Vers une vérification efficace des protocoles de sécurité

Rapport de spécification fonctionnelle – Projet 4INFO

Octobre 2019

Sophie Delannoy
Pierre-Yves Le Rolland-Raumer
Julio Santilario-Berthilier
Emma Scalart
Aminata Sissokho
Guillaume Sonnet
Julius Wenzel

Encadrants :
Barbara Fila
Stéphanie Delaune

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Sommaire

1 Introduction

La vérification des protocoles de sécurité n'est pas une chose aisée et peut s'avérer être un problème indécidable dans certains cas. Il existe cependant une situation qui permet d'affirmer la décidabilité du problème de vérification d'un protocole de sécurité. Il s'agit des protocoles respectant des règles de typage strictes et pouvant par ailleurs se représenter sous la forme d'un graphe orienté acyclique. En effet, il est prouvé que les protocoles entrant dans cette catégorie peuvent être vérifiés automatiquement [1].

L'objectif de ce projet est alors de produire un logiciel capable de répondre à la question de décidabilité de la sécurité d'un protocole. La première étape abordée dans le rapport de pré-étude était la maîtrise des principes théoriques sur lesquelles la vérification des protocoles se repose. Les primitives cryptographiques, le système de typage, les fonctions ρ_{in} et ρ_{out} ainsi que leurs dépendances sont des notions qu'on considère connues dans le rapport présent.

L'enjeu actuel repose alors sur la définition concrète des caractéristiques du logiciel et des moyens qui seront utilisés pour les implémenter. Partant d'un savoir faire théorique, comment rendre automatique et systématique le processus de décidabilité de la vérification d'un protocole ? C'est après cette réflexion que différentes étapes d'implémentations ont pu être distinguées, ayant respectivement pour but de parser le protocole d'entrée, construire le graphe à partir du protocole, tester l'acyclicité du graphe, et gérer l'affichage du graphe. Leurs spécifications et leurs interactions font donc l'objet de ce rapport.

2 Architecture générale

Dans cette section, une vue globale des objectifs du logiciel sera présenté afin de donner une première intuition de son architecture et des interactions entre les modules.

2.1 Rappels sur les protocoles

L'étape de pré-étude du projet a permis d'identifier les différentes étapes menant à la vérification d'un protocole. Pour rappel la description d'un protocole de sécurité est composée d'une suite d'actions, in et out, correspondant respectivement à la réception et à l'émission de messages, qui peuvent être chiffrés ou non, et qui sont formellement représentés par des termes. Un terme peut être composé de plusieurs sous-termes, dont la plus petite entité est une constante. Chaque constante possède un nom et un type.

C'est le passage de la description des messages échangés au cours d'un protocole à l'aide des noms des constantes, à l'utilisation de leurs types seulement qui constitue une étape clé de notre analyse de protocole. En effet c'est à partir de la description typées du protocole que les fonctions ρ_{in} et ρ_{out} sont construites.

Nous illustrons cette construction par un exemple du protocole Denning-Sacco présenté en figure 1, qui permet d'établir une communication entre deux participants :

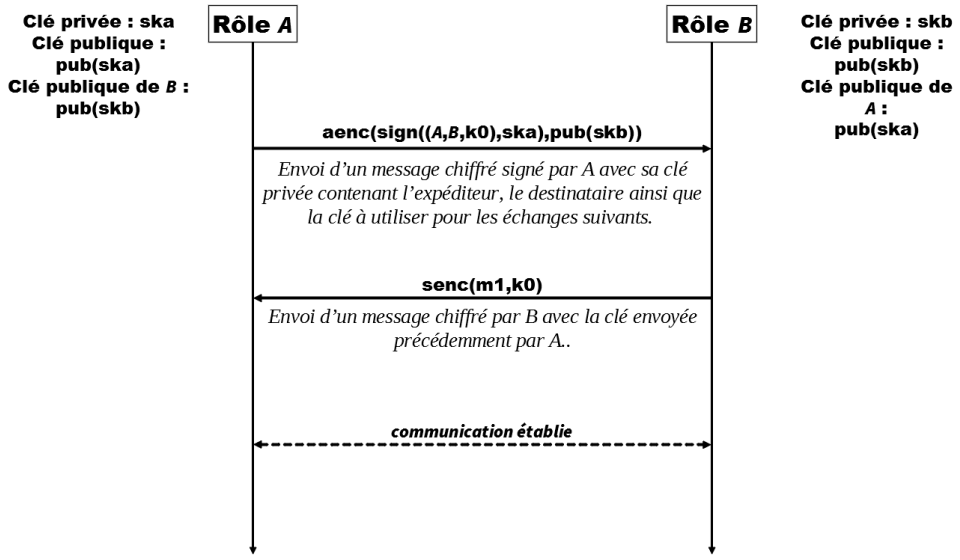

FIGURE 1 – Communication entre B et A en utilisant le protocole de Denning-Sacco (version améliorée)

Les deux actions actions décrites sur le schéma deviennent en spécification typée :

$$\mathsf{out}(1, \mathsf{aenc}(\mathsf{sign}((\mathsf{typea}, \mathsf{typeb}, \mathsf{noncek}), \mathsf{kaenca}), \mathsf{pub}(\mathsf{kaencb})))$$

$$\mathsf{in}(1, \mathsf{senc}(\mathsf{noncem}, \mathsf{noncek}))$$

Les fonctions ρ_{in} et ρ_{out} correspondantes :

- pour l'action out, alors on obtient :

$$\{(\mathsf{aenc}(\mathsf{sign}(\langle \mathsf{typea}, \mathsf{typeb}, \mathsf{noncek} \rangle, \mathsf{kaenca}), \mathsf{pub}(\mathsf{kaencb})), \varepsilon) \# (\emptyset; \emptyset)\}$$

- pour l'action in, alors on obtient :

$$\rho_{in}(\tau) = \{\tau, \mathsf{noncem}, \mathsf{noncek}\}$$

Finalement la fonction ρ_{in}correspond à la liste de tous les sous-termes accessibles à partir du terme de l'action in, et la fonction ρ_{out} liste tous les types déchiffrables ainsi que la clé correspondante nécessaire.

Pour plus de précisions sur les étapes intermédiaires de construction des fonctions ρ_{in} et ρ_{out} ainsi que la construction du graphe de dépendance d'un protocole de sécurité, se référer aux sections 4 et 5 du rapport de pré-étude du projet [2].

C'est en s'appuyant sur les principes théoriques décrits dans le rapport de pré-étude du projet [2] que notre logiciel doit être capable :

- d'interpréter des documents décrivant un protocole de sécurité et en donner une représentation informatique ;
- de calculer les dépendances d'un protocole pour construire son graphe de dépendance ;
- d'afficher ce dernier de manière optimale ;
- de déterminer l'acyclicité du graphe de dépendance pour s'assurer de la vérifiabilité du protocole.

2.2 Architecture envisagée

Pour atteindre les objectifs du point de vue utilisateur décrits à la section 2.1, les fonctionnalités recherchées ont été séparées en différents modules. Le graphe de la structure générale de notre programme, figure 2 , reprend ces modules et expose leurs interactions.

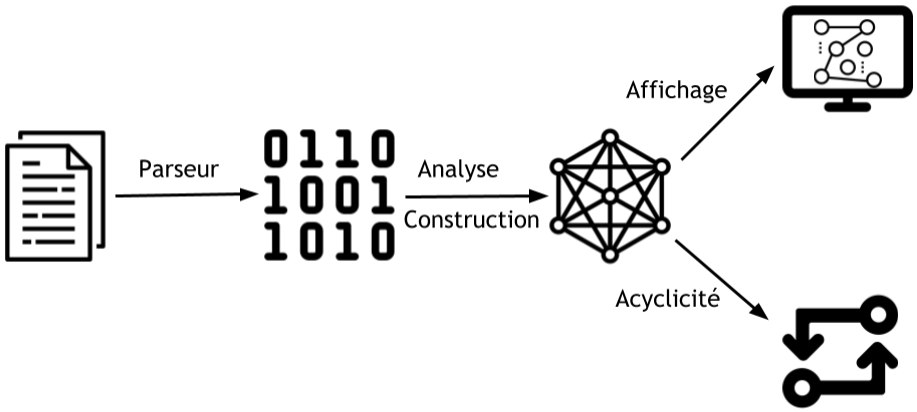

FIGURE 2 – Architecture de Cyclos

La suite chronologique des actions de notre programme correspondra à la liste suivante, à laquelle nous avons indexée l'icone du module correspondant.

1. Lecture d'un fichier type .txt décrivant un protocole rédigé selon le format évoqué dans la section 3.1 ;
2. Découpage et analyse du fichier pour en extraire la description du protocole et donner en sortie des objets le représentant ;

3. 0110 1001 1010 Représentation du protocole par une structure de donnée composée des objets résultant de l'étape précédente ;
4. 0110 1001 1010 Parcours de cette structure pour construire les fonctions ρ_{in} et ρ_{out} ;
5. L'analyse de ces fonctions permet de composer des listes de dépendances ;
6. La comparaison de ces listes ressort les dépendances entre les actions du protocole ;
7. Création du graphe sous forme d'une structure de données à partir des conclusions de l'étape précédente ainsi que des étapes du protocole ;
8. Détermination de l'acyclicité du graphe de dépendance obtenu ;
9. Affichage du graphe ;
10. Possibilité de sauvegarder le graphe.

Plus de précisions sur le fonctionnement de ces étapes seront données dans les sections suivantes.

3 Spécifications détaillées

À travers cette section, nous détaillerons les spécifications des différents modules évoqués à la section 2.2.

La nomenclature des spécifications se fera avec la règle suivante : un premier sigle symbolise le module concerné, à savoir

- **PA** pour le parseur ;
- **AC** pour l'analyse et la construction des graphes ;
- **ACY** pour le test de l'acyclicité ;
- **AF** pour l'affichage du graphe ;
- **IU** pour l'interface utilisateur.

En plus de ce sigle sera ajouté un mot pour identifier la spécification proposée ainsi qu'un numéro si cette dernière nécessite d'autres spécifications. À titre d'exemple la spécification **[PA-TYPE-1]** est la première règle parlant des types du parseur.

3.1 Parseur

Le but du parseur est de transformer la description d'un protocole en données utilisables par un programme. La description du protocole fournit par l'utilisateur doit respecter un format particulier. Il partage beaucoup de similarités avec le langage théorique introduit par Stéphanie Deleaune dans [1], notamment en ce qui concerne les opérations cryptographiques (détaillées dans le rapport de pré-études dans la section 3). Cette section s'attellera à détailler le format que l'utilisateur devra utiliser pour décrire un protocole – qui est directement corrélé avec ce que le parseur devra analyser ; ainsi que certaines fonctionnalités du parseur.

- **[PA-STRUCTURE-1]** Le fichier pris en entrée par le parseur doit être divisé en 3 parties :
  - La déclaration des constantes publiques (soit les données connues par l'attaquant au début de l'exécution du protocole) ;
  - La déclaration des constantes privées (soit les données non connues par l'attaquant au début de l'exécution du protocole) ;
  - La description du protocole.
- **[PA-STRUCTURE-2]** Chaque partie sera identifiée à l'aide d'une balise unique ;
- **[PA-STRUCTURE-3]** La déclaration de constantes devra se faire dans la partie déclaration des constantes publiques, et déclaration des constantes privées ;
- **[PA-DONNEE-1]** Chaque constante possède un nom et un type et un seul ;
- **[PA-DONNEE-2]** Le nom d'une constante est unique ;
- **[PA-DESCRIPTION-1]** Une transaction est une séquence d'actions – telles qu'énoncé dans la section 2.1 ;
- **[PA-DESCRIPTION-2]** La description du protocole est composée du détail des transactions possibles pour chaque participant (hors attaquant) avec des rôles différents ;
- **[PA-DESCRIPTION-3]** Chaque transaction sera introduite par une balise unique ;
- **[PA-DESCRIPTION-4]** Une balise spécifique sera utilisée pour différencier les émissions des réceptions de termes ;
- **[PA-TERMES]** Les termes doivent respecter les règles énoncées dans [1] concernant l'arité des opérations ainsi que certaines propriétés au niveau du type des constantes ;
- **[PA-ENTREE]** Le parseur doit pouvoir lire des fichiers au format texte, type .txt, encodés en UTF-8 ;
- **[PA-ANALYSE-1]** Le parseur doit savoir interpréter les différentes balises du document pour lancer les traitements adaptés ;

- **[PA-ANALYSE-2]** Le parseur doit savoir analyser la syntaxe établie via les spécifications précédentes pour en déduire les types des constantes ainsi que les termes échangés ;
- **[PA-ÉTIQUETTE]** Le parseur doit ajouter devant chaque étape d'une transaction une étiquette unique permettant de symboliser cette étape pour des traitements ultérieurs ;
- **[PA-SORTIE]** Le parseur donne, une fois son analyse effectuée, une représentation informatique de la description de protocole et des constantes employées compatible avec les spécifications de la section 3.2 ;
- **[PA-ERREUR]** Si le parseur rencontre une erreur lors de la lecture du document, il doit arrêter son analyse et faire remonter l'erreur à l'utilisateur en spécifiant la ligne du document où il a rencontré un problème, ainsi que la nature de ce dernier ;
- **[PA-COMMENTAIRE]** L'utilisateur doit pouvoir écrire des commentaires, qui ne seront pas interprétés par le parseur ;
- **[PA-TYPE]** Le parseur doit pouvoir déduire l'honnêteté des types des constantes du protocole sauf si l'utilisateur le déclare lui-même via une balise spéciale. Si tel est le cas, le parseur devra suivre les indications de l'utilisateur et abandonner ses déductions.

Un exemple de ce à quoi pourrait ressembler une description de protocole respectant nos spécifications peut être trouvé à la figure 3.

```
%Ceci est un commentaire

pub_dec : %Section pour décrire les constantes publiques

%Format pour déclarer les constantes -> nom : type
a : typea
b : typeb

priv_dec : %Section pour décrire les constantes privées

ska : kaenca
skb : kaencb
k0 : noncek
xa0 : noncem

protocol : %Section pour décrire le protocole

%Format pour déclarer une transaction -> nom : tr
%Déclaration d'une transaction composée d'une étape unique
zero : tr
out((pub(ska),pub(skb))) %Emission du terme (pub(ska),pub(skb))

%Déclaration d'une transaction composée de deux étapes
first : tr
out(aenc(sign((a,b,k0),ska),pub(skb)))
in(senc(xa0,k0)) %Réception du terme senc(xa0,k0)
```

FIGURE 3 – Exemple d'une description de protocole satisfaisant nos spécifications

3.2 Analyse et construction du graphe

Dans cette partie, nous décrirons les principales spécifications concernant la représentation informatique de la description de protocole ainsi que de son analyse pour construire le graphe de dépendance. Des précisions concernant ce dernier et ses propriétés sont disponibles dans [2] aux sections 4 et 5 ainsi que dans [1].

- **[AC-DONNEE]** La représentation informatique du protocole concerne les transactions, les termes et les types rencontrés lors de l'analyse du parseur ;
- **[AC-PARADIGME]** La représentation informatique du protocole et les outils l'analysant devront être codés avec le paradigme de programmation orientée objet ;
- **[AC-TYPE]** Chaque type rencontré lors de l'analyse du parseur sera une instance d'une classe les symbolisant. Cette classe possédera comme attribut au moins le nom du type, et devra connaître l'honnêteté ainsi que le caractère public du type ;
- **[AC-TERME]** Chaque terme rencontré lors de l'analyse du parseur sera une instance d'une classe les symbolisant. Cette classe peut symboliser une constante représentée par son type ou bien une primitive cryptographique, auquel cas en fonction

de l'arité de cette dernière, la classe devra connaître ou non des sous-termes. La position relative des sous-termes par rapport au terme parent devra être déductible et respecter l'ordre tel qu'examplifié dans la figure 4 sur un terme du protocole de Denning-Sacco ;

- **[AC-TRANSACTION]** Chaque transaction rencontrée lors de l'analyse du parseur sera une instance d'une classe les symbolisant. Cette classe contiendra la liste ordonnée de tous les termes impliqués dans la transaction, ainsi que des informations sur le contexte où le terme est employé (émission ou réception, étiquette du parseur...) ;
- **[AC-OPERATEUR-1]** Les opérateurs ρ_{in} et ρ_{out} devront être implémentés. Ils devront respecter les règles énoncés dans [1] et dans le rapport de pré-étude [2] à la section 5.3 ;
- **[AC-OPERATEUR-2]** Les opérateurs ρ_{in} et ρ_{out} devront retourner une réponse sous forme d'un objet. La classe de l'objet sera différente selon l'opérateur ;
- **[AC-DEPENDANCE]** Les objets analysant le parseur devront détecter les dépendances du protocole. Il en existe trois :
  - Les dépendances séquentielles ;
  - Les dépendances de données ;
  - Les dépendances de clés.

  Ces dépendances sont toutes détaillées dans [2] à la section 5.3 ;
- **[AC-GRAPHE-1]** Le graphe de dépendance sera construit avec les règles suivantes issues de [2] à la section 5.3 :
  - Les sommets du graphe correspondent aux étapes constituant toutes les transactions de la description du protocole ;
  - Les arêtes sont déduites des dépendances entre deux étapes des transactions du protocole.
- **[AC-GRAPHE-2]** Une représentation informatique du graphe de dépendance devra être présente dans l'implémentation.

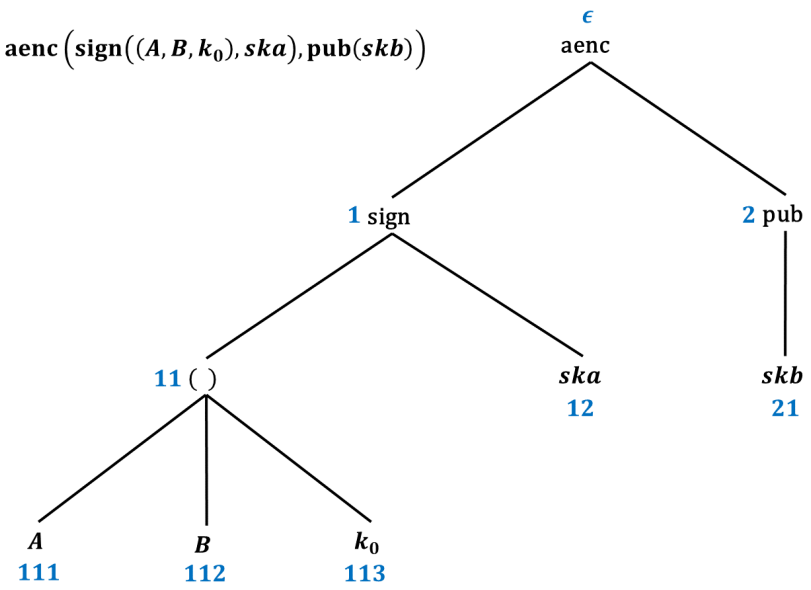

FIGURE 4 – Exemple de terme, annoté avec les positions relatives de ses sous-termes. Les chiffres en bleu correspondent à l'annotation des sous-termes, et ϵ correspond à la *racine* du terme.

3.3 Test de l'acyclicité du graphe

Une fois le graphe de dépendance calculé, notre système doit vérifier qu'il ne présente pas de cycles (soit qu'il est acyclique). Pour rappel, un cycle est une suite d'arêtes reliant un sommet à lui-même. La présence de cycles dans notre graphe indique que certaines données manipulées lors d'un échange peuvent être extraites et réinjectées par l'attaquant à un autre endroit dans le protocole. Comme il ne s'agit clairement pas d'un comportement normal, cela pourrait compromettre le protocole par la suite. Si des cycles sont présents dans le graphe, il est intéressant de les signaler pour que l'utilisateur puisse obtenir des informations pour améliorer son protocole et le rendre plus sûr. Cette partie s'intéressera à la spécification de la recherche de cycles dans le graphe de dépendance.

- **[ACY-RECHERCHE]** Le système doit permettre une recherche exhaustive des cycles du graphe ;
- **[ACY-CIBLE]** La recherche des cycles s'effectuera sur la représentation interne du graphe évoquée à la section 3.2 avec la spécification **[AC-GRAPHE-2]** ;
- **[ACY-STOCKAGE]** Chaque cycle trouvé devra être stocké dans le système. Il est important de garder la liste des sommets constituants le cycle pour de prochaines opérations.

Il n'y a pas de restrictions particulières sur l'approche envisagée pour la recherche de cycles dans le graphe de dépendance.

3.4 Affichage du graphe

Une fois le test de l'acyclicité effectué sur le graphe de dépendance, il faut produire un affichage graphique de ce dernier. Cette partie du système n'est pas à négliger : l'affichage du graphe de dépendance doit permettre une compréhension détaillée et aisée des dépendances du protocole. Cette section permettra de détailler ces différents aspects.

- **[AF-SOURCE]** L'affichage du graphe de dépendance prendra comme base la représentation interne du graphe de dépendance, ainsi que les cycles éventuellement découverts dans ce dernier ;
- **[AF-SOMMET]** L'affichage du sommet doit inclure l'étiquette donnée par le parseur correspondant à l'étape symbolisée par le dit-sommet ;
- **[AF-ARÊTES-1]** L'affichage des arêtes doit inclure la position du sous-terme concerné par la dépendance symbolisée par l'arête ;
- **[AF-ARÊTES-2]** L'affichage des arêtes doit être différent selon le type de dépendance symbolisé ;
- **[AF-ENCOMBREMENT]** L'affichage du graphe de dépendance doit limiter au maximum l'encombrement du graphe – arêtes croisées, sommet répartis de manière éparse – pour garantir un affichage clair ;
- **[AF-CYCLES]** Si jamais le graphe de dépendance présente un ou des cycles, ces derniers doivent être mis en valeur par l'affichage ;
- **[AF-LÉGENDE]** En plus d'afficher les sommets et les arêtes du graphe de dépendance, une légende permettant la compréhension du format de l'affichage devra être fournie.

3.5 Interface graphique

Une interface graphique sera fournie avec notre solution pour la manipuler avec plus d'aisance. L'utilisateur n'aura qu'à lancer un fichier exécutable et à se laisser guider par l'interface à travers les différentes étapes pour obtenir l'affichage du graphe de dépendances. Cette section va nous permettre de définir ce que nous attendons de cette interface graphique.

- **[IG-UTILISATION]** L'interface graphique devra être utilisée dès le lancement de l'application. Elle proposera dans un premier temps à l'utilisateur de choisir un fichier décrivant un protocole, puis de l'analyser et d'afficher son graphe de dépendance ;
- **[IG-FICHIER]** L'interface graphique doit permettre à l'utilisateur de spécifier le chemin menant à un fichier au format texte décrivant un protocole, pour que celui-ci soit analysé ;

- **[IG-ANALYSE]** L'interface graphique doit être capable de lancer les différents modules permettant l'analyse du fichier ;
- **[IG-AFFICHAGE-1]** L'interface graphique doit être capable de manipuler l'affichage graphique du graphe de dépendance détaillé en section 3.4 ;
- **[IG-AFFICHAGE-2]** L'interface graphique doit pouvoir afficher en plus du graphe de dépendance la description du protocole, annotée par le parseur via la spécification **[PA-ÉTIQUETTE]**.

Les spécifications ci-dessus nous paraissent être un minimum. Toutefois, on peut imaginer une plus grande interactivité entre l'utilisateur et l'affichage du graphe de dépendances, qui permettrait de renforcer le lien entre la description du protocole et le graphe de dépendance résultant. Voici une liste de souhaits de ce que nous aimerions implémenter en plus :

- **[OPTION-IG-1]** La possibilité de déplacer des sommets du graphe si le premier affichage proposé n'est pas à la convenance de l'utilisateur ;
- **[OPTION-IG-2]** La mise en évidence des lignes concernées dans la description du protocole lorsque l'utilisateur passe avec sa souris sur un arc ou un sommet.

4 Langages et technologies envisagées

Nous avons choisi d'utiliser un unique langage principal pour couvrir toutes les étapes de notre application. Cela nous permet de faciliter les transitions de l'une à l'autre et de simplifier la compréhension du code. De plus, nous avons pu trouver un langage qui correspond à toutes les critères auxquels nous devions répondre : le langage Python. En effet, ce langage contient notamment la librairie *Lark* et le package *NetworkX* ; il est également compatible avec la bibliothèque multi-plateforme *Qt* qui va nous permettre de gérer l'interface graphique. Cette bibliothèque est écrite en C++, cela sera la seule portion de code qui ne sera pas en Python. Nous détaillerons dans cette section l'utilisation que nous allons faire de ces différents modules afin de répondre aux spécifications de chaque partie du code.

4.1 Parseur

La librairie *Lark* permet une définition de grammaires de façon intuitive en utilisant le language EBNF (Extended Backus-Naur Form). Celui-ci décrit la signification de chaque symbole, Lark lie ensuite ces définitions à des appels de fonctions. On retrouve un exemple de ce processus à la figure 5.

Le parseur permet, à partir d'une primitive cryptographique, d'appeler la fonction correspondante ; le but étant de constituer une arborescence de termes, chacun caractérisé par un nom et un type.

```
calc_grammar = """
 ?start : sum
      | NAME "=" sum -> assign_var
 ?sum : product
      | sum "+" product -> add
      | sum "-" product -> sub
   product : atom
      | product "*" atom -> mul
      | product "/" atom -> div
[...]
def assign_var(self, name, value) :
   self.vars[name] = value
   return value
```

FIGURE 5 – Exemple de définition de grammaire tiré de [3]

Le fonctionnement du parseur va se dérouler comme suit :

L'application prend en paramètre un fichier texte. Un élément à gérer est la non-conformité des définitions de protocole. En effet, l'utilisateur peut oublier de refermer une parenthèse ou bien proposer des échanges normalement incompatibles avec l'honnêteté des types. La meilleure solution nous a semblé, dans ce cas là, de refuser le fichier en entrée en signalant à l'utilisateur qu'il a commis une erreur. La détection d'erreurs de syntaxe est plutôt simple dans le sens où si l'analyseur de grammaire est correctement défini, une parenthèse en trop sera par exemple un symbole qu'il ne saura pas interpréter et une en moins également.

Le parseur fait la séparation entre les différents rôles joués par les acteurs du protocole grâce à des balises (qui seront sous forme de commentaires). Chaque fois qu'il rencontre une balise, il passe au rôle suivant. Concernant les termes de ces rôles, il va joindre chaque expression du fichier correspondant à une expression décrite dans la grammaire que nous auront défini grâce à Lark avec la règle qui lui est associée. Une règle correspond à un nœud de l'arbre que le parseur va générer comme sortie. Les entrées qui lui sont associées représentent ses nœuds fils. Cette opération est répétée jusqu'à ce que le parseur ait parcouru tout le fichier : la liste de ces nœuds constitue le résultat du parseur.

4.2 Analyse et construction du graphe

Python permet la programmation orienté objet, ce que nous allons utiliser pour créer la graphe. Le résultat du parseur étant les nœuds de notre graphe, ils seront créés en tant qu'objets Python par le parseur. Ils seront gérés par le programme avec une liste, que le module d'analyse doit d'abord parcourir pour calculer les valeurs de ρ_{in} ou ρ_{out} de chaque nœud. Nous allons utiliser la structure d'arbre des termes dans les nœuds pour construire ces ensembles de façon récursive. Ainsi, nous pouvons facilement calculer les fonctions pour les sous-termes. Les ensembles résultant de ρ_{in} et ρ_{out} deviendront attributs de nos objets nœud. Après ces calculs, notre programme va parcourir la liste des nœuds pour trouver des dépendances. Pour chaque nœud, nous devons vérifier, s'il y a d'autres nœuds avec lesquelles un critère de dépendance est satisfait. Si c'est le cas, il

faut ajouter une dépendance à notre graphe de NetworkX. Pour donner un exemple, on voit dans le listing 1 le pseudocode pour la recherche des dépendances de données.

```
Global Variables:
    graph : Our NetworkX Graph

procedure dataDependencies (nodes : List of Nodes)
    for each node in nodes do
        if node.action = IN then
            for each term in node.rhoin do
                for each node2 in nodes do
                    if
                        node2.action = OUT and
                        node2.rhoout contains term
                    then
                        graph.addEdge(node, node2)
                    end
                end
            end
        end
    end
end
```

Listing 1 – Pseudocode pour la détection des dépendance de données

Ainsi, le constructeur du graphe va créer tous les arcs du graphe. Le graphe NetworkX résultant sera utilisé pour l'analyse de l'acyclicité et pour l'affichage.

4.3 Test de l'acyclicité du graphe

La recherche d'acyclicité est comprise dans les fonctionnalités du package *NetworkX*.

A partir du graphe NetworkX obtenu, nous allons pouvoir déduire l'acyclicité. Nous avons ici deux niveaux de complexité quant aux informations que nous décidons de communiquer à l'utilisateur afin de lui indiquer l'acyclicité du graphe :

- Uniquement préciser si le graphe est ou non acyclique ;
- Retourner la liste des cycles que l'outil a trouvé.

En effet, NetworkX est capable d'identifier les cycles des graphes qu'il génère et de retourner une liste de ces cycles. On retrouve ces méthodes à la figure 6. Cette option répond mieux aux spécifications données en section 3.3 et nous permet de mettre les cycles en avant lors de la phase d'affichage.

4.4 Affichage du graphe

Là encore, nous allons utiliser des propriétés de NetworkX. On peut distinguer en trois axes les solutions que nous avons sélectionnées afin de répondre au mieux aux spécifications décrites en section 3.4.

`cycle_basis` (G[, root])	Returns a list of cycles which form a basis for cycles of G.
`simple_cycles` (G)	Find simple cycles (elementary circuits) of a directed graph.
`find_cycle` (G[, source, orientation])	Returns a cycle found via depth-first traversal.
`minimum_cycle_basis` (G[, weight])	Returns a minimum weight cycle basis for G

FIGURE 6 – Exemple de méthodes NetworkX pour trouver des cycles dans une graphe

- Étiquettes : NetworkX admet n'importe quel objet en tant que sommet. Ce seront donc nos objets nœuds qui serviront de sommet lors de l'affichage du graphe. Ils devront contenir la position de l'objet dans le protocole afin que l'utilisateur puisse facilement retrouver où il en est à partir de cette position ;
  [AF-SOMMET] [AF-ARÊTES-1]
- Croisements : Grâce à un système de poids, les sommet vont se repousser ou s'attirer selon les dépendances qui les relient afin que le graphe contienne le moins possible de croisements, qui le rendraient désordonné et donc difficile à lire. Pour réaliser cette partie, nous pourrons soit utiliser des propriétés d'analyse de structure de NetworkX, soit implémenter nous même le système des poids ;
  [AF-ARÊTES-2] [AF-CYCLES]
- Coloration & forme : Certaines méthodes de NetworkX permettent de colorer des parties du graphe, on les retrouve aux figures 7 et 8. Cela va nous permettre de changer la forme des arêtes afin de les différencier en fonction des dépendances qu'elles représentent. Nous pourrons également colorer les cycles du graphe — partie intéressante pour l'utilisateur — afin qu'ils soient facilement identifiables.
  [AF-ENCOMBREMENT]

Une fois le graphe prêt à être affiché, il pourra être intégré comme figure à une interface graphique Qt afin qu'il puisse être accessible pour l'utilisateur.

`draw_networkx_nodes` (G, pos[, nodelist, ...])	Draw the nodes of the graph G.
`draw_networkx_edges` (G, pos[, edgelist, ...])	Draw the edges of the graph G.

FIGURE 7 – Exemple de méthodes NetworkX pour colorer une partie du graphe

4.5 Interface graphique

Le framework Qt est dédié à la conception d'interfaces graphiques. Son choix nous a paru idéal dans le cadre de notre application car sa compatibilité avec Python fait qu'il s'intégrera sans difficulté dans notre projet. L'interface graphique intervient de la première à la dernière étape du projet.

L'interface graphique sera divisée en deux parties : une où sera affiché le protocole, l'autre où sera affiché le graphe. Dans un premier temps notre interface devra proposer une

- **nodelist** (*list, optional*) – Draw only specified nodes (default G.nodes())
- **node_size** (*scalar or array*) – Size of nodes (default=300). If an array is specified it must be the same length as nodelist.
- **node_color** (*color or array of colors (default='#1f78b4')*) – Node color. Can be a single color or a sequence of colors with the same length as nodelist. Color can be string, or rgb (or rgba) tuple of floats from 0-1. If numeric values are specified they will be mapped to colors using the cmap and vmin,vmax parameters. See matplotlib.scatter for more details.
- **node_shape** (*string*) – The shape of the node. Specification is as matplotlib.scatter marker, one of 'so^>v<dph8' (default='o').

FIGURE 8 – Exemple de paramètres dans NetworkX permettant de modifier l'aspect d'une partie du graphe

zone d'entrée (côté protocole) dans laquelle l'utilisateur pourra renseigner le chemin vers le fichier texte contenant la description du protocole qu'il souhaite tester. La page contiendra également un *Widget* qui permettra de lancer la génération du graphe. Le fichier texte sera alors renseigné au parseur qui procédera comme détaillé dans la section 4.1. Toute cette partie correspond au mock-up en figure 9.

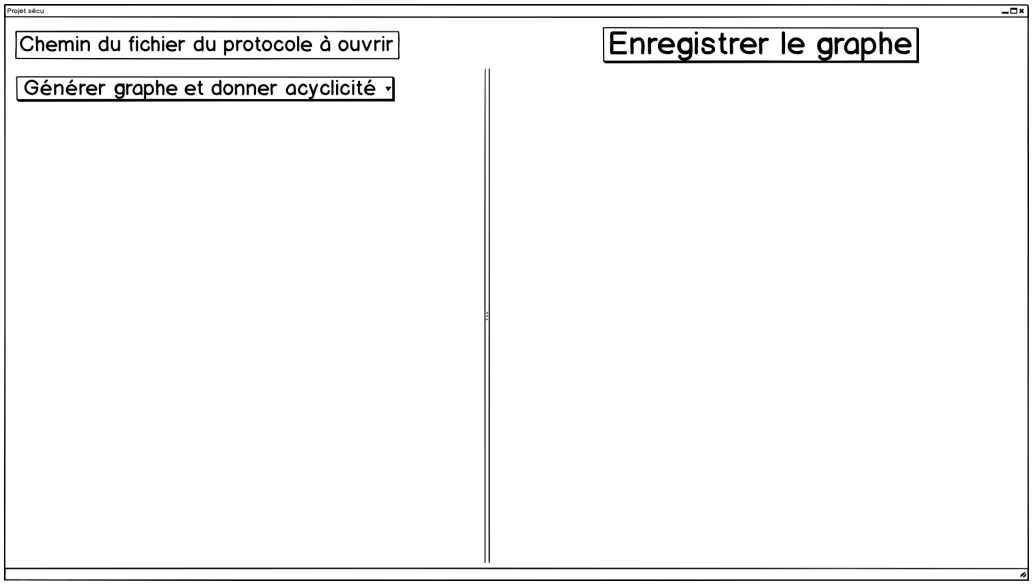

FIGURE 9 – Mock-up de la page d'accueil de l'application

Une fois arrivé à la phase d'affichage, l'application affichera le détail du protocole à gauche de la fenêtre et le graphe obtenu au terme de la section 4.4 à droite. Il sera alors

possible de sauvegarder le graphe ou de charger un autre fichier grâce aux Widgets situés dans la barre au sommet de l'interface. Cette dernière partie est illustrée figure 10.

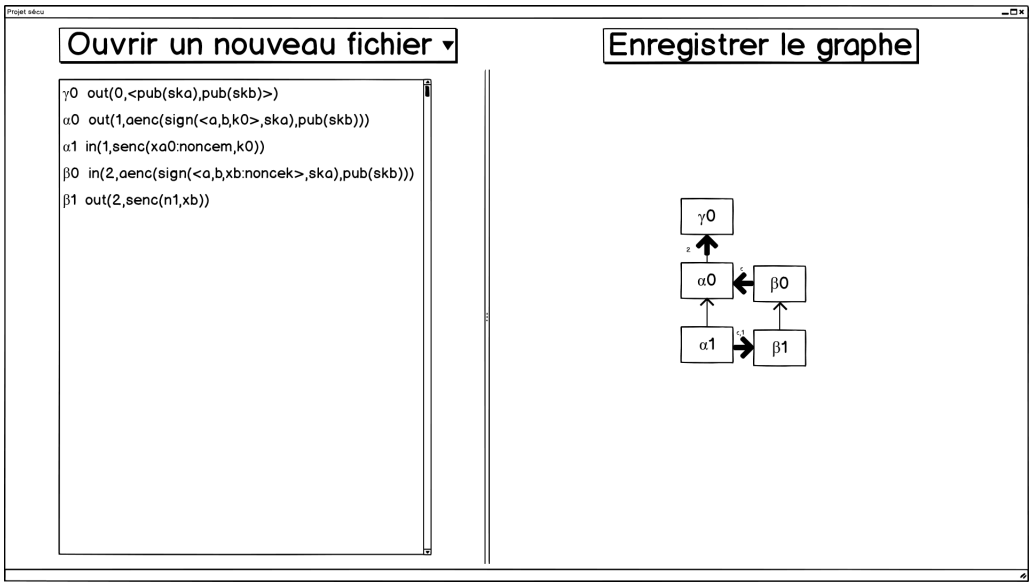

FIGURE 10 – Mock-up de la page de résultats de l'application

5 Rétrospectives et perspectives

Pendant les derniers semaines, nous avons essayé différentes solutions pour les différentes parties de notre programme. Nous avons testé, si certaines langages ou librairies sont bien adaptés à notre problème. Les résultats de nos recherches nous ont aussi données une vision plus précise sur les risques que nous avions élaboré dans le rapport de pré-étude.

5.1 Évolution des choix de technologie

Cette section a pour but de détailler comment nous sommes arrivées à nos choix de technologies finales. Nous expliquons pourquoi certaines technologies nous paraissent peu adaptées. Les points forts des languges et technologies retenus sont décrits dans la section 4.

5.1.1 OCaml

Le parseur que nous devons implémenter existe déjà : un ancien doctorant de Stéphanie Delaune, Antoine Dallon, avait programmé un parseur dans le langage OCaml. Ce parseur fait partie du programe SAT-Equiv, qui est décrit dans [4] et [5]. Nous avons eu la possibilité de faire notre projet entièrement en OCaml, en utilisant le parseur existant. La raison principale de refaire tout en Python était l'absence de librairies puissantes pour

les graphiques en OCaml. Bien que OCaml a des librairies pour l'affichage graphique, ils sont assez basiques et peu adaptés pour travailler avec des graphes. Prenant en compte qu'une bonne qualité d'affichage de notre graphe de dépendance était un des objectifs de notre projet, nous avons craint de perdre beaucoup de temps avec l'implémentation des algorithmes qui sont déjà codés dans autres langages de programmation. Nous avons quand même analysé le code source du parseur et nous allons nous en inspirer pour créer le nôtre.

5.1.2 Prolog

Pour l'analyse du résultat du parseur et pour la création du graphe de dépendances, l'utilisation du langage Prolog nous paraissait pertinente. L'idée était de transformer le protocole en une base de connaissances, la combiner avec les règles qui définissent notre graphe de dépendance et interroger cette combinaison avec l'interpréteur de Prolog pour trouver les dépendances. Un avantage de cette solution serait l'absence d'algorithmique : nous ne devrions pas décrire comment il faut faire pour obtenir ρ_{in}, ρ_{out} et les dépendances, c'est le moteur de recherche de Prolog qui s'en occuperait. Par contre, il est très difficile de réaliser l'affichage en Prolog. Aussi le parseur posait problème, car la lecture des fichiers en Prolog peut être compliquée. En outre, chaque protocole devrait génerer sa propre base d'axiomes. En conséquent, notre base à interroger aurait été très dynamique et cette programmation dynamique en Prolog est difficile à maîtriser.

5.1.3 Idée de solution patchwork

En constatant que certaines modules de notre solution pouvaient bien fonctionner avec certains paradigmes de programmation, nous avons eu l'idée d'utiliser plusieurs langages de programmation pour notre projet. Par exemple, ils existent des librairies Python, qui permettent d'exécuter des requêtes Prolog. Nous les aurions utilisées pour créer notre Graphe que nous aurions traité plus tard avec Prolog.

Une raison principale de ne pas poursuivre une telle solution patchwork était une portabilité très restreinte. Un utilisateur qui aurait voulu installer notre solution aurait eu besoin de beaucoup de dépendances, paquets Python, d'un interpréteur Prolog, etc.

5.1.4 Alternatives pour l'affichage

Nous avons cherché différentes solutions pour l'affichage du graphe. Une solution était QuickQuanava [6], une librairie pour l'affichage des graphes, qui reposait sur QtQuick et C++. Elle possède beaucoup de possibilités pour modifier le graphe à posteriori, mais intègre peu de fonctionnalités pour un affichage clair.

Une autre solution était la librairie JGraph [7], qui reposait sur Java et la librairie Swing. Mais cette dernière étant très vieille et sans grandes mises à jour depuis 2006, nous avons cherchés d'autres solutions. Il existe une librairie plus moderne, JGraphT [8], mais elle ne possède pas de fonctionnalités d'affichage.

Une autre solution très puissante est Gephi [9]. Gephi est une application pour afficher des graphes écrite en Java. Mais les modules qui contiennent les algorithmes sur les graphes peuvent être téléchargés et intégrés dans un autre projet Java. (Ces modules sont appellés « toolkit »). Ils sont quand même très complexes et nous trouvons difficile d'intégrer notre structure de classes dans la structure imposée par le toolkit.

5.2 Évolution des risques

Au rapport précédent, nous avions évoqué certains risques liés aux technologies employées, à la gestion de l'équipe et à certains de nos questionnements. Une figure relative est présentée en figure 11. Ces risques et les craintes associées ont évolué depuis le début du projet. Les craintes que l'on pouvait avoir sur les technologies utilisées sont en partie dissipées. Certes l'utilisation de librairie jusqu'alors inconnue à nos yeux ainsi que de l'API Qt nécessiteront un certain temps d'apprentissage. Mais l'utilisation du langage Python est une nouvelle rassurante, étant donné que c'est un langage avec lequel nous avons eu le temps de nous familiariser au long de notre cursus. Concernant la gestion de l'équipe, le départ de certains membres du groupe reste un enjeu fort, en particulier après cette phase d'études, qui nous a montré l'ampleur du travail à fournir. Mais d'un autre point de vue, la proximité du langage avec nos connaissances nous rassure quand à la faisabilité du projet.

Au niveau de nos questionnements, notre principale appréhension résidait dans la question suivante : "Qu'est-ce qu'un beau graphe ?". Cependant il nous a semblé que l'optimisation esthétique de l'affichage du graphe ne sera qu'à traiter dans un second temps, car les librairies déjà existantes proposent des affichages en règle générale satisfaisant. Ces aspects nous semblent désormais moins inquiétants que ce qu'ils pouvaient être de prime abord.

6 Conclusion

À travers les différentes sections de ce rapport, nous avons exposé l'avancement du projet. En effet, désormais plus qu'un simple sujet de travail, il s'agit pour nous d'un projet concret que nous voyons dors et déjà comment entamer. Les premières spécifications étant faites, il nous semble désormais possible d'aller plus loin encore dans nos questionnements techniques afin que nous soyons prêts à réaliser la solution du problème posé. Il reste quelques problèmes à résoudre, notamment au niveau de l'utilisation des librairies dans nos modules et des algorithmes à utiliser. Cependant, ces points pratiques seront bientôt à leur tour étudiés.

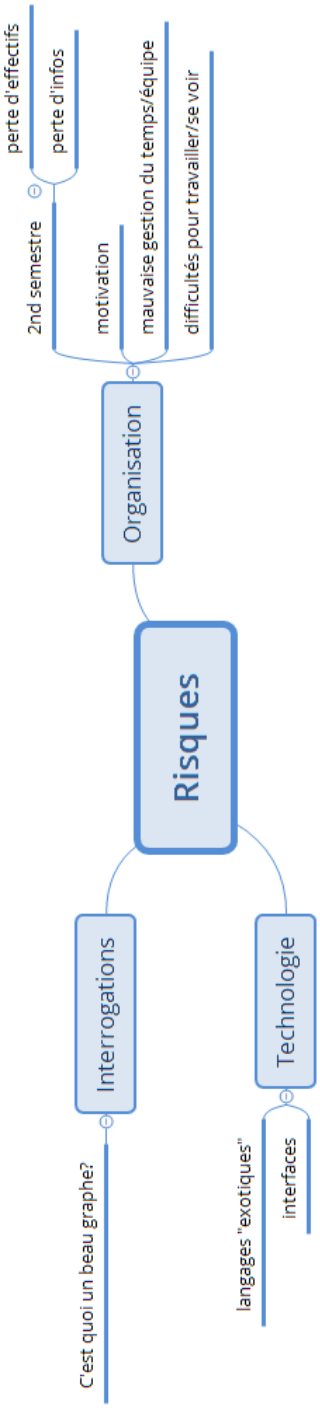

FIGURE 11 – Risques détectés, sous forme de carte heuristique

Bibliographie

[1] Stéphanie Delaune. « Deciding secrecy for a large class of security protocols ». In : (2019). URL : https://people.irisa.fr/Stephanie.Delaune/notes-projetINFO4.pdf (visité le 23/10/2019).

[2] Sophie Delannoy et al. « Vers une vérification efficace des protocoles de sécurité Rapport de pré-étude - Projet 4INFO ». In : (oct. 2019).

[3] Erez Shinan. *Exemple of use of the Lark parser*. 2017. URL : https://github.com/lark-parser/lark/blob/master/examples/calc.py (visité le 21/11/2019).

[4] Véronique Cortier, Antoine Dallon et Stéphanie Delaune. « Efficiently Deciding Equivalence for Standard Primitives and Phases ». In : *Computer Security*. Sous la dir. de Javier Lopez, Jianying Zhou et Miguel Soriano. Cham : Springer International Publishing, 2018, p. 491-511. ISBN : 978-3-319-99073-6.

[5] V. Cortier, A. Dallon et S. Delaune. « SAT-Equiv : An Efficient Tool for Equivalence Properties ». In : *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. Août 2017, p. 481-494. DOI : 10.1109/CSF.2017.15.

[6] *QuickQanava*. 2019. URL : http://cneben.github.io/QuickQanava/index.html (visité le 27/11/2019).

[7] *jgraph/jgraphx*. 2019. URL : https://github.com/jgraph/jgraphx (visité le 27/11/2019).

[8] *JGraphT*. 2019. URL : https://jgrapht.org/ (visité le 27/11/2019).

[9] *Gephi Toolkit*. 2019. URL : https://gephi.org/toolkit/ (visité le 27/11/2019).