Stage de fin de L3

Ecole normale supérieure
Département d'informatique

Rendre les contrôles d'accès sur les réseaux sociaux compréhensibles

Rapport de stage

Théo Delemazure
theo.delemazure@ens.fr

Juin-Août 2018

Sous la direction de
Pierre Bourhis
Romain Rouvoy
Walter Rudametkin

Inria Lille
Parc scientifique de la Haute Borne
40, avenue Halley
59650 Villeneuve d'Ascq - France

Table des matières

1 Introduction

1.1 Présentation de l'équipe et du projet momentum

Dans le cadre de mon année de L3 en informatique à l'ENS, j'ai été amené à faire un stage à l'Inria Lille - Nord Europe dans l'équipe de recherche Spirals (également membre de l'UMR CNRS CRIStAL) spécialisée dans l'étude des systèmes distribués et l'ingénierie logicielle.

Mon stage fait quant à lui partie du projet Momentum *"Sécurité des données et transparence des algorithmes"* du CNRS porté par Pierre Bourhis.

1.2 Motivations du stage

La sécurité des données et la vie privée sur Internet sont des sujets préoccupants, comme en témoignent les différents scandales et plaintes à l'encontre de Facebook et d'autres réseaux sociaux depuis le début de l'année 2018 [3].

Alors que ces réseaux tentent de répondre aux scandales en proposant aux utilisateurs de meilleurs moyens de sécuriser leurs données, certains chercheurs ont constaté que vérifier une politique de contrôle d'accès est dans les faits beaucoup plus délicat que d'apparence. Par exemple, si je poste une photo sur Facebook avec comme audience *My friends* mais qu'une personne qui n'est pas mon ami est identifiée dessus, alors cette personne aura accès à cette photo, et potentiellement tous ses amis avec.

Les motivations du stage étaient donc de formaliser les différents mécanismes de contrôle d'accès aux données des réseaux sociaux (notamment Facebook) et de comprendre l'impact de ces mécanismes sur la diffusion réelle des données personnelles. Il fallait pour cela analyser les différentes API Facebook pour y trouver les règles de contrôle d'accès ainsi que de potentielles failles. Puis, la seconde partie du stage s'intéressait à la conception d'un algorithme permettant d'expliquer et de faciliter l'utilisation des contrôles d'accès. Les recherches que nous avons menées et la curiosité m'ont cependant poussé à explorer la thématique de l'accessibilité aux données et des liens inter-réseaux sociaux.

1.3 Deux axes de recherches distincts

Pendant ce stage, nous avons exploré deux grands axes de recherche autour des contrôles d'accès et de l'accessibilité des données.

D'un côté, nous avons modélisé les réseaux sociaux et leurs contrôles d'accès afin de se poser des questions formelles sur ces contrôles d'accès et tenter de les résoudre.

De l'autre côté, nous avons cherché à exploiter ces règles de contrôles d'accès. En effet, nous avons été amené pendant nos recherches à constater certaines failles liées à la désanonymisation et au ciblage des utilisateurs dont les contrôles d'accès sont mal choisis.

2 Expliquer et vérifier les contrôles d'accès

Pour construire le formalisme de réseau social sur lequel nous devrons nous appuyer, nous nous sommes notamment inspirés du modèle Webdamlog [10, 4], et plus généralement du formalisme de datalog distribué qui permet de modéliser les échanges entre les utilisateurs. L'intérêt du Webdamlog réside notamment dans sa gestion des contrôles d'accès, qui est construite pour des environnements peer-to-peer. Notre modèle nécessite l'utilisation de clés primaires et réutilise des mécanismes de mise à jour avec clé primaire du papier *Collaborative data-driven workflows* [5].

L'accessibilité aux données est un domaine qui intéresse les chercheurs, notamment la déduction et l'inférence de données à partir de règles logiques [6].

2.1 Création des outils : notre formalisme

Nous avons donc construit un modèle pour représenter les règles dans les réseaux sociaux et j'y ferai référence dans la suite comme SocialNetworkLog (SNL).

2.1.1 Modelisation d'un reseau social : SocialNetworkLog

La modélisation des réseaux sociaux que nous avons choisie se divise en trois couches : tout d'abord, la base de données centralisée, détenue par le réseau, qui contient les tables. Par exemple, la table contenant

les photos serait notée `Photos(id,author,url)` où `id` correspond à la clé primaire de la photo, et les autres éléments sont des variables/constantes correspondant à des attributs.

Deuxièmement, les vues sur ces bases de données qui sont modifiables par les utilisateurs et qui sont paramétrées par les utilisateurs. La vue des photos postées par un utilisateur serait donc notée `MyPhotos[author](id,url)`. Ici, on donne le nom de l'utilisateur en paramètre et on obtient une vue de ses photos en sortie.

Enfin, il y a les vues statiques qui sont définies en fonction des autres vues et tables du réseau. Par exemple, la vue sur les photos où apparaît un utilisateur `PhotosAbout[user](id,url)`).

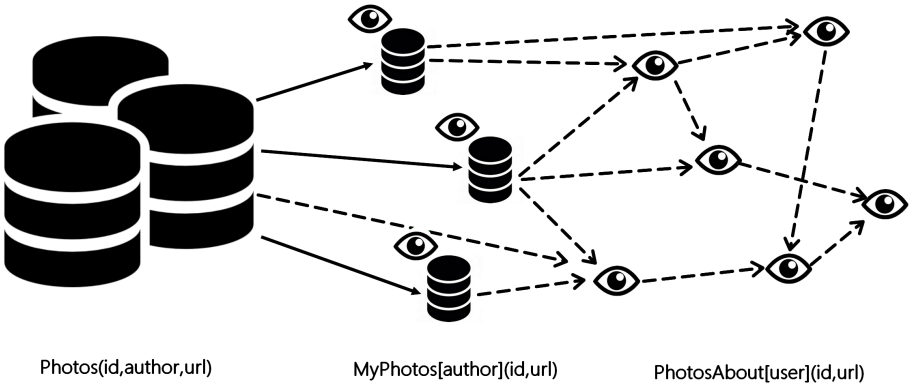

FIGURE 1 – Schéma des 3 couches de la modélisation SOCIALNETWORKLOG

Définition des relations Le réseau social possède une base de données centralisée où sont stockées toutes les données. Une relation extensionnelle est caractérisée par un nom de table et une arité $(name; arity)$.

Définition 1 (Atome). *Un atome de relation extensionnelle* $(Relation; n)$ *est noté* $Relation(id, x_2^n)$ *où* id *est la clé primaire du tuple (i.e. son identifiant unique) et* x_2^n *est une liste de variables/constantes correspondant aux attributs de la relation. Toute relation a une clé primaire qui correspond au premier attribut.*

Définition des vues Les utilisateurs ont accès aux données uniquement au travers de vues sur la base de données centralisée. On les note $\texttt{View}[\bar{k}](\bar{x})$ où $\bar{x}$ correspond aux attributs de retour de la vue et $\bar{k}$ correspond aux attributs paramètres de la vue.

L'ensemble des éléments de cette vue accessibles pour un utilisateur $user_2$ est noté $\texttt{View}[\bar{k}]_{user_2}(\bar{x})$. On note $_F$ la vue système qui permet d'avoir accès à toutes les données, peu importe les contrôles d'accès de l'utilisateur.

Comme dit plus haut, on distingue deux types de vues : les **vues modifiables** et les **vues statiques**.

Définition 2 (Vue modifiable). *Une vue modifiable est une projection/sélection simple d'une relation extensionnelle sur un utilisateur. Elle est paramétrée par ce même utilisateur.*
$View[user](id, \overline{x_0}) \leftarrow R_1(id, \overline{y_1}), R_2(\overline{y_2}), \ldots, R_n(\overline{y_n}), v_1 \theta v_1^0, \ldots, v_l \theta v_l^0$
Les $\overline{y_i}$ *sont des variables (pour les attributs) et les inégalités sont imposées ensuite entre des couples de variables.*

Définition 3 (Vue statique). *Une vue statique est définie comme requête conjonctive d'autres vues et de relations. On note une règle de création de vue statique*

$$\begin{aligned} View[\bar{k}](\bar{x}) \leftarrow\ & Rel_1(\overline{x_1}); \ldots; Rel_k(\overline{x_k}); \\ & View_{k+1}[\overline{k+1}](\overline{x_{k+1}}); \ldots; View_n[\bar{n}](\overline{x_n}), \\ & v_1 \theta v_1^0, \ldots, v_l \theta v_l^0 \end{aligned}$$

avec des inégalités entre deux variables ou entre une variable et une constante.

Notations et ensembles On note R l'ensemble des relations, V l'ensemble des vues, V^m l'ensemble des vues modifiables et V^s des vues statiques du réseau. On note également U l'ensemble des utilisateurs du réseau pour une instance donnée.

Mise a jour des vues modi ables Puisque les vues modifiables sont de simples projections/sélections qui conservent l'attribut servant de clé primaire, on peut ajouter et supprimer des éléments au travers de la vue facilement (cf. [5]).

De nition 4 (Ajout). *Un ajout fait par* $user_2$ *dans une vue modifiable paramétrée par* $user_1$ *est notée* $A[user_1]_{user_2}(\overline{x}) := +_{user_2} View[user_1](id,\overline{x})$.

De nition 5 (Suppression). *Une suppression faite par* $user_2$ *dans une vue modifiable paramétrée par* $user_1$ *est notée* $S[user_1]_{user_2}(\overline{x}) := -_{user_2} View[user_1](id,\overline{x})$.

De nition 6 (Séquence d'action). *Une séquence d'actions est exécutable selon une règle de la forme* $A_1[u_1](\overline{x_1}), \ldots, A_n[u_n](\overline{x_n})\ , S_1[u'_1](\overline{x_1^0}), \ldots, S_n[u'_n](\overline{x_n^0})\ \grave{}\ View_1[\overline{k_1}](\overline{y_1}), \ldots, View_n[\overline{k_n}](\overline{y_n})$ *où les variables de la tête apparaissent dans le corps où sont des variables libres fournies par l'utilisateur selon les règles de l'interface d'insertion.*

2.1.2 Modelisation des contrôles d'acces

Il faut maintenant définir comment sont accessibles ces données en fonction des contrôles d'accès choisis par les utilisateurs. Les contrôles d'accès des tables de la base de données centralisée sont choisis par le réseau : certaines tables sont publiques (`users, pages, ...`) et d'autres sont privées (`photos, tag, ...`).

Relations de contrôle d'acces Les contrôles d'accès peuvent se faire au niveau de la vue entière (**coarse-grained**) ou au niveau du tuple (**ne-grained**). On a donc deux tables de contrôle d'accès :

De nition 7 (Contrôle d'accès *coarse-grained*). $aclv[u](View,Rule,\text{PRIV})$ *avec le nom de la vue, la règle de contrôle d'accès et le privilège (*READ, INSERT,DELETE*).*

De nition 8 (Contrôle d'accès *fine-grained*). $aclvf[u](View,Rule,id)$ *avec le nom de la vue, la règle de contrôle d'accès et la clé du tuple concerné.*

Relations de blacklist On ajoute à cela la notion de blacklist : il y a une **blacklist ne-grained** pour les éléments *fine-grained* (par exemple, les personnes qui ne peuvent pas voir une photo), et une **blacklist globale** où l'utilisateur est totalement bloqué. On les note respectivement `blf[u](View,G,id)` (avec le nom de la vue, l'ensemble de personnes black-listées et la clé de l'élément concerné) et `bl[u](u')`.

On note `include(user,E)` la fonction qui vaut `True` si et seulement si l'utilisateur user appartient à l'ensemble `E`.

Veri cations des contrôles d'acces L'autorisation pour un utilisateur $user_2$ d'écrire dans la vue modifiable `View` paramétrée par $user_1$ est donné par `aclv[`$user_1$`](View,Rule,`INSERT/DELETE`), Rule[`$user_1$`](`$user_2$`)`. Il suffit alors d'ajouter pour toute règle de mise à jour ces éléments dans la requête conjonctive.

Il faut également ajouter le contrôle de *blacklist* globale (dans les deux sens : aucun des deux ne doit être bloqué par l'autre). Dans la suite, on considère que cette règle est toujours implicite.

De même, il y a deux conditions pour lire un tuple avec contrôle d'accès *fine-grained* de clé `id` dans la vue modifiable `View` : vérifier la règle de contrôle d'accès et ne pas faire partie de la *blacklist*.

$$\begin{aligned} \texttt{View[user}_1\texttt{]}_{user_2}\texttt{(id,}\overline{x}\texttt{)} \grave{}\ & \texttt{View[user}_1\texttt{]}_F\texttt{(id,}\overline{x}\texttt{),} \\ & \texttt{aclvf[user}_1\texttt{]}_F\texttt{(View,Rule,id),} \\ & \texttt{Rule[user}_1\texttt{]}_F\texttt{(user}_2\texttt{),} \\ & \texttt{blf[user}_1\texttt{]}_F\texttt{(View,E,id),} \\ & : \texttt{include(user}_2\texttt{,E)} \end{aligned}$$

Pour les vues avec contrôle d'accès *coarse-grained*, on crée simplement une règle de la forme suivante pour se ramener au cas *fine-grained* :

$$\begin{aligned} \texttt{aclvf[user}_1\texttt{](View,Rule,id)} \grave{}\ & \texttt{aclv[user}_1\texttt{](View,Rule,READ),} \\ & \texttt{View[user}_1\texttt{](id,}\overline{y}\texttt{)} \end{aligned}$$

Enfin, **les contrôles d'acces sur les vues statiques** dépendent simplement des contrôles d'accès de leur(s) règle(s) de création.

Mise a jour des contrôles d'acces L'ensemble des règles de contrôle d'accès possibles pour la vue $View$ et le privilège PRIV est donné par un ensemble de règles $\Omega_{View;\text{PRIV}}$.

Les contrôles d'accès coarse-grained sont initialisés par le réseau lui-même car déjà présents à l'inscription. La modification de ces contrôles d'accès est une simple séquence de mise à jour avec vérification que la nouvelle règle appartient à l'ensemble $\Omega_{View;\text{PRIV}}$.

Pour les éléments fine-grained, l'utilisateur peut choisir la règle de contrôle d'accès lorsqu'il insère l'élément dans la vue `View`. La modification se fait de la même façon que dans le cas précédent.

On note C l'ensemble des **regles de production de vue** et C^{acl} l'ensemble des **regles de contrôle d'acces**. Une règle de contrôle d'accès est définie par la liste des vues de la règle et les inégalités entre les attributs de ces vues (= et ~~6=~~).

On note F un réseau $F = (R; V^m; V^s; C; C^{acl}; \Omega; A\ )$ où Ω est une fonction qui a une vue modifiable V et un privilège PRIV associe $\Omega_{V;\text{PRIV}}$ et A est l'ensemble des séquences d'actions du réseau.

2.1.3 Representation par des graphes orientes

La representation par graphe des règles SNL permet de visualiser les chemins que peut avoir parcouru une donnée avant d'arriver dans une vue. Ces graphes sont composés de 3 ensembles : $G = (N; A; V)$.

Un noeud $n \in N$ représente une règle. Les vues $v \in V$ sont des ensembles de noeuds (l'ensemble de leurs règles de formations). Enfin, une arête $a \in A \quad V \quad N$ relie une vue dans le corps d'une règle au noeud correspondant à cette règle.

Cette représentation étant surtout utile pour se représenter l'aspect global d'un réseau, je ne vais pas en expliquer tous les détails. De plus, cette représentation génère des pertes d'informations. En revanche, elle permet de souligner que le modèle est *non récursif* avec des graphes acycliques.

A titre d'exemple, la figure 2 montre que les données de la vue de $u_v \in U$ sur `PhotosAbout[`u_1`]` proviennent de photos publiées par n'importe quel utilisateur de FACEBOOK et accessibles par u_v (donc de la table `myPhotosAccessibles[`u_2`]`) sur lesquelles u_1 est taggé. Les photos accessibles sont définies comme les photos sur lesquelles l'utilisateur u_v a les droits d'accès en lecture OU sur lesquelles un de ses amis u_3 est taggé et l'a ajouté à l'audience.

Ainsi, si Bob poste une photo sur laquelle apparaît Alice et Marie, que Charlie est ni ami avec Bob ni Alice mais avec Marie, *il pourra potentiellement voir la photo de Bob ainsi que le fait que Alice soit taggée.*

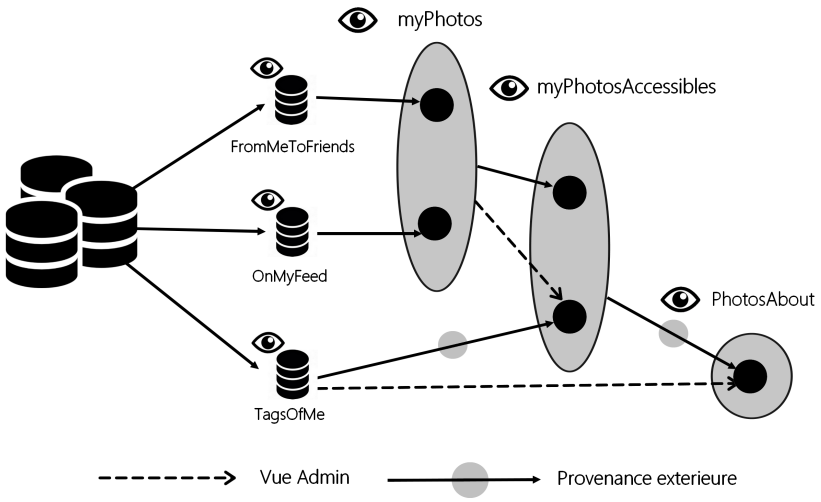

FIGURE 2 – Une partie de la représentation en graphe du réseau FACEBOOK

2.2 Les problèmes de fuites de données

Maintenant que le modèle est posé, nous nous sommes demandé s'il était possible de résoudre certains problèmes concernant les fuites de données, leurs complexités, et sous quelles hypothèses.

2.2.1 De nition des problemes

Fonction de contrôles d'acces Un ensemble de contrôles d'accès (en lecture) valide est une fonction qui, à chaque vue modifiable, associe une règle de contrôle d'accès valide. Plus formellement, $acl_u : V^m \quad ! \ C$ tel que $8V iew \ 2 \ V^m; acl_u(V iew) \ 2 \ \Omega_{V iew;\text{READ}}$.

On peut définir les contrôle d'accès maximum acl_{max} qui à chaque vue associe le contrôle d'accès le plus permissif autorisé par le réseau (généralement *Public*).

Politique de contrôles d'acces Une politique de contrôles d'accès pour une vue $V iew$ et un utilisateur u est donné par une règle de contrôle d'accès $Q_{V iew} \ 2 \ C$ tel que u^0 a le droit de voir des données au travers de la vue $V iew$ *si et seulement si* il vérifie $Q_{V iew}(u; u^0) = True$.

Soit un utilisateur u avec des contrôles d'accès acl_u sur ses vues modifiables et ayant une politique de contrôle d'accès Q où $Q_{V iew}$ est la politique concernant la vue (non forcément modifiable) $V iew$. On rappelle que F correspond à un réseau.

Fuite de donnees Connaissant l'instance I de la base de données du réseau, on dit qu'il y a une *fuite de données (*DL*)* sur la vue $V iew$ pour u par rapport à sa politique $Q_{V iew}$ si un utilisateur tierce u^0 qui ne vérifie pas la règle $Q_{V iew}$ a accès à certaines données de $V iew$ selon les contrôles d'accès de u. Plus formellement, on note :

$$dataLeakage[V iew; I; F; Q_{V iew}] \ , \ 9u^0 \ 2 \ U; V iew[u](I) \ _{u^0} \ 6= \ ; \\ \wedge \ : Q_{V iew}(u; u^0)$$

Trou de securite Pour un réseau F et des contrôles d'accès de u, il existe *un trou de sécurité (*SG*)* si il existe une instance I de la base de données cohérente avec les contrôles d'accès de u et telle que I induit une fuite de données comme définit précédemment. Plus formellement, on note :

$$SecurityGap[V iew; acl_u; F; Q_{V iew}] \ , \ 9I; aclv[u](I) = acl_u \\ \wedge \ dataLeakage[V iew; I; F; Q_{V iew}]$$

Fuite de donnees possible De même, si u a une vue partielle J sur l'instance I de la base de données, on dit qu'il y a *fuite de données possible (*PDL*)* si il existe une instance I cohérente avec la vue partielle J telle que I induit une fuite de données. Plus formellement, on note :

$$possibleDataLeakage[V iew; J; F; Q_{V iew}] \ , \ 9I; I \quad u = J \\ \wedge \ dataLeakage[V iew; I; F; Q_{V iew}]$$

Fuite par action possible Pour une séquence d'action $A \ 2 \ A$, une instance d'action I_A (i.e. les données ajoutées dans la table par l'utilisateur) et une vue J sur l'instance I de la base de données, on note $A(I_A; J)$ la vue sur la nouvelle instance de la base de données après avoir effectué la séquence d'action A. On dit alors qu'il y a une *fuite par action possible* suite à l'action A avec une instance d'action A_I sur la base de données s'il n'y avait pas de fuite de données avant cette action et que l'action peut entraîner une fuite de données. Plus formellement, on note :

$$ActionLeakage[V iew; A; I_A; J; F; Q_{V iew}] \ (\ ) \ : PossibleDataLeakage[V iew; J; F; Q_{V iew}] \\ \wedge \ PossibleDataLeakage[V iew; A(I_A; J); F; Q_{V iew}]$$

2.2.2 Complexite des problemes

Après avoir défini les problèmes, j'ai cherché leurs complexités ainsi que des algorithmes permettant de les résoudre, selon certaines hypothèses de départ (Figure 3).

Hypotheses	DL	SG	PDL	AL
Cas général	PTIME	NEXPTIME-C	NEXPTIME-C	NEXPTIME-C
Q_{View} 2 CQ	PTIME	NEXPTIME-C	NEXPTIME-C	NEXPTIME-C
Profondeur bornée	PTIME	NPTIME-C	NPTIME-C	NPTIME-C

FIGURE 3 – Complexités des problèmes selon différentes hypothèses

Complexite de la fuite de donnees Nous allons d'abord démontrer que le problème de *dataLeakage* se résout en temps polynomial. Soit une instance I de la base de données et un utilisateur u avec une politique Q_{View} pour la vue $View$.

Theoreme 1. *$DataLeakage[View; I; F; Q_{View}]$ est PTIME selon la taille des entrées.*

Sketch of proof. Pour tout utilisateur u^0 apparaissant dans l'instance I du réseau social décrit par F, on calcule facilement (en tant qu'admin) si u^0 vérifie Q_{View} et s'il voit des données dans $View$. On a donc :

- On exécute simplement les règles une à une en augmentant en profondeur à chaque fois jusqu'à atteindre la vue $View$. Le nombre de calcul est donc linéaire selon le nombre de règles du réseau.
- Le calcul de $Q_{View}(u; u^0)$ est quant à lui polynomial en la taille de Q_{View}
- On fait ça pour chaque u 2 $User(I)$, donc on reste polynomial en la taille de I

Le problème est donc PTIME en la taille des données. □

Profondeur d'un reseau On introduit la *notion de rang* qui sera utile pour la suite. Pour chaque vue, on peut définir un rang $Rk(V)$, le rang d'une relation de la base de données est de 0, celui d'une vue modifiable est de 1, et celui d'une vue statique est défini par la formule suivante :
$maxfRk(V^0) + 1j$ Il y a une règle avec V dans la tête et V' dans le corps g

La profondeur d'un réseau est définie comme le rang maximum de ses vues.

Complexite du trou de securite dans le cas general Le papier *"The Impact of Virtual Views on Containment"* [7] démontre que le problème de containment de requêtes datalog non-récursives est *co-NEXPTIME-complete*. On dit qu'une requête Q_1 est contenue dans une requête Q_2 si pour toute entrée de la base de données, le résultat de Q_1 est contenu dans celui de Q_2. Autrement dit, $8x; Q_1(x)$) $Q_2(x)$.

On montre dans la preuve suivante que l'on peut réduire le problème de trou de sécurité au problème de containment en transformant les règles DATALOG en règle SOCIALNETWORKLOG.

Theoreme 2. *Dans le cas général, $SecurityGap[View; acl_u; F; Q_{View}]$ est NEXPTIME-Complet*

Sketch of proof. On montre tout d'abord que le problème est NP-Hard en le réduisant au problème de containment. Soit deux requêtes Datalog booléenne Q_1 et Q_2. Chacune de ces requêtes est composée d'*input predicate*, d'*intensionnal predicate* et de règles non récursives. Les *input predicates* de Q_1 et Q_2 sont partagés.

Pour chaque *input predicate* R d'arité n, on créé une relation dans la base de données centralisée d'arité $n+1$: On rajoute une colonne pour la clé utilisateur. On crée également une vue modifiable d'arité n liée à cette relation.

Pour chaque *intensional predicate* R d'arité n de Q_1 et Q_2, on créé une vue statique avec les mêmes règles SocialNetworkLog que les règles datalog et en lecture "administrateur" ($_F$).

On fait de même pour les prédicats $Goal_1[u]$ et $Goal_2[u]$ en réécrivant leur règle en SOCIALNETWORKLOG. On observe alors que ces deux vues contiennent l'élément $True$ si et seulement si leur requête respective est vérifiée.

On crée une vue modifiable fraîche `Test` et on donne son contrôle d'accès : $acl[u^0](Test; u)$ ` $Goal_1[u](True)$. On choisit comme politique de contrôle d'accès la politique suivante : $Q_{View}(u^0; u)$ ` $Goal_2[u](True)$. Il y a une fuite de données si et seulement si il est possible d'atteindre le $Goal_1$ et pas le $Goal_2$ pour une même instance, soit $:(Q_1 \vee Q_2)$. Or on sait que le problème d'inclusion de règle datalog est Co-NEXPTIME-Complet donc le problème de trou de sécurité est NEXPTIME-Hard.

On montre maintenant que le problème est NEXPTIME. Pour chaque instance canonique J on vérifie si $detaLeakage[View; I; F; Q_{View}]$ est vérifié. Pour calculer la taille de ces instances, on retrace pour chaque vue et contrôle d'accès une origine possible d'un tuple. On note k la profondeur du réseau. Alors pour chaque vue, on crée moins de $(jCj \quad maxf$ nombre de vue dans le corps de la règle $g)^k = N_1(C)^k$ tuples où $N_1(C)$ et k dépendent des entrées. On a donc au final $jIj < jVj \quad N_1(C)^k$, donc I est de taille au plus exponentielle. Comme le calcul de *dataLeakage* est polynomial, alors *SecurityGap* est NEXPTIME en la taille des données, et par conséquent NEXPTIME-Complet. □

On peut montrer de manière similaire que les autres problèmes répondent à cette complexité.

Complexite pour un reseau de profondeur bornee Supposons maintenant qu'il existe un entier N tel que tout réseau en paramètre du problème a une profondeur bornée par N (ce qui ne semble pas absurde dans le cas de FACEBOOK par exemple). On montre que dans ce cas la complexité est nettement inférieure.

Theoreme 3. *Si on suppose que* $9N; 8V; 8V \ 2 \ V; Rk(V) < N$, *alors* $securityGap[View; acl_u; F; Q_{View}]$ *est NP-Complet selon la taille des entrées*

Sketch of proof. Le problème est NP de façon évidente d'après la preuve précédente (avec $k \quad N$). Pour montrer la complétude, on se ramène au problème de 3-coloriage d'un graphe. Soit $G = (V; E)$ un graphe quelconque.

On construit le réseau dans lequel les utilisateurs sont les couleurs, ainsi les relations de la base de données sont $Users(c)$, $Friends(c; c')$ conditionné par $c \neq c'$ et $8n \ 2 \ V$ on créé une table pour ce noeud : $n(id; c) \ 2 \ B$. Les vues modifiables sont donc $8n; n[c](id)$ et $Friends[c](c')$.

On construit maintenant les règles de vues statiques du réseau :

- $8e = (n; n') \ 2 \ E; e[c](id_n; id'_n) \ ` \ n[c](id_n); n'[c](id'_n)$: la vue contient un tuple si et seulement si elle est mal coloriée.
- $MultiColored[c](c') \ ` \ e[c'] \ _F(id_1; id_2)$: cette vue est non vide si et seulement si le graphe est mal colorié
- $Colored[c](True) \ ` \ (8n \ 2 \ V; n[c_n](id_n))$: cette vue contient $True$ si toutes les arêtes sont coloriées.
- $8n \ 2 \ V; ValidColor_n[c](c') \ ` \ n[c'](id_n); c' \ 2 \ f1; 2; 3g$: 3 règles par noeud permettent de déterminer si le noeud possède une couleur entre 1 et 3.
- $PossibleColors[c](c') \ ` \ ValidColor_n[c](c')$: n règles de ce type qui permettent de répertorier l'ensemble des couleurs du graphe.
- $ValidColor[c](True) \ ` \ (8n \ 2 \ V; ValidColor_n[c](c_n)$: indique que chaque noeud possède une couleur entre 1 et 3
- $ThreeColors[c](True) \ ` \ PossibleColors[c](c_1); PossibleColors[c](c_2); PossibleColors[c](c_3);$ $c_1 \neq c_2; c_2 \neq c_3; c_3 \neq c_1$: Indique qu'il existe au moins une apparition de chaque couleur.

Soit un utilisateur c'. Son contrôle d'accès sur la vue $Friends$ suit la règle

$$acl_{c'}(Friends; c_1) \ ` \ Friends[c'](c_1); ValidColor[c_1](True); Colored[c_1](True); ThreeColors[c_1](True)$$

Cette table est non vide si et seulement si le graphe est colorié entre 1 et 3 (en supposant que c_1 ait un ami au moins). Sa politique de contrôle d'accès est quant à elle

$$Q_{Friends}(c'; c_1) \ `Friends[c'](c_1); ValidColor[c_1](True); Colored[c_1](True); ThreeColors[c_1](True); \\ Multicolored[c_1](c_2)$$

Cette table est non vide si et seulement si le graphe est colorié entre 1 et 3 ET mal colorié.

Ainsi, on a un *trou de sécurité* s'il existe une instance telle que chaque couleur est valide, chaque noeud est coloré par une couleur entre 1 et 3 mais il n'existe pas de couleur apparaissant sur les deux noeuds reliant une arête. Plus formellement :

$$\begin{aligned} securityGap[Friends; acl_c; F; Q_{Friends}] \ , \quad & 9I; 9c_1 \ 2 \ Colors(I); Friends[c](c_1) \\ & \wedge ValidColor[c_1](True) \\ & \wedge Colored[c_1](True) \wedge ThreeColors[c_1](True) \\ & \wedge (8c_2 \ 2 \ Colors(I); : MultiColored[c_1](c_2)) \\ , \quad & 9c \ 2 \ f1; 2; 3g^V 8(n; n') \ 2 \ E; c(n) \neq c(n') \\ , \quad & 3 \quad Colored(G) \end{aligned}$$

□

Ici encore, on peut prouver la NP-Completude pour les autres problèmes en utilisant le 3-coloriage de graphe.

Les preuves de complexité dans le cas d'une CQ pour Q_{View} sont quasiment identiques au cas du Théorème 2 puisqu'on utilise le même papier qui permet de relier ce problème à celui de query-containment.

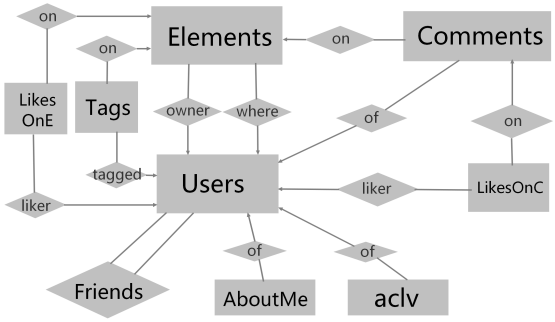

FIGURE 4 – Les relations de la base de données centralisée du réseau USER-FACEBOOK

2.3 Modélisation de Facebook

Maintenant que le modèle est posé, nous avons pu modéliser le réseau social FACEBOOK en SocialNetworkLog. Ce réseau social a été choisi, car c'est le plus connu et le plus complexe en matière de contrôle d'accès. Par gain de place, la modélisation qui va suivre ne concerne qu'une sous-partie de FACEBOOK, que l'on appellera USER-FACEBOOK.

Representations des relations et vues modi ables La figure 4 ci-dessus représente le modèle entité-relation de la base de données centralisé de USER-FACEBOOK. On définit dans un deuxième temps les vues modifiables et paramétrées par les utilisateurs. Les figures 5 et 6 résument les règles de création de ces vues, qui sont en général de simples projections/sélections d'une table de la base de données centralisée.

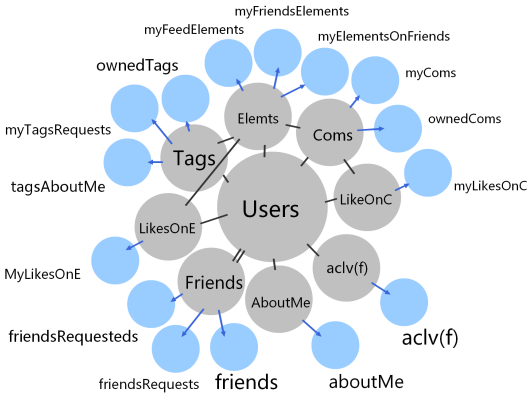

FIGURE 5 – Les vues modifiables de la base de données centralisée du réseau USER-FACEBOOK

De nition des sequences d'actions de Facebook On peut alors définir des séquences d'actions sur ces vues modifiables. Parmi ces séquences d'actions, on distingue notamment **les actions triviales des actions non triviales**. On note par exemple les ajouts triviaux :

$$+_{u'}\texttt{friendsRequests[u](u')} \quad `$$

On peut de même parler de *suppression triviale* et de *mise à jour triviale*. On a également des règles quasi-triviale où il faut voir l'élément sur lequel on veut ajouter des informations (par exemple, il faut voir une photo pour pouvoir la commenter).

$$+\texttt{myTagsRequests[u](id,elem}_{id}\texttt{,...)} ` \texttt{MyElements[u](elem}_{id}\texttt{,...)}$$

Enfin, il y a des règles non triviales, comme la validation d'une demande d'ami ou de tag :

$$-\texttt{friendsRequests[u](f)},\ +\texttt{friends[u](f)} ` \texttt{friendsRequests[u](f)}$$

Description	Finesse	Regle
Demandes d'amis que j'ai reçues	CG	friendsRequests[u](f) ` Friends(f,u,False)
Liste d'amis	CG	friends[u](f) ` Friends(f,u,True) friends[u](f) ` Friends(u,f,True)
Mes publications sur mon mur	FG	myFeedElements[u](id,$\overline{X_0}$) ` Elements(id,u,u,X)
Mes publications sur le mur de mes amis	CG	myElementsOnFriends[u](id,$\overline{X_0}$) ` Elements(id,u,u_2,X), u_2 $\neq$ u
Tags de moi sur des publications	CG	TagsAboutMe[u](id,$\overline{X_0}$) ` Tags(id,elem_{id},u,u_2,True,X)
Demandes de tags sur mes publications	CG	myTagsRequests[u](id,$\overline{X_0}$) ` Tags(id,elem_{id},u_2,u,False,X)

FIGURE 6 – Quelques exemples de règles de vues modifiables

Le tableau suivant résume les actions possibles : (Quasi) Trivial Add, Trivial Delete, Trivial Update, Non-Trivial Sequence.

table	TA	TD	TU	NTS
aboutMe	✓	✓	✓	
friendsRequests	✓	✓		✓
friendsRequested		✓		
friends		✓		✓
myFeedElements	✓	✓	✓	
myElementsOnFriends	✓	✓	✓	
myFriendsElements	✓	✓	✓	
tagsAboutMe		✓		
myTagsRequests	(Q) ✓	✓		✓
ownedTags	(Q) ✓	✓		✓
myComs		✓	✓	
ownedComs	(Q) ✓	✓		
myLikesOnElements	(Q) ✓	✓		
myLikesOnComments	(Q) ✓	✓		

FIGURE 7 – Actions possibles dans USER-FACEBOOK

Regles de contrôles d'acces On va maintenant définir les règles de contrôles d'accès possibles sur ces vues. On fait tout d'abord la liste des règles de contrôle d'accès possibles :

Nobody	R[u](u') ` ()
Only me	R[u](u') ` Users(u)
Friends	R[u](u') ` Friends[u](u')
Friends of friends	R[u](u') ` Friends[u](f), Friends[f](u')
Everyone	R[u](u') ` Users(u')
Group of friends (G)	R[u](u') ` Friends[u](u'), Include(u',G)
Friends students of (School)	R[u](u') ` Friends[u](u'), Schools[u'](School)
Friends living in (City)	R[u](u') ` Friends[u](u'), Cities[u'](City)

On peut alors pour chaque vue modifiable et chaque privilège (READ, WRITE, DELETE) associer un ensemble de ces règles.

Exemples de vues statiques Maintenant que les fondations de la modélisation sont posées, on peut écrire des règles de vues statiques dans le formalisme SOCIALNETWORKLOG.

— Une vue sur la liste d'amis qui prend en compte les amis communs (SESSION(user) correspond à l'utilisateur qui exécute la vue) :

```
commonFriends[u,u'](f) ` Friends[u](f), Friends[f](u')
                       ` Friends[u'](f), Friends[f](u)
                       ` Friends[u](f), Friends[u'](f)
AccessiblesFriends[u](f) ` Friends[u](f)
                         ` commonFriends[u,SESSION(user)](f)
```

— Une vue sur les publications du mur FACEBOOK

$$\begin{aligned}
\texttt{publications[u]}(id_{elem},\ldots) \;`\; & \texttt{MyElements[u]}(id_{elem},\ldots)\\
`\; & \texttt{MyFriendsElements[u]}(id_{elem},id_{friend},\ldots)\\
\texttt{AccessiblesPublications[u]}(id_{elem},\ldots) \;`\; & \texttt{publications[u]}(id_{elem},\ldots)\\
`\; & \texttt{publications[u]}_{F}(id_{elem},\ldots),\\
& \texttt{tagsAboutMe[u']}(id_{tag},id_{elem})
\end{aligned}$$

— Une vue sur les éléments où un utilisateur u est identifié

$$\begin{aligned}
\texttt{publicationsAbout[u]}(id_{elem},\ldots) \;`\; & \texttt{AccessiblesPublications[u']}(id_{elem},\ldots);\\
& \texttt{tagsAboutMe[u]}_{F}(id_{tag},id_{elem})
\end{aligned}$$

Ces exemples permettent notamment de voir que l'ensemble des éléments qui sont réellement visibles est plus complexes à définir que ce que laissent entendre nos propres contrôles d'accès.

3 Explorer et exploiter les contrôles d'accès

Cette deuxième partie s'intéresse aux failles dans les contrôles d'accès que l'on a détecté dans FACEBOOK, et qui permettent de faire ressortir beaucoup d'information et de mener des attaques de désanonymisation.

3.1 État de l'art des attaques Facebook

Failles de l'API publicitaire De nombreux chercheurs ont déjà trouvés des failles dans les différentes APIs de FACEBOOK, notamment les API publicitaires et développeurs. Des failles dans l'onglet PII de *l'API publicitaire* permettaient de trouver des informations très précises (comme le numéro de téléphone) sur n'importe quel utilisateur [9, 8]. Nos recherches ont permis de voir que ces failles avaient été résolues et que l'API publicitaire était plutôt bien sécurisée.

Failles de l'API developpeurs De même, le récent scandale *Cambridge Analityca* portait notamment sur les informations que l'on concédait aux applications via *l'API développeurs*. Là encore, cette API semble très sécurisée, et nous en avons conclu que le seul moyen d'avoir des informations sur quelqu'un était de lui demander la permission pour obtenir un *token d'accès* et que FACEBOOK ait accepté la diffusion de l'application. De plus, le nombre d'informations disponibles via ce moyen a été réduit.

Failles de l'API utilisateurs Nous avons donc décidé de me pencher plutôt du côté de **l'API utilisateurs**, qui est très peu étudiée. Pourtant, tout le monde y a accès, sans beaucoup d'efforts. Les recherches que nous avons menées sur cette API utilisateur nous ont conduit à un outil controversé de FACEBOOK : le *Facebook Graph Search*, disponible uniquement aux États-Unis et qui permet aux utilisateurs de faire des recherches du type "*My Friends who liked École Normale Supérieure*". Cependant, cet outil est bien plus puissant que ce qu'il semble être en surface. Et cela pour deux raisons : il existe un nombre très important de mots clés, et il est possible de faire des recherches complexes (Intersection / Union).

Nous avons découvert qu'un grand nombre d'outils indépendants exploitaient déjà Graph Search pour "Stalker" des utilisateurs [2], mais aucun d'eux n'utilise plus de la moitié des mots-clés différents, et aucun

d'eux ne permet de faire des recherches complexes (notamment, l'union n'est présente sur aucun de ces outils). Nous avons donc créé un outil qui prend en entrée des requêtes en langage compréhensible et renvoie un lien *Facebook Graph Search*.

3.2 Explorer le graphe Facebook avec FQL2

Grammaire du langage La grammaire FQL2 est assez simple : un document est une succession de requêtes. Une requête est soit une union/intersection de requête, soit une jointure entre les résultats d'une requête (ou une requête atomique non exécutable) et une table, soit une requête atomique :

$$Q = Q \setminus QjQ \ [\ QjRelation(Q)jRelation(N)jA$$

Chacune de ces possibilités peut être réécrite dans le formalisme SocialNetworkLog. Par exemple, l'union de deux requêtes $Q_1 \setminus Q_2$ peut s'écrire `Q'(x) ` Q1(x);Q2(x)`.

Les quatre première possibilités sont des relations binaire et la dernière est une relation unaire.

Les di erents types de retour Toutes les données étant typées, une requête renvoie des données d'un type précis qui correspond à une table de la base de données Facebook. Une jointure pour une relation donnée nécessite que la requête sur laquelle on fait la jointure ait le bon type. Les 10 tables/types de ce langage sont les suivantes :

```
Users   Groups    Languages     Apps     Stories
Pages    Events      Places    Photos     Vidéos
```

Mots-cles du langage Il existe un nombre important de mots-clés de relations qui permettent de faire des jointures (*Relation*(*Q=N*)). Voici quelques exemples de ce que l'on peut chercher :

```
residents       current-cities    pages-liked       likers        classmates
 friends            spouses       users-born-in     visitors     places-visited
languages-spoken     speakers         students        schools        employers
   employees          groups          members      groups-admin    events-joined
events-created    events-invited     apps-used       events-at       photos-at
```

On peut également chercher, étant donné une personne ou un groupe de personnes, les photos/vidéos/publications postées, où les utilisateurs du groupe sont identifiés, likées, commentées ou encore recommandées pour ce groupe.

Les requêtes atomiques se divisent en deux groupes : celles qui sont directement exécutables (A) et celles qui ne le sont pas (N). Les requêtes atomiques non exécutables correspondent à des entités uniques : `me` et `id(123456789)` avec l'id qui correspond à l'élément (utilisateur, page, groupe, etc.). Voici quelques exemples de requêtes atomiques exécutables :

```
          (fe)males                people-interested-in-(fe)males
         all [type]                    [type] category (cat)
[type] named "elem_name"                  workers (cat)
  religious-view(cat)                  political-view(cat)
```

On peut également chercher des évènements, photos, vidéos ou publications selon leur date (avant, après, pendant une date ou entre deux dates). Un exemple de requête est présenté figure 10 et la documentation complète du langage est en référence [1].

Voici donc un langage qui nous permet de faire des recherches complexes au travers du graphe de Facebook. On note tout de même que les résultats de ces recherches respectent toujours les contrôles d'accès choisis par les utilisateurs. Dans la prochaine partie, je vais présenter un exemple d'attaque utilisant ce langage qui permet de désanonymiser des utilisateurs.

3.3 Désanonymisation des utilisateurs

Tinder est une application de rencontre lancée en 2012 et qui cumule plus de 100 millions de téléchargements à ce jour. Sur cette application, les utilisateurs sélectionnent les profils qui les intéressent parmi les personnes alentour. Sur ces profils, il n'est à priori pas simple de retrouver l'identité des utilisateurs. Cependant, le moyen le plus utilisé pour se créer un compte Tinder est d'utiliser son compte Facebook. Nous allons voir que grâce aux informations que dévoile notre profil aux autres utilisateurs, il devient très facile de retrouver notre compte Facebook, lorsque celui-ci est mal sécurisé.

3.3.1 Dis moi ce que tu aimes, je te dirai qui tu es

Elements du pro l Tinder Parmi les informations qui apparaissent sur un profil *Tinder* on trouve notamment : le prénom (le même que celui sur FACEBOOK), l'âge, la localisation, une bio, des photos, l'emploi, les études, et surtout des "intérêts". Pour trouver les intérêts d'un utilisateur, *Tinder* récupère les informations des 100 dernières pages FACEBOOK "likées" par celui-ci et tout utilisateur qui verra son profil aura accès aux "intérêts communs". Cela correspond à l'intersection entre ses 100 dernières pages likées et les 100 dernières de l'utilisateur dont il voit le profil.

Il est également possible de lier son compte *Tinder* à son compte *Instagram*, mais dans ce cas, l'anonymat n'existe déjà plus.

Attaque par interêts communs Nous allons donc nous intéresser ici à ces *Intérêts communs*. Notamment, nous voudrions savoir à partir de combien d'intérêts sur une personne il est possible de la retrouver, connaissant son prénom ?

Si on suppose que les intérêts sont indépendants entre eux, alors les probabilités nous disent qu'en prenant des pages françaises, 10 intérêts sont suffisants pour retrouver l'identité d'un utilisateur. En réalité, les intérêts ne sont pas indépendants : quelqu'un qui aime la page d'un groupe de rap sera plus susceptible d'aimer un chanteur de rap similaire qu'une émission de téléréalité (Figure 8)

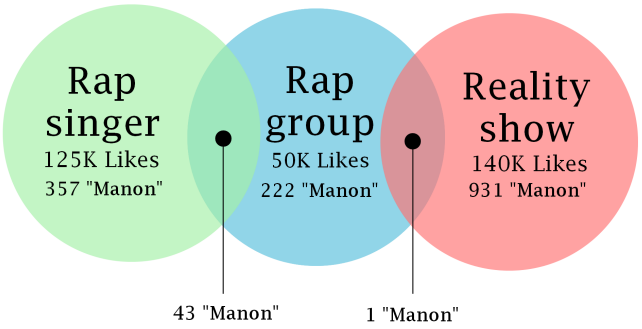

FIGURE 8 – Intersections d'audiences

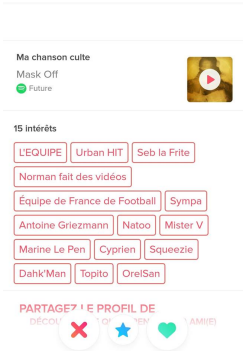

FIGURE 9 – Exemple d'intêrets communs

Cependant, si les intérêts sont assez dissociés ou que le prénom est peu banal, il est aisé de retrouver le profil FACEBOOK de l'utilisateur avec l'outil présenté dans la partie précédente. En, effet, on sait que l'application présente les *intérêts communs*. Nous avons donc créé un faux profil Tinder lié à un profil FACEBOOK avec 100 intérêts les plus populaires en France en ce moment, de sorte à maximiser les chances d'avoir des intérêts communs.

Utilisations de FQL2 Lorsque l'on obtient des intérêts communs, nous utilisons l'outil FQL2 pour retrouver l'identité de l'utilisateur. Par exemple pour la figure 9, si l'individu s'appelle *Louis*, on effectue la recherche de la figure 10.

```
SEARCH users named "Louis" INTER men INTER ?lequipe INTER ?urbanhit INTER ?seblafrite
                           INTER ?norman INTER ?equipedefrance INTER ?sympa
                           INTER ?griezmann INTER ?natoo INTER ?misterv
                           INTER ?lepen INTER ?cyprien INTER ?squeezie
                           INTER ?dahkman INTER ?topito INTER ?orelsan;
```

FIGURE 10 – Exemple de requête d'attaque

Chaque élément de la forme `?nom_de_la_page` correspond à une vue de l'audience d'une page. Par exemple `VIEW ?lequipe = id(76283417065)->likers;` où 76283417065 correspond à l'identifiant de la page *L'équipe*. Il est également possible d'utiliser le lieu de travail ou d'étude pour réduire encore le nombre de résultats. Enfin, on retrouve l'utilisateur à l'aide de ses photos Tinder.

Avec un tel nombre d'intérêts, il y a de fortes chances que l'on retrouve l'identité d'un utilisateur, à moins que celui-ci ait sécurisé les contrôles d'accès à ses intérêts. Cependant, mes expériences (et celles d'autres chercheurs avant moi) ont montré que peu de gens sécurisent réellement ces informations. Notamment, nous avons trouvé qu'entre 70 et 80 % de mes amis FACEBOOK ont laissé les contrôles d'accès par défaut "Public" sur les intérêts, études et emplois.

3.3.2 Resultats des experiences

Resultats des experiences Nous avons donc mené cette expérience, tout d'abord avec un profil d'homme qui cherche des femmes, puis dans le cas inverse. Dans la première expérience, sur 300 profils, nous obtenions un intérêt commun avec 39% d'entre eux. Et parmi ces derniers, nous avons pu retrouver l'utilisatrice dans 32% pour des cas, soit au total 12% d'utilisatrices retrouvées.

Dans la seconde expérience, nous avons localisé le faux profil à Lille (lieu de l'expérience), et les personnes ciblées apparaissaient plus souvent en haut de la liste d'utilisateurs. Nous avons donc eu de meilleurs résultats : 48% des profils avaient au moins un intérêt commun et parmi ceux-ci, 60% m'ont permis de relier le profil Tinder à un compte FACEBOOK, soit au total 29% des profils affichés.

Common Interests	Users	Found	Too many results	Not ident -ifiable	Doubt
0	156				
1	19	10	8	0	1
2	29	15	11	2	1
3	33	16	9	8	1
4	18	12	3	2	0
5	15	8	1	4	2
6	12	9	1	2	0
7	2	2	0	0	0
8	4	4	0	0	0
9	3	3	0	0	0
10	2	2	0	0	0
11	4	3	0	0	1
12	1	1	0	0	0
13	0	0	0	0	0
14	1	1	0	0	0
15	1	1	0	0	0
total	300	87	33	18	6

FIGURE 11 – Statistiques sur les 300 profils de la deuxième expérience

Analyse des resultats Pour expliquer que certains profils avec des intérêts communs n'ont pas pu être retrouvés, on a notamment le fait qu'il y avait parfois trop de résultats pour retrouver l'utilisateur rapidement. De plus, nous avons vu que dans plus de 20% des cas, les utilisateurs avaient sécurisé leurs informations.

Pour expliquer que la plupart des profils n'avaient pas d'intérêt commun avec mon faux profil, nous avons trouvé deux raisons. Premièrement, il existe un nombre non négligeable de compte *Tinder* non-reliés à un profil FACEBOOK et de faux profils sans intérêt. Deuxièmement, 100 intérêts ne sont pas suffisants pour couvrir la totalité des utilisateurs.

Extension de l'attaque Pour que ce type d'attaque soit réellement puissant, il serait possible de créer un nombre plus conséquent de faux profils avec pour chacun des intérêts différents et pour chaque profil *Tinder*, mémoriser les intérêts observés jusqu'alors. Une fois qu'on a un nombre satisfaisant d'intérêts sur une personne, il suffit de faire une recherche avec notre outil et il y a de fortes chances d'obtenir un seul profil FACEBOOK (le nombre de faux positifs avec 15 intérêts est négligeable).

Securites des applications Les sécurités mises en place par FACEBOOK et Tinder sont facilement surmontables. Notamment, il est parfois difficile de créer un compte FACEBOOK. Pour créer un compte Tinder, il suffit d'un numéro de téléphone, sauf qu'il est possible d'utiliser plusieurs fois le même numéro.

Une autre sécurité de Facebook est de limiter le nombre de recherches par jour. Pour contourner cela, on fait simplement l'union de toutes les recherches d'utilisateurs. Il risque d'y avoir de faux positifs, mais leur nombre reste négligeable.

Au final, le meilleur moyen pour rester anonyme est de sécuriser ses contrôles d'accès et de ne pas lier son compte Facebook à tous ses réseaux.

4 Conclusion et perspectives de recherches

Au travers de ce stage, nous avons donc construit un formalisme des contrôles d'accès sur les réseaux sociaux ainsi que les premières démonstrations sur les complexités des problèmes de fuite de données. Cela permettra par la suite à d'autres problématiques autour des contrôles d'accès sur les réseaux sociaux de se développer, notamment la déduction de données, que l'on a nous-même un peu abordé.

En parallèle, nous avons mis en avant un langage de requête dans Facebook, que nous continuons de compléter lorsque nous trouvons de nouvelles possibilités. Il est possible de trouver un grand nombre d'applications à ce langage, comme celle présentée ci-dessus. Malgré les possibilités offertes par ce langage, les retours des requêtes n'enfreignent jamais les contrôles d'accès des utilisateurs. Il est donc important de *sensibiliser* les utilisateurs quant à la gestion des contrôles d'accès et de leurs liens inter-réseaux sociaux, comme l'a montré notre expérience sur *Tinder*.

D'autres questions restent ouvertes sur cet outil de recherche, notamment l'ordre d'apparition des utilisateurs dans la liste des résultats : *est-ce que cet ordre dépend de critères sur les utilisateurs que ceux-ci ont pourtant cachés ?*. Plus généralement, peut-on déduire des données cachées par l'utilisateur grâce à l'outil de recherche ?

En conclusion, ce stage a été pour moi une expérience enrichissante qui m'a permis de découvrir le milieu de la recherche. Je voudrais particulièrement remercier mes maîtres de stage pour leur écoute et leur disponibilité : *Pierre Bourhis, Romain Rouvoy et Walter Rudametkin.*

Références

[1] Documentation fql2 : https://theo.delemazure.fr/more/Doc.pdf.

[2] Facebook stalk scan tool : https://stalkscan.com/.

[3] Plainte de la quadrature du net contre facebook : https://gafam.laquadrature.net/wp-content/uploads/sites/9/2018/05/facebook.pdf.

[4] Abiteboul, S., Bourhis, P., and Vianu, V. A formal study of collaborative access control in distributed datalog. In *ICDT 2016-19th International Conference on Database Theory* (2016).

[5] Abiteboul, S., and Vianu, V. Collaborative data-driven workflows : think global, act local. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems* (2013), ACM, pp. 91–102.

[6] Benedikt, M., Bourhis, P., Ten Cate, B., and Puppis, G. Querying visible and invisible information. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (2016), ACM, pp. 297–306.

[7] Benedikt, M., and Gottlob, G. The impact of virtual views on containment. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 297–308.

[8] Krishnamurthy, B., and Wills, C. E. On the leakage of personally identifiable information via online social networks. In *Proceedings of the 2nd ACM workshop on Online social networks* (2009), ACM, pp. 7–12.

[9] Venkatadri, G., Andreou, A., Liu, Y., Mislove, A., Gummadi, K. P., Loiseau, P., and Goga, O. Privacy risks with facebook's pii-based targeting : Auditing a data broker's advertising interface. In *IEEE Symposium on Security and Privacy (SP)* (2018), pp. 221–239.

[10] Zaychik Moffitt, V., Stoyanovich, J., Abiteboul, S., and Miklau, G. Collaborative access control in webdamlog. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 197–211.