Nouveaux Points de Coupure et Primitives pour les Préoccupations de Renforcement de Sécurité

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

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

RÉSUMÉ. *Nous présentons dans cet article deux nouveaux points de coupure et deux nouvelles primitives pour programmation par aspect nécessaires au renforcement systématique de sécurité. Les deux points de coupure proposés permettent l'identification de certains points de jointure dans un graphe de flot de contrôle. Le premier est le* GAFlow*, l'ancêtre garanti le plus près, qui retourne l'ancêtre le plus près étant sur le chemin de tous les points d'intérêt. Le deuxième est le* GDFlow*, le descendant garanti le plus près, qui détermine le point de jointure descendant le plus près pouvant être traversé par tous les chemins émanant des points de coupure d'intérêt. Les deux primitives proposées sont appelées* exportParameter *et* importParameter *et sont utilisées pour passer les paramètres entre deux points de coupure. Elles permettent d'analyser le graphe d'appel d'un programme pour déterminer comment changer les signatures des fonctions pour passer les paramètres associés au renforcement de sécurité. Nous croyons que ces points de coupures et primitives sont nécessaires pour plusieurs pratiques de renforcement de sécurité et, à notre connaissance, aucun des existants peut offrir leurs fonctionnalités. De plus, nous démontrons la viabilité et la justesse de notre proposition en élaborant et implantant les algorithmes de points de coupure et primitives et montrant les résultats des études de cas.*

ABSTRACT. *In this paper, we present two new pointcuts and two new Primitives to Aspect-Oriented Programming (AOP) languages that are needed for systematic hardening of security concerns. The two proposed pointcuts allow to identify particular join points in a program's control flow graph (CFG). The first one is the* GAFlow*, the Closest Guaranteed Ancestor, which returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The second one is the* GDFlow*, the Closest Guaranteed Descendant, which returns the closest child join point that can be reached by all paths starting from the pointcuts of interest. The two proposed primitives are called* exportParameter *and* importParameter *and are used to pass parameters between two pointcuts. They allow to analyze a program's call graph in order to determine how to change function signatures for the passing of parameters associated with a given security hardening. We find these pointcuts and primitives to be necessary because they*

Technique et science informatiques., pages 0 à 0

are needed to perform many security hardening practices and, to the best of our knowledge, none of the existing ones can provide their functionalities. Moreover, we show the viability and correctness of our proposed pointcuts and primitives by elaborating and implementing their algorithms and presenting the results of explanatory case studies.

MOTS-CLÉS : *Sécurité, Programmation Orientée Aspect, Graphe de Flots de Contrôle, Graphe d'Appels, Point de Coupure, Primitive*

KEYWORDS: *Security Hardening, Aspect-Oriented Programming, Control Flow Graph (CFG), Call Graph, Pointcut, Primitive*

1. Motivations

La sécurité informatique gagne actuellement un rôle prédominant dans l'industrie des technologies de l'information. Cette 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 d'une telle manière. Les systèmes d'un certain âge offrent un défi supplémentaire quand ils doivent être adaptés aux environnements réseau et web, malgré qu'ils ne furent pas conçus pour évoluer dans un cadre disposant d'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 lorsque 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 est possible.

Conséquemment, l'intégration de la sécurité dans les logiciels représente un domaine de recherche fort intéressant et actuel. Nous avons défini précédemment (Mourad *et al.*, 2006) le renforcement de sécurité logicielle comme étant *tout processus, méthodologie, produit ou combinaison de ces derniers étant utilisés pour ajouter des fonctionnalités de sécurité, enlever des vulnérabilités, ou prévenir leur exploitation dans les logiciels existants*. Peu de concepts et approches pour le renforcement de sécurité logicielle ont émergé dans la littérature, et consistent principalement de patrons de sécurité et de bonnes pratiques de programmation sécuritaire (Schumacher, 2003; Howard *et al.*, 2002). Leurs pratiques de renforcement de sécurité sont typiquement effectuées par intégration et l'injection manuelle de composants et code rattaché à la sécurité dans le logiciel (Seacord, 2005; Howard *et al.*, 2002; Wheeler, 2003; Bishop, 2005). Cette tâche exige que les développeurs aient une bonne connaissance du fonctionnement internes du programme, en plus d'une expertise avancée dans la domaine de sécurité appliquée.

Toutefois, sécuriser un logiciel demeure une procédure critique et difficile. Si elle est appliquée manuellement, surtout pour des systèmes de grande taille (p.ex. plusieurs milliers, sinon millions de lignes de code), une grande expertise de sécurité et beaucoup de temps sont requises. Il y a aussi toujours un risque d'introduction de nouvelles vulnérabilités. De plus, il est difficile de trouver les développeurs qui sont experts en même temps dans les domaines de sécurité et de fonctionnalités du logiciel. Dans le fond, c'est un problème ouvert souligné par plusieurs gestionnaires. Les approches actuelles ne prennent pas en compte ces problèmes, et l'industrie nécessite des solutions le faisant. Notre approche cherche donc à créer des méthodes et des approches pour intégrer systématiquement des modèles et composantes de sécurité dans les logiciels (Mourad *et al.*, 2007b; Mourad *et al.*, 2007a).

Une approche pour atteindre ces objectives est d'extraire les préoccupations de sécurité du reste de l'application, de manière à ce qu'ils puissent être résolus de manière

indépendante tout en étant appliqués globalement. Récemment, plusieurs propositions se sont distinguées pour l'injection de code de sécurité via le style de programmation orienté aspect (Bodkin, 2004; DeWin, 2004; Huang *et al.*, 2004; Shah, 2003). À cet égard, la programmation par aspect semble être un paradigme prometteur pour le renforcement de sécurité des logiciels, un domaine n'ayant pas été touché adéquatement par les paradigmes antérieurs. L'idée centrale de la programmation par aspect 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.

Toutefois, la programmation par aspect ne fut pas conçu à la base pour résoudre des problématiques de sécurité, et plusieurs manquements ont été signalés (Masuhara *et al.*, 2003; Harbulot *et al.*, 2005; Myers, 1999; Bonér, 2005; Kiczales, 2003). Nous étions incapables de mettre en pratique des améliorations de sécurité à cause de certaines fonctionnalités manquantes. Par exemple, alors que nous implantions une solution pour sécuriser les connections d'une application client-serveur, nous devions injecter du code dans le *main*, en plus d'utiliser des structures de données agglomérant différentes informations d'état pour la librairie GnuTLS. Dans ce cas, bien des appels sont exécutés, incluant des initialisation/dé-initialisation et construction de structures de données de la librairie, même si la fonctionnalité n'est pas utilisée. Bien que cette solution ait marché pour une petite application ayant une simple fonction, elle ne serait pas applicable pour une application complexe ayant de multiples fonctions. De plus, cette approche serait problématiques pour des applications embarquées, alors que les ressources disponibles sont très limitées.

De plus, durant nos expériences de renforcement de sécurité, nous devions passer des variables supplémentaires pour la librairie GnuTLS (p.ex. TLS session) entre les composants de l'application. De telles limitations nous ont forcé à faire des acrobaties de programmation, résultant en l'intégration de modules et de data structures supplémentaires et le changement de certaines fonctions dans l'application pour transmettre les variables. Une telle solution n'est pas réaliste pour des grandes applications ayant de fortes et complexes dépendances entre ses composants. Un changement dans un composant pourrait nécessiter plusieurs modifications complexes en cascade (p.ex. changements dans la conception de logiciel).

Par conséquent, nous proposons deux nouveaux points de coupure nécessaires afin d'identifier des points d'intérêts dans le graphe de flot de contrôle (CFG) d'un programme. Les points de coupure proposées sont *GAFlow*, l'ancêtre garanti le plus près et le *GDFlow*, le descendant garanti le plus près. Le *GAFlow* retourne le point de jointure le plus près, ancêtre de tous les points d'intérêt et sur tous les chemins d'exécution menant à ces derniers. Le *GDFlow* retourne le point de jointure le plus près, descendant de tous les points d'intérêts et sur tous les chemins d'exécution provenant de ces derniers. De plus, nous proposons deux primitives utilisées pour passer les pa-

ramètres entre deux points de coupure. Les deux primitives sont *exportParameter* et *importParameter*. Elles permettent d'analyser le graphe d'appel d'un programme pour déterminer comment changer les signatures des fonctions pour passer les paramètres associés au renforcement de sécurité.

Ces points de coupure et primitives sont nécessaires afin d'adresser les problèmes mentionnés ci-haut et réaliser les améliorations de sécurité discutées dans la section 3, puisque aucune des propositions actuelle est en mesure de le faire. Bien que l'utilité des points de coupure puissent être d'intérêt dans d'autres domaines, nous limiterons notre discussion au domaine du renforcement de sécurité.

Nous avons organisé cet article de la manière suivante : nous spécifions premièrement les points de coupures et les primitives dans la section 2. Par après, nous discutons de l'utilité de notre proposition et de ses avantages dans la section 3. Ensuite, dans la section 4, nous présentons les méthodologies et les algorithmes nécessaires pour l'implantation des points de coupure, primitives et dominateur, accompagnés d'une méthode d'étiquetage hiérarchique de graphe. Cette section contient également les résultats de nos études 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 et des Primitives

Dans cette section, nous définissons la syntaxe et réalisons les points de coupure et les primitives proposés. Le Tableau 1 montre cette syntaxe qui définit le point de coupure *p* et la déclaration des conseils `advice <p>`.

```
p : := call(s) | execution(s) | GAFlow(p) | GDFlow(p) | p||p | p&&p

parameter : := <type> <identifier>
paramList : := parameter [,paramList]
e : := ExportParameter(<paramList>)
i : := ImportParameter(<paramList>)
advice <p> : (before|after|around) (<arguments>)
[ : e | i | e,i] {<advice-body>}
```

Tableau 1. *Syntaxe pour les Points de Coupure et les Primitives*

Le symbole *s* désigne une signature de fonction. Les *GAFlow* et *GDFlow* sont les nouveaux points de coupure que nous proposons. Leurs paramètres sont également un point de coupure *p*. *e* et *i* sont respectivement les nouvelles primitives, *exportParameter* et *importparameter*. Les arguments sont les paramètres à passer et à recevoir. Nous définissons chaque point de coupure et primitive dans les suivantes.

2.1. *Points de coupure* GAFlow *et* GDFlow

Le point de coupure *GAFlow* a comme entrée 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 tous les noeuds de l'ensemble d'entrée et (2) le noeud à travers lequel passent tous les chemins qui les rejoint. Dans le pire des cas, le *GAFlow* sera le point de départ (p.ex. `main`) du programme.

Le point de coupure *GDFlow* a comme entrée 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) le premier noeud commun sur tous les chemins émanant des noeuds sélectionnés. Dans le pire des cas, le *GDFlow* sera le point de terminaison du programme.

2.2. *Primitives* ExportParameter *et* ImportParameter

Les deux primitives *exportParameter* et *importParameter* doivent être combinées de manière à ce que l'information soit transmise d'un point de jointure à un autre. Leur paramètre est le variable à transmettre. Le noeud d'origine est le point de jointure où *exportParameter* est utilisé, tandis que le noeud de la destination est le point de jointure où *importParameter* est utilisé.

3. Discussion

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

3.1. *Utilité de* GAFlow *et* GDFlow *pour le Renforcement de Sécurité*

Plusieurs pratiques de renforcement de sécurité nécessitent l'injection de code autour d'un ensemble de points de jointure ou chemins d'exécution (Seacord, 2005; Howard *et al.*, 2002; Wheeler, 2003; Bishop, 2005). Des exemples de tels cas sont l'injection d'initialisation, de dé-initialisation et de construction de structures de données pour les librairies de sécurité, changement de niveaux de privilèges, garantie d'atomicité, l'enregistrement, etc. Le modèle actuel en programmation par aspect 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 ne permet d'identifier un point de jointure commun à un ensemble de points de jointure et satisfaisant les critères de *GAFlow* et *GDFlow* où nous pouvons injecter le code juste quand c'est nécessaire et une fois pour tous. Il nous faut donc de nouveaux points de coupure permettant aux programmeurs d'aspects de créer des solutions abstraites et adaptables aux changements de programmes. Dans ce qui suit, nous présentons l'utilité et la nécessité de nos points de coupure proposés pour le renforcement de sécurité.

3.1.1. *Initialisation, Dé-initialisation et Construction de Structures de Données pour les Librairies de Sécurité*

En implantant une solution pour sécuriser les connections d'une application client-serveur, nous devions injecter du code dans le *main*, en plus d'utiliser des structures de données agglomérant différentes informations d'état pour la librairie GnuTLS. Dans ce cas, bien des appels sont exécutés, incluant des initialisation/dé-initialisation et construction de structures de données pour les libraires, même si la fonctionnalité n'est pas utilisée. Bien que cette solution ait marché pour une petite application ayant une simple fonction, elle ne serait pas applicable pour une application complexe ayant de multiples fonctions. De plus, cette approche serait problématiques pour des applications embarquées, alors que les ressources disponibles sont très limitées. Nos points de coupure permettent de résoudre ce problème en exécutant ces appels seulement pour les branches du code pour lesquelles elles sont nécessaires en identifiant le *GAFlow* et le *GDFlow*. En utilisant les deux points de coupure de concert, nous évitons aussi de devoir utiliser des variables globales documentant le statut de la librairie. Dans l'exemple ci-bas, nous montrons un extrait de l'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, tel que montré dans le Listing 1. Dans le Listing 2, nous présentons un aspect amélioré ciblant le *GAFlow* et le *GDFlow* et offrant des résultats applicables.

Listing 1 – Partie d'Aspect Sécurisant des connections avec GnuTLS

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

Listing 2 – Partie d'Aspect Sécurisant des Connections avec GnuTLS GAFlow et GDFlow

```
advice GAFlow(call("% connect(...)") || call("% send(...)") || call("%
    recv(...)") || call("% close(...)")): before(){
```

```
  //Initialization of the API
  hardening_socketInfoStorageInit();
  hardening_initGnuTLSSubsystem(NONE);
  }

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

3.1.2. *Principe du Moindre Privilège*

Pour les processus implantant 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 points de coupure peuvent être utilisées pour opérer sur un groupe d'opérations nécessitant le même privilège en injectant du code ajustant les droits d'accès aux points *GAFlow* et *GDFlow*. L'exemple dans le Listing 3 (fait à partir d'une combinaison d'exemples provenant de (Howard *et al.*, 2002)) montre un aspect hypothétique implantant 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 *GAFlow* et le *GDFlow*, ce que nous montrons dans le Listing 4.

Listing 3 – Aspect 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 GAFlow(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 GDFlow(abc): after(){
  HANDLE hThread = GetCurrentThread();
  SetThreadToken(&hThread,NULL); //removes restrict token
}
```

3.1.3. *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 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 *GAFlow* et *GDFlow*.

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();
}
```

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 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 amé-

liorer la situation, il faut procéder à la programmation d'un aspect moins général, ce qui nécessite une bonne connaissance de l'application, allant donc à l'encontre des principes de programmation par aspect. 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 GAFlow(abc): before(){
  static Semaphore sem = new Semaphore(1);
  try{
  sem.acquire();
  } catch(InterruptedException e) {//...}
}

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

3.1.4. *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 *GAFlow* et/ou le *GDFlow* de manière à injecter du code servant à l'enregistrement avant ou après un ensemble de transactions d'intérêt. Un exemple serait similaire à celui de l'atomicité.

3.2. *Avantages Généraux de* GAFlow *et* GDFlow

Il est clair que les points de coupure que nous proposons supportent le principe de séparation des préoccupations en permettant l'implantation de modifications au programme sur un ensemble de points de coupure selon la préoccupation (comme démontré précédemment). Puisque l'utilité des points de coupure puissent être d'intérêt dans d'autres domaines, nous allons plus loin en montrant des avantages plus généraux rattachés à notre 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 il faut faire.

– *Facilité de Maintenance* : les programmeurs peuvent changer la structure de logiciels sans se préoccuper des aspects associés. Sans *GAFlow* et *GDFlow*, 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 d'une certaine fonction, nous devons changer le nouvel ancêtre commun couvrant tous les chemins, alors que nos points de coupure le font automatiquement.

– *Temps d'Exécution et Consommation de Mémoire* : les pré-opérations et post-opérations sont injectées une seule fois dans le programme, où elles sont nécessaires. Cela évite de surcharger l'initialisation du programme, ce qui autrement diminuerait son niveau de réponse apparent. Certaines opérations coûteuses peuvent être évitées si les branches de code les exigeant ne sont jamais exécutées, épargnant donc des cycles de processeur et diminuant la mémoire utilisée. De plus, nous évitons une injection autour de chaque point de jointure d'intérêt, solution par défaut offerte par les langages de programmation par aspect. Cela serait remplacé par une seule injection autour du *GAFlow* et *GDFlow*.

3.3. *Utilité de* ExportParameter *et* ImportParameter *pour le Renforcement de Sécurité*

Dans cette section, nous illustrons l'utilité de *importParameter* et *exportParameter* pour le renforcement de sécurité en (1) présentant un exemple servant à sécuriser le canal entre deux points connectés du réseau en utilisant les technologies de programmation par aspect actuels, (2) montrant le besoin de passer les paramètres et (3) présentant la solution de cet exemple en utilisant notre proposition.

3.3.1. *Sécuriser une Connexion Utilisant les Technologies de Programmation par Aspect Actuelles*

Sécuriser le canal entre deux points est une des solutions utilisées pour éviter les violations de confidentialité, d'intégrité et le détournement de connexions. Le protocole TLS (Transport Layer Security) est l'une des options populaires pour accomplir ceci. Nous présentons donc dans cette section un extrait d'une étude de cas, dans laquelle nous avons implanté en utilisant TLS un aspect de sécurisation de connexion avec AspectC++.

Dans le but de généraliser notre solution et la rendre applicable sur une vaste gamme d'applications, nous devons supposer que certaines communications ne seront pas sécurisées (p.ex. les interfaces de connexion pouvant être utilisées pour effectuer des communications interprocessus). Dans le cas où la communication doit être sécurisée, les appels de fonctions opérant sur les canaux sont remplacés par ceux offerts par la librairie TLS. Par contre, les autres fonctions ne seront pas touchées. De plus, nous gérons bien les cas où ces fonctions sont dispersées dans différentes composantes.

Listing 7 – Extrait d'un Aspect AspectC++ Renforçant des Connexions Utilisant GnuTLS

```
aspect SecureConnection {

 advice execution ("% main(...)") : around () {
   hardening_socketInfoStorageInit();
   hardening_initGnuTLSSubsystem(NONE);

   tjp->proceed();

   hardening_deinitGnuTLSSubsystem();
```

```
    hardening_socketInfoStorageDeinit();
  }

  advice call("% connect(...)") : around () {

    //variables declared
    hardening_sockinfo_t socketInfo;
    const int cert_type_priority[3] = { GNUTLS_CRT_X509, GNUTLS_CRT_OPENPGP,
         0};

    //initialize TLS session info
    gnutls_init (&socketInfo.session, GNUTLS_CLIENT);
    gnutls_set_default_priority (socketInfo.session);
    gnutls_certificate_type_set_priority (socketInfo.session,
        cert_type_priority);
    gnutls_certificate_allocate_credentials (&socketInfo.xcred);
    gnutls_credentials_set (socketInfo.session, GNUTLS_CRD_CERTIFICATE,
        socketInfo.xcred);

    //Connect
    tjp->proceed();
    if(*tjp->result()<0) {perror("cannot connect "); exit(1);}

    //Save the needed parameters and the information that distinguishes
        between secure and non-secure channels
    socketInfo.isSecure = true;
    socketInfo.socketDescriptor=*(int *)tjp->arg(0);
    hardening_storeSocketInfo(*(int *)tjp->arg(0), socketInfo);
    //TLS handshake
    gnutls_transport_set_ptr(socketInfo.session, (gnutls_transport_ptr)(*(
        int *)tjp->arg(0)));
    *tjp->result() = gnutls_handshake (socketInfo.session);
  }

  //replacing send() by gnutls_record_send() on a secured socket
  advice call("% send(...)") : around () {

    //Retrieve the needed parameters and the information that distinguishes
        between secure and non-secure channels
    hardening_sockinfo_t socketInfo;
    socketInfo = hardening_getSocketInfo(*(int *)tjp->arg(0));

    //Check if the channel, on which the send function operates, is secured
        or not
    if (socketInfo.isSecure)
       //if the channel is secured, replace the send by gnutls_send
       *(tjp->result())  = gnutls_record_send(socketInfo.session, *(char**)
           tjp->arg(1), *(int *)tjp->arg(2));
    else
       tjp->proceed();
  }
};
```

Dans le Listing 7, nous voyons un extrait du code AspectC++ sécurisant une connexion. Nous ajoutons premièrement dans le fonction `main` le code nécessaire pour initialiser l'API GNU/TLS. Ensuite, nous ajoutons avant et après la fonction `connect` le code responsable d'initialiser la session TLS et d'amorcer la communication. Finalement, nous remplaçons les fonctions responsables d'envoyer et recevoir les données `send/receive` par leurs versions sécurisés de TLS.

En observant aussi le Listing 7, le lecteur remarquera l'apparition de la structure de données `hardening_sockinfo_t`, ainsi que de fonctions associées. Nous avons

souligné ces dernières afin de faciliter la lecture. Ces fonctions et structures de données furent développées afin de distinguer entre les canaux sécurisés et non-sécurisés ainsi que pour exporter les paramètres entre les composantes de l'application au moment de l'exécution. Le problème majeur dans la transmission du paramètre entre les différentes composantes est l'incompatibilité de type entre l'interface de connexion des librairies réseau standard avec les structures de données GnuTLS.

Afin d'éviter l'utilisation directe de mémoire partagée, nous choisissons une table de hachage utilisant le numéro d'interface de connexion comme clé. Nous enregistrons et récupérons ainsi toutes les informations nécessaires dans une structure de données conçue à cette fin. Une information supplémentaire que nous conservons est si le canal est sécurisé ou non. Ainsi, tous les appels à `send()` et `recv()` sont modifiés par une vérification faite à l'exécution permettant d'utiliser les bonnes fonctions de transmission. Cette approche de partage du paramètre représente des coûts de mémoire et de temps d'exécution qui pourrait être évité par une primitive automatisant le transfert de données à l'intérieur d'aspects sans augmenter la complexité du logiciel. De plus, des expériences avec une autre mesure de sécurité (p.ex. l'encryptage de sections de mémoire) on montré que l'utilisation d'une table de hachage ne pouvait être généralisée facilement pour un langage tel que le `C`, puisque l'étendue de valeurs de certains types (p.ex. `unsigned char`) est trop limitée pour que certaines clés puissent être utilisées.

3.3.2. *Le Besoin de Passer les Paramètres*

Notre revue de la littérature (Seacord, 2005; Howard *et al.*, 2002; Wheeler, 2003; Bishop, 2005) et nos travaux antérieurs (Mourad *et al.*, 2007b) ont démontré qu'il est parfois nécessaire de transmettre de l'information sur l'état d'une partie d'un programme à un autre quand nous renforçons la sécurité. Par exemple, en observant le Listing 7, nous devons transmettre la structure de données `gnutls_session_t` du conseil autour de `connect` à celui autour de `send`, `receive` et `close`. Bien que cela soit nécessaire pour renforcer la connexion, les modèles de programmation par aspect actuels ne permettent pas de faire une telle manoeuvre. De telles limitations nous ont forcé à faire des acrobaties de programmation (mentionnées dans la sous-section précédant), résultant en l'intégration de modules et de data structures supplémentaires et le changement de certaines fonctions dans l'application pour transmettre les variables. Une telle solution n'est pas réaliste pour des grandes applications ayant de fortes et complexes dépendances entre ses composants. Un changement dans un composant pourrait nécessiter plusieurs modifications complexes en cascade (p.ex. changements dans la conception de logiciel).

3.3.3. *Sécuriser une Connexion Utilisant* ExportParameter *et* ImportParameter

Nous avons modifié le même exemple du Listing 7 en utilisant notre proposition pour le passage de paramètre. Le Listing 8 montre un extrait du nouveau code. Tous les structures de données et algorithmes (soulignés dans le Listing 7) sont enlevés et remplacés par les primitives d'importation et d'exportation. Un *exportParameter* est ajouté au conseil pour la fonction `connect`, exportant les paramètres `session`

et `xcred`. De plus, un *importParameter* est ajouté au conseil pour la fonction `send`, important ainsi le paramètre `session`.

Listing 8 – Sécurisation de Connexion Utilisant GnuTLS et le Passage de Paramètres Proposé

```
aspect SecureConnection {

 advice execution ("% main(...)") : around () {
   gnutls_global_init ();
   tjp->proceed();
   gnutls_global_deinit();
 }

 advice call("% connect(...)") : around () : exportParameter(gnutls_session
      session, gnutls_certificate_credentials xcred){
   //variables declared
   static const int cert_type_priority[3] = { GNUTLS_CRT_X509,
       GNUTLS_CRT_OPENPGP, 0};

   //initialize TLS session info
   gnutls_init (&session, GNUTLS_CLIENT);
   gnutls_set_default_priority (session);
   gnutls_certificate_type_set_priority (session, cert_type_priority);
   gnutls_certificate_allocate_credentials (&xcred);
   gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);

   //Connect
   tjp->proceed();
   if(*tjp->result()<0) {perror("cannot connect "); exit(1);}

   //TLS handshake
   gnutls_transport_set_ptr (session, (gnutls_transport_ptr) (*(int *)tjp->
       arg(0)));
   *tjp->result() = gnutls_handshake (session);
 }

 //replacing send() by gnutls_record_send() on a secured socket
 advice call("% send(...)") : around () : importParameter(gnutls_session
     session){

 //Check if the channel, on which the send function operates, is secured or
      not
   if (session != NULL)
      //if the channel is secured, replace the send by gnutls_send
      *(tjp->result())  = gnutls_record_send(*session, *(char**) tjp->arg
          (1), *(int *)tjp->arg(2));
   else
      tjp->proceed();
 }
};
```

4. Méthodologie, Algorithmes et Implantation

Dans cette section, nous présentons les méthodologies et les algorithmes élaborés pour l'étiquetage du graphe, et ceux pour la détermination de l'ensemble des dominateurs, *GAFlow*, *GDFlow*, *importParameter* and *exportParameter*. Des algorithmes opérant sur les graphes sont bien connus depuis des décennies (p.ex. trouver les ancêtres, les descendants et les chemins). Malgré cette richesse théorique, nous

ne connaissons pas de méthode permettant de déterminer le *GAFlow* et le *GDFlow* pour un ensemble arbitraire de points de jointure d'un CFG qui prend compte de tous les chemins possibles. Par contre, les algorithmes utilisés pour calculer les ensembles dominateur et post-dominateur d'un noeud de graphe de flots de contrôle prennent en charge de tels critères. Il est donc possible de les étendre et les utiliser pour établir les algorithmes de *GAFlow* et *GDFlow*.

Nous proposons donc deux ensembles d'algorithmes pour *GAFlow* et *GDFlow*. La première proposition est basée sur les dominateurs et post-dominateurs d'un graphe de flots de contrôle classique, tandis que la deuxième opère sur un graphe étiqueté. Les développeurs sont libres de choisir l'un ou l'autre pour leur implantation, selon leurs besoins. 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 départ, nous considérons chaque point de départ comme un programme différent dans notre analyse. La majorité de ces attentes ont déjà été utilisées pour fins de simplification (Gomez, 2003). Une fois ce modèle étant en place, nous nous assurons que nos algorithmes retourneront un résultat et que ce résultat sera un seul et unique noeud pour toutes les entrées.

4.1. GAFlow *et* GDFlow *via les Dominateurs et Post-Dominateurs*

Le problème d'identification des dominateurs et post-dominateurs dans un graphe de flots de contrôle est bien documenté dans la littérature et plusieurs algorithmes furent proposés, améliorés et implantés (Cooper *et al.*, 2001; Holloway *et al.*, 1998a). Pour calculer l'information de dominance, tel que présenté par (Cooper *et al.*, 2001), le compilateur peut annoter chaque noeud du graphe de flots de contrôle avec les ensembles *DOM* and *PDOM*.

DOM(b) : Un noeud *n* dans le graphe de flots de contrôle domine *b* si *n* est sur tous les chemins entre le noeud de départ et *b*. *DOM(b)* contient tout noeud *n* dominant *b*, incluant *b*.

Les dominateurs du noeud *n* sont déterminés par la solution maximale de l'équation de flots de données suivante :

$$Dom(entry) = \{entry\} \quad [1]$$

$$Dom(n) = \left(\bigcap_{p \in preds(n)} Dom(p) \right) \bigcup \{n\} \quad [2]$$

Ici, *entry* est le noeud de départ du graphe. Le dominateur du noeud de départ est le noeud de départ. L'ensemble des dominateurs pour tout autre noeud *n* est l'intersection des ensembles de dominateurs pour tous les prédécesseurs *p* de *n*. Le noeud *n* est

également dans l'ensemble des dominateurs de *n*. Pour résoudre ces équations, nous pouvons utiliser l'algorithme itératif de (Cooper *et al.*, 2001).

Nous avons élaboré et implanté l'Algorithme 1 pour calculer l'ensemble des dominateurs. Nous utilisons un algorithme d'identification de chemins possibles entre deux noeuds provenant de (Dijkstra, n.d.), bien que tout algorithme qui donne des résultats similaires puisse être utilisé. Ce choix est complètement laissé au développeur expert dans ce domaine.

L'algorithme proposé permet de déterminer les noeuds dominants non-triviaux du noeud n. La solution au problème s'articule autour de la détermination de tous les chemins reliant les noeuds n et $entryPoint$, tout en ne conservant que les noeuds communs entre ces chemins. L'algorithme utilisé pour trouver les chemins utilise un *marking map overlay* créé récursivement sur les noeuds, en débutant par n et terminant par $entryPoint$. Plus précisément, à chaque noeud marqué, nous obtenons un tableau contenant des ensembles clé-valeur, avec les clés correspondant aux noeuds précédents et une incrémentation des valeurs de marquage, en relation avec les noeuds en question. Seul le noeud n a comme marquage lui-même comme clé, et comme valeur de marquage 0.

Quand le marquage est complété, nous pouvons explorer les noeuds récursivement de manière à tracer les chemins. En principe, à chaque noeud exploré, les marquages du noeud parent sont comparés à ceux du noeud actuel de manière à déterminer une séquence adjacente de valeurs croissantes représentant une branche non visitée. La procédure utilise une pile sur laquelle nous ajoutons le noeud actuel au début de son exécution, et où ce dernier est retiré lors à la fin de l'exécution. Cette pile est la liste des ascendants, et donc le chemin exploré actuellement. Si le noeud actuel est la cible (c-à-d. le noeud $entryPoint$), ce chemin est ajouté à la liste des chemins trouvés. Durant l'exécution, quand nous trouvons une nouvelle branche non visitée, le marquage correspondant est temporairement retiré du tableau du noeud actuel, et la procédure est appelée sur le noeud parent. Quand l'exécution de cet appel est complété, le marquage est ajouté de nouveau au noeud, permettant ainsi la détection d'autres chemins passant par ce noeud.

PDOM(b) : Un noeud *n* dans un graphe de flots de contrôle post-domine *b* si *n* est sur tous les chemins reliant *b* au noeud de sortie du graphe. *PDOM(b)* contient tout noeud *n* post-dominant *b*, incluant *b*.

Une méthode simple pour calculer l'ensemble de post-dominateurs est d'inverser la direction des flèches dans le graphe, et d'appliquer l'algorithme du dominateur à partir du noeud de sortie (Holloway *et al.*, 1998a). Le post-dominateur du noeud de sortie est le noeud de sortie. Dans le cas où de multiples points de sortie sont présents, nous considérons chaque point de sortie comme un programme différent pour notre analyse (dans le fond, chaque point de sortie va être un point de départ après l'application du mécanisme d'inversion du graphe).

Algorithm 1 Algorithme Déterminant l'Ensemble des Dominateurs

```
1: Set DOM(Node entryPointNode, Node n)
2: mark(entryPointNode, n) ;
3: Stack pathList = new Stack() ;
4: tracePath(entryPointNode, n, new Stack(), pathList) ;
5: Set meetSet = {} ;
6: if !pathList.isEmpty() then
7:    meetSet.addAll(pathList.pop()) ;
8:    for each path pth in pathList do
9:       meetSet = meetSet ⋂ (Set)pth
10:   end for
11: end if
12: return meetSet ;
13:
14: markNode(Node targetNode, Node currentNode, Node branchingNode, int
    markIndex)
15: if ∄ currentNode.pathMarkingMap.get(branchingNode) then
16:   currentNode.pathMarkingMap.put(branchingNode, markIndex) ;
17:   if currentNode != targetNode then
18:      markIndex = markIndex + 1 ;
19:      for each parent p in currentNode.parentList do
20:         markNode(targetNode, p, currentNode, markIndex) ;
21:      end for
22:   end if
23: end if
24:
25: tracePath(Node targetNode, Node currentNode, Stack ascendList, Stack
    pathList)
26: ascendList.push(currentNode) ;
27: if currentNode == targetNode then
28:   List path = new List() ;
29:   path.addAll(ascendList) ;
30:   pathList.add(path) ;
31: else
32:   for each parent p in currentNode.parentList do
33:      if ∃ p.pathMarkingMap.get(currentNode) then
34:         pathMarkValue=p.pathMarkingMap.get(currentNode) ;
35:         for each markingKey in currentNode.pathMaringMap.keySet() do
36:            int markingValue = currentNode.pathMarkingMap.get(markingKey) ;
37:            if markingValue + 1 == pathMarkValue then
38:               currentNode.pathMarkingMap.remove(markingKey) ;
39:               tracePath(targetNode, p, ascendList, pathList) ;
40:               currentNode.pathMarkingMap.put(markingKey, markingValue) ;
41:               break ;
42:            end if
43:         end for
44:      end if
45:   end for
46: end if
47: ascendList.pop();
```

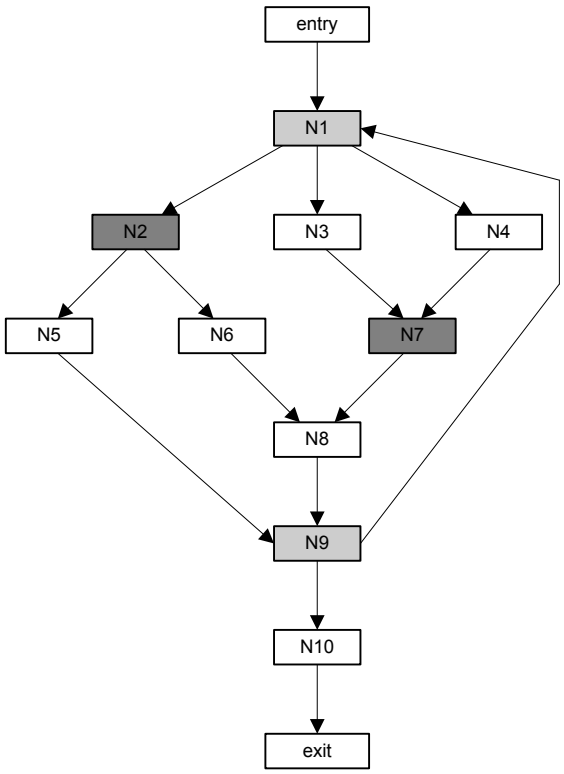

Figure 1. *Graphe Illustrant le GAFlow et GDFlow de N2 et N7*

4.1.1. *Point de Coupure* GAFLow

De manière à calculer le *GAFlow*, nous avons conçu un mécanisme utilisant les algorithmes pour le dominateur. Nous calculons d'abord l'ensemble des dominateurs communs de tous les noeuds en paramètres de *GAFlow*. Ensuite, nous retirons les noeuds en paramètres de l'ensemble calculé. Le dernier noeud dans cet ensemble, retourné par l'Algorithme 2, sera l'ancêtre guaranti le plus près.

Nous avons implanté l'Algorithme 2 en utilisant l'Algorithme 1 pour calculer le *GAFlow*. À l'aide de cette implantation, nous avons déterminé le *GAFlow* de l'exemple de la Figure 1, avec les résultats représentés dans le Tableau 2.

4.1.2. *Point de Coupure* GDFLow

Le descendant garanti le plus près est déterminé à l'aide d'un mécanisme utilisant l'algorithme de calcul des post-dominateurs. Nous calculons d'abord l'ensemble des

Algorithm 2 Algorithme Déterminant le *GAFlow* à l'Aide du Dominateur

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
1: `GAFlow`(NodeSet SelectedNodes) :
2: $CommonDomSet \leftarrow \emptyset$
3: **for all** $node \in SelectedNodes$ **do**
4: $\quad CommonDomSet \leftarrow CommonDomSet \cup (DOM(node) - node)$
5: **end for**
6: **return** $GetLastNode(CommonDomSet)$

Selected Nodes	**Common Dominator Set**	***GAFlow***
N2, N7	entry, N1	N1
N5, N6	entry, N1, N2	N2
N4, N6, N10	entry, N1	N1
N8, N9	entry, N1	N1

Selected Nodes	N4, N6, N10
DOM(N4)	entry, N1
DOM(N6)	entry, N1, N2
DOM(10)	entry, N1, N9
CommonDominatorSet (N4, N6, N10)	entry, N1
GAFLow	N1

Tableau 2. *Résultants d'Exécution de l'Algorithme 2 on Figure 1*

post-dominateurs communs de tous les noeuds en paramètres de *GDFlow*. Ensuite, nous retirons les noeuds en paramètres de l'ensemble calculé. Le premier noeud dans cet ensemble, retourné par l'Algorithme 3, sera le descendant garanti le plus près.

Algorithm 3 Algorithme Déterminant le *GDFlow* à l'Aide du Post-Pominateur

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
1: `GDFlow`(NodeSet SelectedNodes) :
2: $CommonPostDomSet \leftarrow \emptyset$
3: **for all** $node \in SelectedNodes$ **do**
4: $\quad CommonPostDomSet \leftarrow CommonPostDomSet \cup (DOM(node) - node)$
5: **end for**
6: **return** $GetFirstNode(CommonPostDomSet)$

Similairement à l'Algorithme 2, nous avons implanté pour calculer le *GDFlow* l'Algorithme 3 en inversant la direction des flèches du graphe de flots de contrôle, en débutant par le noeud de terminaison et en utilisant l'Algorithme 1 (Holloway *et al.*, 1998a). Les résultats de cette implantation de *GDFlow* sur le graphe illustré dans la Figure 1 sont présentés au Tableau 3.

Selected Nodes	**Common Post-Dominator Set**	***GDFlow***
N2, N7	N9, N10, exit	N9
N4, N5, N6	N9, N10, exit	N9
N6, N7	N8, N9, N10, exit	N8
N8, N9	N10, exit	N10

SelectedNodes	N4, N5, N6
PDOM(N4)	N7, N8, N9, N10, exit
PDOM(N5)	N9, N10, exit
PDOM(N6)	N8, N9, N10, exit
CommonPostDominatorSet (N4, N5, N6)	N9, N10, exit
GDFLow	N9

Tableau 3. *Résultats de l'Exécution de l'Algorithme 3 sur la Figure 1*

4.2. GAFlow *et* GDFlow *Utilisant un Graphe Étiqueté*

Pour une solution alternative pour la détermination de *GAFlow* et *GDFlow*, nous avons aussi choisi d'adapter un algorithme d'étiquetage de graphe. Cet algorithme fut développé par nos collègues et nous l'avons légèrement modifié pour nos besoins. L'Algorithme 4 explique cette méthode.

Chaque noeud dans la hiérarchie est étiqueté d'une manière semblable à une table des matières d'un livre (p.ex. 1., 1.1., 1.2., 1.2.1., ...), comme nous l'avons montré dans l'Algorithme 4, où l'opérateur $+_c$ représente une concaténation de chaînes de caractères avec conversion de type implicite des opérandes. Pour l'appliquer, nous avons exécuté l'Algorithme 4 sur le noeud de départ avec l'étiquette "0.", ce qui numérote récursivement tous les noeuds.

Nous avons implanté l'Algorithme 4 et l'avons testé sur un CFG hypothétique. Veuillez référer à la Figure 2 pour la visualisation des résultats. Ce CFG nous servira d'exemple pour le reste de cet article.

4.2.1. *Point de Coupure* GAFlow

Afin de calculer le *GAFlow*, nous avons développé un mécanisme 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 *GAFlow*. Nous nous assurons que le *GAFlow* est un noeud à travers duquel tous les chemins rejoignent les noeuds de l'ensemble d'entrée en prenant compte de toutes les étiquettes des noeuds. Le tout est élaboré dans l'Algorithme 5.

De plus, nous avons implanté l'Algorithme 5 et nous l'avons appliqué au graphe de la Figure 2. Nous avons choisi, en étude de cas, quelques noeuds du graphe et l'avons soumis à l'algorithme de *GAFlow*. Nos résultats sont présentés dans le Tableau 4 et la Figure 3.

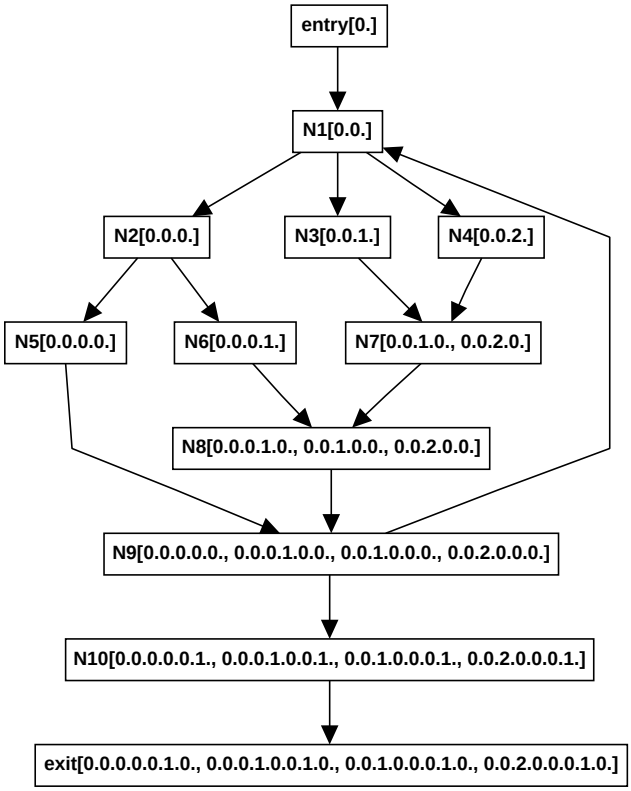

Figure 2. *Graphe Étiqueté*

Selected Nodes	***GAFlow***
N2, N7	N1
N5, N6	N2
N4, N6, N10	N1
N8, N9	N1

Tableau 4. *Résultats d'Exécution de l'Algorithme 5 sur la Figure 2*

4.2.2. *Point de Coupure* GDFlow

Le même mécanisme d'inversion du graphe (Holloway *et al.*, 1998a) utilisé pour calculer le post-dominateur peut également être utilisé pour calculer le *GDFLow* avec un graphe étiqueté. Une fois la direction des flèches inversée, le graphe peut donc être étiqueté, et l'Algorithme 5 peut être utilisé pour déterminer le *GDFLow*.

Algorithm 4 Algorithme d'Étiquetage Hiérarchique de Graphes

```
1: labelNode(Node s, Label l) :
2: s.labels ← s.labels ∪ {l}
3: NodeSequence children = s.children()
4: for k = 0 to |children| − 1 do
5:     child ← children[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
20:
21: Prefixes(Label l) :
22: LabelSet labels ← ∅
23: Label current ←""
24: for i ← 0 to l.length() do
25:     current.append(l.charAt(i))
26:     if Label1.charAt(i) = '.' then
27:         labels.add(current.clone())
28:     end if
29: end for
```

Selected Nodes	*GDFlow*
N2, N7	N9
N4, N5, N6	N9
N6, N7	N8
N8, N9	N10

Tableau 5. *Résultats d'Exécution de Reverse Edge Direction et l'Algorithme 5 sur la Figure 2*

En utilisant cette approche et la même implantation de l'Algorithme 4 sur le graphe de la Figure 2, nous avons obtenu les résultats présentés dans le Tableau 5 et la Figure 4.

Algorithm 5 Algorithme pour Déterminer le *GAFlow*

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
Require: $Graph$ has all its nodes labeled
1: `gaflow`(NodeSet SelectedNodes) :
2: $LabelSequence\ Labels \leftarrow \emptyset$
3: **for all** $node \in SelectedNodes$ **do**
4: $Labels \leftarrow Labels \cup node.labels()$
5: **end for**
6: **return** $GetNodeByLabel(FindCommonPrefix(Labels))$
7:
8: `FindCommonPrefix` (LabelSequence Labels) :
9: **if** $|Labels| = 0$ **then**
10: **return** error
11: **else if** $|Labels| = 1$ **then**
12: **return** $Labels.removeHead()$
13: **else**
14: $Label\ Label1 \leftarrow Labels.removeHead()$
15: $Label\ Label2 \leftarrow Labels.removeHead()$
16: **if** $|Labels| = 2$ **then**
17: **for** $i \leftarrow 0$ to $min(Label.length(), Label2.length()$ **do**
18: **if** $Label1.charAt(i) \neq Label2.charAt(i)$ **then**
19: **return** $Label1.substring(0, i-1)$
20: **end if**
21: **end for**
22: **return** $Label1.substring(0, min(Label.length(), Label2.length()))$
23: **else**
24: $Label\ PartialSolution \leftarrow FindCommonPrefix(Label1, Label2)$
25: $Labels.append(PartialSolution)$
26: **return** $FindCommonPrefix(Labels)$
27: **end if**
28: **end if**

4.3. *Primitives* ExportParameter *et* ImportParameter

Dans cette section, nous présentons la méthodologie et les algorithmes implantant notre proposition de passage des paramètres, combinés à des résultats expérimentaux.

L'algorithme 7 permet de passer un paramètre entre deux noeuds d'un graphe d'appels insensible au contexte (Grove *et al.*, 2001), pour lequel chaque noeud représente une fonction et chaque flèche représente un site d'appel. Pour s'assurer que le paramètre soit déclaré et initialisé en tout le temps, n'importe quoi le chemin d'exécution pris dans le programme, on a élaboré en dessus de cet algorithme un mécanisme basé sur le *GAFlow*. Ce mécanisme permet d'exporter le paramètre sur tous les chemins possibles allant de l'origine à la destination.

Nous accomplissons cela en exécutant les étapes suivantes : Nous (1) utilisons le CFG pour identifier le *GAFlow* des deux points d'origine (*exportParameter*) et de

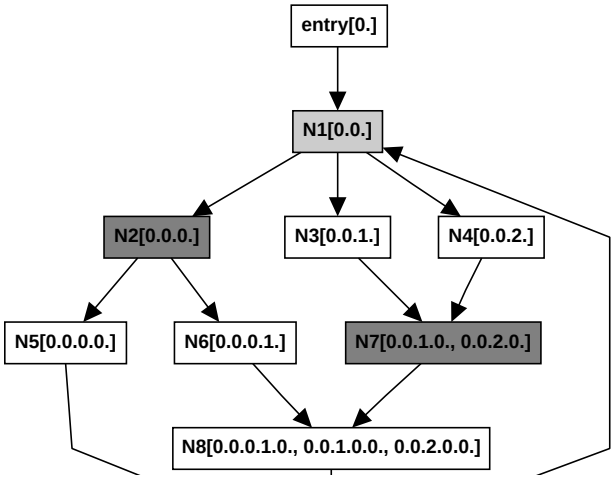

Figure 3. *Illustration du* GAFlow *de N2 and N7*

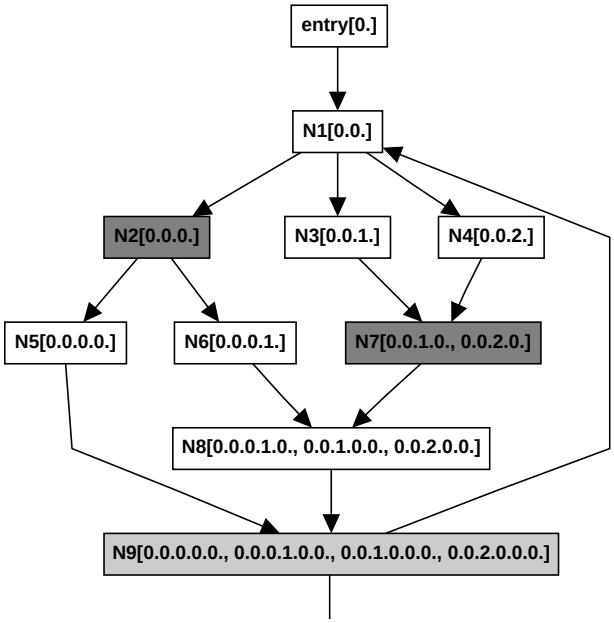

Figure 4. *Illustration de* GDFlow *de N2 and N7*

destination (*importParameter*), (2) localisons les trois noeuds représentant l'origine, la destination et le *GAFlow* dans le graphe d'appels, (3) déclarons et initialisons le

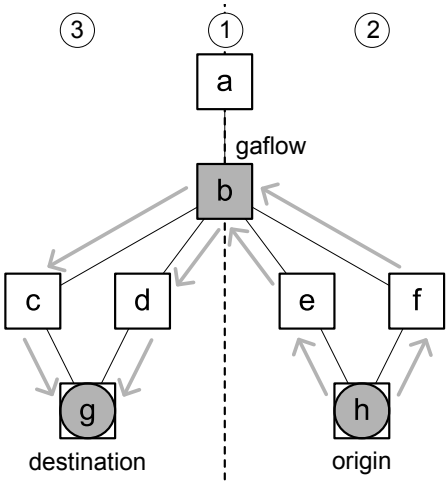

Figure 5. *Passage d'un Paramètre dans un Graphe d'Appels*

paramètre au noeud représentant le *GAFlow* dans le graphe d'appels, (4) exécutons l'Algorithme 7 pour passer le paramètre du noeud d'origine vers le noeud de *GAFlow*, et (5) exécutons de nouveau le même algorithme pour passer le paramètre du noeud de *GAFlow* au noeud de la destination. Cette procédure est décrite dans l'Algorithme 6.

Le *GAFlow* d'un ensemble des points est toujours appelé avant les points eux même (critère de *GAFlow*). En passant le paramètre de l'origine au *GAFlow* et ensuite à la destination, nous nous assurons que le paramètre sera définitivement déclaré et initialisé, même si la destination est appelée avant l'origine. Sinon, le paramètre pourrait être communiqué sans être initialisé, ce qui créerait des erreurs logiciels impactant aussi la conformité de la solution. Toutefois, dans tous les cas de sécurité que nous avons traité, l'origine est toujours appelé avant la destination. Par exemple, dans l'étude de cas pour sécuriser la connexion, les fonctions responsables de connexion sont toujours appelées avant les fonctions responsables d'échanger les informations, sinon il y aura une erreur d'exécution. Cela s'applique dans tous les cas où la destination dépend de l'origine ou dans tous les cas où il y a une séquence d'opérations exécutées pour offrir une fonctionnalité (dans le fond c'est les seuls cas ou on aura besoin de passer des paramètres).

Nous pouvons voir une illustration de l'Algorithme 6 dans la Figure 5 avec l'exemple suivant. Nous voulons passer un paramètre de h à g. En premier lieu, nous identifions leur *GAFlow* b. Ensuite, nous passons le paramètre par tous les chemins de h à b, puis de b à g.

Algorithm 6 Algorithme pour Passer un Paramètre entre deux Points de Jointure

1: function `passParameter`(Node origin, Node end, Parameter param) :
2: **if** $origin = destination$ **then**
3: **return** $success$
4: **end if**
5: $start \leftarrow GuaranteedAncestor(origin, end)$
6: $passParamOnBranch(start, origin, param)$
7: $node.addLocalVariable(param)$
8: $passParamOnBranch(start, end, param)$

La méthodologie présentée dans l'algorithme 7 permet de changer les signatures de fonctions et leurs appels de manière à préserver la conformité syntaxique et leur intention (c.à.d. fonctions compileraient et se comporteraient de la même manière qu'avant). On trouve tous les chemins entre les deux noeuds considérés dans le graphe d'appels. Pour chaque chemin, on propage le paramètre de la fonction appelée vers l'appelant, débutant donc par la fin du chemin. Dans d'autres mots, les signatures des fonctions impliquées entre les deux noeuds sont augmentées avec un paramètre `inout`. Tous les appels passent le paramètre tel quel dans le cas des fonctions qui sont dans le chemin de transmission (c.à.d. noeuds *b*, *c*,*d*,*e*,*f*). Pour fins d'optimisation, les appelants sont modifiés une seule fois et les noeuds modifiés sont enregistrés.

Nous avons implanté un simple programme similaire au scénario illustré dans la Figure 5. Ce programme est illustré dans le Listing 9. Il s'agit d'un client qui envoie une requête et reçoit une réponse du serveur. Nous avons simulé l'exécution des algorithmes de primitives proposées et appliqué l'aspect du Listing 8 manuellement, produisant le programme du Listing 10. Nous avons ensuite exécuté ce programme en capturant les échanges de paquets de données, démontrant que la communication était effectivement cryptée. L'implantation pratique et l'intégration de ces primitives dans un outil d'un langage d'aspect est toujours en cours.

Listing 9 – Extrait de Programme à Sécuriser

```
const char * HTTPrequest = "GET␣/␣HTTP/1.1␣\nHost:␣localhost\n\n";

int dosend(int sd, char * buffer, unsigned int bufSize){
  return send(sd, buffer, bufSize, 0);
}

int doreceive(int sd, char * buffer, unsigned int bufSize){
    return recv(sd, buffer, bufSize, 0);
}

int doConnect(int sd, struct sockaddr_in servAddr){
    return connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));
}

int main (int argc, char *argv[]) {
    /* ...   */
      /* create socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* connect to server */
```

Algorithm 7 Algorithme Passant un Paramètre entre deux Noeuds dans un Graphe d'Appels

```
1: function passParamOnBranch(Node origin, Node destination, Parameter param) :
2: if origin = destination then
3:    return success
4: end if
5: paths ← findPathsBetween(origin, destination)
6: for all path ∈ paths do
7:    path.remove(origin)
8:    while ¬path.isEmpty() do
9:       currnode ← path.tail()
10:      path.remove(currnode)
11:      if ¬node.signature.isModified() ∧
         ¬∃parameter ∈ currnode.signature() : parameter = param then
12:         node.signature.addParameter(param)
13:         node.signature.markModified()
14:         modifyFunctionsCallsTo
            (currNode, param)
15:      end if
16:   end while
17: end for
18: return success
19:
20: function modifyFunctionsCallsTo(Node currnode, Parameter param) :
21: for all caller ∈ currnode.getCallers() do
22:    for all call ∈ caller.getCallsTo(node) : ¬call.modified do
23:       call.parameters.add(param)
24:       call.modified = true
25:    end for
26: end for
```

```
    rc= doConnect(sd, servAddr);

    /*send/receive*/
    rc = dosend(sd);
    fprintf(stderr,"Sent␣%u␣characters:\n%s\n", rc, HTTPrequest);
    memset((void *)buf, 0, MAX_MSG);
    rc=doreceive(sd, buf, MAX_MSG);
    fprintf(stderr,"Received␣%u␣characters:\n%s", rc, buf);

    /* Shutdown */
    close(sd);

    /* ...   */
}
```

Listing 10 – Programme Sécurisé Résultant

```
const char * HTTPrequest = "GET␣/␣HTTP/1.1␣\nHost:␣localhost\n\n";

int dosend(int sd, char * buffer, unsigned int bufSize, gnutls_session_t *
    session){
  if (session != NULL) return gnutls_record_send(*session, buffer, bufSize);
  else return send(sd, buffer, bufSize, 0);
}

int doreceive(int sd, char * buffer, unsigned int bufSize, gnutls_session_t
    * session){
  if (session != NULL) return gnutls_record_recv(*session, buffer, bufSize);
  else return recv(sd, buffer, bufSize, 0);
}

int doConnect(int sd, struct sockaddr_in servAddr, gnutls_session_t *
    session, gnutls_certificate_credentials_t * xcred){

  static const int cert_type_priority[3] = { GNUTLS_CRT_X509,
      GNUTLS_CRT_OPENPGP, 0};
  int rc;
  gnutls_init (session, GNUTLS_CLIENT);
  gnutls_set_default_priority (*session);
  gnutls_certificate_type_set_priority (*session, cert_type_priority);
  gnutls_certificate_allocate_credentials (xcred);
  gnutls_credentials_set (*session, GNUTLS_CRD_CERTIFICATE, *xcred);

  rc = connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));
  if (rc >= 0){
    gnutls_transport_set_ptr (*session, (gnutls_transport_ptr) sd);
    rc = gnutls_handshake (*session);
  }
  return rc;
}


int main (int argc, char *argv[]) {
    gnutls_global_init ();

    /* ... */

      /* create socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if(sd<0) {
      perror("cannot␣open␣socket");
      exit(1);
    }

    doConnect(sd, servAddr,&session,&xcred);
    dosend(sd,HTTPrequest,strlen(HTTPrequest) + 1,&session);
    fprintf(stderr,"Sent␣%u␣characters:\n%s\n", rc, HTTPrequest);
    memset((void *)buf, 0, MAX_MSG);
    doreceive(sd, buf, MAX_MSG,&session);
    fprintf(stderr,"Received␣%u␣characters:\n%s", rc, buf);

    /* Shutdown */
    close(sd);
    gnutls_bye(session, GNUTLS_SHUT_RDWR);
    gnutls_deinit(session);
    gnutls_certificate_free_credentials(xcred);
    gnutls_global_deinit();

    return 0;

}
```

5. État de l'Art

Plusieurs limitations de programmation par aspect pour les préoccupations de sécurité ont été documentées, et plusieurs améliorations furent suggérées. Nous résumons ici ces contributions. De plus, nous présentons les travaux connexes par rapport à notre approche, ainsi qu'aux algorithmes proposés. Toutefois, ces propositions adressent d'autres problèmes de sécurité.

Un point de coupure pour la sécurité, basé sur les graphes de flots de données et utilisé pour identifier les points de jointure selon l'origine des valeurs, est défini et formulé dans (Masuhara *et al.*, 2003). Les auteurs expriment l'utilité de leur point de coupure en présentant un exemple montrant son applicabilité pour détecter des failles d'intégrité dans une application web. Par exemple, ce point de coupure permet de découvrir si une donnée envoyée sur le réseau provient d'information lue d'un fichier confidentiel.

Harbulot et Gurd proposent dans (Harbulot *et al.*, 2005) un modèle de point de coupure sur les boucles. Ils montrent le besoin d'un point de jointure qui prédit si une boucle se répétera indéfiniment. Ce point de coupure détecte les boucles infinies utilisées par les attaquants afin de causer des dénis de service.

Dans (Bonér, 2005), 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. 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 (Kiczales, 2003). 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 paramètre du point de jointure.

Des points de coupure pour les accès aux variables locales furent suggérées par (Hadidi *et al.*, 2006) afin d'améliorer l'utilité de programmation par aspect pour la sécurité. Ils permettent d'accéder aux valeurs des variables locales à l'intérieur d'une méthode. Cette approche permettrait de protéger la confidentialité et l'intégrité de valeurs critiques. Leur idée est basée sur l'approche de Myers (Myers, 1999) qui décrit une extension à Java nommée JFlow. Ce langage vérifie statiquement des annotations de flots d'information dans un program, et offre un modèle d'étiquetage décentralisé, polymorphisme d'étiquettes, vérification des étiquettes durant l'exécution et inférence automatique d'étiquettes. JFlow supporte également les objets, les hiérarchies d'objets, les tests de type dynamique, le contrôle d'accès et les exceptions.

Aberg et al. a présenté dans (Aberg *et al.*, 2003) un système d'aspect qui adresse les notifications d'événement entrecroisantes dispersées sur le code de linux pour soutenir Bossa, un cadre à base d'événement pour le développement d'un programmateur de tâche. Ce système d'aspect utilise la logique temporelle pour décrire avec précision

des points d'insertion de code et des séquences d'instructions qui exigent des événements à insérer. Dans chaque cas, le choix de l'événement dépend des propriétés d'un ou d'une séquence d'instructions. Ils proposent de guider l'insertion d'événement en utilisant un ensemble de règles, en s'élevant à un aspect, qui décrit les contextes de flots de contrôle dans lesquels chaque événement devrait être produit.

Dans un papier de position (Cottenier *et al.*, 2007), Cottenier et al. a soutenu que les technologies d'"Aspect-Oriented Modeling" ont le potentiel de simplifier le déploiement et la capacité de raisonner d'une catégorie de préoccupations entrecroisantes qui ont été classées dans la littérature comme aspects d'états (stateful aspects). Les aspects d'états déclenchent sur une séquence de points de jointure plutôt que sur un seul point de jointure. Ils ont identifié trois propriétés pour les langages d'"Aspect-Oriented Modeling" qui leur permettent de fournir des solutions plus naturelles pour le problème d'aspects d'états. Ils ont également présenté un exemple d'aspect JAsCo qui capture une séquence d'événements (p.ex. methodA - methodB - methodC) et attache un conseil au dernier événement (c-à-d. methodC).

Quant au monde des graphes, plusieurs algorithmes opérant sur les graphes pour trouver des ancêtres, descendants, chemins, etc. ont été proposés. À notre connaissance, aucun ne permet d'obtenir le *GAFlow* et le *GDFlow* comme nous l'avons défini. Dans (Cooper *et al.*, 2001), les auteurs proposent un algorithme simple et rapide pour calculer l'information de dominance (l'ensemble de domination et de post-domination) de noeuds de graphes de flots de contrôle. Ils ont également évalué les différentes propositions de l'état de l'art. L'implantation d'un de ces algorithmes (classe DominaceInfo) est inclue dans la librairie d'analyse de flots de contrôle MachineSUIF (Holloway *et al.*, 1998a). Elle utilise une librairie pour les graphes de flots de contrôle (Holloway *et al.*, 1998b) et permet l'analyse des dominateurs et l'analyse de boules naturelles. Les algorithmes de points de coupure proposés utilisent l'information de dominance des noeuds considérés pour déterminer leur *GAFlow* et leur *GDFlow*.

D'autres approches utilisent la théorie des treillis et calculent efficacement la limite supérieure minimale (LUB) et la limite inférieure maximale (GLB) (Aït-Kaci *et al.*, 1989). Toutefois, leurs résultants ne garantissent pas que tous les chemins traverseront par ces points, ce qui est une exigence centrale de *GAFlow* et *GDFlow*. De plus, les treillis ne supportent pas la variété d'expression d'un graphe de flots de contrôle, puisque ce dernier peut être un graphe cyclique dirigé.

Ryder (Ryder, 1979) a offert une des premières contributions pour la construction de graphes d'appels insensibles au contexte dans les langages procéduraux. Sa contribution fut suivie de la notion de sensibilité au contexte par Callahan et al. (Callahan *et al.*, 1990). Dans le cas de langages orientés objet, Grove et al. (Grove *et al.*, 1997) ont présenté un algorithme de création de graphes sensibles au contexte. L'ensemble des méthodes de construction de graphes a été documenté par Grove et Chambers dans (Grove *et al.*, 2001). Cette étude offre une comparaison empirique des différentes méthodes de constructions, basée sur leur exécution sur des logiciels de grande taille. Les algorithmes de primitives proposés opèrent sur un graphe d'appels.

6. Conclusion

La programmation par aspect semble être un paradigme prometteur pour le renforcement de sécurité des logiciels. Toutefois, cette technologie ne fut pas conçue à la base pour s'occuper de problèmes de sécurité et plusieurs travaux de recherche ont montré les limites dans ce domaine. Similairement, nous avons exploré dans cet article les manques dans les technologies de programmation par aspect pour appliquer plusieurs améliorations de sécurité. Dans ce contexte, nous avons proposé de nouveaux points de coupure et primitives pour les préoccupations de renforcement de sécurité : le *GAFlow*, *GDFlow*, *exportParameter* et *importParameter*. Le *GAFlow* 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 *GDFlow* 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. Les deux primitives passent des paramètres d'un conseil à l'autre à travers le graphe d'appels du programme. Ces points de coupure et primitives sont nécessaires afin d'adresser les problèmes mentionnés ci-haut et réaliser les améliorations de sécurité discutées. À notre connaissance, aucune des propositions actuelle est en mesure de le faire.

En ce qui a trait à notre travail à venir, nous cherchons présentement à implanter et intégrer les points de coupure et les primitives que nous avons proposées dans les tisseurs de programmation par aspect actuels. De plus, nous adressons d'autres limitations de programmation par aspect pour la sécurité tout en proposant des solutions.

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 (Défense Nationale) / CRSNG (Conseil de Recherches en Sciences Naturelles et en Génie du Canada).

7. Bibliographie

Aberg R. A., Lawall J. L., Sudholt M., Muller G., Meury A.-F. L., « On the automatic evolution of an OS kernel using temporal logic and AOP », *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE03)*, IEEE Press, 2003.

Aït-Kaci H., Boyer R., Lincoln P., Nasr R., « Efficient Implementation of Lattice Operations », *ACM Trans. Program. Lang. Syst.*, vol. 11, n° 1, p. 115-146, 1989.

Bishop M., « How Attackers Break Programs, and How to Write More Secure Programs », 2005. `http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002/index.html` (accessed 2007/04/19).

Bodkin R., « Enterprise Security Aspects », *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04 :AOSDSEC)*, 2004.

Bonér J., « Semantics for a Synchronized Block Join Point », 2005. http://jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-join-point/ (accessed 2007/09/26).

Callahan D., Carle A., Hall M. W., Kennedy K., « Constructing the Procedure Call Multigraph », *IEEE Trans. Softw. Eng.*, vol. 16, n° 4, p. 483-487, 1990.

Cooper K., Harvey T., Kennedy K., « A Simple, Fast Dominance Algorithm », *Software Practice and Experience*, 2001.

Cottenier T., van den Berg A., Elrad T., « Stateful Aspects : The Case for Aspect-Oriented Modeling », *Proceedings of the 10th international workshop on Aspect-oriented modeling*, ACM Press, 2007.

DeWin B., Engineering Application Level Security through Aspect Oriented Software Development, PhD thesis, Katholieke Universiteit Leuven, 2004.

Dijkstra E., « Dijkstra's algorithm », n.d. http://en.wikipedia.org/wiki/Dijkstra_algorithm (accessed 2007/10/12).

Gomez E., « CS624- Notes on Control Flow Graph », 2003. http://www.csci.csusb.edu/egomez/cs624/cfg.pdf.

Grove D., Chambers C., « A framework for Call Graph Construction Algorithms », *ACM Trans. Program. Lang. Syst.*, vol. 23, n° 6, p. 685-746, 2001.

Grove D., DeFouw G., Dean J., Chambers C., « Call Graph Construction in Object-Oriented Languages », *SIGPLAN Not.*, vol. 32, n° 10, p. 108-124, 1997.

Hadidi D., Belblidia N., Debbabi M., « Security Crosscutting Concerns and AspectJ », *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*, McGraw-Hill/ACM Press, 2006.

Harbulot B., Gurd J. R., « A Join Point for Loops in AspectJ », *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005), March*, 2005.

Holloway G., Smith M. D., « The Machine-SUIF Control Flow Analysis Library. Harvard University », 1998a. http://www.eecs.harvard.edu/machsuif/software/nci/cfa.html (accessed 2007/09/24).

Holloway G., Smith M. D., « The Machine-SUIF Control Flow Graph Library. Harvard University », 1998b. http://www.eecs.harvard.edu/hube/software/nci/cfg.html (accessed 2007/09/24).

Howard M., LeBlanc D. E., *Writing Secure Code*, Microsoft Press, Redmond, WA, USA, 2002.

Huang M., Wang C., Zhang L., « Toward a Reusable and Generic Security Aspect Library », *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04 :AOSDSEC)*, 2004.

Kiczales G., « The Fun has Just Begun, Keynote talk at AOSD 2003 », 2003. http://www.cs.ubc.ca/~gregor/papers/kiczales-aosd-2003.ppt (accessed 2007/04/19).

Masuhara H., Kawauchi K., « Dataflow Pointcut in Aspect-Oriented Programming », *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, p. 105-121, 2003.

Mourad A., Laverdière M.-A., Debbabi M., « Security Hardening of Open Source Software », *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*, McGraw-Hill/ACM Press, 2006.

Mourad A., Laverdière M.-A., Debbabi M., « A High-Level Aspect-Oriented based Language for Software Security Hardening », *Proceedings of the International Conference on Security and Cryptography*, Secrypt, 2007a.

Mourad A., Laverdière M.-A., Debbabi M., « Towards an Aspect Oriented Approach for the Security Hardening of Code », *Proceedings of the 3rd IEEE International Symposium on Security in Networks and Distributed Systems*, IEEE Press, 2007b.

Myers A. C., « JFlow : Practical Mostly-Static Information Flow Control », *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '99)*, ACM Press, New York, NY, USA, p. 228-241, 1999.

Ryder B. G., « Constructing the Call Graph of a Program », *IEEE Transactions on Software Engineering*, vol. 5, n° 3, p. 216- 226, 1979.

Schumacher M., *Security Engineering with Patterns*, Springer, 2003.

Seacord R. C., *Secure Coding in C and C++*, SEI Series, Addison-Wesley, 2005.

Shah V., An Aspect-Oriented Security Assurance Solution, Technical Report n° AFRL-IF-RS-TR-2003-254, Cigital Labs, 2003.

Wheeler D. A., *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010*, 2003. `http://www.dwheeler.com/secure-programs/` (accessed 2007/04/19).