Points de coupure pour les préoccupations de renforcement de sécurité utilisant le flot de contrôle

Marc-André Laverdière — Azzam Mourad — Andrei Soeanu — Mourad Debbabi

Laboratoire de Sécurité Informatique,
Institut d'Ingénierie des Systèmes d'Information de l'Université Concordia,
Université Concordia, Montréal, Québec, Canada
{ma_laver, mourad, a_soeanu, debbabi}@ciise.concordia.ca

RÉSUMÉ. *Nous présentons dans cet article deux nouveaux points de coupure pour les langages orientés aspects (AOP) qui sont nécessaires au renforcement systématique de sécurité. Ils permettent l'identification de certains points de jointure dans un graphe de flot de contrôle. La première primitive est le Closest Common Ancestor (CCA) (c.a.d. l'ancêtre commun le plus près), qui retourne l'ancêtre le plus près étant sur le chemin de tous les points d'intérêt. La seconde primitive proposée est le First Common Descendant (FCD) (c.a.d le premier descendant commun), qui détermine le point de jointure descendant le plus près pouvant être traversé par tous les chemins émanant des points de coupure d'intérêt. Nous croyons que ces points de coupures sont nécessaires pour plusieurs pratiques de renforcement de sécurité et, au meilleur de nos connaissances, aucune combinaison de points de coupure peut offrir leur fonctionnalité. De plus, nous démontrons la viabilité et la justesse de notre proposition en élaborant et implémentant leur algorithmes et montrant les résultants d'une étude de cas.*

ABSTRACT. *In this paper, we present two new control flow based pointcuts to Aspect-Oriented Programming (AOP) languages that are needed for systematic hardening of security concerns. They allow to identify particular join points in a program's control flow graph (CFG). The first proposed primitive is the Closest Common Ancestor (CCA), which returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The second proposed primitive is the First Common Descendant (FCD), which returns the closest child join point that can be reached by all paths starting from the pointcuts of interest. We find these pointcuts to be necessary because they are needed to perform many security hardening practices and, to the best of our knowledge, none of the existing or combination of pointcuts can provide their functionalities. Moreover, we show the viability and correctness of our proposed pointcuts by elaborating and implementing their algorithms and presenting the result of a testing case study.*

MOTS-CLÉS : *Sécurité, Programmation Orientée Aspect, Flot de Contrôle, Point de Coupure*

KEYWORDS: *Security Hardening, Aspect-Oriented Programming, Control Flow, Pointcut*

Remerciements

Cette recherche est le résultat d'une collaboration fructueuse entre le Laboratoire de Sécurité Informatique de l'Université Concordia, DRDC-Valcartier Défense nationale Canada et Bell Canada, grâce à une subvention dans le cadre du Programme de partenariat de recherche MDN/CRSNG.

1. Motivations & Contexte

Dans le monde informatique actuel, la sécurité gagne un rôle prédominant. L'industrie fait face à des problèmes de confiance publique à la découverte de vulnérabilités, et les consommateurs s'attendent à ce que la sécurité soit livrée d'emblée, même pour des programmes n'ayant pas été conçus à cette fin. Les systèmes d'un certain âge offrent un défi supplémentaire quand ils doivent être adaptés aux environments réseau et web, malgré qu'ils ne furent pas conçus pour évoluer dans un cadre avec un tel niveau de risque. Des outils et des directives pour la sécurité sont disponibles pour les développeurs depuis quelques années, mais leur adoption est limitée jusqu'à présent. Les mainteneurs doivent maintenant améliorer la sécurité des programmes et sont souvent sous-équipés pour le faire. Dans certains cas, peu de recours sont disponibles, particulièrement pour les logiciels propriétaires (COTS) qui ne sont plus supportés, ou quand le code source est perdu. Par contre, quand le code source est disponible, par exemple dans le cas des logiciels libres, une plus grande gamme d'améliorations de sécurité sont possibles.

Peu de concepts et approches dans la littérature ont émergé afin d'aider les développeurs à améliorer la sécurité des logicielles. À cet égard, l'AOP semble être un paradigme prometteur pour le renforcement de sécurité des logicielles, un domaine n'ayant pas été touché adéquatement par les paradigmes antérieurs, comme la programmation orientée objet. L'idée centrale de l'AOP veut que des systèmes informatiques puissent être programmés plus adéquatement en spécifiant séparément les différentes préoccupations. Une infrastructure technologique permet d'intégrer ces préoccupations en un tout cohérent. Les techniques de ce paradigme furent introduites précisément afin de résoudre les problèmes de développement inhérents aux préoccupations entrecroisantes, en faisant donc une solution intéressante pour les problèmes de sécurité [BOD 04, DEW 04, HUA 04, Cig 03, SLO 03].

Toutefois, l'AOP ne fut pas conçu à la base pour s'occuper de problèmes de sécurité, résultant en des manquements dans les technologies actuelles [MAS 03, HAR 05, MYE 99, BON 05, KIC 03]. Nous étions incapables de mettre en pratique certaines améliorations à cause de ces fonctionnalités manquantes. De telles limitations nous ont forcés, dans leur implémentation à faire des acrobaties de programmation, résultant en l'intégration de modules supplémentaires devant être intégrés à l'application, impliquant un coût de performance, de mémoire et de développement. De plus, le code généré par cette stratégie est de complexité supérieure en ce qui a trait à l'audit et l'évaluation.

Conséquemment, nous proposons des points de coupure nécessaires afin d'identifier des points d'intérêts dans le graphe de flot de contrôle (CFG) d'un programme. Les primitives proposées sont le Closest Common Ancestor (CCA) (c.a.d. l'ancêtre commun le plus près) et le First Common Descendant (FCD) (c.a.d le premier descendant commun). Le CCA retourne le point de jointure le plus près étant également ancêtre de tous les points d'intérêt et étant sur tous les chemins d'exécution menant à ces derniers. Le FCD retourne le point de jointure le plus près étant également descendant de tous les points d'intérêts et étant sur tous les chemins d'exécution provenant de ces derniers. Ces points de coupure sont nécessaires à l'exécution de plusieurs améliorations de sécurité et aucun des points de coupures peuvent offrir leurs fonctionnalités. Le constat n'est pas le même en combinant les points de coupure disponibles dans la pratique et la littérature, car nous étions incapables de trouver une méthode qui isolerait un seul noeud dans notre CFG qui satisfaisait les critères définis pour CCA et FCD

Nous avons organisé cet article de la manière suivante : nous spécifions premièrement les points de coupures dans la section 2 avant de discuter de l'utilité de notre proposition et de ses avantages dans la section 3. Ensuite, dans la section 4, nous présentons les algorithmes nécessaires pour l'implémentation des points de coupure présentés, accompagnés d'une méthode d'étiquettage hiérarchique de graphe. Cette section contient également les résultats de notre étude de cas. Nous faisons une transition vers l'état de l'art dans la section 5 et concluons dans la section 6.

2. Définition des points de coupure

Nous incluons ici une syntaxe qui définit un point de coupure p qui inclut notre proposition :

$$p \ ::= \texttt{call}(s) \mid \texttt{execution}(s) \mid \texttt{CCA}(p) \mid \texttt{FCD}(p) \mid p \,||\, p \mid p \,\&\&\, p$$

où s est une signature de fonction. Les CCA et FCD sont les nouvelles primitives de point de coupure que nous proposons. Leur paramètres sont également un point de coupure p.

Nous définissons chaque point de coupure dans les suivantes.

2.1. *CCA*

La primitive `CCA` opère sur le CFG étiqueté d'un programme. Son entrée est un ensemble de points de jointure définis en tant que point de coupure et sa sortie est un seul point de jointure. En d'autres mots, si nous considérons des notations de CFG, son entrée est un ensemble de noeuds et sa sortie est un seul noeud. Cette sortie est l'ancêtre commun qui constitue (1) le noeud parent le plus près de toutes les noeuds de

l'ensemble d'entrée et (2) le noeud au travers duquel passent tous les chemins qui les rejoint. Dans le pire des cas, le CCA sera le point de départ (e.g. `main`) du programme.

2.2. *FCD*

La primitive FCD opère sur le CFG étiqueté du programme. Son entrée est un ensemble de points de jointure définis en tant que point de coupure et sa sortie est un seul point de jointure. En d'autres mots, si nous considérons des notations de CFG, son entrée est un ensemble de noeuds et sa sortie est un seul noeud. Cette sortie est (1) un descendant commun à tous les noeuds sélectionnés et (2) est le premier noeud commun sur tous les chemins émanant des noeuds sélectionnés. Dans le pire des cas, le FCD sera le point de terminaison du programme.

3. Discussion

Dans cette section, nous discuttons des avantages, inconvénients et limitations de notre proposition. La détermination du CCA et FCD est possible uniquement dans la mesure où laquelle il est possible d'obtenir un graphe de flot de contrôle qui représente parfaitement le programme analysé. Bien que cela soit possible dans la majorité des cas, certains programmes (p. ex. utilisant la réflection en Java) ne peuvent être analysés pour déterminer leur CFG complet.

Dans les suivantes, nous examinons les avantages généraux de nos points de coupure, ainsi que leurs avantages spécifiques au renforcement de sécurité.

3.1. *Avantages généraux du CCA and FCD*

Il est clair que les primitives que nous proposons supportent le principe de séparation des préoccupations en permettant l'implémentation de modifications au programme sur un ensemble de points de jointure selon la préoccupation (comme démontré précédemment). Nous allons plus loin en montrant des avantages plus généraux rattachés à noter proposition :

– *Facilité d'utilisation* : les programmeurs peuvent choisir des endroits dans le graphe de flot de contrôle d'une application en évitant de déterminer manuellement le point précis où le faire.

– *Facilité de maintenance* : les programmeurs peuvent changer la structure de logiciels sans se préoccuper des aspects associés. Sans cca et fcd, il aurait fallu spécifier les endroits d'injection, et ces derniers ne seraient plus nécessairement appropriés après les changements structuraux. Par exemple, si nous changeons, à l'aide d'un *refactoring*, le chemin d'exécution à une certaine fonction, nous devons changer le nouvel ancêtre commun couvrant tous les chemins, alors que nos points de coupure le feront automatiquement.

– *Optimisation* : les pré-opérations et post-opérations sont injectées une seule fois dans le programme, où elles sont nécessaires. Nous évitons également de tout mettre dans le `main`. Cela évite de surcharger l'initialisation du programme, ce qui diminue le niveau de réponse apparent du programme. Certains opérations coûteuses peuvent être évitées si les branches de code les exigeant ne sont jamais exécutées, épargnant donc des cycles de processeur et de la mémoire. De plus, nous évitons une injection autour de chaque point de jointure d'intérêt, la solution par défaut offerte par les langages AOP. Cela serait remplacé par une seule injection autour du CCA et FCD.

– *Augmenter le niveau d'abstraction* : les programmeurs peuvent développer des librairies d'aspects plus réutilisables et abstraites grâce à ce mécanisme.

3.2. *Utilité de CCA et FCD pour le renforcement sécurité*

Plusieurs pratiques de renforcement de sécurité nécessitent l'injection de code autour d'un ensemble de points de jointure ou chemins d'exécution [SEA 05, HOW 02, WHE 03, BIS]. Des exemples de tels cas sont l'injection d'initialisation et de déinitialisation de librairies de sécurité, changements de niveaux de privilèges, garantie d'atomicité, l'enregistrement, etc. Le modèle actuel en AOP permet seulement d'identifier un ensemble de points de jointure d'un programme et d'injecter le code avant, après, ou autour de chacun d'entre eux. Toutefois, aucun des points de coupure permettent d'identifier un point de jointure commun à un ensemble de points de jointure où nous pouvons injecter le code une fois pour tous. Bien qu'il serait possible, dans certains cas, de faire une telle injection, cela requiert une combinaison complexe de points de coupure et/ou l'intégration de modules additionnels. Une telle solution aura donc un impact au niveau du temps d'exécution, de l'utilisation de la mémoire et augmentera la complexité du code. Il nous faut donc de nouvelles primitives permettant aux programmeurs d'aspects de créer des solutions abstraites, efficaces et adaptables aux changements de programmes. Dans les suivantes, nous présentons l'utilité et la nécessité de nos points de coupure proposés pour le renforcement de sécurité.

3.2.1. *Initialisation et de déinitialisation de librairies de sécurité*

Pour l'initialisation de librairies de sécurité (autorisation, cryptographie, etc.), nos primitives permettent d'initialiser la librairie seulement pour les branches du code pour lesquelles elles sont nécessaires en identifiant le CCA et le FCD. En utilisant les deux primitives de concert, nous évitons aussi de devoir utiliser des variables globales documentant le statut de l'initialisation. Dans l'exemple ci-bas, nous montrons un extrait d'un aspect que nous avons élaboré pour sécuriser les connections d'une application de type client. Avec les points de coupure actuels, nous ciblons le `main` afin d'ajouter le code d'initialisation et de déinitialisation, tel que montré dans le Listing 1. Une autre solution est de charger et décharger avant et après chaque utilisation, mais cette solution risque de créer des problèmes avec les structures de données rattachées à l'API, ces dernières pouvant être partagées entre les fonctions. Dans le cas de larges applications, ou de systèmes embarqués, les deux solutions créent une accumulation

de code injecté pouvant créer une perte importante de ressources du système. Dans le Listing 2, nous voyons un aspect amélioré offrant des résultats plus efficaces et applicables en utilisant nos points de coupure.

Listing 1 – Aspect sécurisant des connections avec GnuTLS

```
advice execution ("% main (...) ") : around () {
  // Initialization of the API
  hardening_socketInfoStorageInit ();
  hardening_initGnuTLSSubsystem( NONE );
  tjp -> proceed ();
  hardening_deinitGnuTLSSubsystem ();
  hardening_socketInfoStorageDeinit ();
  *tjp -> result () = 0;
}
```

Listing 2 – Aspect sécurisant des connections avec GnuTLS et nos points de coupure

```
advice cca(call("% connect(...)")): before(){
  //Initialization of the API
  hardening_socketInfoStorageInit();
  hardening_initGnuTLSSubsystem(NONE);
  }

advice fcd(call("% connect(...)"), call("% send(...)"), call("% recv(...)"
    ), call("% close(...)")): after(){
  //deinit api
  hardening_deinitGnuTLSSubsystem();
  hardening_socketInfoStorageDeinit();
}
```

3.2.2. *Principe du moindre privilège*

Pour les processus implémentant le principe du moindre privilège, il est nécessaire d'augmenter et de diminuer les droits avant et après une opération à risque. Dans certains cas, il faut également se départir de certains droits pour certaines opérations. Nos primitives peuvent être utilisées pour opérer sur un groupe d'opérations nécessitant le même privilège en injectant du code adjustant les droits d'accès aux points CCA et FCD. Notre approche est applicable dans le cas où il n'y a pas d'opérations non privilégiées dans le chemin d'exécution entre les points de changement de privilège. L'exemple dans le Listing 3 (fait à partir d'une combinaison d'exemples provenant de [HOW 02]) montre un aspect hypothétique implémentant une restriction de privilège autour de certaines opérations. Il utilise les jetons de restrictions et l'api SAFER de Windows XP. Cette solution injecte le code avant et après chacune des opérations, impliquant une perte de performance, particulièrement dans le cas où les opérations a,b et c seraient exécutées consécutivement. Cela serait évité en utilisant le cca et le fcd, ce que nous montrons dans le Listing 4.

3.2.3. *Atomicité*

Dans le cas où une section critique traverserait plusieurs éléments du programme (comme les appels de fonctions), il faut mettre en place des mécanismes d'exclusion

Listing 3 – Aspect hypothétique pour le moindre privilège

```
pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");

advice abc: around(){
  SAFER_LEVEL_HANDLE hAuthzLevel;
  // Create a normal user level.
  if (SaferCreateLevel(SAFER_SCOPEID_USER, SAFER_LEVELID_CONSTRAINED,
                       0, &hAuthzLevel, NULL)){
    //  Generate the restricted token that we will use.
    HANDLE hToken = NULL;
    if (SaferComputeTokenFromLevel(hAuthzLevel, NULL, &hToken,0,NULL)){
      //sets the restrict token for the current thread
      HANDLE hThread = GetCurrentThread();
      if (SetThreadToken(&hThread,hToken)){
        tjp->proceed();
        SetThreadToken(&hThread,NULL); //removes restrict token
      }
      else{//error handling}
    }
    SaferCloseLevel(hAuthzLevel);
  }
}
```

Listing 4 – Aspect amélioré pour le moindre privilège

```
pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");

advice cca(abc): before(){
  SAFER_LEVEL_HANDLE hAuthzLevel;
  // Create a normal user level.
  if (SaferCreateLevel(SAFER_SCOPEID_USER, SAFER_LEVELID_CONSTRAINED,
                       0, &hAuthzLevel, NULL)){
    //  Generate the restricted token that we will use.
    HANDLE hToken = NULL;
    if (SaferComputeTokenFromLevel(hAuthzLevel, NULL, &hToken,0,NULL)){
      //sets the restrict token for the current thread
      HANDLE hThread = GetCurrentThread();
      SetThreadToken(&hThread,NULL);
    }
    SaferCloseLevel(hAuthzLevel);
  }
}

advice fcd(abc): after(){
  HANDLE hThread = GetCurrentThread();
  SetThreadToken(&hThread,NULL); //removes restrict token
}
```

mutuelle (p.ex. moniteurs et sémaphores) autour de la section critique. Les débuts et fins de la section critique peuvent être visées en utilisant les points de jointure cca et fcd.

Le Listing 5, bien qu'il apparait correct, peut créer des effets secondaires si un des appels (supposons a et b) étaient conçus comme faisant partie d'une même section critique, et donc dans le même chemin d'exécution. Dans ce cas, le verrou serait libéré

Listing 5 – Aspect ajoutant de l'atomicité

```
static Semaphore sem = new Semaphore(1);

pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");

advice abc: before(){
  try{
    sem.acquire();
  } catch(InterruptedException e) {//...}
}

advice abc: after(){
  sem.release();
}
```

après a, et acquis de nouveau avant b, ce qui permet à une autre section critique d'exécuter, et potentiellement endommager des données partagées avec b. Pour améliorer la situation, il faut procéder à la programmation d'un aspect moins général, et nécessitant une bonne connaissance de l'application, allant donc à l'encontre des principes AOP. En utilisant notre proposition, par contre, le verrou est acquis et libéré de manière indépendante des points de jointure désirés mais garantit qu'ils seront collectivement considéré comme une section critique. Le Listing 6 montre cette amélioration.

Listing 6 – Aspect améliorant l'ajout d'atomicité

```
pointcut abc: call("% a(...)") || call("% b(...)") || call("% c(...)");

advice cca(abc): before(){
  static Semaphore sem = new Semaphore(1);
  try{
  sem.acquire();
  } catch(InterruptedException e) {//...}
}

advice fcd(abc): after(){
  sem.release();
}
```

3.2.4. *Enregistrement*

Il se peut qu'on veuille qu'un ensemble d'opérations d'intérêt puissent être enregistrées non pas individuellement, mais comme groupe, car leur entrée dans le journal serait redondante ou sans utilité réelle. Il est donc désirable d'utiliser le CCA et/ou le FCD de manière à injecter du code servant à l'enregistrement avant out après un ensemble de transactions d'intérêt. Un exemple serait similaire à celui de l'atomicité, et nous ne le répèterons pas pour des raisons d'espace.

4. Algorithmes et implémentation

Dans cette section, nous présentons les algorithmes élaborés pour l'étiquetage du graphe, et ceux pour la détermination de CAA et FCD. Nous supposons que notre CFG est formé de manière traditionnelle, avec un seul noeud de départ et un seul noeud de terminaison. Dans le cas d'un programme avec plusieurs points de départs, nous considérons chaque point de départ comme un programme différent dans notre analyse. Dans le cas où nous avons plusieurs points de terminaison, nous les considérons comme un seul point. La majorité de ces attentes ont déjà été utilisées pour fins de simplification [GOM]. Ce modèle étant en place, nous nous assurons que nos algorithmes retourneront un résultant (dans le pire des cas, ce sera le noeud de départ ou de terminaison) et que ce résultat sera un seul et unique noeud pour toutes les entrées.

4.1. *Étiquetage*

Des algorithme opérant sur les graphes sont bien connus depuis des décennies et plusieurs opérations (comme trouver les ancêtres, les descendants et les chemins) sont considérées comme étant connues de en informatique. Malgré cette richesse théorique, nous ne connaissons pas de méthode permettant de déterminer efficacement le CCA et le FCD pour un ensemble arbitraire de points de jointure d'un CFG qui prend compte de tous les chemins possibles. De manière à déterminer le CCA et FCD, nous avons choisi d'adapter un algorithme d'étiquetage hiérarchique de graphes développé par nos collègues afin de répondre à nos besoins. L'Algorithme 1 décrit cette méthode.

Chaque noeud dans la hiérarchie est étiqueté d'une manière semblable à une table des matières d'un livre (e.g. 1., 1.1., 1.2., 1.2.1., ...), comme nous l'avons montré dans l'Algorithme 1, où l'opérateur $+_c$ représente une concaténation de chaînes de caractères avec conversion de type implicite des opérandes. Pour l'appliquer, nous exécutions l'Algorithme 1 sur le noeud de départ avec l'étiquette "0.", ce qui numérote récursivement tous les noeuds.

Nous avons implémenté l'Algorithme 1 et l'avons testé sur un CFG hypothétique. Veuillez référer à la Figure 1 pour la visualisation des résultats. Ce CFG hypothétique nous servira d'exemple pour le reste de cet article.

4.2. *CCA*

Afin de calculer le CCA, nous avons développé un méchanisme qui opère sur le graphe étiqueté. Nous comparons toutes les étiquettes hiérarchiques des noeuds de l'ensemble d'entrée et trouvons le plus grand préfixe commun qu'elles partagent. Le noeud possédant cette étiquette est le CCA. Nous nous assurons que le CCA soit un noeud au travers duquel tous les chemins rejoignent les noeuds de l'ensemble d'entrée en prenant compte de toutes les étiquettes de chaque noeud. Le tout est élaboré dans l'Algorithme 2.

Algorithm 1 Algorithme d'étiquetage hiérarchique de graphes

```
1: labelNode(Node s, Label l) :
2: s.labels ← s.labels ∪ {l}
3: childrenSequence ← s.children()
4: for k = 0 to |childrenSequence| − 1 do
5:    child ← childrenSequence_k
6:    if ¬hasProperPrefix(child, s.labels) then
7:       labelNode(child, l +_c k +_c ".") ;
8:    end if
9: end for
10:
11: hasProperPrefix(Node s, LabelSet parentLabels) :
12: if s.label = ϵ then
13:    return false
14: end if
15: if ∃s ∈ prefixes(s.label) : s ∈ parentLabels then
16:    return true
17: else
18:    return false
19: end if
```

Algorithm 2 Algorithme pour déterminer le CCA

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
Require: $Graph$ has all its nodes labeled

```
1: Labels ← ∅
2: for all node ∈ SelectedNodes do
3:    Labels ← Labels ∪ node.labels()
4: end for
5: return FindCommonPrefix(Labels)
```

De plus, nous avons implémenté l'Algorithme 2 et nous l'avons appliqué au graphe de la Figure 1. Nous avons choisi, en étude de cas, quelques noeuds du graphe et l'avons soumis à l'algorithme CCA. Nos résultats, présentés dans la Table 1, montrent l'efficacité de l'algorithme.

Noeuds Choisis	**CCA**
N2, N8, N13	N1
N6, N11	N2
N14, N13	N1
N14, N15	N14

Tableau 1. *Résultats d'exécution de l'Algorithme 2 sur la Figure 1*

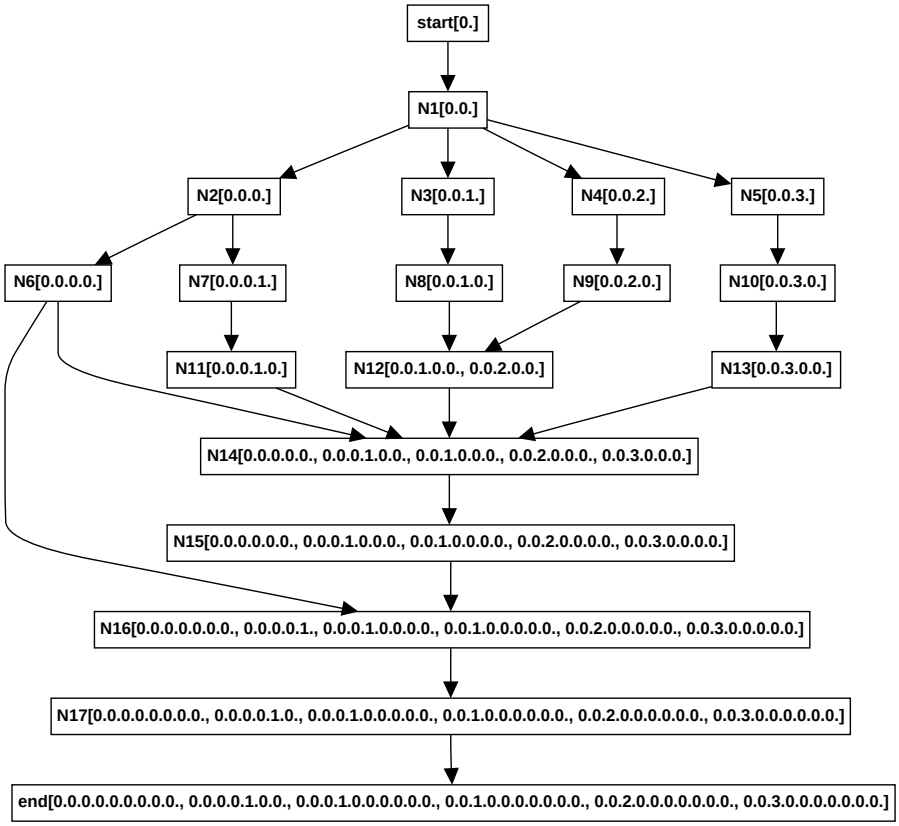

Figure 1. *Graphe étiqueté*

4.3. *FCD*

Nous avons également élaboré le mécanisme du FCD sur le CFG étiqueté d'un programme. L'Algorithme 3 montre comment obtenir une liste ordonnée de tous les descendants communs des noeuds d'entrée du point de coupure. Le principe de cet algorithme est de calculer l'ensemble des descendants de chaque noeud d'entrée et ensuite d'en calculer l'intersection. L'ensemble résultant contient les descendants communs de tous les noeuds d'entrée. Ensuite, nous les ordonnons selon leur longueur de chemin.

L'Algorithme 4 détermine le FCD. Il utilise d'abord les résultats de l'Algorithme 3, qu'il les considère comme liste de solutions potentielles. Ensuite, il itère sur cette liste jusqu'à ce qu'il trouve le noeud au travers lequel tous les chemins provenant

Algorithm 3 Algorithme pour déterminer les descendants communs

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
Require: $Graph$ has all its nodes labeled
findCommonDescendants(NodeSet SelectedNodes) :
$PossibleSolutions \leftarrow Graph.allNodes()$
for all $node \in SelectedNodes$ **do**
  $PossibleSolutions \leftarrow PossibleSolutions \cap node.AllDescendants()$
end for
Create $OrderedSolutions$ by sorting $PossibleSolutions$ by increasing path length between the solution and the nodes in $SelectedNodes$
return $OrderedSolutions$

Noeuds Choisis	Descendants Communs	FCD
N2, N8, N13	N14, N15, N16, N17, end	N16
N6, N11	N14, N15, N16, N17, end	N16
N14, N13	N15, N16, N17, end	N15
N14, N15	N16, N17, end	N16

Tableau 2. *Résultats d'exécution des Algorithmes 3 et 4 sur la Figure 1*

des noeuds d'entrée passent. Durant cette vérification, nous appelons le noeud présentement examiné le candidat. Pour chaque candidat, nous comptons le nombre d'étiquettes du candidat qui ont des préfixes identique aux étiquettes du noeud d'entrée considéré. À la fin de la première itération, nous gardons le candidat pour lequel nous avons trouvé en premier le plus grand décompte d'étiquettes. Ce candidat est le point de départ de la prochaine itération et il en est ainsi de suite, jusqu'à ce que tous les noeuds d'entrée soient examinés. Le candidat final de la dernière itération est retournée par l'algorithme comme étant le FCD.

En utilisant la même implémentation de l'Algorithme 1 et l'étude de cas illustrée en Figure 1, nous avons programmé les Algorithmes 3 et 4 et rapportons nos résultats dans la Table 2.

5. État de l'art

Un point de coupure pour la sécurité, basé sur les graphes de flot de données, utilisé pour identifier les points de jointure selon l'origine des valeurs est défini et formué en [MAS 03]. Ce point de coupure n'est pas pleinement implémenté à l'heure actuelle. Par exemple, ce point de coupure permet de détecter si les données envoyées sur le réseau provient d'information lue d'un fichier confidentiel.

Harbulot et Gurd [HAR 05] proposent un modèle de point de coupure sur les boucles qui montre le besoin d'un point de jointure sur les boucles qui prédit si une

Algorithm 4 Algorithme pour déterminer le FCD

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
Require: $Graph$ has all its nodes labeled

```
1: fcd(NodeSet SelectedNodes) :
2: PossibleSolutions ← findCommonDescendants(SelectedNodes)
3: Candidate ← 0
4: for all node ∈ SelectedNodes do
5:    Candidate ← findBestCandidate(PossibleSolutions, Candidate, node)
6: end for
7: return PossibleSolutions_Candidate
8:
9: findBestCandidate(NodeQueue possibleSolutions, int Candidate, Node
   selectedNode)
10: PreviousFoundPrefixes ← 0
11: for i ← Candidate to |possibleSolutions| − 1 do
12:    sol ← possibleSolutions_i
13:    foundPrefixes ← countProperPrefixes(sol, node)
14:    if (PreviousFoundPrefixes < foundPrefixes) ∨ ∃child ∈
       sol.children() : hasProperPrefix(sol, child.labels()) then
15:       Candidate ← i
16:    end if
17: end for
18: return Candidate
19:
20: countProperPrefixes(Node candidate, Node selectedNode) :
21: count ← 0
22: for all candidateLabel ∈ candidate.labels() do
23:    for all selectedNodeLabel ∈ selectedNode.labels() do
24:       if ∃p ∈ prefixes(candidateLabel) : p = selectedNodeLabel then
25:          count + +
26:       end if
27:    end for
28: end for
29: return count
```

boucle se répétera indéfiniment. Ce point de coupure détecte les boucles infinies utilisées par les attaquants afin de causer des dénis de service.

Un autre approche se concentre sur l'accès aux variables locales a été proposé par Myers [MYE 99]. Il affirme que ce point de coupure est nécessaire afin d'augmenter l'efficacité d'AOP en sécurité, puisqu'il permet de faire le suivi des valeurs des variables locales à une méthode. Il semble que ce point de coupure puisse être utilisé pour protéger la confidentialité des variables locales.

Dans [BON 05], Bonér présente un point de coupure qui permet de détecter le début d'un bloc de synchronisation et qui ajoute du code de sécurité limitant l'utilisation du CPU et le nombre d'instructions exécutées. Il a également exploré l'utilisation de ce point de coupure afin de calculer le temps nécessaire à l'acquisition d'un verrou et la gestion des enfilades. Cette contribution est utile en sécurité et peut aider à prévenir des attaques de déni de service.

Un point de coupure `pcflow` fut présenté par Kiczales dans un discours d'ouverture [KIC 03], mais ne fut jamais ni défini, ni intégré dans un language AOP. Un tel point de coupure pourrait permettre de choisir les points à l'intérieur du flot de contrôle à partir du début de l'exécution jusqu'au paramêtre du point de jointure. De plus, il a introduit l'idée d'obtenir un `min`imum de deux points de coupure `pcflow`, mais n'a pas défini clairement ce que ce `min` est ou comment l'obtenir.

Quant au monde des graphes, plusieurs algorithmes opérant sur les graphes pour trouver des ancêtres, descendants, chemins, etc. ont été proposés. Selon le meilleur de nos connaissances, aucun ne permet d'obtenir le CCA et le FCD comme nous l'avons défini. Les treillis ont été démontrés comme étant une représentation acceptable de programmes pour l'analyse statique [COU 77, KIL 73], et le calcul de borne supérieure minime (LUB) et borne inférieure maximale (GLB) est très efficace [AIT 89]. Toutefois, ces opérations ne garantissent pas que tous les chemins traverseront pas le noeud qu'ils offrent en résultat, ce qui est une exigence centrale du CCA et FCD.

6. Conclusion

AOP semble être un paradigme prometteur pour le renforcement de sécurité des logicielles. Toutefois, cette technologie ne fut pas conçu à la base pour s'occuper de problèmes de sécurité et plusieurs travaux de recherche ont montré ces limites dans cette domaine. Similairement, on a exploré dans cet article les manquements dans AOP pour appliquer plusieurs améliorations de sécurité. Dans ce contexte, nous avons proposé deux nouveaux points de coupure pour les préoccupations de renforcement de sécurité : le CCA et FCD. Le CCA retourne le point de jointure le plus près étant également ancêtre de tous les points d'intérêt et étant sur tous les chemins d'exécution menant à ces derniers. Le FCD retourne le point de jointure le plus près étant également descendant de tous les points d'intérêts et étant sur tous les chemins d'exécution provenant de ces derniers. Nous avons d'abord exploré les limites des langages AOP actuels pour certains problèmes de sécurité. Ensuite, nous illustrons l'utilité de nos points de coupure proposés pour accomplir certains renforcements de sécurité. Ensuite, nous avons défini les points de coupure et présenté leurs algorithmes. Finalement, nous avons présenté les résultats de notre implémentation sur une étude de cas.

7. Bibliographie

[AIT 89] AIT-KACI H., BOYER R. S., LINCOLN P., NASR R., « Efficient Implementation of Lattice Operations », *Programming Languages and Systems*, vol. 11, n° 1, 1989, p. 115-146.

[BIS] BISHOP M., « How Attackers Break Programs, and How to Write More Secure Programs », `http://nob.cs.ucdavis.edu/Ÿbishop/secprog/sans2002/index.html`.

[BOD 04] BODKIN R., « Enterprise Security Aspects », 2004, `http://citeseer.ist.psu.edu/702193.html`.

[BON 05] BONÉR J., « Semantics for a Synchronized Block Join Point », 2005, Blogue :`http://jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-joint-point/`.

[Cig 03] CIGITAL LABS, « An Aspect-Oriented Security Assurance Solution », rapport n° AFRL-IF-RS-TR-2003-254, 2003, Cigital Labs.

[COU 77] COUSOT P., COUSOT R., « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints », *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, 1977, ACM Press, New York, NY, p. 238–252.

[DEW 04] DEWIN B., « Engineering Application Level Security through Aspect Oriented Software Development », PhD thesis, Katholieke Universiteit Leuven, 2004.

[GOM] GOMEZ E., « CS624- Notes on Control Flow Graph », `http://www.csci.csusb.edu/egomez/cs624/cfg.pdf`.

[HAR 05] HARBULOT B., GURD J., « A Join Point for Loops in AspectJ », *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005), March*, 2005.

[HOW 02] HOWARD M., LEBLANC D. E., *Writing Secure Code*, Microsoft Press, Redmond, WA, USA, 2002.

[HUA 04] HUANG M., WANG C., ZHANG L., « Toward a Reusable and Generic Security Aspect Library », *AOSD :AOSDSEC 04 : AOSD Technology for Application level Security, March*, 2004.

[KIC 03] KICZALES G., « The Fun has Just Begun (discours programme) », *International Conference on Aspect-Oriented Software Development*, 2003, Disponible seulement à `http://www.cs.ubc.ca/gregor/`.

[KIL 73] KILDALL G. A., « A unified approach to global program optimization », *POPL '73 : Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, 1973, ACM Press, p. 194–206.

[MAS 03] MASUHARA H., KAWAUCHI K., « Dataflow Pointcut in Aspect-Oriented Programming », *Programming Languages and Systems*, 2003, p. 105-121.

[MYE 99] MYERS A., « JFLow : Practical Mostly-Static Information Flow Control », *Symposium on Principles of Programming Languages*, 1999, p. 228-241.

[SEA 05] SEACORD R., *Secure Coding in C and C++*, SEI Series, Addison-Wesley, 2005.

[SLO 03] SLOWIKOWSKI P., ZIELINSKI K., « Comparison Study of Aspect-oriented and Container Managed Security », 2003.

[WHE 03] WHEELER D., *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010*, 2003, `http://www.dwheeler.com/secure-programs/`.