x = (x,cons (x,1) nil);;
```

extr_base : $\forall\alpha\ .\ \alpha \rightarrow \alpha$-Extr

De même, on définit la transformation d'API, les fonctions résultantes préservant la propriété de correction.

```
let extractable_api api =
  let (prod,expo,repr) = api in
  let extr_prod x y =
    let (xg,xl) = x in
    let (yg,yl) = y in
      (prod xg yg,mult_list_by_list xl yl) in
  let extr_expo x e =
    let (xg,xl) = x in
    (expo xg e, expo_list e xl) in
  let extr_repr x =
    let (xg,xl) = x in
      repr xg in
    (extr_prod,extr_expo,extr_repr);;
```

extractable_api : $\forall\alpha\ \beta.\ (\alpha,\beta)$-Group_ api $\rightarrow (\alpha$-Extr$,\beta)$-Group_ api

TAB. 13 – construction de l'API truquée

D.3 Maîtrise de l'aléa

Reprenons la définition du critère de réduction d'un attaquant *UF-KOA* vers le logarithme discret de la partie précédente. Parmi les arguments donnés à l'attaquant, on trouve un paramètre ω, qui est censé donner à la réduction le contrôle de l'aléa utilisé par l'attaquant. Cependant, ce contrôle est assez limité ; notamment, donner deux rubans corrélés n'assure aucunement que les résultats de l'attaquant soient corrélés. La seule propriété assurée est le caractère fonctionnel de l'attaquant, c'est-à-dire que l'appeler deux fois sur les mêmes arguments donnera le même résultat.

Dans la construction de notre attaquant, nous utilisons cependant de l'aléa externe, mais, encapsulé par no_recall, il garde son apparence fonctionnelle.

On pourrait ici simplifier la formalisation, car l'attaquant n'ayant recours à aucun oracle durant son calcul, la réduction ne peut pas faire de "forking". Utiliser deux fois le même ruban d'aléa est inutile, on peut donc enlever le contrôle de la réduction sur l'aléa de l'attaquant sans perdre de généralité.

D.4 La méta-réduction explicitée

```
let pseudo_att get_elem hash api dl_oracle =
  let (prod,expo,repr) = api in                  séparation des fonctions
  fun x -> let (y,m,omega) = x in                séparation des arguments
  let r <= get_elem in                           récupération d'un élément de la liste des challenges
  let c = hash (m,r) in
  let s <= dl_oracle (prod r (expo y c)) in      utilisation de l'oracle DL pour l'usurpation
    val(s,c)
  in
let meta_reduction hash red oracles =
  let ((api,dl_oracle),g,ylist) = oracles in          séparation des oracles
  let (prod,expo,repr) = api in                       et des fonctions de l'API
```

```
12  let api2 = extractable_api api in                    construction l'API truquée
13  let (prod2,expo2,repr2) = api2 in                    séparation de ses fonctions
14  let (get_elem,get_rest) <= gets_from_list ylist in   primitive d'accès séquentiel aux éléments de la liste
15  let hash2 sm = let (m,x) = sm in hash (m,repr2 x) in traduction de la fonction de hachage
16  let get_elem_extr = let y <= get_elem in
17                      val(extr_base y) in              traduction des éléments de la liste
18  let (att,get_log) <= no_recall_and_log (             construction de l'attaquant pour la réduction
19                        pseudo_att (get_elem_extr)     (enregistrant les appels)
20                          hash2 (extractable_api api)  (et fonctionnant avec l'API truquée)
21                          (extr_fun dl_oracle)) in
22  let r0 <= get_elem in                                création du challenge pour la réduction
23  let k0 <=  extractable_reduction red ((api,att),g,r0) in
24                                   exécution de la réduction (avec arguments truqués)
25  let log <= get_log in            récupération du registre de l'attaquant
26  let rest <= get_rest in          récupération des éléments de la liste non utilisés
27  let first_logarithms =list_map (extract_logarithm_from_log_line g r0 k0) log in
28                                   calcul des log discrets
29  let last_logarithms <=list_map_eval dl_oracle rest in
30                                   demande des log discrets restants à l'oracle DL
31  val(cons k0 (rev_append first_logarithms (rev last_logarithms)));;
32                                   réponse au challenge n-DL
```

meta_reduction : $\forall\alpha\ \alpha_g\ \alpha_m\ \alpha_r\ .((\alpha_m \times \alpha_g) \to \text{int}) \to ((\alpha\text{-Extr}, \alpha_g, \alpha_m, \alpha_r)\text{-red}_{\_}\ \text{inst} \to \mathbf{T}\text{int})$
$\to (\alpha\text{-Extr}, \alpha_g)\text{-DL}_{\_}\ \text{inst} \to \mathbf{T}(\text{int-List})$

TAB. 14 – Méta-Réduction de Paillier et Vergnaud

```
1  let theorem_premise q n api hash g red =
2    apply_crit (crit_red_n_UF_KOA_to_DL q n api hash g) red;;
```

theorem_premise : $\forall\alpha\ \alpha_g\ \alpha_m\ \alpha_r\ .\text{int} \to \text{int} \to (\alpha, \alpha_g)\text{-Group}_{\_}\ \text{api} \to ((\alpha_m \times \alpha_g) \to \text{int}) \to \alpha$
$\to ((\alpha\text{-Extr}, \alpha_g, \alpha_m, \alpha_r)\text{-red}_{\_}\ \text{inst} \to \mathbf{T}\text{int}) \to \mathbf{T}\text{bool}$

```
1  let theorem_conclusion q n api hash g red =
2    apply_crit (crit_n_DL q n api g) (meta_reduction hash red);;
```

theorem_conclusion : $\forall\alpha\ \alpha_g\ \alpha_m\ \alpha_r\ .\text{int} \to \text{int} \to (\alpha, \alpha_g)\text{-Group}_{\_}\ \text{api} \to ((\alpha_m \times \alpha_g) \to \text{int}) \to \alpha$
$\to ((\alpha\text{-Extr-Extr}, \alpha_g, \alpha_m, \alpha_r)\text{-red}_{\_}\ \text{inst} \to \mathbf{T}\text{int}) \to \mathbf{T}\text{bool}$

TAB. 15 – Prémisse et conclusion du théorème sur la méta-réduction

E Une piste vers la formalisation du Forking

Quitte à définir un oracle relais, on peut supposer que l'attaquant n'a accès qu'à un seul oracle. Nous cherchons à se donner la possibilité, dans la réduction, de faire "forker" le calcul de l'attaquant lorsqu'il appelle un oracle, c'est-à-dire le re-executer en lui répondant un autre résultat, mais nous voudrions aussi qu'il réagisse lors du deuxième calcul comme s'il s'agissait de la première exécution.

Cette propriété peut sembler simple à énoncer : il suffit que l'attaquant ait un type de la forme $\mathbf{T}(\tau_{oracle} \to \tau_{res})$ où τ_{res} est un type pur. Cependant, lors de la réduction nous voulons généralement que l' oracle donné à l'attaquant ait

```
let exec n init bigstep oracle =
  let s <= init in                          intialisation
  let result <=
    fold_int                                boucle n fois
    (val (bigstep (s,none)))                éxecution du premier morcau
    (fun i r -> let r_ev <= r in            récuperation du résultat du l'étape précédent
       match Sum r_ev with                  séparation : calcul terminé ou requête à un oracle
         |left x -> let (s,oarg) = x in     si c'est une requête
           let ores <= oracle oarg in       caculer le résultat de l'oracle
           val(bigstep (s,some ores))       et lancer l'étape suivante
         |right res -> val(r_ev)            sinon, ne rien faire
       endmatch)
    n
    in
    match Sum result with
      |left x -> exit          si l'on a pas atteint le résultat final, échouer
      |right x -> val x        sinon, répondre le résultat final
    endmatch;;
```

$$\text{exec} : \forall \alpha_S\ \alpha_{oarg}\ \alpha_{ores}\ \alpha_{res}.int \to \mathbf{T}\alpha_S \to (\tau_S \times (\tau_{ores}\text{-option})) \\ \to (\tau_S \times \tau_{oarg}) + \tau_{res}) \to (\alpha_{oarg} \to \mathbf{T}\alpha_{ores}) \to \alpha_{ores}$$

TAB. 16 – Éxecution normale du programme découpé

des effets de bord (pouvoir utiliser de l'aléa, ou enregistrer la liste des appels). Le type τ_{res} ne peut alors être pur, sauf si l'attaquant n'utilise pas son oracle.

Une des solutions consiste à supposer, étant donné un attaquant $A : (\tau_{oarg} \to \mathbf{T}\tau_{ores}) \to \mathbf{T}\tau_{res}$ l'existence de deux programmes $init : \mathbf{T}\tau_S$ et $step : (\tau_S \times (\tau_{ores}\text{-option})) \to ((\tau_S \times \tau_{oarg}) + \tau_{res})$ tel que $exec\ n\ init\ bigstep == A$ pour un n assez grand.

L'idée derrière cette hypothèse est de découper en morceaux le calcul de l'attaquant à chacun de ses appels à l'oracle. En voyant l'attaquant comme une machine de Turing communiquant avec un oracle on peut se convaincre aisément de l'existence du couple $(init, exec)$.

Il serait cependant plus satisfaisant de construire une primitive supplémentaire dans le langage, transformant un attaquant en un tel couple.

De plus, lorsque l'on utilise ce couple au lieu de l'attaquant original pour utiliser le forking lemma dans une reduction, nous ne voulons pas autoriser la réduction à modifier ou même à lire l'état interne de l'attaquant lorsqu'il appelle les oracles, nous voulons seulement qu'elle puisse appeler *bigstep* plusieurs fois avec le même état interne. On peut imposer une telle limitation en utilisant l'algébricité définie dans la partie précédente : c'est cette fois l'attaquant qui sert d'API à la réduction. Nous voulons donc que la réduction ait pour type : $\forall \alpha\ .\mathbf{T}\alpha \times (\alpha \times (\tau\text{-option}_{ores})) \to ((\alpha \times \tau_{oarg}) + \tau_{res}) \to \mathcal{T}$.

F Implémentation complète des trois exemples

Lors des tests de la correction de la réduction, on notera en italique, les résultats des calcul de l'interpreteur.

F.1 Implémentation complète de l'exemple hash-then-sign

```
let apply_crit crit data att =
  let (oracle,test) <= crit data in
  let res <= (att oracle) in
  test res ;;

let test_in e =
  fold_List
    false
    (fun h q b -> or b (e=h));;

let find_ant f e =
  fold_List
    (val none)
    (fun h q r ->
       let c <= f h in
         if (c=e) then val(some h)
         else r
    );;

let logging f =
  let l <= ref nil in
val(couple
      (fun x ->
         let ll<= !l in
         l:=cons x ll;
        f x
      )
      (!l)
   )
   ;;

let limit_calls n f =
  let m <= ref n in
  val(fun x ->
       let m1<= !m in
      if (m1 = 0) then
        exit
      else begin
        m := (m1-1);
        f x
      end
   );;

let crit_ex_forgery data =
  let (nb_calls, sign_scheme) = data in
  let (gen,sign,verif) = sign_scheme in
  let (pk,sk)<= gen in
  let (sign_oracle,log) <= logging (sign sk) in
  let sign_oracle2 <= limit_calls nb_calls sign_oracle in
  let test signedmessage =
    let (message,signature) = signedmessage in
    let b1 <= verif pk message signature in
    let l <= log in
    val (b1 && (not (test_in message l))) in
  val ((sign_oracle2,pk),test);;

let crit_collision hash =
  val(couple unit
      (match_Couple
              (fun m1 m2 ->
                 let h1 <= hash m1 in
                let h2 <= hash m2 in
               val ((not(m1 = m2)) && (h1 = h2))
    )));;

```

```
let combine_crit crit1 crit2 data =
  let (data1,data2) =data in
    let (o1,v1) <= (crit1 data1) in
    let (o2,v2) <= (crit2 data2) in
        val ((o1,o2),
             fun x -> match Sum x with
               |left y -> v1 y
               |right y -> v2 y
                  endmatch);;

let crit_ex_forgery_or_collision =
combine_crit crit_ex_forgery crit_collision
;;

let hash_then_sign hash sign_scheme =
  let (gen,sign,verif) = sign_scheme in
    triple
      gen
      (fun sk x -> let y<= hash x in sign sk y)
      (fun pk x s -> let y<= hash x in  verif pk y s)
  ;;

let reduction hash att oracles =
  let (sign_oracle_pk,dummy) = oracles in
  let (sign_oracle, pk) = sign_oracle_pk in
  let (sign_oracle2,log) <= logging sign_oracle in
  let (m,s) <= att (sign_oracle2,pk) in
  let d <= hash m in
  let l <= log in
  let r <= find_ant hash d l in
  match Option r with
    |some m2 -> val(right (m,m2))
    |none -> val(left (d,s))
  endmatch;;

let theorem_premise sc hash n att =
  let sc2 = hash_then_sign hash sc in
  apply_crit crit_ex_forgery (n,sc2) att;;

let theorem_conclusion sc h n att =
  let att2 = reduction h att in
    apply_crit
      crit_ex_forgery_or_collision
      ((0,sc),h)
      att2
      ;;
```

La suite de l'exemple présente un test de valadité de la correction de la réduction. On défini un schéma test, ayant plusieures failles, et differents attaquants, att1 tirant avantage de l'insécurité du schéma de signature de base, et att2 des collisions de la fonction de hachage.

```
let n = 42;;
let k = 3;;
let r = 435;;

let gen = val(r,k*r);;
let sign sk m = val(sk+m);;
let verif pk m s = val(s = ((k*pk)+m));;

let sc = (gen,sign,verif);;

let h m = val(m mod n);;
let sc2 = hash_then_sign h sc;;

```

```
let att1 o =
  let (so,pk) = o in
    val(1,(3*pk)+1);;

let att2 o =
  let (so,pk) = o in
  let s<=so 200 in val(200+42,s);;


let r01 <= apply_crit crit_ex_forgery (0,sc2) att1;;
     r01 = true
let r02 <= apply_crit crit_ex_forgery (0,sc2) att2;;
     Failure
let r12 <= apply_crit crit_ex_forgery (1,sc2) att2;;
     r12 = true

let s01 <= apply_crit crit_ex_forgery_or_collision ((0,sc),h) (reduction h att1);;
     s01 = true
let s02 <= apply_crit crit_ex_forgery_or_collision ((0,sc),h) (reduction h att2);;
     Failure
let s12 <= apply_crit crit_ex_forgery_or_collision ((1,sc),h) (reduction h att2);;
     r12 = true
```

F.2 Construction de Goldreich Goldwasser et Micali [GGM86]

La construction de [GGM86] utilise un générateur pseudo-aléatoire pour construire une fonction pseudo-aléatoire sur le domaine $\{0,1\}^n$.

Générateur pseudo-aléatoire Un générateur pseudo-aléatoire est une fonction, qui étant donnée une clé tirée aléatoirement dans un ensemble de taille finie, renvoie une séquence infinie de bits, censée être indistinguable d'une séquence aléatoire.

Fonction pseudo-aléatoire Une fonction pseudo-aléatoire est paramétrée par une clé de taille finie, et, pour une clée tirée aléatoirement, est censée être indistingable d'une fonction aléatoire.

La construction GGM, pour un générateur prenant des clés de k bits, n'utilise que les $2k$ premiers bits des séquences aléatoire générée. On restreindra donc la taille de la séquence pseudo-aléatoire générée par les générateur pseudo-aléatoire. Il n'est en fait pas même necéssaire de supposer que les les entrées/sorties sont des chaînes de bits : un générateur pseudo-aléatoire, étant donné un élément $x \in D$ tiré aléatoirement, calcul un couple $(y_0, y_1) \in D^2$, indistinguable d'un élément aléatoire de D^2. Et l'on peut même se passer de la définition de générateur pseudo-aléatoire en les considérant comme des fonctions pseudo-aléatoire telle que $f\ x\ 0 = y_0$ et $f\ x\ 1 = y_1$.

Avec cette présentation, le polymorphisme nous à permit de remarquer que la construction GGM se généralisait, pouvant prendre en argument n'importe quelle fonction pseudo-aléatoire ayant un type de la forme $\alpha \to \beta \to \mathbf{T}\alpha$. Elle construit alors une fonction pseudo-aléatoire de type $\alpha \to \beta\text{-List}_n \to \alpha$, (où la notation abusive $\beta\text{-List}_n$ désigne les listes de taille n).

```
let make_crit_ind arg1 arg2 distrib=    définition des critères d'indistingabilité
  let b <= rand_bool in                 tirage aléatoire d'un bit
  let e1 = arg1 distrib in              construction du premier oracle
```

```
  let e2 = arg2 distrib in              construction du second oracle
  val(if b then e1 else e2,             choix de l'un des deux oracle en fonction de b
   fun b2 -> val(b=b2));;               victoire de l'attaquant s'il devinne b
make_crit_ind = <Func> : #'a'b:('a -> 'b) -> ('a -> 'b) -> 'a
    -> T('b * (bool -> Tbool))

let assoc x =                           recherche dans une liste d'association
  fold_List
    none
    (fun e l r ->
       let (y,fy) = e in
         if (x=y) then some fy
         else r);;
assoc = <Func> : #'a'b:'b -> (('b * 'a) List) -> 'a Option

let length =                            mesure de la longueur d'une liste
  fold_List
    0
    (fun a b n -> n+1);;
length = <Func> : #'a:('a List) -> int

let get_rand_function distrib =         construction d'une fonction aléatoire
  let f = fun x -> distrib in
  no_recall f;;
get_rand_function = <Func> : #'a'b: T'a -> T('b -> T'a)

let check_arg_length n f l =            limitiation d'une fonction au argument de longueur n
  if (length l) = n then f l
  else exit;;
check_arg_length = <Func> : #'a'b:int -> (('a List) -> T'b) -> ('a List) -> T'b

let get_rand_function_list n distrib =  construction d'une fonction aléatoire
  let f <= get_rand_function distrib in sur les listes de taille n
  val(check_arg_length n f);;
get_rand_function_list = <Func> : #'a'b:int -> T'b -> T(('a List) -> T'b)

let crit_ind_PRF prf =                  critère d'indistingabilite entre une fonction pseudo-aléatoire
  make_crit_ind                         et une fonction aléatoire
    (get_rand_function)
    (fun d -> let r <= d in val (prf r));;
crit_ind_PRF = <Func> : #'a'b:('a -> 'b -> T'a) -> T'a
    -> T(T('b -> T'a) * (bool -> Tbool))

let crit_ind_PRF_list n prf =           critère similaire, gerant la limitation de taille des listes
  make_crit_ind
    (get_rand_function_list n)
    (fun d -> let r <= d in
              prf r);;
crit_ind_PRF_list = <Func> : #'a'b:int -> ('b -> T(('a List) -> T'b)) -> T'b
    -> T(T(('a List) -> T'b) * (bool -> Tbool))

let ggm n prf key =                     définition de la construction GGM
  let aux k = fold_List k               consistant à appliquer reccursivement la fonction
    (fun h q r -> let a<= r in prf a h) pseudo-aléatoire, avec le résultat précédent comme nouvel clé
  in                                    et comme deuxième arguments, les éléments succesifs de la liste
  val(check_arg_length n (aux (val key)));;
ggm = <Func> : #'a'b:int -> ('b -> 'a -> T'b) -> 'b -> T(('a List) -> T'b)

let bounded_fold_List m fend fiter =
  fold_int
    fend
    (fun i r l -> match List l with
         nil -> exit
       |cons h q -> let a <= r q in
           fiter a h
    endmatch) m;;
```

```
bounded_fold_List = <Func> : #'a'b:int -> (('a List) -> T'b)
    -> ('b -> 'a -> T'b) -> ('a List) -> T'b

let cut_List n =                        fonction coupant une liste en deux partie
  bounded_fold_List n                   au niveau du n-ième élément
    (fun l -> val (nil,l))
    (fun x h -> let (g,d) = x in
     val (cons h g,d));;
cut_List = <Func> : #'a:int -> ('a List) -> T(('a List) * ('a List))

let ggm_hybrid n m prf distrib =        construction GGM hybride, utilisant un vraie
  let distrib2 =                        fonction aléatoire sur la première partie de la liste
    let key <= distrib in               puis en procédant comme GGM sur le reste
      ggm m prf key in
  let rf <= get_rand_function_list (n-m) distrib2 in
  val(fun l ->
        let (g,d) <= cut_List (m-n) l in
        let aux <= rf g
        in aux d);;
ggm_hybrid = <Func> : #'a'b:int -> int -> ('b -> 'a -> T'b)
    -> T'b -> T(('a List) -> T'b)

let ggm_hybrid_step n m step_rf prf = construction étant équivalente au GGM hybride de niveau m+1 sur n
  let distrib2 =                        si step_rf est un vraie fonction alétoire
    let keys <= step_rf in              et équivalente au GGM hybride de niveau m sur n
    val(fun a ->                        step_rf est l'instantiation de prf pour une clé aléatoire
          let key <= keys a in
          ggm m prf key) in
  let rf <= get_rand_function_list ((n-m)-1) distrib2 in
    val(fun l ->
  let (g,d) <= cut_List (n-m) l in
  match List d with
    |nil -> exit
    |cons h q ->
       let e1 <= rf q in
       let e2 <= e1 h in
         e2 g
    endmatch
       );;
ggm_hybrid_step = <Func> : #'a'b:int -> int -> T('a -> T'b)
    -> ('b -> 'a -> T'b)  -> T(('a List) -> T'b)

utilisation d'un distingueur sur la fonction pseudo-aléatoire GGM pour distinguer la fonction
de base, en inserant le problème au niveau m sur n
let reduction_ggm_step n m prf att step_rf =
  let lprf = ggm_hybrid_step n m step_rf prf in
    att lprf;;
reduction_ggm_step = <Func> : #'a'b'c:int -> int -> ('b -> 'a -> T'b)
    -> (T(('a List) -> T'b) -> 'c) -> T('a -> T'b) -> 'c

let reduction_ggm_rand n prf att step_rf=    réduction finale, inserant le problème
  let m<=rand_int n in                       à un niveau aléatoirement choisi
  reduction_ggm_step n m prf att step_rf;;
reduction_ggm_rand = <Func> : #'a'b'c:int -> ('b -> 'a -> T'b)
 -> (T(('a List) -> T'b) -> T'c) -> T('a -> T'b) -> T'c

let theorem_ggm_premise n distrib prf att =  programme permetant d'énoncer l'hypothèse de l'énoncé de sécurité
  let lprf = ggm n prf in
  apply_crit
a      (crit_ind_PRF_list n lprf)
      distrib
      att;;
theorem_ggm_premise = <Func> : #'a'b:int -> T'b -> ('b -> 'a -> T'b)
    -> (T(('a List) -> T'b) -> Tbool) -> Tbool

let theorem_ggm_conclusion n distrib prf att = programme permettant d'énoncer la conclusion de l'énoncé de sécurité
```

```
  let att2 = reduction_ggm_rand n prf att in
    apply_crit
      (crit_ind_PRF prf)
      distrib
      att2;;
theorem_ggm_conclusion = <Func> : #'a'b:int -> T'a -> ('a -> 'b -> T'a)
    -> (T(('b List) -> T'a) -> Tbool) -> Tbool

```

F.3 Méta-réduction de Paillier et Vergnaud

```
let assoc x =
  fold_List
    none
    (fun e l r ->
       let (y,fy) = e in
         if (x=y) then some fy
         else r);;

let no_recall_and_log f =
  let l<= ref nil in
  val( fun x ->
    let ll<= !l in
    match Option (assoc x ll) with
      |some fx -> val fx
      |none ->
         let fx <= f x in
           l:=cons (x,fx) ll;
                 val fx
    endmatch,
          !l);;

let no_recall f =
  let l<= ref nil in
  val( fun x ->
    let ll<= !l in
    match Option (assoc x ll) with
      |some fx -> val fx
      |none ->
         let fx <= f x in
           l:=cons (x,fx) ll;
                 val fx
    endmatch);;

let get_rand_function distrib =
  let f = fun x -> distrib in
  no_recall f;;

let gets_from_list l =
  let rest <= ref l in
  val(
    let ll <= !rest in
    match List ll with
      |cons h q -> rest:=q; val h
      |nil -> exit
    endmatch,
  !rest);;

let limit_calls n f =
  let m <= ref n in
val(    fun x ->
       let m1<= !m in
      if (m1 = 0) then
        exit
      else begin
        m := (m1-1);
```

```
         f x
       end
   );;

let rev_append =
    fold_List
      (fun x -> x)
      (fun h q f -> fun qq -> f (cons h qq));;

let rev l = rev_append l nil;;

let list_map f =
  fold_List
    nil
    (fun h q r -> cons (f h) r);;

let list_map_eval f =
  fold_List
    (val nil)
    (fun h q r ->
       let rr <= r in
       let hh <= f h in
           val(cons hh rr));;

let mult_list_by_base b e =
  fold_List
    (cons (b,e) nil)
    (fun h q r ->
       let (b1,e1) = h in
         if (b=b1) then
           cons (b,e+e1) q
         else
           cons h q )
    ;;

let mult_list_by_list l =
  fold_List
    l
    (fun h q r ->
       let (b,e) = h in
       mult_list_by_base b e r);;

let expo_list e =
  fold_List
    nil
    (fun h q r ->
       let (b,e1) = h in
         cons (b,e1*e) r);;

let extractable_api api =
  let (prod,expo,repr) = api in
  let extr_prod x y =
    let (xg,xl) = x in
    let (yg,yl) = y in
      (prod xg yg,mult_list_by_list xl yl) in
  let extr_expo x e =
    let (xg,xl) = x in
    (expo xg e, expo_list e xl) in
  let extr_repr x =
    let (xg,xl) = x in
      repr xg in
    (extr_prod,extr_expo,extr_repr);;

let extr_base x = (x,cons (x,1) nil);;

let extractable_reduction red oracles =
```

```
  let (api_att,g,y) = oracles in
  let (api,att) = api_att in
  red ((extractable_api api,att),
       extr_base g,extr_base y);;

let subst_extr_base b y =
  let (bg,bl) = b in let (yg,yl) = y in
  let new_yl =
  fold_List nil
    (fun h q r ->
       let (b1,e1)=h in
         if (b1=bg) then mult_list_by_list (expo_list e1 bl) r
                    else mult_list_by_base b1 e1 r)
   yl in
   (yg,new_yl);;

let extr_fun f x = let (xg,xl)=x in f xg;;

let extract_log x =
  let (xg,xl) = x in xl;;

let pseudo_att get_elem hash api dl_oracle =
  let (prod,expo,repr) = api in
  fun x -> let (y,m,omega) = x in
  let r <= get_elem in
  let c = hash (m,r) in
  let s <= dl_oracle (prod r (expo y c)) in
    val(s,c);;

let get_expo g y =
  let (yg,l) = y in
  fold_List 0 (fun h q r -> let (b,e)=h in
         if (b=g) then r+e else r) l;;

let extract_logarithm_from_log_line g r0 k0 line =
  let (inp,out) = line in
  let (y,m,omega) = inp in
  let (s,c) = out in
  let a = get_expo g y in
  let b = get_expo r0 y in
    s-(c*(a+(k0*b)));;

let meta_reduction hash red oracles =
  let (api__dl_oracle,g,ylist) = oracles in
  let (api,dl_oracle) =  api__dl_oracle in
  let (prod,expo,repr) = api in
  let api2 = extractable_api api in
  let (prod2,expo2,repr2) = api2 in
  let (get_elem,get_rest) <= gets_from_list ylist in
  let hash2 sm = let (m,x) = sm in hash (m,repr2 x) in
  let get_elem_extr = let y <= get_elem in val(extr_base y) in
  let (att,get_log) <= no_recall_and_log (pseudo_att (get_elem_extr)
                         hash2  (extractable_api api) (extr_fun dl_oracle)) in
  let r0 <= get_elem in
  let k0 <=  extractable_reduction red ((api,att),g,r0) in
  let log <= get_log in let rest <= get_rest in
  let first_logarithms = list_map (extract_logarithm_from_log_line g r0 k0) log in
  let last_logarithms <= list_map_eval dl_oracle rest in
  val(cons k0 (rev_append first_logarithms (rev last_logarithms)));;

let pseudo_att_crit q api hash g x =
    let (prod,expo,repr) = api in
    let (y,m,omega) = x in
    let sk = get_expo g y in
    let k <= rand_int q in
    let r = expo g k in
```

```
    let c = hash (m,repr r) in
  val(((k+(c*sk)) mod q),c);;

let crit_red_n_UF_KOA_to_DL q n api hash g =
  let api2 = extractable_api api in
  let (prod2,expo2,repr2) = api2 in
  let g2 = extr_base g in
  let skey <= rand_int q in
  let h = expo2 g2 skey in
  let att <= no_recall (pseudo_att_crit q api2 hash (extr_base g)) in
  let att2 <= limit_calls n att in
    val(((api2,att2),g2,h), (fun x -> val(x=skey)));;

let crit_DL q api g =
  let (prod,expo,repr) = api in
  let e <= rand_int q in
  let h = expo g e in
  val(
    (api,g,h),
    (fun x -> val(x=e)));;

let verif_2_lists verif =
  fold_List
    (fun l -> match List l with |nil -> true |cons h q -> false endmatch)
    (fun h q r ->
         fun l -> match List l with
           |nil -> false
           |cons h2 q2 -> (verif h h2) && (r q2)
    endmatch);;

let gen_list gen=
    fold_int
      (val nil)
      (fun m r ->
         let l <= r in
         let x <= gen in
         val(cons x l));;

let crit_n_DL q n api g =
  let api2 = extractable_api api in
  let (prod,expo,repr) = api in
  let (prod2,expo2,repr2) = api2 in
  let g2 = extr_base g in
  let input <= gen_list (let r<= (rand_int q) in val(expo2 g2 r)) (n+1) in
  let dl_oracle <= limit_calls n (fun h -> val(get_expo g h)) in
  let verif_dl = verif_2_lists (fun a e -> let (y,yl)=a in ((expo g e) = y)) in
val(((api2,dl_oracle),g2,input),fun l -> val(verif_dl input l));;

let theorem_premise q n api hash g red =
  apply_crit (crit_red_n_UF_KOA_to_DL q n api hash g) red;;

let theorem_conclusion q n api hash g red =
  apply_crit (crit_n_DL q n api g) (meta_reduction hash red);;
```