POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Différences de protection entre les versions d'un logiciel : étude de cas des changements, liens causaux et perspectives pour la réparation automatique des défaillances

KARL JULIEN
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Différences de protection entre les versions d'un logiciel : étude de cas des changements, liens causaux et perspectives pour la réparation automatique des défaillances

présenté par **Karl JULIEN**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Giuliano ANTONIOL, président
Ettore MERLO, membre et directeur de recherche
Foutse KHOMH, membre

DÉDICACE

À la mémoire de Victor et Auréline.

REMERCIEMENTS

Dans un premier temps, je souhaite remercier Giuliano Antoniol, Foutse Khomh et Ettore Merlo, qui ont accepté de faire partie du jury du présent mémoire.

Je remercie plus particulièrement Ettore Merlo, superviseur de la maitrise, pour sa confiance et ses précieux conseils ayant établi la direction du projet.

Je remercie également mes camarades de laboratoire, tout particulièrement Marc-André Laverdière pour ses conseils et pour avoir construit les bases de cette maitrise, et Nicolas Cloutier pour son aide au lancement du projet.

Je voulais enfin remercier ma famille, et particulièrement mes parents Hugues et Karine et ma soeur Lou pour leur soutien outre-Atlantique tout au long de mes études. Je remercie également mes parents pour avoir pris le temps de relire ce mémoire pour sa correction orthographique.

RÉSUMÉ

Les applications Web mettent en place des politiques de contrôle d'accès pour protéger leurs utilisateurs en restreignant les accès aux actions et aux données. Lors de l'évolution de l'application, en introduisant des nouvelles fonctionnalités ou en corrigeant des erreurs, les développeurs peuvent accidentellement briser la politique de contrôle d'accès et injecter des vulnérabilités dans l'application.

L'une des techniques applicables pour prévenir et réparer l'injection de ces vulnérabilités est l'utilisation d'analyses statiques afin d'étudier les propriétés du programme, et de comparer les propriétés de deux versions successives. Si les propriétés sont différentes, il est possible qu'une vulnérabilité ait été injectée.

Pour aborder ce problème, nous extrayons donc les propriétés de protection des instructions d'un programme par Analyse de Flux de Traversement de Motif (PTFA), et nous considérons les différences de protection entre deux versions d'un projet comme un ensemble de vulnérabilités à corriger.

Après avoir revu l'antériorité de l'étude du problème de réparation automatique de logiciel, nous étudions les cas de différence de protection afin de proposer deux modèles de réparation de ces différences, en recherchant des patchs dans deux espaces différents.

Le premier consiste à réverser les changements faits entre les deux versions que l'on soupçonne comme responsables des différences de protection. Cette solution s'appuie sur l'étude des Changements Impactant la Protection (PIC), et est précédée d'une étude des limites de ceux-ci. Nous définissons un cadre pour ce modèle et nous démontrons formellement qu'il permet de générer un patch corrigeant les vulnérabilités ciblées. Nous le mettons ensuite en application afin de mesurer ses performances sur trois jeux de données, comportant au total 148 paires de versions issues de deux applications php différentes, WordPress et MediaWiki, et à deux niveaux de granularité de versionnage différents.

Le deuxième consiste à insérer des motifs de sécurité aux emplacements adéquats du programme. Nous définissons un cadre pour ce modèle et nous montrons formellement qu'il permet de générer un patch corrigeant les vulnérabilités ciblées.

ABSTRACT

Web applications need access control policies to restrict the access to data and actions in order to protect their users. During the evolution of the application, while adding functionalities and correcting bugs, developers can unintentionally break the access control policy and introduce vulnerabilities into the application.

A possible approach to prevent and repair the introduction of these vulnerabilities is the application of static analysis in order to extract the properties of the application, and to compare the properties between two successive versions. When the properties are different, a vulnerability has possibly been introduced.

To study this problem, we extract protection properties of the statement of the program with a Pattern Traversal Flow Analysis (PTFA), and we take the protection differences between two versions as a set of vulnerabilities to correct.

After having reviewed the state-of-the-art of software automatic repair and patch generation, we study the protections difference in order to introduce two models to repair those differences, using two distinct research spaces.

The first model uses a partial reversion of the changes suspected to be the cause of the protection differences. This approach is based on the *Protection Impacting Changes* (PIC) analysis, and is deduced from a study of the limits of this analysis. We define a scope for this model and we prove that it successfully produces patch correcting the targetted vulnerabilities. In order to measure its performance, we apply this model on three data sets, adding up to 148 version pairs, from two distinct php applications, WordPress and MediaWiki, and two granularity levels of versioning.

The second model uses the insertion of security patterns at the appropriate places. We define a scope for this model, and we prove that it successfully produces patch correcting the targetted vulnerabilities.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AST	Arbre de Syntaxe Abstraite (Abstract Syntax Tree)
CFG	Graphe de Flux de Contrôle (Control Flow Graph)
CVE	Vulnérabilités et Expositions Communes (Common Vulnerabilities and Exposures)
DAC	Contrôle d'Accès Discrétionnaire (Discretionary Access Control)
DPD	Différence de Protection Définitive (Definite Protection Difference)
LOC	Lignes De Code (Lines Of Code)
MAC	Contrôle d'Accès Obligatoire (Mandatory Access Control)
PIC	Changements Impactant la Protection (Protection Impacting Changes)
PTFA	Analyse de Flux par Traversement de Motifs (Pattern Traversal Flow Analysis)
RBAC	Contrôle d'Accès Basé sur les Rôles (Role Based Access Control)
SSA	Forme Statique à Assignation Unique (Single Static Assignement)
XSS	Script Inter-Sites (Cross-Site Scripting)

CHAPITRE 1 INTRODUCTION

Les facteurs humains du développement et de la maintenance de code informatique engendrent des erreurs [2]. La présence de tels bugs dans un programme provoque un coût important de détection et de réparation des erreurs. En moyenne, entre 50 et 75 % du temps de développement est consacré à cette partie du développement logiciel [3]. Les applications Web évoluant dans un environnement hostile, la présence de vulnérabilités non détectées ont des conséquences particulièrement importantes pour la sécurité des utilisateurs, et pour la fiabilité de l'application.

La violation de contrôle d'accès est reconnue comme l'un des problèmes de sécurité causant les vulnérabilités les plus préoccupantes, comme le montre sa présence dans le TOP 10 d'OWASP (Open Web Application Security Project) des risques de sécurité les plus critiques pour les applications Web [4]. Des conséquences de telles vulnérabilités sont l'accès à des informations sensibles, comme avec l'escalade de privilège détectée dans Kubernetes en Décembre 2018 [5], ou encore l'accès et la modification de données d'un autre utilisateur.

Les causes d'erreurs et de vulnérabilités dans les applications Web étant nombreuses et diversifiées [6], leur résolution exhaustive est un problème complexe. La formalisation des erreurs est souvent difficile et propre à un type d'erreur.

Néanmoins, le coût et l'imperfection de la résolution manuelle des erreurs motivent la recherche de méthodes automatiques de détection et de réparation de ces vulnérabilités.

1.1 Définitions et concepts de base

1.1.1 Modèle de contrôle d'accès

Un modèle de contrôle d'accès permet de définir une politique limitant l'accès à des données ou des actions dans une application Web.

Différents modèles de contrôle d'accès existent selon le besoin de l'application.

Les trois familles classiques de modèles de contrôle d'accès [7, 8] sont le Contrôle d'Accès Discrétionnaire (DAC), le Contrôle d'Accès Obligatoire (MAC) et le Contrôle d'Accès Basé sur les Rôles (RBAC). Ces trois familles ne sont pas disjointes.

Le Contrôle d'Accès Discrétionnaire est basé sur des autorisations données à des utilisateurs ou des groupes sur des objets. Les autorisations peuvent être données par les propriétaires des fichiers. La limite de ce modèle est qu'il permet un transfert indirect d'autorisations entre

les utilisateurs (un utilisateur ayant une autorisation pour lire un document peut le partager avec un autre ne l'ayant pas).

Le Contrôle d'Accès Obligatoire est un modèle centralisé, où les utilisateurs, les actions et les objets sont associés à un niveau de sécurité. Les utilisateurs n'ont donc pas accès à des actions des objets d'un niveau supérieur. Un niveau de sécurité élevé (administrateur...) témoigne d'un niveau de confiance élevé en l'utilisateur, et donc la possibilité de lui partager plus de données ou d'actions.

Un modèle de Contrôle d'Accès Basé sur les Rôles (RBAC) [9] est un modèle permettant de limiter l'accès à des données ou des actions selon des rôles attribués à des utilisateurs. La plupart des applications Web (WordPress, MediaWiki, Moodle...) reposent sur un système de ce type.

Formellement, un modèle $RBAC_0$ standard est composé de 4 ensembles U (utilisateur), R (rôle), P (permission) et S (session), de deux relations $PA \in P \times R$ (assignation de privilèges) et $UA \in U \times R$ (assignation de rôles), et de deux fonctions définissant les sessions, $user : S \rightarrow U$ et $roles : S \rightarrow \mathcal{P}(R)$. Il existe d'autres modèles, avec des contraintes supplémentaires entre les ensembles. Chaque utilisateur a donc un ensemble de rôles, et chaque rôle est associé à un ensemble de privilèges. Un exemple de modèle de politique RBAC est présenté en Figure 1.1.

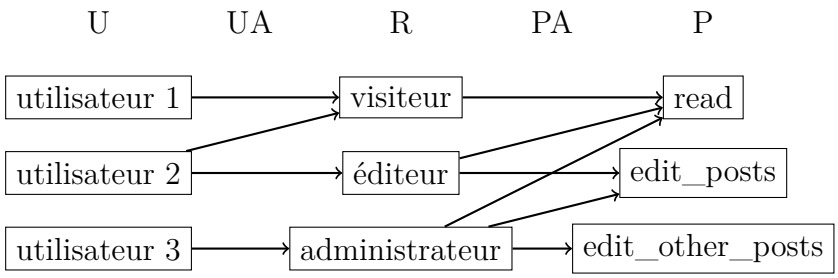


Figure 1.1 Exemple d'un modèle RBAC

1.1.2 Représentations de programmes

Afin d'extraire des informations d'un programme ou d'y apporter des réparations, il est nécessaire de le représenter sous un format permettant l'application d'analyses.

La représentation la plus générale et la plus directe d'un programme est l'Arbre de Syntaxe Abstraite (AST), produit de l'analyse syntaxique. Il s'agit d'une représentation sous forme d'arbre de la syntaxe du programme, sans considération de sa signification et du comportement attendu du programme. Cette forme est obtenue en parsant le texte constituant le

programme.

L'AST d'un programme peut ensuite être transformé en des formats prenant en considération la sémantique du programme.

Une autre représentation d'un programme est le Graphe de Flux de Contrôle (CFG) [10]. Le CFG permet d'appliquer des analyses de flux de contrôle au programme. Celui-ci est défini comme un graphe orienté (V, E) dont les noeuds de l'ensemble V sont des séquences linéaires d'instructions, et dont les arêtes de l'ensemble E représentent les contrôles de flux.

Dans les diverses représentations intermédiaires utilisées dans nos analyses, par simplification, les graphes sont représentés avec une unique instruction par noeud.

```
1          function foo(){
2          $a=0;
3          if (condition){
4                  $a=1;
5          }
6          }
```

Figure 1.2 Exemple de programme dans le langage php

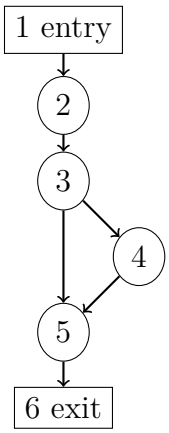
Une représentation Single Static Assignement (SSA) [11] est également fréquemment utilisée comme représentation intermédiaire d'un code. Cette représentation permet d'appliquer des analyses de flux de données au programme. Celle-ci consiste à ajouter des variables intermédiaires de sorte à assigner chaque variable une unique fois. Ainsi, la valeur d'une variable est entièrement déterminée par l'instruction l'assignant.

Pour déterminer la variable SSA correspondant à la valeur courante d'une variable dans une référence à celle-ci, il est nécessaire de définir une instruction ϕ utilisée aux jonctions d'arcs pour résoudre les ambiguïtés sur la variable référée. Pour un noeud ayant deux arcs entrants, l'instruction $v3 = \phi(v1, v2)$ signifie que $v3$ prend la valeur $v1$ ou $v2$ selon l'arc que l'exécution emprunte en amont.

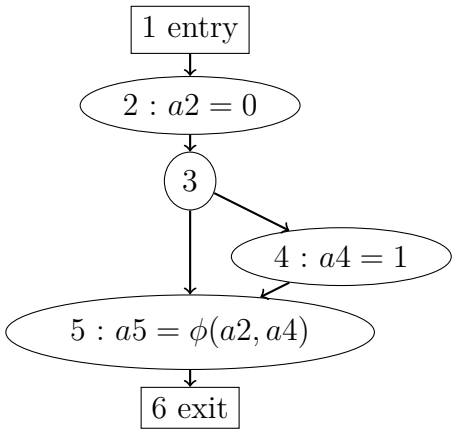
Certaines analyses nécessitent également une représentation inter-procédurale du code pour être effectuée.

Plusieurs modèles prenant en compte cet aspect existent. Ceux-ci se distinguent entre eux par leur sensibilité au contexte d'appel. Un modèle sensible au contexte distingue les graphes des fonctions selon leur contexte d'appel, et un modèle insensible au contexte.

Les analyses sur un modèle insensible au contexte peut explorer des chemins incohérents, puisque les appels et les retours de fonction ne sont pas reliés entre eux, et les contextes



(a) CFG correspondant à la Figure 1.2



(b) Graphe SSA correspondant à la Figure 1.2

Figure 1.3 Représentations du code en Figure 1.2

d'appels ne sont pas discernés.

Un exemple de modèle prenant en compte les transitions inter-procédurales est le cadre de l'analyse de Sous-ensemble Inter-procédural Fini Distribué (IFDS) [12]. Celui-ci relie les graphes de flux de données des procédures sur les arêtes d'appel, en reliant le noeud d'appel au début de la fonction appelante, la fin de la fonction avec le noeud de retour dans la fonction appelante, et le noeud d'appel au noeud de retour. Les chemins valides inter-procéduraux sont utilisés afin de permettre une sensibilité au contexte des analyses tout en ne développant pas le graphe des procédures pour chacun de leurs noeuds d'appel. Les chemins valides inter-procéduraux désignent les chemins dont les appels et les retours de fonction s'alternent de manière cohérente, de façon à ce que la sortie d'une procédure se fasse bien sur le noeud de retour associé au noeud ayant appelé cette procédure.

1.1.3 Analyse *Pattern Traversal Flow Analysis*

Comme vu dans la section 1.1.2, les différentes représentations d'un programme mettent en avant différents aspects de ce programme, permettant l'application d'analyses différentes. Le modèle d'Analyse de Flux par Traversement de Motif (PTFA) d'un programme est ainsi une représentation d'un programme mettant en lumière la vérification ou non d'un prédicat lors de l'exécution d'une instruction, et permettant les analyses relatives à ce prédicat.

Le modèle se présente comme un graphe à 4 colonnes [13], chacune représentant une copie des noeuds du CFG du programme avec des propriétés de protection différentes. Les noeuds du modèle sont donc notés $q_{i,j,k}$, avec i l'identifiant du noeud du CFG, j le booleen représentant la vérification du prédicat lors de l'appel du méthode, et k le booléen représentant la vérification

du prédicat lors de l'exécution du noeud i. Une valeur de 1 pour k signifie ainsi que le prédicat a été vérifié et est vrai, et une valeur de 0 signifie que le prédicat a été vérifié et est faux, ou que la valeur du prédicat est inconnue. Les arêtes du nouveau graphe représentent les arêtes du CFG, et la colonne qu'elles ciblent est déterminée par la colonne de la source et de l'impact de la transition sur le niveau de protection. Les changements de colonnes se font donc lors des octroiements, des révocations et des vérifications de privilèges, ainsi que des appels de méthodes. En plus de ces états, un état initial q_0 est ajouté au modèle, et a des transitions vers tous les états $q_{v,0,0}$ avec v un noeud de départ du CFG.

Formellement, le modèle PTFA [14] est un automate définit comme un tuple $(Q_{\mathcal{A}}, T_{\mathcal{A}}, q_0, V_{\mathcal{A}}, G_{\mathcal{A}}, A_{\mathcal{A}})$, où $Q_{\mathcal{A}}$ est l'ensemble des états tels que décrits précédemment, $T_{\mathcal{A}}$ l'ensemble des transitions possibles entre ces états, $q_0 \in Q_{\mathcal{A}}$ l'état initial, $V_{\mathcal{A}}$ l'ensemble des variables, $G_{\mathcal{A}}$ l'ensemble de conditions de garde se référant aux variables et annotant les transitions, et $A_{\mathcal{A}}$ l'ensemble des assignations de variables annotant également les transitions. Seuls les ensembles $Q_{\mathcal{A}}$ et $T_{\mathcal{A}}$ ainsi que l'état initial q_0 sont exploités dans les analyses s'appuyant sur ce modèle que nous présentons par la suite. Les deux ensembles $Q_{\mathcal{A}}$ et $T_{\mathcal{A}}$ correspondant au modèle PTFA du prédicat *current_user_can*("p") du programme exposé en Figure 1.4 ayant comme seul noeud initial l'entrée du fichier A.php. Ils sont représentés par un graphe à quatre colonnes sur la Figure 1.5.

A.php

```
1          $a=0;
2          if (current_user_can("p")){
3          $a=1;
4          fee();
5          }
6          $b=1
```

B.php

```
1          function fee(){
2          $a=0;
3          }
```

Figure 1.4 Exemple d'un programme contenant des instructions de protection

Le modèle PTFA différencie les contextes d'appels dans un état protégé ou non protégé par un privilège, et permet donc de réaliser des analyses de protection restant continues sur les arcs inter-procéduraux.

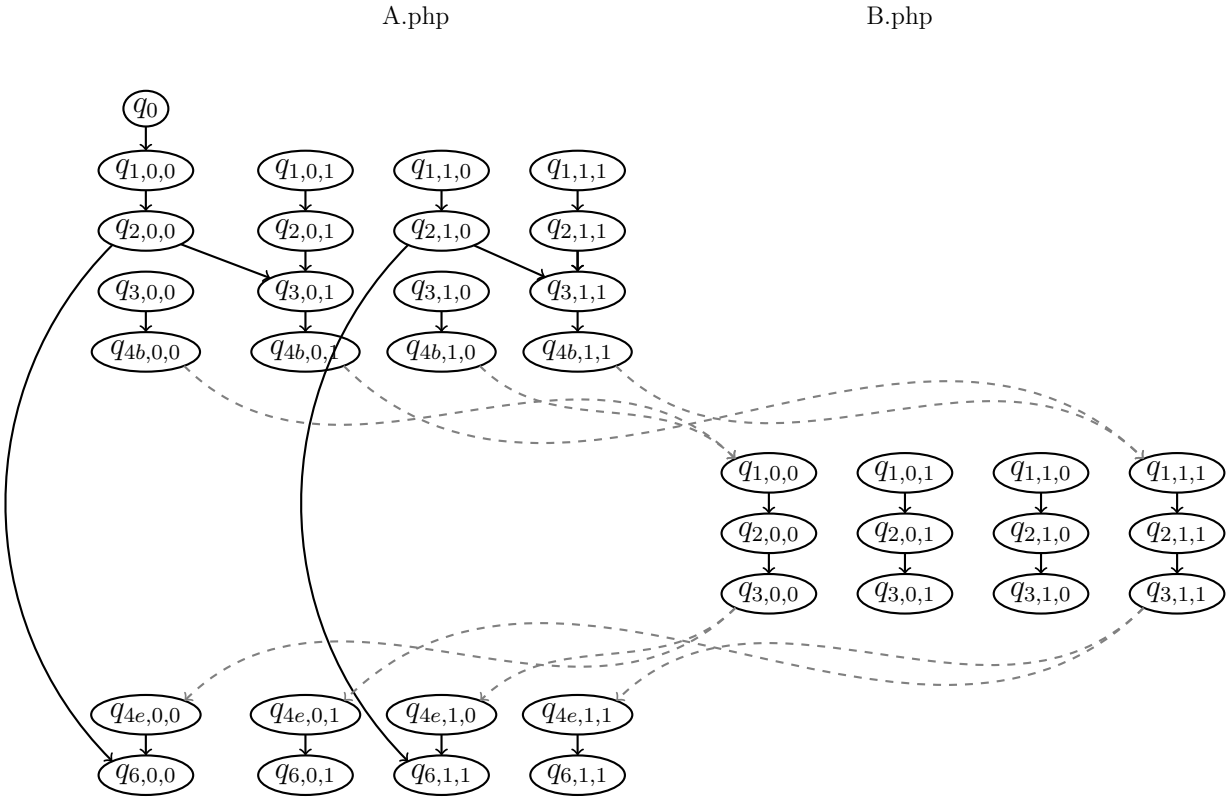


Figure 1.5 Modèle PTFA correspondant au programme de la Figure 1.4

1.1.4 Analyse d'accessibilité et protection définitive

Soit la version A d'un programme, son CFG (V, E), et l'automate PTFA $\mathcal{A}$ à ce programme au regard d'un privilège $priv$.

On appelle $V'_{\mathcal{A}}$ et $T'_{\mathcal{A}}$ les sous-ensembles des états et des transitions d'un modèle PTFA accessibles depuis q_0. Un noeud $v \in V$ du CFG est dit définitivement protégé s'il n'existe pas de chemin non protégé allant de l'état initial v_0 vers $q_{v,0,0}$ ou $q_{v,1,0}$, donc si les états $q_{v,0,0}$ et $q_{v,1,0}$ ne sont pas accessibles, les états accessibles étant définis par les chemins y accédant [15, 16] (voir Equation 1.1).

$$
\begin{aligned}
& defProt(v) \\
& \equiv \\
& \forall p = (v_0, \cdots, v) \in paths(CFG), prot(p) \\
& \equiv \\
& \neg\exists p = (v_0, \cdots, v) \in paths(CFG), \neg prot(p) \\
& \equiv \\
& q_{v,0,0} \notin V'_{\mathcal{A}} \wedge q_{v,0,1} \notin V'_{\mathcal{A}}
\end{aligned}
\tag{1.1}
$$

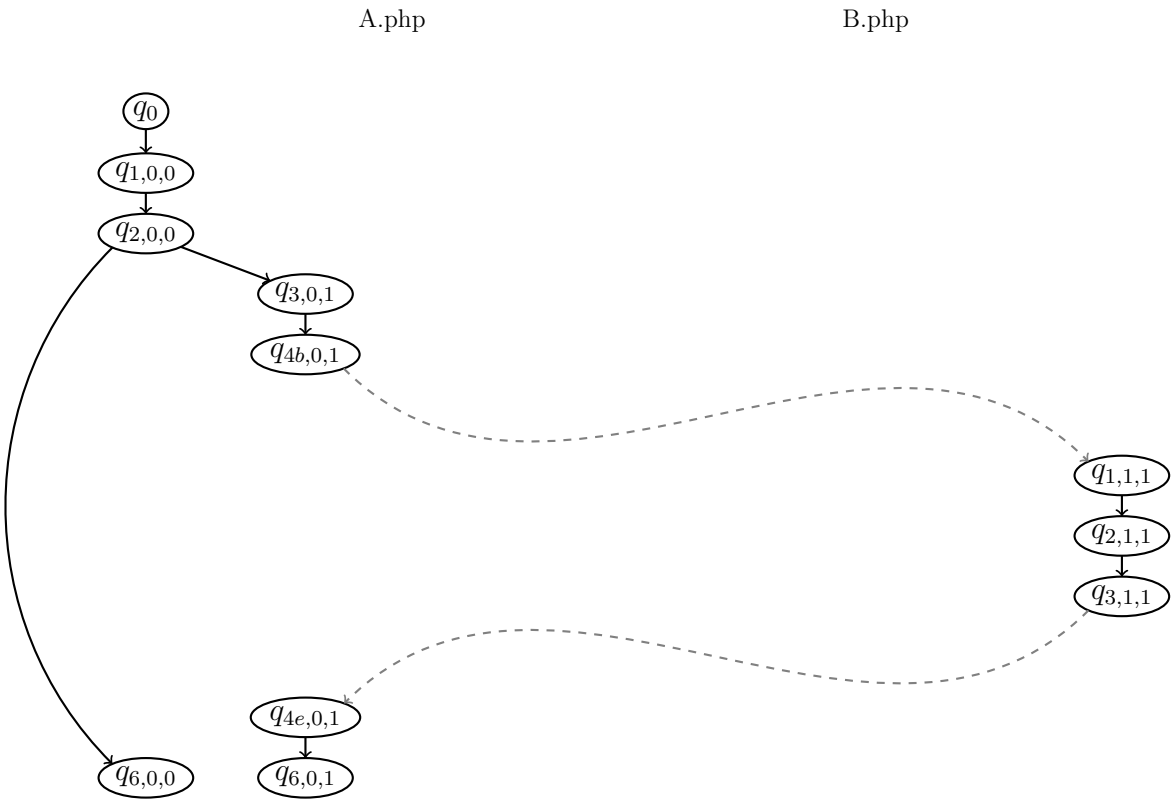


Figure 1.6 Partie du Modèle PTFA correspondant au programme de la Figure 1.4 accessible depuis le noeud d'entrée de A.php

La Figure 1.6 représente les parties accessibles $V'_{\mathcal{A}}$ et $T'_{\mathcal{A}}$ du modèle PTFA présenté en Figure 1.5 depuis le noeud d'entrée de la méthode foo. Les noeuds définitivement protégés sont ceux pour lesquels les états non protégés ne sont pas atteignables, donc les noeuds 3, 4b et 4e du fichier A.php, et les noeuds 1, 2 et 3 du fichier B.php.

1.1.5 Extraction des *Definite Protection Differences* (DPD)

Après avoir calculé les propriétés de protection des différentes instructions pour chaque prédicat choisi, Laverdière et Merlo [16] comparent ces propriétés entre deux versions successives du programme. Ils déterminent ainsi les protections susceptibles d'être erronées dans la version postérieure.

Pour ce faire, une correspondance est dans un premier temps établie entre les lignes des deux versions. Les lignes que le logiciel de différenciation indique comme ajoutées, supprimées, ou modifiées (ce qui est considéré comme une suppression et un ajout de ligne), ne sont pas mis dans cette correspondance. Une fonction partielle injective $lineMap_{A,B}$ représente cette association entre les lignes de deux versions d'un programme A et B. L'outil diff [17] de GNU est utilisé afin de construire la correspondance entre les lignes. On définit :

$$\begin{aligned} addLin(A,B) &= Lines(B) \setminus range(lineMap_{A,B}) \\ delLin(A,B) &= Lines(A) \setminus domain(lineMap_{A,B}) \end{aligned} \tag{1.2}$$

Où *domain* et *range* retournent le domaine de définition et l'image d'une fonction, et $Lines(V)$ représente l'ensemble des lignes de l'ensemble V. En effet, les lignes ajoutées entre les versions A et B sont les lignes de la version B n'ayant pas d'antécédent dans la version A par la fonction $lineMap_{A,B}$. De même, les lignes supprimées entre les versions A et B sont les lignes de la version A n'ayant pas d'image dans la version B par la fonction $lineMap_{A,B}$. Formellement, une ligne peut être représentée par une chaîne de caractères correspondant au nom du fichier la contenant et un entier correspondant au numéro de la ligne dans ce fichier.

$vertexMap_{A,B}$ est une fonction partielle injective qui représente l'association reliant les noeuds des lignes de $lineMap_{A,B}$ entre les deux versions du programme. Pour une ligne $a \in domain(lineMap_{A,B})$ de la version A et la ligne correspondante de la version B, $b = lineMap_{A,B}(a)$, les noeuds de l'AST contenus dans ces lignes sont numérotés selon leur ordre d'apparition dans l'AST, et reliés entre eux en respectant cet ordre.

Pour un privilège donné, les ensembles de Différences de Protection Définitive (DPD) entre deux versions A et B sont calculés ainsi :

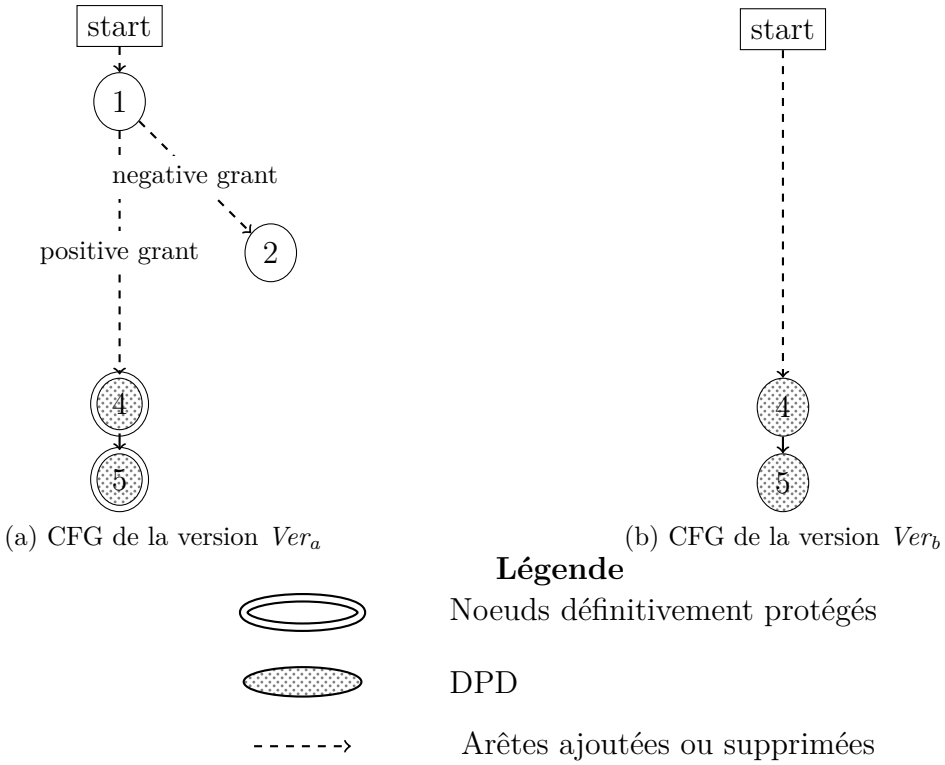
$$\begin{aligned} DefLoss(A,B) &= \{(v_a, v_b) \mid (vertexMap_{A,B}(v_a) = v_b) \wedge defProt(v_a) \wedge \neg defProt(v_b)\} \\ DefGain(A,B) &= \{(v_a, v_b) \mid (vertexMap_{A,B}(v_a) = v_b) \wedge \neg defProt(v_a) \wedge defProt(v_b)\} \\ DPD(A,B) &= DefLoss(A,B) \cup DefGain(A,B) \end{aligned} \tag{1.3}$$

La Figure 1.8 montre les résultats de l'analyse de protection définitive ainsi que les DPD correspondant aux versions présentées en Figure 1.7.

Dans l'implémentation utilisée dans le laboratoire, la correspondance est calculée avec la suite d'outils Diffutils de GNU.

```
1 -       if (!current_user_can('p')){
2 -       die();
3 -       }
4         $b="padding";
5         $c="padding";
```

Figure 1.7 Exemple d'une paire de versions contenant des DPD



(a) CFG de la version Ver_a

(b) CFG de la version Ver_b

Figure 1.8 CFG de la paire de versions en Figure 1.7

Les DPD sont des différences de protection définitive entre les versions, donc des injections potentielles de vulnérabilités lors de la modification du programme. Ces changements peuvent être volontaires, s'il s'agit d'un gain de protection par exemple, ou si la protection définitive n'était pas nécessaire dans la version antérieure. Des vulnérabilités RBAC peuvent également ne pas être des DPD, par exemple si la protection est conditionnelle, ou si le motif de sécurité

n'est pas reconnu par l'analyse PTFA. Par exemple, l'extrait de WordPress présenté en figure 1.9 ne constitue pas une attribution de privilège dans le modèle PTFA associé au privilège *manage_network_users*, puisque les arcs sécurisés ne sont pas tous bloqués. Néanmoins, il s'agit d'un schéma de sécurité dont la suppression aboutirait à une vulnérabilité.

1.1.6 Extraction des *Protection Impacting Changes* (PIC)

Une fois que les DPD sont détectées, l'objectif est d'aider les développeurs à connaitre la source de ceux-ci, pour pouvoir les réparer. L'approche de l'extraction des PIC proposée par Laverdière et Merlo [18] vise ainsi à extraire un sous-ensemble de l'ensemble des changements se rapprochant de l'ensemble des causes racines des DPD (c'est à dire des changements ayant causé les DPD). Nous définissons les PIC comme un sous-ensemble de l'ensemble des changements d'arêtes entre deux graphes PTFA, soit l'union des ensembles d'ajouts et de suppressions d'arêtes définis dans l'Equation 1.4. La sélection des arêtes appartenant à l'ensemble des PIC se fait à partir des ensembles des arêtes accessibles T_a et T_b des modèles PTFA des versions A et B respectivement.

```
1 if ( is_multisite()
2 && ! current_user_can( 'manage_network_users' )
3 && $user_id != $current_user->ID
4 && ! apply_filters( 'enable_edit_any_user_configuration', true )
5 ) {
6 wp_die( __( 'Sorry, you are not allowed to edit this user.' ) );
7 }
```

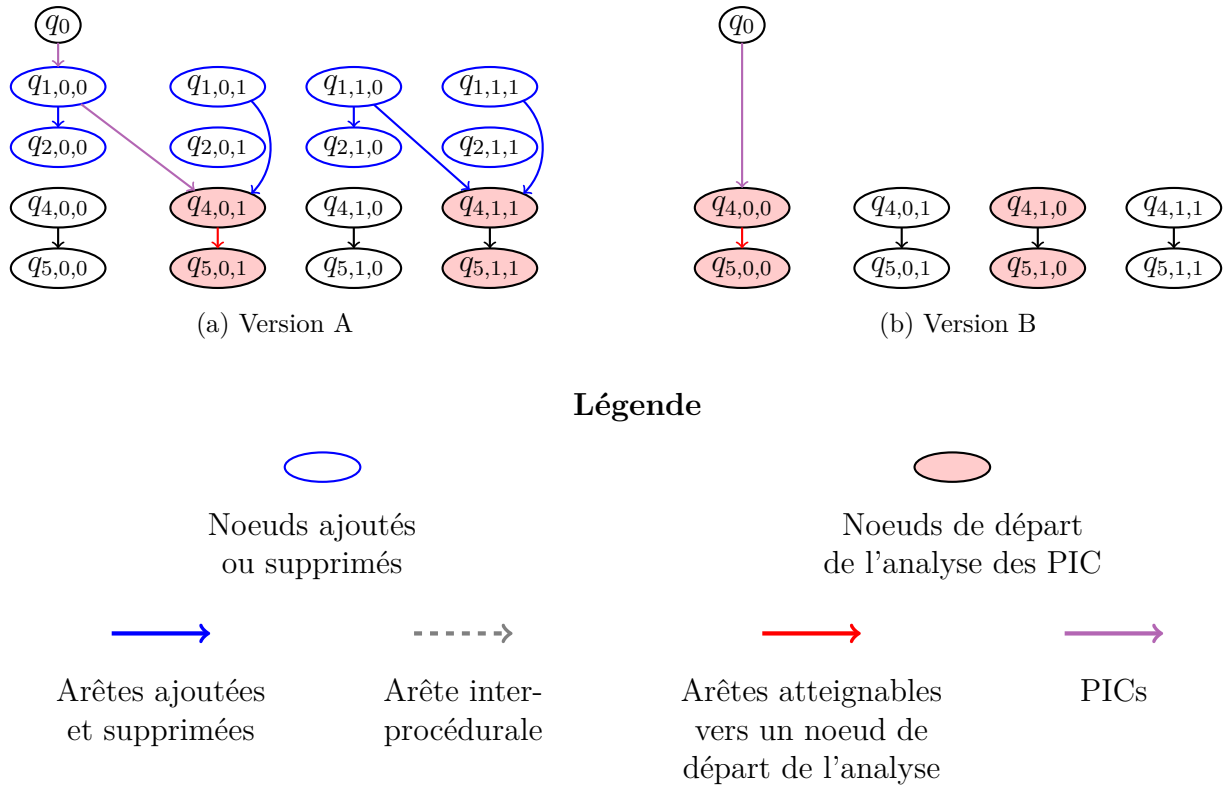
Figure 1.9 Lignes 88-94 du fichier wp-admin/user-edit.php de la version de WordPress [1] suivant le commit ea8078f.

$$
\begin{aligned}
addedStates(A,B) &= \{q_{i,j,k} \in Q_a, i \notin domain(vertexMap_{B,A}) \vee q_{vertexMap_{B,A}(i),j,k} \notin Q_a\} \\
deletedStates(A,B) &= \{q_{i,j,k} \in Q_a, i \notin domain(vertexMap_{A,B}) \vee q_{vertexMap_{A,B}(i),j,k} \notin Q_b\} \\
addedEdges(A,B) &= \left\{ q_{i_1,j_1,k_1}, q_{i_2,j_2,k_2} \in T_b, \left| \begin{array}{l} q_{i_1,j_1,k_1} \in addedStates \vee \\ q_{i_2,j_2,k_2} \in addedStates \vee \\ \begin{pmatrix} q_{vertexMap_{B,A}(i_1),j_1,k_1}, \\ q_{vertexMap_{B,A}(i_2),j_2,k_2} \end{pmatrix} \notin T_a \end{array} \right. \right\} \\
deletedEdges(A,B) &= \left\{ q_{i_1,j_1,k_1}, q_{i_2,j_2,k_2} \in T_a, \left| \begin{array}{l} q_{i_1,j_1,k_1} \in deletedStates \vee \\ q_{i_2,j_2,k_2} \in deletedStates \vee \\ \begin{pmatrix} q_{vertexMap_{A,B}(i_1),j_1,k_1}, \\ q_{vertexMap_{A,B}(i_2),j_2,k_2} \end{pmatrix} \notin T_b \end{array} \right. \right\}
\end{aligned}
\tag{1.4}
$$

Les PIC provoquant les pertes et les gains de protection sont définis séparément. Les PIC provoquant une perte de protection sont les suppressions d'arêtes des chemins protégés et les ajouts d'arêtes des chemins non protégés menant à la perte. Les PIC provoquant un gain de protection sont les ajouts d'arêtes des chemins protégés et les suppressions d'arêtes de chemins non protégés menant au gain.

$$
\begin{aligned}
partialPIC(changes,i,k) &= changes \ \cap \begin{pmatrix} ReachableEdges(q_{i,0,k}) \cup \\ ReachableEdges(q_{i,1,k}) \end{pmatrix} \\
&\forall (v_a,v_b) \in DefLoss(A,B), \\
PIC_{loss,edges}(v_a,v_b) &= \begin{pmatrix} partialPIC(deletedEdges(A,B),v_a,1) \ , \\ partialPIC(addedEdges(A,B),v_b,0) \end{pmatrix} \\
&\forall (v_a,v_b) \in DefGain(A,B), \\
PIC_{gain,edges}(v_a,v_b) &= \begin{pmatrix} partialPIC(deletedEdges(A,B),v_a,0) \ , \\ partialPIC(addedEdges(A,B),v_b,1) \end{pmatrix} \\
allPIC_{edges}(A,B) &= \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} \bigcup\limits_{(v_a,v_b)\in DefLoss(A,B)} \pi_1(PIC_{loss,edges(v_a,v_b)}) \end{pmatrix} \cup \\ \begin{pmatrix} \bigcup\limits_{(v_a,v_b)\in DefGain(A,B)} \pi_1(PIC_{gain,edges(v_a,v_b)}) \end{pmatrix} \end{pmatrix}, \\ \begin{pmatrix} \begin{pmatrix} \bigcup\limits_{(v_a,v_b)\in DefLoss(A,B)} \pi_2(PIC_{loss,edges(v_a,v_b)}) \end{pmatrix} \cup \\ \begin{pmatrix} \bigcup\limits_{(v_a,v_b)\in DefGain(A,B)} \pi_2(PIC_{gain,edges(v_a,v_b)}) \end{pmatrix} \end{pmatrix} \end{pmatrix}
\end{aligned}
$$

Dans l'équation 1.5, π_1 désigne la projection qui à une paire (x, y) associe x, et π_2 désigne la projection qui à une paire (x, y) associe y.



(a) Version A (b) Version B

Figure 1.10 Modèles PTFA correspondant à la paire de versions en Figure 1.7

Par exemple, pour la paire de versions présenté en Figure 1.7, les deux DPD sont des pertes de sécurité aux noeuds 4 et 5. Comme présenté en Figure 1.10, les PIC sont donc l'intersection des arêtes ajoutées et supprimées avec les arêtes atteignable vers un noeud de départ de l'analyse, donc les suppressions d'arcs sécurisés et les ajouts d'arcs dans un chemin menant à $q_{4,1,k}$ et $q_{5,1,k}$ et les suppressions d'arcs dans un chemin menant à $q_{4,0,k}$ ou $q_{5,0,k}$.

Les PIC sont donc la suppression de l'arc entre les noeuds q_0 et $q_{1,0,0}$, la suppression de l'arc entre les noeuds $q_{4,0,1}$ et $q_{1,0,0}$ et l'ajout de l'arc entre les noeuds q_0 et $q_{4,0,0}$.

Projection des PIC

Afin d'avoir des résultats exploitables pour les développeurs, ces PIC sont projetés sur les modifications de lignes de code, en utilisant une heuristique retenant les changements de lignes contenant les sources et les cibles des arcs de l'ensemble des PIC. Cette heuristique est explicité dans les équations 1.5, où line est la fonction qui à un état PTFA renvoie la ligne

du noeud correspondant, et source et target sont les fonctions renvoyant respectivement le noeud source et le noeud cible d'une arête.

$$
\begin{aligned}
&\forall (v_a, v_b) \in DefLoss(A,B) \\
PIC_{loss,lines}(v_a, v_b) = &\left(\begin{array}{l} \left(\begin{array}{l} sources(\pi_1(PIC_{loss,edges}(v_a, v_b))) \cup \\ targets(\pi_1(PIC_{loss,edges}(v_a, v_b))) \end{array} \right) \cap delLin(A,B), \\ \left(\begin{array}{l} sources(\pi_2(PIC_{loss,edges}(v_a, v_b))) \cup \\ targets(\pi_2(PIC_{loss,edges}(v_a, v_b))) \end{array} \right) \cap addLin(A,B) \end{array} \right) \\
&\forall (v_a, v_b) \in DefGain(A,B) \\
PIC_{gain,lines}(v_a, v_b) = &\left(\begin{array}{l} \left(\begin{array}{l} line(source(\pi_1(PIC_{gain,edges}(v_a, v_b)))) \cup \\ line(target(\pi_1(PIC_{gain,edges}(v_a, v_b)))) \end{array} \right) \cap addLin(A,B), \\ \left(\begin{array}{l} line(source(\pi_2(PIC_{gain,edges}(v_a, v_b)))) \cup \\ line(target(\pi_2(PIC_{gain,edges}(v_a, v_b)))) \end{array} \right) \cap delLin(A,B) \end{array} \right) \\
allPIC_{lines}(A,B) = &\left(\begin{array}{l} \left(\begin{array}{l} \left(\bigcup_{(v_a,v_b) \in DefLoss(A,B)} \pi_1(PIC_{loss,lines(v_a,v_b)}) \right) \cup \\ \left(\bigcup_{(v_a,v_b) \in DefGain(A,B)} \pi_1(PIC_{gain,lines(v_a,v_b)}) \right) \end{array} \right), \\ \left(\begin{array}{l} \left(\bigcup_{(v_a,v_b) \in DefLoss(A,B)} \pi_2(PIC_{loss,lines(v_a,v_b)}) \right) \cup \\ \left(\bigcup_{(v_a,v_b) \in DefGain(A,B)} \pi_2(PIC_{gain,lines(v_a,v_b)}) \right) \end{array} \right) \end{array} \right)
\end{aligned}
\tag{1.5}
$$

Dans l'exemple en Figure 1.8, la seule ligne contenue dans cette projection est la ligne 1. On en déduit que la suppression de l'instruction conditionnelle de la ligne 1 est la seule cause reportée des pertes de privilèges.

Réversion des PIC

Afin de valider l'analyse des PIC sur une paire de versions (A, B), une approche proposée est d'inverser les PIC de la version B. Si l'ensemble des PIC contient l'ensemble des causes racines, alors la réversion de ceux-ci devrait annuler l'ensemble des DPD. Une réversion est donc considérée comme réussie si celle-ci n'a plus de DPD avec la version antérieure.

Afin de minimiser les effets de bord des réversions (suppression de la définition d'une variable référencée par exemple), l'implémentation des réversions consiste à restituer les fichiers de la version A lorsque ceux-ci contiennent un PIC.

Nous désignons la nouvelle version ainsi obtenue comme la version réversée de la version B.

Dans l'exemple en Figure 1.8, les modifications à inverser sont l'ajout de l'instruction en ligne 1, étant reportée comme un PIC (comme vu précédemment), ainsi que l'ajout des instructions en lignes 2 et 3, qui sont un enfant de l'instruction en ligne 1 dans l'AST et dont l'ajout n'a donc plus de sens sans l'ajout de la ligne 1. En pratique, ce lien entre ces modifications n'est pas fait explicitement, mais la granularité par fichier des réversions permet de le faire implicitement.

La version reversée de la version B sera donc égale à la version A.

La réversion des fichiers des PIC a eu un taux de réussite de 89%, 93% et 100% pour les trois jeux de paires de versions, respectivement les versions publiques de WordPress et MediaWiki, et les commits réparant les CVE de WordPress. Les causes des cas d'erreur seront détaillées en Section 3.3.

Réversion individuelle des PIC relatifs à une DPD

L'analyse des PIC prenant en entrée un ensemble de DPD à réparer et la réversion prenant en entrée un ensemble de PIC, il est possible de calculer les PIC pour un sous-ensemble de différences de sécurité à réparer au lieu de les calculer pour toutes les différences de sécurité de la paire de versions.

En particulier, la réversion individuelle consiste à appliquer la réversion aux ensembles des PIC relatifs à un seul DPD choisi d'une paire de versions. Pour une seule paire de versions, nous pouvons donc faire autant de réversions individuelles que de DPD présentes.

La réversion individuelle d'une DPD est considérée comme réussie si et seulement si l'instruction associée n'est pas dans l'ensemble des DPD entre la version antérieure et la version réversée.

1.2 Hypothèse de recherche

Pour ce mémoire, nous émettons l'hypothèse de recherche suivante :

— Les DPD sont représentatives des injections de vulnérabilités dans l'application.

L'ensemble des analyses évoquées dans les définitions sont relatives à l'étude des DPD, que nous détectons et que nous tentons de réparer. La validité de nos résultats repose donc sur l'hypothèse que les différences de protection définitive sont représentatives des vulnérabilités de sécurité réelles. Cette hypothèse est supportée par la détection de différences de protection

définitive dans 19/31 (61.3%) des commits réparant les CVE (Common Vulnerabilities and Exposures), montrant un lien entre les vulnérabilités réelles et les DPD.

1.3 Motivation de la recherche

Comme présenté en Section 1.1.5, le calcul des différences de protection définitives par les analyses PTFA vise à détecter des vulnérabilités de contrôle d'accès injectées lors de la modification du programme. L'analyse des PIC permet quant à elle de calculer un ensemble de changements identifiés comme des causes racines potentielles de ces vulnérabilités. 41.4% des versions de WordPress et 21.9% des paires de versions de MediaWiki contiennent des différences de protection, que nous considérons comme des vulnérabilités par l'hypothèse de recherche. L'analyse des PIC identifie un sous-ensemble des changements de lignes contenant en moyenne respectivement 20.3% et 10.3% de tous les changements [19] pour aider les développeurs à identifier et à réparer les vulnérabilités injectées par leurs changements.

1.4 Objectifs de recherche

Objectif de recherche 1 : Identifier les limites de l'analyse des PIC causant des échecs de réversion.

Antérieurement au commencement de la maitrise, M-A. Laverdiaire a identifié que la réversion des fichiers des PICs réparait partiellement des versions présentant des vulnérabilités. 88.51% et 92.86% (respectivement pour WordPress et MediaWiki) des versions présentant des DPD sont ainsi correctement réversées.

Le premier objectif de recherche est ainsi d'investiguer les cas d'erreur du calcul des PIC afin d'identifier les limites de l'analyse.

Objectif de recherche 2 : Définir des nouvelles stratégies de réparation de DPD utilisant la réversion de PIC.

Une fois les limites de l'analyse des PIC étudiées, nous déterminons les possibilités de l'exploiter pour construire une stratégie de réparation des failles.

Nous utiliserons pour cela la réversion des PIC (voir Section 1.1.6), qui avait été utilisée comme moyen de validation de l'analyse des PIC. Nous nous sommes alors aidé de la connaissance des limites des analyses acquises avec l'objectif de recherche 1 pour adapter cette analyse en une stratégie de réparation de vulnérabilités.

Objectif de recherche 3 : Quantifier l'apport des stratégies proposées par rapport à la stratégie de réversion simple des PIC.

Une fois cette stratégie définie avec l'objectif de recherche 2, il est nécessaire de quantifier l'intérêt de l'application de cette stratégie, et son avantage par rapport à la stratégie de réversion des PIC décrite en Section 1.1.6, et par rapport une stratégie triviale qui consisterait à restaurer la version précédente pour annuler les différences de sécurité. Pour cela, nous devrons définir notamment des mesures sur les fonctions de réversion.

Objectif de recherche 4 : Proposer une stratégie de réparation utilisant des insertions de motifs.

Enfin, nous avons proposé un autre modèle de réparation des failles de sécurité, reposant sur l'insertion d'un motif sur des arêtes choisies.

Ce modèle permet de proposer une solution de génération de patchs appartenant à un espace de recherche différent de celui étudié dans l'objectif de recherche 1.

1.5 Plan du mémoire

Pour remplir ces objectifs, nous abordons trois thèmes après avoir présenté l'antériorité des travaux similaires. Tout d'abord, nous justifions les choix réalisés pendant la recherche, en nous appuyant notamment sur une revue des résultats de l'analyse des PIC. Ensuite, nous présentons une stratégie de réparation des DPD utilisant la réversion de changements, et nous étudions les résultats de cette stratégie. Enfin, nous proposons une stratégie de réparation des pertes de protection reposant sur des insertions de motifs.

CHAPITRE 2 REVUE DE LITTÉRATURE CRITIQUE

2.1 Analyses de détection de vulnérabilités

Pour notre étude, nous utilisons une analyse PTFA afin de calculer les propriétés protection des noeuds, et ainsi trouver les DPD d'une paire de versions d'un programme. Des analyses similaires ont été étudiées dans d'autres travaux, pour divers bugs et vulnérabilités. Dans cette section, nous abordons ces travaux afin de les comparer à l'approche que nous utilisons.

2.1.1 Analyses de Teintes

Un type d'analyse classique permettant de détecter des vulnérabilités est l'analyse de teintes. Celle-ci consiste à étudier la propagation de données depuis une source jusqu'à un puits. Elle vérifie ainsi que des fonctions permettant de nettoyer les variables sont présentes dans les chemins d'exécution reliant les sources aux puits. En effet, les variables provenant des entrées sont dites "teintées", puisqu'elles peuvent contenir des données corrompues par un attaquant.

De telles analyses permettent par exemple de mettre en évidence la présence de vulnérabilités permettant à un attaquant de réaliser une injection SQL. Ce type d'attaque consiste à insérer des requêtes SQL illégitimes dans un argument à l'entrée du programme, afin que cette requête soit transférée vers les puits par le programme et exécutée. Les fonctions de nettoyage associées à de type d'attaque vérifiera que les chaînes de caractères en entrée ne contiennent pas de requêtes SQL.

Une autre vulnérabilité des applications Web que ces analyses peuvent détecter sont les vulnérabilités de Script Inter-Sites (XSS). Les attaques exploitant cette vulnérabilité consistent à injecter un script dans l'application pour voler des données d'authentification à un utilisateur pendant sa connexion.

L'outil FlowDroid [20, 21] vise ainsi à détecter les vulnérabilités d'injection SQL. Pour ce faire, il effectue ainsi une analyse de teinte composée d'une analyse vers l'arrière permettant de trouver les alias des variables suivie d'une analyse vers l'avant, détectant si les variables provenant des sources rencontrent un nettoyeur. Pour réaliser ces analyses, il utilise le cadre et l'algorithme de Sous-ensemble Inter-procédural Fini Distribué (IFDS) [12]. Cet algorithme permet de résoudre des problèmes de propagation dans des flux de données en un temps polynomial.

L'outil FlowTwist [22] effectue également une analyse de teinte pour détecter les vulnéra-

bilités d'injection SQL. En plus de réaliser une analyse vers l'arrière permettant de vérifier l'intégrité des données (c'est-à-dire que les données arrivant aux puits ne peuvent être corrompues par des entrées), il réalise une analyse vers l'avant afin de vérifier la confidentialité des données (c'est-à-dire que les données envoyées à l'utilisateur ne peuvent contenir des données confidentielles). Ces analyses sont également faites avec une extension de l'algorithme IFDS.

L'outil Pixy [23] utilise également une approche d'analyse statique du flux de données afin de détecter les vulnérabilités d'injection SQL et des vulnérabilités XSS. Comme FlowDroid, Pixy utilise une analyse des alias afin de teinter tous les alias d'une variable lorsque celle-ci est teintée.

Des approches dynamiques existent également afin de détecter l'utilisation de données teintées dans des puits. Les outils utilisant ces approches, tels que TaintTrace [24] et TaintDroid [25] tracent ainsi les variables pendant une exécution, en étiquetant les variables teintées par une entrée, et en rapportant une erreur si ces variables atteignent des puits sans avoir rencontré une fonction de nettoyage. Le principal défi de ces approches est le ralentissement important que l'approche apporte à l'exécution de l'application, qui est de 5.5 fois pour TaintTrace et de 0.14 fois pour TaintDroid.

Ces analyses sont analogues aux analyses de la propagation des protections par les privilèges utilisées pour l'analyse des DPD (Section 1.1.5), mais utilisent les graphes de flux de données, prenant en considération des problématiques différentes, comme l'analyse d'alias. Les cadres comme celui de l'algorithme IFDS utilisant le flux de données ne sont donc pas applicables à notre problème, même si des analyses similaires peuvent être effectuées.

2.1.2 Analyses de vérification de modèle

D'autres analyses classiques permettant de détecter des vulnérabilités sont les analyses de vérification de modèle. Les analyses de DPD (Section 1.1.5) rentrent dans cette catégorie.

Ainsi, Wang et al. [26] propose une approche permettant de détecter des bugs classiques dans le langage C, tels que les débordements de tableau. Il définit des contraintes et des assertions associées à chaque type d'instruction relative aux pointeurs ou aux tableaux, et génère un nouveau programme où ces contraintes et ces insertions sont explicitées comme des instructions sur des variables temporaires. Les assertions détectent alors la violation des conditions d'une des instruction, et détectent alors l'occurrence d'une erreur dans le programme. Une analyse de Slicing permet enfin de réduire le nombre d'instructions ajoutées en supprimant celles n'étant pas pertinentes.

2.2 Réparation automatique de vulnérabilités

La réparation de bugs dans un programme est une tâche longue, coûteuse et répétitive. La présence de motifs récurrents d'erreurs et de réparations motive la recherche de techniques de réparation automatique de ces erreurs.

Cette revue de littérature vise à introduire les enjeux principaux de la réparation automatique de programmes et à présenter les stratégies récurrentes. Dans un premier temps, nous ferons une revue critique des différentes approches pour la formalisation des problèmes de réparation automatique. Ensuite, nous soulignerons les enjeux fréquemment rencontrés dans les différentes études. Enfin, nous verrons les différentes stratégies appliquées pour la réparation automatique.

2.2.1 Formalisation de la réparation automatique de programmes

Définition d'une erreur, oracles

La définition d'un oracle permettant de valider la correction d'un programme est nécessaire [27] au traitement d'un problème de réparation automatique. Divers types d'oracles sont présents dans la littérature, et les techniques de réparation induites en dépendent fortement.

Un premier oracle récurrent est la validation de cas de test [28–33]. Cette définition par les cas de test des erreurs à corriger est utilisée dans l'immense majorité des outils de réparation automatique existants. Une autre stratégie présente dans la littérature est de cibler un type d'erreur, et d'ainsi avoir un oracle simple qui détecte la présence potentielle de telles erreurs dans un programme. L'appréciation de cet oracle est alors réalisée par des mesures faites lors de l'exécution. AutoPag [34], TAP [35] ainsi qu'Infinitel [36] sont de tels systèmes. AutoPag [34], TAP [35] ciblent les erreurs de dépassement des bornes d'un tableau, tandis qu'Infinitel [36] vise les boucles infinies. Pour chacun de ces systèmes, les erreurs sont relevées lors d'une exécution du programme, et les informations relatives à l'erreur sont reportées à l'outil de génération de réparation. Enfin, la vérification de propriétés Satisfiability Modulo Theories (SMT) sur un graphe d'états [37, 38] est également utilisée pour définir la présence d'erreurs à réparer. Contrairement aux autres stratégies introduites, celle-ci ne nécessite pas une exécution du programme et l'occurrence explicite des erreurs pour les détecter.

Une limite reconnue de l'utilisation des cas de test comme oracle est l'incohérence de certains patchs générés, qui se conforment trop aux cas de test, et la difficulté de les appliquer dans des cas réels, les tests étant souvent insuffisants par rapport aux baselines utilisées pour évaluer la réparation automatique [32]. De plus, de tels outils de réparation d'erreurs utilisent

généralement les mêmes ensembles de programmes erronés à réparer, tels que Defects4J [39] ou QuickBug [40]. Cela permet une comparaison rapide entre les différents outils de ce type, mais un problème inhérent à ce constat est le surajustement des réparations de bugs sur les bases d'erreurs utilisées, les développeurs adaptant et testant leurs outils sur ces mêmes bases [41, 42].

Qi et al. [43] montrent l'importance du choix de définition de l'erreur et de nos attentes par rapport à sa réparation. Il montre entre autres que les spécifications des cas de test utilisés par GenProg [28] sont insuffisantes pour générer des patchs corrects, et que par conséquent les réparations automatiques proposées se limitent à la suppression de la fonctionnalité comprenant l'erreur. L'article propose enfin un algorithme, Kali, qui se contente de générer un patch supprimant la fonctionnalité, de manière plus simple et plus efficace que GenProg puisque l'espace de recherche s'en trouve réduit. Ce constat souligne également l'important d'une revue manuelle des versions réparées.

Mesure de la qualité d'une méthode de réparation

Outre la contrainte de la validation d'un oracle par le programme réparé, d'autres mesures de la qualité d'une méthode de réparation permettent de juger un algorithme. En particulier, Samanta et al. [37] définissent formellement une fonction de coût d'une réparation. Ils considèrent que les réparations sont définies comme une fonction assignant une action de modification à chaque instruction. Ces actions ont chacune un coût donné (par exemple, l'action de laisser l'instruction inchangée a toujours un coût nul) et le coût de la réparation est la somme des coûts des modifications. Cette mesure correspond donc au volume des modifications réalisées pour la réparation.

Weimer et al. [44] utilise d'un algorithme génétique pour réaliser des réparations automatiques, en s'appuyant sur une fonction de fitness mesurant le comparant le nombre de cas de test passant entre le candidat et la fonction non réparée. Il traite également le problème de la minimisation du nombre de changements réalisés en recherchant le sous-ensemble de changements le plus petit faisant passer tous les cas de tests en partant d'un ensemble de changements réparant le programme.

Les mesures vues ci-dessus sont des mesures syntaxiques, mesurant les différences de code nécessaires à la réparation mais non la différence de comportement. Cependant, deux programmes avec peu de changements entre eux peuvent néanmoins avoir des comportements différents. Qlose [45] considère les distances syntaxiques et sémantiques entre plusieurs programmes afin de quantifier leur dissimilarité. La distance sémantique mesure la distance entre l'exécution d'un jeu de test donné sur les programmes, et permet donc de mesurer la proxi-

mité de leur comportement. Qlose permet de combiner ces deux mesures afin d'évaluer les réparations en agrégeant plusieurs critères.

Dans GenProg et Autopag, la qualité de la réparation est mesurée en comparant les performances temporelles des programmes avant et après réparation. Cette mesure explicite l'impact de la réparation sur l'exécution du programme, alors que les mesures relatives au nombre de changements reflètent la distorsion de code causée par la réparation.

Espace des modifications

Une autre partie nécessaire à la formalisation d'un problème de réparation automatique est la définition d'un espace des réparations possibles. L'espace des modifications impacte la réparabilité des erreurs, et sur la difficulté de la recherche de réparations.

Martinez et Monperrus [46] proposent une division de l'espace de recherche d'une modification comme un produit cartésien de trois ensembles : l'ensemble des positions dans le programme, l'ensemble des formes de modification (173 formes différentes), et la synthèse de la modification (avec les valeurs des données de la modification). Ils étudient le nombre d'essais nécessaires afin de trouver les formes de la réparation (se limitant donc à l'un des trois ensembles du produit), et tente de le réduire en utilisant une approche probabiliste. Ce nombre d'essais reste néanmoins exponentiel en la taille de la réparation. Ce résultat illustre la nécessité de réduire l'espace de recherche afin de trouver efficacement des réparations.

Dans GenProg [28] [44], l'espace de réparations est l'ensemble de mutations de la forme $(s \leftarrow \{\})$ (suppression d'une instruction), $(s \leftarrow s', s' \leftarrow s)$ (échange de deux instructions) ou $(s \leftarrow \{s, s'\})$ (copie d'une instruction), avec s et s' des instructions du programme.

L'ensemble est alors assez réduit pour permettre l'application d'un algorithme génétique avec les mutations proposées, mais certaines réparations, nécessitant par exemple l'ajout d'une instruction n'étant pas déjà présente dans le programme, ne sont pas accessibles avec ces réparations. Z. Qi et al. [43] montrent cependant que cet espace de recherche est trop restreint pour générer des réparations réalistes et correspondant aux attentes des développeurs.

Dans SPR [29], l'espace de recherche est plus vaste. Il est composé des introductions et modifications de conditions, des insertions d'opérations de flux de contrôle, des insertions d'initialisations, des remplacements de valeurs et des copies d'instructions. Plus de réparations sont alors atteignables, et l'algorithme réalise une concaténation de réparations afin d'avoir un espace de recherche assez large tout en étant efficace.

Le système de réparation PAR [30], visant un problème de réparation générale d'erreurs, limite son espace de recherche aux 6 motifs de réparation qu'il considère comme les plus

fréquents (30% des patchs observés). Ces motifs étant généraux (ajout d'une condition, ajout d'un paramètre à une méthode...), plusieurs modifications restent possibles pour un même motif à une même location.

Le système de réparation ARC [47], qui répare les erreurs de synchronisation dans un code utilisant la programmation concurrente, cible un espace de recherche contenant uniquement 12 mutations précises, qui consistent à des ajouts et des retraits de synchronisation. Pour une de ces mutations et un emplacement où elle peut être appliquée, une seule solution est alors possible. Une telle restriction améliore l'efficacité de la recherche d'un patch, mais limite l'ensemble des programmes réparables.

Le système Autopag [34] a également un espace de recherche très limité, puisque les réparations se limitent à des motifs de réparation spécifiques pour chaque erreur relevée. Par exemple, pour un accès hors bornes `p[i]`, le patch remplacera simplement l'expression par `p[i mod size]`. Le système TAP [35] résout le même type d'erreurs, et insère une instruction `if(cond) exit(-1);`, où la variable `cond` est calculée lors de la recherche d'erreurs. Pour ces outils, une seule forme de mutation est applicable, et le problème de réparation se restreint donc à la localisation des emplacements où les réparations doivent être appliquées.

2.2.2 Enjeux et problèmes de la réparation automatique

Comme dit précédemment, le premier enjeu de la réparation automatique est la formulation du problème et des attentes envers le patch généré. Cette partie est essentielle à cause de la taille de l'espace des modifications possibles si celui-ci n'est pas définit correctement, et de la variabilité des définitions des erreurs à réparer selon le problème traité.

Dans cette section, nous traitons des autres enjeux récurrents étudiés dans la littérature.

Effets de bord

La réparation d'une erreur peut avoir des effets de bord. En effet, les mutations proposées par le patch modifie le comportement du programme, et peuvent avoir un impact sur le reste du programme. L'un des thèmes récurrents est de minimiser ces effets de bord.

Par exemple, la stratégie de résolution des débordements de tableaux d'Autopag [34], qui renvoie à la lecture d'un autre élément du tableau, peut avoir un impact important sur le comportement du programme. Lors de la correction d'un accès hors borne, une valeur potentiellement non voulue sera attribuée à une variable, et pourra induire un comportement non désiré du programme. L'article présentant cet outil, en plus de recommander une revue manuelle systématique des réparations générées, traite à part le cas où l'outil modifie des

instructions impactant la condition d'une boucle, et risque de créer une boucle infinie. Il génère alors une modification différente de celle-ci dans ce cas. La stratégie de TAP [35] ne peut pas rencontrer ce type de problème, puisqu'elle consiste à arrêter l'exécution du programme avant l'occurence de ce type d'erreur, et n'introduit donc pas de comportements qui ne sont pas présents dans la version non réparée.

Correction de la réparation

La réparation d'un programme peut injecter des erreurs sortant du cadre de l'oracle défini. Les articles traitant de la réparation automatique différencient en général la *plausibilité* d'un patch, qui détermine si un patch généré est validé par la stratégie de réparation, et sa *correction*, qui détermine si l'outil génère un patch exact, au delà des critères de l'oracle.

Afin d'évaluer la correction de la stratégie de réparation automatique au delà des critères de validation intrinsèques à la stratégie de génération, deux techniques d'évaluation de la correction sont généralement appliquées [48] : l'application de cas de test disjoints de ceux utilisés pour l'apprentissage du patch, et l'évaluation manuelle des patchs.

Comme mentionné en Section 2.2.1, une étude [42] sur GenProg et TrpAutoRepair a appliqué des jeux de tests par boite blanche à des patchs plausibles créés par ces stratégies. Pour les deux outils, les patchs générés passent en médiane 75% des cas de test, bien que l'outil présentait le patch comme réussi.

D'autres études mesurent la correction des patchs générés grâce à une évaluation manuelle. Ainsi, Martinez et al. [49] ont analysé manuellement 84 patchs générés par 3 systèmes différents, jGenProg [28], jKali [43], et Nopol [33], trois systèmes de génération de patchs basés sur la correction de suites de tests, sur la base d'erreurs de Defects4J [39]. Il trouve que sur ces 84 patchs, malgré la validation des oracles par ceux-ci, seuls 11 sont corrects, 61 sont incorrects, et 12 nécessitent une expertise pour déterminer la correction.

Réparation composée

Un autre enjeu de la réparation automatique de programmes est la recherche et l'application de réparations composées. Le problème de la génération d'une modification composée peut avoir une complexité combinatoire avec des approches naives *Generate and Validate*. Beaucoup d'outils se limitent donc à la génération d'une réparation simple pour cela, comme l'insertion ou la suppression d'une instruction [33]. D'autres proposent des approches prenant en compte la possibilité que l'application d'un seul motif peut être insuffisant pour trouver une réparation.

Hercules [50] est un outil considérant la possibilité de réparations composées. Lorsque celui-ci trouve une localisation candidate pour la réparation, il cherche également les parties du code similaires à celle-ci, par leur contexte et leur évolution au fil de l'histoire des commits. Lorsqu'un patch est choisi et appliqué à une location, il est ainsi répliqué aux autres locations similaires, réparant l'erreur partout où elle est répliquée. L'outil considère donc un espace de recherche plus riche, puisqu'il inclut des réparations plus complexes que lorsqu'elles sont appliquées à une seule location.

Apprentissage des patchs humains

Un autre problèmes courant est l'apprentissage et l'application de patchs existants au programme à réparer. L'utilisation de patchs corrects humains permet d'éviter la génération de patchs incohérents, puisque ceux-ci sont issus d'une réparation correcte, ayant été écrite et validée par un développeur. Les approches de réparation peuvent apprendre des patchs humains soit en les extrayant automatiquement, soit en utilisant un motif récurrent comme réponse systématique à une même erreur.

Afin d'appliquer une réparation que nous avons apprise d'un autre commit à un contexte différent du programme à réparer, l'outil sharpFix [51] introduit une stratégie consistant à faire correspondre les variables, méthodes et classes utilisées dans le contexte de la réparation minée à des variables, méthodes et classes du contexte du code à réparer. Le nom des objets et les relations d'encapsulement des classes sont utilisés pour former cette correspondance. Les éléments du patchs sont alors transformés selon cette correspondance pour être testés sur le programme à réparer. L'étude a montré que l'utilisation de ces correspondances a permis à l'outil de générer plus de patchs qu'il a pu le faire sans celles-ci.

Prophet [31] est une approche qui extrait des patchs réussis, et calcule les probabilités de réussites de ces patchs. Pour ce faire, l'outil apprend avec les différents patchs existants des coefficients à appliquer à un vecteur de propriétés du programme et du patch administré à ce programme. Les coefficients ainsi appris sont ensuite appliqués au programme à réparer avec les différents patchs à tester aux différents points du programme, estimant ainsi la probabilité de succès du patch.

2.2.3 Application de méthodes empiriques pour la réparation

Une technique classique de réparation de programmes consiste à générer des patchs candidats, puis de les tester afin de chercher un patch admissible.

Les outils s'appuyant sur cette méthode diffèrent par la manière dont les patchs sont générés

et par la mesure de la plausibilité d'un candidat. En effet, comme discuté dans la Section 2.2.1, ces méthodes sont fortement dépendantes de la définition des erreurs utilisée, puisque c'est celle-ci qui décidera de l'acceptation d'un patch.

Dans l'outil *Pattern-based program repair* [30], les patchs candidats sont inspirés de patchs manuels récurrents. Ils sont créés en appliquant des motifs de réparation fréquents autour de l'emplacement de l'erreur. Les candidats sont ensuite comparés entre eux en terme des cas de test qu'ils corrigent pour pouvoir en sélectionner un.

Un autre outil, Prophet [31], utilise l'espace de réparations de SPR [29] décrit en Section 2.2.1. Il apprend la probabilité de succès de chaque candidat possible d'une base de données de patchs manuels identifiés comme réussis. Il teste ensuite les patchs candidats dans l'ordre décroissant de probabilité de succès, et retourne le premier patch réussissant l'ensemble des cas de test définissant la correction du programme.

Nopol [33] est également un outil visant à réparer des programmes à partir de cas de test échouant en modifiant et en insérant des conditions. Les tentatives de réparations des programmes se font en deux étapes. Tout d'abord, l'outil détecte les localisations candidates pour la réparation. Pour chaque cas de test échouant, on lance l'exécution du cas de test en forçant la condition candidate à être vraie, puis fausse. Si pour chaque cas de test défaillant, l'une des deux solutions répare le cas de test, l'instruction conditionnelle est ajoutée comme localisation candidate.

Une fois une localisation candidate trouvée, l'algorithme cherche un paramètre booléen valant la valeur qui était forcée lors de l'étape de la localisation des réparations des cas de test défaillants, et qui vaut la valeur initiale de la condition pour chaque cas de test réussi. Si un tel paramètre est trouvé, alors le remplacement de la condition par ce dernier répare les cas de test défaillants tout en n'impactant pas les cas de test réussis.

ACS [32] répare les erreurs en insérant ou en modifiant des conditions. Pour un cas de test échouant, ACS identifie la dernière instruction exécutée. Il tente ensuite d'insérer une instruction conditionnelle à cet endroit "`if (c) return x;`" ou "`if (c) throw E;`", où `x` et `E` sont les valeurs attendues pour le cas de test pour l'oracle. Si aucune des conditions testées ne fonctionne, il essaye de modifier la dernière instruction conditionnelle exécutée en lui appliquant le motif de modification "`if (b)`" $\rightarrow$ "`if(b&&c)`" ou "`if(b||c)`" selon si la condition est passante ou non dans l'exécution du cas de test. Les conditions c testées sont choisies grâce à un classement de conditions candidates combinant trois heuristiques proposées par les auteurs : la priorisation des conditions les plus récentes en termes de dépendance, l'analyse des documents du projet et l'extraction des conditions dans des extraits similaires d'autres projets.

L'outil Qlose [45] cherche quant à lui à minimiser la distance entre le programme réparé et le programme originel tout en respectant des contraintes (validation des cas de test). Il applique ainsi un algorithme d'optimisation MaxSMT afin de trouver une réparation validant les cas de test étant la plus proche possible du programme originel. La distance utilisée pour l'optimisation est une combinaison d'une distance syntaxique (différence des expressions du programme) et d'une distance sémantique (différence de comportement des programmes face à des cas de test).

2.2.4 Application de méthodes formelles pour la réparation

Des méthodes formelles et des analyses statiques des programmes défaillants sont également utilisées pour générer des patchs automatiques.

Autopag [34], qui vise à réparer les accès hors bornes, utilise une analyse de flux de données afin de trouver l'ensemble des causes des accès hors bornes détectés. Il répare ensuite ces causes en leur appliquant des motifs, comme détaillé en Section 2.2.1.

VFix [52] est un outil visant à réparer les exceptions de pointeur nul provoquées par des cas de test. Celui-ci calcule le flux de données d'une variable qui provoque une exception de pointeur nul, et détermine ainsi les causes possibles de l'erreur. Une réparation sera appliquée à l'instruction du flux de données appartenant au plus grand nombre de chemins d'exécutions, afin de rendre le patch aussi correct que possible au delà des cas de test, considérant ainsi le problème de manque de spécification des oracles basés sur des cas de test évoqué en Section 2.2.1. Il essaye ensuite d'insérer une instruction permettant de passer la partie du programme provoquant l'exception lors de l'exécution, ou d'initialiser la variable problématique si la première solution n'est pas applicable.

L'outil FootPatch [38] cible les erreurs d'allocations de mémoire. Il corrige les erreurs détectées par l'outil Infer [53], comme les fuites de mémoire, grâce à des analyses statiques basées sur la logique de séparation. Une fois l'erreur détectée, les spécifications désirées pour la réparation sont renseignées manuellement sous forme de triplets de Hoare avec les préconditions et les postconditions attendues pour le programme réparé. L'outil cherche ensuite dans le reste du programme des méthodes vérifiant les spécifications recherchées, et les désigne comme candidats pour la réparation. Le candidat le plus adéquat pour les types de variables ciblées est alors sélectionné.

2.2.5 Détection et réparation de vulnérabilités de contrôle d'accès

Dans le cadre de notre recherche, nous cherchons à réparer des vulnérabilités de contrôle d'accès.

Plusieurs approches ont été proposées dans la littérature afin de réparer des vulnérabilités de ce type.

Xu et Peng [54] proposent une méthode de réparation des politiques de contrôle d'accès basé sur les attributs (ABAC) dans le langage XACML, en localisant des fautes d'accès grâce à des cas de test. Les éléments de la politique se trouvant le plus dans les chemins des tests défaillants sont considérés comme suspicieux, et des mutations aléatoires sont appliquées aux éléments les plus suspicieux. Si une mutation améliore l'ensemble de tests passants, elle est conservée, et le processus est répété sur les autres éléments suspicieux jusqu'à la réparation de la politique, ou la reconnaissance d'un échec de la réparation.

Ce travail se rapproche de notre objectif puisqu'il vise la réparation d'accès de contrôle erronés. Néanmoins, il en diffère par le type d'objet réparé, puisqu'il répare des politiques d'accès en langage XACML, alors que nous ciblons l'implémentation de ces politiques. Il en diffère également par la définition des fautes, puisqu'elle est ici basée sur la validation de cas de test. Les techniques de localisation et de réparation que nous explorerons sont donc différentes de celles qu'il propose.

Fix Me Up [55] prend en entrée une politique de sécurité RBAC, c'est à dire une liste d'instructions à sécuriser avec des vérifications. Il parcourt tous les chemins allant vers les instructions à sécuriser par un parcours en profondeur en traversement arrière sensible au contexte d'appel des méthodes. L'algorithme applique alors une réparation au contexte d'appel non sécurisé de l'instruction sensible, en insérant une vérification de privilège.

Muthukumaran et al. [56] proposent un outil de réparation de contrôle d'accès par insertion de vérifications d'autorisation avant les instructions sensibles identifiées par un utilisateur. L'outil tient également compte des problématiques de redondance des vérifications et de sécurisation inutile d'instructions non sensibles à la sécurité, afin d'optimiser l'insertion de vérifications.

L'objectif de Fix Me Up et de l'approche présentée par Muthukumaran est similaire à notre objectif de recherche puisqu'il vise à corriger des vulnérabilités RBAC. Néanmoins, la technique de détection des niveaux de sécurité diffère par sa complexité, les deux études antérieures utilisant des analyses sensibles aux contextes d'appel, tandis que les analyses PTFA distinguent uniquement deux contextes d'appel, les appels dans un contexte définitivement sécurisé, et les appels dans un contexte non définitivement sécurisé. Les réparations que l'on

vise prendront donc une granularité de contexte plus large et mieux adaptée aux analyses RBAC, réduisant les calculs nécessaires à l'analyse.

L'objectif de recherche diffère également par le type de ces vulnérabilités. Fix Me Up et l'outil présenté par Muthukumaran réparent les programmes enfreignant une politique de sécurité. Notre objectif d'utilisation des DPD comme oracle de détection de vulnérabilités, quant à lui, ne nécessite pas de spécifications de l'utilisateur quant aux méthodes à sécuriser. Les réparations induites visent donc les vulnérabilités injectées par un commit.

2.2.6 Synthèse et application à nos objectifs de recherche

Dans le cadre de cette recherche, nous souhaitons mettre en place une stratégie de réparation automatique d'erreurs injectées entre deux versions : la version antérieure est considérée comme correcte par l'hypothèse de recherche (Section 1.2) et utilisée comme une référence à laquelle on compare la version postérieure, que l'on répare si elle est incorrecte par rapport à la version antérieure. Nous pouvons ainsi définir la présence d'erreur en utilisant la version antérieure. En d'autres termes, l'oracle de plausibilité est exprimé comme un prédicat réflectif $p(v_a, v_b) \in \mathcal{B}$, avec v_a la version de référence, et v_b la version à réparer. Contrairement aux travaux réalisés sur la réparation automatique s'appuyant sur des jeux de tests [28–33] ou sur des spécifications manuelles [37, 38] pouvant eux-même contenir des erreurs, ce problème ne dépend donc pas d'un oracle définit par le développeur, mais se sert de la version antérieure comme oracle.

Les réparations visées par l'objectif de recherche étant ciblées sur un type précis d'erreur (la violation du prédicat), les approches utilisées face à ce type de problème sont généralement les applications de méthodes formelles (Section 2.2.4). Celles-ci permettent de garantir la réparation de l'erreur dans le patch généré.

L'étude de la littérature montre également la nécessité de la mesure de la qualité de la stratégie. Dans les études revues, deux démarches sont abordées pour mesurer le poids du patch : un poids statique basé sur la distance syntaxique des deux programmes, et un poids dynamique basé sur la distance sémantique des deux programmes, mesurée lors de l'exécution de cas de test. Notre étude reposant entièrement sur des analyses statiques, nous choisissons de mesurer le poids des patchs générés avec une distance syntaxique.

Dans notre étude, nous proposons une approche de minimisation des changements incrémentielle (Section 4.2.5), similaire à celle proposée par Weimer et al. [44]. Néanmoins, l'approche qu'ils proposent est quadratique en fonction de la cardinalité de l'ensemble des changements, tandis que la notre est linéaire en fonction de la taille des changements. Néanmoins, notre ap-

proche peut retourner une solution non minimale, contrairement à celle avancée par Weimer et al.

A la lumière de l'étude de la littérature, différents enjeux relatifs à la réparation automatique sont considérés, et devront être pris en compte lors de la mise en place de la stratégie de réparation.

— Le premier enjeu relevé est la possibilité d'effets de bord causés par le patch. Dans notre approche, nous nous assurons que les patchs ne causent pas des défaillantes collatérales.

— Un autre enjeu récurrent est la génération de réparations composées. Pour résoudre cet enjeu, notre approche doit pouvoir réparer des programmes avec des vulnérabilités à plusieurs positions.

— Enfin, la correction des patchs générés au delà de l'oracle doit être vérifiée. L'oracle étant définit formellement, il ne rencontre pas la possibilité de spécification incomplète, mais il faut s'assurer que l'application remplit toujours les fonctions attendues. L'espace de recherche des patchs doit être construit en sorte d'assurer la correction de ceux-ci.

CHAPITRE 3 ÉTUDE DES DIFFÉRENCES DE SÉCURITÉ ET DES CHANGEMENTS IMPACTANT LA PROTECTION

3.1 Approche proposée

Dans la Section 1.1, nous avons présenté des analyses statiques différentielles relatives aux propriétés de protection définitive des instructions vis-à-vis d'un privilège. L'utilisation d'analyses statiques différentielles [57, 58], c'est à dire basées sur les différences entre les versions, pour détecter des vulnérabilités, permet d'éviter de devoir définir des spécifications pour déterminer les erreurs, et ainsi de se détacher des limites vues en Section 2.2.1 liées à ces spécifications. En effet, on considère les vulnérabilités injectées entre deux versions. Les propriétés de la version antérieure sont utilisées comme un oracle pour les propriétés de la version postérieure.

Notre objectif est de définir des algorithmes de génération de patchs corrigeant ces vulnérabilités.

Dans ce Chapitre, nous déterminons l'environnement dans lequel nous appliquerons ces fonctions, et nous présentons l'étude des résultats de l'analyse des PIC débouchant sur ces définitions.

3.2 Choix du langage et des systèmes à analyser

Les différentes analyses présentées en Section 1.1 ont été appliquées sur des projets PHP. Ce choix est motivé par le vaste choix de systèmes RBAC disponibles en PHP (WordPress, MediaWiki, Moodle). PHP étant un langage interprété, le laboratoire utilise son propre front-end traduisant le code dans un langage intermédiaire.

La première partie de ce front-end est un parseur, extrayant l'AST correspondant à chaque fichier php de l'application. L'AST est ensuite converti dans un format adéquat aux différentes analyses contenant les informations pertinentes, telles que les instructions du programme et les arêtes d'accès, ainsi que les arêtes accordant ou révoquant un privilège, accolées sur les motifs de sécurité choisis. Il est important de signaler que ces motifs étant dépendants du système analysé, cette conversion est propre à un système.

Trois systèmes de gestion de contenu vastement utilisés sont généralement étudiés dans les études relatives aux analyses PTFA :

— WordPress [59] est un système général et flexible utilisé par 34.1% des sites Web.

— MediaWiki [60] est un système utilisé dans des sites de partage de connaissance, notamment par Wikipedia.

— Moodle [61] est un système d'environnement pédagogique.

Ces trois applications sont donc largement utilisées, et proposent un système de contrôle d'accès. Elles sont toutes trois implémentées en PHP. Un résumé des propriétés et de la politique de contrôle d'accès est proposé dans le Tableau 3.1.

Tableau 3.1 Description de différentes applications utilisant un modèle de contrôle d'accès

Système	Langage	Roles	Privilèges	Dernière version	Taille	Téléchargement
Wordpress	PHP	6	64 [59]	5.2.1	40.6 MB	[59]
Mediawiki	PHP	7	82 [60]	1.32.1	133 MB	[60]
Moodle	PHP		634 [61]	3.7	171 MB	[62]

La complexité de l'étude d'un ensemble de paires de versions est détaillée dans l'équation 3.1. Les différentes analyses étant faites sur chaque privilège séparément, elles sont linéaires par rapport au nombre de privilèges étudiés. Elles sont également linéaires sur le nombre d'arêtes du CFG [13], donc sur la taille du projet. Pour cette raison, les analyses ne permettent pour l'instant pas d'analyser un système tel que Moodle, dont la taille et le nombre de privilèges est trop élevé, et seuls WordPress et MediaWiki ont été étudiés.

$$C_{PTFA} = \mathcal{O}(\ \overbrace{|Edges(CFG)|}^{\text{Nombre d'arêtes par version}} \times \overbrace{p}^{\text{Nombre de privilèges étudiés}} \times \overbrace{v}^{\text{Nombre de versions étudiées}}\ ) \quad (3.1)$$

3.2.1 Jeux de données disponibles pour les expériences

Les ensembles de tests considérés dans les différentes expériences sont ceux utilisés dans les études précédentes sur les DPD [16] et les PIC [18, 19].

Ce jeu de données est constitué de trois ensembles de paires de versions :

— Pour les applications WordPress et MediaWiki, les ensembles de paires des versions successives de l'application sont étudiés.

L'ensemble des paires de WordPress comporte 210 paires de versions, dont 87 comportant des DPD [19] (donc pour lesquelles la version postérieure nécessite une correction). Par la suite, nous nommerons ce jeu de données WP. L'ensemble des paires de WordPress comporte 192 paires de versions, dont 42 comportant des DPD [19]. Par la suite, nous nommerons ce jeu de données MW.

— Pour WordPress, des commits réparant des vulnérabilités connues et répertoriées dans la base de données Common Vulnerabilities and Exposures (CVE) [63] ont également été identifiées manuellement par M.-A. Laverdière. Pour chaque vulnérabilité relative au contrôle d'accès, le commit la réparant est déterminé. L'ensemble des commits ainsi identifiés comprend 31 paires de versions, dont 19 comportant des DPD. Par la suite, nous nommerons ce jeu de données CVE.

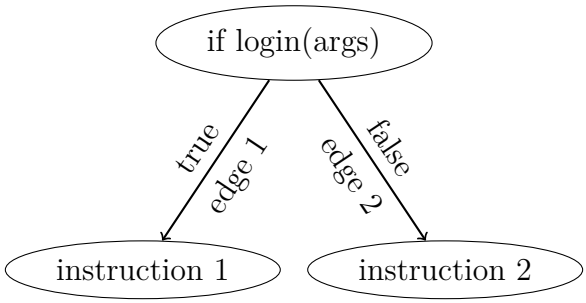
3.2.2 Extension vers l'analyse d'autres systèmes avec Wala

Afin d'ouvrir les résultats à d'autres systèmes, une partie préliminaire de la recherche est d'étendre les analyses courantes aux projets Java. Afin d'extraire les informations pertinentes du projet, nous utilisons Wala [64], une librairie d'analyse statique capable notamment d'extraire le CFG interprocédural d'instructions SSA correspondant à un projet. Nous exportons ensuite ce CFG dans un format adéquat pour les analyses PTFA.

Wala analysant directement le code java compilé, le CFG extrait décrit le comportement du programme en bas niveau. Les expressions et les conditions, entre autres, sont divisées en éléments unitaires. Le traitement des exceptions et du polymorphisme de fonctions est également résolu, et des arêtes sont placées à chaque accès possible.

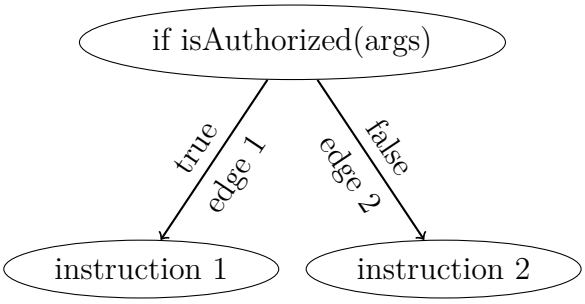
Une fois le projet extrait, il est nécessaire de faire une extraction des motifs de vérification de privilèges adaptée au projet à analyser. Cette étape est propre à l'application étudiée. Nous avons choisi d'utiliser le projet WebGoat [65], un modèle d'application web développé à des fins didactiques par OWASP présentant délibérément des vulnérabilités classique. L'application comporte des leçons, que l'utilisateur peut mettre en pratique en exploitant les vulnérabilités de WebGoat. Pour ce projet, nous avons choisi de considérer les assertions *login*, *isAuthorized*, *isAuthenticated*, *impliedAuthentication*. Les assertions *login*, *isAuthorized* et *isAuthenticated* vérifient respectivement que l'utilisateur est identifié, autorisé et authentifié. Les motifs à reconnaitre sont simplement les instructions conditionnelles respectivement annotées de l'appel de méthode *login* (Figure 3.1), *isAuthorized* (Figure 3.2) ou *isAuthenticated* (Figure 3.3). L'assertion *impliedAuthentication* vérifie quant à elle que si une authentification est requise, alors l'utilisateur est authentifié ($requireAuthentication \implies isAuthenticated$). Il s'agit d'un pattern de sécurité commun dans WebGoat. Cette assertion est vérifiée si *isAuthenticated* est vraie ou *requireAuthentication* est faux. L'assertion est donc vérifiée sur un chemin quand l'arête vraie de la vérification de *isAuthenticated* ou l'arête fausse de la vérification de *requireAuthentication* est traversée, et le motif à reconnaitre est choisi en conséquence (Figure 3.4).

La simplicité des motifs à reconnaître est permise par la traduction en conditions élémentaires



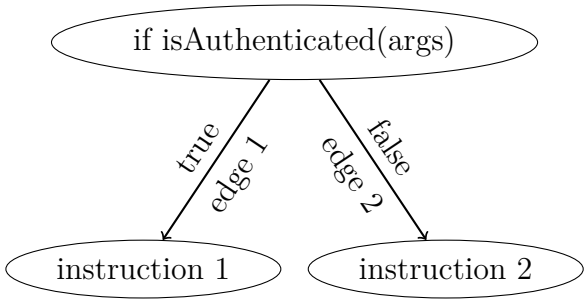
Actions :
Edge 1 → grant login
Edge 2 → ungrant login

Figure 3.1 Motif d'attribution du privilège login



Actions :
Edge 1 → grant authorized
Edge 2 → ungrant authorized

Figure 3.2 Motif d'attribution du privilège authorized



Actions :
Edge 1 → grant authenticated
Edge 2 → ungrant authenticated

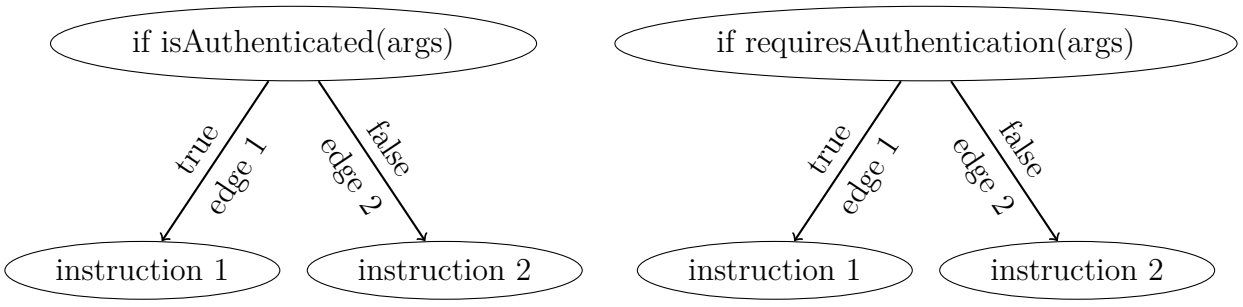
Figure 3.3 Motif d'attribution du privilège authenticated

par l'extraction de Wala.

3.3 Extension des résultats des PIC et des réversions

Lors de l'étude des analyses des PIC, une analyse manuelle des résultats des analyses présentées dans la Section 1.1 est nécessaire afin d'étudier les possibilités de les exploiter pour construire une stratégie de réparation automatique.

L'analyse de PIC proposée en Section 1.1.6 a pour vocation de trouver les modifications de code étant des causes racines des changements de sécurité entre deux versions d'un programme. Elle a été conduite sur les jeux de données décrits en Section 3.2.1. Néanmoins, les différences entre l'ensemble calculé des PIC et celui des causes racines des changements de sécurité [19] montrent que dans certains cas, cet ensemble de modifications ne contient pas



Actions :

Edge 1 → grant impliedAuthentication

Actions :

Edge 2 → grant impliedAuthentication

Figure 3.4 Motif d'attribution du privilège impliedAuthentication

toutes les sources de changements.

En effet, il a été identifié que pour les ensembles WP et MW, respectivement 10/87 (11.5%) et 3/42 (7.1%) [19] des paires de versions présentent une erreur dans les réversions des PICs, montrant que l'ensemble des PIC ne contient pas toutes les causes de l'occurrence de DPD pour ces versions. Les différents cas d'erreur seront étudiés plus en détail en Section 3.4.

3.3.1 Extension des résultats des PIC et des réversions aux CVE

Afin d'étendre les résultats, nous avons appliqué l'analyse des PIC à l'ensemble des CVE décrit en Section 3.2.1.

Au total, 198/1808 (11.0%) changements ont été identifiés comme des PIC lors de l'analyse des 19 versions. 10.42 changements, soit 15.8% de tous les changements, sont identifiés comme des PIC pour chaque version en moyenne.

Afin d'étudier les PIC de cet ensemble, nous avons dans un premier temps appliqué une réversion globale des PIC.

Il résulte de cette expérience que 19/19 (100%) des commits ont été réversés avec succès.

De plus, le nombre de changements dans les paires de versions de cet ensemble étant plus faible (1 021 472.0 changements de LOC pour MW en moyenne, 12 176.2 changements de LOC pour WP en moyenne, et 95.16 changements de LOC pour CVE en moyenne calculés avec l'outil diff de GNU [17]), il est possible de faire une revue manuelle des changements de l'ensemble CVE afin d'évaluer l'analyses des PIC.

M.-A. Laverdière a revu manuellement l'ensemble des changements des paires de l'ensemble

des CVE, et a construit un oracle des modifications causant les DPD. Dans ce mémoire, les termes modifications causant les DPD et causes racines désignent un ensemble de modifications entièrement responsables des DPD entre deux versions.

Lors de la maitrise, nous avons dans un premier temps complété cet oracle avant de le valider en vérifiant qu'une inversion manuelle des modifications de code de cet oracle corrige les DPD. Ainsi, nous nous libérons de l'approximation causée par la granularité par fichier des algorithmes de réversion.

L'oracle content au total 61 modifications, soit 3.21 par version en moyenne.

Une fois l'oracle construit validé, nous avons comparé l'ensemble des modifications de l'oracle avec l'ensemble des PIC calculé.

Nous trouvons donc que 57/61 (93.4%) des éléments de l'oracle sont contenus dans l'ensemble des PIC, et que 57/198 (28.9%) des PIC sont des éléments de l'oracle. Pour 15/19 (78.9%) des versions, l'ensemble des PIC contient entièrement l'ensemble des causes des différences de protection. Les différents ensembles considérés sont représentés dans la Figure 3.5. L'ensemble *allChanges* représente l'ensemble de toutes les modifications de codes entre les deux versions. Le détail des tailles des ensembles calculés est résumé dans le Tableau 3.2. Les colonnes Q1, Q3 et DS désignent respectivement Quartile 1, Quartile 3 et Déviation Standard.

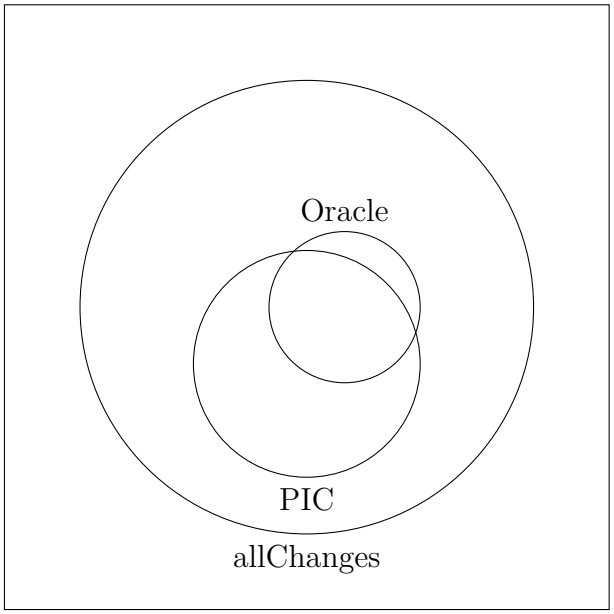


Figure 3.5 Diagramme de Venn des différents ensembles étudiés pour l'ensemble des paires de versions du jeu de données CVE

En d'autres mots, l'analyse identifie 93.4% des causes des différences de protection dans un

Tableau 3.2 Résumé des statistiques de la comparaison entre l'ensemble des PIC et l'ensemble des éléments de l'oracle

Mesures	**Moyenne**	**Q1**	**Médiane**	**Q3**	**DS**
$\|allChanges\|$	95.16	28.00	48.00	85.50	144.76
$\|PIC\|$	10.42	3.00	6.00	7.50	14.93
$\|Oracle\|$	3.21	2.00	2.00	4.50	1.91
$\|Oracle \setminus PIC\|$	0.21	0.00	0.00	0.00	0.41
$\|Oracle \cap PIC\|$	3.00	1.50	2.00	4.50	1.97
$\|PIC \setminus Oracle\|$	7.42	0.00	2.00	4.50	15.17
$\|allChanges \setminus (PIC \cup Oracle)\|$	84.53	21.00	42.00	77.50	130.46

sous-ensemble de 28.9% des changements.

Afin de déterminer si ces résultats sont significatifs et témoignent d'une efficacité de l'analyse des PIC pour identifier les causes racines des différences de protection, nous calculons la valeur-p de l'ensemble des PIC comme outil de recherche de causes racines. Pour ce faire, nous calculons la probabilité pour un ensemble de la taille de l'ensemble des PIC d'identifier au moins autant d'éléments de l'oracle que l'ensemble des PIC. Notez que ce calcul se fait sur l'union des changements des paires de versions.

Cette probabilité étant égale au nombre d'ensembles de modifications de la taille de l'ensemble des PIC contenant au moins autant d'éléments de l'oracle que l'ensemble de PIC (soit la somme des nombre d'ensembles de modifications de la taille de l'ensemble des PIC contenant exactement i éléments avec i allant de $|Oracle \cap PIC|$ à $|Oracle|$) divisé par le nombre d'ensembles de modifications de la taille de l'ensemble des PIC.

$$
\begin{aligned}
p\text{-value} &= \frac{\sum_{i=|Oracle \cap PIC|}^{|Oracle|} \left(\binom{|Oracle|}{i} * \binom{|allChanges \setminus Oracle|}{|PIC| - i} \right)}{\binom{|allChanges|}{|PIC|}} \\
&= \frac{\sum_{i=57}^{61} \left(\binom{61}{i} * \binom{1747}{198 - i} \right)}{\binom{1808}{198}} \\
&\approx 2.1 \times 10^{-53}
\end{aligned}
\tag{3.2}
$$

La valeur-p sur nos données de l'ensemble des PIC est donc de 2.1×10^{-53}. En d'autres termes, un ensemble de la même taille que l'ensemble des PIC construit aléatoirement a une probabilité de 2.1×10^{-53} de contenir au moins autant d'éléments de l'oracle que l'ensemble

des PIC. Cela montre donc que l'ensemble des PIC est efficace pour localiser les éléments de l'oracle.

3.3.2 Application de la réversion individuelle sur les cas d'erreur des réversions

Afin de repérer et de classifier les causes de ces écarts, nous produisons les réversions individuelles en suivant la procédure décrite en Section 1.1.6 pour chaque DPD des paires de versions des ensembles WP et MW pour lesquelles la réversion échouent. Nous révisons ensuite ces résultats afin de distinguer différents cas.

Nous étudions pour cela le jeu de données des paires de versions de WP dont les réversions globales échouent. Nous avons 10 telles paires de versions sur le jeu de données WP, et 3 sur le jeu de données MW.

Sur ce jeu de données WP, 98.2% des réversions individuelles réussissent, et toutes les réversions individuelles réussissent pour 5/10 (50%) des paires de versions. Pour 1/10 (10%) des paires de versions, toutes les réversions individuelles échouent. En moyenne, pour une paire de versions, 91.9% des réversions individuelles réussissent. Sur ce jeu de données MW, toutes les réversions individuelles réussissent.

Nous observons donc une diversité de comportement des paires de versions pour lesquelles la réversion global échoue, qui traduit une diversité des causes d'erreurs de ces réversions globales. Nous utilisons ce résultat ainsi que la revue de ces paires de versions afin de classifier et d'expliquer ces causes.

3.4 Classification des causes d'erreurs des réversions

Nous considérons séparément les paires pour lesquelles au moins une réversion individuelle échoue et ceux pour lesquelles toutes les réversions individuelles réussissent. Lorsqu'au moins une réversion individuelle échoue, celle-ci peut causer l'échec de la réversion complète. Lorsque toutes les réversions individuelles réussissent, la source de l'échec de la réversion complète peut venir d'une potentielle erreur lors de l'union des PIC des différentes DPD, ou de l'inversion du niveau de sécurité d'un noeud n'étant originellement pas une DPD lors de la réversion.

3.4.1 Ecarts dus à la modification implicite d'arcs

Laverdière [19] a relevé un facteur d'erreur de l'analyse des PIC, qui est la possibilité que des arcs (comme des arcs d'appels de fonction) soient modifiés implicitement et que ces modifications ne soient pas considérées lors des analyses.

Par exemple, la modification des propriétés des classes et des méthodes transforme le CFG interprocédural, et peut donc impacter la sécurité définitive des instructions. Néanmoins, comme le CFG ne relie pas ces arcs aux modifications les impactant, ceux-ci ne sont pas reportés comme des PIC.

Comme illustration, dans la paire de versions représentée dans la Figure 3.6, la ligne 2 est sécurisée vis-à-vis du privilège p dans la première version puisque la méthode appelée en ligne 1 octroie le privilège p, mais pas dans la deuxième version, pour laquelle la méthode appelée en ligne 1 n'octroie plus aucun privilège. Les seules lignes modifiées étant les noms des classes, celles-ci ne sont donc pas atteinte par l'analyse d'accessibilité, mais ces changements modifient le graphe d'appel de méthode donc le graphe de flux de contrôle.

```
 1          A::securise()
 2          $a="padding";
 3
 4
 5  -       class A {
 6  +       class B {
 7          function securise(){
 8          if (!current_user_can('p')){
 9                  die();
10          }
11          }
12          }
13
14  -       class B {
15  +       class A {
16          function securise(){
17                  $b="padding";
18          }
19          }
```

Figure 3.6 Exemple d'une erreur de réversion due à un changement d'arc implicite

L'occurrence de ce type d'écart provoque également des erreurs de réversion individuelle, puisque les changements d'arêtes implicites ne seront également pas considérées lors de la traversée des arêtes atteignables à partir de la DPD problématique.

3.4.2 Ecarts dus à la projection du graphe sur le code

Lors de la revue des cas d'erreur des réversions, une autre cause d'erreurs a été identifiée.

Dans certains cas, nous avons observé que la projection des PIC depuis l'ensemble des modifications d'arêtes du CFG sur l'ensemble des modifications de lignes de code pouvait éliminer de l'ensemble des PIC certaines modifications causant des DPD.

En effet, les PIC sont initialement définis comme un sous-ensemble des modifications d'arcs du modèle PTFA (voir Section 1.1.6), mais afin de reporter les PIC comme des données exploitables par les développeurs, ceux-ci sont projetés sur les modifications de lignes de code. Comme détaillé en Section 1.1.6, la projection des arêtes sur l'ensemble des lignes modifiées retourne l'ensemble des lignes modifiées contenant l'origine ou la cible d'une des arêtes. Néanmoins, la cause du changement d'arête peut ne pas être trouvée par la projection de ce changement d'arête.

Par exemple, dans la Figure 3.7, le retrait de la ligne 3 cause une perte de protection définitive en ligne 5, une arête entre la ligne non protégée 2 et la ligne 5 étant ajoutée. Cet ajout d'arête est la cause de la perte de protection, et il est le seul PIC détecté par l'analyse. Néanmoins, la projection de cet ajout d'arête sur les changements de code ne contiendra pas le retrait de la ligne 3, n'étant ni la source, ni la cible ce cette arête. Le PIC sera donc perdu lors de la projection, et l'ensemble de PIC projeté sur les changements de lignes sera donc vide.

```
1          if (!current_user_can('p')){
2                  $a="padding";
3  -               die();
4          }
5          $b="padding";
```

Figure 3.7 Exemple d'une erreur due à la projection des PIC

3.4.3 Effet de bord de la réversion des PIC

Comme dit précédemment, lors de l'analyse des réversions individuelles de DPD des paires de versions de WP, nous avons observé que pour 50% des paires de versions pour lesquelles la réversion globale échouent à inverser les différences de protection, l'ensemble des réversions individuelles d'une DPD réussissent.

Pour les écarts présentés précédemment, dus à l'implémentation des PIC, soit à travers de la modification d'arcs implicites (Section 3.4.1) soit à travers de la projection des PIC sur les modifications de lignes (Section 3.4.2), les réversions individuelles des versions impactées échouent également, puisque les changements causant les DPD sont invisibles vis-à-vis de l'analyse des PIC.

Lors de l'étude de ces cas, un autre type d'anomalie a été observé, et alors que les deux autres sont liées à des choix d'implémentation, celle-ci est directement due au modèle des PIC. Bien que l'analyse des PIC détecte toutes les arêtes dont la modification est la cause d'un changement de sécurité, la réversion de ces modifications peut avoir des effets de bord

sur le niveau de sécurité d'autres instructions, et injecter des vulnérabilités dans l'application. Par exemple, pour le programme en Figure 3.8, les différences de protection définitive entre les deux versions vis-à-vis du privilège p sont donc des gains de protection à l'instruction en ligne 5 du ficher B et le noeud de sortie de la fonction fee. Les seuls PIC entre les deux versions sont donc la suppression d'un arc PTFA $q_{B1,0,0} \rightarrow q_{B5,0,0}$ et l'ajout d'un arc protégé $q_{B2,0,0} \rightarrow q_{B5,0,1}$, puisque les différences d'arcs entre les lignes 2 et 4 du fichier A ne sont pas dans le chemin en amont du gain de protection.

La réversion de ces PIC inverse donc uniquement les modifications du fichier A. Nous remarquons ainsi que ces changements seuls, en plus d'inverser la sécurité des noeuds présentant des différences de protection, inversent également l'instruction en ligne 5 du fichier A. Ainsi, même si les causes racines des différences de sécurité entre les versions A et B ont bien été détectées, leurs réversions ne suffisent pas à réparer les vulnérabilités, et une vulnérabilité est injectée.

File A.php

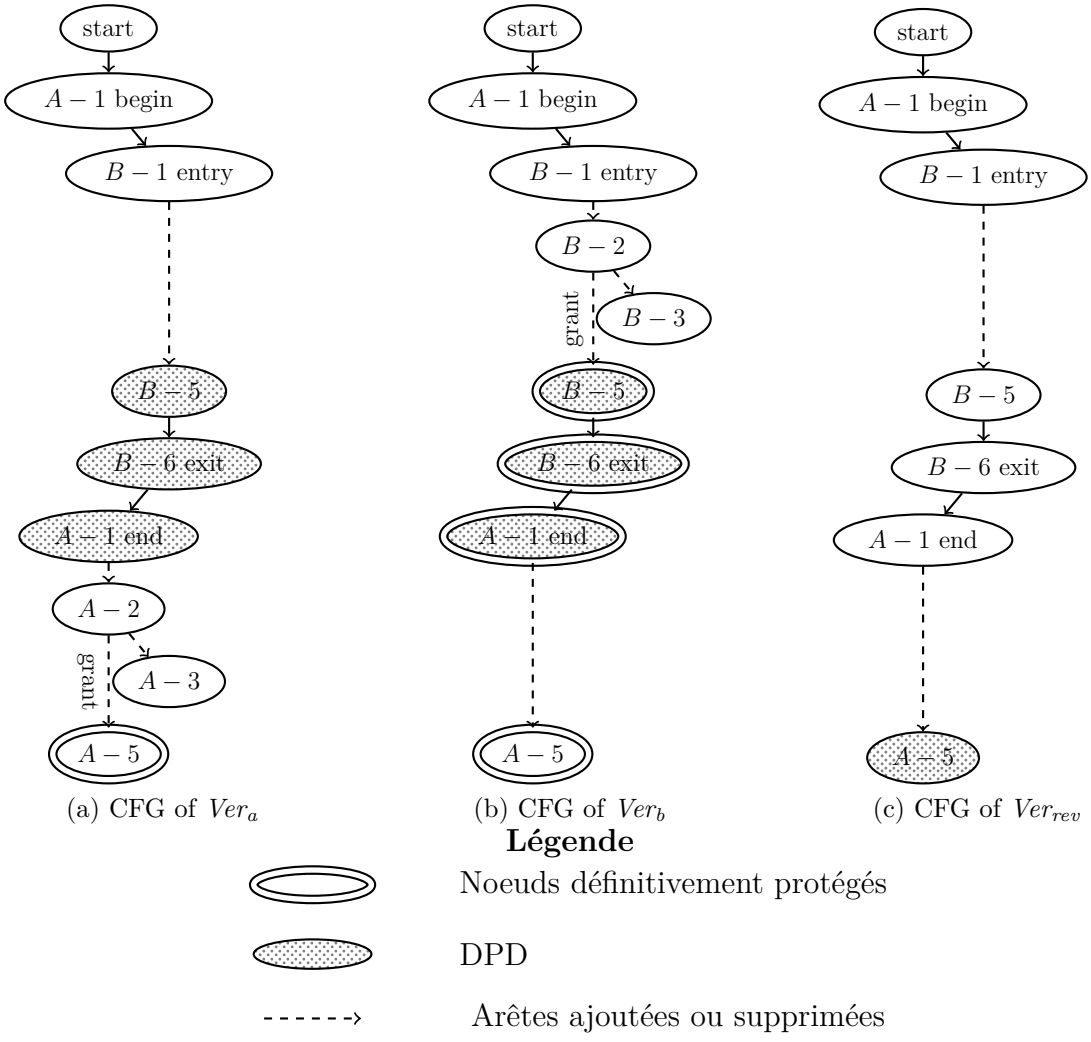
```
1          fee();
2  -       if (!current_user_can('p')){
3  -               die();
4  -       }
5          $a="padding";
```

File B.php

```
1          function fee(){
2  +       if (!current_user_can('p')){
3  +               die();
4  +       }
5          $b="padding";
6          }
```

Figure 3.8 Exemple d'une paire de versions provoquant une erreur du calcul de réversion due à des effets de bord.

Nous avons donc identifié trois types d'écarts possibles entre les PIC calculés et les causes racines. Ces écarts peuvent causer des erreurs avec la fonction de réversions des PIC définie en section 1.1.6, et justifient ainsi la part de versions pour lesquels la réversion ne répare pas les différences de protection (11% des paires de WP et 7% des paires de MW). Les erreurs liées aux modifications d'arcs implicites (Section 3.4.1) et à la projection des PIC sur les changements de lignes (Section 3.4.2) ont pour conséquence que certaines causes des DPD ne sont pas détectées par l'analyse des PIC. Les erreurs liées aux effets de bord (Section 3.4.3) ont pour conséquence que la réversion de certains PIC changent la protection d'autres noeuds.



(a) CFG of Ver_a (b) CFG of Ver_b (c) CFG of Ver_{rev}

Figure 3.9 Représentation de la paire de versions de la Figure 3.8

Dans l'ensemble CVE, 4/61 des éléments de l'oracle des causes racines des DPD n'ont pas été identifié comme des PIC. Parmi ces 4 éléments, 2 sont dus à la projection des PIC sur les changements de lignes, et 2 sont dus à des effets de bord. Néanmoins, ces écarts n'ont pas causé pas d'erreurs dans la réversion des PIC, comme cette réversion est faite avec une granularité par fichier. En effet, chacun des fichiers contenant chacun de ces écarts contiennent au moins un PIC, les éléments de l'oracle sont finalement réversés. Ainsi, 100% des réversions des PIC ont réussi sur l'ensemble CVE. Ce n'est néanmoins pas vrai dans le cas général, comme nous l'avons vu dans les exemples exposés précédemment, et comme nous pouvons le voir dans les ensembles WP et MW, pour lesquels respectivement 89% et 93% des réversions sont réussies.

CHAPITRE 4 RÉPARATION PAR RÉVERSION DES CHANGEMENTS

Dans cette partie, nous cherchons à exploiter la réversion des PIC, introduite en Section 1.1.6 pour réparer automatiquement un programme. Nous utilisons les connaissances acquises sur les cas d'erreur des PIC en Section 3 pour en déduire une stratégie de réparation de code.

4.1 Motivations

Afin de dériver la réversion de PIC en une stratégie de réparation, il convient de corriger les effets de bord induits par la réversion des PIC (Section 3.4.3), et de prendre en compte les autres écarts vus en Sections 3.4.1 et 3.4.2. Nous nous servirons donc de nos observations sur les cas d'écarts entre la fonction de réversion et nos attentes pour une fonction de réparation pour construire une nouvelle fonction.

Tout d'abord, nous avons vu que la réversion des PIC peut causer l'introduction de nouvelles DPD. Afin de corriger ces DPD, l'approche considérée est de réitérer la réversion jusqu'à un point fixe. Dans la Section suivante, nous montrerons notamment la convergence de la réitération de la réversion.

Ensuite, nous avons vu que certains changements peuvent ne pas être considérées pendant l'analyse des PICs, évitant ainsi la réparation des DPD. Nous prendrons ces écarts en compte en proposant une fonction de réversion ne faisant pas intervenir la projection. Si la version n'est toujours pas réparée, nous prendrons la version de référence comme fonction réversée.

Enfin, nous introduisons une stratégie permettant de minimiser les changements causés par la réversion en essayant de retirer des ensembles de changements de cette réversion pour rapprocher la version réparée de la version d'origine.

4.2 Définitions formelles

Afin d'exprimer nos différentes stratégies de réparation par réversion, nous définissons formellement les fonctions de réversion à un haut niveau, pouvant être appliquées pour la réparation d'un prédicat arbitraire, et à une paire de versions arbitraire.

Lors de nos définitions, nous nous référons à l'ensemble des versions par $Vers$. Ses éléments peuvent être des versions existantes d'une application, mais également des versions artificielles obtenues par la réparation d'une version existante. En effet, certaines de nos stratégies nécessitant d'appliquer itérativement des fonctions de réversion, nous avons besoin que nos

définitions soient compatibles avec les versions intermédiaires obtenues après une réversion.

Nos définitions permettent l'application des différentes fonctions sur une paire de versions arbitraire. Néanmoins, en pratique, nous n'aurons pas à les appliquer sur toutes les combinaisons de versions sur lesquelles ces définitions sont valides, mais uniquement sur un sous-ensemble de ces combinaisons, celui des versions successives d'une application.

4.2.1 Notations

Dans nos définitions, nous nous référons aux ensembles **addLin** et **delLin** tels que définis en Section 1.1.5 :

$$\begin{aligned} addLin(A,B) &= Lines(B) \setminus range(lineMap_{A,B}) \\ delLin(A,B) &= Lines(A) \setminus domain(lineMap_{A,B}) \end{aligned} \tag{4.1}$$

En posant A et B deux versions.

On note $\mathbb{B} = \{true, false\}$ l'algèbre de Boole.

Étant donné un prédicat $p : Vers \times Vers \to \mathbb{B}$, pour tout $(A, B) \in Vers \times Vers$, nous posons $p(A, B) \equiv (p(A, B) = true)$ et $\neg p(A, B) \equiv (p(A, B) = false)$.

Soit $\mathcal{E} \in \mathcal{P}(Vers \times Vers)$. On dit que $\mathcal{E}$ est un espace de recherche si et seulement si pour tout $(A, B) \in \mathcal{E}$, $(A, A) \in \mathcal{E}$.

Cet ensemble permet de limiter les fonctions définies aux versions que nous rencontrerons lors de leur application sur un ensemble de test. Lorsque nous créons l'espace de recherche correspondant à un ensemble de test, celui-ci doit contenir les paires de versions que l'on recherche à reverser, mais également les paires formées avec les versions antérieures et les versions réversées possibles.

Étant donnée une paire de versions de $\mathcal{E}$, (A, B) que nous cherchons à réverser, nous nous réfèrerons à la version A comme la version antérieure, et à la version B comme la version postérieure. Cette notation n'a pas d'implication sur la chronologie de la production ou de la sortie des deux versions du projet. La version antérieure est considérée comme la version de référence, que l'on considère comme correcte, et la version postérieure est la version considérée comme potentiellement vulnérable, et c'est la version que nous cherchons à réparer.

4.2.2 Définitions des fonctions de réversion et relations entre elles

Définition 4.2.1. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive (i.e. $p(A, A)$ est vraie pour tout $(A, A) \in \mathcal{E}$). Une fonction $f_p : \mathcal{E} \to Vers$ est une fonction de p-revert si et seulement si elle*

vérifie les axiomes suivants :

(i) Pour tout $(A, B) \in \mathcal{E}$*, en notant* $R = f_p(A, B)$ *:*

$$\begin{cases} \textbf{\textit{addLin}}(A, R) \subseteq & \textbf{\textit{addLin}}(A, B) \wedge \\ \textbf{\textit{delLin}}(A, R) \subseteq & \textbf{\textit{delLin}}(A, B) \end{cases} \tag{4.2}$$

(ii) $\forall (A, B) \in Vers \times Vers, p(A, B) \implies f_p(A, B) = B$

(iii) $\forall (A, B) \in \mathcal{E}, (A, f_p(A, B)) \in \mathcal{E}$

En d'autres termes, une fonction de réversion est une fonction qui, avec en entrée une paire de versions, retourne une nouvelle version dont la différence avec la version antérieure est un sous-ensemble des différences entre les versions antérieure et postérieure. Notez que la fonction est définie sur l'ensemble de recherche $\mathcal{E}$. Ces versions peuvent être deux versions distinctes d'un même projet, mais également des versions induites par une de nos stratégies de réversion. En particulier, la paire de versions en entrée n'est pas nécessairement une paire de versions successives.

Notez que la définition de fonction p-revert dépend explicitement du prédicat p utilisé. Étant donnés deux prédicats p_1 et p_2, une fonction p_1-revert n'est pas une fonction de p_2-revert. Nous utilisons le terme "fonction de réversion" pour désigner une fonction de p-revert lorsqu'il n'y a pas d'ambiguïté sur le prédicat p utilisé. Nous appelons le résultat d'une telle fonction sur une paire de versions la version réversée.

Propriété 4.2.1. *Soit* $(A, B, R) \in Vers^3$

$$\begin{cases} \textbf{\textit{addLin}}(A, R) \subseteq & \textbf{\textit{addLin}}(A, B) \wedge \\ \textbf{\textit{delLin}}(A, R) \subseteq & \textbf{\textit{delLin}}(A, B) \end{cases} \iff \begin{cases} \textbf{\textit{addLin}}(R, B) \subseteq & \textbf{\textit{addLin}}(A, B) \wedge \\ \textbf{\textit{delLin}}(R, B) \subseteq & \textbf{\textit{delLin}}(A, B) \end{cases} \tag{4.3}$$

Démonstration. Supposons que :

$$\begin{cases} \textbf{addLin}(A, R) \subseteq & \textbf{addLin}(A, B) \wedge \\ \textbf{delLin}(A, R) \subseteq & \textbf{delLin}(A, B) \end{cases} \tag{4.4}$$

On a :

$$\begin{aligned}
\mathbf{addLin}(R,B) &\subseteq (\mathbf{addLin}(R,A) \cup \mathbf{addLin}(A,B)) \setminus \mathbf{delLin}(A,B) \\
&\subseteq (\mathbf{delLin}(A,R) \cup \mathbf{addLin}(A,B)) \setminus \mathbf{delLin}(A,B) \\
&\subseteq \mathbf{addLin}(A,B) \setminus \mathbf{delLin}(A,B) \\
&\qquad\qquad (\text{car } \mathbf{delLin}(A,R) \subseteq \mathbf{delLin}(A,B)) \\
&\subseteq \mathbf{addLin}(A,B)
\end{aligned} \tag{4.5}$$

De même, $\mathbf{delLin}(R,B) \subseteq \mathbf{delLin}(A,B)$

Donc

$$\begin{cases} \mathbf{addLin}(A,R) \subseteq & \mathbf{addLin}(A,B) \wedge \\ \mathbf{delLin}(A,R) \subseteq & \mathbf{delLin}(A,B) \end{cases} \Longrightarrow \begin{cases} \mathbf{addLin}(R,B) \subseteq & \mathbf{addLin}(A,B) \wedge \\ \mathbf{delLin}(R,B) \subseteq & \mathbf{delLin}(A,B) \end{cases} \tag{4.6}$$

Supposons que :

$$\begin{cases} \mathbf{addLin}(R,B) \subseteq & \mathbf{addLin}(A,B) \wedge \\ \mathbf{delLin}(R,B) \subseteq & \mathbf{delLin}(A,B) \end{cases} \tag{4.7}$$

On a :

$$\begin{aligned}
\mathbf{addLin}(A,R) &= \mathbf{delLin}(R,A) \\
&\subseteq (\mathbf{delLin}(R,B) \cup \mathbf{delLin}(B,A)) \setminus \mathbf{addLin}(B,A) \\
&\subseteq (\mathbf{delLin}(R,B) \cup \mathbf{delLin}(B,A)) \setminus \mathbf{delLin}(A,B) \\
&\subseteq \mathbf{delLin}(B,A) \setminus \mathbf{delLin}(A,B) \\
&\qquad\qquad (\text{car } \mathbf{delLin}(R,V_b) \subseteq \mathbf{delLin}(A,B)) \\
&\subseteq \mathbf{delLin}(B,A) \\
&\subseteq \mathbf{addLin}(A,B)
\end{aligned} \tag{4.8}$$

De même, $\mathbf{delLin}(A,R) \subseteq \mathbf{delLin}(A,B)$

Donc

$$\begin{cases} \mathbf{addLin}(R,B) \subseteq & \mathbf{addLin}(A,B) \wedge \\ \mathbf{delLin}(R,B) \subseteq & \mathbf{delLin}(A,B) \end{cases} \Longrightarrow \begin{cases} \mathbf{addLin}(A,R) \subseteq & \mathbf{addLin}(A,B) \wedge \\ \mathbf{delLin}(A,R) \subseteq & \mathbf{delLin}(A,B) \end{cases} \tag{4.9}$$

□

Cette propriété nous offre la possibilité, pour les démonstrations suivantes, de démontrer qu'une fonction est une fonction de réversion en utilisant la deuxième équation de l'équiva-

lence plutôt que la première (correspondant à l'axiome 4.2.1.(i)).

Nous définissons maintenant la validité d'une fonction de p-revert, afin de formaliser l'objectif des fonctions de réversions, qui est de réparer la version postérieure par rapport à la version antérieure au regard du prédicat p.

Définition 4.2.2. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive. Une fonction de p-revert f_p est dite valide si et seulement si pour tout $(A, B) \in \mathcal{E}$, $p(A, f(A, B))$ est vrai.*

En d'autres termes, une fonction de p-revert est valide si et seulement si le prédicat p est vérifié entre la version antérieure et la version réversée.

Nous définissons ensuite des relations permettant de comparer plusieurs fonctions de réversion entre elles.

Tout d'abord, nous définissons une relation comparant la quantité de changements réversés par les fonctions.

Définition 4.2.3. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive. On définit la relation d'ordre $\prec$ sur l'ensemble des fonctions de p-revert ainsi :*

Pour toutes fonctions de p-revert f_p et f'_p :

$$f \prec f'_p \iff \left(\forall (A, B) \in \mathcal{E}, \begin{cases} \boldsymbol{addLin}(f_p(A, B), B) \subseteq \boldsymbol{addLin}(f'_p(A, B), B) & \wedge \\ \boldsymbol{delLin}(f_p(A, B), B) \subseteq \boldsymbol{delLin}(f'_p(A, B), B) \end{cases} \right) \tag{4.10}$$

Démonstration. La réflexivité et la transitivité de la relation $\prec$ sont directement induites par la réflexivité et la transitivité de la relation d'inclusion.

Pour montrer l'antisymétrie, considérons deux fonctions de p-revert f_p et f'_p tels que $f_p \prec f'_p$ et $f'_p \prec f_p$.

$$\forall (A, B) \in \mathcal{E}, \begin{cases} \mathbf{addLin}(f_p(A, B), B) \subseteq \mathbf{addLin}(f'_p(A, B), B) & \wedge \\ \mathbf{delLin}(f_p(A, B), B) \subseteq \mathbf{delLin}(f'_p(A, B), B) & \wedge \\ \mathbf{addLin}(f'_p(A, B), B) \subseteq \mathbf{addLin}(f_p(A, B), B) & \wedge \\ \mathbf{delLin}(f'_p(A, B), B) \subseteq \mathbf{delLin}(f_p(A, B), B) \end{cases} \tag{4.11}$$

D'où, par antisymétrie de la relation d'inclusion,

$$\forall (A, B) \in \mathcal{E}, \begin{cases} \mathbf{addLin}(f_p(A, B), B) = \mathbf{addLin}(f'_p(A, B), B) \wedge \\ \mathbf{delLin}(f_p(A, B), B) = \mathbf{delLin}(f'_p(A, B), B) \end{cases} \tag{4.12}$$

D'où

$$\forall (A, B) \in \mathcal{E}, f_p(A, B) = f'_p(A, B) \tag{4.13}$$

Et finalement, $f_p = f'_p$

Donc la relation $\prec$ est antisymétrique, et $\prec$ est une relation d'ordre.

□

$f_p \prec f'_p$ signifie donc que f'_p réverse toujours un sous-ensemble des éléments réversés par f_p.

Ensuite, nous définissons une relation permettant de comparer les ensembles des versions que les fonctions de réversion permettent de réparer.

Définition 4.2.4. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive. On définit la relation de préordre $\vdash$ sur l'ensemble des fonctions de p-revert ainsi :*

$$f_p \vdash f'_p \iff \Big(\ \forall (A, B) \in \mathcal{E}, \Big(\ p(A, f_p(A, B)) \implies p(A, f'_p(A, B)) \ \Big) \Big) \tag{4.14}$$

Démonstration. La réflexivité et la transitivité de cette relation sont des conséquences directes de la réflexivité et de la transitivité de l'implication. □

$f_p \vdash f'_p$ signifie donc que f'_p permet de réparer un sur-ensemble des versions réparées par f_p.

4.2.3 Réitération des fonctions de réversions

En prenant $noDiff$ le prédicat tel que pour tout $(A, B) \in \mathcal{E}$, $noDiff(A, B) \iff DPD(A, B) = \emptyset$, la fonction de réversion des PIC rev_{lines} définie en Section 1.1.6 est donc une fonction de $noDiff$-revert. En effet, l'absence de DPD implique l'absence de PIC, et la construction de l'algorithme de réversion en Section 1.1.6 vérifie les relations d'inclusion.

En Section 3.4.3, nous avons vu que la réversion des PIC pouvait avoir des effets de bord et provoquer de nouvelles DPD.

Pour une paire de versions $(A, B) \in \mathcal{E}$ dont la réversion présente des effets de bord, on a $\neg noDiff(A, rev(A, B))$. De plus, les DPD entre A et $rev(A, B)$ n'étant pas incluses dans les DPD entre A et B, il est possible que les DPD introduites soient associées à des PIC n'ayant pas encore été réversés, donc appartenant aux changements entre A et $rev(A, B)$. On peut alors s'attendre à ce qu'une réitération de la réversion des PIC sur la nouvelle paire de versions puisse en réparer une partie.

Dans cette Section, nous définissons une stratégie motivée par cette observation, consistant à réitérer la fonction de réversion jusqu'à l'obtention d'un point fixe, réparant ainsi les DPD introduites pouvant l'être dans le cas de la réversion au regard du privilège *noLoss*. Nous démontrons également la convergence de cette réitération.

Définition 4.2.5. *Soient $p : Vers \times Vers \to \mathbb{B}$ une relation réflexive et f_p une fonction de p-revert. Pour tout $n \in \mathbb{N}^*$, soit $f_p^n : Vers \times Vers \to Vers$ définie récursivement ainsi :*

$$\forall (A, B) \in \mathcal{E} \begin{cases} f_p^0(A, B) = B \\ \forall k \in \mathbb{N}, f_p^{k+1}(A, B) = f_p(A, f_p^k(A, B)) \end{cases} \tag{4.15}$$

Propriété 4.2.2. *Soient $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive, f_p une fonction de p-revert et $n \in \mathbb{N}$.*

f_p^n est une fonction de p-revert et $f_p \vdash f_p^n$.

Démonstration. Par la Définition 4.2.1, f_p^n est une fonction de p-revert si il satisfait 4.2.1.(i) et 4.2.1.(ii).

L'axiome 4.2.1.(i) est satisfait par la transitivité de l'inclusion. L'axiome 4.2.1.(ii) est satisfait par la construction récursive de f_p^n.

Enfin, l'axiome 4.2.1.(iii) est satisfait par la transitivité de l'égalité.

f_p^n est donc une fonction de p-revert.

$f_p \vdash f_p^n$ par récurrence et par 4.2.1.(ii) appliqué sur f_p □

En d'autres termes, la réitération d'une fonction de réversion répare au moins autant de paires de versions que la fonction de réversion elle-même.

Nous montrons maintenant que la réitération d'une fonction de réversion est stationnaire sur un point fixe. Pour ce faire, on montre que les ensembles des modifications entre A et $f_p^n(A, B)$ forment une suite décroissante (au sens de l'inclusion) d'ensembles finis, donc sont stationnaires.

Propriété 4.2.3. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation récursive et f_p une fonction de p-revert.*

$\forall (A, B) \in \mathcal{E}, \exists n \in \mathbb{N} \mid \forall m \geq n, f_p^m(A, B) = f_p^n(A, B)$

Démonstration. Soit $(A, B) \in \mathcal{E}$.

Pour tout $n \in \mathbb{N}$, par 4.2.1.(i), et comme $f_p^{n+1}(A, B)) = f_p(A, f_p^{n+1}(A, B))$, on a :

$$\begin{cases} \mathbf{addLin}(A, f_p^{n+1}(A,B)) \subseteq & \mathbf{addLin}(A, f_p^{n}(A,B)) \wedge \\ \mathbf{delLin}(A, f_p^{n+1}(A,B)) \subseteq & \mathbf{delLin}(A, f_p^{n}(A,B)) \end{cases} \tag{4.16}$$

Donc $(\mathbf{addLin}(A, f_p^{n+1}(A,B)))_{n\in\mathbb{N}}$ et $(\mathbf{delLin}(A, f_p^{n+1}(A,B)))_{n\in\mathbb{N}}$ sont des suites décroissantes (en terme d'inclusion) d'ensembles finis, donc elles sont stationnaires. En effet, la suite de leurs cardinaux est décroissante dans $\mathbb{N}$, et $\leq$ étant un ordre bien fondé sur $\mathbb{N}$, elle est stationnaire, et deux ensembles de même cardinal dont l'un est inclus dans l'autre sont égaux.

Donc il existe un entier $n \in \mathbb{N}$ tel que $\mathbf{addLin}(A, f_p^m(A,B)) = \mathbf{addLin}(A, f_p^n(A,B))$ et $\mathbf{delLin}(A, f_p^m(A,B)) = \mathbf{delLin}(A, f_p^n(A,B))$ pour tout $m \geq n$, et donc $f_p^n(A,B) = f_p^m(A,B)$ comme les lignes ajoutées et supprimées par rapport à la version A déterminent entièrement une version.

□

Définition 4.2.6. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive et f_p une fonction de p-revert. On définit $f_p^\infty : \mathcal{E} \to Vers$ et $n_\infty : \mathcal{E} \to \mathbb{N}$ comme :*

$$\begin{aligned} &\forall (A,B) \in \mathcal{E}, \\ &\quad n_\infty(A,B) = \min(\{n \in \mathbb{N} \mid f_p^n(A,B) = f_p^{n+1}(A,B)\}) \\ &\quad f_p^\infty(A,B) = f_p^{n_\infty(A,B)}(A,B) \end{aligned} \tag{4.17}$$

Ce minimum existe par la Propriété 4.2.3.

Propriété 4.2.4. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive et f_p une fonction de p-revert.*

f_p^∞ est une fonction de p-revert et $f_p \vdash f_p^\infty$.

Démonstration. La propriété est une conséquence directe de la Propriété 4.2.2 □

f_p^∞ désigne donc le point fixe de la réitération des fonctions de réversion.

Dans le cas des PIC, puisque pour toute paire de versions (A,B), $rev_{lines}(A, rev_{lines}^\infty(A,B)) = rev_{lines}^\infty(A,B)$, donc par conséquent $allPIC_{lines}(A, f_p^\infty(A,B)) = \emptyset$. En d'autres mots, soit le point fixe n'a pas de DPD avec la version antérieure, soit les DPD n'ont pas de PIC associés. Si le point fixe n'a pas de DPD avec la version antérieure, alors les DPD sont réparées. Sinon, les causes des DPD présentes sont des modifications d'arcs implicites ou sont des modifications n'ayant pas été retenues par la projection, comme décrit en Section 3.4. Dans les deux cas, les erreurs dues à des effets de bords sont corrigées si leurs causes racines ne rentrent pas dans ces cas d'erreur.

4.2.4 Mesures des fonctions de réversion

Définition 4.2.7. *Soient $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive et f_p une fonction de p-revert. $\overline{f} : \mathcal{E} \to Vers$ est défini ainsi :*

$$\begin{cases} \forall A, B \in Vers, \\ \qquad \overline{f_p}(A,B) = f_p(A,B) \quad si\ p(A, f_p(A,B)) \\ \qquad \overline{f_p}(A,B) = A \qquad\qquad\qquad\qquad sinon \end{cases} \tag{4.18}$$

Propriété 4.2.5. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive et f_p une fonction de p-revert. $\overline{f_p}$ est une fonction de p-revert valide.*

Démonstration. Soit $(A, B) \in \mathcal{E}$.

Si $p(A, f_p(A, B))$, alors $\overline{f_p}(A, B) = f_p(A, B)$. Par conséquence, et par la Définition 4.2.1 :

- $\begin{aligned} \mathbf{addLin}(A, \overline{f_p}(A,B)) &= \mathbf{addLin}(A, f_p(A,B)) \\ &\subseteq \mathbf{addLin}(A,B) \end{aligned}$
- $\begin{aligned} \mathbf{delLin}(A, \overline{f_p}(A,B)) &= \mathbf{delLin}(A, f_p(A,B)) \\ &\subseteq \mathbf{delLin}(A,B) \end{aligned}$
- $\begin{aligned} p(A,B) &\implies f_p(A,B) = B \\ &\implies \overline{f_p}(A,B) = B \end{aligned}$
- $\begin{aligned} p(A, \overline{f_p}(A,B)) &= p(A, f_p(A,B)) \\ &= true \end{aligned}$

Sinon, alors $\overline{f_p}(A, B) = A$. Par conséquence :

- $\mathbf{addLin}(A, \overline{f_p}(A,B)) = \mathbf{addLin}(A,B)$
- $\mathbf{delLin}(A, \overline{f_p}(A,B)) = \mathbf{delLin}(A,B)$
- $p(A,B) \implies \overline{f_p}(A,B)) = B$
- $\begin{aligned} p(A, \overline{f_p}(A,B)) &= p(A,A) \\ &= true \text{ (as p is reflexive)} \end{aligned}$

Donc, les axiomes, 4.2.1.(i) et 4.2.1.(ii) ainsi que la condition de validité de la Définition 4.2.2 sont satisfaits, et $\overline{f_p}$ est une fonction de p-revert valide. □

La fonction $\overline{f_p}$ est une fonction de réversion qu'on force à être valide, en remplaçant toutes les versions qui n'ont pas pu être réparées par la version antérieure. L'introduction de cette

fonction permet de comparer les performances de fonctions de réversion pouvant être non valides. Nous l'utiliserons dans les définitions de mesures afin de calculer le poids des réversions, en assignant un poids maximal aux versions que la fonction ne parvient pas à réparer.

Nous introduisons deux mesures évaluant l'efficacité d'une fonction de réversion sur un ensemble de paires de versions. Le taux de succès mesure le taux de paires de versions que la fonction réverse correctement. Le taux de réversion mesure la distance syntaxique entre la version que nous souhaitons réverser et la version réversée.

Définition 4.2.8. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive et f_p une fonction de p-revert. On définit S_{f_p}, le taux de succès de f_p sur un ensemble fini de paires de versions ainsi :*

$$\forall V \in VersPairsSet, S_{f_p}(V) = \frac{|\{(A,B) \in V \mid p(A, f_p(A,B))\}|}{|V|} \tag{4.19}$$

Avec $VersPairsSet = \{V \in \mathcal{P}(Vers \times Vers) \mid |V| < \infty\}$

Définition 4.2.9. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive et f_p une fonction de p-revert. On définit σ_{f_p} le taux de réversion de f_p sur une paire de versions comme :*

$$\forall (A,B) \in \mathcal{E}, \sigma_{f_p}((A,B)) = \frac{|\boldsymbol{addLin}(\overline{f_p}(A,B),B)| + |\boldsymbol{delLin}(\overline{f_p}(A,B),B)|}{|\boldsymbol{addLin}(A,B)| + |\boldsymbol{delLin}(A,B)|} \tag{4.20}$$

σ_{f_p} représente donc le taux de changements que nous devons réverser pour corriger le programme avec une fonction de réversion f_p. En utilisant la fonction $\overline{f_p}$, nous forçons la fonction dont nous mesurons le poids à être valide, et nous obtenons ainsi les changements réellement nécessaires pour réparer les versions avec cette fonction. En d'autres termes, σ_{f_p} est une mesure prenant en compte le taux de succès de f_p et le taux de modifications réversées par f_p sur les paires de versions correctement réversées.

Notez que l'objectif est de trouver une fonction de réversion f_p telle que σ_{f_p} soit aussi faible que possible, puisque cela signifie que la version réparée est syntaxiquement plus proche de la version postérieure du couple. Pour la mesure de taux de réussite S_{f_p}, notre objectif est qu'il soit aussi élevé que possible.

Dans son application à la fonction rev_{lines} de réversion des PIC, le taux de réversion peut être mis en parallèle avec le taux de PIC [18] introduit notamment en Section 1.3. Néanmoins, celui-ci expose le taux de changements à réverser plus fidèlement que le taux de PIC, puisqu'il considère les changements réellement réversés, donc tous les changements appartenant à un fichier contenant au moins un PIC. De plus, il considère uniquement les réversions réellement réussies, comme décrit précédemment.

Nous démontrons maintenant que le taux de succès S et le taux de réversion σ sont meilleurs pour f_p^n que pour f_p. Pour ce faire, pour chaque paire de versions (A, B), nous séparons les cas où $p(A, f_p(A, B))$ est vrai des cas où $p(A, f_p(A, B))$ est faux, et nous montrons que dans les deux cas f_p^n donne des résultats ayant de meilleures mesures que f_p.

Propriété 4.2.6. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive, f_p une fonction de p-revert, $V \in VersPairsSet, (A, B) \in \mathcal{E}$ et $n \geq 2$.*

$S_{f_p^n}(V) \geq S_{f_p}(V)$ *et* $\sigma_{f_p^n}(A, B) \leq \sigma_{f_p}(A, B)$.

Démonstration. Si $p(A, f_p(A, B))$, alors

Par la propriété 4.2.2, f_p^{n-1} est une fonction de p-revert, et par récursion $f_p^n(A, B) = f_p^{n-1}(A, f_p(A, B))$.

Donc $f_p^n(A, B) = f_p^{n-1}(A, f_p(A, B)) = f_p^{n-1}(A, B)$ (car $p(A, f_p(A, B))$). Par récursion, on trouve $f_p^n(A, B) = f_p(A, B)$, et donc $p(A, f_p^n(A, B))$ est vraie.

D'où $\sigma_{f_p^n}(A, B) = \sigma_{f_p}(A, B)$.

Sinon $\neg p(A, f_p(A, B))$, donc :

$$\begin{aligned} \sigma_{f_p}(A, B) &= 1 \\ &\geq \sigma_{f_p^n}(A, B) \end{aligned} \tag{4.21}$$

Dans tous les cas, $\sigma_{f_p}(A, B) \geq \sigma_{f_p^n}(A, B)$ et $p(A, f_p(A, B)) \implies p(A, f_p^n(A, B))$.

Par conséquence, $S_{f_p^n}(V) \geq S_{f_p}(V)$.

□

4.2.5 Stratégie Incrémentielle

Dans cette Section, nous visons à introduire une stratégie améliorant le taux de réversion d'une fonction de réversion valide en essayant d'enlever des ensembles de modifications à la réversion avant de vérifier que ce changement n'impacte pas la validité de la fonction. Nous réduisons ainsi la quantité de changements que la fonction doit réverser.

Dans un premier temps, nous définissons une fonction de sélection comme une fonction permettant de générer une séquence finie de sous-ensembles de changements de lignes entre paire de versions (A, B).

Définition 4.2.10. *On définit une fonction de sélection comme une fonction s telle que pour toute paire de versions (A, B), $s(A, B) = (n, (Add_i)_{i \leq n}, (Del_i)_{i \leq n})$, où $n \in$*

$[\![1; |\boldsymbol{addLin}(A,B)| + |\boldsymbol{addLin}(A,B)|]\!]$, *$(Add_i)_{i\leq n}$ est une séquence de n sous-ensembles de* $\boldsymbol{addLin}(A,B)$ *(i.e. $\forall i \leq n, Add_i \in \mathcal{P}(\boldsymbol{addLin}(A,B))$) et $(Del_i)_{i<n}$ est une séquence de n sous-ensembles de* $\boldsymbol{delLin}(A,B)$ *(i.e. $\forall i \leq n, Del_i \in \mathcal{P}(\boldsymbol{delLin}(A,B))$).*

Un exemple de telle fonction est la fonction associant à une paire de versions les séquences des changements appartenant à un même fichier, donc où en posant $(n, (Add)_{i\leq n}, (Del)_{i\leq n}) = s(A,B)$, n est le nombre de fichiers modifiés et Add_i et Del_i sont respectivement les lignes ajoutées et supprimées du i-ème fichier.

L'objectif de la stratégie incrémentielle est de vérifier successivement si les éléments des séquences finies produites par une fonction de sélection impactent la réussite de la réversion, pour retirer de la réversion des ensembles de changements non nécessaires à sa réussite, et ainsi réduire la quantité de changements à réverser. Pour ce faire, nous définissons la fonction $Incr(f,s)$ correspondante à la stratégie incrémentielle appliquée à la fonction de réversion f au regard de la fonction de sélection s.

Définition 4.2.11. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive, f_p une fonction de p-revert, et s une fonction de sélection.*

Soit $(A,B) \in \mathcal{E}$. On pose $(n, (Add_i)_{i\leq n}, (Del_i)_{i\leq n}) = s(A,B)$.

Si $p(A, f_p(A,B))$, on définit (F_i) et (G_i) ainsi :

$$\begin{aligned} F_0 &= f_p(A,B) \\ G_0 &= F_0 \end{aligned} \tag{4.22}$$

Pour $1 \leq i \leq n$, nous posons F_i tel que :

$$\begin{aligned} \boldsymbol{addLin}(F_i,B) &= \boldsymbol{addLin}(G_{i-1},B) \setminus Add_i \\ \boldsymbol{delLin}(F_i,B) &= \boldsymbol{delLin}(G_{i-1},B) \setminus Del_i \end{aligned} \tag{4.23}$$

Ces équations définissent F_i puisqu'une version peut être entièrement définie à partir des lignes ajoutées et supprimées par rapport à une autre version.

Et nous posons G_i tel que :

$$G_i = \begin{cases} F_i & si\ (A,F_i) \in \mathcal{E} \wedge p(A,F_i) \\ G_{i-1} & sinon \end{cases} \tag{4.24}$$

On pose alors $Incr(f_p,s)(A,B) = G_n$.

Si $\neg p(A, f_p(A,B))$, on pose $Incr(f_p,s)(A,B) = f_p(A,B)$.

La taille n des séquences d'ensembles de changements générés par la fonction de sélection doit être choisie pour faire un compromis entre le taux de réversion que nous souhaitons obtenir, qui sera meilleur pour un n élevé, puisque nous pourrons approcher l'ensemble des causes racines avec une plus fine granularité. Néanmoins, la stratégie incrémentielle nécessite autant d'applications de la fonction de réversion que d'éléments de la séquence contenant des modifications réversées par la fonction originale.

Pour une paire de versions (A, B) pour laquelle la réversion est réussie, la stratégie incrémentielle consiste donc à soustraire successivement les ensembles des séquences $(Add_i)_{i\leq n}$ et $(Del_i)_{i\leq n}$ d'une réversion réussie, afin de déterminer si le retrait d'un ensemble de modification de l'ensemble des modifications réversées impacte ou non la réussite de la réversion. Si l'ensemble testé n'impacte pas la réussite de la réversion, nous le retirons des modifications réversées par $Incr(f_p, s)$, améliorant ainsi le taux de réversion sans impacter le taux de réussite. Si l'ensemble testé n'impacte pas la réussite de la réversion, nous ne le retirons pas des modifications réversées par $Incr(f_p, s)$. Pour tout $i \leq n$, la version F_i représente la version pour laquelle nous observons la valeur du prédicat p avec la version A pour déterminer si Add_i et Del_i peuvent être retirés des ensembles des modifications à réverser, et G_i représentent l'état de la version reversée après les i premières incrémentations. A la fin du processus, G_n est donc la version réversée réduite par la stratégie incrémentielle.

Pour synthétiser, les ensembles de modifications réversées par $Incr(f_p, s)$ sont des sous-ensembles des ensembles des modifications réversées par f_p, auxquels nous avons retiré successivement des ensembles de modifications n'ayant pas d'impact sur la protection.

Propriété 4.2.7. *Soit $p : \mathcal{E} \to \mathbb{B}$ une relation réflexive, f_p une fonction de p-revert et s une fonction de sélection. Alors $Incr(f_p, s)$ est une fonction de p-revert vérifiant $f_p \vdash Incr(f_p, s)$ et $f_p \succ Incr(f, s)$.*

Démonstration. Soit A et B tels que $p(A, f_p(A, B))$.

On définit n ainsi que les versions $(G_i)_{i\leq n}$ conformément à la Définition 4.2.11.

De plus, par construction, $G_0 = f_p(A, B)$, et pour tout $i < n$, $\mathbf{addLin}(G_{i+1}, B) \subseteq \mathbf{addLin}(G_i, B)$ et $\mathbf{delLin}(G_{i+1}, B) \subseteq \mathbf{delLin}(G_i, B)$.

D'où par récurrence et par transitivité de l'inclusion :

$$\begin{aligned}\mathbf{addLin}(Incr(f_p, s)(A, B), B) =& \mathbf{addLin}(G_n, B)\\ \subseteq& \mathbf{addLin}(f_p(A, B), B)\\ \subseteq& \mathbf{addLin}(A, B)\end{aligned}$$

(par la définition 4.2.1.(i) appliquée à f_p et la propriété 4.2.1) (4.25)

et

$$\begin{aligned}\mathbf{delLin}(Incr(f_p, s)(A, B), B) =& \mathbf{delLin}(G_n, B)\\ \subseteq& \mathbf{delLin}(f_p(A, B), B)\\ \subseteq& \mathbf{delLin}(A, B)\end{aligned}$$

(par la définition 4.2.1.(i) appliquée à f_p et la propriété 4.2.1) (4.26)

D'où, par l'équation 4.26, $Incr(f_p, s)$ vérifie l'axiome 4.2.1.(i).

De plus, si $p(A, B)$, alors $f_p(A, B) = B$ comme f_p est une fonction de p-revert. Par conséquent, $\mathbf{delLin}(f_p(A, B), B) = \emptyset$ et $\mathbf{addLin}(f_p(A, B), B) = \emptyset$, donc $\mathbf{delLin}(Incr(f_p, s)(A, B), B) = \emptyset$ et $\mathbf{addLin}(Incr(f_p, s)(A, B), B) = \emptyset$ par l'équation 4.26, et donc $Incr(f_p, s)(A, B) = B$. Donc $Incr(f_p, s)$ vérifie l'axiome 4.2.1.(ii).

Donc $Incr(f_p, s)$ est une fonction de p-revert, et par l'équation 4.26, $f_p \succ Incr(f_p, s)$.

Enfin, en posant $(A, B) \in \mathcal{E}$, et en définissant n ainsi que les versions $(G_i)_{i \leq n}$ conformément à la Définition 4.2.11, on sait que $p(A, G_0)$ (car $G_0 = F_0 = f_p(A, B)$), et que pour tout $i < n$, $p(A, G_i) \implies p(A, G_{i+1})$ par construction.

Donc par récurrence et par transitivité de l'implication, $p(A, G_n)$, et donc $p(A, Incr(f_p)(A, B))$.

D'où $f \vdash Incr(f_p, s)$. □

Notez que les versions réversées $Incr(f_p, s)$ ne sont pas nécessairement optimales, et dépendent notamment de l'ordre des séquences de sous-ensembles de changements générées pas s. Néanmoins, la Propriété 4.2.7 garantit que ces versions sont au moins aussi correctes que celles réversées par f_p, et que $Incr(f_p, s)$ réverse au plus autant de modifications que f_p. La recherche d'une solution optimale nécessiterait a priori le calcul d'une version pour chaque combinaison d'éléments des séquences générées par s, alors que notre approche ne calcule

qu'une version pour chaque élément de ces séquences.

4.3 Implémentation des fonctions

Nos fonctions de réversion ayant une granularité par fichier, étant donné un jeu de données D, on note $\mathcal{E}$ l'ensemble des paires composées des versions antérieures des paires de D et des versions intermédiaires entre les versions antérieures et leurs versions postérieures associées, c'est à dire où les fichiers sont sélectionnés entre les deux versions pour donner une nouvelle version de l'espace de recherche.

En Section 1.1.6, l'ensemble des PIC est défini comme l'ensemble des changements d'arêtes identifiés comme impactant la protection. Une projection de ces PIC sur les changements de lignes est également définie. Nous appelons PIC_{edges} l'ensemble des PIC et PIC_{lines} cette projection de l'ensemble des PIC. Comme décrit en Section 3.3, cette projection peut rater certains changements de lignes responsables des changements d'arêtes.

Une première fonction de réversion, que nous appelons rev_{lines}, a été définie en Section 1.1.6. Celle-ci consiste à inverser les fichiers contenant un élément de PIC_{lines}. Nous définissons également une fonction de réversion rev_{edges}, consistant à inverser les fichiers contenant les sources ou les cibles des éléments de PIC_{edges} (définis comme des ensembles d'arêtes en Section 1.1.6). Ces deux fonctions sont des fonctions de $noDiff$-revert.

Par définition, l'ensemble des fichiers réversés par rev_{edges} est un sur-ensemble des fichiers réversés par rev_{lines}. La réussite ou non de la réversion peut différer entre les deux fonctions.

Nous appliquons alors les définitions de la Section 4.2.2 aux fonctions de réversion définies en Section 1.1.6. Nous avons ainsi 4 stratégies de réversion induites pour chaque fonction de réversion rev donnée : rev, $\overline{rev}$, rev^{∞} et $\overline{rev^{\infty}}$. Nous présentons donc le processus de l'application de chacune de ces stratégies. Chacun de ces processus peut être appliqué à $rev = rev_{edges}$ et à $rev = rev_{lines}$.

— La chaîne correspondant aux fonctions rev est présentée en Figure 4.1. Elle retourne simplement le résultat de la fonction de réversion utilisée. Il n'y a aucune garantie sur la réussite de la réversion, puisque le prédicat $LossGain$ n'est pas vérifié à la sortie du processus.

— La chaîne correspondant aux fonctions $\overline{rev}$ est présentée en Figure 4.2. Elle calcule le résultat de rev, le retourne si le prédicat est vérifié, et retourne la Version A sinon. Puisque le prédicat est vérifié à la sortie, et par la Propriété 4.2.5, cette fonction de réversion est valide.

— La chaîne correspondant aux fonctions rev^{∞} est présentée en Figure 4.3. Cette stratégie réitère la fonction de réversion rev jusqu'à l'obtention d'un point fixe. Ce point fixe existe par 4.2.3, donc le processus termine. Néanmoins, il n'y a aucune garantie sur la réussite de la réversion, puisque le prédicat $LossGain$ n'est pas vérifié à la sortie du processus.

— La chaîne correspondant aux fonctions $\overline{rev^{\infty}}$ est présentée en Figure 4.4. Cette stratégie est similaire à celle de rev^{∞}, mais si le prédicat n'est pas vérifié avec le résultat final, la Version A est retournée. Puisque le prédicat est vérifié à la sortie, et par la Propriété 4.2.5, cette fonction de réversion est valide.

Nous implémentons également la stratégie incrémentielle, en prenant comme fonction de sélection s_{file}, partitionnant les changements d'une paire de versions (A, B) selon le fichier contenant le fichier. En posant $(n, (Add_i)_{i\leq n}, (Del_i)_{i\leq n}) = s_{file}(A, B)$, Add_i et Del_i sont donc respectivement les ensembles des ajouts et des suppressions de lignes du i-ème fichier de l'application.

Ainsi, la stratégie consiste à rétablir un à un les fichiers de la réversion afin d'identifier ceux qui ne modifient pas la sécurité, et peuvent donc être rétablis pour diminuer le poids σ de la réversion en gardant la réversion valide.

Notez que la complexité temporelle de l'application de cette stratégie à une fonction de réversion f sur une paire de versions (A, B) est égale à celle de la stratégie de réversion f multipliée par le nombre de partitions à tester, donc le nombre de fichiers de B réversés par f. En effet, si pour un fichier i $addLin(f(A, B), B) \cap Add_i = \emptyset$ et $delLin(f(A, B), B) \cap Del_i = \emptyset$, alors par la Définition 4.2.11, la version obtenue à la i-ème itération est la même que la précédente ($F_i = G_{i-1}$ avec les notations de la définition). Les seuls fichiers pour lesquels une opération de réversion est nécessaire pour l'application de la stratégie sont donc ceux réversés par la fonction f.

4.4 Résultats

Nous avons appliqué notre stratégie basée sur les réversions aux ensembles paires de versions de WP, MW et CVE présentés en Section 3.2.1 en comparant les réversions simples et multiples (rev et rev^{∞}) afin de mesurer son apport.

Pour chaque ensemble considéré, nous appliquons notre stratégie avec une réversion basée sur les PIC au niveau des lignes (après projection), mais également avec une réversion basée sur les PIC au niveau des arêtes (avant projection). Nous comparons les résultats de ces deux fonctions de réversion, et les réversions réitérées avec les réversions uniques.

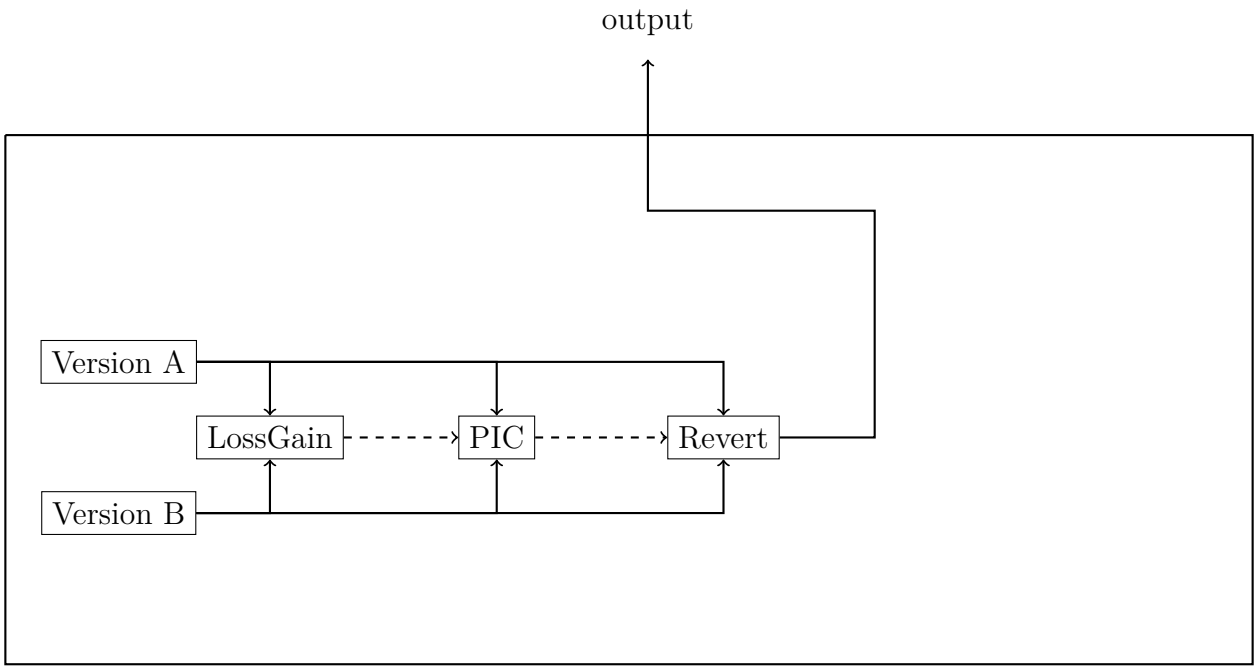


Figure 4.1 Représentation du processus de calcul de rev

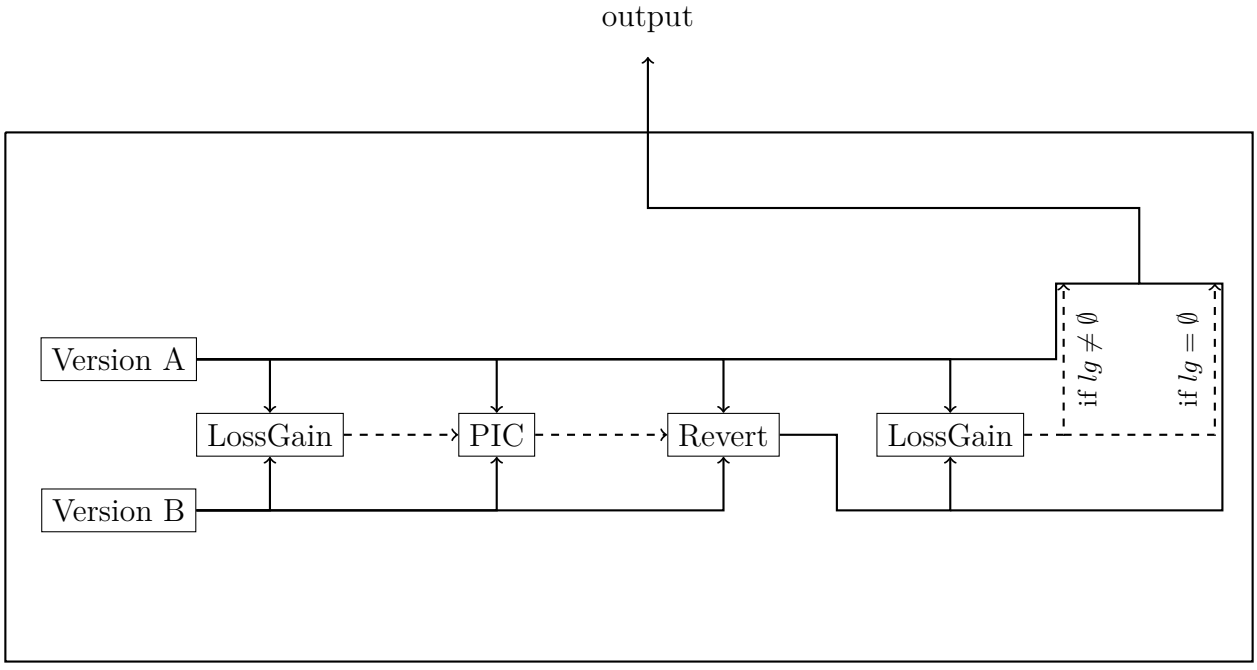


Figure 4.2 Représentation du processus de calcul de $\overline{rev}$

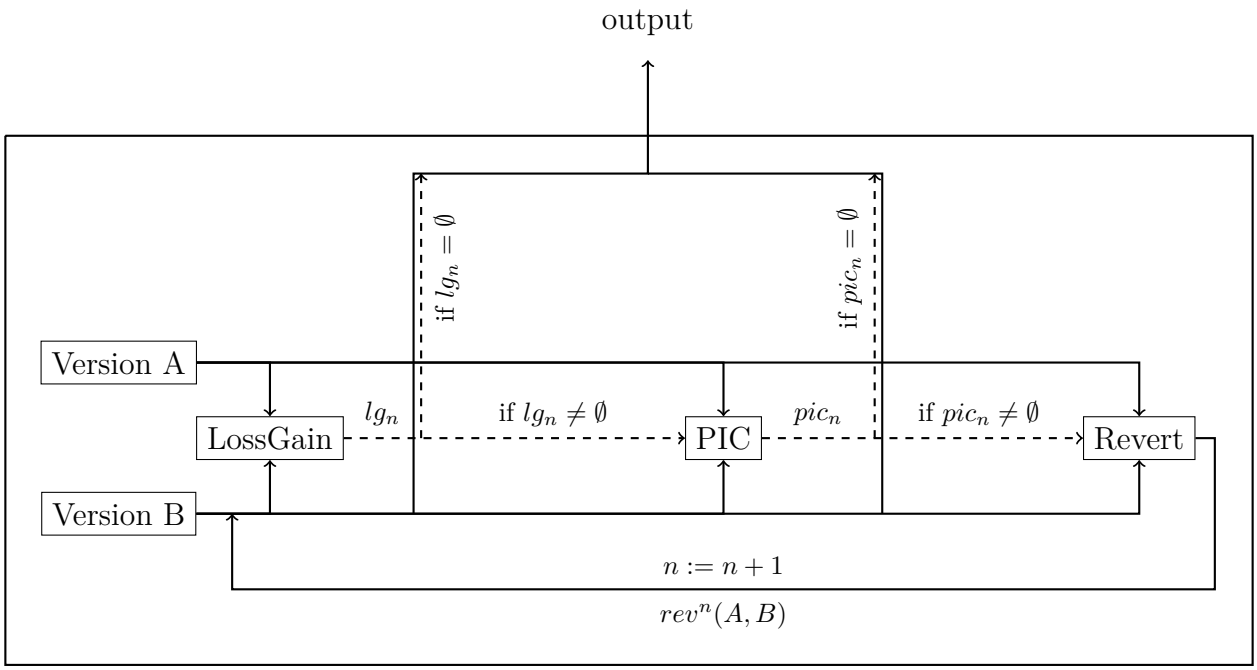


Figure 4.3 Représentation du processus de calcul de rev^{∞}

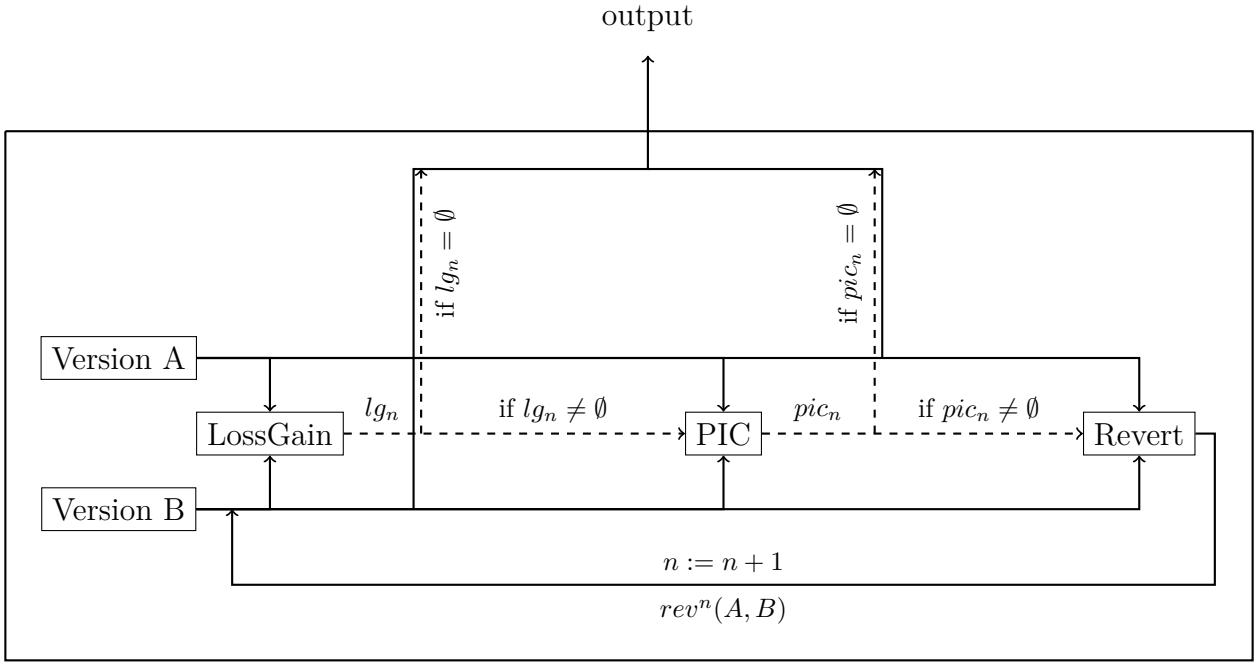


Figure 4.4 Représentation du processus de calcul de $\overline{rev^{\infty}}$

Comme vu en Section 4.3, la stratégie incrémentielle devant calculer une réversion pour chaque fichier réversé par la fonction de base (donc ici pour chaque fichier contenant des PIC), elle est très coûteuse. En effet, il y a autant d'analyses à réaliser que de partitions considérées, donc dans notre cas de fichiers réversés par la fonction de réversion de base, c'est-à-dire de fichiers contenant des PIC. Le nombre moyen de fichiers contenant des PIC est de 2.26 pour l'ensemble CVE, de 63.89 pour l'ensemble WP, et de 469.88 pour l'ensemble MW. Pour cette raison, nous choisissons d'appliquer cette stratégie uniquement sur l'ensemble CVE.

Pour chaque ensemble étudié, nous calculons enfin le taux de succès ainsi que la moyenne du taux de réversion.

Tableau 4.1 Mesures des résultats des différentes fonctions de réversion

(a) Mesures des résultats de la fonction rev_{lines}

	WP	MW	CVE
$S_{rev_{lines}}$	88.51%	92.86%	100%
mean($\sigma_{rev_{lines}}$)	65.80%	40.06%	43.47%
med($\sigma_{rev_{lines}}$)	75.78%	41.26%	37.84%

(b) Mesures des résultats de la fonction rev^{∞}_{lines}

	WP	MW	CVE
$S_{rev^{\infty}_{lines}}$	95.40%	92.86%	100%
mean($\sigma_{rev^{\infty}_{lines}}$)	63.52%	40.06%	43.47%
med($\sigma_{rev^{\infty}_{lines}}$)	70.29%	41.26%	37.84%

(c) Mesures des résultats de la fonction rev_{edges}

	WP	MW	CVE
$S_{rev_{edges}}$	90.80%	100%	100%
mean($\sigma_{rev_{edges}}$)	69.78%	36.56%	44.26%
med($\sigma_{rev_{edges}}$)	80.96%	37.83%	37.84%

(d) Mesures des résultats de la fonction rev^{∞}_{edges}

	WP	MW	CVE
$S_{rev^{\infty}_{edges}}$	100%	100%	100%
mean($\sigma_{rev^{\infty}_{edges}}$)	67.48%	36.56%	44.26%
med($\sigma_{rev^{\infty}_{edges}}$)	79.06%	37.83%	37.84%

(e) Mesures des résultats de la fonction $Incr(rev_{lines}, s_{file})$

	WP	MW	CVE
$S_{Incr(rev_{lines},s_{file})}$	-	-	100%
mean($\sigma_{Incr(rev_{lines},s_{file})}$)	-	-	34.19%
med($\sigma_{Incr(rev_{lines},s_{file})}$)	-	-	19.80%

Notez que la mesure σ_f prend en compte le taux de réussite de la réversion, puisqu'il considère la version antérieure comme version étudiée. En d'autres termes, σ_f désigne la part de changements à réverser pour avoir un taux de réussite de 100%.

La fonction de réversion rev_{lines} ne prend en compte aucun des écarts mentionnés en Section 3.3.

La fonction de réversion rev^{∞}_{lines} corrige les écarts dus aux effets de bord. Les réversions réussies par rev_{lines} sont conservées, et certaines réversions défaillantes de rev_{lines} sont corrigées. Cet apport concerne 6.89% des paires de l'ensemble des paires de versions de WP, et aucune

paire des autres ensembles.

La fonction de réversion rev_{edges} corrige les écarts dus à la projection des PIC. Cependant, comme celle-ci considère un sur-ensemble des changements considérés par la fonction rev_{lines}, celle-ci a un poids moyen et médian plus important sur l'ensemble des paires de versions de WordPress. Néanmoins, grâce à la correction de l'écart des projections et au plus grand nombre de patchs réussis par cette fonction, son poids moyen et médian est plus faible que celui de rev_{lines} sur l'ensemble des paires de versions de MW.

La fonction de réversion rev^{∞}_{edges} corrige les écarts dus aux effets de bord ainsi que les écarts dus à la projection. Elle a ainsi un taux de réussite de 100% sur les trois ensembles étudiés.

Enfin, la stratégie incrémentielle réduit la moyenne du taux de fichiers à réverser de 22.75%, et la médiane de 47.67% pour l'ensemble CVE.

Pour les paires de versions des ensembles de données étudiés, le calcul des fonctions de réversion réitérées rev^{∞}_{edges} et rev^{∞}_{lines} ne dépassent pas deux itérations. En d'autres termes, $rev^{\infty}_{edges_{|E}} = rev^{2}_{edges_{|E}}$ et $rev^{\infty}_{lines|E} = rev^{2}_{line|E}$ pour tout $E \in \{\mathrm{WP}, \mathrm{MW}, \mathrm{CVE}\}$. Toutefois, il est possible que pour d'autres jeux de données, deux itérations ne suffisent pas.

CHAPITRE 5 RÉPARATION PAR INSERTION DE MOTIFS

Dans cette partie, nous proposons une fonction de réparation des pertes de protection définitive. Nous nous concentrons donc sur l'ensemble $DefLoss(A, B)$ défini en Section 1.1.5, pour un unique privilège donné. Pour réparer plusieurs privilèges, plusieurs exécutions successives de l'algorithme peuvent être appliquées sur le programme.

Ce choix est justifié par le fait que l'injection d'une vulnérabilité dans une version se traduit par l'accès à une instruction par un utilisateur qui n'y avait pas accès dans la version antérieure, donc dans d'autres mots par une perte de protection définitive. De plus, alors qu'il est aisé de supprimer un chemin non protégé, il est compliqué d'inverser un gain de privilège en ajoutant un chemin non protégé sans conséquence sur le comportement du programme.

Les définitions des fonctions de réparation présentées dans cette partie sont construites en parallèle des définitions proposées dans le Chapitre 4.

5.1 Motivations

Dans cette partie, nous recherchons des réparations en considérant un espace de recherche différent de celui des réparations par réversion. En introduisant des motifs dans le code au lieu d'inverser des changements, nous évitons de supprimer des fonctionnalités que le développeur a introduit lors des changements, et nous aboutissons à une version réparée plus proche de la version originelle.

Nous définirons ainsi plusieurs stratégies visant à réparer toutes les pertes de protections tout en réduisant le nombre de réparations effectuées.

5.2 Définitions formelles

5.2.1 Notations

Pour les différentes définitions et démonstrations de ce chapitre, nous utiliserons les notations suivantes :

Pour une version $V \in Vers$, nous notons $CFG(V)$ le CFG de la version, défini comme le couple de l'ensemble des noeuds et de l'ensemble d'arêtes du graphe en Section 1.1.2. Nous notons $PTFA(V)$ le modèle PTFA de la version, défini comme un tuple en Section 1.1.3. Nous notons $Reachable(PTFA(V))$ l'ensemble des états et des transitions PTFA accessibles depuis

le noeud de départ, comme défini en Section 1.1.4. Nous notons $Vertexes(V)$ l'ensemble des noeuds du CFG de V, et $Edges(V)$ l'ensemble des arêtes du CFG de V.

Étant donné un état PTFA $q_{i,j,k}$, on note $vertex(q_{i,j,k}) = i$, $context(q_{i,j,k}) = j$ et $prot(q_{i,j,k}) = k$.

Pour T un ensemble de transitions du modèle PTFA d'une version V, nous notons $Paths(T) = Paths(V)$ l'ensemble des chemins construits par des transitions de T, où la cible d'une transition est la source de la prochaine transition dans le chemin.

Nous représentons un chemin avec la notation $(p_0, \ldots, p_i, \ldots, p_n)$ avec $(p_i)_{i \leq n}$ des états du graphe PTFA. Notez qu'un chemin noté ainsi peut ne pas avoir de sens dans le modèle si les paires $(p_i, p_{i+1})_{i<n}$ ne sont pas des transitions du modèle, donc si $(p_0, \ldots, p_i, \ldots, p_n) \notin Paths(T)$.

Pour une version $V \in Vers$ et un noeud $c \in Vertexes(V)$, on note $interpred_V(c) = \{c' \mid (c', c) \in Edges(V)\}$ l'ensemble des prédécesseurs du noeud c dans le CFG.

On note un espace de recherche $\mathcal{E} \in \mathcal{P}(Vers \times Vers)$ contenant les paires à réverser ainsi que les réparations possibles que notre modèle peut générer.

5.2.2 Définition générale d'une fonction de réparation

Dans un premier temps, nous définissons à haut niveau une fonction de réparation d'une version, qu'on appelle version postérieure d'un programme au regard d'un prédicat la comparant à une autre version, qu'on appelle version antérieure. Cette définition nous servira à définir les objectifs que nous voulons fixer pour notre stratégie de réparation par insertion de motif.

Cette fonction est définie sur l'espace de recherche défini précédemment. Elle pourra donc être appliquée pour tous les couples dont la version antérieure est une version de référence, et la version postérieure est une réparation candidate pour la version à réparer correspondante à cette version de référence.

Définition 5.2.1. *Soit $p : \mathcal{E} \to \mathbb{B}$ un prédicat. Une fonction $f : \mathcal{E} \to Vers$ est une fonction de p-repair si :*

(i) $\forall (A, B) \in \mathcal{E}$,

$$domain(VertexMap_{A,f(A,B)}) \subseteq domain(VertexMap_{A,B}) \tag{5.1}$$

(ii) $\forall (A, B) \in \mathcal{E}$,

$$p(A, B) \implies f(A, B) = B \tag{5.2}$$

(iii) $\forall (A, B \in \mathcal{E})$,

$$(A, f(A, B)) \in \mathcal{E} \tag{5.3}$$

Nous définissons une fonction de réparation valide comme une fonction de réparation générant un patch vérifiant le prédicat avec la version antérieure. Cette notion de fonction de p-repair valide est analogue à celle de p-revert valide définie en Section 4.2.2.

Définition 5.2.2. *Soit f une fonction de p-repair. f est une fonction de p-repair valide si et seulement si $\forall (A, B) \in \mathcal{E}, p(A, f(A, B))$ est vrai.*

De la même manière que pour les fonctions de réversion de p, nous pouvons réitérer une fonction de réparation de p sur une paire de versions. Nous montrons que la réitération d'une fonction de p-repair est également une fonction de p-repair, en vérifiant qu'elle satisfait les axiomes d'une fonction de p-repair. Ainsi, en réappliquant la fonction de réparation, nous pourrons réparer des erreurs que la première réparation aurait manquée, comme pour les fonction de p-revert.

Définition 5.2.3. *Soit $p : \mathcal{E} \to \mathbb{B}$ un prédicat et f une fonction de p-repair. Pour tout $n \in \mathbb{N}$, on définit $f^n : \mathcal{E} \to Vers$ récursivement :*

$$\forall (A, B) \in \mathcal{E} \begin{cases} f^0(A, B) = B \\ \forall k \in \mathbb{N}, f^{k+1}(A, B) = f(A, f^k(A, B)) \end{cases} \tag{5.4}$$

Propriété 5.2.1. *Soit $p : \mathcal{E} \to \mathbb{B}$ un prédicat, f une fonction de p-repair et $n \in \mathbb{N}$.*

f^n est une fonction de p-repair.

Démonstration. Soit $n \in \mathbb{N}$. Par la Définition 5.2.1, f^n est une fonction de réparation si elle satisfait 5.2.1.(i) et 5.2.1.(ii).

Par l'axiome 5.2.1.(i) appliqué à la fonction de réparation f, pour tout $k \in \mathbb{N}$, pour tout $(A, B) \in \mathcal{E}$, on a :

$$\begin{aligned} domain(VertexMap_{A,f^k+1(A,B))}) &= domain(VertexMap_{A,f(A,f^k(A,B))}) \\ &\subseteq domain(VertexMap_{A,f^k(A,B)}) \\ &\quad \text{(car } f \text{ est une fonction de } p\text{-repair)} \end{aligned} \tag{5.5}$$

Donc par récurrence, on a

$$domain(VertexMap_{A,f^n(A,B)}) \subseteq domain(VertexMap_{A,f^0(A,B)}) \quad \subseteq domain(VertexMap_{A,B}) \tag{5.6}$$

D'où 5.2.1.(i) est vérifié pour la fonction f^n

On sait que 4.2.1.(ii) est valide pour la fonction de réparation f.

Supposons que 4.2.1.(ii) est valide pour f^k, avec $k \in \mathbb{N}$. On a :

$$\begin{aligned} p(A,B) &\implies f^k(A,B) = B \\ &\implies p(A, f^k(A,B)) \\ &\implies f^{k+1}(A,B) = f(A, f^k(A,B)) = f^k(A,B) = B \end{aligned} \tag{5.7}$$

D'où, par récurrence, 5.2.1.(ii) est vérifié pour la fonction f^n.

Donc f^n est donc une fonction de p-repair. □

5.2.3 Construction d'une fonction de réparation des DPD

Pour utiliser nos définitions, nous devons préciser le prédicat que nous voulons réparer. Dans la suite de cette partie, nous travaillons donc sur la réparation du prédicat *noLoss* défini ainsi :

Définition 5.2.4. *Soit noLoss le prédicat défini ainsi :*

$$\forall (A,B) \in \mathcal{E}, noLoss(A,B) = (DefLoss(A,B) = \emptyset) \tag{5.8}$$

Pour réaliser une fonction de *noLoss*-repair valide, notre stratégie consiste à localiser les arêtes que nous devons protéger avec un privilège, et de définir une action de protection de ces arêtes.

Nous définissons ainsi ces deux fonctions à haut niveau. Toutes les définitions de cette section sont relatives au prédicat *noLoss*.

Une fonction de localisation retourne un ensemble d'arêtes de la version postérieure, qui sont les arêtes que nous cherchons à sécuriser. Si la version postérieure est déjà correcte par rapport à la version antérieure et au regard du prédicat *noLoss*, la fonction doit retourner l'ensemble vide, puisqu'aucune correction n'est nécessaire.

Définition 5.2.5. *Une fonction $l : \mathcal{E} \to Edges$ est une fonction de localisation si elle vérifie les axiomes suivants :*

(i) $\forall (A, B) \in \mathcal{E}$,

$$l(A, B) \subseteq E_B \tag{5.9}$$

(ii) $\forall (A, B) \in \mathcal{E}$,

$$noLoss(A, B) \implies l(A, B) = \emptyset \tag{5.10}$$

Une fonction de garde est une fonction qui, étant donnés une paire de versions et un ensemble d'arêtes, crée une nouvelle version où les arêtes de l'ensemble sont remplacées par des motifs de sécurité.

Afin de proposer une approche indépendante du langage utilisé ou de l'application à réparer, nous ne fixons pas le motif à insérer, mais nous encadrons son application en définissant des axiomes sur l'application du motif sur un ensemble d'arêtes (donc sur la fonction de garde) afin de nous assurer qu'il répare bien les propriétés désirées tout en ne changeant pas le programme plus que nécessaire. Ces axiomes nous serviront à démontrer la validité de la fonction de réparation construite.

Définition 5.2.6. *Soit* $g : \{(B, S) | B \in Vers, S \in \mathcal{P}(Edges(B))\} \to Vers$. *g est une fonction de garde si pour tout* $(A, B) \in \mathcal{E}$, *en posant* $(V_b, E_b) = CFG(B), (Q_b, T_b, q_{0b}, Var_b, G_b, A_b) = PTFA(B)$, *les axiomes suivants sont vérifiés :*

(i)

$$g(B, \emptyset) = B \tag{5.11}$$

(ii) $\forall S \in \mathcal{P}(E_b)$
Soit $G = g(B, S)$

$$domain(VertexMap_{B,G}) = V_b \tag{5.12}$$

(iii) $\forall S \in \mathcal{P}(E_b)$
Soit $G = g(B, S)$

$$\forall (u_0, u_1) \in E_b \setminus S, (VertexMap_{B,G}(u_0), VertexMap_{B,G}(u_0)) \in Edges(G) \tag{5.13}$$

(iv) $\forall S \in \mathcal{P}(E_b)$

Soit $G = g(B, S)$, $VM = VertexMap_{B,G}$ *et* $(V_g, E_g) = CFG(G)$

$$\begin{gathered}\forall (c_0, c_1) \in E_g \mid c_1 \in range(VM),\\ (c_0 \in range(VM) \wedge (VM^{-1}(c_0), VM^{-1}(c_1)) \in E_b) \vee\\ \Big(c_0 \notin range(VM) \wedge \big(\forall p = (p_1, \ldots, p_i, \ldots, p_n) \in Paths(G) \mid\\ (vertex(p_1) \in range(VM) \wedge vertex(p_n) = c_1 \wedge vertex(p_{n-1}) = c_0 \wedge\\ (\forall 1 < j < n, vertex(p_j) \notin range(VM))),\\ (VM^{-1}(vertex(p_1)), VM^{-1}(vertex(p_n))) \in S\big)\Big)\end{gathered} \tag{5.14}$$

(v) $\forall S_1 \subseteq E_b$, *on pose* $G = g(B, S_1)$, $VM = VertexMap_{B,G}$
$\forall S_2 \subseteq E_b \mid S_1 \cap S_2 = \emptyset \wedge (\forall (c_0, c_1) \in S_2, c_0 \in domain(VM) \wedge c_1 \in domain(VM) \wedge (VM(c_0), VM(c_1) \in Edges(G)$
On pose $S_2' = \{(VM(c_0), VM(c_1)) \mid (c_0, c_1) \in S_2\}$

$$g(g(B, S_1), S_2') = g(B, S_1 \cup S_2) \tag{5.15}$$

(vi) $\forall S \in \mathcal{P}(E_b)$
Soit $G = g(B, S)$
Soient $(V_G, E_G) = CFG(G)$ *et* $(Q_G, T_G, q_{0G}, Var_G, A_G) = PTFA(G)$
Soit $VM = VertexMap_{B,G}$
$\forall j, k \in \{0, 1\}$
$\forall (c_0, c_1) \in S$
$\forall p = (p_0, \ldots, p_i, \ldots, p_n) \in Paths(T_G) \mid \forall i < n, vertex(p_i) \notin range(VM)$

$$(q_{VM(c_0),j,k}, p_0, \ldots, p_i, \ldots, p_n, q_{VM(c_1),j',0}) \notin Paths(T_R) \tag{5.16}$$

(vii) $\forall S \in \mathcal{P}(E_b)$
Soit $G = g(B, S)$
Soient $(V_G, E_G) = CFG(G)$ *et* $(Q_G, T_G, q_{0G}, Var_G, A_G) = PTFA(G)$
Soit $VM = VertexMap_{B,G}$
$\forall j, k \in \{0, 1\}$
$\forall (c_0, c_1) \in S$

$$\exists p = (p_0, \ldots, p_i, \ldots, p_n) \in Paths(T_G) \mid \forall i' < n, vertex(p_{i'}) \notin range(VM) \wedge \\ (q_{VM(c_0),j,k}, p_0, \ldots, p_i, \ldots, p_n, q_{VM(c_1),j,1}) \in Paths(T_G) \tag{5.17}$$

(viii) $\forall S \in \mathcal{P}(E_b)$

$$VertexMap_{A,g(B,S)} = VertexMap_{B,g(B,S)} \circ VertexMap_{A,B} \tag{5.18}$$

(ix) $\forall S \in \mathcal{P}(E_b), (A, g(B, S)) \in \mathcal{E}$

Pour résumer, nos exigences sur la fonction de garde sont les suivantes :

— L'application du motif sur un ensemble vide d'arête ne doit pas modifier le programme (i).

— L'application du motif ne doit pas supprimer des noeuds du CFG (ii). Cet axiome limite l'impact de la réparation sur le comportement du programme.

— L'application du motif ne doit pas supprimer d'arêtes autres que les arêtes sur lesquelles il est appliqué (iii). Cet axiome limite l'impact de la réparation sur le comportement du programme.

— L'application du motif sur un ensemble d'arêtes n'ajoute des arêtes qu'entre les noeuds des arêtes sur lesquelles il est appliqué (iv).

— L'application successive du motif sur deux ensembles d'arêtes disjoints est équivalent à l'application du motif sur l'union de ces deux ensembles (v). Cet axiome nous permettra notamment de réitérer la fonction de réparation.

— L'application du motif sur des arêtes enlève la possibilité pour un chemin de traverser ces arêtes en restant non protégé (vi). Cet axiome garantit qu'un chemin ne peut pas passer par les motifs de sécurité insérés sans être protégé à sa sortie.

— L'application du motif n'impacte pas la fonction de comparaison avec la version de référence (viii). Cet axiome garantit qu'on compare bien les protections des mêmes noeuds entre les versions réparée et antérieure qu'entre les versions postérieure et antérieure.

— L'application du motif sur des arêtes laisse la possibilité pour un chemin de traverser ces arêtes. (vii). Cet axiome permet de s'assurer qu'on peut traverser le motif de sécurité en étant protégé, et limite donc l'impact de la réparation sur le comportement du programme.

— L'espace de recherche est stable par l'application d'un motif (ix)

Pour résumer, la fonction de garde protège les chemins d'exécution passant par les arêtes que l'on désire protéger sans impacter le autres chemins.

Une fois les fonctions de localisation et de réparation fixées, nous pouvons les combiner pour obtenir une fonction de réparation en appliquant la fonction de garde à l'ensemble des arêtes identifiées par la fonction de localisation.

Définition 5.2.7. *Soit l une fonction de localisation, g une fonction de garde. On définit $R_{l,g}$ la fonction de réparation associée à l et g ainsi*

$$\forall (A, B) \in \mathcal{E}, R_{l,g}(A, B) = g(B, l(A, B)) \tag{5.19}$$

Propriété 5.2.2. *Soit l une fonction de localisation, et g une fonction de garde.*

$R_{l,g}$ est une fonction de noLoss-repair.

Démonstration. Soit l une fonction de localisation et g une fonction de garde.

Soit $(A, B) \in \mathcal{E}$.

Par 5.2.6.(ix), $R_{l,g}$ vérifie 5.2.1.(iii).

Par 5.2.6.(viii), on a :

$$VertexMap_{A,g(B,l(A,B))} = VertexMap_{B,g(B,l,A,B)} \circ VertexMap_{A,B} \tag{5.20}$$

Et donc $domain(VertexMap_{A,R_{l,g}(A,B)}) \subseteq domain(VertexMap_{A,B})$

D'où $R_{l,g}$ vérifie 5.2.1.(i).

De plus, si $noLoss(A, B)$, par 5.2.5.(ii), $l(A, B) = \emptyset$, donc par 5.2.6.(i) :

$$R_{l,g}(A, B) = g(B, l(A, B)) = g(B, \emptyset) = B \tag{5.21}$$

Donc $R_{l,g}(A, B)$ vérifie 5.2.1.(ii), et $R_{l,g}(A, B)$ est bien une fonction de *noLoss*-repair. □

Nous établissons maintenant des contraintes à vérifier pour la fonction de localisation afin de construire une fonction de réparation valide.

La première contrainte est la non redondance de la fonction de localisation. Cette contrainte garantit qu'en réitérant la fonction $R_{l,g}$, les arêtes localisées par les différentes itérations sont

toutes différentes, et sont toutes des arêtes de la version postérieure initiale, et non des arêtes qui ont été rajoutées dans les fonctions précédentes.

Définition 5.2.8. *Soit l une fonction de localisation. l est une fonction de localisation non redondante si et seulement si pour toute fonction de garde g, on a :*

$\forall (A, B) \in \mathcal{E}, \forall B \in Vers, \forall S \in \mathcal{P}(Edges(B))$

On pose $G = g(B, S)$ *et* $VM = VertexMap(B, G)$

$$\begin{aligned} \forall (c_0, c_1) \in l(A, G), & (c_0 \in range(VM)) \wedge \\ & (c_1 \in range(VM)) \wedge \\ & ((VM^{-1}(c_0), VM(c_1)^{-1}) \in Edges(B) \setminus S) \end{aligned} \tag{5.22}$$

La motivation de la recherche d'une fonction de localisation non redondante est que celle-ci limite les arêtes pouvant être localisées par des itérations successives aux arêtes de la version B. Il existe donc une itération à partir de laquelle l'ensemble des arêtes sera vide, et le processus terminera.

La deuxième contrainte qu'on peut appliquer à une fonction de localisation est la complétude. Une fonction de localisation est dite complète lorsque la fonction localise nécessairement au moins une arête quand une version est incorrecte vis-à-vis de la version de référence.

Définition 5.2.9. *Soit l une fonction de localisation. l est une fonction de localisation complète si et seulement si pour tout $(A, B) \in \mathcal{E}$, $\neg noLoss(A, B) \implies l(A, B) \neq \emptyset$.*

Nous réitérons la fonction de réparation obtenue avec une fonction de localisation non redondante et complète, et nous montrons que la suite des versions ainsi réparées est stationnaire, et que le point fixe est réparé.

Pour ce faire, pour tout $n \in \mathbb{N}^*$, nous définissons $L^n_{l,g}$ comme l'union des résultats de la fonction de localisation lors des itérations de la réparation. Nous montrons ensuite par récurrence avec la Propriété 5.2.3 que l'application de la fonction de garde sur l'ensemble $L^n_{l,g}$ équivaut à l'application successive des fonctions de garde sur les éléments de cette union, c'est-à-dire que $R_{L^n_{l,g},g} = R^n_{l,g}$. En effet, la non redondance de la fonction de localisation l nous assure que l'union construisant $L^n_{l,g}$ est disjointe, et que toutes les arêtes appartenant à $L^n_{l,g}$ sont dans B. Le prédicat 5.2.6.(v) nous permet alors de conclure que l'application de motifs sur l'union des localisations d'arêtes équivaut à l'application successive des motifs sur chacune des localisations d'arêtes.

Nous montrons ensuite que pour tout couple de versions (A, B), $(L^n_{l,g})_{n\mathbb{N}}(A, B)$ est stationnaire, puisqu'il s'agit d'une suite croissante de parties de $Edges(B)$.

Définition 5.2.10. *Soit g une fonction de garde et l une fonction de localisation complète et non redondante.*

On définit la fonction de localisation $L^n_{l,g}$ pour tout $n \in \mathbb{N}^*$ récursivement :

$$L^1_{l,g}(A,B) = l(A,B) \tag{5.23}$$

Et pour tout $n \geq 1$, en posant $G_n = R_{L^n_{l,g}(A,B),g}(A,B)$

$$L^{n+1}_{l,g}(A,B) = L^n_{l,g}(A,B) \cup \{VertexMap_{B,G_n}(c_0), VertexMap_{B,G_n}(c_1) \mid (c_0,c_1) \in l(A,G_n)\} \tag{5.24}$$

Démonstration. Nous montrons par récurrence que la définition de $L^{n+1}_{l,g}(A,B)$ est valide et que $L^{n+1}_{l,g}$ est une fonction de réparation pour tout $n \in \mathbb{N}$

On sait que $L^1_{l,g}(A,B) = l(A,B)$, $L^1_{l,g}(A,B)$ est une fonction de localisation.

Il nous reste donc à montrer que pour $n \in \mathbb{N}^*$ la définition de $L^{n+1}_{l,g}(A,B)$ est valide et que $L^{n+1}_{l,g}$ est bien une fonction de localisation. Pour cela, il suffit de montrer que :

1. $\forall (c_0,c_1) \in l(A,G_n), c_0 \in domain(VertexMap_{B,G_n}) \wedge c_1 \in domain(VertexMap_{B,G_n})$ (pour que $VertexMap_{B,G_n}(c_0)$ et $VertexMap_{B,G_n}(c_1)$, utilisés dans la définition de $L^{n+1}_{l,g}(A,B)$, soient définis)
2. $L^{n+1}_{l,g}(A,B) \subseteq Edges(B)$
3. $noLoss(A,B) \implies L^{n+1}_{l,g}(A,B) = \emptyset$

Tout d'abord, soit $(c_0,c_1) \in l(A,G_n)$. Comme l est une fonction de localisation non redondante, on sait par la Définition 5.2.8 que $c_0 \in domain(VertexMap_{B,G_n})$, $c_1 \in domain(VertexMap_{B,G_n})$ (donc (1) est vérifié), et $((VertexMap_{B,G_n}(c_0), VertexMap_{B,G_n}(c_1) \in Edges(B)$.

D'où, comme $L^n_{l,g}(A,B) \subseteq Edges(B)$

$$\begin{aligned} L^{n+1}_{l,g}(A,B) &= L^n_{l,g}(A,B) \cup \{VertexMap_{B,G_n}(c_0), VertexMap_{B,G_n}(c_1) \mid (c_0,c_1) \in l(A,G_n)\} \\ &\subseteq Edges(B) \end{aligned}$$

D'où (2) est vérifié.

Enfin, si $noLoss(A,B)$, comme $L^n_{l,g}$ est une fonction de localisation, $L^n_{l,g} = \emptyset$ par 5.2.5.(ii).

De plus,

$$\begin{aligned} G_n &= R_{L^n_{l,g}(A,B),g}(A,B) \\ &= R_{\emptyset,g}(A,B) \\ &= B \end{aligned}$$

Par conséquent,

$$\begin{aligned} L^{n+1}_{l,g}(A,B) &= L^n_{l,g}(A,B) \cup \{VertexMap_{B,G_n}(c_0), VertexMap_{B,G_n}(c_1) \mid (c_0,c_1) \in l(A,G_n)\} \\ &\subseteq Edges(B) \end{aligned}$$

D'où, comme l est une fonction de localisation, $l(A,G_n) = l(A,B) = \emptyset$ par 5.2.5.(ii).

Donc finalement :

$$\begin{aligned} L^{n+1}_{l,g}(A,B) &= L^n_{l,g}(A,B) \cup \{VertexMap_{B,G_n}(c_0), VertexMap_{B,G_n}(c_1) \mid (c_0,c_1) \in l(A,G_n)\} \\ &= \emptyset \cup \emptyset \\ &= \emptyset \end{aligned} \tag{5.25}$$

D'où (3) est vérifié, la définition de $L^{n+1}_{l,g}(A,B)$ est valide et $L^{n+1}_{l,g}$ est bien une fonction de localisation. □

Propriété 5.2.3. *Soit g une fonction de garde et l une fonction de localisation complète et non redondante.*

$\forall n \in \mathbb{N}, R^n_{l,g} = R_{L^n_{l,g},g}$

Démonstration. Soit $(A,B) \in \mathcal{E}$

Montrons par récurrence que la proposition (P_n) : "$R^n_{l,g}(A,B) = R_{L^n_{l,g},g}(A,B)$" est valide pour tout $n \in \mathbb{N}^*$

$L^1_{l,g} = l$, d'où $R^1_{l,g}(A,B) = R_{L^1_{l,g},g}(A,B)$

Donc (P_1) est valide

Soit $n \in \mathbb{N}^*$ tel que (P_n) est valide, et donc $R^n_{l,g}(A,B) = R_{L^n_{l,g},g}(A,B)$. Montrons que (P_{n+1}) est valide.

On poste $G = R_{L^n_{l,g}(A,B)}$, $VM = VertexMap(B,G)$, $S_1 = L^n_{l,g}(A,B)$, $S'_2 = l(A,G)$, $S_2 =$

$\{(VM^{-1}(u_0), VM^{-1}(u_1)) \mid (u_0, u_1) \in S'_2\}$

Comme l est non redondante, $\forall (c_0, c_1) \in l(A, G)(VM^{-1}(c_0), VM^{-1}(c_1)) \in Edges(B) \setminus L^n_{l,g}(A, B)$

D'où $S_2 \subseteq Edges(B)$, et par 5.2.6.(v) (l'application successive d'une fonction de garde sur B avec des ensembles d'arêtes équivaut à l'application de la fonction de garde avec l'union des ensembles d'arêtes si cette union est disjointe et si toutes les arêtes appartiennent à la version B) :

$$g(g(B, S_1), S'_2) = g(B, (S_1 \cup S_2) \tag{5.26}$$

D'où :

$$\begin{aligned} R^{n+1}_{l,g}(A, B) &= R_{l,g}(B, R^n_{l,g}(A, B)) \\ &= g(R^n_{l,g}(A, B), S'_2) \\ &= g(g(B, S_1), S'_2) \\ &= g(B, S_1 \cup S_2) \\ &\text{(par l'Equation 5.26)} \\ &= g(B, (L^{n+1}_{l,g}(A, B))) \end{aligned} \tag{5.27}$$

D'où (P_{n+1}) est valide.

Donc, par récurrence, (P_n) est valide pour tout $n \in \mathbb{N}^*$.

□

Propriété 5.2.4. *Soit g une fonction de garde et l une fonction de localisation complète et non redondante.*

$$\forall (A, B) \in \mathcal{E}, \exists n \in \mathbb{N}, noLoss(A, R^n_{l,g}(A, B)) \tag{5.28}$$

Démonstration. Soit $n \in \mathbb{N}^*$ En posant $G_n = R_{L^n_{l,g}(A,B)}$

$$L^{n+1}_{l,g}(A, B) = L^n_{l,g}(A, B) \cup \{VertexMap_{B,G_n}(c_0), VertexMap_{B,G_n}(c_1) \mid (c_0, c_1) \in l(A, G_n)\} \tag{5.29}$$

Par conséquent, $(S_n)_{n\in\mathbb{N}^*} = (L^n_{l,g}(A, B))_{n\in\mathbb{N}^*}$ est une suite croissante (au sens de l'inclusion) d'ensembles.

l étant une fonction de localisation non redondante, $\forall n \in \mathbb{N}, L^n_{l,g}(A, B) \in Edges(B)$ et

$\{VertexMap_{B,G_n}(c_0), VertexMap_{B,G_n}(c_1) \mid (c_0, c_1) \in l(A, G_n)\} \cap L^n_{l,g}(A, B) = \emptyset$

Comme $Edges(B)$ est fini et $(S_n)_{n\in\mathbb{N}}$, alors il existe $i \in \mathbb{N}$ tel que $S_{i+1} = S_i$.

D'où , en notant $S'_2 = \{VertexMap_{B,G_n}(c_0), VertexMap_{B,G_n}(c_1) \mid (c_0, c_1) \in l(A, G_n)\}$

$$
\begin{aligned}
& L^i_{l,g}(A, B) = L^{i+1}_{l,g}(A, B) \\
\Longrightarrow\ & L^i_{l,g}(A, B) = L^i_{l,g}(A, B) \cup S'_2 \\
\Longrightarrow\ & S'_2 \subseteq L^i_{l,g}(A, B) \\
\Longrightarrow\ & S'_2 = S'_2 \cap L^i_{l,g}(A, B) \\
\Longrightarrow\ & l(B, L^i_{l,g}(A, B)) = \emptyset \\
& \qquad (S'_2 \cap L^i_{l,g}(A, B) = \emptyset \text{ car } l \text{ est une fonction de localisation non redondante}) \\
\Longrightarrow\ & noLoss(B, L^i_{l,g}(A, B)) \\
& \qquad (\text{comme } l \text{ est une fonction de localisation complète})
\end{aligned}
$$

□

Les suites $(L^n_{l,g})_{n\in\mathbb{N}}$ et $(R^n_{l,g})_{n\in\mathbb{N}}$ sont donc stationnaires. Nous définissons donc $R^\infty_{l,g}$ comme la fonction de réparation qui retourne le point fixe de la réitération de $R_{l,g}$ sur une paire de versions. La propriété précédente garantit que $R^\infty_{l,g}$ est une fonction de p-repair valide.

Définition 5.2.11. *Soit g une fonction de garde et l une fonction de localisation complète et non redondante.*

Soit :

$$
\begin{aligned}
n_\infty(A, B) &= min\{n \in \mathbb{N}, noLoss(A, R^n_{l,g}(A, B))\} \\
R^\infty_{l,g}(A, B) &= R^{n_\infty(A,B)}_{l,g}(A, B) \\
L^\infty_{l,g}(A, B) &= L^{n_\infty(A,B)}_{l,g}(A, B)
\end{aligned}
\tag{5.30}
$$

Le minimum existe par la Propriété 5.2.4.

Propriété 5.2.5. *Soit g une fonction de garde et l une fonction de localisation complète et non redondante.*

$R^\infty_{l,g}$ est une fonction de noLoss-repair valide.

Démonstration. Soit $(A, B) \in \mathcal{E}$.

$R^\infty_{l,g}(A, B) = R^{n_\infty(A,B)}_{l,g}(A, B)$ (par la Définition 5.2.11)

Et $noLoss(A, R^{n_\infty(A,B)}_{l,g})$ est vraie (par la Définition 5.2.11)

Donc $noLoss(A, R_{l,g})^\infty$ est vraie

Donc $R^{\infty}_{l,g}$ est une fonction de *noLoss*-repair valide. □

5.2.4 Application à l'algorithme SuspiciousEdges

Nous définissons désormais un algorithme SuspiciousEdges (Algorithme 1), construit pour repérer les arêtes responsables des pertes de protection.

Afin de démontrer que cet algorithme permet de localiser les arêtes à sécuriser, nous montrons tout d'abord qu'il s'agit d'une fonction de localisation selon la Définition 5.2.5, puis nous montrons que celle-ci vérifie les propriétés de non redondance (Définition 5.2.8) et de complétion (Définition 5.2.9).

Algorithm 1 SuspiciousEdges

```
1:  procedure SuspiciousEdges
        Input :A, B
2:      (V_a, E_a) ← CFG(A)
3:      (V_b, E_b) ← CFG(B)
4:      (Q_a, T_a, q0_a, Var_a, G_a, A_a) ← PTFA(A)
5:      (Q_b, T_b, q0_b, Var_b, G_b, A_b) ← PTFA(B)
6:      (Q'_b, T'_b) = Reachable(PTFA(B))
7:      for c_a ∈ Vertexes(A) do
8:          if c_a ∈ domain(vertexMap_{A,B}) then
9:              c_b ← vertexMap_{A,B}(c_a)
10:             if ¬defProt(c_b) ∧ defProt(c_a) then
11:                 for cp_b ∈ interpred(c_b) do
12:                     if ∃j, j', (q_{cp_b,1,j}, q_{c_b,0,j'}) ∈ T'_b then
13:                         S ← S ∪ {(cp_b, c_b)}
14:                     else
15:                         if ∃j, j', (q_{cp_b,j,0}, q_{c_b,j',0}) ∈ T'_b then
16:                             if cp_b ∈ range(vertexMap_{A,B}) then
17:                                 cp_a ← (vertexMap_{A,B})^{-1}(cp_b)
18:                                 if ¬defProt(cp_a) then
19:                                     S ← S ∪ {(cp_b, c_b)}
20:                             else
21:                                 S ← S ∪ {(cp_b, c_b)}
        Output :S
```

Afin de démontrer que SuspiciousEdges est une fonction de localisation, nous déduisons la composition de l'ensemble d'arêtes en sortie de $SuspiciousEdges(A, B)$, et nous en déduisons la vérification des axiomes en observant la structure de l'algorithme.

Propriété 5.2.6. *SuspiciousEdges est une fonction de localisation.*

Démonstration. Pour montrer que SuspiciousEdges est une fonction de localisation, nous devons montrer qu'elle vérifie les axiomes 5.2.5.(i) et 5.2.5.(ii).

Soit $(A, B) \in \mathcal{E}$. Par construction de l'algorithme, tout élément dans l'ensemble S de sortie de $SuspiciousEdges(A, B)$ est construit comme un couple (c_b, cp_b) tel que $cp_b \in interpred(c_b)$. D'où $SuspiciousEdges(A, B) \subseteq E_B$, et SuspiciousEdges vérifie l'axiome 5.2.5.(i) de la définition d'une fonction de localisation.

Si $noLoss(A, B)$, alors pour tout $c_a \in domain(vertexMap_{A,B})$, on a $defProt(vertexMap_{A,B}(c_a)) \vee \neg defProt(c_a)$, donc par construction de l'algorithme SuspiciousEdges, $(cp_b, vertexMap_{A,B}(c_a)) \notin S$ pour tout noeud cp_b tel que $(cp_b, c_b) \in Edges(B)$.

Par conséquent, $S = \emptyset$, et SuspiciousEdges vérifie l'axiome 5.2.5.(ii) de la définition d'une fonction de localisation.

□

Nous montrons maintenant que la fonction $SuspiciousEdges$ est non redondante. Pour rappel, cela signifie que pour une paire de versions (A, B) avec $(A, B) \in \mathcal{E}$, tout ensemble S d'arêtes de B et toute fonction de garde g, $SuspiciousEdges(A, g(B, S)$ retourne uniquement des arêtes de $Edges(B)$, et ne retourne aucune arête de S.

Pour ce faire, nous procédons par l'absurde. Nous supposons par l'absurde que $SuspiciousEdges(A, g(B, S)$ retourne des arêtes n'étant pas dans $Edges(B)$, ou des arêtes étant dans S. Dans chacun de ces deux cas, nous faisons ressortir une contradiction permettant de finalement conclure que la fonction $SuspiciousEdges$ est non redondante.

Propriété 5.2.7. *SuspiciousEdges est une fonction de localisation non redondante.*

Démonstration. Soit g une fonction de garde.

On note $(V_a, E_a) = CFG(A)$, $(V_b, E_b) = CFG(B)$ et $(Q_b, T_b, q_{0b}, Var_b, A_b) = PTFA(B)$

Soient $A, B \in \mathcal{E}$. Soit $S \in \mathcal{P}(E_b)$.

On note $G = g(B, S)$, $(V_G, E_G) = CFG(G)$, $(Q_G, T_G, q_{0G}, Var_G, A_G) = PTFA(G)$ et $VM = VertexMap_{B,g(B,S)}$

Soit $(c_0, c_1) \in SuspiciousEdges(A, g(B, S))$.

Montrons par l'absurde que $c_0 \in range(VM) \wedge c_1 \in range(VM) \wedge (VM^{-1}(c0), VM^{-1}(c1)) \in E_b \setminus S$.

Nous supposons ainsi que $\neg(c_0 \in range(VM) \wedge c_1 \in range(VM) \wedge (VM^{-1}(c0), VM^{-1}(c1)) \in E_b)$ ou $(VM^{-1}(c0), VM^{-1}(c1)) \in S$. (*)

Si $(VM^{-1}(c0), VM^{-1}(c1)) \in S$, alors par 5.2.6.(vi), $\forall j, k, j', (q(c_0, j, k), q(c_1, j', 0)) \notin T_G$.

D'où $\forall j, j', q(c_0, j, 1), q(c_1, j', 0) \notin T_G$ et $\forall j, j', q(c_0, j, 0), q(c_1, j', 0) \notin T_G$, donc par construction de l'algorithme SuspiciousEdges, $(c_0, c_1) \notin SuspiciousEdges(A, g(B, S))$, menant à une contradiction.

Sinon, alors $\neg(c_0 \in range(VM) \wedge c_1 \in range(VM) \wedge (VM^{-1}(c0), VM^{-1}(c1)) \in E_b)$.

On sait que $(c_0, c_1) \in SuspiciousEdges(A, g(B, S))$, donc par construction de l'algorithme, $c1 \in range(VertexMap_{A,g(B,S)})$.

De plus, par 5.2.6.(viii), on a $VertexMap_{A,g(B,S)} = VM \circ VertexMap_{A,B})$, d'où :

$$range(VertexMap_{A,g(B,S)}) \subseteq range(VM)$$

Donc $c1 \in range(VM)$

Donc par 5.2.6.(iv) : $c_0 \notin range(VM)$

Donc : $\forall a \in \mathbb{N}, \forall o = (q_{0G}, o_1, \ldots, o_i, \ldots, o_a, o_{a+1}) \mid vertex(o_a) = c_0 \wedge vertex(o_{a+1}) = c_1$,

$\exists x < a \mid (\forall x < y \leq a, vertex(o_y) \notin range(VM)) \wedge (vertex(o_x) \in range(VM))$ (car $vertex(q_0) \in range(VM)$ et $vertex(o_a) \notin range(VM)$)

De plus, également par 5.2.6.(iv) :

$$\begin{gathered} \forall p = (p_1, \ldots, p_i, \ldots, p_n) \in Paths(G) \mid \\ (vertex(p_1) \in range(VM) \wedge vertex(p_n) = c_1 \wedge vertex(p_{n-1}) = c_0 \wedge \\ (\forall 1 < j < n, vertex(p_j) \notin range(VM))), \\ (VM^{-1}(vertex(p_1)), VM^{-1}(vertex(p_n))) \in S \end{gathered} \tag{5.31}$$

D'où $(VM^{-1}(o_x), VM^{-1}(o_{a+1})) \in S$

Par 5.2.6.(vi) appliqué à $(o_x, \ldots, o_i, \ldots, o_{a+1})$, $prot(o_{a+1}) = 1$

Pour résumer : $\forall a \in \mathbb{N}, \forall o = (q_{0G}, o_1, \ldots, o_i, \ldots, o_a, o_{a+1}) \mid (vertex(o_a) = c_0 \wedge vertex(o_{a+1}) = c_1), prot(o_{a+1}) = 0$

D'où, en posant $(Q'_G, T'_G) = Reachable(PTFA(B))$, $\forall j, j', k, q(c_0, j, k), q(c_1, j', 0) \notin T'_G$, donc par construction de l'algorithme SuspiciousEdges, $(c_0, c_1) \notin SuspiciousEdges(A, g(B, S))$, menant à une contradiction.

Donc dans tous les cas, (*) abouti à une contradiction, d'où finalement $c_0 \in range(VM) \wedge c_1 \in range(VM) \wedge (VM^{-1}(c0), VM^{-1}(c1)) \in E_b \setminus S$.

SuspiciousEdges est donc bien non redondante.

□

Nous montrons maintenant que la fonction *SuspiciousEdges* est complète. Pour rappel, cela signifie que la fonction de localisation reporte toujours une arête si le programme à réparer n'est pas déjà correct.

Pour ce faire, nous supposons que le programme contient au moins une perte de sécurité, et nous posons un chemin non sécurisé allant vers cette perte. Nous prenons une arête bien choisie de ce graphe, et nous traversons la structure de l'algorithme *SuspiciousEdges* pour montrer que dans tous les cas, si la version a une perte de protection par rapport à la version de référence, cette arête appartient à l'ensemble de sortie de *SuspiciousEdges*, qui est par conséquent non nul.

Propriété 5.2.8. *SuspiciousEdges est une fonction de localisation complète.*

Démonstration. Soit $(A, B) \in \mathcal{E}$ tel que $\neg noLoss(A, B)$. On a donc $DefLoss(A, B) \neq \emptyset$.

Soit $(c_a, c_b) \in DefLoss(A, B)$.

Par définition de $DefLoss$, on sait que $c_b = VertexMap_{A,B}(c_a), DefProt(c_a)$ et $\neg DefProt(c_b)$.

Comme $\neg DefProt(c_b)$, $\exists j \in \{0, 1\}, p = (q_0, p_1 \ldots, p_i, \ldots, p_n, q_{c_b,j,0}) \in path(B)$.

Soit p_i le premier noeud de ce chemin tel qu'en posant $c_i = vertex(p_i)$, $(prot(p_i) = 0) \wedge c_i \in range(VertexMap_{A,B}) \wedge Defprot((VertexMap_{A,B})^{-1}(c_i))$ (un tel noeud existe parce que $q_{c_b,j,0}$ vérifie cette propriété, et $i > 0$ parce que q_0 ne la vérifie pas).

Comme $i > 0$, p_{i-1} existe, et en posant $c_{i-1} = vertex(p_{i-1})$, $prot(p_{i-1}) = 1 \vee (vertex(c_{i-1}) \notin range(VertexMap_{A,B})) \vee (\neg Defprot((VertexMap_{A,B})^{-1}(c_{i-1})))$ comme c_i est le premier noeud pour lequel ce n'est pas le cas.

Si $prot(c_{i-1}) = 1$, comme $prot(c_i) = 0$ et $(c_{i-1}, c_i) \in T'_b$, donc $(vertex(c_{i-1}), vertex(c_i)) \in S$ (par construction de l'algorithme 1)

Sinon, $prot(c_{i-1}) = 0$, alors $\neg DefProt(vertex(c_{i-1}))$, $prot(c_i) = 0$ et $(c_{i-1}, c_i) \in T'_b$ (parce que $prot(c_i, p) = 0$)

Donc si $vertex(c_{i-1}) \notin range(VertexMap_{A,B})$, alors $(vertex(c_{i-1}), vertex(c_i)) \in S$ (par construction de l'algorithme 1)

Sinon, alors $\neg Defprot((VertexMap_{A,B})^{-1}(vertex(c_i)))$. Donc, $(vertex(c_{i-1}), vertex(c_i)) \in S$ (par construction de l'algorithme 1)

Donc dans tous les cas, $S \neq \emptyset$ □

5.2.5 Mesures et minimisation avec l'algorithme GuiltyEdges

Nous avons alors prouvé que l'algorithme SuspiciousEdges répare toujours l'ensemble des pertes de protection. Néanmoins, l'algorithme SuspiciousEdges peut appliquer le motif sur plus d'arêtes que nécessaire, comme le montre l'exemple en Section 5.3, nous pouvons donc essayer de l'améliorer. Dans cette Section, nous proposons plusieurs mesures pouvant mesurer la performance d'une fonction de réparation de motifs et d'une fonction de localisation.

Nous introduirons par la suite un algorithme GuiltyEdges permettant de filtrer la sortie de l'algorithme SuspiciousEdges et de retirer les arêtes où l'insertion d'un motif serait redondant.

Afin de fournir un outil d'évaluation des fonctions de réparation de la forme $R_{l,g}$, nous définissons deux mesures ciblant deux aspects de la réparation. Tout d'abord, la mesure SE est l'effet de bord de la fonction de réparation : elle se réfère à la part de noeuds que la fonction de réparation protège en plus de ceux qui devaient être corrigés. Il s'agit d'une mesure sémantique, puisqu'elle mesure l'écart des comportements des programmes.

Définition 5.2.12. *Soit f une fonction de noLoss-repair.*

Soit SE_f l'effet de bord de f définie ainsi :

$$\forall (A,B) \in \mathcal{E}, SE_f = \frac{\left| \begin{array}{l} \{x \mid \exists y, (x,y) \in DefGain(B, f(A,B))\} \setminus \\ \{x \mid \exists y, (x,y) \in DefGain(B,A)\} \end{array} \right|}{|range(VertexMap_{B,f(A,B)}|} \tag{5.32}$$

La deuxième mesure W (Définition 5.2.13) est définie sur la fonction de localisation, et mesure simplement la part de l'ensemble des arêtes localisées par rapport à l'ensemble des arêtes du programme. Il s'agit d'une mesure syntaxique, puisque l'ensemble des arêtes localisées est celui des arêtes qui seront modifiées.

Définition 5.2.13. *Soit l une fonction de localisation. On définit W_l le poids de l défini ainsi :*

$$\forall (A,B) \in \mathcal{E}, W_l = \frac{|l(A,B)|}{|Edges(B)|} \tag{5.33}$$

Nous définissons maintenant l'algorithme GuiltyEdges (Algorithme 2). Celui-ci se sert du résultat de la réparation afin de filtrer un ensemble d'arêtes localisées en conservant uniquement les arêtes préalablement identifiées si leur origine n'est pas définitivement protégée dans

la version réparée ou si ces arêtes étaient des révocations de privilèges avant la réparation. Ainsi, le motif est appliqué à moins d'endroits et le code est moins changé, donc la mesure W est réduite.

Algorithm 2 GuiltyEdges

1: **procedure** GUILTYEDGES
   Input :$versB, versR, S$
2: $(V_b, E_b) \leftarrow CFG(versB)$
3: $(V_r, E_r) \leftarrow CFG(versR)$
4: $G = \emptyset$
5: **for** $(cp, c) \in S$ **do**
6: **if** $(\neg defProt(vertexMap_{versB,versR}(cp))) \vee (q_{cp,1,k}, q_{c,0,k'}) \in T_{versB}$ **then**
7: $G \leftarrow G \cup \{(cp, c)\}$
   Output :G

Définition 5.2.14. *Soit g une fonction de garde et l une fonction de localisation telle que $R_{l,g}$ est une fonction de noLoss-repair valide.*

On définit $\mathring{R}_{l,g} = R_{GuiltyEdges(l),g}$

Nous montrons désormais qu'en appliquant l'algorithme GuiltyEdges à une fonction de réparation valide de la forme $R_{l,g}$, nous obtenons toujours une fonction de réparation valide ayant les mêmes effets de bord.

Pour ce faire, nous montrons dans un premier temps un lemme énonçant qu'en retirant de l'ensemble des arêtes localisées une arête n'étant pas une révocation, et dont l'origine est définitivement protégée après réparation, on ne change pas les niveaux de protection des noeuds.

Nous démontrons ce lemme comme une double implication : nous montrons tout d'abord que si un noeud n'est pas protégé avec la réparation incluant l'arête, il ne l'est pas avec la réparation excluant l'arête. Nous montrons ensuite la réciproque, c'est-à-dire que si un noeud n'est pas protégé avec la réparation excluant l'arête il ne l'est pas avec la réparation incluant l'arête.

Pour chacune de ces implications, nous considérons un chemin non protégé débouchant sur le noeud dans la version pour laquelle on sait qu'il est non protégé, et nous l'exploitons pour construire un chemin non protégé débouchant sur le noeud correspondant dans l'autre version, montrant ainsi que le noeud correspondant n'est pas protégé.

Lemme 5.2.1. *Soient $(A, B) \in \mathcal{E}$, g une fonction de garde sur $\{(A)\}$, $S \in \mathcal{P}(Edges(B))$ et $(cp_b, c_b) \in Edges(B) \setminus S$.*

On pose $(Q_b, T_b, q_{0b}, Var_b, A_b) = PTFA(B)$

Soient $R = g(V, S)$ *et* $R2 = g(B, S \cup \{(cp_b, c_b)\})$.

Si $Defprot(VertexMap_{B,R2}(cp_b)) \wedge \forall k, k' \in \{0,1\}((cp_b, k, 1), (c_b, k', 0)) \notin T_b$, *alors :*

$$\forall c \in Vertexes(B), (Defprot(VertexMap_{B,R2}(c)) \Leftrightarrow Defprot(VertexMap_{B,R}(c))) \quad (5.34)$$

Démonstration. Soient $(A, B) \in \mathcal{E}$, et g une fonction de garde sur $\{(A)\}$. Soient $S \in \mathcal{P}(Edges(B))$ et $(cp_b, c_b) \in Edges(B) \setminus S$ tels que $Defprot(VertexMap_{B,g(B,S\cup\{(cp_b,c_b)\})}(cp_b))$ et $((cp_b, k, 1), (c_b, k', 0)) \notin T_b$.

On pose $(Q_b, T_b, q_{0b}, Var_b, A_b) = PTFA(B)$

Soient $R = g(B, S)$ et $R2 = g(B, S \cup \{(cp_b, c_b)\})$

Par 5.2.6.(v), $R2 = g(R, \{(VertexMap_{B,R}(cp_b), VertexMap_{B,R}(c_b))\})$

Pour tout $c \in Vertexes(B)$:

Si $\neg Defprot(VertexMap_{B,R2}(c))$, alors $p = (p_0, \ldots, p_i, \ldots, p_n) \in path(R2)$ avec $p_0 = q_{0R2}$, $vertex(p_n) = VertexMap_{B,R2}(c)$ et $prot(p_n) = 0$. Nous posons ce chemin.

Nous construisons récursivement un chemin $p' = (p_0, \ldots, p'_i, \ldots, p'_m)$ de R tel que $p'_0 = q_{0R}$, $vertex(p'_m) = VertexMap_{B,R}(c)$ et $prot(p'_m) = 0$ et une séquence d'entiers $(j_i)_{i<m}$, tels que $\forall i < m, (vertex(p_{j_i}) = VertexMap_{R,R2}(vertex(p'_i))) \wedge (prot(p_{j_i}) = prot(p'_i))$.

On a $prot(q_{0R2}) = prot(q_{0R}) = 0$

Soit $i < m$

Par construction du chemin, $(p_{j_i} = p'_i) \wedge (prot(p_{j_i}) = prot(p'_i))$.

Si $vertex(p_{j_i+1}) \in range(VertexMap_{R,R2})$, comme $vertex(p_{j_i}) \in range(VertexMap_{R,R2})$, alors $(VertexMap^{-1}_{R,R2}(p_{j_i}), VertexMap^{-1}_{R,R2}(p_{j_i+1})) \in Edges(R)$ par 5.2.6.(iv). Nous ajoutons alors $q_{c,j',k'}$ avec $c = VertexMap^{-1}_{R,R2}(vertex(p_{j_i+1}))$, $j' = prot(p_{j_i+1})$ et $k' = context(p_{j_i+1})$ à p', et nous posons $j_{i+1} = j_i + 1$.

Comme $prot(p'_i) = prot(p_{j_i})$, $prot(p_{j_{i+1}}) = prot(p'_{i+1})$.

Sinon, alors soit $x = min(\{i' > 0, p_{j_i+i'} \in range(VertexMap_{R,R2})\})$ (le minimum existe car $vertex(p_n) = VertexMap_{B,R2}(c) \in range(VertexMap_{R,R2})$ et est plus grand que 1 comme $vertex(p_{j_i+1}) \notin range(VertexMap_{R,R2})$) et $p'' = (p_{j_i}, \ldots, p_{j_i+i'}, \ldots, p_{j_i+x})$

Par 5.2.6.(iv), $(vertex(p_{j_i}), vertex(p_{j_i+x})) = (VertexMap_{B,R2}(cp_b), VertexMap_{B,R2}(c_b))$

De plus, comme $Defprot(VertexMap_{B,R2}(cp_b))$, alors $prot(p_{j_i}) = prot(p_i) = 1$

Par 5.2.6.(vi), $prot(c_b) = 1$.

Comme $((cp_b, k, 1), (c_b, k', 0)) \notin T_b$ et $(VertexMap_{B,R}(cp_b), VertexMap_{B,R}(c_b)) \in Edges(R)$, alors nous posons $vertex(p'_{i+1}) = VertexMap_{B,R}(c_b)$, $j_{i+1} = j_i + x$, $prot(p'_{i+1}) = 1 = prot(p_{j_{i+1}})$ et $context(p'_{i+1}) = context(p_{j_{i+1}})$.

Le processus itératif termine lorsque nous atteignons $VertexMap_{B,R}(c)$.

Nous avons donc finalement construit itérativement un chemin $p' = (p'_0, \ldots, p'_i, \ldots, p'_m) \in Paths(R)$ tel que $vertex(p'_0) = q_{0R}$, $prot(p'_m) = 0$ et $vertex(p'_m) = VertexMap_{B,R}(c)$, d'où $\neg Defprot(VertexMap_{V,R}(c))$.

On a donc montré que $\neg Defprot(VertexMap_{B,R2}(c)) \implies \neg Defprot(VertexMap_{B,R}(c))$

Réciproquement, si $\neg Defprot(VertexMap_{B,R}(c))$, alors $\exists k \in \{0, 1\}, p = (p_0, \ldots, p_i, \ldots, p_n) \in Paths(R)$ avec $p_0 = q_{0R}$ et $p_n = q_{VertexMap_{B,R}(c),0,k}$.

Nous posons un tel chemin.

Nous construisons récursivement un chemin $p' = (p'_0, \ldots, p'_i, \ldots, p'_m)$ de $R2$ avec $p'_0 = q_{0R2}$, $vertex(p'_m) = VertexMap_{B,R}(c)$ et $prot(p'_m) = 0$ et une séquence d'entiers $(j_i)_{i<n}$, tels que $\forall i < n, (VertexMap_{R,R2}(vertex(p_i)) = vertex(p'_{j_i})) \wedge (prot(p_i) = prot(p'_{j_i}))$.

On a $prot(p'_0) = prot(p_0) = 0$

Nous construisons p' dans $R2$ ainsi :

Soit $i < n$.

Si $p_i \neq VertexMap_{B,R}(cp_b)$ ou $p_{i+1} \neq VertexMap_{B,R}(c_b)$, par 5.2.6.(iii), on a :

$(VertexMap_{R,R2}(p_i), VertexMap_{R,R2}(p_{i+1})) \in Edges(R2)$.

On pose donc $j_{i+1} = j_i + 1$ et $p'_{j_{i+1}} = VertexMap_{R,R2}(p_{i+1})$.

Comme $prot(p', p'_{j_i}) = prot(p, p_i)$, $prot(p', p'_{j_{i+1}}) = prot(p, p_{i+1})$

Sinon, $p_i = cp_b$ et $p_{i+1} = c_b$. $prot(p', p'_{j_i}) = prot(p, p_i) = 1$ (parce que $Defprot(VertexMap_{B,R2}(cp_b))$)

Comme $((cp_b, k, 1), (c_b, k, 0)) \notin T_b$, alors $prot(p_{i+1}) = 1$

Par (vii), $\exists x \in \mathbb{N}, p'' = (q_{cp_b,k,1}, p''_1, \ldots, p''_i, \ldots, p''_x) \in Paths(R2)$, $vertex(p''_n) = VertexMap_{B,R2}(c_b)$, $prot(p''_n) = 1$.

On ajoute les noeuds $(p''_1, \ldots, p''_i, \ldots, p''_x)$ à p', et on pose $j_{i+1} = j_i + x$. On a donc $prot(p'_{j_{i+1}}) = prot(p_{i+1}) = 1$

Le processus itératif termine lorsque nous atteignons $VertexMap_{B,R}(c)$.

Donc par récurrence, tous les noeuds de p' communs avec p ont le même niveau de protection que dans p. Nous avons donc construit un chemin non protégé allant vers c, d'où c n'est pas protégé dans $R2$.

Nous avons donc finalement construit itérativement un chemin $p' = (p'_0, \ldots, p'_i, \ldots, p'_m) \in Paths(R2)$ tel que $vertex(p'_0) = q_{0R2}$, $prot(p'_m) = 0$ et $vertex(p'_m) = VertexMap_{B,R2}(c)$, d'où $\neg Defprot(VertexMap_{B,R2}(c))$.

On a donc montré que $\neg Defprot(VertexMap_{B,R}(c)) \implies \neg Defprot(VertexMap_{B,R2}(c))$

Donc finalement, l'équivalence $Defprot(VertexMap_{B,R2}(c)) \Leftrightarrow Defprot(VertexMap_{B,R}(c))$ est vérifiée. □

Propriété 5.2.9. *Soit g une fonction de garde et l une fonction de localisation telle que $R_{l,g}$ est une fonction de noLoss-repair valide.*

$\mathring{R}_{l,g}$ est une fonction de noLoss-repair valide.

Et $SE_{\mathring{R}_{l,g}} = SE_{R_{l,g}}$.

Démonstration. Soit $(A, B) \in \mathcal{E}$.

Soient $(V_b, E_b) = CFG(B)$, $(Q_b, T_b, q_{0b}, Var_b, G_b, A_b) = PTFA(B)$

Soit $R = \mathring{R}_{l,g}(A, B)$ et $R2 = R_{l,g}$.

Pour toute arête (cp_b, c_b) de $GuiltyEdges(A, R2, l(A, B))$, par construction de $GuiltyEdges$, $Defprot(VertexMap_{B,R2}(cp_b)) \wedge (\forall k \in \{0, 1\}(q_{cp_b,k,1}, q_{c_b,k,0}) \notin T_b)$.

Donc, en appliquant le Lemme 5.2.1 en retirant les arêtes de $GuiltyEdges(A, R2, l(A, B))$ une à une, on montre que, pour tout $c \in Vertexes(B)$, $Defprot(VertexMap_{B,R}(c)) = Defprot(VertexMap_{B,R2}(c))$. (comme $domain(VertexMap_{B,R}) = domain(VertexMap_{B,R2}) = Vertexes(B)$)

Par conséquent, R et $R2$ ont les mêmes DPD avec A, d'où la propriété. □

En résumé, étant donné une fonction de réparation valide de la forme $R_{l,g}$, l'algorithme GuiltyEdges permet de filtrer le résultat de l pour induire une fonction de réparation $\mathring{R}_{l,g}$ valide causant moins de changements sur les fonctions réparées.

5.3 Illustration de la génération d'un patch par insertion de motifs

5.3.1 Exécution de SuspiciousEdges

Nous appliquons l'algorithme SuspiciousEdges à l'exemple présenté en Figure 5.1 au regard du privilège p. Comme montré en Figure 5.2, les instructions 3, 5 et 6 subissent une perte définitive de protection due au retrait de la condition.

Un résumé de l'exécution de l'algorithme sur les arêtes de l'exemple est présenté en Tableau 5.1. Pour rappel, l'algorithme retourne des arêtes de la version postérieure. Les deux arêtes retenues par l'algorithme sont $(1, 3)$ et $(4, 5)$.

— L'arête $(1, 3)$ est retenue parce que le noeud 3 comporte une perte de sécurité, l'arête $(1, 3)$ n'est pas un octroiement de privilège et le noeud 1 n'est protégé ni dans la version A, ni dans la version B.

— L'arête $(4, 5)$ est retenue parce que le noeud 5 comporte une perte de sécurité, l'arête $(4, 5)$ n'est pas un octroiement de privilège et le noeud 4 est un nouveau noeud.

— Un exemple d'arête n'étant pas retenue est l'arête $(5, 6)$. En effet, le noeud 6 comporte une perte de sécurité, l'arête $(5, 6)$ n'est pas un octroiement de privilège et le noeud 5 n'est pas protégé dans la version B, mais il l'est dans la version A. Cela veut dire que l'arête à protéger est à chercher en amont du noeud 5, et que la réparation de la perte de sécurité du noeud 5 évitera de devoir sécuriser l'arc $(5, 6)$.

Nous appliquons donc un motif de sécurité aux arêtes localisées. Pour cet exemple, il suffit d'insérer une instruction "`if (!current_user_can('p')) die();`" entre les deux noeuds de l'arête.

Le code du nouveau couple de versions $(A, R_{SuspiciousEdges,g})$ est présenté en Figure 5.3 et leurs CFG en Figure 5.4. Le couple n'a pas de perte de protection définitive (voir Figure 5.4), donc il n'y a pas besoin d'autres itérations de SuspiciousEdges.

Le coût sémantique SE de la réparation est de $1/7 = 0.14$, puisqu'en plus des instructions "`$b="padding";`", "`$d="padding";`" et "`$e="padding";`" que nous souhaitions protéger, l'instruction "`$f="padding";`" est protégée par le privilège p.

Le coût syntaxique W de la réparation est de $2/6 = 0.33$, puisque la taille de l'ensemble $SuspiciousEdges(A, B)$ est de 2.

5.3.2 Exécution de GuiltyEdges

Nous appliquons donc GuiltyEdges au triplet $(S, B, R_{SuspiciousEdges,g}(A, B))$. Un résumé de l'exécution de l'algorithme sur les arêtes de S est présenté dans le Tableau 5.2. La seule arête

```
1                   $a="padding";
2           -       if (current_user_can('p')){
3                   $b="padding";
4           +       $c="padding";
5                   $d="padding";
6                   $e="padding";
7           -       }
8                   $f="padding";
```

Figure 5.1 Représentation de couple de versions (A, B) nous servant d'exemple de référence

Tableau 5.1 Résumé de l'exécution de l'algorithme SuspiciousEdges sur l'exemple en Figure 5.1

c_a	1	2	3	5	6	8
$c_a \in domain(vertexMap_{verA,verB})$	$True$	$False$	$True$	$True$	$True$	$True$
c_b	1		4	5	6	8
$\neg defProt(c_b) \wedge defProt(c_a)$	$False$		$True$	$True$	$True$	$False$
cp_b			1	3	5	
$\exists k, (q(cp_b, 1, k), q(c_b, 0, k)) \in T_b$			$False$	$False$	$False$	
$\neg defProt(cp_b)$			$True$	$True$	$True$	
$\exists k, (q(cp_b, 0, k), q(c_b, 0, k)) \in Tb$			$True$	$True$	$True$	
$cp_b \in domain(vertexMap_{verB,verA})$			$True$	$False$	$True$	
cp_a			2		5	
$\neg defProt(cp_a)$			$True$		$False$	
$cp_b, c_b \in S$	$False$	$False$	$True$	$True$	$False$	$False$

retenue par l'algorithme est $(1, 3)$. L'arête $(4, 5)$ a donc été retirée de l'ensemble des arêtes localisées.

En effet, après l'application du motif aux arêtes localisées par *SuspiciousEdges*, le noeud $vertexMap_{verB,verR}(4) = 5$ est protégé. L'arête $(4, 5)$ est donc redondante avec l'opération de sécurité présente en amont.

Nous appliquons donc un motif à l'arête $(1, 3)$ depuis la version B. La version résultante est présentée dans la Figure 5.5.

Le coût sémantique SE de la réparation est de $1/7 = 0.14$.

Le coût syntaxique W de la réparation est de $1/6 = 0.17$, et a donc été amélioré par rapport à l'algorithme *SuspiciousEdges* seul.

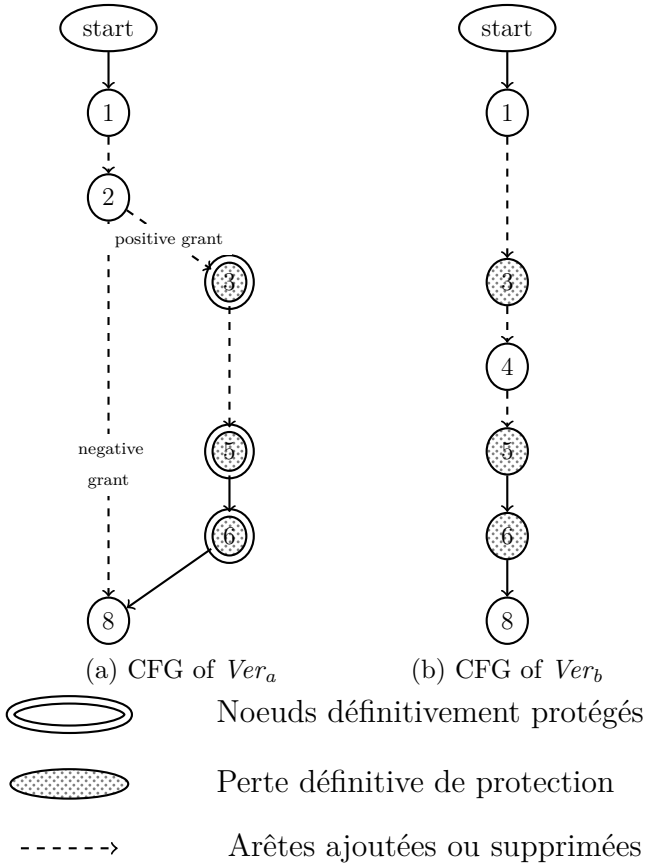
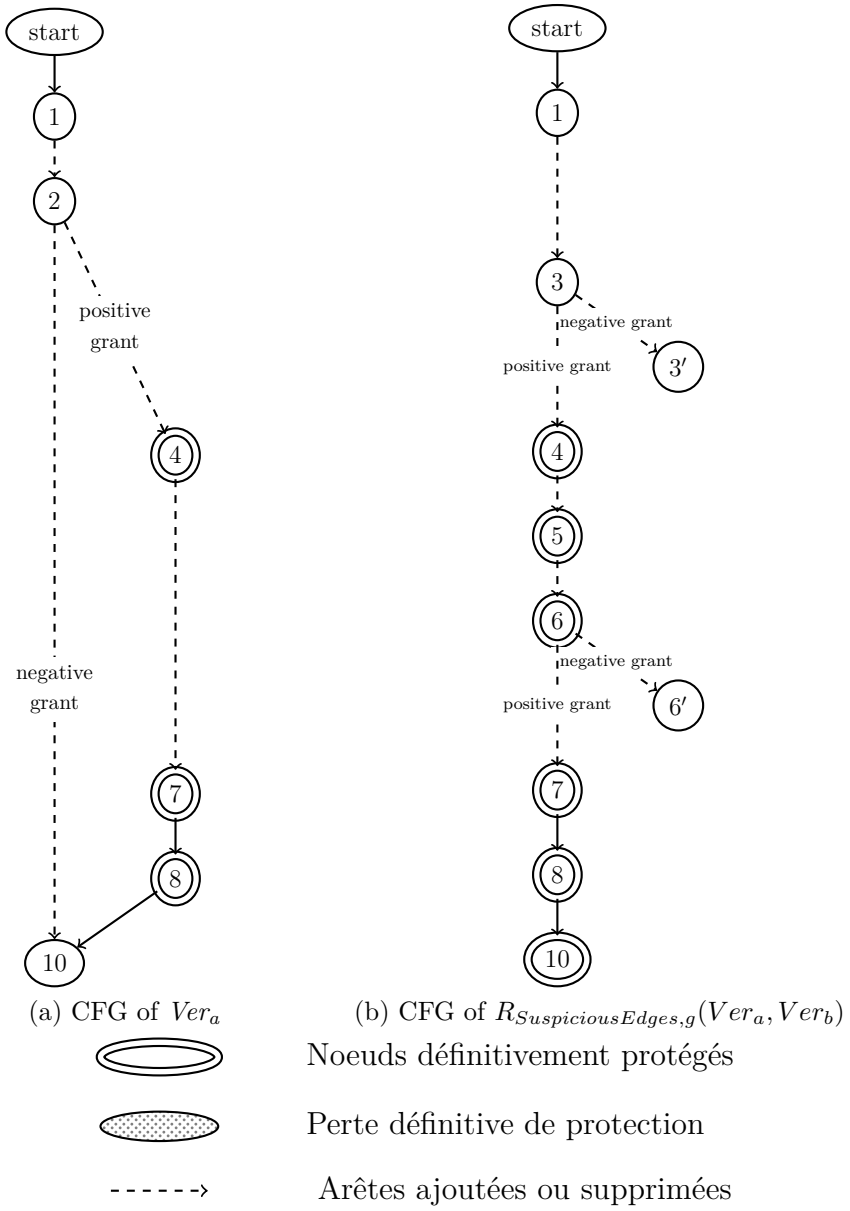


Figure 5.2 Couple de versions du code en Figure 5.1

```
1                 $a="padding";
2         -       if (current_user_can('p')){
3         +       if (!current_user_can('p')) die();
4                 $b="padding";
5         +       $c="padding";
6         +       if (!current_user_can('p')) die();
7                 $d="padding";
8                 $e="padding";
9         -       }
10                $f="padding";
```

Figure 5.3 Représentation du couple de versions $(A, R_{SuspiciousEdges,g}(A, B))$



(a) CFG of Ver_a (b) CFG of $R_{SuspiciousEdges,g}(Ver_a, Ver_b)$

Figure 5.4 Représentation du couple de versions en Figure 5.3

Tableau 5.2 Résumé de l'exécution de l'algorithme GuiltyEdges sur l'exemple en Figure 5.1, appliqué à la sortie de l'algorithme SuspiciousEdges

c_p	1	4
c	3	5
$vertexMap_{verB,verR}(cp)$	1	5
$(\neg defProt(vertexMap_{verB,verR}(cp))) \vee (q_{cp,1,k}, q_{c,0,k'}) \in T_{versB}$	$True$	$False$
$cp_b, c_b \in G$	$True$	$False$

```
1          $a="padding";
2          if (!current_user_can('p')) die();
3          $b="padding";
4          $c="padding";
5          $d="padding";
6          $e="padding";
7          $f="padding";
```

Figure 5.5 Représentation de la version du code réparé $\mathring{R}_{suspiciousEdges,g}(A, B)$

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Au cours de cette maitrise, nous avons ainsi étudié les résultats des analyses des Changements Impactant la Protection. Nous avons étendu l'analyse à l'ensemble des CVE et comparé les résultats à un oracle des changements. Nous avons ensuite classifié les faux négatifs de cette analyse en investiguant les versions pour lesquelles l'ensemble des PIC ne contient pas toutes les causes des différences de protection.

Nous avons ensuite introduit deux approches de réparation automatique de vulnérabilités de contrôle d'accès injectées dans une version.

La première approche utilise la réversion d'une partie des changements de la paire de versions. Nous avons pour cela introduit des définitions relatives aux fonctions de réversions, afin d'étudier leur utilisation pour générer des patchs. Plusieurs fonctions de réversion ont ainsi pu être définies, avec des éléments motivés par la classification des faux négatifs de l'analyse des PIC. Nous avons également défini une mesure permettant de les comparer entre eux, puis nous avons calculé les mesures correspondant aux différentes fonctions de réversion considérées.

Les jeux de tests utilisés comprennent au total 148 versions, sur 2 applications différentes et deux niveaux de granularité de version.

La taille des jeux de tests utilisés est du même ordre que celui des jeux de tests traditionellement utilisés pour tester la réparation automatique (22 pour Nopol [33], 16 puis 105 pour GenProg [28, 66], 18 pour ACS [32]).

Les fonctions de réversions proposées améliorent jusqu'à 47.67% le taux de réversion médian par rapport à la fonction de réversion originelle, lors de l'application de la stratégie incrémentielle sur l'ensemble de versions CVE.

L'application des différentes fonctions sur les jeux de tests montre l'apport des ajouts faits à la réversion, rev_{edge} et rev^{∞}_{lines} améliorant toutes deux le taux de réversion pour au moins l'un des jeux de tests par rapport à rev_{lines}.

Ensuite, nous avons proposé une approche utilisant la réparation par insertion de motifs, et nous avons démontré formellement que celle-ci répare les pertes de protection définitive.

6.2 Limitations de la solution proposée

6.2.1 Limites liées aux représentations intermédiaires

L'étude subit dans un premier temps les limitations classiques de l'analyse statique.

```
1          class A{
2          function foo(){
3                  A::getMethod('fee')
4                  $a='padding'
5          }
6          function fee(){
7                  if !current_user_can('p'){
8                          die();
9                  }
10         }
11         }
```

Figure 6.1 Exemple d'appel réflexif en PHP

Les appels réflexifs ne sont ainsi pas considérés lors des différents parcours. Ainsi, dans le programme présenté en Figure 6.1, l'appel réflexif de la fonction fee en ligne 3 n'est pas considéré, et l'instruction en ligne 4 n'est donc pas marquée comme définitivement protégée comme elle le serait en considérant les chemins dynamiques.

Des approximations sont également faites lors de la génération des représentations intermédiaires. Par exemple, une approximation des appels de fonctions est faite lors de la création des arcs interprocéduraux sur les noeuds d'appels. La décision de la fonction appelée ne prend pas en compte les arguments de la méthode appelée mais uniquement son nom et sa classe lors de ce choix. Ainsi, deux fonctions avec des arguments différents ne seront pas distinguées dans le graphe d'appel.

6.2.2 Limites liées à l'analyse PTFA

Une limitation de l'étude est évoquée lors de l'introduction de l'hypothèse de recherche. L'ensemble des analyses est réalisée sur le modèle PTFA relatif à la vérification d'un privilège, et les vulnérabilités considérées sont les pertes et gains définitifs de protection entre deux versions. Néanmoins, le code peut contenir certains motifs plus complexes, qui dépassent la simple vérification binaire d'un privilège donné, dont l'analyse PTFA ne peut pas permettre de détecter les anomalies.

M-A. Laverdière [19] relève entre autres deux motifs de vérification de privilège n'induisant pas de DPD.

```
1 if(current_user_can("p") || $a == $b){
2 $c="padding";
3 }
```

(a) Exemple de vérification de privilège qualifiée

```
1 if ($a){
2 $b='p1'
3 }
4 else{
5 $b='p2'
6 }
7 if(current_user_can($b)){
8 $c="padding";
9 }
```

(b) Exemple de vérification de privilège variable

Figure 6.2 Exemples de vérifications de privilèges ne causant pas de DPD

Le programme en Figure 6.2a montre une vérification de privilège qualifiée, c'est à dire couplée à un autre prédicat. Ici, l'instruction en ligne 2 n'est donc pas protégée par le privilège "p", puisqu'il existe un chemin non protégé y accédant. Le programme en Figure 6.2b montre une vérification de privilège variable, c'est-à-dire que le privilège vérifié est encapsulé dans une variable. Ici, l'instruction à la ligne 8 n'est protégé définitivement ni par le privilège "p1" ni par le privilège "p2".

Ces deux motifs sont utilisés dans les applications étudiées, et correspondent à des politiques de contrôle d'accès allant au delà de la vérification booléenne d'un privilège. Ils peuvent donc avoir des vulnérabilités associées, mais celles-ci sont hors du cadre de l'analyse PTFA, et ne sont donc pas considérées dans cette étude.

Cette limite est quantifiée par l'étude des commits de WordPress réparant les CVE relatifs aux vulnérabilités RBAC. Dans les 31 commits réparant ces CVE, 19 comportaient des DPDs. Parmis ceux ne contenant pas de DPD, 5 présentaient des motifs de vérification de privilège qualifiée, et 3 présentaient des motifs de vérification de privilèges variables. 5 des versions ne présentaient ni de motif de vérification de privilège qualifiée, ni de motif de privilèges variables, mais présentaient d'autres motifs de protection, tels que l'utilisation de tableaux pour vérifier les privilèges d'un utilisateur.

6.3 Améliorations futures

6.3.1 Extensions des ensembles de tests

Les jeux de tests utilisés comprennent au total 148 versions, sur 2 applications différentes et 2 granularités de version. Les deux applications sont en PHP, et les résultats sont peu généralisables à des applications ayant d'autres propriétés que celles étudiées, comme des

applications écrites dans un autre langage.

La première amélioration pouvant être faite à l'étude est une extension des jeux de données sur lesquels les analyses sont réalisées, permettant ainsi d'étudier un champ d'application plus large. L'utilisation de l'extracteur pour des programmes Java (Section 3.2.2) permettrait par exemple l'ouverture vers l'étude de projets Java.

6.3.2 Amélioration des taux de réversion

En Section 4.4, nous exposons les résultats des taux de réversion des différentes stratégies de réversion. Bien que nos stratégies permettent de réparer l'ensemble des DPD, nous pouvons remarquer que les taux restent relativement élevés par rapport à nos attentes pour des changements : dans l'article [19], nous identifions que 28.3% des PIC sont des changements causant réellement les DPD sur les commits réparant les CVE, ce qui montre qu'il reste une marge d'amélioration.

Le premier facteur expliquant cet écart est la stratégie de détection de PIC, retenant l'ensemble des changements appartenant au chemin d'un DPD. Cette stratégie est très conservative, et considère un grand nombre de changements n'impactant pas la sécurité.

Le second facteur est la granularité de la réversion, qui est faite par fichier dans le cadre de ce mémoire. Ce choix permet de limiter les effets de bords des réversions, comme établit en Section 3.4.3, mais élève également le taux de réversion. Une granularité plus fine permettrait de réverser moins de changements collatéraux, et d'améliorer ainsi le taux de réversion obtenu.

Les différents facteurs énoncés d'amélioration de l'ensemble des causes racines sont donc des voies à explorer pour l'amélioration du taux de réversion.

6.3.3 Application du modèle PTFA sur d'autres prédicats

Afin d'étendre les analyses et la génération de patchs à d'autres motifs de vulnérabilités, une voie de recherche est l'application du modèle à des prédicats plus complexes. Cette piste a été abordée en Section 3.2.2, lors de l'étude du prédicat *impliedAuthentication*, qui étudie la vérification du privilège *isAuthenticated* qualifié par le prédicat *needAuthentication*.

Pour les applications utilisant une politique RBAC, des prédicats relatifs aux rôles peuvent également permettre de réduire le temps d'analyse nécessaire en regroupant certains privilèges liés entre eux : la différence entre le nombre de rôles et de privilèges pour WordPress et MediaWiki (Tableau 3.1) et la linéarité de l'étude en le nombre de prédicats étudiés (Equation 3.1) montrent l'intérêt de l'exploration de cette voie.

6.3.4 Amélioration du temps de traitement de la stratégie incrémentielle

Une stratégie de réversion proposée était la stratégie incrémentielle, qui consiste à définir une séquence d'ensembles de changements pour un couple de versions d'un programme, et de tester lesquels de ces ensembles n'ont pas d'impact lors de la réversion de ce programme, pour les retirer le cas échéant.

Nous avons testé la stratégie avec comme séquence d'ensemble les ensembles de changements se trouvant dans le même fichier.

La stratégie améliore significativement le taux de réversion des fonctions de réversion, mais celle-ci est coûteuse, puisqu'elle nécessite une application de la fonction originelle pour chaque élément de la séquence d'ensembles.

Une voie d'amélioration est donc de produire une séquence d'ensembles réduisant le traitement de la stratégie.

— La première solution pour cela est d'améliorer le taux de réversion de la fonction de réversion avant application de la stratégie incrémentielle en utilisant par exemple l'une des voies proposées en Section 6.3.2. En effet, les ensembles testés sont ceux qui sont réversés par la fonction de réversion originelle. En diminuant le nombre de changements faits par cette fonction, on diminue donc le temps de traitement de la stratégie incrémentielle.

— Une autre solution est de regrouper les ensembles, afin de tester ensemble ceux qui ont de fortes chances de ne contenir aucune cause de DPD.

  Afin de calculer les probabilités qu'un ensemble contienne une cause de DPD, une possibilité serait d'entraîner un algorithme d'apprentissage sur les différents ensembles pour lesquels nous savons qu'ils contiennent une cause racine ou non. Nous aurions alors la probabilité qu'un des ensembles contienne la cause d'un DPD par rapport aux attributs de cet ensemble, et nous pourrions appliquer la stratégie sur des unions des ensembles ayant une faible probabilité de causer un DPD.

  Un travail similaire à cette proposition a été proposé par A. Barrak et al. [67], qui propose un algorithme prédisant si un commit contient ou non des changements faisant partie des PIC calculés pour les ensembles WP et MW. Pour l'appliquer à notre stratégie, nous devons changer la granularité par commit par une granularité par fichier, et appliquer la stratégie pour prédire les ensembles contenant des causes racines au lieu de l'ensemble contenant des PIC.

RÉFÉRENCES

[1] Wordpress. [En ligne]. Disponible : https://github.com/WordPress/WordPress

[2] W. Al-Ahmad, K. Al-Fagih, K. Khanfar, K. Alsamara, S. Abuleil et H. Abu-Salem, "A taxonomy of an it project failure : root causes," *International Management Review*, vol. 5, n°. 1, p. 93–104, 2009.

[3] B. Hailpern et P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, n°. 1, p. 4–12, 2002.

[4] Owasp top 10 - 2017. [En ligne]. Disponible : https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[5] Kubernetes privilege escalation and access to sensitive information in opfenshift products and services - cve-2018-1002105. [En ligne]. Disponible : https://access.redhat.com/security/vulnerabilities/3716411

[6] S. Pertet et P. Narasimhan, "Causes of failure in web applications," Technical Report CMU-PDL-05-109, Carnegie Mellon University, Rapport technique, 2005.

[7] P. Samarati et S. C. de Vimercati, "Access control : Policies, models, and mechanisms," dans *International School on Foundations of Security Analysis and Design*. Springer, 2000, p. 137–196.

[8] R. S. Sandhu et P. Samarati, "Access control : principle and practice," *IEEE communications magazine*, vol. 32, n°. 9, p. 40–48, 1994.

[9] R. S. Sandhu, "Role-based access control," dans *Advances in computers*. Elsevier, 1998, vol. 46, p. 237–286.

[10] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, n°. 7, p. 1–19, juill. 1970. [En ligne]. Disponible : http://doi.acm.org/10.1145/390013.808479

[11] B. Alpern, M. N. Wegman et F. K. Zadeck, "Detecting equality of variables in programs," dans *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1988, p. 1–11.

[12] T. Reps, S. Horwitz et M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," dans *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, p. 49–61.

[13] D. Letarte et E. Merlo, "Extraction of inter-procedural simple role privilege models from php code," dans *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, p. 187–191.

[14] F. Gauthier et E. Merlo, “Fast detection of access control vulnerabilities in php applications,” dans *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, p. 247–256.

[15] D. Letarte, F. Gauthier et E. Merlo, “Security model evolution of php web applications,” dans *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, p. 289–298.

[16] M.-A. Laverdière et E. Merlo, “Classification and distribution of rbac privilege protection changes in wordpress evolution,”,” dans *To appear in Proc. 15th Int’l Conf. Privacy, Security and Trust (PST’17)*, 2017.

[17] (2017) Gnu diffutils. GNU. [En ligne]. Disponible : https://www.gnu.org/software/diffutils/

[18] M.-A. Laverdière et E. Merlo, “Detection of protection-impacting changes during software evolution,” dans *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, p. 434–444.

[19] M.-A. Laverdière-Papineau, “Finding differences in privilege protection and their origin in role-based access control implementations,” Thèse de doctorat, École Polytechnique de Montréal, 2018.

[20] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau et P. McDaniel, “Highly precise taint analysis for android applications,” 2013.

[21] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau et P. McDaniel, “Flowdroid : Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” dans *Acm Sigplan Notices*, vol. 49, n°. 6. ACM, 2014, p. 259–269.

[22] J. Lerch, B. Hermann, E. Bodden et M. Mezini, “Flowtwist : efficient context-sensitive inside-out taint analysis for large codebases,” dans *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, p. 98–108.

[23] N. Jovanovic, C. Kruegel et E. Kirda, “Pixy : A static analysis tool for detecting web application vulnerabilities,” dans *2006 IEEE Symposium on Security and Privacy (S&P’06)*. IEEE, 2006, p. 6–pp.

[24] W. Cheng, Q. Zhao, B. Yu et S. Hiroshige, “Tainttrace : Efficient flow tracing with dynamic binary rewriting,” dans *11th IEEE Symposium on Computers and Communications (ISCC’06)*. IEEE, 2006, p. 749–754.

[25] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel et A. N. Sheth, “Taintdroid : an information-flow tracking system for realtime privacy

monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, nº. 2, p. 5, 2014.

[26] L. Wang, Q. Zhang et P. Zhao, "Automated detection of code vulnerabilities based on program analysis and model checking," dans *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008, p. 165–173.

[27] M. Monperrus, "Automatic software repair : a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, nº. 1, p. 17, 2018.

[28] C. Le Goues, T. Nguyen, S. Forrest et W. Weimer, "Genprog : A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, nº. 1, p. 54–72, Jan 2012.

[29] F. Long et M. Rinard, "Staged program repair with condition synthesis," dans *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA : ACM, 2015, p. 166–178. [En ligne]. Disponible : http://doi.acm.org/10.1145/2786805.2786811

[30] D. Kim, J. Nam, J. Song et S. Kim, "Automatic patch generation learned from human-written patches," dans *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA : IEEE Press, 2013, p. 802–811. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?id=2486788.2486893

[31] F. Long et M. Rinard, "Automatic patch generation by learning correct code," dans *ACM SIGPLAN Notices*, vol. 51, nº. 1. ACM, 2016, p. 298–312.

[32] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang et L. Zhang, "Precise condition synthesis for program repair," dans *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, p. 416–426.

[33] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre et M. Monperrus, "Nopol : Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, nº. 1, p. 34–55, Jan 2017.

[34] Z. Lin, X. Jiang, D. Xu, B. Mao et L. Xie, "Autopag : Towards automated software patch generation with source code root cause identification and repair," dans *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '07. New York, NY, USA : ACM, 2007, p. 329–340. [En ligne]. Disponible : http://doi.acm.org/10.1145/1229285.1267001

[35] S. Sidiroglou-Douskos, E. Lahtinen et M. Rinard, "Automatic discovery and patching of buffer and integer overflow errors," Computer Science and ArtificialIntelligence Labora-

tory Technical Report MIT-CSAIL-TR-2015-018, Massachusetts Institute of Technology, Rapport technique, 2015.

[36] S. R. L. Marcote et M. Monperrus, "Automatic repair of infinite loops," *CoRR*, vol. abs/1504.05078, 2015. [En ligne]. Disponible : http://arxiv.org/abs/1504.05078

[37] R. Samanta, O. Olivo et E. A. Emerson, "Cost-aware automatic program repair," dans *International Static Analysis Symposium*. Springer, 2014, p. 268–284.

[38] R. van Tonder et C. Le Goues, "Static automated program repair for heap properties," dans *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, p. 151–162.

[39] R. Just, D. Jalali et M. D. Ernst, "Defects4j : A database of existing faults to enable controlled testing studies for java programs," dans *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA : ACM, 2014, p. 437–440. [En ligne]. Disponible : http://doi.acm.org/10.1145/2610384.2628055

[40] D. Lin, J. Koppel, A. Chen et A. Solar-Lezama, "Quixbugs : A multi-lingual program repair benchmark set based on the quixey challenge," dans *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications : Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA : ACM, 2017, p. 55–56. [En ligne]. Disponible : http://doi.acm.org/10.1145/3135932.3135941

[41] T. Durieux, F. Madeiral, M. Martinez et R. Abreu, "Empirical review of java program repair tools : A large-scale experiment on 2,141 bugs and 23,551 repair attempts," *arXiv preprint arXiv :1905.11973*, 2019.

[42] E. K. Smith, E. T. Barr, C. Le Goues et Y. Brun, "Is the cure worse than the disease ? overfitting in automated program repair," dans *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, p. 532–543.

[43] Z. Qi, F. Long, S. Achour et M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," p. 24–36, 2015. [En ligne]. Disponible : http://doi.acm.org/10.1145/2771783.2771791

[44] W. Weimer, S. Forrest, C. Le Goues et T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, n°. 5, p. 109–116, 2010.

[45] L. D'Antoni, R. Samanta et R. Singh, "Qlose : Program repair with quantitative objectives," dans *International Conference on Computer Aided Verification*. Springer, 2016, p. 383–401.

[46] M. Martinez et M. Monperrus, “Mining software repair models for reasoning on the search space of automated program fixing,” *Empirical Software Engineering*, vol. 20, n°. 1, p. 176–205, Feb 2015. [En ligne]. Disponible : https://doi.org/10.1007/s10664-013-9282-8

[47] D. Kelk, K. Jalbert et J. S. Bradbury, “Automatically repairing concurrency bugs with arc,” dans *International conference on multicore software engineering, performance, and tools*. Springer, 2013, p. 73–84.

[48] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li et C. Pasareanu, “On reliability of patch correctness assessment,” p. 524–535, 2019.

[49] M. Martinez, T. Durieux, R. Sommerard, J. Xuan et M. Monperrus, “Automatic repair of real bugs in java : a large-scale experiment on the defects4j dataset,” *Empirical Software Engineering*, vol. 22, n°. 4, p. 1936–1964, Aug 2017. [En ligne]. Disponible : https://doi.org/10.1007/s10664-016-9470-4

[50] S. Saha, R. K. Saha et M. R. Prasad, “Harnessing evolution for multi-hunk program repair,” dans *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, p. 13–24.

[51] Q. Xin et S. P. Reiss, “Better code search and reuse for better program repair,” dans *Proceedings of the 6th International Workshop on Genetic Improvement*. IEEE Press, 2019, p. 10–17.

[52] X. Xu, Y. Sui, H. Yan et J. Xue, “Vfix : Value-flow-guided precise program repair for null pointer dereferences,” dans *Proceedings of ICSE*, 2019.

[53] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O’Hearn, I. Papakonstantinou, J. Purbrick et D. Rodriguez, “Moving fast with software verification,” dans *NASA Formal Methods Symposium*. Springer, 2015, p. 3–11.

[54] D. Xu et S. Peng, “Towards automatic repair of access control policies,” dans *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, Dec 2016, p. 485–492.

[55] S. Son, K. S. McKinley et V. Shmatikov, “Fix me up : Repairing access-control bugs in web applications.” dans *NDSS*, 2013.

[56] D. Muthukumaran, T. Jaeger et V. Ganapathy, “Leveraging choice to automate authorization hook placement,” dans *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, p. 145–156.

[57] S. K. Lahiri, K. Vaswani et C. A. Hoare, “Differential static analysis : opportunities, applications, and challenges,” dans *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, p. 201–204.

[58] S. K. Lahiri, K. L. McMillan, R. Sharma et C. Hawblitzel, “Differential assertion checking,” dans *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, p. 345–355.

[59] Wordpress. WordPress. [En ligne]. Disponible : https://wordpress.org

[60] (2017) Mediawiki. Wikimedia Foundation. [En ligne]. Disponible : https://www.mediawiki.org/wiki

[61] Capabilities. Moodle. [En ligne]. Disponible : https://docs.moodle.org

[62] Moodle. Moodle. [En ligne]. Disponible : https://download.moodle.org

[63] National vulnerability database. National Institute of Standards and Technology. [En ligne]. Disponible : https://nvd.nist.gov/vuln

[64] J. Dolby et M. Sridharan, “Static and dynamic program analysis using wala,” dans *Proceedings of the 2010 ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (June 2010)*, vol. 1.

[65] Owasp webgoat project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[66] C. Le Goues, M. Dewey-Vogt, S. Forrest et W. Weimer, “A systematic study of automated program repair : Fixing 55 out of 105 bugs for $8 each,” dans *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, p. 3–13.

[67] A. Barrak, M.-A. Laverdière, F. Khomh, L. An et E. Merlo, “Just-in-time detection of protection-impacting changes on wordpress and mediawiki,” dans *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 2018, p. 178–188.