Preuves en Cryptographie – Rapport projet 1

Implémentation d'un langage de programmation probabiliste.

Mardi 2017-02-07 ; Master 2 - IFI ; SI5

Groupe :

- Thibaud CANALE,
- Lucas MARTINEZ,
- Cindy NAJJAR,
- Lucas SOUMILLE.

I. Partie 1 : Implémentation de ProbLang

1. Choix des technologies

Pour l'implémentation de ProbLang, nous avons fait le choix du langage de programmation Java. Ce choix s'explique principalement par la possibilité d'utiliser la bibliothèque ANTLr4 qui permet une génération automatique d'un parser à partir d'une grammaire donnée. De plus, c'est sur ce langage que nous avons le plus d'expérience ce qui nous permettra d'éviter des pertes de temps dues notamment à l'installation d'un environnement de développement.

2. Éléments nécessaires pour l'implémentation

Par conséquent, la première étape du projet a été la réalisation d'une grammaire représentant le "langage de programmation probabiliste" tel qu'introduit en cours. Cette grammaire, écrite dans le fichier "Grammar.g4" permet la création d'un analyseur syntaxique ("parser"). En utilisant ce dernier, nous allons pouvoir récupérer les diverses instructions du programme au moyen des règles décrites dans la grammaire.

Nous retrouvons dans la classe AntlrAPI, un ensemble de fonctions implémentées permettant d'interagir avec les objets mis à disposition par la bibliothèque ainsi que les règles de la grammaire. Le constructeur de la classe, prenant en paramètre le programme que nous souhaitons analyser, permet de créer un "parser" pour celui-ci. Tout au long du rapport, nous allons détailler le fonctionnement de ces méthodes.

Dans cette implémentation, Il nous a fallu aussi représenter les différentes entités nécessaires pour permettre l'application des règles sémantiques. C'est le package *business* qui contient ces classes. Un programme et une mémoire représentent une configuration. Ensuite, une liste de configurations avec une probabilité mesurée pour chacune des configurations donnent une distribution.

3. Implémentation de ProbLang

Maintenant que nous possédons un parser pour notre programme et les objets nécessaires à l'application des règles, nous allons nous intéresser aux fonctionnalités du programme ProbLang.

a. Vérification syntaxique

La première étape du programme ProbLang consiste à vérifier que le programme donné en paramètre est syntaxiquement correct c'est-à-dire qu'il correspond à la grammaire définie dans le cours.

Pour faire une première "passe" sur le programme, nous utilisons l'instruction `this.parser.program();`. La méthode *program* correspond en fait à l'appel de la règle

program sur le programme. En se référant à la grammaire, nous appliquons la règle `program : c(';' c)*;` qui permet de récupérer l'ensemble des instructions composant le programme.

C'est la méthode *launchValidationProcess* de la classe AntlrAPI qui permet de faire cette vérification. À la fin de la "passe" après l'appel au "parser", nous regardons le nombre d'erreurs syntaxiques relevées par ce dernier. Si le total est différent de 0, nous jetons une exception de type *ErrorSyntaxException*.

b. Distribution initiale

Avant d'appliquer les règles sémantiques, nous devons générer la distribution initiale (d0) du programme. Pour cela, nous créons une instance de nos classes représentant nos entités, plus précisément une distribution contenant la configuration initiale avec le programme et la mémoire. Bien entendu, la probabilité de cette configuration est égale à 1.0.

Pour initialiser la mémoire avec le nom des variables présentes dans le programme à analyser, nous avons implémenté la fonction *getVarTokens* dans la classe AntlrAPI qui parcourt l'ensemble des instructions représentées par une liste de *CContext* (Le c correspond à la règle c de la grammaire). Nous regardons si chacune des instructions est une affectation, si c'est le cas nous ajoutons à la mémoire initiale le nom de la variable. Si l'instruction est un if ou un while, la recherche s'effectue récursivement dans leur bloc.

c. Parcours du programme à analyser

Notre implémentation se rapproche davantage de la génération d'une chaîne de Markov que sur l'utilisation de la formule "distribution transformer". Ce choix d'implémentation est dû aux valeurs de retour des fonctions de la bibliothèque Antlr qui facilite une analyse plutôt itérative des instructions.

De ce fait, nous reproduisons le même parcours de programme que pour la recherche de variables. À chaque instruction rencontrée, nous appliquons une règle spécifique (que nous détaillerons plus loin) en fonction du type d'instruction. Cette règle prendra en paramètre la distribution précédente et en générera une nouvelle. À la fin de la fonction *programRuleApplication*, la variable *outputDistribution* contient la distribution finale pour notre programme à analyser.

d. Détails des règles appliquées

Comme énoncé précédemment, à chaque type d'instruction correspond une règle. Le programme ProbLang est composé de trois règles différentes : pour l'affectation, pour un if et pour un while, implémentées respectivement dans *AffectationRule*, *IfRule* et *WhileRule*. Chacune de ces classes étendent la classe abstraite ARule. C'est dans la fonction *applyRule* que nous retrouvons le code logique de chacune des règles. À la suite de l'application d'une règle, nous supprimons l'instruction consommée dans l'ensemble des programmes de toutes les configurations.

i. Règle pour l'affectation

La règle d'affectation est appliquée sur une instruction si cette dernière contient un appel à la règle VAR de la grammaire. Nous appliquons la règle pour toutes les configurations de la distribution d'entrée.

Il faut ensuite déterminer le type d'affectation pour l'instruction sélectionnée. Le type dépend du membre droit de l'affectation.
Si ce dernier est une expression c'est à dire une constante, une variable ou une opération, alors nous déterminons la valeur de l'opérande au moyen la fonction *getValueForAffectation* de la classe ExprHelper (détails de la fonction dans le paragraphe e). Ensuite, nous mettons à jour la mémoire avec la nouvelle valeur pour la variable correspondant à l'opérande de droite. La probabilité de cette configuration reste inchangée car ce type d'affectation est déterministe.

Au contraire, si l'opérande droite est une fonction probabiliste alors le traitement est différent. Une fonction probabiliste entraîne la création de nouvelles configurations. Ces configurations sont calculées dans la méthode *getConfigurationForAffectation* de la classe ProbFuncHelper. Pour déterminer la probabilité des configurations, nous prenons la probabilité de la configuration précédente auquel nous multiplions par la valeur *1/ValeursPossiblesFonctions*. Les nouvelles configurations sont ajoutées dans la distribution de sortie.

ii. Règle pour le if

La règle pour le if est appliquée sur une instruction si cette dernière contient un appel à la règle IfInst de la grammaire. Nous appliquons la règle pour toutes les configurations de la distribution d'entrée.
La première étape dans l'analyse d'un if pour une configuration est de déterminer la valeur de l'expression dans la condition.
En fonction de la valeur de retour de l'expression, nous exécutons le programme ProbLang sur le bloc du if ou du else. Les programmes des configurations issues de l'un ou l'autre des blocs sont mis à jour pour continuer l'analyse du programme initial, avant d'être ajouté à la distribution de sortie.

iii. Règle pour le while

La règle pour le while est appliquée sur une instruction si cette dernière contient un appel à la règle WhileInst de la grammaire. Nous appliquons la règle pour toutes les configurations de la distribution d'entrée.
Au contraire de la règle if, ici l'expression dans la condition du while doit être vérifiée après chaque exécution de ProbLang sur le bloc.
Lorsque cette dernière n'est plus vérifiée nous récupérons la distribution de sortie de l'exécution du dernier bloc que nous ajoutons à la distribution de sortie de la règle.

e. Analyse des expressions et des fonctions probabilistes

Dans la classe *ExprHelper*, nous avons implémenté la fonction *getValueForAffectation* qui retourne la valeur de l'expression dans la première instruction de la configuration. L'opérande droit peut être une constante ; si c'est le cas alors la fonction retourne la valeur de cette constante.
Il est aussi possible que la partie droite de l'expression soit une variable ce qui induit une recherche dans la mémoire pour retourner la valeur de cette variable.
Enfin, le dernier cas est que l'opérande droite est une expression comme par exemple une addition ou une multiplication. Nous analysons cette partie pour déterminer les opérateurs ainsi que les différentes constantes. Dans le cas d'une expression à plusieurs opérateurs, nous faisons le calcul dans leur ordre d'apparition et pas par rapport à l'ordre de priorité de chaque opérateur.

La fonction *getValueForOperation* permet de réaliser sémantiquement les opérations du programme à analyser.

Dans la classe ProbFuncHelper, nous avons implémenté *getConfigurationForAffectation* qui génère les configurations avec leur probabilité associée. Le programme ProbLang permet de gérer plusieurs fonctions probabilistes.
Il est possible de générer les configurations pour la fonction ensemble correspondant à la notation { X, … }.
Aussi quand nous utilisons la notation Zq, la variable à droite de l'affectation prendra alors comme valeurs possibles { 0, ... , q-1 }.
La fonction "powerensemble" est supportée, elle correspond notamment à cet exemple x:={0,1}². Dans l'affichage des valeurs créées par la fonction, les zéros inutiles sont supprimés. Par conséquent, le résultat 0 dans la variable x en mémoire correspond à 00.
Afin de pouvoir tester les programmes présents dans les sujets de TD, nous avons implémenté la fonction countones qui retourne le nombre de 1 présent dans la valeur d'une variable.

II. Partie 2 : Implémentation de la preuve d'El Gamal

La deuxième partie de ce projet est l'implémentation de la preuve CPA d'El Gamal. L'idée ici est de démontrer que ElGamal est CPA au moyen d'une preuve par le jeu entre le programme et un adversaire. Ainsi à partir du programme initial, nous allons établir une suite de dérivations et nous montrerons l'équivalence de chacun des éléments de la suite. Nous allons montrer que pour un adversaire il est autant difficile de trouver la valeur du paramètre *b* du schéma d'El Gamal que de déterminer si un adversaire est dans DDH0 ou DDH1.

Dans cette preuve, l'adversaire peut faire appel autant qu'il le souhaite à l'oracle qui prend en paramètre deux entrées et retourne le chiffré d'une de ces entrées en fonction de la

valeur de *b.* Dans notre implémentation, nous avons décidé que l'adversaire ferait seulement un seul appel à cette oracle.

1. Équivalence entre deux programmes

Afin de réaliser la preuve d'El Gamal, nous avons dû implémenter l'équivalence entre programmes. Deux programmes sont équivalents pour une variable donnée si pour chaque valeur de cette variable, la probabilité dans les deux programmes est identique.

Pour tester l'équivalence entre deux programmes, nous faisons appel à la méthode *areEquiv* de la classe CompareProg. Il est possible de spécifier les variables sur lesquelles nous voulons déterminer l'équivalence. Si aucune n'est spécifiée alors nous déterminons les variables communes aux deux programmes pour le calcul.

La première partie de l'implémentation consiste à calculer la distribution finale des deux ProbLang. Une fois les résultats obtenus, nous calculons la probabilité des variables spécifiées ou par défaut qui sont utilisées pour la comparaison pour chacune des distributions finales. Pour déterminer, il nous suffit ensuite de comparer les dictionnaires deux à deux, si pour chaque couple variable et valeur de cette variable donne la même probabilité pour les deux programmes alors ces derniers sont équivalents.

2. Dérivation des programmes d'El Gamal

Comme nous l'avons vu en cours pour réaliser la dérivation de l'El Gamal, nous utilisons plusieurs opérations comme la suppression de code mort, la mise de fonction en *inline* ou encore l'échange d'instructions dans le programme.

Nous avons réalisé ces étapes à la main et ajouté les programmes générés dans le fichier *ElGamalProgs*. Ce sont les retours de ces fonctions qui vont permettre la création des ProbLang correspondant au schéma d'El Gamal et ainsi pouvoir tester les équivalences.
Nous avons ajouté des instructions intermédiaires par rapport aux programmes décrits par le cours, ceci s'explique par nos choix d'implémentation. L'ensemble des programmes commencent par une affectation à la variable q pour qu'elle soit enregistrée en mémoire. De plus, nous faisons des calculs intermédiaires dans l'oracle car comme dit précédemment nous n'avons pas implémenté la priorité des opérateurs.
Ces changements syntaxiques n'impactent pas la sémantique du programme, ce qui va nous permettre d'établir la preuve.

3. Détails de la preuve

Au moyen de la bibliothèque UniCrypt, nous avons généré un groupe cyclique avec q = 11 et le générateur = 4. Ces valeurs sont fixées par défaut.
L'adversaire fait un seul appel à l'oracle dans notre preuve, il essaie de chiffrer les valeurs x0 = 1 et x1 = 3 ; ces valeurs n'ont pas été choisies au hasard car elles appartiennent au groupe cyclique énoncé plus haut.

La première partie de la preuve consiste à démontrer les équivalences successives entres CPA-EG et DDH0.
La deuxième partie consiste à démontrer les équivalences successives entre EG' et DDH1.
Enfin nous montrons l'équivalence entre DDH0 et DDH1 pour la valeur d = 1 (l'adversaire pense qu'il est dans le jeu DDH1), ce qui montre que de déterminer notre présence dans DDH0 ou DDH1 correspond à une expérience de lancer une pièce et à essayer de deviner son résultat. La probabilité d'être dans un des deux cas est égale à ½.
Les équivalences successives entre El Gamal et DDH nous permettent d'affirmer qu'il est alors impossible pour l'attaquant de savoir si la valeur chiffrée par l'oracle est x0 ou x1.
Ainsi, nous pouvons conclure que si les conditions du schéma d'El Gamal sont respectées alors ce schéma est prouvé CPA.