UNIVERSITÉ DE MONTRÉAL

DÉTECTION, PROTECTION, ÉVOLUTION ET TEST DE DÉFAILLANCES À L'AIDE D'UN MODÈLE INTER-PROCÉDURAL SIMPLE

DOMINIC LETARTE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2011

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

DÉTECTION, PROTECTION, ÉVOLUTION ET TEST DE DÉFAILLANCES À L'AIDE D'UN MODÈLE INTER-PROCÉDURAL SIMPLE

présentée par : LETARTE Dominic.
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury constitué de :

M. GUÉHÉNEUC, Yann-Gaël, Doct., président
M. MERLO, Ettore, Ph.D., membre et directeur de recherche
Mme. BOUCHENEB, Hanifa, Doctorat, membre
Mme. TAWBI, Nadia, Ph.D., membre

Remerciements

Je tiens à remercier le Center for Advanced Studies d'IBM Canada (CAS Canada) pour le généreux soutien et particulier les personnes avec lesquelles j'ai le plus collaboré : Fahad Javed, Joanna Ng, Nattavut Sutyanyong et Calisto Zuzarte.

Résumé

Il est utile de disposer d'outils pour aider à effectuer des opérations de maintenances dans les logiciels. Plusieurs types d'outils sont disponibles à cet effet, pour cette étude nous nous concentrerons sur quatre types d'outils soit, les outils de détection, d'évolution, de correction et de tests. Cette thèse étudie ces quatre types d'outils en fonction de la maintenance à effectuer en rapport avec les défaillances de type SQL-injection dans une application Web écrite en PHP. Les approches que nous proposons emploient l'analyse statique, l'analyse dynamique, la réingénierie du code source et un algorithme génétique pour réaliser ces tâches de maintenance. Un modèle inter-procédural du code source en PHP est construit et est utilisé pour détecter et faire le suivi de l'évolution des vulnérabilités identifiées. Un modèle de requêtes SQL légitimes est construit par analyse statique et dynamique afin de protéger automatiquement les applications Web écrite en PHP. Une approche pour la génération automatique de tests basée sur un algorithme génétique est aussi présentée. Ces approches ont été validées en les appliquant sur plusieurs version d'une application connue pour contenir des vulnérabilités SQL-injections soit : le logiciel phpBB qui est une application gérant un babillard électronique qui utilise la base de données MySql pour stocker l'information d'une manière persistante. Une expérimentation a été réalisée avec la version 2.0.0 de phpBB et a permis de détecter automatiquement des vulnérabilités en utilisant une analyse statique de flux inter-procédurale. Ces résultats ont été reproduits en utilisant le modèle checking ce qui vient confirmer et renforcer l'approche. Aussi, 31 versions de phpBB ont été utilisées pour suivre l'évolution des vulnérabilités identifiées. Un algorithme génétique est utilisé pour générer des cas de tests qui visent un cas particulier dans l'application. Cette expérimentation a été effectuée sur l'optimiseur de requêtes de la base de données DB2. Les résultats montrent que l'algorithme génétique permet de générer des cas de tests plus rapidement qu'un générateur aléatoire.

Abstract

Automated tools can be helpful for doing maintenance tasks on computer software. Many kinds of tools are available for doing so; in this study we concentrate on four kinds of tools that are: detection tools, evolution tools, corrective tools and testing tools. In this thesis we study those four kinds of tools in the perspective of doing maintenance related to SQL-injections vulnerabilities in applications written in PHP. We propose to use static analysis, dynamic analysis, source code reengineering and a genetic algorithm for doing theses tasks. An inter-procedural model of the PHP source code is built for detecting SQL-injections vulnerabilities. A model of legitimate SQL queries is built by using static analysis and dynamic analysis; this model is used in an automated source code reengineering that implement an automated protection against SQL-injections vulnerabilities. An approach to automatically generate targeted testing cases by using a genetic algorithm is also presented. A case study using theses approaches have been done. We have used phpBB that is a software known for the abundance of SQL-injection vulnerabilities. An experimentation using the approach of inter-procedural static analysis has detected vulnerabilities in phpBB v2.0.0. These results have been reproduced using model checking instead of static analysis in the objective of gaining more confidence in both static analysis and model checking approaches. Also, 31 versions of phpBB have been used for studying the evolution of SQL-injections vulnerabilities. Finally, a genetic algorithm is used to automatically generate targeted testing cases. This last experimentation has been done on the DB2 database SQL query optimizer, results show that the genetic algorithm is faster then a random generator to generate targeted testing cases.

Table des matières

Liste des tableaux

Liste des figures

LISTE DES ANNEXES

Liste des sigles et abréviations

API	Interface de programmation applicative (Application Programming Interface).
AST	Arbre syntaxique abstrait (Abstract Syntax Tree).
BD	Base de données.
CFG	Graphe de flux de contrôle (Control Flow Graph).
COBOL	Langage de programmation principalement utilisé pour les applications de gestion (COmmon Business Oriented Language).
GXL	Format d'échange de graphe (Graph eXchange Language).
IFC	Analyse statique sur les flux d'information influancant le contrôle (Information Flow Control analysis).
JSA	Librairie d'analyse de chaînes de caractères en Java (Java String Analysis).
KSLOC	Millier de lignes de code source. (Kilo SLOC, voir SLOC)
LOC	Nombre de ligne de code incluant les lignes vides et les commentaires (Lines Of Codes).
NPE	Exception dû à un pointeur nul (Null Pointer Exception).
PHP	Langage de programmation principalement utilisé pour produire des pages Web dynamiques.
SGBD	Système de gestion de base de données.
SLA	Contrat de qualité de service (Service Level Agreement).
SLOC	Nombre de lignes de code source excluant les lignes vides et les commentaires (Source Lines Of Codes).
SOA	Architecture orientée services (Service Oriented Architecture).
SQL	Langage de programmation pour l'accès aux bases de données (Structured Query Language).
URL	Addresse Web sous la forme standardisé d'un *localisateur uniforme de ressource*(Uniform Resource Locator).
XML	Langage de balisage extensible (eXtensible Markup Language).

CHAPITRE 1

INTRODUCTION

La plus grande part des coûts encourus lors de la réalisation d'une application informatique, lorsque l'on considère tout son cycle de vie, sont dus aux opérations de maintenance. Quelque soit le style ou la méthode de développement utilisée, une grande part de l'effort sera consacré à des opérations de maintenance évolutive ou de maintenance corrective pour le logiciel.

Il est donc utile, voir même nécessaire, de disposer d'outils pour aider à réaliser ces opérations de maintenances. Plusieurs types d'outils sont disponibles afin d'aider les développeurs à réaliser des tâches reliées à la maintenance. Pour cette étude, nous nous concentrerons sur quatre types d'outils pour détecter des vulnérabilités soit des outils de détection, d'évolution, de correction et de tests.

Les outils de détection permettent de rechercher, d'identifier et de localiser dans le code source d'une application des données ou des patrons d'intérêts. Cette tâche peut être réalisée avec un outil généraliste comme la commande grep dans l'environnement Unix qui permet de rechercher une chaîne de caractères dans les fichiers sources d'une application. Des outils plus spécialisés sont aussi possibles comme l'analyse statique qui va prendre en compte le modèle syntaxique et sémantique de l'application permettant une compréhension beaucoup plus raffinée du fonctionnement de l'application. Ces outils permettent aux développeurs d'acquérir rapidement une meilleure compréhension de l'application en leur permettant d'identifier beaucoup plus facilement des propriétés spécifiques dans le code source. Ces tâches, effectuées par les développeurs, seraient beaucoup plus longues et ardues si elles devaient être réalisées manuellement.

Les outils analysant l'évolution d'un logiciel permettent de surveiller et de suivre des caractéristiques d'une application pendant une période de temps incluant plusieurs versions d'un même logiciel. Ces outils vont permettre d'identifier ou de quantifier les changements qui sont intervenues entre deux versions successives d'une application. Ces informations vont permettre de mieux planifier la suite du développement en aidant à prévoir et planifier, entre autres, les ressources nécessaires pour les opérations de maintenance suivantes qui découlent des dernières opérations effectuées.

Les outils de correction permettent d'effectuer des changements automatiquement dans le code source d'une application. On peut imaginer l'opération simpliste consistant à *chercher et remplacer* (search and replace) dans le code source. Des techniques plus avancées sont

aussi possibles reposant la réingénierie du code source. L'analyse statique va permettre une meilleure compréhension du code source à un niveau beaucoup plus élevé et permettre de régénérer du code incluant des opérations de maintenance sophistiquée. Non seulement un gain de temps est possible mais surtout la mécanisation de ces opérations va permettre d'obtenir un niveau de fiabilité et de discipline qu'il est difficile d'obtenir avec un développeur humain.

Les outils de test vont permettre d'aider à réaliser les tests de l'application. Ces outils peuvent aider à mécaniser et automatiser les tests rendant la réalisation de cette tâche plus facile et reproductible. Dans notre cas nous nous intéressons plus particulièrement à la création de nouveaux tests qui visent spécifiquement certaines propriétés des applications. La création de tests est une tâche qui peut sembler simple et monotone mais la création de cas de tests qui visent des propriétés particulières d'un logiciel peut être très difficile pour un humain ou pour une machine. Les outils qui permettent d'automatiser la création de tests en fonction de critères spécifiques permettent de réaliser un gain de temps appréciable.

Cette thèse étudie la réalisation de quatre opérations de maintenance en rapport avec les défaillances de type SQL-injection dans une application Web écrite en PHP ceci en utilisant une analyse inter-procédurale simple. Lorsqu'une application est développée, il est possible que certains types de menaces ne soient pas prises en compte. Soit délibérément pour simplifier ou accélérer le travail des développeurs ou bien tout simplement parce que ce type de menace n'était pas encore connu des développeurs au moment du développement original de l'application. Quoi qu'il en soit, il résulte un besoin d'effectuer des tâches de maintenance pour corriger la situation.

Les injections SQL sont possibles lorsqu'il existe une interface de programmation entre une application et une base de données reposant sur la communication de requêtes SQL en mode texte à la base de données et lorsque l'application intégre des données venant d'un attaquant potentiel à l'intérieur de requêtes transmises à la base de données. Lorsque ces deux conditions sont réunies, il est possible que l'attaquant introduise un fragment de code SQL à l'intérieur des données qui sont manipulées par l'application et réussisse ainsi à détourner l'exécution d'une requête SQL en lui faisant exécuter le fragment de code inclus. Il existe plusieurs méthodes pour prévenir ce type d'attaques, elles reposent sur l'élimination de toutes possibilités de la conjonction de ces conditions.

Nous présentons les approches que nous avons développées pour la détection, la prévention et l'évolution des SQL-injections dans un logiciel. Plusieurs technologies ont été utilisées pour mettre en œuvre ces technologies. Le Tableau 1.1 présente les technologies employées pour chacune des approches que nous présentons dans cette thèse.

L'approche que nous proposons pour la détection des vulnérabilités SQL-injection repose

Tableau 1.1 Technologies employées pour chacune des approches.

	Analyse statique	Analyse dynamique	Réingénierie du code source	Model checking	Algorithme génétique
Protection	chapitre 3	chapitre 3	chapitre 3		
Detection	chapitre 4			chapitre 7	
Evolution	chapitre 5				
Test	chapitre 6				chapitre 6

sur une analyse de flux inter-procédural. Elle permet d'identifier les énoncés qui sont susceptibles d'être vulnérables à des attaques de type SQL-injection venant d'attaquant internes ou externes à l'application. Une seconde approche reposant sur le model checking plutôt que sur l'analyse de flux vient reproduire les résultats obtenus par l'analyse statique de flux et offrir plus de possibilités de raisonnements théorique par rapport à cette approche.

L'approche que nous proposons pour la prévention corrective des attaques SQL-injection emploie l'analyse statique, l'analyse dynamique et la réingénierie du code source pour protéger automatiquement des applications existantes contre les attaques de type SQL-injection. Un modèle des requêtes SQL légitimes employés par l'application est construit par une analyse dynamique et une réingénierie du code source permet d'intégrer automatiquement des barricades de sécurité dans le code source qui vont s'assurer au moment de l'exécution de l'application que les requêtes produites sont bel et bien conformes au modèle des requêtes légitimes produit par l'analyse dynamique.

L'approche utilisée pour suivre l'évolution des énoncés vulnérables aux SQL-injections est basée sur l'analyse statique développée pour la détection de ces vulnérabilités. L'analyse de flux développée est très rapide en pratique ce qui la rend attrayante pour être utilisée avec un grand nombre de versions ou avec un processus de surveillance de l'évolution en continu. Ceci afin de pouvoir détecter en temps réel l'apparition de nouvelles vulnérabilités dans le code source d'une application.

L'approche de test proposée dans cette thèse permet de cibler un énoncé ou un enchaînement d'énoncés. Cette approche se base sur un modèle de l'application extrait par analyse statique et sur un algorithme génétique qui utilise la rétroaction d'une application instrumentée pour générer rapidement des cas de tests.

Ces approches ont été validées en les appliquant sur une application connue pour contenir des vulnérabilités SQL-injections, le logiciel phpBB est une application gérant un babillard électronique utilisant la base de données MySql pour stocker l'information d'une manière persistante. Aussi, la base de données DB2 d'IBM a été utilisée pour valider l'approche de

génération de cas de tests. Le Tableau 1.2 montre les applications et les versions utilisées pour les expérimentations réalisées dans chacun des chapitres de cette thèse.

Tableau 1.2 Logiciels utilisés pour les expérimentations.

Chapitre		phpBB v2.0.0	phpBB 1.0.0 à 2.0.23 (31 versions)	DB2 v. 9.5 Optimizer
3	Protection	X		
4	Détection par analyse de flux	X		
5	Évolution		X	
6	Test			X
7	Détection par Model Checking	X		

La détection des vulnérabilités par analyse de flux dans le chapitre 4 et la détection des vulnérabilités en utilisant le model checking du chapitre 7 ont tout les deux permis d'identifier des vulnérabilités SQL-injections présentes dans la version 2.0.0 de phpBB.

L'approche du suivi de l'évolution a été employée sur 31 versions successives de phpBB. Il a été possible d'identifier correctement l'apparition de vulnérabilités SQL-injection dans une version de phpBB, la présence continue de ces SQL-injection dans plusieurs versions ainsi que la disparition de ces vulnérabilités.

L'évaluation de la génération des tests a été effectuée sur l'optimiseur de requêtes SQL de la base de données DB2 suite à l'obtention d'une collaboration avec le *Centre for Advanced Studies* d'IBM à Toronto. L'approche que nous proposons s'est démontrée plus rapide qu'une approche de génération aléatoire pour générer des cas de test qui visent un enchaînement de règles dans l'optimiseur de requêtes de la base de données DB2.

Les travaux découlant dans cette thèse ont fait l'objet de communications scientifique sous forme d'un rapport technique [1], de deux posters dans des conférences [2, 3] de deux symposium doctoraux dans des conférences [4, 5] et de quatre articles avec comité de lecture dans des conférences [6, 7, 8, 9]. Le Tableau 1.3 présente ces communications scientifiques classées selon le thème et le chapitre de cette thèse auquel ils sont associés.

Les principales contributions de cette thèse sont :

– une analyse statique basée sur une analyse de flux inter-procédurale détectant automatiquement les énoncés vulnérables en tenant compte des écarts entre les niveaux d'autorisations effectifs et la politique de sécurité d'accès à la base de données ;
– la validation de l'analyse de flux inter-procédurale en utilisant une perspective scientifique différente soit le model checking ;

Tableau 1.3 Communications scientifiques.

chapitre 3	—	Protection	[7]
chapitre 4	—	Détection par analyse de flux	[1, 3, 6]
chapitre 5	—	Évolution	[10, 11]
chapitre 6	—	Test	[2]
chapitre 7	—	Détection par model checking	[4, 5, 8, 9]

– une approche pour la protection des applications Web écrites en PHP qui utilise une analyse dynamique pour construire automatiquement un modèle syntaxique des requêtes SQL légitimes et qui prend en compte les attaques internes et externes lors de la détection des vulnérabilités de type SQL-injection ;
– un algorithme génétique incluant une fonction d'adéquation (fitness) basée sur un modèle d'enchaînement qui est approprié pour générer des cas de test sur une application de grande taille ;
– la vérification expérimentale en utilisant l'application phpBB ou l'optimiseur de la base de données DB2 pour valider les approches de détection, de protection, d'évolution et de tests présentée dans cette thèse.

Le chapitre 2 présente une revue de la littérature. Les chapitres 3 à 7 présentent respectivement les thèmes de la protection, de la détection, de l'évolution et des tests. Le chapitre 8 conclut cette thèse. L'Annexe A présente un exemple court qui construit un automate de sécurité à partir d'un pseudo code en utilisant la technique présentée dans le chapitre 7. L'Annexe B explore deux représentations alternatives de l'automate de sécurité. L'Annexe C considère une approche pour utiliser moins de variables dans l'automate de sécurité.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Introduction

Ce chapitre présente une revue de la littérature en fonction des thèmes déjà présentés. La revue se concentre en particulier sur l'identification des outils déjà publiés qui utilisent des techniques différentes pour aborder les problèmes de détections, protection et d'évolution des vulnérabilités et aussi des techniques méta-heuristiques pour la génération de tests.

2.2 Protection

AMNESIA [12, 13] est un outil pour la protection des applications envers les SQL-injections. L'outil utilise une technique de sécurisation basée sur un modèle de sécurité et combine l'analyse statique et l'analyse dynamique. Cette approche construit un modèle des requêtes SQL légitimes qui peuvent être générées par l'application en utilisant Java String Analysis (JSA).

Les modèles utilisés par AMNESIA sont conçus comme un automate fini non-déterministe (NDFA) dont l'alphabet est celui des mots clés du langage SQL incluant les mots réservés, les opérateurs, les constantes et le symbole réservé β représentant une entrée de l'usager. À l'exécution, les requêtes générées par l'application sont validées avec l'automate construit statiquement. Notre approche a été inspirée par AMNESIA dans le sens de construire un modèle statique des requêtes légitimes, d'analyser syntaxiquement les requêtes générées pendant l'exécution et de vérifier la conformité des requêtes à partir de l'arbre syntaxique abstrait.

Il y a des différences significatives dans la manière dont les modèles statiques sont construits. AMNESIA construit un modèle statique des requêtes SQL légitimes en réalisant une analyse statique de l'application et en particulier en utilisant JSA pour déduire les requêtes SQL qu'une application peut produire. L'objectif est de distinguer les requêtes générées par l'application de celles injectées par l'usager. L'approche est plus faible pour protéger contre les attaquants internes puisque le code malicieux est reconnu par JSA comme faisant partie de l'application et par définition digne de confiance. Nous proposons de construire un modèle statique des requêtes SQL à partir de l'analyse dynamique de l'application web à partir de cas de tests et de cas d'utilisations qui sont légitimes.

Une autre différence est le langage dans lequel l'application est écrite PHP ou Java. Même si la construction de l'ensemble légitime des requêtes SQL à partir de l'analyse statique

d'une application est conceptuellement similaire pour la plupart des langages impératifs. En pratique les différences sémantiques entre les langages peuvent empêcher la réutilisation de la plus grande partie d'une analyse des chaînes de caractères écrites pour un autre langage. Par exemple une analyse des chaînes de caractères en PHP pourrait ne pas partager beaucoup de code avec JSA à cause des différences syntaxiques et sémantiques entre les deux langages. Inversement pour notre approche, l'analyseur des requêtes SQL peut être conservé.

Context-Sensitive String Evaluation [14] (CSSE) détecte et empêche les attaques SQL-injection pour les applications écrites en PHP. CSSE modifie l'interpréteur PHP pour retracer les données venant de l'usager à l'intérieur d'une requête SQL générée par l'application et modifie aussi l'interpréteur de PHP pour effectuer les vérifications appropriées pour prévenir les SQL-injections. Le pistage des données de l'usager est effectué en annotant automatiquement toutes les données venant de l'usager. Cette métadonnée nommée *impur* (tainted) sera propagée à travers toutes les opérations de manipulation de chaînes de caractères et des variables contenant des données venant de l'usager. Il n'est donc pas nécessaire d'acquérir une connaissance du fonctionnement interne de l'application web à protéger et il n'est pas nécessaire non plus d'imposer une nouvelle discipline ou une nouvelle pratique aux développeurs. La précision de la méthode est 100% mais elle ne prend pas en compte les attaquants internes à l'application qui pourrait avoir modifié le code source de l'application.

JDBC Checker [15] combine des techniques d'automates théoriques et une variation du problème de joignabilité (reachability) dans un langage libre de contexte pour faire une vérification des types dans les requêtes SQL générées. Cet outil signale des erreurs potentielles ou s'assure de leur absence à l'intérieur de requêtes SQL générées dynamiquement. Pour la détection des SQL-injections, JDBC checker joue un rôle similaire à celui de JSA pour AMNESIA.

SQL DOM [16] extrait automatiquement un modèle objet des classes fortement typées à partir du schéma d'une base de données. Ce modèle est utilisé pour générer des requêtes sécuritaires qui accèdent à la base de données à partir de code C#. Ainsi, cette approche déplace le raisonnement à partir de l'espace des requêtes générées dynamiquement qui peuvent créer des problèmes à l'exécution vers l'espace de la vérification de la conformité avec un modèle de classes au moment de la compilation d'un langage déclaratif. Ceci permet une vérification fine des types des paramètres utilisés au moment de l'exécution sur les appels à l'objet SQL DOM. Le contenu syntaxique SQL dans un paramètre fourni par un usager peut être aisément détecté et rejeté. Encore une fois, l'application sera protégée contre les données malicieuses venant d'un usager mais elle restera vulnérable aux erreurs ou aux attaquants internes.

Il existe des analogies entre SQL DOM et notre approche puisque la structure de leur

modèle objet est similaire à la structure syntaxique abstraite d'une requête lorsque les noms des tables et des colonnes propres à une application sont intégrés dans le langage analysé (comme c'est notre cas). Le modèle objet qui est construit correspond à l'ensemble de toutes les requêtes qui peuvent être construites pour un schéma d'une base de données. Cependant l'approche que nous présentons restreint l'ensemble de références aux seules requêtes qui peuvent être effectivement et légitimement utilisées à un point d'appel particulier de la base de données. Cet ensemble peut être significativement plus petit que l'ensemble de toutes les requêtes conformes au schéma de la base de données.

Safe Query Objects [17] représente les requêtes à la base de données comme des objets typés statiquement tout en supportant l'exécution à distance sur serveur de base de données. Les objets *Safe Query* utilisent une association entre les objets relationnels de la base de données et la méta programmation réflexive. Cette association est utilisée pour traduire des classes de requêtes en requêtes traditionnelles. En ce sens Safe Query Objects peut détecter et prévenir les SQL-injections de la même manière que SQL DOM.

Une approche qui utilise l'analyse statique et l'analyse dynamique pour protéger les applications contenant des procédures stockées (stored procedures) contre les SQL-injections a été présentée dans [18]. Le graphe de flux de contrôle de la procédure stockée est généré par analyse statique. Au moment de l'exécution, le graphe est utilisé pour propager des propriétés de sécurité ou de non-sécurité attribuées aux données de l'usager afin de déterminer s'ils occasionnent des changements sémantiques à la requête SQL. Cette approche fonctionne bien sur une base de donnée exemple utilisée avec SQL Server 2005 mais elle est limitée aux procédures stockées.

2.3 Vulnérabilités SQL-injection

L'analyse de l'information sur les flux de contrôles (Information Flow Control) IFC [19] est basée sur les chemins conditionnels dans le graphe des dépendances du programme (Program Dependence Graph, PDG) pour les applications Java. Ils intègrent dans leur approche le concept de la non-interférence sécuritaire pour arriver à évaluer les fuites de sécurité. La non-interférence est un concept classique dans le domaine de la sécurité [20, 21, 22].

Leur approche est complétement automatique, sensible aux flux de contrôles (flow-sensitive), sensible aux contextes d'appels (context-sensitive) et sensible aux objets de la programmation orientée objets (object-sensitive). Ils ont indiqué des problèmes pour la mise à l'échelle, un goulot d'étranglement dans l'analyse des pointeurs (point-to analysis) et un niveau de précision limité.

Notre approche est compatible avec le principe de non-interférence que nous avons appli-

qué aux niveaux d'autorisations et à la sécurité requise pour les accès à la base de données. En revanche, notre encodage spécial des contextes [23] nous permet de fusionner les contextes en un nombre réduit de sommaires de manière que la performance de l'algorithme soit linéaire avec une mise à l'échelle très prometteuse.

L'outil WebSSARI [24] fait des analyses statiques de types basés sur les flux de contrôles (Flow Sensitive Type Based) pour les applications en PHP. Les sections de code vulnérables sont instrumentées avec des barricades au moment de l'exécution protégeant contre les SQL-injections ainsi que le cross-site-scripting en provenance d'un usager du système. Un système de types à deux niveaux tainted/untainted semblable à celui de Perl est appliqué sur les entrées venant de l'usager puis un enrichissement vers des types plus précis contenant plus de niveaux (c.-à-d. : tainted-string, tainted-integer...) est présenté.

La possibilité de mise à l'échelle est bonne, la performance en terme du temps d'exécution semble raisonnable tandis que la précision de l'analyse semble permettre un nombre significatif de faux positifs. Les barricades de protection contre les SQL-injection de WebSSARI permettent de nettoyer les données de l'usager en remplaçant les caractères dangereux par des séquences équivalentes mais sécuritaires plutôt qu'en exécutant des vérifications en temps réel. Ceci peut être considéré comme une lacune à l'égard de nouvelles attaques utilisant des séquences de caractères dangereux inconnues. Aussi cette analyse est largement concentrée sur la protection contre les entrées malicieuses venant de l'usager, les attaquants internes ne sont pas pris en considération.

AMNESIA [13] est un outil de protection contre les SQL-injection qui utilise un système de sécurité basé sur un modéle qui combine l'analyse statique et l'analyse dynamique des applications en Java. Leur approche construit un modèle statique des requêtes SQL légitimes qui peuvent être générés par l'application Java en utilisant l'analyse de chaînes de caractères en Java (Java String Analyisis, JSA) [25].

Dans [26], une autre approche basée sur la validation syntaxique des requêtes est présentée. Les requêtes sont validées syntaxiquement avant d'être soumises à la base de données. Leur technique est basée sur la comparaison de l'arbre syntaxique d'une requête SQL avant l'intégration des données venant de l'usager avec l'arbre syntaxique de la requête après l'inclusion des données de l'usager.

Leur approche est efficace et peut être facilement mise à l'échelle. Les limitations sont dues à l'hypothèse que les données fournies par l'usager sont toujours représentées par des constantes dans la construction de la requête SQL et qu'ainsi les données de l'usager sont toujours des feuilles dans l'arbre syntaxique. Ce n'est pas toujours le cas pour par exemple la construction des requêtes complexes construites en plusieurs étapes qui ne font pas simplement remplacer des constantes par des données de l'usager. Ou bien pour les utilisateurs

privilégiés les données de l'usager peuvent directement contenir une partie de la requête SQL. Leur approche pourrait ne pas être suffisamment sensible aux changements des noms des tables et des colonnes dans les requêtes alors que l'on assume que différentes tables peuvent avoir des autorisations d'accès différentes.

2.4 Évolution du logiciel

Le problème de l'étude de l'évolution des logiciels peut être retracé jusqu'aux premiers travaux de Lehman [27]. Récemment, plusieurs travaux ont été consacrés à l'analyse des logiciels sur plusieurs versions [28, 29, 30, 31, 32]. Les travaux [28, 29, 30] partagent l'idée d'utiliser des métriques pour modéliser l'évolution des logiciels. Lehman, Perry et Ramil [30] comparent l'évolution générale de plusieurs grands systèmes pour en déduire des répercussions sur la maintenance de ces systèmes. Gall *et al.* [29] utilisent une base de données des versions du logiciel produit pour observer l'évolution de l'équipement de télécommunication afin de prédire les modules de ces équipements qui devraient être sujets à une réingénierie. Cette approche a été étendue aux systèmes orientés objets par Mattsson et Bosch dans [28]. Cette méthode récupère le taux de changement dans chacun des modules et met en évidence l'évolution du système à une variété de niveaux de détails.

Burd et Munro [31] étudient l'évolution du regroupement des grappes des données et des sections de code COBOL par une analyse des changements (ajout, retrait et modification) dans les composantes des grappes. L'objectif étant l'identification des modules candidats à la réingénierie plutôt que la traçabilité des versions du logiciel.

Holt et Pak [32] ont développé GASE, un analyseur graphique pour l'analyse de l'évolution des logiciels. L'outil met en évidence les changements entre les versions d'un logiciel au niveau architectural. L'approche de visualisation de GASE comporte une composante permettant de comparer des paires d'éléments avec une mise en évidence à l'aide de couleurs.

Antoniol et al. [33] présentent une méthode pour construire et maintenir des propriétés et des liens de traçabilités pendant l'évolution d'une série de versions de logiciels orientées objets, l'étude correspond à 9 versions de la librairie LEDA et 31 versions de l'outil DDD.

L'évolution du noyau de Linux a été observée dans [34]. Dix-neuf versions de 2.4.0 à 2.4.18 ont étés traitées et analysées en identifiant le code dupliqué dans les modules internes au moyen d'une approche basée sur des métriques. De surcroît le code dupliqué tend à rester stable pendant plusieurs versions suggérant une structure stable qui évolue en douceur.

Une représentation approximative des caractéristiques du code orientées objets ayant pour but la modélisation et l'analyse des grands systèmes orientées objets a été présentée

dans [35]. Un algorithme avec une complexité linéaire est utilisé pour détecter le code dupliqué ou presque dupliqué dans des systèmes orienté objets sur 11 versions du système Eclipse.

Une approche automatique inspirée de la recherche d'information dans un espace vectoriel a été présentée dans [36]. Elle détecte les interruptions dans l'évolution des classes. Ainsi, elle retrace le cycle de vie d'une classe même lorsqu'une classe disparaît à cause d'un remplacement par une classe similaire ou pour des raisons de fusionnement ou de séparation des classes pouvant ainsi identifier les cas possibles de refactorisation. L'approche a été utilisée pour identifier des opérations de refactorisations réelles effectuées sur plus de 40 versions d'un serveur de domaine open source en Java.

2.5 Algorithmes génétiques et test des SGBD

RAGS [37] (Random Generation of SQL) génère stochastiquement des requêtes SQL valides plus rapidement qu'un humain pourrait le faire. Il exécute aussi les requêtes, il peut identifier des erreurs observables comme les pertes de connection, les erreurs de compilation, les erreurs d'exécution, les plantages du SGBD mais il n'utilise pas de rétroaction de la part du système qui est testé.

Bati, Giakoumakis, Herbert et Surna [38] utilisent une approche génétique pour tester une base de données. Ils utilisent une rétroaction venant de la base de données sous test dans le but de guider le processus de génération des tests vers un composant spécifique de la base de données. Les cas de tests sont créés incrémentiellement en utilisant une approche génétique qui vise une composante dans son ensemble comme l'optimiseur de requêtes ou l'exécuteur de requêtes mais qui ne permet pas de viser spécifiquement une règle ou un petit groupe de règles individuellement.

Elmongui, Narasayya et Ramamurthy [39] présente une infrastructure pour tester des règles de transformations des requêtes SQL avec pour objectif de tester l'exactitude et la couverture des règles. La base de données est étendue avec un nouvel API qui exporte des patrons dérivés des définitions des règles. Cette technique est efficace pour viser une règle individuellement dans l'optimiseur mais elle est plus proche d'une approche stochastique et elle requiert que l'optimiseur construise et rende disponible un nouvel API.

Mishra, Koudas et Zuzarte [40] présentent une technique pour générer des requêtes de test qui tiennent explicitement en compte les données présentes dans la base de données. Ils étudient le problème de générer des requêtes qui vont satisfaire des contraintes de cardinalités sur des sous-expressions intermédiaires précises. Ils peuvent viser un problème de test précis mais ils n'utilisent pas d'approches méta-heuristiques ou de rétroaction lors de l'exécution de la requête pour raffiner les requêtes qui sont générées.

Plusieurs évaluations empiriques des algorithmes génétiques pour la génération de données de test ont été effectuées en dehors du domaine des optimiseurs de requêtes SQL la plupart du temps sur des programmes de petites tailles.

Romano, Di Penta et Antoniol [41] utilisent une approche basée sur la recherche (search based approach) pour générer des données de test pour identifier les exceptions dues à des pointeurs nuls dans du code source Java. L'approche utilise d'abord une analyse statique inter-procédurale du code source pour identifier les chemins entre les paramètres en entrées et les exceptions potentielles dues à des pointeurs nuls (NPE). La deuxième étape utilise un algorithme génétique pour générer des cas de test qui vont couvrir les chemins identifiés. L'approche a été utilisée et évaluée sur six programmes Java open source où des bogues de type NPE ont été artificiellement introduits.

Wong et al. [42] génèrent des fonctions utilitaires pour détecter les anomalies logicielles dans les systèmes informatiques autonomiques. Leur approche ne génère pas de cas de tests pour le système évalué mais surveille l'état des ressources comme l'utilisation de la mémoire, l'utilisation du processeur, le nombre de fils d'exécution pour déterminer l'état de santé du système. Un algorithme génétique est utilisé pour produire automatiquement de telles fonctions d'auto-surveillance. Une évaluation a été effectuée sur le serveur web Jigsaw (environ 100 000 lignes de code source) en utilisant dix métriques et cinq types d'anomalies logicielles à détecter.

Di Penta et al. [43] utilisent un algorithme génétique pour générer des données et des configurations pour tester les contrats de qualité de service (SLA) dans les architectures orientées services (SOA). Leur approche a été testée avec deux services : un service gérant un workflow de production audio et un service de génération de graphiques.

Pargas, Harrold et Peck [44] utilisent un algorithme génétique pour générer automatiquement des données de test, une étude de cas a été effectuée sur six programmes en C de ving et un à quatre-vingt-deux lignes de code. L'utilisation du parallélisme a amélioré le temps d'exécution d'une manière presque linéaire, le parallélisme a été obtenu en exécutant plusieurs processus simultanément sur un réseau de station de travail Sun.

Michael, Mcgraw et Schatz [45] examinent l'efficacité des algorithmes génétiques sur neuf programmes d'environ trente lignes de code et sur un autre programme plus grand. Le plus grand programme est une application réelle pour le pilotage automatique d'un appareil, l'application est nommée b737 et est généré par un outil CASE avec 2 046 SLOC incluant 69 points de décisions et 75 conditions. Pour le programme plus grand, l'expérience a requis entre 30 minutes jusqu'à plusieurs heures d'exécution sur une station de travail Sparc-10 mais il est suggéré qu'il serait possible d'obtenir de meilleurs temps d'exécution parce que l'environnement n'était pas optimisé pour ce type d'expériences.

Wegener, Baresel et Sthamer [46] ont développé un environnement de test évolutionnaire qui génère d'une manière complètement automatique des données de test pour la plupart des méthodes de test structurel. Sept objets de tests allant de 5 à 154 lignes de codes ont été testés, les temps d'exécutions pour les sept objets varient entre 160 et 254 secondes.

Mansour et Saleme [47] évaluent la génération de données pour les tests des chemins. Ils ont comparé empiriquement les techniques de recuit simulé (simulated anealing), les algorithmes génétiques et la recherche locale. Des tests ont été effectués sur huit programmes entre dix-sept et quatre-vingt-cinq lignes de code source, les programmes acceptaient des nombres entiers ou réels comme entrées. Une idée du temps d'exécution est donnée en mentionnant que le temps d'exécution moyen pour l'algorithme génétique sur un des programmes avec des données entières est de 0.5 s sur un Pentium 500 MHz.

Harman [48] présente une analyse théorique des scénarios dans lesquelles des algorithmes évolutifs sont plus appropriés pour la génération de cas de tests structurels. Ces prédictions théoriques sont validées par des observations empiriques sur six programmes réels allant de 70 à 2 210 lignes de code source. Lorsque les types de données inclus dans la signature d'une fonction sont explicites et simples, des vecteurs de nombres entiers générés par la méthode de recherche peuvent être directement utilisés comme valeurs d'entrées pour la fonction. Dans les autres cas, les valeurs d'entrées doivent être associées avec des types structurés.

Ghiduk, Harrold et Girgis [49] présentent une technique automatique de génération de données de tests qui utilise un algorithme génétique pour générer des données de test. Cette technique applique le concept de la relation de dominance (paires defs-use) pour définir une nouvelle fonction d'adéquation à objectif multiples. Les résultats d'une série d'études empiriques sont présentés, un ensemble de neuf programmes allant de 20 à 61 lignes de code source ont étés utilisés avec un algorithme génétique, les temps d'exécution varient de 3 à 314 secondes.

Andrews, Li et Menzies [50] décrivent un générateur génétique-aléatoire de données de tests à deux niveaux qui utilise d'abord un algorithme génétique pour identifier les paramètres de génération et ensuite génère aléatoirement des données pour des tests unitaires qui optimise la couverture du code. Une expérience par étude de cas a été effectuée sur seize classes concrètes avec des constructeurs publiques dans `java.util`. La taille des classes étant de 9 à 562 lignes de code source avec des temps d'exécution moyen de 6.2 s sur un Sun UltraSPARC-IIIi pour obtenir une couverture de 100% des conditions/décisions faisables.

Aussi approprié, Kapfhammer et Soffa [51] présentent une technique de surveillance sensible aux bases de données pour tester les applications basées sur une base de données.

2.6 Modèle inter-procédural de rôles et de privilèges

Des approches globales et générales au model checking inter-procédural sont présentées dans [52, 53, 54]. Ces approches modélisent précisément les aspects inter-procéduraux de l'analyse du code source mais elles sont complexes à mettre en œuvre et à exécuter. L'approche que nous présentons dans le chapitre 7 peut être considérée comme un cas particulier d'une approche globale. L'approche propage un niveau de sécurité binaire dans un modèle inter-procédural et une propriété de sécurité simple doit être vérifiée. Dans ce contexte, notre approche peut requérir moins de ressources et être algorithmiquement moins complexe selon la taille du système analysé. Plus de travaux sont cependant requis pour préciser la performance sur de plus grands systèmes.

Dans [55], une technique pour calculer les droits d'accès requis en utilisant une analyse de flux inter-procédurale sensible aux flux de données et aux flux de contrôles est présentée. Le problème visé est de déterminer l'ensemble de droits de sécurité requis pour exécuter une application. Ceci est un problème complémentaire au problème étudié. Dans le contexte de [55], la sécurité requise par les bibliothèques de fonctions utilisées est connue mais la sécurité nécessaire par le code source doit être déterminée. Dans le contexte du chapitre 7, les requis de sécurité des opérations dans le code source sont connus et nous voulons en vérifier la conformité.

Dans [56], une approche basée sur une analyse statique précise et pouvant être mise à l'échelle qui effectue une analyse de pointeurs est proposée pour détecter des vulnérabilités comme les injections SQL, le cross-site scripting et les attaques de *http splitting*. Toutes les vulnérabilités présentées répondent à un patron spécifique dans le code source analysé statiquement, les vulnérabilités sont détectées en utilisant la perspective d'une propagation booléenne des objets infectés (tainted analysis).

Leur approche qui résout le problème de la propagation booléenne d'objets d'informations infectés est similaire à notre approche selon cette perspective. Néanmoins, en modélisant le problème de la sécurité booléenne en utilisant une méthode formelle et en utilisant le model checking nous obtenons la possibilité d'extensions vers des modèles plus complexes et vers la vérifications de propriétés arbitraires sur le modèle de sécurité extrait cependant cette perspective n'est pas encore développée.

Dans [57], Schmidt simplifie et clarifie la représentation des analyses de flux en tant que vérification formelle en employant l'interprétation abstraite pour générer des traces de programmes et en employant le mu-calcul pour exprimer des propriétés sur les traces. Un effet surprenant de ce travail est la mise en évidence que deux analyses de flux classiques étaient problématiques mais ont pu être corrigées simplement.

Notre approche est différent de [57] parce que nous n'utilisons pas de traces ni le mu-calcul mais nous utilisons plutôt le graphe d'appel inter-procédural d'un système ainsi que des propriétés de logique temporelle simple. La complexité de l'évaluation du point fixe du modèle de sécurité est linéaire pour notre modèle de sécurité booléenne. Nous partageons avec [57] la perspective de valider à l'aide de méthodes formelles une analyse de flux algorithmique, de plus la complexité linéaire de l'analyse de flux est préservée.

CHAPITRE 3

PROTECTION DES APPLICATIONS EN PHP CONTRE LES ATTAQUES PAR INJECTION SQL

Les sites Web peuvent être des sites statiques, des programmes ou des bases de données. Cependant, la plupart du temps c'est une combinaison des trois utilisant une base de données relationnelle comme un outil de stockage des données. Les sites Web requièrent beaucoup d'attention pour la configuration et la programmation afin d'assurer la sécurité, la confidentialité et la légitimité de l'information échangée via le site Web.

Les attaques de type injection SQL exploitent des défaillances dans la validation des données textuelles utilisées pour construire les requêtes à la base de données. Des données confectionnées d'une manière malicieuse peuvent menacer la confidentialité et déjouer les politiques de sécurité d'un site Web utilisant sur une base de données pour stocker ou accéder à de l'information.

Ce chapitre présente une approche originale qui combine l'analyse statique, l'analyse dynamique et la réingénierie du code source pour protéger automatiquement les applications écrites en PHP contre les attaques de type injection SQL.

Ce chapitre présente aussi les résultats d'une expérimentation effectuée sur une version ancienne de phpBB (version 2.0.0, 37 193 lignes de codes en PHP version 4.2.2). Les résultats montrent que notre approche améliore d'une manière significative la résistance de phpBB v2.0.0 aux attaques de type injection SQL.

3.1 Introduction

Les applications Web sont utilisées pour échanger de l'information avec des usagers à travers un réseau de communication. Ces échanges impliquent souvent des interactions initiées par un usager qui peuvent déclencher l'émission d'une requête à une base de données. Malheureusement, ces interactions sont souvent fondées sur l'hypothèse que les données envoyées par l'usager sont légitimes. Ensuite, ces données seront utilisées pour construire une requête SQL qui sera évaluée par la base de données. Ces applications seront alors vulnérables à des attaques de type injection sql qui exploitent des défaillances dans la validation et la sécurisation des données textuelles soumises par l'usager.

1. Ce chapitre a été publié sous le titre « Automated Protection of PHP Applications Against SQL-injection Attacks » [7] lors de la conférence *11th European Conference on Software Maintenance and Reengineering* (CSMR).

Dans certains cas, des données construites d'une manière malicieuse contiendront un fragment de code en langage SQL. Les requêtes à la base de données qui intégreront ce fragment de code produiront des requêtes dont la sémantiques est différente de celle voulue par les développeurs de l'application Web et pourrait compromettre les politiques de sécurités de la base de données sous-jacente.

Les attaques de type injection SQL ont été décrites dans la littérature [58, 59]. Même si le fonctionnement des attaques injection SQL est bien compris et que des mécanismes pour s'en protéger existent, comme par exemple : la programmation défensive, une validation sophistiquée des entrées, la validation dynamique et l'analyse statique du code source, le problème persiste quand même pour plusieurs raisons. La programmation défensive et la validation de données entrantes visent à prévenir l'insertion de chaînes de caractères malicieuses dans les requêtes SQL mais elles requièrent une bonne discipline de la part des programmeurs qui doivent utiliser ces techniques systématiquement. Elles sont de fait plus appropriées pour les nouveaux développements qui sont bien planifiés et exécutés. Elle demandent un effort de programmation appréciable et sont plus difficiles à mettre en œuvre sur des systèmes existants. La validation dynamique des requêtes générés est mieux adaptée aux systèmes existants mais elle est susceptible à l'obsolescence après avoir été mise en œuvre. À mesure que l'application évolue, de nouvelles interactions légitimes sont ajoutées, modifiées ou retirées qui devront être reflétées dans les validations dynamiques sous peine de voir apparaître des faux positifs ou des faux négatifs. Finalement, l'analyse dynamique peut souffrir d'un manque de précision et d'un manque de performance à l'exécution lorsque des constructions complexes sont employées dans le langage étudié.

Une approche originale pour protéger automatiquement les applications écrites en PHP contre les attaques de type injection SQL est présentée ici. L'approche combine l' *analyse statique* du code source PHP avec l'analyse statique des requêtes SQL et avec l'*analyse dynamique* pour construire un modèle syntaxique des requêtes SQL légitimes. L'approche utilise aussi la *réingénierie du code source* pour protéger les applications existantes des attaques mentionnées.

Les contributions principales de ce chapitre sont :

- une analyse dynamique pour construire automatiquement un modèle syntaxique des requêtes SQL légitimes. Ces modèles sont utilisés pour construire des barricades contre les injections SQL ;
- la protection automatique des applications existantes écrites en PHP en insérant automatiquement des barricades de protection dans le code source ;
- une évaluation expérimentale de l'approche en utilisant un jeu de test légitime et un jeu d'attaques.

La section 3.2 introduit l'approche proposée pour prévenir les attaques de type injection SQL. La section 3.3 décrit la construction automatique des barricades basées sur un modèle syntaxique. La section 3.4 présente une expérimentation basée sur un cas d'étude. La section 3.5 discute des résultats présentés.

3.2 Approche préventive

En général, les attaques reposant sur des injections et, en particulier, les attaques de type injection SQL exploitent des défaillances dans la vérification des données entrantes qui permettent à l'attaquant de manipuler les requêtes SQL qui seront exécutées par la base de données.

La communication entre l'application et la base de données est établie avec des appels à un API de programmation de la base de données. Un exemple typique est l'appel $mysql(str, db)$ où *str* est une chaîne de caractères textuelles qui contient une requête SQL et *db* est une référence à la base de données employée.

Plusieurs serveurs de base de données du domaine publique ou commerciaux sont disponibles. Puisque nous implantons nos barricades de protection à l'intérieur de l'application Web et avant la base de données, les barricades ne sont pas dépendantes de la base de données spécifique employée. Aussi, nous ne ferons pas la différence entre une base de données (BD) et un système de gestion d'une base de donnée (SGBD) à moins que la différence ne soit nécessaire. Pour simplifier les explications et l'expérimentation, nous supposons l'utilisation du système de gestion d'une base de données MySql [60]. Nous supposons aussi que le gestionnaire de la base de donnée implémente un sous-ensemble étendu du standard SQL [61].

Comme proposé dans [12], le modèle de protection présenté requiert que toute communication entre les variables employées dans l'application (contenant des données de l'application ou de l'usager) et l'API de la base de données passe par des *barricades de protection basées sur un modèle des requêtes* (secured model-based guards) qui vont effectuer les vérifications de sécurité appropriées.

Cette approche déplace le fardeau de la prévention et de la protection envers les attaques de type injection SQL de l'ensemble des programmeurs vers une plus petite équipe de programmeurs spécialisés responsable des questions de sécurité. Cette équipe spécialisée est responsable du développement et de l'entretien des *barricades de protection*. De plus, la taille du code qui doit être révisé et sécurisé est réduite d'une manière importante puisqu'au lieu de devoir appliquer des politiques de sécurisation sur l'ensemble de l'application, l'approche proposée permet de cibler la politique de sécurisation seulement sur les *barricades de protection* qui sont d'une taille réduite comparée à l'ensemble de l'application.

Les attaques par injection exploitent la communication en format textuel entre l'application et l'API de communication de la base de données. Lorsque des variables venant du langage de programmation sont transformées et intégrées dans la communication textuelle entre une application Web et la base de données, les informations sémantiques sur les types des variables sont perdues.

Comme décrit dans [17], la situation serait différente si la communication s'effectuait au travers d'un API qui conserve l'information sur les types des variables. Dans un API utilisant une architecture typée, les variables peuvent être vérifiées syntaxiquement, le domaine des variables peut aussi être vérifié et, dans une certaine mesure, la sémantique peut ainsi être validée.

C'est une approche envisageable pour un nouveau système mais les applications existantes utilisent pour la plupart un API basé sur des chaînes de caractères pour le dialogue entre la base de données et l'application Web. Dans ce contexte, les vérifications de sécurité doivent être effectuées sur les chaînes de caractères qui sont passées comme paramètres aux routines de l'API de communication.

Aussi, des chaînes de caractères qui contiennent des données malicieuses peuvent être stockées dans la base de données et être récupérées pour être traitées plus tard. Ce traitement pourra alors être corrompu par les données malicieuses. Ce problème est généralement une attaque de second ordre ou d'ordre généralement plus élevée [62]. Lorsque des données sont stockées dans la base de données et qu'elles contiennent un fragment de code en SQL, sa récupération et son utilisation subséquente dans une autre requête SQL peut conduire à une attaque par injection SQL de second ordre (higher order SQL-injection). Il est probable que les énoncés vulnérables aux attaques par injection SQL d'ordre plus élevé ne sont les mêmes qui sont vulnérables aux simples attaques par injection SQL du premier ordre.

Une autre possibilité d'attaque par injection SQL vient de codes sources malicieux qui peuvent être introduits dans l'application Web par un attaquant interne pouvant modifier le code source. Un code malicieux pourra effectuer des changements dans les requêtes SQL textuelles selon certaines données entrantes. Un exemple pourrait être l'ajout de certaines requêtes additionnelles si le nom de l'usager en cours est un nom spécifique.

Dans ce chapitre, nous explorons la construction automatique de barricades basées sur les résultats d'une analyse dynamique en utilisant des scénarios d'exécutions légitimes.

3.3 Construction des barricades sur basées un modèle syntaxique

Notre approche construit des barricades de sécurité qui sont en fait des analyseurs syntaxiques spécifiques le language SQL qui produisent une séquence de jetons représentant une

requête SQL. Les barricades filtrent les requêtes aux différents points d'appel de la base de données en comparant les séquences de jetons avec un ensemble de référence qui a été obtenu par une analyse dynamique de l'application évaluée.

Nous utilisons le terme *logiciel sensible aux injection SQL* pour désigner un logiciel dans lequel une attaque de type injection SQL serait possible. Les *énoncés vulnérables* sont des énoncés effectuant des requêtes à la BD pouvant contenir en partie des données venant de l'usager. Ces données venant de l'usager sont susceptibles de provoquer une attaque du type injection SQL. Nous utilisons le terme *données révélant une attaque injection SQL* pour les données entrantes ou stockées qui contribuent à réussir une attaque par injection SQL sur un *énoncé vulnérable*. Les opérations qui permettent l'entrée de *données révélant une attaque par injection SQL* sont appelées *attaques par injection SQL*, similairement du code malicieux qui cause le même problème est appelé *programme révélant une attaque par injection SQL*. Un exemple tel apparaît quand un nom d'usager spécifique permet d'obtenir l'accès à des tables confidentielles de la base de données en utilisant une requête conçue spécialement et malicieusement pour accéder à la base de données. Clairement, pour obtenir une attaque par injection SQL une conjonction de plusieurs éléments doit être présente : des *données révélant une attaque par injection SQL* ou un *programme révélant une attaque par injection SQL* et un *énoncé vulnérable*.

Notre approche est basée sur la prévention de cette *conjonction inappropriée* qui réunit des données entrantes, du code et des accès vulnérables à la base de données qui va permettre de réaliser une attaque par injection SQL. Intuitivement, notre approche va insérer des barricades avant les accès à la base de données, celles-ci vont empêcher la conjonction des conditions permettant une attaque par injection SQL de se mettre en place.

Tel que montré dans la Figure 3.1, nous instrumentons d'abord le code PHP pour recueillir un ensemble de requêtes SQL qui sont utilisées d'une manière légitime aux différents points d'appels à la base de données. Ces requêtes seront considérées comme des cas de tests légitimes (trusted test cases) comme expliqué avec plus de détails dans la section 3.4. L'instrumentation n'est pas difficile à réaliser puisqu'il est simplement nécessaire d'ajouter une instruction d'écriture avant chaque appel à la base de données comme dans la Figure 3.2.

Après avoir instrumenté le code source PHP, nous exécutons les cas de tests légitimes pour recueillir les requêtes textuelles correspondant aux requêtes SQL légitimes générées aux différents points d'appels à la base de données. Le processus requis pour construire les barricades à partir de la représentation textuelle des requêtes SQL est simple et direct. Les requêtes légitimes sont analysées à l'aide d'un analyseur syntaxique et les arbres syntaxiques abstrait (AST) résultants sont conservés pour chacun des points d'appels à la base de données. Tous les AST sont stockés indépendamment ainsi il n'y a pas de généralisation effectuées entre

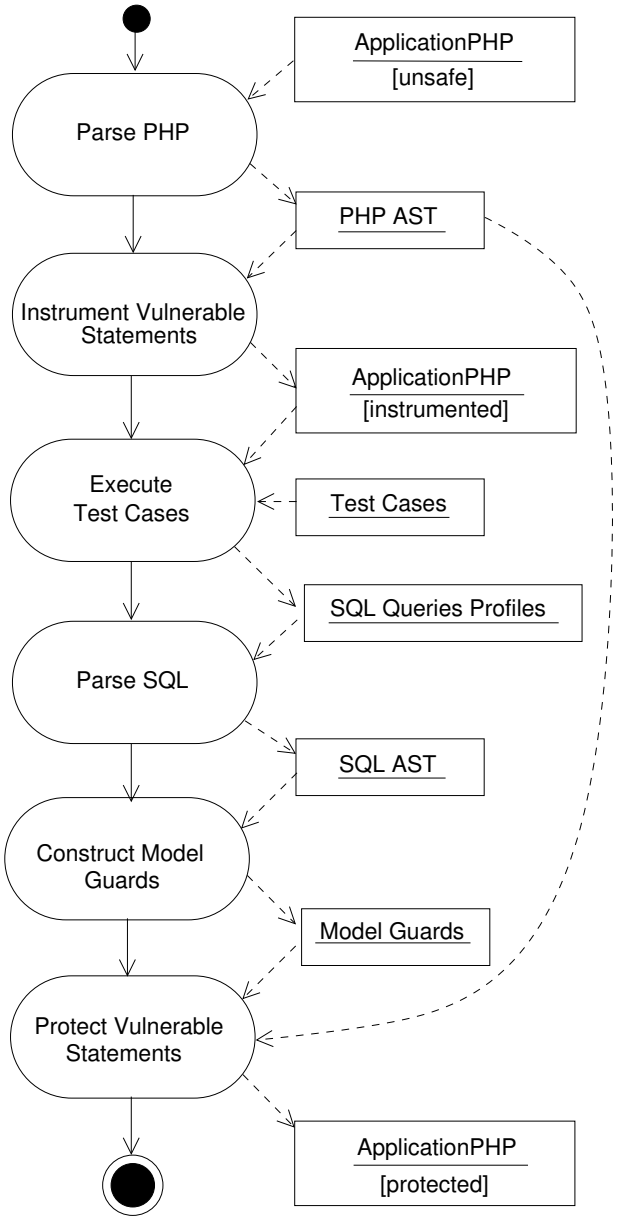

Figure 3.1 Diagramme d'activités décrivant l'automatisation de la protection

les requêtes à un même point d'appel. Cependant les AST sont tout de même une forme de généralisation puisque les constantes, les chaînes de caractères et d'autres types de données pouvant provenir de l'usager sont stockés par type de jetons syntaxique plutôt que par la valeur de leurs représentations textuelles. Réciproquement, les identificateurs normalement générés par l'application qui réfèrent au schéma de la base de données sont considérés comme faisant partie de la structure syntaxique de la requête. Ainsi les noms des tables et des colonnes sont considérés partie prenante de la structure syntaxique. Ceci permet d'empêcher les substitutions malicieuses des noms des tables et des colonnes dans les requêtes autrement

```
...
mysql(str, db)
...
```

deviendra :

```
...
fwrite(file, str)
mysql(str, db)
...
```

Figure 3.2 Instrumentation produisant les cas de tests légitimes.

légitimes. Ainsi plusieurs requêtes SQL peuvent partager la même représentation syntaxique mais avec des valeurs différentes pour certaines données.

Une barricade de sécurité à un point d'appel spécifique invoquera l'analyseur syntaxique de SQL avec la requête courante pour obtenir l'AST correspondant à la requête en cours. La représentation textuelle de l'AST sous la forme d'une liste de types de jetons sera comparé aux AST légitimes stockés pour ce même site d'appel à la base de données. La présence d'un AST identique à celui en cours d'évaluation dans l'ensemble des AST légitimes permettra à la requête d'être transmise à la base de données pour être évaluée normalement. À l'opposé, une requête dont l'AST est introuvable dans l'ensemble des AST légitimes pour ce point d'appel sera bloquée par une approche conservatrice de la sécurité de l'application.

Les barricades de sécurité sont en fait des analyseurs syntaxiques spécifiques pour ces sous-langages qui filtrent les requêtes aux différents points d'appel de la base de données. En pratique, les sous-langages sont suffisamment simples qu'ils sont représenté comme une séquence de jetons. La comparaison entre les séquence de jetons peut alors être simplement être une comparaison lexicographique d'une chaîne de caractère. La recherche dans un ensemble de référence peut facilement et efficacement être implanté comme des tables de hachages. La chaîne d'une séquences de jeton représente alors les feuilles de l'arbre syntaxique représenté en ordre infixe et précédé par l'identificateur du point d'appel à la base de données.

Nous avons en effet stocké les AST légitimes comme une chaîne de jetons (token) où les jetons sont représentés par leurs types ou par leurs valeurs textuelles selon le choix déjà expliqué. Les noms des tables utilisées localement par l'application ont été considérés comme des jetons du type des identificateurs. Désormais, nous appellerons ces chaînes de jetons des *patrons de référence*.

Un exemple d'une requête légitime est illustré par la Figure 3.3 qui est obtenu à partir

de l'appel à la base de données présent à la ligne 221 du fichier *search.php*. Pour obtenir plus de pécision le nom des colonnes présentes dans les tables de la base de données sont reconnues par le parseur comme des type de jetons différent pour chacune des colonnes. Dans cette requête, *phpbb200t_posts* est le nom d'une table dépendant de la configuration locale de phpBB tandis que *post_id* et *poster_id* sont des noms qui sont fixés dans la logique de l'application ce qui permet des les utiliser facilement dans le parseur. Le patron de référence illustré est produit ainsi.

La requête SQL émise la ligne 221 du fichier *search.php* :

```
1 SELECT post_id
2 FROM phpbb200t_posts
3 WHERE poster_id = 2 ;
```

produira le patron de référence suivant :

```
search.php:221 SELECT POST_ID FROM SQL_ID WHERE POSTER_ID OP_EQUAL INTEGER
```

Figure 3.3 Requête SQL et patron de référence correspondant.

,

On peut remarquer en observant la Figure 3.3 que *phpbb200t_post* a été reconnu comme un identificateur tandis que *post_id* et *poster_id* ont été reconnus comme des mots clés de l'application. Ceci permet d'empêcher qu'une attaque puisse conserver la même structure syntaxique mais changer le nom de certains champs avec le nom de champs contenant des données sensibles. Dans l'exemple, *post_id* et *poster_id* peuvent être utilisés si une requête légitime avec la même structure syntaxique contient les mêmes identificateurs mais ne peuvent pas être changés pour d'autres identificateurs s'il n'y a pas de requêtes correspondantes dans les requêtes légitimes. D'une manière similaire, la valeur entière 2 pourra seulement être remplacé par une valeur du même type et non pas d'un type différent. Plus d'exemples à propos des patrons de référence sont présentés dans la section suivante.

Les énoncés vulnérables peuvent être sécurisés en remplaçant ces énoncés par des barricades basées sur les patrons de références qui vont effectuer les vérifications appropriées pour chacun des points d'appels à la base de données avant de permettre que la requête soit exécutée comme illustré dans la Figure 3.4.

L'implantation des barricades basées sur les patrons de références et la réingénierie du code modifiant les appels de $mysql(str, db)$ en des appels aux barricades demande peu d'efforts. Un exemple d'une barricade écrite en Java et utilisant des patrons de référence est donné par la Figure 3.5.

$$\ldots$$
$$call\_site_i : mysql(str,\ db)$$
$$\ldots$$

devient :

$$\ldots$$
$$call\_site_i : sqlModelGuard\_i(str,\ db)$$
$$\ldots$$

Figure 3.4 Remplacement d'un énoncé vulnérable par une barricade de sécurité.

,

3.4 Expérimentation

Pour valider notre approche, nous avons adopté une perspective orientée sur la réalisation d'une preuve de concept visant à rendre une application existante sécuritaire du point de vue des attaques par injection SQL internes ou externes à l'application.

Pour notre expérimentation nous avons choisi la version 2.0.0 de phpBB en association avec la version 4.0.26 de MySql. Des failles de sécurité ont été divulguées publiquement pour cette version de phpBB. Notre objectif étant de prendre une ancienne version comportant des défaillances et de la transformer en une nouvelle application avec les mêmes fonctionnalités mais sans les vulnérabilités aux attaques injection SQL.

Nous avons trouvé 285 occurrences d'énoncés vulnérables parmi les 406 énoncés effectuant un appel à la base de données dans la version de phpBB étudiée. Nous avons utilisé l'analyse des vulnérabilités décrite dans [6] pour identifier ces vulnérabilités. Pour recueillir les patrons des requêtes légitimes, nous avons instrumenté automatiquement phpBB en insérant des sondes de sécurité avant les énoncés vulnérables. Les sondes sont une instruction simple qui enregistre la requête SQL en cours dans un fichier texte qui pourra être récupéré plus tard.

Pour réaliser l'analyse syntaxique des requêtes SQL, nous avons construit notre parseur en considérant la syntaxe de MySql disponible dans [60]. Pour le parseur de PHP nous avons modifié une grammaire à code source libre rendue disponible par Satyam [63]. La taille des grammaires utilisées est donnée par le Tableau 3.1. D'autres grammaires sont aussi disponibles dans l'archive de JavaCC [64].

Un algorithme utilisant le patron du visiteur traverse l'arbre syntaxique abstrait (AST visitor) pour ajouter l'instrumentation incluant les sondes ou les barricades selon les règles d'implantations utilisées dans l'environnement de JavaCC.

Les cas de tests légitimes ont été obtenues en utilisant l'application phpBB pendant

```
1  class sqlModelCheckerCl {
2
3    staticHashSetModel = newHashSet();
4
5    public static void init() {
6      // initialize model checker
7      // with all reference patterns
8      ...
9      model.add("search.php:221 SELECT ...");
10     ...
11   }
12
13   String parseQuery(String query) {
14     // SQL parser implementation
15   }
16
17   bool sqlModelGuard(String callSite, String query) {
18     if (model.contains(callSite + parseQuery(query))) {
19       mysql(query);
20     }
21     else {
22       System.exit(1);
23     }
24   }
25 }
```

Figure 3.5 Exemple conceptuel de barricades basées sur des patrons de références.

,

quelques heures en essayant de couvrir tous les cas d'utilisations pour tous les points d'appels à la base de données dans le code source. Nous avons obtenu 1 590 requêtes légitimes dont certaines étaient des répétitions de la même requête. Une fois les doublons éliminés 417 requêtes distinctes émanant de 107 énoncés vulnérables sont conservées.

Quelques requêtes légitimes partageant la même structure syntaxique mais avec des valeurs différentes pour certaines variables produisent des patrons de référence identiques. Ainsi les 417 requêtes légitimes distinctes produisent 155 patrons de référence distincts. La Figure 3.6 montre la distribution de la longueur des patrons de référence. La taille minimale d'un patron de référence est de 4 jetons, le maximum de 256 jetons et la moyenne de 35 jetons avec un écart-type de quatre jetons.

La Figure 3.7 montre la distribution des patrons de référence en fonction des différents

Tableau 3.1 Taille des analyseurs syntaxiques employés.

	PHP	SQL
Nombre de règles de grammaire	73	2 418
Taille de la grammaire (LOC)	1 221	34 133
Taille du parseur (LOC)	11 033	118 358

points d'appels. La plupart des points d'appels sont associés avec un seul patron de référence venant des requêtes légitimes utilisées pour cette expérimentation. Une moyenne de 1.4 patrons par point d'appel est observée avec un écart-type de moins de un patron. Le nombre maximal de patrons pour un point d'appel est de 5.

Nous avons identifié que neuf patrons de référence commencent par *SELECT OP_ASTERISK FROM SQL_ID ...* puis se poursuivent par des conditions ou des instructions d'ordonnancements. Même si ces requêtes sont reconnues comme légitimes, après la révision des patrons de référence produits, une certaine attention sur ces requêtes serait utile puisque l'injection de caractères spéciaux dans *SQL_ID* pourrait permettre facilement d'accéder à toute l'information dans la base de données.

Rendre phpBB sécuritaire en fonction des injection SQL consiste à remplacer les appels à *mySql*() par des appels à des barricades utilisant des patrons de référence pour chacun des points d'appels à *mySql*(). Comme pour le problème de l'instrumentation de phpBB avec des sondes pour recueillir les requêtes légitimes nous avons utilisé un algorithme basé sur le patron du visiteur de l'arbre syntaxique abstrait basé sur la grammaire PHP pour effectuer le remplacement des appels à *mySql*() par des appels aux barricades utilisant des patrons de référence. En tout 285 points d'appels vulnérables ont étés remplacés par des appels aux barricades appropriés.

Du point du vue d'un développeur de phpBB, les seuls changement au code source de phpBB a été le remplacement des appels à *mySql*() par des appels aux barricades utilisant les patrons de référence. Aussi 15 lignes de code supplémentaire en PHP sont incluses pour appeler un programme de 510 lignes en Java implémentant le parseur de SQL. Le modèle mental de l'application est très peu modifié par l'ajout automatique de la protection.

3.4.1 Injections SQL

Puisque l'intersection entre les cas de tests légitimes et les attaques est un ensemble vide (à moins d'erreurs dans la création des requêtes légitimes qui ont étés malencontreusement acceptées pendant la révision des tests). Pour s'assurer de la fiabilité de l'approche proposée,

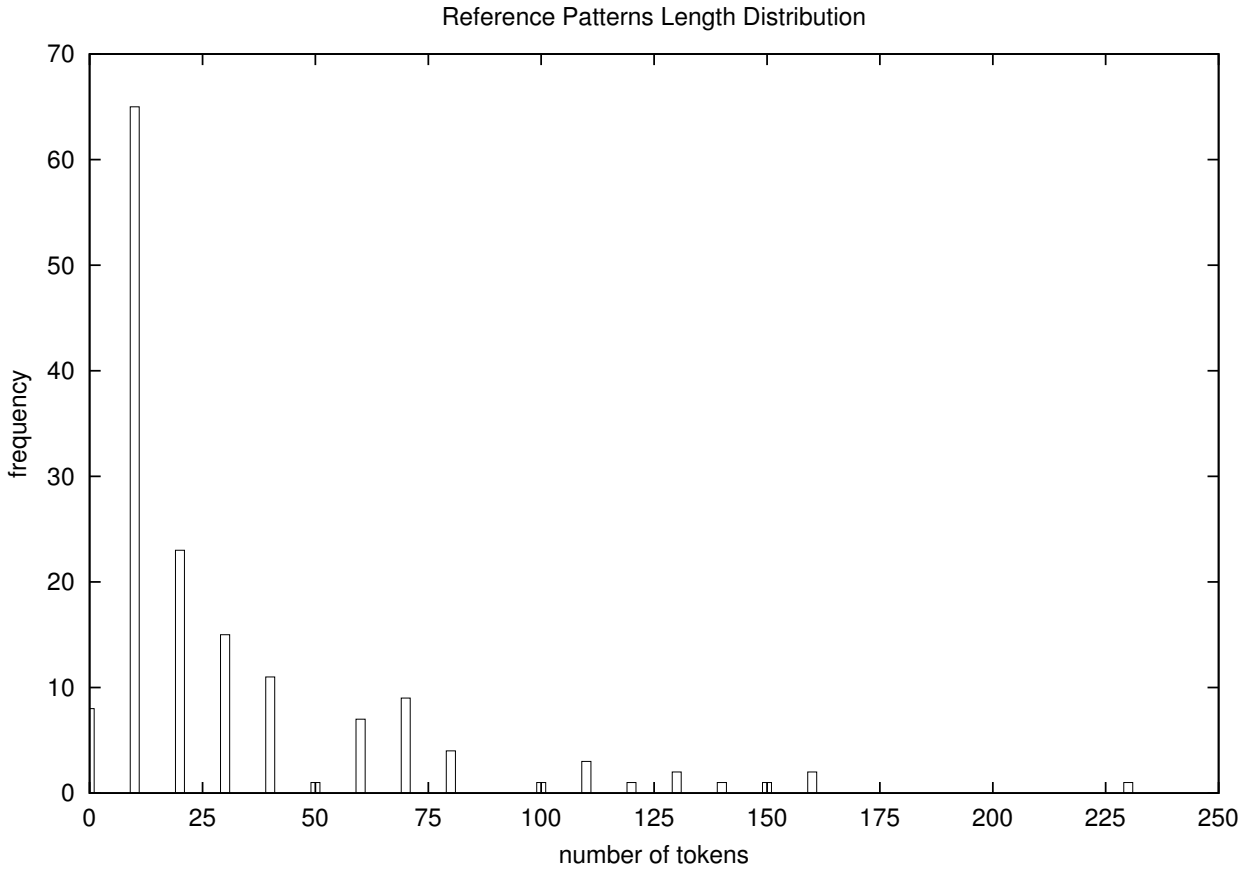

Figure 3.6 Distribution de la longueur des patrons de référence.

le système protégé automatiquement à été soumis à 176 attaques distinctes qui s'étaient révélés efficaces avant que le système ne soit protégé. Les attaques par injection SQL ont étés construites manuellement à partir d'informations disponibles publiquement sur le Web. L'exploitation des vulnérabilités est plus difficile et complexe à réaliser que de recueillir des requêtes légitimes ce qui explique pourquoi le nombre des attaques est beaucoup plus petit que le nombre de requêtes légitimes.

La raison d'être de ces expérimentations est d'évaluer le nombre de faux positifs et de faux négatifs obtenus en exécutant les cas de tests légitimes et les attaques avec la version protégée phpBB. Les cas de tests légitimes et les attaques ont étés séparés en deux classes : *legitimate cases* et *injection attacks*. L'exécution des *legitimate cases* produira des patrons de référence connus tandis que l'exécution des *injection attacks* devrait être bloquée dans la version protégée de phpBB.

La Figure 3.8 montre la distribution de la taille des 176 *injection attacks* qui ont étés uti-

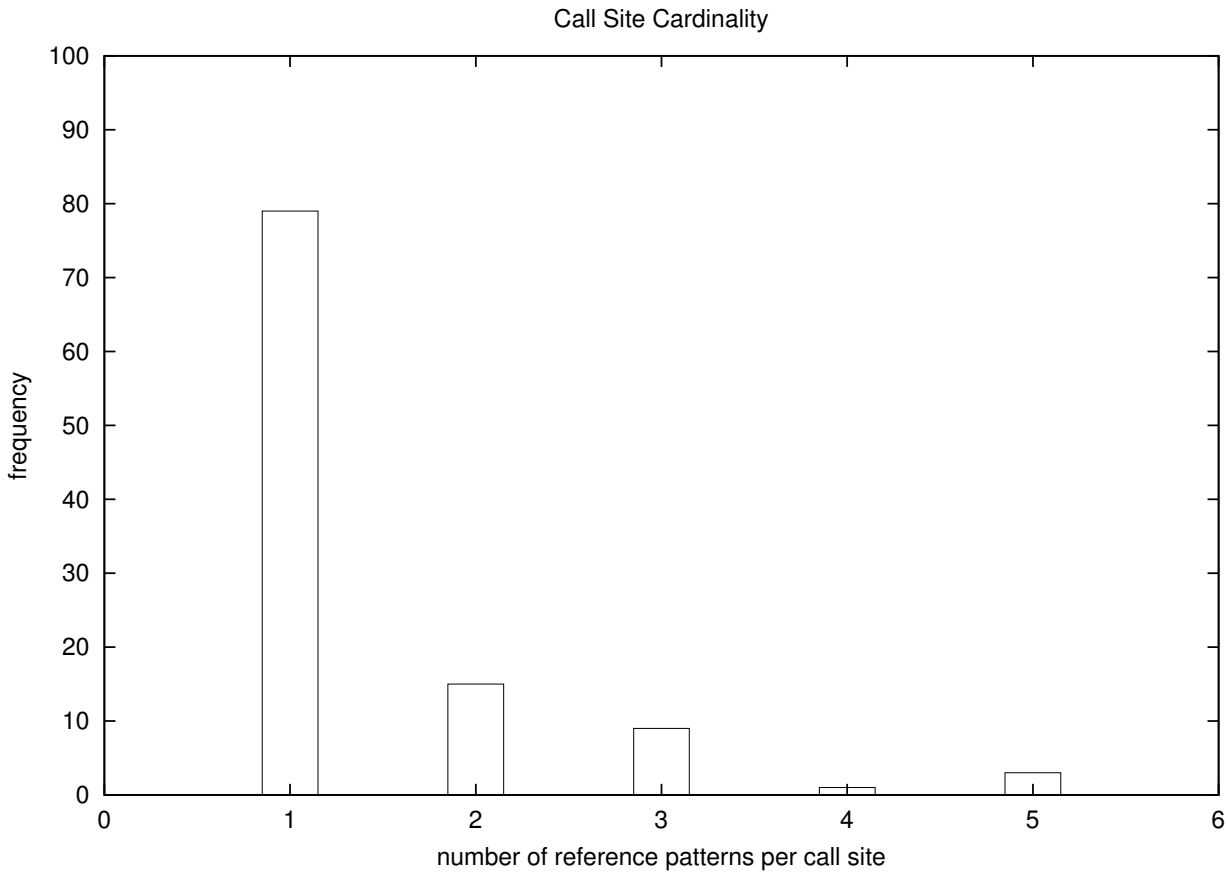

Figure 3.7 Distribution des patrons en fonction des points d'appels.

lisées pour notre expérimentation. Plus précisément, pour ces 176 attaques par injection (*injection attacks*), seulement trois points d'appels sont visés par ces attaques soit : *privmsg.php* à la ligne 236, *search.php* à la ligne 601, et *viewtopic.php* à la ligne 150. Ce nombre limité de point d'appels visés est dû au temps requis pour construire manuellement ces requêtes particulières.

Un script Perl a été utilisé pour permettre l'exécution des requêtes légitimes ainsi que des attaques par injection. Le script a reproduit les URL et les paramètres employés lors la construction des requêtes Web. Le tableau 3.2 présente les résultats obtenus lors de l'exécution des cas de tests légitimes et des attaques.

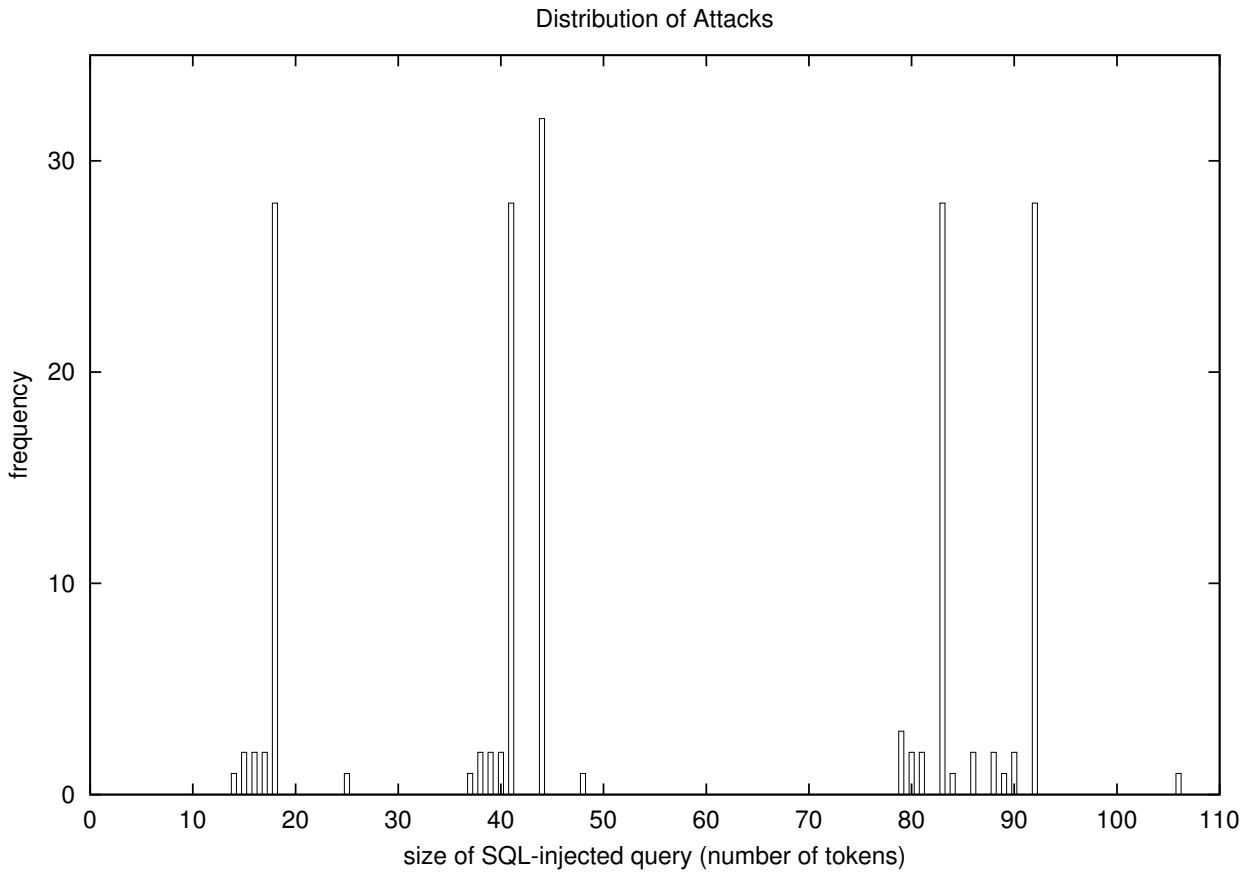

Figure 3.8 Taille des requêtes injectées.

3.5 Discussion

Les résultats expérimentaux confirment les prévisions d'un très haut taux de succès et d'un nombre peu élevé de faux positifs et de faux négatifs. En effet, aucun de ces deux derniers cas n'a été détecté dans l'expérimentation présentée. Intuitivement, ces résultats ne sont pas surprenants puisqu'à cause de la taille limitée du jeu de test expérimental, il est difficile d'imaginer une attaque qui possède la même structure syntaxique qu'une requête légitime incluant aussi les mêmes identificateurs légitimes utilisés dans l'application.

Néanmoins, des fausses alarmes et des erreurs de classification des attaques peuvent se produire si des requêtes SQL inappropriées étaient acceptées dans l'ensemble des requêtes légitimes. En général la précision de l'approche par protection automatique d'une application Web existante est dépendante de la représentativité des jeux de test légitimes et des scénarios d'utilisations légitimes. Les barricades de sécurité sont construites automatiquement à partir

Tableau 3.2 Résultats expérimentaux.

	Tests	Attaques
Nombre total	1 590	176
Acceptée	1 590	0
Acceptée (%)	100	0
Rejectée	0	176
Rejectée (%)	0	100

d'une approximation dynamique des spécifications de sécurité de l'application. D'une certaine manière cette approche peut souffrir du problème de l'exhaustivité de l'analyse dynamique mais nous croyons que l'impact en est limité parce qu'un nombre relativement peu élevé de patrons est présent pour chacun des points d'appels.

Notre approche est basée sur la supposition qu'il n'existe qu'une faible variété de structures syntaxiques pour les requêtes SQL produite pour chacun des points d'appels à la base de données puisque chacun des points d'appels représente un groupe de requêtes lié à une certaine composition légitime de variables.

Les systèmes existants n'ont pas, en général, été conçus en appliquant un modèle de sécurité particulier. Pour permettre l'adoption du modèle de sécurité proposé, une transition automatisée doit être utilisée pour convertir les systèmes existants en des systèmes plus sécuritaires. Comme il a déjà été mentionné, cette transition requière aussi qu'un transfert des responsabilités et de certaines tâches soit effectué. Des tâches auparavant réalisées par des développeurs seront maintenant sous la responsabilité de l'équipe sécurité. Cette approche aide à effectuer une transition vers un système plus sécuritaire à partir d'un système existant qui n'avait pas été conçu avec un modèle de sécurité en tête. Notre approche automatisée de la construction des barricades est simple et il y a plusieurs possibilités pour rendre ces barricades plus spécialisées en exploitant la puissance du parseur de requêtes qui est utilisé pour analyser les requêtes SQL soumises à l'application. L'efficacité de l'automatisation repose sur l'hypothèse d'un nombre fini et réduit de structures syntaxiques pour les requêtes SQL à un point d'appel de la base de données.

D'une autre manière, les développeurs de l'équipe sécurité peuvent construire manuellement des barricades de sécurité en se basant sur des spécifications de sécurité. Cette approche manuelle serait aussi possiblement réalisable avec une meilleure précision en terme d'exhaustivité pour les structures complexes mais cette tâche exigerait beaucoup d'efforts et pourrait produire des résultats questionnables sur des systèmes existants et anciens.

3.5.1 Exhaustivité

Trois catégories d'applications peuvent être identifiées : celles qui interagissent à l'aide des formulaires fixes, celles qui acceptent un sous-ensemble du langage SQL en se basant sur une forme de grammaire d'interaction et celles qui acceptent directement des requêtes SQL arbitraires.

La première catégorie d'applications contient les applications qui interagissent avec l'utilisateur au moyen de formulaires fixes. Cette catégorie d'application devrait être bien protégée par notre approche en autant que les points d'appels à la base de données coïncident avec la sémantique distincte des formulaires tel que la structure syntaxique peut être validée à un point d'appel de la base de données. Cette approche ne fonctionnerait pas si toutes les requêtes étaient redirigées vers un même point d'appel à la base de données. Plus de recherche serait nécessaire pour démêler la construction de la requête lorsqu'un seul appel à la base de données est utilisé pour transmettre toutes les requêtes de l'application.

Les stratégies pour la protection des applications sont souvent déficientes lorsque de nouvelles attaques sont conçues. De nouvelles variations syntaxiques malicieuses ne seraient pas reconnues par notre approche car elles n'apparaîtraient pas dans l'ensemble de requêtes légitimes et ces nouvelles variations seraient rejetées. Notre approche est robuste essentiellement parce qu'elle reconnaît toutes les requêtes explicitement acceptables qui seront transmises à la base de données. N'importe quelles variations à la structure syntaxique d'une requête qui n'apparaîtraient pas dans l'ensemble des requêtes légitimes seront rejetées dans tous les cas. Malheureusement le coût de cette robustesse est la présence de faux positifs mais ce coût devient négligeable lorsque l'application est basée sur des formulaires fixes.

Les applications de la seconde catégorie peuvent accepter un nombre variable de formulaires ou d'interactions et ces contraintes doivent s'inscrire à l'intérieur de limites imposées par une grammaire de l'application. On peut imaginer un formulaire de recherche avancée pour un moteur de recherche, l'utilisateur peut spécifier un nombre arbitraire de combinaisons impliquant des identificateurs et des opérateurs. Dans ces cas, le modèle de sécurité devra implanter la grammaire sous-jacente et s'assurer que les requêtes générées ne s'écartent pas des structures syntaxiques permises. L'approche de protection présentée dans ce chapitre possède une utilité plus limitée pour protéger cette catégorie d'application.

Les applications de la troisième catégorie permettent l'utilisation sans contraintes de toute la syntaxe de SQL. Les points d'appels à la base de données qui reçoivent ces requêtes complexes et qui utilisent potentiellement toutes les possibilités syntaxiques de SQL sous forme de requêtes textuelles, devraient être réservé à l'usage d'utilisateurs privilégiés ou des administrateurs du système. Des exemples peuvent être trouvés dans la partie dédiée aux administrateurs du système d'une application. Dans ces cas, la syntaxe seule n'est pas

suffisante, par définition, pour filtrer les requêtes et d'autres mécanismes d'autorisation et d'authentification devraient être utilisés.

Ces types d'accès à la base de donnée ne peuvent pas être protégés par l'approche proposée mais en général ces points d'appels devraient seulement être accessibles à partir de sections de l'application qui sont réservés aux administrateurs et ne devraient pas être accessibles aux usagers qui ne sont pas authentifiés. Ainsi ces points d'appels à la base de donnée ne devaient pas être vulnérables à des attaques par injection sql venant d'usagers non autorisés. Une analyse des vulnérabilités qui peut détecter ce type de divergence entre la sécurité requise et le niveau d'autorisation dans le contexte des attaques de type injection SQL est décrite dans [6] et dans le chapitre 4.

Plus d'expérimentation est requis pour augmenter le nombre d'attaques et le nombre de systèmes analysés pour évaluer statistiquement la précision et la couverture d'une manière plus représentative.

L'analyse dynamique utilisée dans notre approche peut être limitée par le manque d'exhaustivité. L'ensemble des cas légitimes n'est probablement pas complet, spécialement lorsque le nombre de patrons syntaxiques pour un point d'appel est élevé. Certaines requêtes légitimes pourraient être rejetées parce qu'elles n'ont pas été incluses dans les cas légitimes et qu'elles n'ont pas contribués à la construction de barricades. Ceci est possible parce que nous tentons de déduire les intentions du concepteur du système à partir d'un ensemble limité de cas d'utilisations légitimes.

Plusieurs techniques peuvent réduire l'impact du manque d'exhaustivité des patrons de références. La plus simple est la révision humaine des patrons de références surtout si elle peut être faite en coordination avec la révision des interactions spécifiées. Cela peut être efficace dans le cas des formulaires fixes et d'un nombre de patrons peu élevé.

Puisque le nombre de patrons par point d'appel à la base de données est d'environ 1.4 (voir la section 3.4 pour plus de détails), il possible d'inspecter l'ensemble des patrons de référence pour l'expérience présentée. Plus de recherche est cependant nécessaire pour évaluer le nombre de patrons par point d'appel dans un éventail plus diversifié d'applications.

En outre, la revue humaine des points d'appels à la base de données peut aider à identifier les points qui seraient mieux protégés par d'autres mécanismes que ceux proposés dans ce chapitre. C'est à dire les points d'appels complexes difficiles à protéger peuvent être migrés vers une grammaire d'interaction ou bien ils pourraient être considérés comme des requêtes SQL arbitraires et protégés par une procédure d'authentification et d'autorisation.

3.5.2 Performance

Une autre question méritant d'être considérée est la pénalité de performance encourue par l'utilisation des barricades basées sur des patrons de référence dans une application.

L'instrumentation d'une application pour recueillir le profilage des requêtes SQL et pour l'analyse dynamique est dépendante selon la taille de l'application à instrumenter. La construction des barricades est proportionnelle au nombre de cas de tests et à la taille des requêtes SQL recueillis pendant l'exécution des cas de tests. Le remplacement des appels à la base de données $mySql()$ par des appels aux barricades dans une application est encore une fois linéaire en fonction de la taille de l'application.

Durant l'exécution de l'application, l'analyse syntaxique d'une requête SQL par une barricade prend un temps proportionnel à la longueur de la requête analysée. La validation d'une requête en fonction des patrons de référence prend à nouveau un temps proportionnel à la longueur de la requête à cause de l'implantation par table de hachage de cette validation. Plus de détails à propos de la longueur des requêtes et des patrons est disponible dans la section 3.4.

Pour le moment, le prototype d'implantation est ralenti par le chargement de la machine virtuelle Java à toute les fois qu'une barricade demande l'analyse d'une requête SQL en utilisant le parseur en Java. Une meilleure implantation est nécessaire et un peu de travail supplémentaire est requis pour identifier une meilleure architecture qui éviterait le chargement à répétition de la machine virtuelle Java. Il serait possible de concevoir un parseur en utilisant le langage PHP ou bien de conserver la machine virtuelle Java chargée en mémoire et d'utiliser une méthode de communication interprocessus durant l'exécution de PHP. D'un autre point de vue, l'approche actuelle même si elle est moins efficace est très portable. Le parseur peut être utilisé facilement avec d'autres langages que PHP. L'adaptation de notre approche à d'autres langages demanderait simplement de réécrire les algorithmes pour instrumenter l'application pour le profilage dynamique ainsi que la réingénierie automatique du code source. Le parseur SQL n'a pas besoin d'être changé. D'un point de vue de la syntaxe du code source, le traitement d'une application en Java pourrait être plus facile qu'une application en PHP à cause des meilleures spécifications de la syntaxe et de la sémantique du langage Java.

Une approche originale qui combine l'analyse statique, l'analyse dynamique et la réingénierie du code source pour protéger automatiquement une application écrite en PHP contre les attaques de type injection SQL a été présentée, implémentée et évaluée. L'application phpBB a été automatiquement protégée en utilisant cette approche. 176 attaques par injection sql, qui ont réussit avant que l'application soit soumise à l'approche de protection proposée, ont aussi été soumises à la version automatiquement protégée de l'application. Les

résultats expérimentaux ont démontrés un très haut taux de succès et un très faible taux de faux négatif (en effet aucun n'a été détecté dans l'expérimentation présentée).

CHAPITRE 4

DÉTECTION DE VULNÉRABILITÉS SQL-INJECTION SENSIBLES AUX ATTAQUANTS INTERNES ET EXTERNES

En général les attaques de type SQL-injection reposent sur des entrées de données textuelles qui sont insuffisamment validées et qui sont ensuite utilisées pour construire des requêtes à une base de données. Des données en entrée pouvant être construites d'une manière malicieuse pourront compromettre la confidentialité et la sécurité des sites Web qui utilisent une base de données pour accéder, emmagasiner ou manipuler des informations. De plus, des attaquants internes pourraient introduire des fragments de codes malicieux à l'intérieur d'une application Web. Ces codes malicieux pourraient se déclencher lors l'entrée de données spécifiques et pourraient par exemple outrepasser la politique de sécurité du site Web. Ce chapitre présente une approche originale basée sur l'analyse statique pour détecter automatiquement les instructions en PHP qui peuvent être vulnérables à des attaques de type SQL-injection déclenchée soit par des données malicieuses (attaque externe) ou par une portion de code malicieux (attaque interne). De nouvelles équations de flux sont présentées, elles propagent et combinent des niveaux de sécurités en suivant les arrêtes d'un graphe de flux de contrôle inter-procédural. Le calcul des niveaux de sécurité présente un temps d'exécution et une taille mémoire linéaire.

4.1 Introduction

Des applications Web sont utilisées pour distribuer de l'information venant d'une organisation vers plusieurs usagers en utilisant un réseau informatique. La plupart du temps, ces applications interagissent avec des usagers et effectuent des requêtes à une base de données (BD). Si, comme c'est souvent le cas, ces requêtes supposent que les données venant des usagers sont légitimes et que le code utilisé pour construire les requêtes est lui aussi légitime, les applications seront alors probablement vulnérables à des attaques de type SQL-injection. Ces attaques reposent sur des faiblesses dans la validation des données textuelles qui sont utilisées pour construire les requêtes SQL [58].

Dans certains contextes spécifiques, des données en entrée malicieuses contenant des instructions SQL ou des fragments d'instructions SQL vont produire des requêtes dont la sé-

1. Une version en anglais de ce chapitre a été publiée sous le titre « Insider and Ousider Threat-Sensitive SQL Injection Vulnerability Analysis in PHP [6] » lors de la conférence *13th Working Conference on Reverse Engineering* (WCRE).

mantique est différente de celle voulue par les concepteurs de l'application Web. Une requête contenant des données malicieuses pourra violer les politiques de sécurité de l'application. De plus, sans égards à la validation des données en entrée, un attaquant interne pourra introduire dans l'application Web une portion de code qui se déclenchera lors de l'entrée de données spécifiques. Par exemple, un attaquant interne pourra introduire une portion de code qui violera intentionnellement la politique de sécurité et de confidentialité.

Dans [65], les attaques de type SQL-injection et *cross site scripting* (XSS) sont décrites comme deux des plus importants problèmes dans les applications Web. Les attaques internes et les références à des codes malicieux sont abordées dans [66, 67, 68, 69, 70] par contre il y a peu de travaux publiés a propos de l'analyse du code source pour détecter des attaques de type SQL-injection internes.

Dans un travail précédent [1], nous avons développé une approche de prévention des SQL-injection basée sur la construction automatique de barricades de sécurité (security wrappers). Les barricades font une vérification syntaxique en comparant une requête SQL avec des patrons de requêtes SQL au moment de l'exécution à chacun des points d'appels à la BD. Les barricades peuvent être construites automatiquement à partir d'une analyse dynamique d'une application Web après qu'elle ait été instrumentée.

Dans ce chapitre, nous abordons le problème de l'identification des énoncés vulnérables à des attaques de type SQL-injection causées par des données externes ou par un code malicieux interne. La perspective de l'analyse de flux statique est utilisée pour effectuer une analyse des niveaux d'autorisations et de la vulnérabilité.

Ce chapitre présente des équations d'analyse de flux originales qui propagent des niveaux de sécurité le long des arêtes d'un graphe de flux de contrôle inter-procédural (GFC). Les nœuds du GFC combinent les différents niveaux de sécurité provenant des arêtes. Les principales contributions de ce chapitre sont :

- La prise en compte des attaques internes et externes lors de la détection des vulnérabilités de type SQL-injection.
- Un algorithme de type point fixe original basé sur des transitions des niveaux d'autorisation.
- Une analyse statique détectant automatiquement les énoncés vulnérables en tenant compte des différences entre les niveaux d'autorisation effectifs et la politique de sécurité d'accès à la base de données.
- Une vérification expérimentale sur l'application phpBB en tant qu'étude de cas.

La section 4.2 présente les équations de flux inter-procédural et l'analyse par point fixe, la section 4.3 décrit l'expérimentation réalisée et ses résultats, la section 4.4 présente une

discussion à propos de l'analyse et des résultats obtenus, la section 2.3 décrit certains travaux de recherche sur des sujets proches et la section 4.5 conclut ce chapitre.

4.2 Analyse inter-procédurale

4.2.1 Définition du problème

Les énoncés vulnérables aux SQL-injection peuvent être définis comme des énoncés accédant à une base de données en requérant un niveaux de sécurité $sLev_i$ pour leur exécution même si ces énoncés peuvent être rejoints avec un niveau d'autorisation effectif $aLev_j$ qui est plus petit que le niveau de sécurité requis $sLev_i$. (Voir [19, 20, 22, 24] pour une discussion sur les treillis de sécurité requise et effective.)

Puisque l'analyse présentée est indépendante de la base de données qui est utilisée, nous utiliserons le SGBD MySql [60] qui implémente un sous-ensemble étendu du langage SQL. La communication entre l'application Web et la base de données s'effectue à l'aide d'appels à des routines d'une interface de programmation applicative (API). Un exemple typique $mysql(str, db)$ est un appel à la base de données MySql où *str* est une chaîne de caractères qui contient la requête SQL à exécuter et *db* est une référence à la base de données.

L'analyse présentée est définie en deux étapes. La première utilise une analyse de flux statique inter-procédurale pour déterminer les niveaux d'autorisations qui sont propagés dans le code source PHP. La deuxième étape compare les niveaux de sécurité requis pour chacun des accès à la base de données avec ceux qui ont étés calculés par l'analyse statique. Les divergences entre les deux valeurs pour un même nœud du graphe de flux de contrôle sont signalées en tant qu'alertes de vulnérabilités à une SQL-injection.

4.2.2 Information de flux et ordre partiel

Les techniques d'analyses de flux ont été historiquement importantes pour la cueillette d'informations à propos de certaines propriétés des programmes informatiques [71, 72, 73]. Les analyses de flux se sont aussi prouvées utiles pour les tâches de maintenance des logiciels, de la compréhension des systèmes et pour le tranchage de programmes (slicing) [74, 75, 76, 77, 78].

Les patrons de concessions d'autorisations sont définis comme un sous-ensemble particulier de nœuds et de transitions dans le graphe de flux de contrôle. Certaines transitions spéciales appartiennent à l'ensemble des transitions concédant la sécurité (AGTS, Autorisation Granting Transition Set). Ces transitions spéciales sont définies à partir d'un nœud identifié par un patron jusqu'aux successeurs du nœud identifié. Ces transitions spéciales vont concéder un niveau de sécurité spécifique aux opérations qui vont les suivre.

L'équation 4.1 montre qu'une transition $(v,\ w)$ propage un entier appelé niveau d'autorisation concédé $GA(v,\ w,\ aLev)$ (Granted Authorization) qui est égal au facteur d'autorisation $ga(v,\ w)$ si (v, w) est une transition spéciale concédant un niveau de sécurité sinon, la transition (v, w) propagera le niveau de sécurité $aLev$ qu'elle aura reçu en entrée, à ses successeurs. Nous supposons que les niveaux de sécurité sont ordonnés c'est à dire que les privilèges concédés au niveau $aLev_i$ sont moindres que ceux concédés au niveau $aLev_j$ lorsque $(i < j)$.

$$GA(v,\ w,\ aLev) = \begin{cases} ga(v,\ w) & if\ (v,\ w) \in AGTS \\ \\ aLev & otherwise \end{cases} \tag{4.1}$$

Les patrons de concessions d'autorisations sont spécifiques à chaque application Web. La Figure 4.1 à la ligne 7 montre un exemple typique d'un patron concédant la sécurité dans l'application Web phpBB. Le script « pagestart.php » quitte subrepticement (*die*) si le niveau de sécurité de l'exécution courante est différent du niveau système. La transition (7,8) de la ligne 7 à la ligne 8 garantie que l'exécution continue au niveau système.

```
1  //
2  // Load default header
3  //
4  $no_page_header = TRUE;
5  $phpbb_root_path = "../";
6  require($phpbb_root_path . 'extension.inc');
7  require('pagestart.' . $phpEx);
8
9  //
10 // Do the job ...
11 //
```

Figure 4.1 Patron concédant la sécurité

Comme convention dans ce chapitre, un niveau d'autorisation égal à zéro correspond un niveau de privilège de niveau usager c'est-à-dire aucun privilèges pour effectuer des opérations réservées. Les niveaux de sécurité plus élevés représentent des niveaux croissant de privilèges jusqu'au niveau administrateur du système.

L'information de flux est défini pour chaque nœud $v \in V$ dans le graphe de flux de contrôle comme un m-tuple $\langle aLev_0, aLev_1, \ldots, aLev_{m-1} \rangle$ où $aLev_i \in N$ est le niveau d'autorisation du nœud v lorsqu'il est appelé dans le contexte i. Les contextes inter-procéduraux correspondent

en principe aux différents enchaînements d'appels de fonctions possibles qui sont obtenues à partir du point d'entrée principal (main) du graphe de flux de contrôle jusqu'à la fonction contenant le noeud courant. Dans cette analyse des « niveaux d'autorisation » les chaînes d'appels distinctes qui propagent le même niveau de sécurité à la fonction appelée en cours produiront des informations de flux qui seront indistinguables des autres chaînes d'appels distinctes qui propagent la même information de flux. En conséquence pour cette analyse des « niveaux d'autorisation » le nombre maximal m de contextes correspond au nombre distinct de niveaux de sécurité considérés.

Le nombre de niveaux de sécurités est souvent beaucoup plus petit que le nombre de chaînes d'appels possibles. C'est un avantage réel pour l'analyse proposée en terme d'économie du temps de calcul puisqu'il n'est pas nécessaire de recalculer les contextes qui produisent des informations indistinguables. Pour les analyses et les expériences présentées, deux contextes donc $m = 2$ pour chacun des nœuds est suffisant pour représenter l'information de flux correspondant aux niveaux de sécurité usager et système.

$IN(v, c)$ et $OUT(v, c)$ représentent respectivement les niveaux d'autorisations avant et après l'exécution d'un nœud v dans le contexte inter-procédural c. Même si les équations de flux sont calculées à partir des transitions concédant la sécurité, il est nécessaire de conserver les valeurs de $IN(v, c)$ et $OUT(v, c)$ pour chacun des nœuds du graphe parce que les nœuds de type *call* peuvent traverser d'une manière inter-procédural plusieurs transitions alors il est nécessaire de calculer séparément $IN(v, c)$ et $OUT(v, c)$ pour chacun des nœuds.

Plusieurs chemins propageant différents niveaux de sécurité sont combinés en utilisant l'opérateur de confluence. Dans le cas de cette analyse, l'opérateur de confluence sélectionne le niveau de sécurité minimale parmi les niveaux présents.

Afin que l'algorithme par point-fixe puisse converger et parce que tous les nœuds qui sont statiquement rejoignables doivent être calculés au moins une fois, la valeur initiale de l'information de flux pour $IN(v, c)$ et $OUT(v, c)$ est $\top = (m + 1)$ pour tous les nœuds dans V et pour tous les contextes. Sauf pour les nœuds de départ des différents graphes de flux de contrôle inter-procéduraux dont la valeur initiale de l'information de flux sera $\perp = 0$.

4.2.3 Algorithme point-fixe

L'algorithme par point-fixe itère sur les nœuds du graphe de flux de contrôle jusqu'à ce qu'un point stable (fixe) apparaisse dans les équations de flux correspondantes. Une description plus détaillée des aspects inter-procéduraux de la construction du graphe de flux de contrôle est disponible dans [23].

Une différence proposée par l'analyse de flux des « niveaux d'autorisation » est que les niveaux d'autorisation sont déterminés à partir des transitions plutôt qu'à partir des nœuds

```
   fixed_point(CFG) {
     // initialization
1    ∀v ∈ V {
2      ∀c ∈ [ 0 .. (max_contexts − 1) ] {
3        if((type(v) = system_start) ∧ (c = 0)) {
4          IN(v, c) = 0
5          OUT(v, c) = AL_Max + 1
6          stack.push(v, c)
         } else {
7          OUT(v, c) = IN(v, c) = AL_Max + 1
         }
       }
     }
     // fixpoint
8    while (not (stack.empty())) {
9      (v, c) ← stack.pop()
10     changes = eval_flow(v, c)
11     if (changes)
12       propagate_flow(v, c)
       }
     }
   }
```

Figure 4.2 Algorithme par point-fixe pour calculer les niveaux d'autorisations.

comme c'est normalement l'usage [73]. Ceci permet d'aider à la compréhension d l'algorithme. Dans le cas de l'analyse des niveaux d'autorisations, le focus sur les transitions est plus intuitif puisqu'il y a beaucoup moins de transitions concédant la sécurité qu'il y a de nœuds dans le graphe de flux de contrôle. Tout de même, un algorithme par point fixe équivalent pourrait être construit même si moins intuitif.

La Figure 4.2 montre l'algorithme par point-fixe utilisé. L'algorithme traite en entrée le CFG et produit en sortie le même CFG annoté avec l'information de flux des niveaux de sécurité. Les fonctions *eval_flow* et *propagate_flow* sont décrites dans les figures 4.3 et 4.4.

Une approche par « work-list » a été utilisée pour le calcul algorithmique du point-fixe. Cette approche propage l'information de flux seulement si le critère de convergence n'a pas encore été atteint. C'est à dire, seulement si l'information entrante modifie le niveau de sécurité du nœud en cours de traitement.

Les équations de flux de la Figure 4.3 et les équations de la propagation de l'information de flux de la Figure 4.4 représentent une adaptation de l'analyse de propagation des risques présentée dans [23] qui conserve le même traitement inter-procédural mais qui remplace la

```
   bool eval_flow(v, c) {
1    changes = false
2    switch (type(v)) {
3      case call :
         // compute IN of called functions
4        ∀ w ∈ called_entry_points(v) {
6          mInfo = min(GA(v, w, IN(v, c)),
                       IN(w, IN(v, c)))
7          if (mInfo < IN(w, IN(v, c)))
8            IN(w, IN(v, c)) = mInfo
9            stack.push(w, IN(v, c))
10         } else {
             // compute OUT of current call node
             // from called functions exit nodes
11           ∀ w ∈ called_exit_points(v) {
12             mInfo = min(GA(w, v, IN(v, c)),
                           OUT(v, c))
14             if (mInfo < OUT(v, c)) {
15               OUT(v, c) = mInfo
16               changes = true
                 }
               }
             }
17       break
18     default :
         // compute OUT from IN
19       if (IN(v, c) < OUT(v, c))
20         OUT(v, c) = IN(v, c)
21         changes = true
     }
22   return(changes)
   }
```

Figure 4.3 Calcul de l'information de flux basée sur les transitions

perspective de la propagation des risques par l'analyse des niveaux de sécurité. Plus de détails sur l'analyse statique de propagation des risques est disponible dans [23].

4.2.4 Analyse des vulnérabilités

Pour les besoins de cette expérimentation, nous avons défini le niveau de sécurité requis pour accéder à la base de données via des requêtes en mode texte sans aucune restriction comme le niveau de sécurité maximal (niveau système) tandis que les accès protégés par une

```
   propagate_flow(v, c) {
1    switch (type(v)) {
2      case exit :
         // back propagate OUT from exit node
         // to OUT ofcalling nodes
3        ∀ w ∈ calling_points(v) {
4          ∀calling_c ∈ [ 0 .. (max_contexts − 1) ] {
5            if (IN(w, calling_c) = c) {
7              mInfo = min(GA(v, w,OUT(v, c)),
                                 OUT(w, calling_c))
8              if (mInfo < OUT(w, calling_c)) {
9                OUT(w, calling_c) = mInfo
10               propagate_flow(w, calling_c)
                 }
               }
             }
           }
11       break
12     default :
         // propagate OUT from current node
         // to IN of successors
13       ∀ w | (v, w) ∈ E {
15         mInfo = min(GA(v, w, OUT(v, c)),
                           IN(w, c))
16         if (mInfo < IN(w, c)) {
17           IN(w, c) = mInfo
18           stack.push(w, c)
           }
       }
     }
```

Figure 4.4 Algorithme de la propagation de l'information de flux basée sur les transitions.

barricade possèdent un niveau de sécurité minimal de *zéro* et peuvent être exécutés au niveau de sécurité usager.

L'analyse des vulnérabilités devient simplement la recherche des accès à la base de données au niveau de sécurité système qui peuvent être rejoint par une exécution au niveau usager.

4.3 Expérimentation

Pour valider notre approche nous avons pris la perspective de l'étude de cas visant à détecter les vulnérabilités SQL-injection pouvant être causé par des attaques internes ou externes. phpBB [79] est une application Web bien connue pour ses possibilités d'attaques

de type SQL-injection qui sont possibles dans ses versions antérieures. En effet, au cours des dernières années un effort particulier a été déployé pour réduire les possibilités d'attaques de type SQL-injection dans phpBB.

Pour l'expérience décrite ici, nous avons choisi la version ancienne 2.0.0 de phpBB pour laquelle plusieurs vulnérabilités ont étés rapportés publiquement. La taille et le nombre des fichiers de code source composant cette version de phpBB est disponible dans le Tableau 4.2. Notre objectif est d'utiliser une version ancienne comportant des failles de sécurité et de détecter automatiquement des vulnérabilités de type SQL-injection qui sont causées par des données entrantes ou possiblement par du code malicieux injecté par un attaquant interne.

L'analyse syntaxique a été effectuée à l'aide d'une grammaire PHP en JavaCC [64] disponible sur le Web produite par Satyam [63] et qui a été légèrement étendue pour pouvoir analyser tout le code source de phpBB correctement. Le Tableau 4.1 montre quelques caractéristiques de l'analyseur syntaxique construit et utilisé pour cette expérimentation.

Tableau 4.1 Taille de l'analyseur syntaxique.

Nombre de règles	73
Taille de la grammaire (LOC)	1 221
Taille du parseur (LOC)	11 033

Le Tableau 4.2 présente un cumulatif des caractéristiques du graphe de flux de contrôle inter-procédural ventilé par chacun des sous-répertoires contenant le code source de phpBB. La valeur LOC dans le tableau 4.2 inclut également les lignes de code source PHP et les lignes contenant du code HTML. Le graphe de flux de contrôle inter-procédural a été produit avec le format GXL (Graph eXchange Language) qui est un format standard XML pour la représentation des graphes [80].

L'expérimentation a été faite sur un ordinateur comportant un processeur Pentium 4 cadencé à 3 GHz avec 1GB de mémoire vive sous le système d'exploitation Linux Fedora Core 5.

Le Tableau 4.3 indique les résultats de l'analyse des « niveaux de sécurité » ventilé pour chacun des sous-répertoires du code source de phpBB. Pour chacun des sous-répertoire, le nombre de nœuds total est indiqué. Aussi pour chacun des deux niveaux de contextes « usager » ou « système » , le nombre et le pourcentage de nœuds au niveaux d'autorisation « usager » ou « système » sont indiqués.

Comme indiqué dans le Tableau 4.3 , le sous-répertoire principal '/' , ainsi que les sous-répertoires '/contrib' , '/db' , '/includes' et '/language' contiennent des routines qui lors-

Tableau 4.2 Taille du CFG de phpBB.

chemin	fichiers	LOC	noeuds	arcs
/	16	11 919	8 204	10 202
/admin	18	7 884	5 265	6 519
/contrib	2	1 057	695	1 212
/db	8	3 149	2 913	3 550
/includes	25	8 213	5 963	7 897
/language	4	1 879	1 316	1 312
Total	73	34 101	24 356	30 692

qu'elles sont appelés au niveau usager ne vont jamais augmenter leurs niveaux d'autorisations au niveau système.

Tableau 4.3 Authorisations.

Chemin	Total noeuds	contexte usagé		contexte système
		niveau usagé noeuds (%)	niveau système noeuds (%)	niveau système noeuds (%)
/	8 204	8 127(99)	0(0)	0(0)
/admin	5 265	930(17)	3 298(62)	141(2)
/contrib	695	292(42)	0(0)	257(36)
/db	8	287(9)	0(0)	192(6)
/includes	5 963	5 193(87)	0(0)	1 176(19)
/language	1 316	1 316(100)	0(0)	0(0)
Total	**24 356**	**16 145(66)**	**3298(13)**	**1766(7)**

Les routines dans les sous-répertoires '/' , et '/language' ne sont jamais appelées au niveau système tandis que les routines dans les sous-répertoires '/admin' , '/contrib' , '/db' et '/includes' peuvent être appelées au niveau système. Néanmoins, les seules routines qui peuvent augmenter le niveau d'autorisation de « usager » à « système » sont dans le sous-répertoire '/admin' comme on pourrait le supposer.

Dans le Tableau 4.3, une seule colonne est utilisée pour le contexte "system context" « niveau système » parce que dans cette version de phpBB les routines appelées au niveau système conservent leur niveau d'autorisation jusqu'à la fin de l'exécution.

Les différences entre les niveaux d'autorisations effectifs et les niveaux de sécurité requis

pour accéder à la base de données déterminent les accès à la base de données qui peuvent être vulnérables aux SQL-injection. Pour chaque fichier source de phpBB le nombre d'accès à la base de données a été calculé ainsi que le nombre d'accès à la base de données qui sont vulnérables.

Le Tableau 4.4 détaille les vulnérabilités en cumulant les valeurs pour tous les fichiers dans un même sous-répertoire. Les douze fichiers dans les sous-répertoire '/db' et '/language' ne font aucun appel à la base de données et trivialement peuvent reconnus être comme n'étant pas vulnérables aux SQL-injections.

Tableau 4.4 Résultats de l'analyse des vulnérabilités.

Chemin	Total accès à la BD	Total accès vulnérable à la BD (%)
/	166	163(98)
/admin	131	15(11)
/contrib	3	3(100)
/db	0	0(0)
/includes	106	104(98)
/language	0	0(0)
Total	**406**	**285(70)**

En observant le Tableau 4.4, on peut constater que parmi les vulnérabilités, quinze proviennent en particulier du sous-répertoire '/admin'. La Figure 4.5 montre une vulnérabilité qui a été identifiée par notre analyse dans le fichier de niveau administrateur admin_styles.php . Cette vulnérabilité est connu comme « File Include Vulnerability, Bugtraq id : 7932 [81, 82] » de la classe « Input Validation Error ». Le patron concédant la sécurité à la ligne 5 de la Figure 4.5 est conditionnel à l'exécution du test à la ligne 1. Selon la discussion dans [82] : « . . . This could reportedly allow an attacker to include a malicious or sensitive local file. Information disclosure or execution of arbitrary commands could be the result. »

La Figure 4.6 montre la distribution des vulnérabilités dans les fichiers qui accèdent à la base de données. Le nombre de vulnérabilités se situe entre aucun dans les fichiers qui sont correctement protégés jusqu'à un maximum de trente trois pour les vulnérabilités dans le fichier privmsg.php dans le sous-répertoire /. La figure indique la distribution des fichier qui contiennent peu ou beaucoup de vulnérabitilés, par exemple il y 15 fichiers contiennent 0 vulnérabilités et il y a 1 seul fichier qui contient 34 vulnérabilités.

```
1 if( empty($HTTP_POST_VARS['send_file']) )
2 {
3    $no_page_header = ( $cancel ) ? TRUE : FALSE;
4    require($phpbb_root_path . 'extension.inc');
5    require('pagestart.' . $phpEx);
6 }
```

Figure 4.5 Exemple d'une vulnérabilité *admin*.

Le Tableau 4.5 montre les performances de l'analyse présentée lorsqu'exécutée sur l'application Web phpBB. Le tableau indique la taille du graphe de flux de contrôle, le nombre de nœuds qui sont rejoignables, le nombre total de nœuds traité par l'algorithme par point-fixe, le ratio entre les nœuds rejoignables et les nœuds traités par le point-fixe, le temps processeur requis pour lire le graphe de flux de contrôle en format GXL, le temps requis pour l'initialisation des valeurs de contextes utilisé par le point-fixe ainsi que le temps nécessaire pour exécuter le point-fixe. Il est notable que le ratio des nœuds traité par le point-fixe est très faible 1.07 et que le temps requis pour la convergence du point-fixe est court lui aussi 0.077 s. La majorité du temps est passée à initialiser tous les contextes inter-procéduraux et à lire le fichier GXL qui contient le graphe de flux de contrôle.

Tableau 4.5 Performance.

Taille du CFG en noeuds	24 356
Nombre de noeuds accessibles	19 647
Noeuds traités par le point fixe	21 209
Ratio des noeuds traités	1.079
Temps requis pour la lecture du CFG	4.361 s
Temps d'initialisation	0.303 s
Temps d'exécution du point fixe	0.077 s
Temps d'exécution total	4.741 s

4.4 Discussion

L'analyse statique des vulnérabilités présentée dans ce chapitre est sensible au flux et aux contextes. L'analyse possède des caractéristiques lui permettant d'avoir des performances spécifiques et avantageuses. Puisque les contextes inter-procéduraux sont fusionnés en un

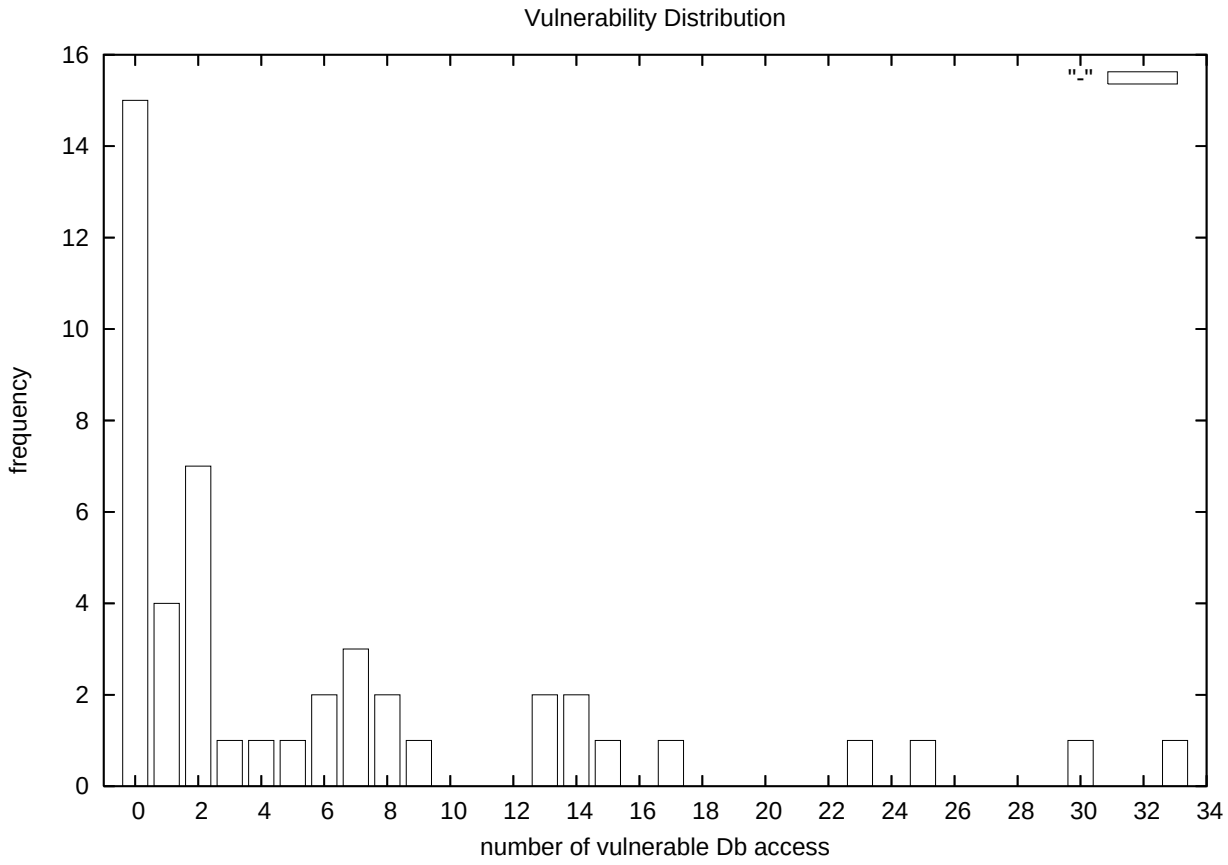

Figure 4.6 Histogramme des vulnérabilités.

nombre réduit de contextes qui représentent des sommaires des contextes de sécurité et puisque le nombre de niveaux d'autorisations distincts ne dépend pas de la taille du graphe de flux de contrôle et que le nombre de niveaux d'autorisation est petit. Alors, la complexité asymptotique et l'espace requis en mémoire de l'analyse peuvent être considérés comme linéaire en fonction de la taille du graphe de flux de contrôle. En effet, l'analyse présenté démontre une complexité asymptotique dans le pire des cas de $\mathcal{O}(k^2 \cdot |V|)$ où dans notre cas $k = 3$, k est le nombre de niveaux d'autorisations incluant la valeur initiale d'initialisation c'est à dire $(user, admin, \top)$ et $|V|$ représente le nombre de nœuds dans le graphe de flux de contrôle. Même si l'expérience présentée est seulement basée sur les niveaux de sécurité usager et système, l'analyse peut intrinsèquement être étendue à un plus grand nombre de niveaux. Cependant, il faut garder à l'esprit que la complexité asymptotique de l'analyse contient une composante qui croît avec la cardinalité des niveaux d'autorisations plus un. Alors, un nombre élevé de niveaux d'autorisations peut avoir un impact négatif sur le temps d'exécution. Une

discussion plus exhaustive concernant la linéarité de l'approche inter-procédurale utilisée par l'analyse décrite dans ce chapitre est disponible dans [23].

L'analyse étudiée est rapide en pratique lorsqu'elle est utilisée sur le système étudié. La convergence du point fixe a été atteinte après avoir traité 1.079 fois les nœuds dans le graphe de flux de contrôle. Ce ratio est très près de un et il est beaucoup moins élevé que la complexité maximale qui est égale à neuf fois la taille du graphe de flux de contrôle. La mise à l'échelle de l'approche sur de grands systèmes semble très prometteuse à cause de la complexité asymptotique qui est linéaire et parce que l'analyse est rapide en pratique sur le système étudié.

Les équations de flux présentées sont conçues d'une manière conservatrice et ne produisent pas de faux négatifs. Il est possible que des faux positifs sont introduits par les approximations de la sémantique du langage PHP utilisé dans les équations de flux.

Les appels externes et les appels aux librairies de fonctions ont étés approximés par leur niveaux de concessions de sécurité assumées. Pour les résultats présentés, nous avons assumé que les appels au système et aux librairies de fonctions propagent en sortie le même niveau d'autorisation que les niveau ausquels ils ont été appelées.

PHP permet des appels de fonctions variables effectués à l'aide de variables qui contiennent des chaînes de caractères arbitraires. Ce type d'appels joue le même rôle que les pointeurs de fonctions dans d'autres langages. Aussi, PHP permet d'utiliser des objets et lorsqu'ils sont combinés avec les appels de fonctions variables, le problème de l'analyse polymorphique se pose. L'analyse des appels de fonctions variables et l'analyse du polymorphisme en PHP n'ont pas été abordées dans ce travail. Les problèmes d'analyses qui sont sensibles aux flux sont la plupart du temps coûteux en temps de calcul et les technique d'analyses qui sont insensibles aux flux souffrent d'un manque de précision.

Malheureusement, en PHP des appels de fonctions variables peuvent être effectués à partir de chaînes de caractères et ainsi les analyses sur les appels de fonctions variables ne peuvent donc pas être effectués de la même manière que d'autres langage qui utilisent les informations de types sur les variables et les objets [78, 83, 84]. En effet, les chaînes de caractères ne sont pas associées à des informations sur les types, l'analyse des types demanderait une analyse analogue à celle présentée pour Java dans [25]. Pour le moment, une approche conservatrice a été adoptée pour l'approximation des appels variable à des fonctions. Ces variables textuelles peuvent contenir le nom de n'importe quelle fonction dans l'application ou de n'importe quelle fonction du système. Conséquemment, toutes les fonctions appelables par le système peuvent recevoir un flux d'information contenant un niveau d'autorisation venant d'un des points d'appels variable à des fonctions. Ce n'est pas un problème au niveau de la performance computationnelle de l'analyse parce que l'analyse utilise un algorithme par point-fixe avec

une complexité linéaire en fonction de la taille du graphe de flux de contrôle et non du nombre potentiel d'appel de fonctions. De plus, dans l'application Web étudiée les appels variables à des fonctions ont été utilisés minimalement et il n'y a pas de conséquences sur la précision des résultats présentés.

Des erreurs dans la conception ou dans la mise en œuvre des équations de flux pourraient produire des faux positifs ou des faux négatifs. La question de la précision de l'analyse présentée en terme de faux positifs et de faux négatifs n'a pas été explicitement examinée dans ce chapitre mais elle pourrait être évaluée dynamiquement en utilisant un ensemble représentatif d'attaques internes et externes. En général, il n'est pas facile d'obtenir ces ensembles d'attaques représentatives et il peut être très coûteux en temps et en effort pour les construire. Cependant, l'ensemble d'attaques utilisé dans [1] a été exécuté sur phpBB. Les 71 accès vulnérables à la base de données ont été visé par 176 attaques et on tous été correctement identifiés comme vulnérable par l'analyse présenté dans ce chapitre.

De plus, la conséquence d'un faux positif serait l'insertion d'une vérification de sécurité redondante dans l'application Web. L'ajout de tests redondants pendant l'exécution peut occasionner un surcroît de travail pour l'application et peut être indésirable du point de vue de performance. Tout de même, du point de qualité du logiciel l'ajout de tests de sécurité redondants peut augmenter la robustesse d'un système de la même manière que dans certains langages des tests redondants de validités des pointeurs ou des index des tableaux vont aussi augmenter la robustesse de l'application. Aussi, les tests de sécurité redondant peuvent réduire les dépendances de sécurité entre les composantes d'un système, ils peuvent ainsi protéger les accès à la base de données des répercussions de sécurité causées par des modifications à un fragment de code qui pourrait avoir des conséquences négative sur la sécurité à d'autres endroit dans le code.

L'analyse présentée est en grande partie indépendante du langage d'implantation de l'application au moins pour toutes les applications implantées dans un langage impératif. Tout langage avec une sémantique d'un style impératif pour lequel un parseur du code source est disponible pourra en principe être analysée avec la méthode proposée.

L'analyse est intrinsèquement sensible aux menaces internes et externes. Les menaces sont traitées en suivant les exécutions au niveau usager et en garantissant qu'aucune ne peut rejoindre un énoncé requérant des autorisations d'exécution du niveau système.

Supposons qu'un accès à une base de donnés a été protégé correctement en utilisant des barricades contre les menaces décrites dans [1] mais que par la suite un programmeur malicieux a inséré une porte arrière dérobée (back door) comme décrit dans la Figure 4.7. Aucun usager, ni même un administrateur du système ne découvrirait le comportement fautif ou étrange pendant les tests ou l'utilisation routinière du système. Heureusement, notre

analyse pourra trouver l'énoncé 2 comme vulnérable et une de deux interventions possibles pourra se produire : soit l'équipe de sécurité protégera par une barricade l'énoncé sensible avant le déploiement du système ou bien, après une inspection de sécurité, les énoncés 1 et 2 seront retirés après être reconnus comme une faille de sécurité.

```
1 if ($loginName = 'malicious_user')
2     mySql( ... parameters ... );
3 else
4     wrappedMySql( ... parameters ... );
```

Figure 4.7 Porte dérobée.

Supposons encore qu'un programmeur malicieux tente de cacher l'utilité du code malicieux par une séquence astucieuse appels de fonctions et de propagation de variables. Si notre analyse ne peut pas une manière conservatrice déterminer le niveau d'autorisation, le niveau d'autorisation sera alors abaissé au niveau usager annulant ainsi les menaces internes qui utilisent l'obfuscation de code pour camoufler leurs intentions malicieuses. Cependant, si le code difficile à analyser est vraiment légitime, les approximations conservatrices introduit par notre analyse vont nécessiter l'exécution de patrons concédant la sécurité supplémentaires. Nous croyons que c'est un prix relativement faible à payer pour garantir la sécurité des accès aux bases de données. Cependant une expérimentation plus exhaustive nécessiterait d'être effectuée pour évaluer la pénalité de performance occasionnée. À notre avis, lorsque des programmeurs développent des systèmes manipulant des informations sensibles, de toutes manières, ils devraient être invités le plus souvent possible à utiliser d'une manière standard et claire les patrons concédant la sécurité. Les développeurs de phpBB on peut-être intuitivement suivi cette ligne de pensée. Tous les fichiers contenant des fonctions administratives commencent clairement par un patron concédant la sécurité qui termine immédiatement l'exécution si l'usager ne possède pas les autorisations du niveau administrateur. L'avantage de notre analyse, dans ce contexte, est que la propagation des niveaux d'autorisations peut être vérifiée et appliquée automatiquement et ne dépend pas de la discipline des développeurs.

L'analyse présentée vise spécifiquement les attaques de type SQL-injection et ne prends pas en compte les autres types de patrons menaçants la sécurité. D'autres analyses et d'autres schémas de protections doivent être déployés pour atteindre des niveaux raisonnables de sécurité et de fiabilité globales.

Les niveaux d'autorisations dans l'application étudiée sont concédés par le patron décrit

dans la section 4.2.2. Les patrons concédant la sécurité sont éminemment dépendant de l'application et ne sont probablement pas réutilisables dans d'autres applications. L'analyse d'applications autre que phpBB requerrait le codage de nouveaux patrons dans le parseur utilisé pour produire le graphe de flux de contrôle.

L'analyse présentée est très rapide en pratique et peu être intégrée dans des processus qui requièrent des exécutions répétés et rapide de la même analyse. Une approche pour contrôler continuellement la qualité d'un logiciel à code source libre a été présentée dans [85]. L'analyse des vulnérabilités présentée pourrait très bien être utilisé comme un module externe dans l'architecture d'analyse continue permettant ainsi la surveillance continue de la conformité aux règles de sécurité concernant les SQL-injection. Dans le contexte de l'évolution de la sécurité, la présence de faux positif devient moins préoccupante, parce que les analyses d'évolutions tendent à ne pas tenir compte des biais constants. En effet, la surveillance continue de la conformité aux règles de sécurités pourrait signaler les changements dans les niveaux d'autorisations. Soit les changements réduisent le nombre d'énoncés vulnérables indiquant ainsi l'action d'un développeur rendant le système plus sécuritaire ou alors les changements augmentent le nombre d'énoncés vulnérables indiquant ainsi la possibilité d'une défaillance introduite par erreur ou par sabotage interne.

4.5 Conclusions

Une approche originale basée sur l'analyse statique inter-procédurale des flux de contrôle a été présentée, implémentée et évaluée. L'analyse est appliquée aux applications PHP pour détecter les vulnérabilités SQL-injection causé par des données malicieuses (attaque externe) ou par du code malicieux (attaque interne). phpBB 2.0.0 est une application Web de taille moyenne d'environ 34 000 lignes de code qui est bien connue pour le nombre significatif de vulnérabilités SQL-injection qu'elle comporte. phpBB a été analysée automatiquement par notre approche, l'analyse a démontrée une excellente performance au point de vue de la vitesse d'exécution et de la consommation de la mémoire. La convergence de l'algorithme par point-fixe a été atteinte en 0.077 s après avoir traité 1.08 fois le nombre de nœuds rejoignables dans le graphe de flux de contrôles. La mise à l'échelle de l'approche sur de grands systèmes est très prometteuse dû à la complexité asymptotiquement linéaire de l'analyse et dû aux performances effectivement très rapides en pratique lors de l'expérimentation sur une application réelle.

Les résultats expérimentaux ont identifiés 406 vulnérabilités dans 15 d'entre elles venant du sous-répertoire /admin. Les vulnérabilités « admin » sont causées par une catégorie d'erreur nommé « validation des entrées » publiée dans les archives de phpBB [81, 82]. Le nombre

de vulnérabilités identifiées pour chacun des fichiers se situe entre aucune pour les fichiers correctement protégés et jusqu'à un maximum de 33 pour les autres fichiers.

L'analyse proposée semble très prometteuse sur le cas de test présenté. Plus de recherches et d'évaluations sont cependant nécessaires pour en évaluer la précision et la performance sur des applications PHP plus diversifiées et d'une plus grande taille. Plus de travaux de recherches sont aussi nécessaires pour déterminer la pénalité en temps d'exécution engendrée par la protection des vulnérabilités identifiées par l'analyse présentée. Une évaluation des temps d'exécution de phpBB serait nécessaire pour répondre à cette question.

Des travaux futurs pourraient aborder d'autres problèmes d'injection en plus des SQL-injection comme ceux possibles dans XPath [86]. D'autres formes d'attaques Web comme le Cross-Site-Scripting (XSS) pourraient être considérés. Aussi, l'analyse présentée pourrait être étendue à d'autres applications qui interagissent avec des bases de données mais qui ne sont pas nécessairement basées sur le Web.

Les résultats préliminaires de l'évaluation de la performance pratique de l'analyse des vulnérabilités indiquent que le déploiement de l'analyse des vulnérabilités dans un environnement de développement agile serait possible. L'intégration de l'analyse des vulnérabilités comme un plug-in dans une architecture d'analyse continue comme décrit dans [85] pourrait être étudier et pourrait permettre le contrôle continue de la conformité aux patrons protégeant contre les SQL-injections tout au long de l'évolution d'une application.

CHAPITRE 5

ÉVOLUTION DES VULNÉRABILITÉS SQL-INJECTIONS DANS UN LOGICIEL EN PHP

Les sites Web sont souvent un mélange de contenus statiques et de code qui interagis avec une base de données relationnelle. Le code implémentant un site Web évolue continuellement pour répondre aux besoins changeant des usagers et de l'évolution de la technologie. À mesure que les sites Web évoluent, de nouvelles versions du code, de nouvelles interactions avec les utilisateurs et de nouvelles fonctionnalités sont ajoutées au site Web et les versions précédentes des fonctionnalités sont retirées ou modifiées.

Les sites Web requièrent une attention soutenue aux paramètres de configurations et à la qualité du code source pour s'assurer de la sécurité, de la confidentialité et de la fiabilité de l'information publiée sur le site Web. Durant l'évolution, d'une version à la suivante, des défaillances de sécurité peuvent être introduites, corrigées ou ignorées.

Ce chapitre présente une étude de l'évolution des vulnérabilités de sécurité. Les vulnérabilités sont détectées en propageant et en combinant les niveaux de sécurité concédés (granted authorization levels) en suivant les arcs dans un graphe de flux de contrôle (CFG) inter-procédural et en tenant compte des niveaux de sécurité requis pour l'accès à la base de données en lien avec les attaques de type SQL-injection.

Ce chapitre rapporte les résultats d'une expérimentation réalisée sur 31 versions de phpBB, une application gérant un babillard électronique en PHP. Les versions 1.0.0 (9 547 LOC) jusqu'à 2.0.22 (40 663 kLOC) ont été utilisées pour cette étude de cas. Les résultats montrent que l'analyse des vulnérabilités peut être utilisée pour observer l'évolution des vulnérabilités de sécurité sur des versions successives d'un même logiciel. Des suggestions pour des travaux ultérieurs sont aussi présentées.

5.1 Introduction

Les systèmes informatiques évoluent continuellement pour répondre aux besoins des usagers qui changent sans arrêts. À mesure que le système évolue, de nouvelles fonctionnalités sont ajoutées et les fonctionnalités existantes sont modifiées ou retirés. Plusieurs applications Web sont utilisées pour distribuer de l'information d'une organisation vers différents usagers

1. Une version de ce chapitre intitulée « SQL-Injection Security Evolution Analysis in PHP [10] » a été présentée au *9th IEEE International Workshop on Web Site Evolution* (WSE) durant la conférence « International Conference on Software Maintenance (ICSM) ».

à travers un réseau. La plupart du temps ces applications qui acceptent des interactions venant des usagers et effectuent des opérations sur la base de données. Normalement ces opérations supposent des données légitimes venant de l'usager et supposent la présence de code source légitime dans l'application pour construire les requêtes SQL. Ces applications peuvent être potentiellement vulnérables à des attaques de type SQL-injections qui exploitent des faiblesses dans la validation des données utilisées pour former les requêtes SQL.

Dans ce chapitre étudie les vulnérabilités de types SQL-injection qui sont causées par des données externes ou causées par du code malicieux présent dans l'application. Ces vulnérabilités sont étudiées en fonction de l'évolution des différences entre le *niveau de sécurité accordé* (granted authorization) et le niveau de *sécurité requis* (required security) aux différents points d'accès à la base de données dans le code source. Le chapitre 4 ainsi que [6] décrivent les équations de flux qui sont utilisées pour étudier l'évolution des vulnérabilités. Le chapitre 3 et [7] décrivent une approche basée sur la prévention des attaques SQL-injection en construisant automatiquement des barrières de protection (wrapper) appelées *barricades*. Les barricades effectuent une vérification des requêtes SQL en utilisant des patrons syntaxiques avant chaque appel à la base de données. Les patrons syntaxiques peuvent être construits automatiquement à partir du profil d'une analyse dynamique obtenue par une version instrumentée de l'application.

5.2 Analyse inter-procédurale

5.2.1 Définition du problème

La communication entre une application et une base de données s'effectue à l'aide de d'appels à un interface de programmation (API) d'accès à la base de données. Un exemple typique d'interaction entre PHP et une base de données MySql [60] est l'appel `mysql(str,db)` où `str` est une chaîne de caractères qui contient une requête SQL et où `db` réfère à la base de données à utiliser. Les appels aux routines de l'API de la base de données sont stéréotypés et sont facilement identifiable dans le code source PHP. Conséquemment, les appels à l'API de la base de données sont désignés comme des *accès BD* sans spécifier l'interaction particulière qui peut être `SELECT`, `UPDATE` ou une autre opération sur la base de données.

Les énoncés vulnérables aux menaces de type SQL-injections peuvent être définis comme des énoncés *accès BD* qui requièrent un niveau de sécurité $sLev$ pour leurs exécutions mais qui peuvent être rejoints par un chemin d'exécution dans un contexte inter-procédural j dont le niveau d'autorisation $aLev_j$ est plus petit que le niveau de sécurité requis $sLev$. Le niveau de sécurité requis pour effectuer des opérations sur la base de données devrait être défini durant la phase de spécification de l'application et peut dépendre des opérations à effectuer

et des tables à manipuler. Le niveau d'autorisation est déterminé dynamiquement durant l'exécution et est typiquement déterminé par une routine qui vérifie un nom d'usager et un mot de passe. Voir [19, 20, 22, 24] pour plus détails sur les niveaux de sécurité requis et concédés.

5.2.2 Analyse des vulnérabilités

L'analyse des vulnérabilités est définie en deux étapes. La première utilise l'analyse statique des flux inter-procéduraux pour déterminer les niveaux de sécurités propagés dans le code source PHP. La deuxième étape compare les niveaux de sécurité requis pour les *accès DB* avec les niveaux de sécurité calculé par l'analyse statique. Les divergences anormales entre les deux valeurs pour un même nœud du CFG sont rapportées comme une vulnérabilité potentielle.

Pour les besoins de l'expérimentation, nous avons défini le niveau d'autorisation requis pour l'accès sans contraintes à la base de données comme le niveau maximal *admin* (niveau système). Les patrons qui concèdent la sécurité sont dépendants de l'application. Un exemple typique d'un patron autorisant l'accès au niveau système *admin* est la ligne 7 dans la Figure 5.1. Le script pagestart.php appelé à la ligne 7 s'arrête subrepticement (die) si le niveau de privilèges de l'exécution est différent du niveau *admin*. La transition de la ligne 7 à la ligne 8 garantie que l'exécution continue au niveau système.

```
1  //
2  // Load default header
3  //
4  $no_page_header = TRUE;
5  $phpbb_root_path = "../";
6  require($phpbb_root_path . 'extension.inc');
7  require('pagestart.' . $phpEx);
8  //
9  // Do the job ...
10 //
```

Figure 5.1 Exemple d'un patron concédant la sécurité

Comme présenté dans le chapitre 4 et [6], l'analyse des vulnérabilités identifie les *accès BD* sans contraintes qui peuvent être rejoints par une exécution au niveau usager. L'analyse inter-procédurale permet de déterminer le niveau de sécurité pour chaque nœud v du CFG et pour chacun des contextes d'appels à partir de laquelle la fonction f_i contenant le nœud

v peut être potentiellement appelée. Les divergences entre le niveau de sécurité concédé par l'exécution et le niveau de sécurité requis pour les *accès BD* déterminent quels énoncés sont vulnérables aux attaques de type SQL-injection. L'évolution des *accès BD* vulnérables pour chacune des versions de phpBB à l'étude est calculée en analysant le nombre des divergences entre le niveau de sécurité concédé et le niveau de sécurité requis ainsi que le nombre des *accès BD* présent dans chacune des versions considérées.

5.3 Expérimentation

phpBB [79] est une application bien connue pour la possibilité des attaques de type SQL-injection dans ses versions antérieures. En effet, pour les dernières versions, un effort a été déployé pour réduire le nombre de vulnérabilités SQL-injections présentes dans phpBB.

Tableau 5.1 Taille du parser utilisé.

Nombre de règles dans la grammaire	73
Taille de la grammaire (LOC)	1 221
Taille du parseur (LOC)	11 033

Pour les expériences présentées nous avons choisi 31 versions de phpBB allant de 1.0.0 jusqu'à 2.0.22. L'analyse syntaxique a été effectuée en utilisant de la grammaire PHP de Satyam [63] implémentée en JavaCC [64]. La grammaire a été étendue en accord avec la syntaxe décrite dans [87]. La taille de la grammaire étendue est de 1 221 LOC et le parseur généré est de 11 033 LOC en code source Java comme indiqué dans la Tableau 5.1.

La taille de phpBB varie d'environ 9 000 LOC pour la version 1.0.0 jusqu'à environ 40 000 LOC pour les dernières versions analysées. Entre les versions 1.4.4 et 2.0.0 la taille de phpBB à augmentée d'environ 22 000 LOC jusqu'à environ 38 000 LOC et est restée relativement stable par après comme indiqué dans la Figure 5.2. Les tailles en LOC incluent le code en PHP et en *html* présent dans le code source de phpBB. La taille du code a été déterminé par le nombre de caractères de changement de ligne présent dans le code source (tel que rapporté par la commande wordcount de unix « `wc -l` »). Le graphe de flux de contrôle inter-procédural a été produit en GXL [80] (Graph eXchange Language) qui est un standard XML pour la représentation des graphes.

L'expérimentation a été effectuée sur un Pentium 4 de Intel cadencé à 3 GHz avec 1 GB de RAM sous Linux Fedora Core 5.

La Figure 5.3 dépeint la tendance évolutive du nombre total d'*accès BD* et l'évolution du

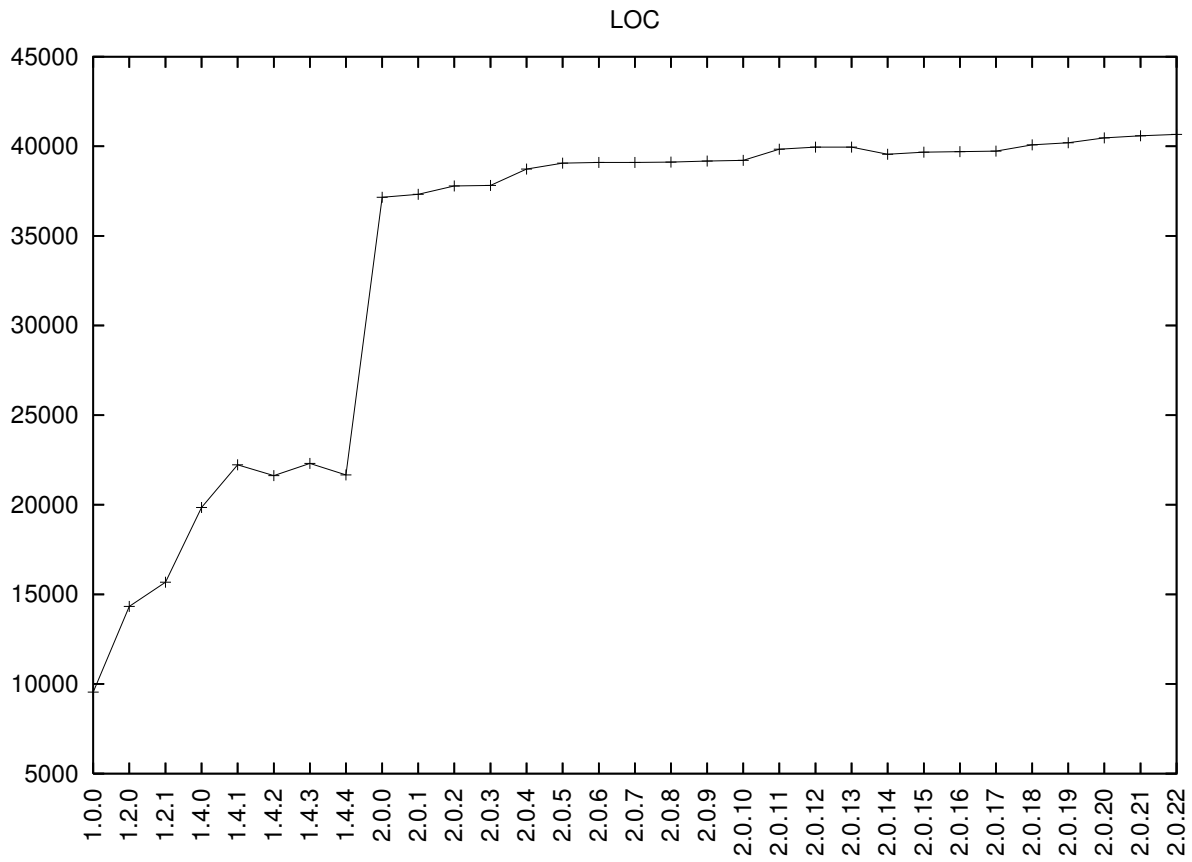

Figure 5.2 Nombre de lignes de code (LOC) par version de phpBB.

nombre d'*accès BD* vulnérable tel que détecté par notre analyse. En passant de la version 1.4.4 à la version 2.0.0, le nombre d'*accès BD* et le nombre d'*accès BD* vulnérables montrent une hausse soudaine tandis que entre les version 2.0.2 et 2.05 le nombre total d'*accès BD* continue à augmenter alors que les *accès BD* vulnérables diminuent. Après la version 2.0.5 le ratio entre le nombre d'*accès BD* vulnérable et le nombre total d'*accès BD* reste stable avant de recommencer à augmenter comme indiqué en pourcentage dans la Figure 5.4.

Les résultats peuvent être raffinés selon le répertoire du code source dans lesquelles les vulnérabilités apparaissent. Les répertoires '/contrib','/db','/includes' et '/language' contiennent des scripts, qui lorsqu'ils sont appelés au niveau de sécurité *user*, ne vont jamais augmenter leur niveau de sécurité au niveau système *admin*.

Les scripts dans les répertoires '/' et '/language' ne sont jamais appelés au niveau système *admin* tandis que les scripts dans les répertoires '/admin', '/contrib', '/db' et '/includes' peuvent être appelés au niveau système *admin*. Néanmoins, les seuls scripts qui peuvent changer le niveau de sécurité de *user* à *admin* sont situés dans le répertoire '/admin' tel que

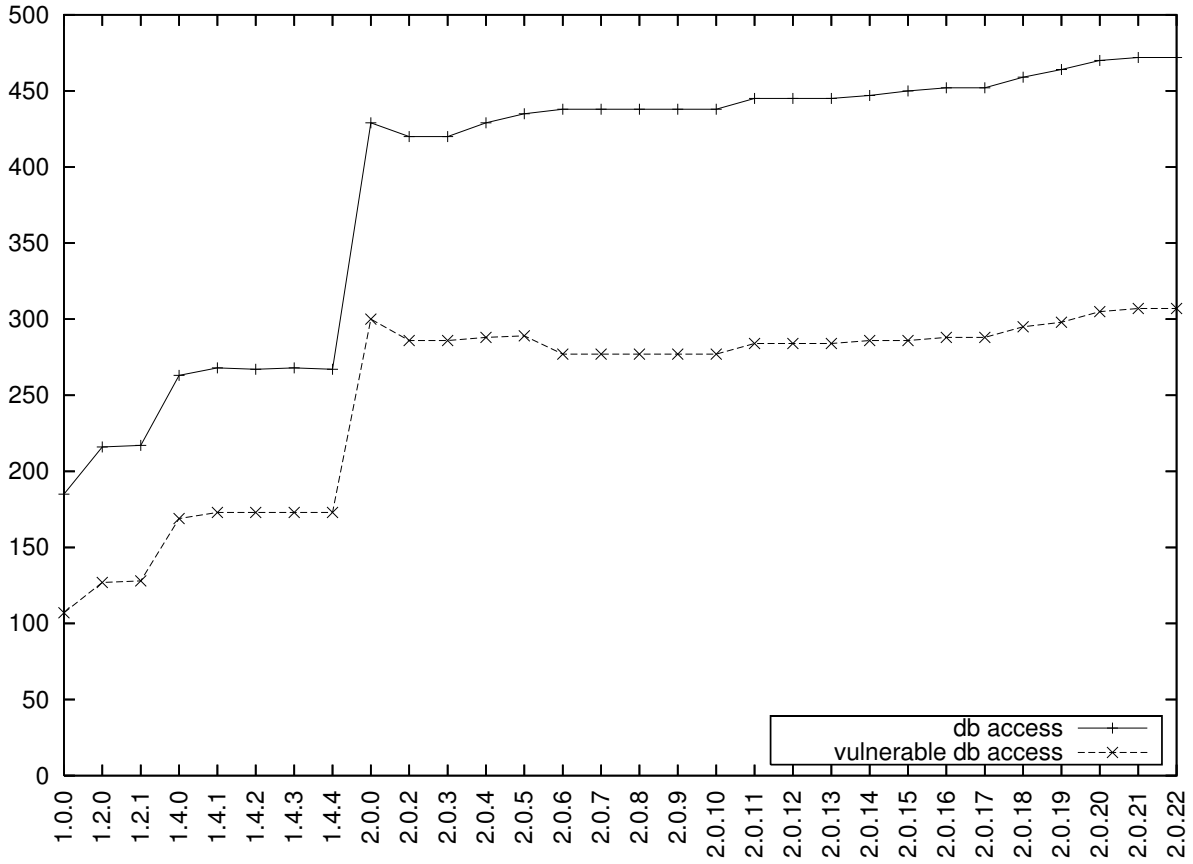

Figure 5.3 Nombre d'*accès BD* par version de phpBB.

l'on pourrait s'y attendre. À partir de ce point de vue, la Figure 5.5 représente l'évolution du nombre total *accès BD* vulnérables pour les fichiers contenus dans le répertoire '/admin'.

Comme on peut l'observer dans la Figure 5.5, 15 *accès BD* vulnérables apparaissent dans la version 2.0.0 et disparaissent à partir de la version 2.0.5. En effet cela correspond à une vulnérabilité identifiée par notre analyse au niveau d'un fichier d'administration admin/admin_styles.php . C'est une vulnérabilité déjà connue et identifiée comme « File Include Vulnerability, Bugtraq id : 7932 » de la classe « Input Validation Error » publiée le 16 juin 2003 [81, 82]. Le patron concédant la sécurité à la ligne 5 est exécuté conditionnellement et permet de contourner son usage tel que rapporté dans la discussion [82] :

> ... This could reportedly allow an attacker to include a malicious or sensitive local file. Information disclosure or execution of arbitrary commands could be the result.

La Figure 5.6 montre les détails de cette vulnérabilité qui est essentiellement basée sur l'exécution conditionnelle de la vérification des privilèges. La vulnérabilité a été éliminée à

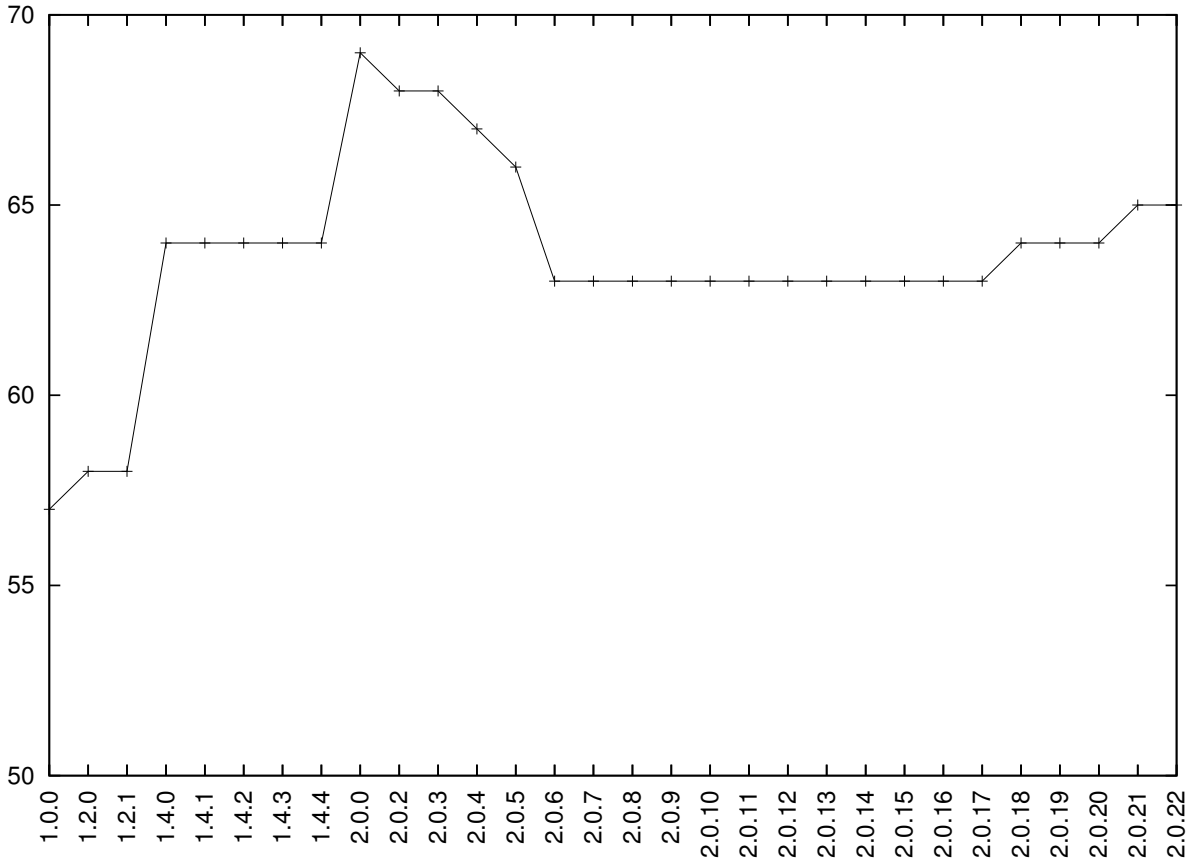

Figure 5.4 Pourcentage du nombre d'*accès BD* vulnérables sur le nombre total d'*accès BD*.

partir de la version 2.0.5 en effectuant en tout temps la vérification des privilèges tel que montré dans la Figure 5.1.

Dans l'expérimentation présentée, le ratio du nombre de nœuds traité par l'analyse de vulnérabilités avant d'atteindre la convergence de l'algorithme de point fixe en rapport avec la taille du CFG varie entre 0.95 et 1.1 comme indiqué dans la Figure 5.7. C'est très près de un suggérant qu'en pratique la convergence de l'analyse des vulnérabilités est très rapide. Pour les versions les plus récentes, le nombre des nœuds traités est un peu plus élevé que le nombre de nœuds du CFG alors que pour les versions les plus anciennes le nombre de nœuds traités peu être légèrement inférieur au nombre de nœuds du CFG. Ces ratios peuvent sembler étranges mais il faut considérer que du code non rejoignable ou abandonné est souvent présent dans le code source des systèmes analysés et qu'il traité de la même manière que le reste du code ainsi le nombre de nœuds traités peut être moins élevé que le nombre de nœuds identifiés d'une manière statique.

Le temps d'exécution du point fixe varie entre 0.1 s dans les versions les plus anciennes

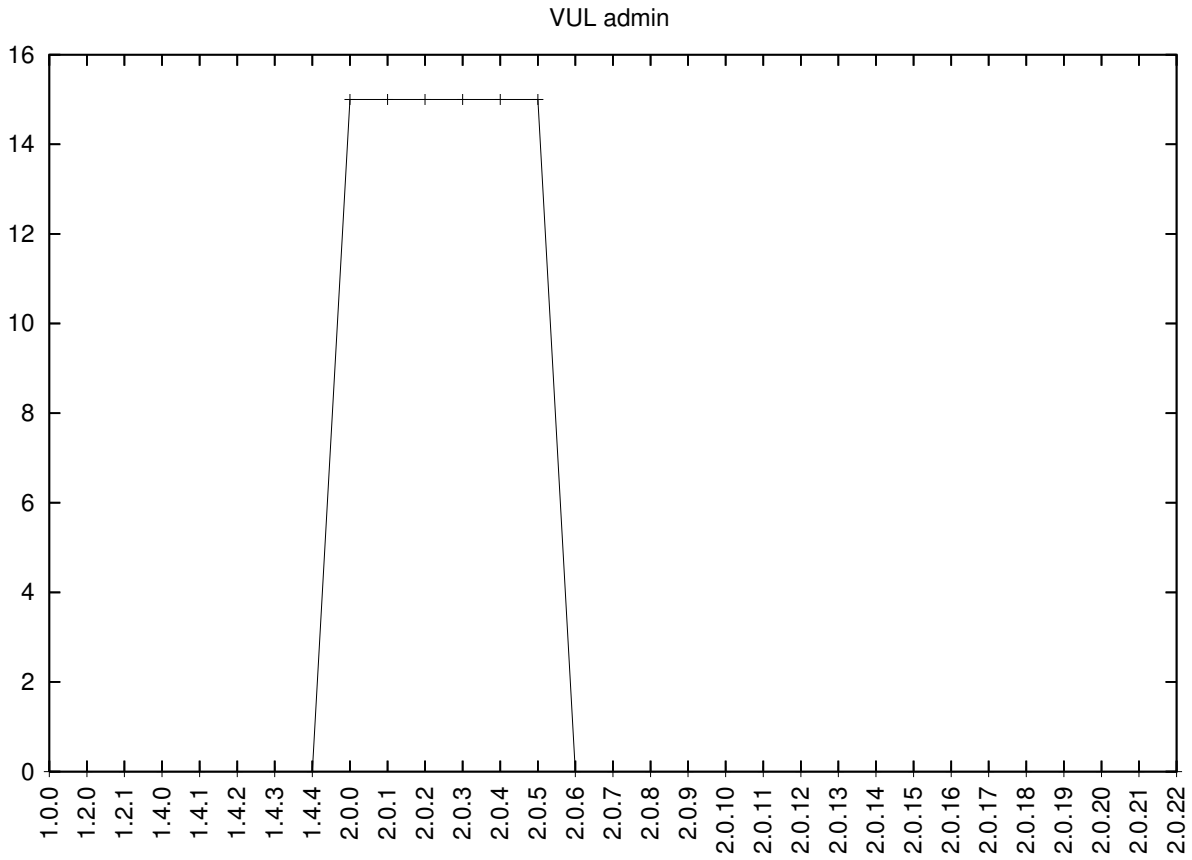

Figure 5.5 Évolution des vulnérabilités dans les scripts du répertoire '/admin'.

jusqu'à 0.5 s pour les versions le plus récentes. Le temps d'exécution total se situe toujours entre 2.5 s et 7 s comme illustré dans la Figure 5.8. La majorité du temps d'exécution, dans l'implémentation actuelle, est consacrée à la lecture du CFG en format GXL et à l'initialisation des tableaux initiaux pour l'analyse de flux.

5.4 Discussion

Dans les figures présentées traitant de l'évolution des vulnérabilités en particulier les Figure 5.3 et 5.4, le nombre d'*accès BD* vulnérables semble relié au nombre d'*accès BD*. Dans les versions étudiées de phpBB tous les accès à la base de données se font à l'aide de requête SQL construites sous forme de chaînes de caractères textuelles. Ces accès peuvent être vulnérables à des attaques de type SQL-injection et sont détectés par notre analyse. Probablement, ce ne sont pas tous les accès à la base de données avec des requêtes en

```
1  if (empty ($HTTP_POST_VARS['send_file'] ) ) {
2      $no_page_header = ( $cancel ) ? TRUE : FALSE;
3      require($phpbb_root_path . 'extension.inc');
4      require('pagestart.' . $phpEx);
5  }
```

Figure 5.6 Exemple de vulnérabilité

mode textuel qui représentent des vulnérabilités possiblement exploitables par un attaquant externe.

D'autre part, nous avons considérés pour les fichiers dans le répertoire '/admin' qu'un niveau de sécurité système *admin* est nécessaire pour tous les *accès BD* dans ces fichiers. Certains de ces *accès BD* dans le répertoire '/admin' ont été détectés comme vulnérables lorsqu'ils étaient rejoignables par une exécution avec un niveau de sécurité *user*. Le nombre d'*accès BD* vulnérables diminue entre les versions 2.0.0 et 2.0.5 et augmente à nouveau par la suite comme indiqué dans la Figure 5.4. Notre interprétation est qu'un effort de correction a été déployé après la sortie de la version 2.0.0 incluant la protection de quelques fichiers dans le répertoire '/admin'.

L'analyse présentée est largement indépendante du langage, au moins pour les langages du groupe des langages impératifs. Tout langage avec une sémantique de style impératif pour lequel un parseur est disponible pourra en principe être analysé avec la méthode proposée. L'analyse est intrinsèquement sensible à la détection des vulnérabilités internes et externes. La détection s'effectue en suivant les exécutions au niveau *user* et en s'assurant qu'elles ne peuvent pas rejoindre un énoncé qui requiert des privilèges *admin* pour s'exécuter. Un autre avantage de notre analyse est que l'évolution des vulnérabilités SQL-injection peut être automatiquement vérifiée et contrainte tout en ne reposant pas sur la discipline des programmeurs.

Le logiciel phpBB est une application Web de taille moyenne écrite PHP qui est bien connue pour sa susceptibilité aux attaques SQL-injection.

Plusieurs versions de phpBB ont été analysées pour suivre l'évolution des vulnérabilités de type SQL-injection. En terme d'espace mémoire utilisé et du temps d'exécution nécessaire, l'analyse démontre un temps d'exécution très performant d'environ 7 s après avoir traité environ 1.1 fois le nombre de nœuds du CFG pour chaque version. La mise à l'échelle de l'approche pour traiter de plus grands systèmes est très prometteuse due à la complexité

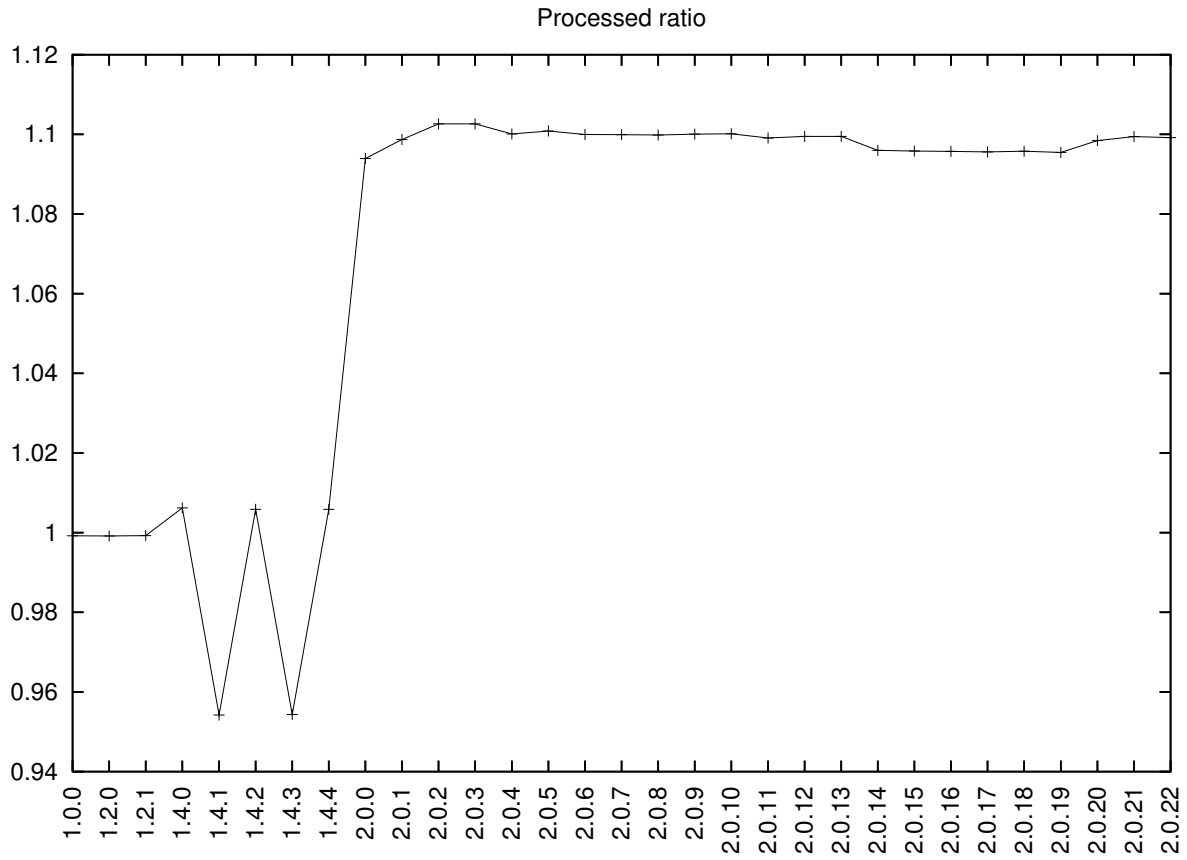

Figure 5.7 Évolution du ratio du nombre de nœuds traités.

asymptotiquement linéaire de l'analyse et sa vitesse d'exécution constatée en pratique sur l'application étudiée.

Le nombre de vulnérabilités dans phpBB semble relié avec la taille du logiciel possiblement parce que peu de protection contre les SQL-injection semble être apporté par l'utilisation des accès textuel à la base de données. Réciproquement, les scripts dans '/admin' semblent être bien protégés dans les versions récentes après que les corrections appropriées aient été effectuées.

L'analyse de l'évolution proposée semble intéressante lorsqu'elle est appliquée sur phpBB comme cas d'étude, mais plus d'études et d'évaluations sont nécessaires pour évaluer sa précision et sa performance sur des systèmes plus vastes et plus diversifiés.

Des travaux futurs devraient évaluer l'évolution d'autres problèmes d'injections comme ceux possibles dans XPATH [86]. D'autres formes d'attaques Web comme le Cross Site Scripting (XSS) [24] devraient aussi être considérées.

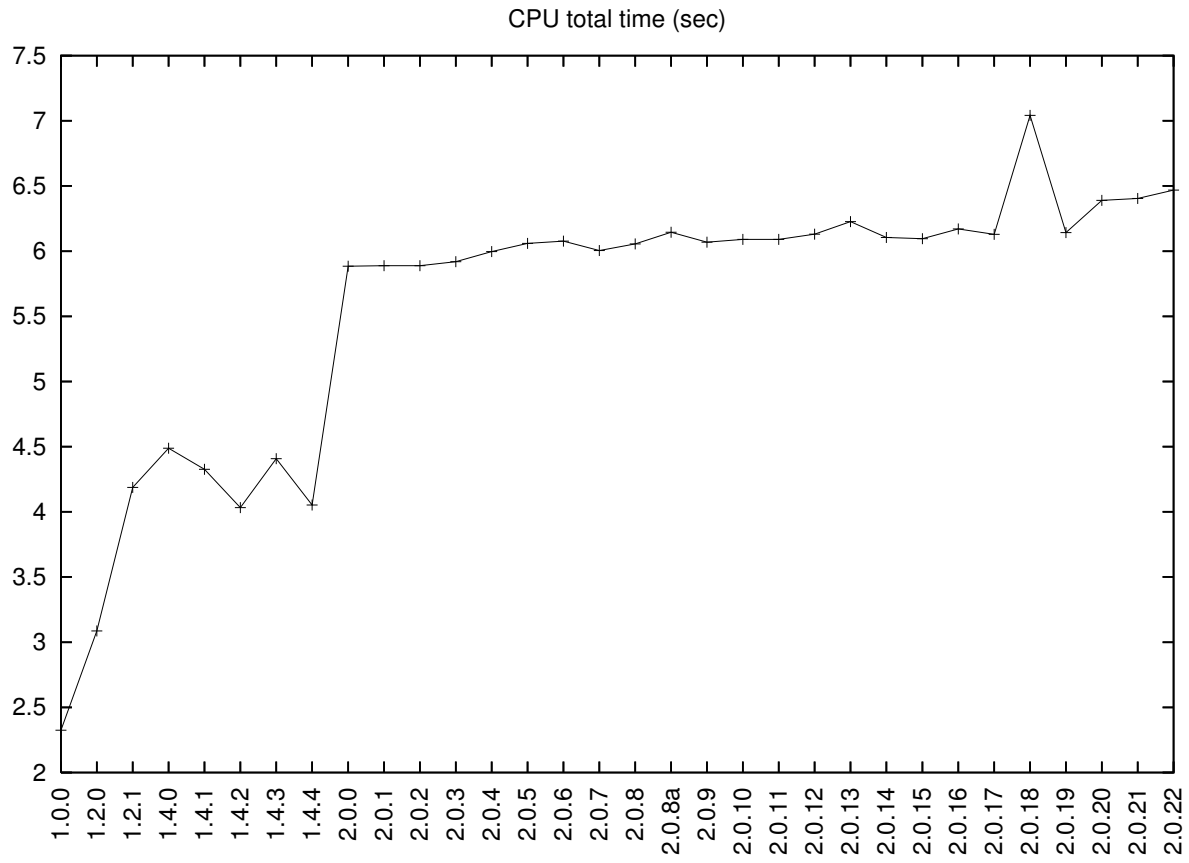

Figure 5.8 Évolution du temps total d'analyse pour une version de phpBB.

CHAPITRE 6

GÉNÉRATION DE CAS DE TESTS PAR MÉTHODES META-HEURISITIQUES POUR LA MAINTENANCE D'UN OPTIMISEUR DE REQUÊTES D'UNE BASE DE DONNÉES

L'utilisation de méthodes méta-heuristiques pour générer des données de test afin de tester les logiciels informatiques s'est affirmée comme une méthode de génération de tests efficace. Dans ce chapitre, nous explorons les problèmes rencontrés lorsqu'un algorithme génétique est utilisé pour générer des données de test pour une application d'une taille industrielle. La création des tests nécessaires pour valider l'optimiseur des requêtes SQL qui est utilisé par un SGBD implique la tâche ardue de créer des tests qui vont valider le système complexe sous-jacent basé sur des règles d'optimisations. Plusieurs approches connues peuvent simplifier cette tâche mais la plupart des approches vont tester l'optimiseur dans son ensemble et ces approches ne sont pas capables de viser spécifiquement une seule règle ou bien une sous-séquence de règles de l'optimiseur. Dans ce chapitre nous explorons l'utilisation d'un algorithme génétique pour générer des requêtes SQL de test qui vont provoquer l'exécution de sous-séquences de règles d'optimisations spécifiques. Les paramètres de l'algorithme génétique tel que la représentation du génome, sa reproduction, l'évaluation de son adaptation et sa sélection sont décrits. Les résultats préliminaires sont présentés et discutés, ils comparent l'approche utilisant un algorithme génétique avec un générateur de requêtes SQL aléatoire. De plus, nous présentons l'optimiseur de requêtes du SGBD DB2 (DB2 SQL Query Optimizer) qui est l'application que nous utiliserons comme étude de cas. Cette application est plus grande et plus complexe autant en terme de ligne de codes que de complexité des données utilisés lorsqu'on la compare avec les logiciels utilisés antérieurement pour étudier la génération de données de test avec des algorithmes génétiques.

6.1 Introduction

La phase des tests représente une part non négligeable de l'effort requis pour développer un logiciel en terme de ressources humaines et du temps requis. Une manière de réduire ce coût est d'essayer de générer automatiquement des cas de tests. Tel qu'indiqué dans la section 2.5 le domaine de la génération de données de test en utilisant des méthodes de recherche (search based test data generation) considère les algorithmes génétiques comme une approche efficace pour générer des cas de test qui satisfont un critère de test particulier.

La majorité dans travaux antérieurs sur les tests méta-heuristiques ont été effectués sur des logiciels de petite taille avec quelques centaines de ligne de code qui acceptent des données en entrée plutôt simple.

Nous proposons d'étudier l'utilisation d'algorithmes génétiques pour générer des données de tests pour un logiciel d'envergure industrielle. L'optimiseur de requêtes de la base de données DB2 d'IBM est un exemple d'une application de ce type. Le fonctionnement de l'optimiseur de requêtes de DB2 est similaire à ce que l'on pourrait s'attendre d'un optimiseur pour un langage informatique généraliste. En effet, SQL est un langage de programmation déclaratif. Comme il a déjà été rapporté dans plusieurs ouvrages d'introduction à DB2, l'optimiseur de requête est le cœur et l'âme de DB2. L'optimiseur analyse les requêtes SQL soumises et détermine quelles sont les tables et les colonnes qui doivent être utilisées puis il sélectionne la méthode la plus efficace et la plus rapide pour satisfaire la requête SQL.

Les optimiseurs de requêtes sont des éléments importants d'un SGBD. Ils sont responsables de générer des plans d'exécution de requêtes qui sont rapide tout en utilisant un minimum de ressources. Plusieurs optimiseurs de requêtes utilisés présentement sont construits en utilisant un système à base de règles, probablement pour faciliter la maintenance et les futures extensions du système. Par exemple, IBM Stardust [88] et Microsoft Cascades Framework [89] sont deux exemples d'optimiseurs de requêtes SQL basés sur des systèmes de règles de transformations.

Il est difficile de concevoir un algorithme générant automatiquement des cas de tests pour un optimiseur de requêtes SQL parce que la génération de requêtes SQL syntaxiquement correctes et respectant le schéma d'une base de données est beaucoup plus difficile que la génération aléatoire de valeurs numériques ou de simple chaînes de caractères. Les requêtes SQL générées doivent aussi être suffisamment complexes pour activer les fonctions plus spécialisées de l'optimiseur qui vont traiter par exemple les requêtes imbriquées ou les jointures multiples utilisant des clés étrangères.

Il est spécialement important de tester les modifications faites dans l'optimiseur de requêtes car même des changements mineurs dans un système basé sur des règles de transformations peuvent avoir des conséquences inattendues. Par exemple, des modifications peuvent engendrer des différences de performance pour des plans d'exécution de requêtes qui ne sont aucunement reliés aux règles qui ont été modifiés.

Un autre défi est la taille et la complexité du code source de l'application. Plusieurs expérimentation antérieures avec des méthodes méta-heuristiques ont étés limités à des programmes de quelques centaines de lignes de code (voir la section 2.5), la taille de l'optimiseur de DB2 est d'au moins un ordre de grandeur plus élevé que cela. Le code source contient

aussi des structures de données complexes et des structures de contrôles avancés qui sont nécessaires pour produire un optimiseur de qualité industrielle.

Il existe plusieurs manières de tester un optimiseur de requêtes SQL comme par exemple les approches stochastique employés par RAGS [37], les test génétiques incluant une rétroaction [38] ou un système de patrons pour tester les systèmes basés sur des règles de transformations [39]. Ces approches sont toutes capables de générer un jeu de test qui vise une couverture étendue des composantes d'un optimiseur de requêtes.

Alors que les optimiseurs existant vieillissent et deviennent plus mature, les nouveaux développement majeurs sont remplacés par de la maintenance corrective et par des ajouts de fonctionnalités spécifiques. Ces tâches vont souvent demandées de modifier un sous-ensemble restreint de règles présentes dans le système. Ce mode opérationnel cause des difficultés à l'étape des tests parce que le temps nécessaire pour exécuter un jeu de test généraliste existant peut être plus long que le temps qui a été nécessaire pour effectuer une tâche de maintenance simple. Ou plus difficile encore, si le jeu de test à été généré en utilisant des connaissances extraites du code source de l'optimiseur comme [38, 39], il pourrait être nécessaire de régénérer le jeu de test pour prendre en compte les règles qui ont étés créées, modifiées ou retirées ce qui nécessitera encore plus de temps avant de pouvoir obtenir le résultat des tests.

Ces tâches de maintenance provoquent des besoins en test plus spécifiques. Il y a un besoin ce générer automatiquement des requêtes qui vont cibler seulement un sous-ensemble de règles plutôt que l'optimiseur en entier. Notre approche est basée sur un algorithme génétique pour créer ce type de requêtes. Elmongui et al. [39] ont présentés un système qui résout un problème similaire mais il est basé sur un ensemble de patrons de règles pour générer des requêtes. D'autres approches pour tester les optimiseurs [90, 91] proposent de générer le contenu d'une bases de données afin de satisfaire des contraintes de cardinalités sur des sous-expressions d'une requête. Aussi dans [40] les requêtes générées prennent en compte le contenu de la base de données.

Le but de ce travail est de guider le processus de création de requêtes SQL vers un ensemble de requêtes qui tendront vers la satisfaction d'un critère de test spécifique comme la couverture d'un groupe de règles qui a été modifié récemment. Nous voulons réaliser cette tâche le plus efficacement possible en utilisant les techniques du test de logiciel basés sur des outils de recherche (search based software testing) plutôt qu'en générant simplement des requêtes aléatoires. L'objectif est de réduire le temps nécessaire pour produire un groupe de requêtes SQL qui répond à des critères de test.

Nous nous proposons d'étudier l'usage des algorithmes génétiques pour générer des données de test pour l'Optimiseur de Requêtes SQL de DB2 d'IBM (IBM DB2 SQL Query Optimizer) visant un enchaînement spécifique d'une séquence de règles d'optimisations. La

portée de ce projet est d'explorer l'utilisation d'algorithmes génétiques pour tester des logiciels complexes définis en terme de la taille du code source, de la complexité de données entrantes et de la complexité du modèle interne du programme visé par les tests.

6.2 Technique

L'objectif de l'algorithme génétique proposé est de diriger le processus de génération de test vers des séquences de règles d'optimisations d'un modèle de chaînage des règles plutôt que de viser directement la couverture du code en terme de couverture des blocs ou des branches.

Les séquences de règles d'optimisations correspondent à des chemins d'exécution dans le code source, il est difficile d'associer une séquence de règles d'optimisations avec une trace d'exécution spécifique parce qu'il existe beaucoup de variations de traces d'exécutions qui vont correspondent à la même séquence de règles d'optimisations. Les séquences de règles d'optimisations peuvent être identifiés explicitement par l'optimiseur lui-même, s'il est configuré en mode de débogage ou de profilage, ou alors elles peuvent être reconnues par des énoncés identifiables ou par des sondes (probes) ajoutés dans la trace d'exécution.

L'enchaînement des règles peut être extrait d'une trace d'exécution en observant la séquence des sondes qui sont présentes dans la trace d'exécution. Des enchaînements de règles d'optimisations différentes vont produire des séquences de sondes différentes, mais le même enchaînement de règles d'optimisations correspondant à la même séquence de sondes peut apparaître dans des traces d'exécutions contenant des énoncés différents entre les sondes. Ces énoncés différents entre les sondes pourraient faire partie ou non du code de l'optimiseur ou bien faire partie d'un élément associé à l'optimiseur tel que des fonctions utilitaires ou des fonctions de gestion de la mémoire.

Les tests basés sur un modèle d'enchaînement des règles sont différents des tests basés sur une métrique de couverture des branches ou de blocs. La génération de requêtes qui vise à tester une paire de règles d'optimisations correspond à couvrir au moins un chemin d'exécution entre le commencement de la première règle jusqu'au début de la seconde règle sans qu'aucune autre règles soit présente entre les deux. Le nombre de chemins ainsi défini entre les deux règles de la paire est potentiellement infini. Les règles contiennent plusieurs appels ainsi que des transferts de contrôle à des énoncés ou des fonctions qui ne font pas partie directement des règles d'optimisations. Ainsi, les tests basés sur un modèle d'enchaînement des règles d'optimisations peuvent être considérés comme un petit sous-ensemble très significatif des tests pour la couverture des chemins basés sur un modèle d'enchaînement des règles.

Les tests d'une séquence de règles d'optimisations dans le cadre d'un modèle d'enchaîne-

ment des règles d'optimisations peuvent être considéré comme des tests de couvertures des chemins. Ce qui est un problème difficile car la couverture des chemins est un problème combinatoire souvent impossible à résoudre. Les algorithmes génétiques sont souvent efficaces pour faire des recherches dans de très vastes espaces comme les problèmes combinatoires.

Les tests basés sur des modèles sont faits en à partir d'un model d'enchaînement des règles d'optimisation du système à tester. Puis des cas de tests sont générés pour viser une séquence de règles spécifique du modèle.

L'équation (6.1) définie le modèle M qui est utilisé pour représenter le système, $RULES$ représente l'ensemble des règles présentent dans le système et E représente l'ensemble des transitions (edges) entre les règles.

L'équation (6.2) définie les symboles utilisés dans ce chapitre pour les règles, les sondes et les fonctions associées. Les fonctions $rule()$ et $id()$ associent un identificateur numérique unique à chacune des règles. La fonction $prb()$ associe une sonde avec une règle.

Une règle débute soit à un énoncé spécifique ou au début d'une fonction qui est défini en tant que point d'entrée de la règle. L'exécution d'une règle correspond à un chemin parmi plusieurs chemins possibles allant du point d'entrée d'une règle jusqu'au point d'entrée de la règle suivante.

$$\begin{aligned} M &= (RULES,\ E) \\ E &= \{\ (r_i,\ r_j)\ \} \end{aligned} \tag{6.1}$$

$$\begin{gathered} PRB = \{stm_i\},\ \textit{set of statements used as rule probes} \\ RULES = \{r_j\},\ \textit{set of optimization rules} \\ prb : RULES \rightarrow PRB \\ id : RULES \rightarrow R\_ID \\ rule : R\_ID \rightarrow RULES \end{gathered} \tag{6.2}$$

Une trace d'exécution tr de l'optimiseur est collecté pendant l'exécution et est définie dans l'équation (6.3) comme une séquence d'énoncés stm. Un sous-ensemble $rule\_profile$ est composé d'énoncés qui sont associé aux sondes. Les transitions entre les règles sont déterminées en observant l'exécution des sondes dans une trace d'exécution conventionnelle.

$$\begin{gathered} tr = (stm_1,\ stm_2,\ \ldots,\ stm_h,\ \ldots,\ stm_n) \\ rule\_profile = (prb_1,\ prb_2,\ \ldots,\ prb_k,\ \ldots,\ prb_m) \\ rule\_profile \subseteq tr \\ m \leq n \\ (\forall prb_w \in rule\_profile, \\ \exists stm_u \in tr \mid stm_u = prb_w) \end{gathered} \tag{6.3}$$

La séquence cible de règles d'optimisations *target_rule_seq* est défini comme une séquence de transitions dans le modèle d'enchaînement des règles comme indiqué dans l'équation (6.4). Un ensemble de ces séquences représente l'objectif à atteindre par l'algorithme génétique. À la fin du processus évolutif, la population finale de l'algorithme génétique va être constitué de requêtes SQL dont le traitement par l'optimiseur exécutera des séquences de règles d'optimisations proche de celle visé par *target_rule_seq*.

$$\begin{aligned} target\_rule\_seq \;=\; & (trId_1,\; trId_2,\; \ldots,\; trId_m) \mid \\ & (\forall trId_i, 1 \leq i \leq m, \\ & (trId_i \in id(RULES))) \wedge \\ & (\forall trId_i, 1 \leq i \leq (m \;-\; 1), \\ & (id(trId_i),\; id(trId_{i+1})) \in E) \end{aligned} \tag{6.4}$$

6.2.1 Algorithme génétique

Les algorithmes génétiques font partie des algorithmes évolutifs. Ceux-ci sont bien adaptés pour faire une exploration basée sur la recherche (search-based exploration) dans un espace de recherche vaste et potentiellement infini. Dans notre cas, l'espace des solutions à explorer est l'ensemble de toutes les requêtes SQL qui peuvent être créées pour tester l'activation des séquences de règles d'optimisations. Un aperçu général adapté de [47] d'un algorithme génétique est présenté dans la Figure 6.1.

```
1 Generate random initial SQL query population;
2 repeat
3 |   Reproduce current generation by applying crossover and mutation;
4 |   Evaluate fitness of reproduced individuals;
5 |   Introduce selected reproduced individuals into next generation;
6 |   Introduce selected individuals from current generation into next one;
7 |   Make next generation become the current one;
8 until evolution criterion is satisfied;
```

Figure 6.1 Aperçu général d'un algorithme génétique.

Les algorithmes génétiques débutent avec une population initiale qui va évoluer génération par génération en utilisant la reproduction, l'évaluation de l'adéquation et la sélection. Les individus de la génération courante de la population sont reproduits via des opérateurs génétiques tel que la mutation et les croisements qui vont créer de nouveaux individus.

Les individus nouvellement créés peuvent être introduit dans la génération suivante dé-

pendamment du degré de satisfaction d'un critère de sélection. Ce critère peut être basé seulement sur les caractéristiques propres à un individu ou bien il peut être basé sur la comparaison de l'individu avec la génération suivante de la population. Les individus d'une génération précédente peuvent être retenus ou non pour la population suivante dépendamment la satisfaction de plusieurs critères comme une contrainte de stabilité sur plusieurs générations, la diversification des génomes dans la population et des contraintes sur la taille de population.

L'algorithme génétique que nous avons développé génère des requêtes SQL dont l'exécution déclenche des séquences de règles d'optimisations identique ou ressemblant le plus possible à des séquences d'optimisations ciblés. Les génomes qui représentent des requêtes avec des combinaisons de gènes intéressants sont reproduits et mutés pour créer de nouveaux génomes. Une fonction évaluant l'adéquation (fitness function) est utilisée pour déterminer si un génome nouvellement créé va être réutilisé plus tard pour créer de nouveaux génomes et ses requêtes associés. La fonction d'adéquation peut être définie de manière à tenir compte de ces objectifs de tests spécifiques.

Notre algorithme génétique génère des requêtes SQL en utilisant un générateur de requêtes aléatoire qui est configuré à l'aide d'un vaste fichier de configuration. Pour l'expérience présentée nous avons choisi une règle d'optimisation dans le système et nous avons laissé l'algorithme générer des requêtes contenant des séquences de règles d'une longueur prédéfinie contenant cette règle.

6.2.2 Paramètres de l'algorithme génétique

L'algorithme utilisé pour produire les résultats expérimentaux présentés est illustré par la Figure 6.2. Plusieurs paramètres de cet algorithme sont discutés dans cette section.

Dans notre cas, le génome d'un individu de la population est l'ensemble des paramètres de configurations qui contrôle le générateur de requêtes SQL aléatoire. Il y a un peu plus d'une centaine de paramètres de configurations stockés dans chaque fichier de configuration. Ces paramètres contrôlent des aspects du processus de génération tel que le nombre minimal et maximal de tables pouvant être utilisés dans une requête ou la probabilité d'utiliser un type particulier de jointure.

Les paramètres sont représentés en mémoire comme un structure de données associant chaque nom d'un paramètre avec sa valeur. L'ordre des paramètres dans le fichier de configuration n'est pas important. Chaque génome peut être représenté comme une table de hachage associant chaque paramètre avec sa valeur.

L'exécution du générateur de requêtes aléatoire avec un fichier configuration spécifique

```
1  P = ∅ // Population
   // Initialize initial population
2  while |P| < INITIAL_SIZE do
3  |  P = P ∪ NewRandomGenome();
4  end
   // Repeat for each generation
5  for index = 1 to N_GENERATIONS do
6  |  NEW_P ← P // genomes of reproduced generation
   |  // Create new genomes by crossover
7  |  NEW_G = ∅;
8  |  for i = 1 to |P| do
9  |  |  g_r ← RandomElement(P);
10 |  |  NEW_G = NEW_G ∪ CrossOver(g_i, g_r);
11 |  end
   |  // Add random genomes
12 |  for i = 1 to RANDOM_SIZE do
13 |  |  NEW_G = NEW_G ∪ NewRandomGenome();
14 |  end
   |  // Evaluate fitness and select genomes
15 |  for g ∈ NEW_G do
16 |  |  Generate random queries from configuration genome g;
17 |  |  Execute queries and collect the optimization rule sequence S;
18 |  |  if Fitness( P, S) then
19 |  |  |  NEW_P = NEW_P ∪ (g, S) ;
20 |  |  end
21 |  end
22 |  P = NEW_P;
23 end
```

Figure 6.2 Algorithme génétique.

produit un ensemble de requête SQL selon une distribution probabiliste de la taille, de la complexité et d'autres facteurs tout en respectant les paramètres du fichier de configuration.

Une population P d'individus i différents est un ensemble de paires $(g,\ S^n)$ où pour chaque individu i, g_i est un paramètre du génome et S_i^n est un ensemble de sous-séquences de règles tel que défini par l'équation (6.6). Les séquences S^n sont obtenues en générant les requêtes SQL spécifiées par le génome g_i et en optimisant ces requêtes avec le SGBD DB2. Une trace d'exécution de l'optimiseur est récupérée pendant l'exécution de la requête et finalement les séquences pertinentes de règles d'optimisation de longueur n sont extraites de la trace et sont stockées pour l'individu i en tant que l'ensemble S_i^n.

```
1 Join.Inner.Weight = 9
2 Join.Left.Weight = 76
3 Join.Outer.Weight = 54
4 Join.Right.Weight = 71
5 JoinPred.FakeJoin.Weight = 0
6 JoinPred.NameJoin.Exclude = ""
7 JoinPred.NameJoin.Weight = 68
8 JoinPred.Options.EQ.Weight = 75
9 JoinPred.Options.GT.Weight = 96
10 JoinPred.Options.GTE.Weight = 3
11 JoinPred.Options.JoinPreds.Include = ""
12 JoinPred.Options.LT.Weight = 43
13 JoinPred.Options.LTE.Weight = 47
14 JoinPred.Options.LocalPreds.Include = ""
15 JoinPred.Options.NEQ.Weight = 98
16 JoinPred.RealJoin.Weight = 2
17 JoinPred.SelfJoin.Allowed = Yes
18 MateFactor.Compute.AllJoins.Factor = 1
19 MateFactor.Compute.FakeJoin.Factor = 1
20 MateFactor.Compute.Maturity.Factor = 1
```

Figure 6.3 Vingt premières ligne d'un fichier de configuration.

La population initiale de génomes est produite aléatoirement en lançant un pilote qui va donner une valeur aléatoire mais valide à chacun des paramètres du fichier de configuration et puis va exécuter le générateur de requêtes en utilisant le fichier de configuration ainsi créés comme illustré par la Figure 6.4.

La fonction d'adéquation évalue l'adéquation d'un génome particulier conformément à l'objectif de l'évolution espéré. Un génome est considéré intéressant s'il soutient l'objectif des tests. Des requêtes SQL sont générés à partir d'un génome selon les paramètres de configurations spécifiés dans celui-ci, la requête est alors exécuté par le SGBD, la trace d'exécution de l'optimiseur est collectée et la séquence de règles d'optimisation est extraite et finalement cette séquence est évaluée par la fonction d'adéquation en fonction de l'objectif des tests.

Pour l'expérience présentée nous avons choisi une règle dans le système et nous avons laissé l'algorithme génétique générer des requêtes contenant cette règle.

Même si plusieurs définitions différentes peuvent être envisagées pour la fonction d'adéquation, dans le cadre de cette expérience la valeur d'adéquation a été calculé selon l'équation (6.5). La fonction Fitness(P, S_x^n) considère le nombre de n-uplets distincts contenant la règle visée rId_a. S^n est l'ensemble de toutes le sous-séquences de longueur n contenant la règle rId_a dans $optimization\_rule\_seq$.

Dans l'équation (6.6), $orId_i$ identifie une règle dans la séquence de règles d'optimisations

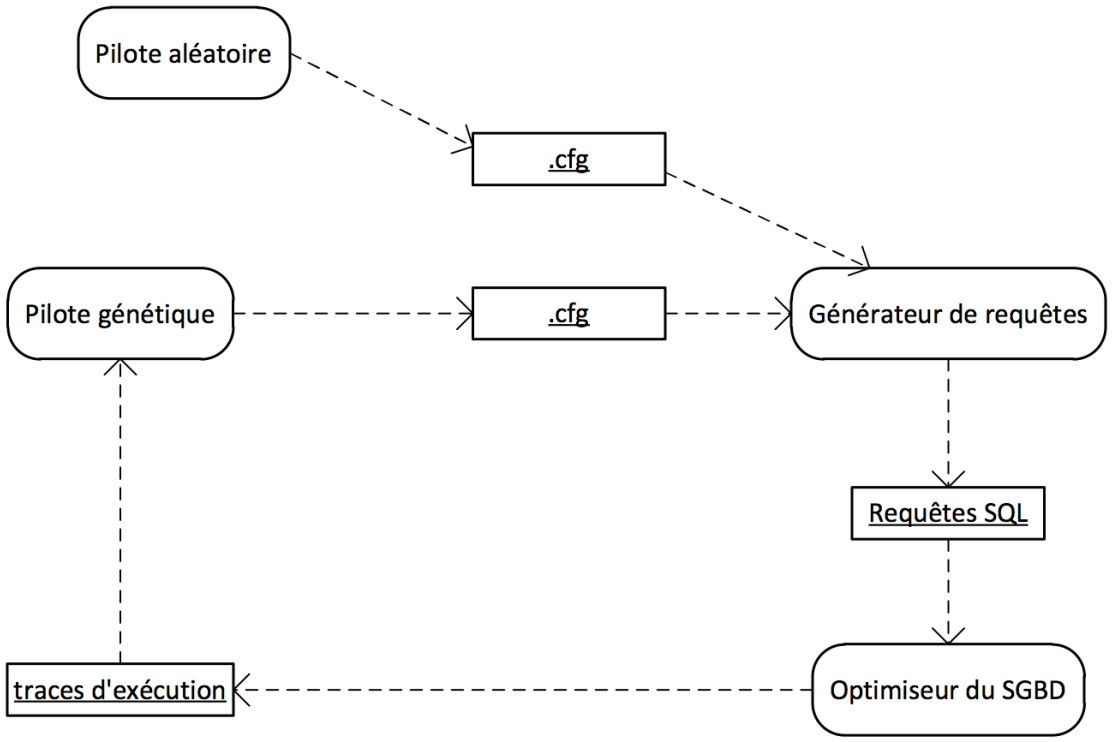

Figure 6.4 Diagramme montrant la rétroaction entre le générateur de requête aléatoire et le SGBD

lors de l'évaluation de son adéquation. *target_rule_seq* doit être une séquence plus courte que *optimization_rule_seq*. La fonction d'adéquation est définie de manière à maximiser la diversité des chaînes de règles contenant la règle visée. Nous avons expérimenté avec des chaînes de règles S^n dont la taille n varie entre un et quatre.

$$\begin{aligned} \text{Fitness}(P, S_x^n) : (g_x,\ S_x^n) \textit{is added to current population} \leftrightarrow \\ \leftrightarrow S_x^n \setminus \left\{ \bigcup_{i \in P} S_i^n \right\} \neq \emptyset \end{aligned} \tag{6.5}$$

$$
\begin{aligned}
target\_rule\_seq \;&=\; (rId_a) \\
optimization\_rule\_seq \;&=\; (orId_1,\; orId_2,\; \ldots,\; orId_m) \\
S^n = \{\; &(orId_x,\; orId_{(x+1)},\; \ldots,\; orId_{(x+n-1)}) \in \\
&optimization\_rule\_seq \mid \\
&(rId_a \;=\; orId_x) \vee \\
&\vee\; (rId_a \;=\; orId_{(x+1)}) \vee \\
&\ldots \\
&\vee\; (rId_a \;=\; orId_{(x+n-1)})\; \}
\end{aligned}
\tag{6.6}
$$

Le processus de sélection produit une nouvelle génération d'individus à partir en partie d'individus de l'ancienne génération et en partie d'individus nouvellement créé après l'étape d'évaluation de l'adéquation. Dans notre approche, tous les individus de l'ancienne génération sont retenus pour la génération suivante. Nous avons choisi de préférer les individus qui génèrent des requêtes déclenchant l'exécution de sous-séquence de règles d'optimisations qui n'avaient pas été exécutées par les requêtes précédentes conformément à l'objectif visé. Les individus nouvellement créés sont ajoutés à la nouvelle génération si leurs requêtes déclenchent l'exécution de nouvelles sous-séquences pertinentes pour l'objectif des tests et qui n'ont pas été vus dans la génération précédente de la population. De plus, un nombre restreint d'individus générés aléatoirement est ajouté à chaque génération.

Plusieurs stratégies peuvent être utilisés pour faire évoluer un génome. Deux des plus communes sont l'entrecroisement et les mutations. L'entrecroisement échange une ou plusieurs parties d'un fichier de paramètres avec des parties d'un second fichier de paramètres pour produire un nouveau fichier paramètres qui contient des parties de ces deux ancêtres. Ce nouveau fichier de paramètres sera considéré comme le génome d'un nouvel individu. La mutation effectue aléatoirement des modifications à certains paramètres dans un fichier de paramètres existant pour produire le génome d'un nouvel individu muté. Dans notre algorithme génétique plutôt que de faire des mutations aléatoires sur certains individus nous avons ajouté de nouveaux génomes entièrement générés aléatoirement à chacune des générations.

6.3 Expérience

Pour notre expérience nous avons utilisé une version du SGBD DB2 en version instrumentée pour le débogage. Pendant que les requêtes sont traitées par le SGBD l'option demandant la production de traces par l'optimiseur durant la phase de réécriture des requêtes est activée et l'enchaînement des règles d'optimisation est collectée.

La Figure 6.4 illustre le procédé. Le système démarre avec la génération de fichiers de

configurations aléatoires (.cfg) par le pilote aléatoire (random driver) puis le générateur de requêtes transforme les fichiers de configurations en requêtes SQL. La base de données DB2 exécute les requêtes et collecte les traces d'exécution de l'optimiseur. Un pilote utilisant un algorithme génétique (genetic driver) utilise les traces d'exécution pour évaluer l'adéquation des requêtes SQL et de leurs fichiers de configurations associés. Le pilote utilisant un algorithme génétique va alors produire une nouvelle génération de fichiers de configurations en utilisant des opérations de reproductions sur le génome de la génération actuelle.

La figure listing 6.3 montre les vingt première lignes d'un fichier de configurations. Les fichiers de configurations employés comporte environ 150 paramètres (et donc environ 150 lignes) qui contrôlent les probabilités de la génération des nombreuses constructions possibles dans les requêtes SQL comme les jointures, les prédicats et les opérateurs utilisés. La majorité des paramètres sont des entiers et les autres étant des chaînes de caractères.

La Figure 6.5 montre un exemple d'une des plus petite requête SQL générée par le générateur de requêtes SQL. Les requêtes générées varient normalement d'une taille de 30 lignes pour les plus petites jusqu'à plus de 150 lignes pour les plus grandes. Toutes les requêtes générées sont syntaxiquement valides, les noms des colonnes et des tables sont extraits du schéma de la base de donnée et les noms des autres identificateurs sont générés séquentiellement.

L'objectif de l'expérience est d'explorer l'hypothèse qu'en privilégiant des séquences de règles d'optimisations spécifiques avec un algorithme génétique, des données de tests pertinentes vont être générées plus rapidement qu'en augmentant simplement le volume ou la complexité de requêtes générés aléatoirement.

Pour notre expérience, nous avons choisi une règle dans le système et nous avons laissé l'algorithme génétique générer des séquences de règles d'une longueur prédéfinie contenant cette règle. La fonction d'adéquation est définie de manière à maximiser la diversité des chaînes de règles d'optimisations. Nous avons expérimenté avec des sous-séquences de règles S^n où la longueur n des séquences varie de deux à quatre.

Le système complet a été installé sur un serveur incluant le pilote aléatoire, le pilote utilisant un algorithme génétique, le générateur de requêtes SQL et le SGBD DB2 produisant les traces d'exécution de l'optimiseur de requêtes SQL. Le système complet a été exécuté pendant environ trente-six heures. Une autre exécution a aussi été produite afin d'obtenir une exécution de référence (baseline). Le générateur de requêtes SQL a été exécuté en avec seulement des fichiers de configurations générés aléatoirement. Pour cette exécution de référence aucune rétroaction de l'optimiseur n'a été utilisée pour produire ces nouvelles requêtes. La génération des requêtes a été arrêtée après trente-six heures soit la même durée que pour l'algorithme génétique. Le nombre de fichiers de configurations traitées durant l'exécution de l'algorithme génétique et durant l'exécution de référence est semblable.

```
SELECT COUNT(T500.WEBPAGE) as C7230,
        T500.TIME as C7431,
        T500.CONTEXTID as C7498
FROM "STARS"."CUSTVISIT" T500
WHERE T500.CUSTID > '1 '
AND (T500.PARENT_VISITID >= ' ' AND T500.PARENT_VISITID < ' ')
OR T500.TIME IN (TIMESTAMP('1990-04-02 09:46:26.000000'), TIMESTAMP('
    1990-03-15 21:49:58.000000'), TIMESTAMP('1990-06-29 13:17:23.000000'),
    TIMESTAMP('1990-07-17 08:07:57.000000'), TIMESTAMP('1990-09-09
    03:36:24.000000'), TIMESTAMP('1990-10-01 19:36:38.000000'), TIMESTAMP('
    1990-10-15 05:55:30.000000'), TIMESTAMP('1990-10-24 07:26:25.000000'),
    TIMESTAMP('1991-05-30 19:57:10.000000'))
AND T500.WEBPAGE < ALL (SELECT T499.PROD_WEBPAGE as C7546
          FROM "STARS"."PRODUCT" T499
          WHERE T499.WARRANTY <> 1
          AND T499.PROD_CID NOT IN (SELECT T476.PROD_CID as C7516
                    FROM "STARS"."PRODUCT" T476
                    WHERE (((((((T476.DESCRIPTION > 'eqitti '
                    OR T476.MEASURE_UNIT NOT IN ('', '', '', 'ft', 'dry-quart')
                    OR T476.WARRANTY IN (6, 9, 3, 1, 3, 7)
                    AND T476.PRODID < '938 '
                    OR T476.PGID NOT IN ('2 ', '2 ', '2 ', '2 ', '1 ', '1 ')
                    AND T476.PROD_CID > x'20061107174852982788000000')
                    AND T499.PGID <> T476.PGID)
                    AND T499.DESCRIPTION <= T476.DESCRIPTION)
                    AND T499.SAMPLE_PRODUCT <> T476.SAMPLE_PRODUCT)
                    AND T499.PROD_WEBPAGE > T476.PROD_WEBPAGE)
                    AND T499.MEASURE_UNIT > T476.MEASURE_UNIT)
                    AND T499.WARRANTY > T476.WARRANTY)
                    AND T499.PRODID > T476.PRODID
                    ORDER BY C7516
                    )
           OR T499.PROD_WEBPAGE = 'http://www.ffabic .com'
           OR T499.PGID IN ('1 ', '1 ', '2 ', '1 ', '2 ', '2 ', '2 ')
          )
GROUP BY T500.TIME,
        T500.CONTEXTID
ORDER BY C7230,
        C7431
;
```

Figure 6.5 Exemple d'une requête générée pour tester un optimiseur de requêtes d'un SGBD.

La Figure 6.6 compare le résultat de l'exécution de référence aléatoire et le résultat de l'exécution avec l'algorithme génétique. Il y a une différence marquée en faveur de l'algorithme génétique représenté par la ligne supérieur en bleu comparativement à la ligne basse en noir représentant l'exécution aléatoire. L'axe vertical montre le nombre de sous-séquences distinctes contenant la règle visée qui ont été trouvée à ce point du processus. La longueur

n des sous-séquences varie de deux à quatre selon les graphiques. L'axe vertical est identifié avec le nombre de génomes générés (correspondant au nombre de fichiers de configurations générés). Par exemple, dans le graphe S^4 lorsque l'algorithme génétique termine 291 génomes ont été générés et 228 sous-séquences différentes contenant la règle visée ont été trouvées. La génération aléatoire de référence s'arrête après approximativement le même nombre de génomes et seulement 166 sous-séquences contenant la règle visée ont été identifiées.

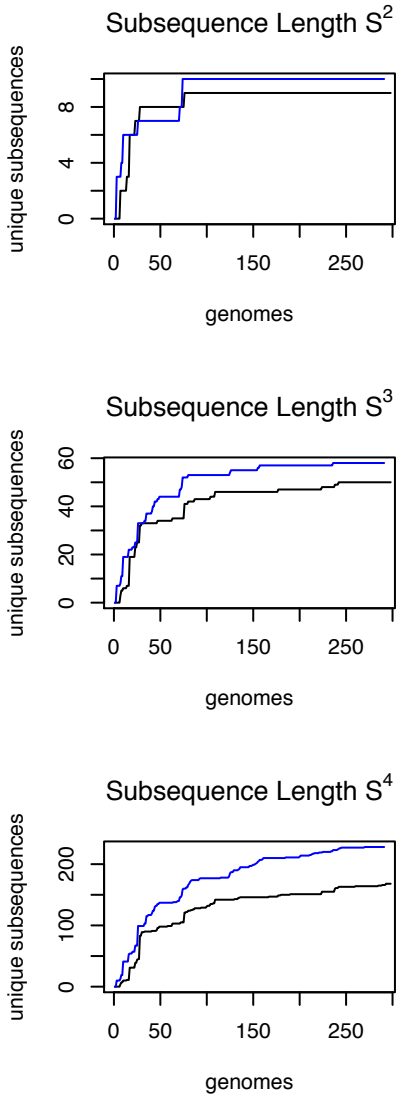

Figure 6.6 Effet de la taille des sous-séquences.

Un problème commun lors de la génération de requêtes SQL par des méthodes probabilistes est la présence de requêtes qui ne retourne aucun résultat ou qui sont sémantiquement invalide. Ce problème apparaît aussi dans le générateur de requêtes SQL lorsque certaines combinaisons de paramètres empêchent la génération de requêtes.

La Figure 6.7 montre les résultats obtenus en visant des sous-séquences de règles de longueur quatre lorsque seulement les requêtes qui ont produit des résultats sont conservées. Nous avons éliminé tous les génomes qui ne produisent pas de requêtes après le processus de génération et aussi ceux qui ont produit des requêtes qui n'activent aucune règle d'optimisation. Les trois graphiques de la Figure 6.7 ont une forme similaire démontrant que l'approche utilisant l'algorithme génétique produit des requêtes qui produisent plus de sous-séquences de règles d'optimisation que la génération aléatoire même en considérant seulement les fichiers de configuration valides.

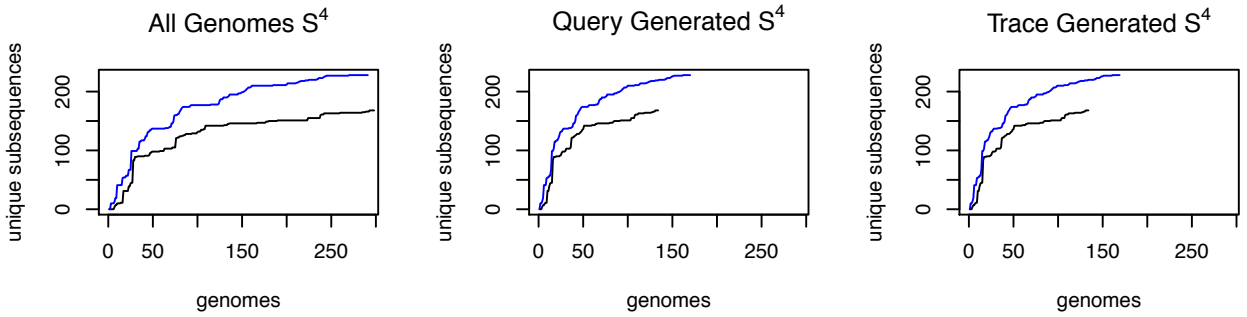

Figure 6.7 Effet de l'élimination des fichiers de configuration qui ne génèrent pas de requêtes et de l'élimination des requêtes qui ne génèrent pas de traces d'exécution.

6.4 Discussion

La Figure 6.6 montre les effets du choix de la longueur des sous-séquences de règle d'optimisation considérés. Les sous-séquences considérées sont respectivement de longueur S^2, S^3 et S^4. Nous observons que la différence entre la génération aléatoire et la génération génétique devient plus marquée lorsque la taille des sous-séquences grandit. L'efficacité de l'algorithme génétique est plus grande comparé à la génération aléatoire quand la taille de la sous-séquence est longue. Cet effet est probablement dû à la croissance du nombre de sous-séquence différentes à explorer lorsque la taille de la sous-séquence grandit. Dans un petit espace à explorer, la génération aléatoire est presque aussi bonne qu'un algorithme génétique. Dans un espace à explorer plus grand, les algorithmes génétiques ont la possibilité de réutiliser des résultats passés intéressants. Cela est encourageant dans l'optique de concevoir des objectifs plus complexes pour l'algorithme génétique comprenant des espaces plus vastes à explorer.

La Figure 6.7 indique que plus de la moitié des génomes générés ne produisent pas une trace d'exécution soit parce qu'aucune requête SQL n'a été produite à partir de ce génome ou

bien parce qu'aucune trace d'exécution de l'optimiseur n'a été produite pendant l'exécution de la requête. De plus, la réduction semble être un peu plus grande pour la génération aléatoire que pour l'algorithme génétique. Précisément pour l'algorithme génétique 58% des génomes (169 des 291 générés) sont productifs dans le sens qu'ultimement une trace d'exécution sera produite pour ce génome. Pour la génération aléatoire, la productivité est de 45% (134 sur les 298 générés) soit une différence de 13% avec l'algorithme génétique. Ceci suggère que l'algorithme est légèrement meilleur pour produire des génomes qui vont produire une trace d'exécution. Cependant la significativité statistique de cette observation doit être validée. Dans tous les cas, la séparation entre les résultats reste valable.

6.5 Conclusion

Nous avons proposé une approche génétique pour générer des requêtes SQL qui vont tester des sous-séquences de règles d'optimisations particulières. Les résultats de l'expérimentation montrent que les différences entre l'approche proposée et la génération aléatoire sont plus intéressants lorsque la taille des sous-séquences grandit de un à quatre. L'élimination des requêtes qui n'activent pas la production de traces d'exécution de l'optimiseur préserve la différence entre les deux approches. En conclusion, l'approche génétique proposée semble être meilleure pour concentrer la génération des tests vers des séquences plus longues et qui sont plus pertinentes.

CHAPITRE 7

EXTRACTION D'UN MODÈLE INTER-PROCÉDURAL DE RÔLES ET DE PRIVILÈGES À PARTIR D'UN CODE SOURCE EN PHP

Les applications web peuvent être soumises à des violations des privilèges des rôles causées par des vulnérabilités dans le code source. Ce chapitre présente un algorithme original pour extraire un modèle simple des privilèges de rôles booléens à partir d'une perspective inter-procédural du code source en PHP.

Les modèles extraits peuvent être vérifiés en fonction des violations des privilèges de rôles à l'aide d'un model checker. L'approche proposée d'extraction a été évaluée sur phpBB qui est un logiciel open source qui implémente un système de babillard électronique. Les propriétés de privilèges de rôles ont étés vérifiés sur les modèles extraits.

Des modèles de sécurité simple et booléens peuvent être extraits et vérifiés en temps linéaire en utilisant les algorithmes présentés. Les approches générales pour le modèle checking démontrent une complexité plus élevée à cause de la complexité computationnelle dû à leurs généralités. Les résultats ont été comparés, avec succès, avec ceux déjà obtenus par une analyse de flux détectant des vulnérabilités similaire.

La performance de l'exécution et les résultats par model checking incluant l'extraction du modèle proposé et le processus de validation du modèle sont présentés et discutés dans ce chapitre.

7.1 Introduction

Plusieurs applications web sont utilisées pour distribuer de l'information d'une organisation vers des usagers à l'aide d'un réseau informatique. Les applications web peuvent souffrir de violations des privilèges des rôles à cause de vulnérabilités dans le code source. Les applications qui acceptent des interactions venant des usagers prennent souvent pour acquis des entrées de données légitimes avec des privilèges légitimes venant des usagers.

Dans certains contextes spécifiques, des données construites d'une manière malicieuse ou une séquence d'actions imprévues peuvent permettre à un usager d'accéder à des opérations privilégiés qui n'étaient destinés qu'à être utilisés par les administrateurs du système. Pouvant ainsi menacer les politiques de sécurisation des données traités par l'application web.

1. Une version courte de ce chapitre intulée « Extraction of Inter-procedural Simple Role Privilege Models from PHP Code [8] » a été présentée lors de la conférence *15th Working Conference on Reverse Engineering* (WCRE). Certains éléments ont aussi été présentés dans [4, 5]

Souvent les attaques reposent sur des validations insuffisantes des données en entrée et sur la sécurisation insuffisante de l'exécution conditionnelle de certains chemins dans le code source donnant accès à des fonctionnalités privilégiées. Aussi même sans tenir compte de la validation des privilèges, des attaquants internes peuvent introduire dans une application web du code malicieux qui lorsqu'il sera activé, par une séquence d'entrée particulière par exemple, pourra violer l'intention des concepteurs du système en matière de sécurisation et d'accès.

Des références aux vulnérabilités des applications web, des attaques internes et du code malicieux peuvent être trouvées dans [65, 66, 67, 68, 69, 70].

Dans le chapitre 4 ainsi que dans [6], le problème d'identifier des énoncés vulnérables aux SQL-injections causé par des saisies de données externes ou par de la programmation interne est traité en adoptant la perspective de l'analyse de flux des *niveaux d'autorisations* et des *vulnérabilités*.

Dans ce chapitre, une approche originale utilisant les méthodes formelles et le model checking est utilisée pour traiter le même problème déjà présenté dans [6].

L'objectif de ce chapitre est de présenter une approche formelle pour l'extraction d'un modèle de sécurité simple à partir d'un code source PHP pour compléter l'approche précédemment présentée qui résout ce problème en utilisant l'analyse de flux.

La motivation pour ce chapitre vient de la difficulté « relativement plus grande » de prouver ou de raisonner formellement à propos de l'exactitude, de la complexité ou de la performance d'une analyse de flux comparativement à la difficulté de faire ces mêmes raisonnements en utilisant le model checking. Le succès obtenu dans la représentation formelle et la vérification formelle renforce l'approche de l'analyse des privilèges des rôles qu'elle soit résolue par l'analyse de flux ou par le modèle checking.

L'objectif est de concevoir une correspondance entre la description algorithmique de la méthode de résolution présentée dans [6] et une formulation en logique temporelle du même problème puis de faire une comparaison expérimentale des résultats en l'appliquant sur phpBB en tant qu'étude de cas.

Les contributions de ce chapitre en adoptant la perspective du model checking décrite pour extraire un modèle inter-procédural de rôles et de privilèges sont :

- la validation de l'analyse de flux en utilisant une perspective scientifique différente
- une meilleure compréhension des propriétés logiques de l'analyse de flux dans [6]
- l'extension de la puissance de l'analyse de flux dans [6] à des requêtes arbitraires exprimées en logique temporelle
- l'acquisition d'expérience de modélisation de logiciels en utilisant la logique temporelle

et en traitant de problèmes propre au model checking comme l'explosion du nombre d'états et les problèmes de performance.

Des modèles de sécurité simples peuvent être extrait et vérifiés en temps linéaire en utilisant les algorithmes présentés tandis que les approches générales pour le model checking inter-procédural démontrent une complexité plus élevée due à la généralité de ces approches. Les résultats ont été comparés, avec succès, avec ceux obtenus précédemment par l'analyse correspondante des vulnérabilités faites par analyse de flux inter-procédurale.

Le modèle de sécurité présenté est un modèle booléen simple (usager et administrateur). Il est possible d'envisager l'extension à des niveaux de sécurités entier dans des travaux futures et aussi à des modèles plus complexes comme les modèles multi-attributs (invité, utilisateur, modérateur, administrateur) ou ceux plus complexes requérant des modèles non-convexes.

L'approche présentée dans ce chapitre permet l'extraction de modèles de sécurité simple à partir d'un code source PHP puis la détection automatique des accès à une base de données dans des applications PHP qui peuvent être vulnérables à des violations des privilèges des rôles déclenchés soit par des entrées malicieuses (menaces externes) ou par du code malicieux (menaces internes).

Les énoncés vulnérables peuvent être définis comme des énoncés qui requièrent un niveau de sécurité administrateur pour leur exécution mais qui peuvent être rejointes par une exécution au niveau d'autorisation usager. La détection automatique des énoncés vulnérables peut être effectuée sur le modèle de sécurité extrait en vérifiant la cohérence entre les niveaux d'autorisations requis et les niveaux effectifs en utilisant les outils standard du model checking.

Dans ce chapitre, les expérimentations ont étés effectués sur les accès à une base de données comme un exemple d'opérations qui sont sujet des violations des privilèges des rôles. Cependant l'approche est générale et peut être appliquée à n'importe quel type d'énoncés requérant un niveau de sécurité particulier.

La section 7.2 introduit la représentation inter-procédurale du graphe de flux de contrôle (CFG) du code source et la représentation des privilèges des rôles. La section 7.3 présente les équations pour extraire le modèle de sécurité à partir du CFG. La section 7.4 présente un algorithme original pour effectuer l'extraction du modèle de sécurité. La section 7.5 présente l'expérimentation effectuée est les résultats. La section 7.6 présente une discussion de l'approche présentée et des résultats obtenus.

7.2 Graphe de flux de contrôle

Soit $CFG = (V_{CFG},\ E_{CFG})$ un graphe de flux de contrôle avec comme point d'entrée unique le nœud $v_{in} \in V_{CFG}$ et un point de sortie unique soit le nœud $v_{out} \in V_{CFG}$.

Les nœuds de V_{CFG} peuvent être de type *call_begin*, *call_end*, *entry*, *exit* ou *generic*. Les nœuds de type *generic* interviennent dans les flux intra-procéduraux tandis que les nœuds de type *call_begin*, *call_end*, *entry*, ou *exit* interviennent dans les flux inter-procéduraux.

Les arcs dans E_{CFG} peuvent être de type *generic*, *call*, *return*, *positive_authorization* ou *negative_authorization*. Les arcs de type *generic* représentent les transferts de contrôles intra-procéduraux qui ne changent pas le niveau de sécurité accordé. Les arcs de type *positive_authorization* représente des transferts de contrôles intra-procéduraux qui change le niveau de sécurité accordé à un niveau *admin*. Les arcs de type *negative_authorization* représentent des transferts de contrôles intra-procéduraux qui ramènent le niveau de sécurité au niveau *user*. Les arcs de type *call* et *return* représentent les flux de contrôles inter-procéduraux.

7.2.1 Aspects inter-procéduraux

Un arc de type *call* relie un nœud de type *call_begin* avec le nœud *entry* de la procédure appelée. Un arc de type *return* relie le nœud *exit* de la procédure appelée au nœud *call_end* approprié de la procédure appelante.

Supposons que la procédure f appelle la procédure g à un point d'appel i. La Figure 7.1 représente un tel point d'appel. Le point d'appel débute avec un nœud du type *call_begin*, dans la Figure 7.1 c'est le nœud v_4 qui correspond au début du point d'appel i dans le code source. Pour chaque point d'appel i dans la fonction appelante, il y deux nœuds correspondant respectivement au nœuds de type *call_begin* (v_1) et *call_end* (v_2). La fonction appelé contient un nœud *entry* au tout début ici v_3 et un noeud *exit* à la toute fin (v_4).

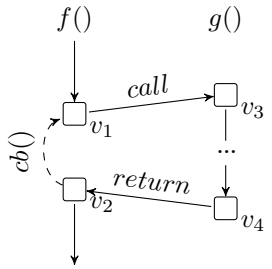

Figure 7.1 Représentation d'un point d'appel dans le code.

Une fonction $cb : V_{CFG} \leftrightarrow V_{CFG}$ est définie pour associer les nœuds *call_end* avec leurs

nœuds correspondants *call_begin*. Cette fonction est utilisée dans la section 7.4 pour extraire le modèle de sécurité de la représentation du CFG.

Un nœud de type *call_begin* au point d'appel i est connecté avec le nœud *entry* correspondant de la fonction g appelée avec un arc de type *call*. Le nœud *exit* de la fonction g est connecté avec le nœud de type *call_end* du point d'appel i avec un arc de type *return*.

Il découle que tous les nœuds de type *call_begin* qui appelle g à partir de points d'appels différents dans f sont connectés avec le même nœud *entry* dans g. D'une manière similaire, le nœud de type *exit* dans g est connecté à tous le nœuds de type *call_end* aux différents points d'appels dans f qui appellent g. Ce modèle inter-procédural fusionne les arcs de retours des différents points d'appels et, même s'il facilite le traitement, il introduit des chemins fallacieux. Par contre ces chemins fallacieux peuvent être contrôlés en utilisant des variables comme discuté dans les sections 7.2.4 et 7.6.

7.2.2 Exemple d'un graphe de flux de contrôle inter-procédural

Le problème de la représentation des graphes de flux de contrôles est abordé içi en utilisant un exemple concis. La Figure 7.2 présente un pseudo code composé d'une fonction $f()$ qui

```
function f()
 1: call g()
 2: if isAdmin then
 3:     call g()
 4:     print secure information
 5: end if
 6: return

function g()
 7: – empty function –
 8: return
```

Figure 7.2 Pseudo code avec appels inter-procéduraux.

fait deux appels à la fonction $g()$. Un des deux appels est fait d'une manière conditionnelle car il est situé à l'intérieur d'une condition *if then*. La Figure 7.3 présente un CFG simple qui correspond au pseudo code de la Figure 7.2. Les nœuds v_1 à v_7 situés à gauche de la figure correspondent à la fonction $f()$ aux lignes 1 to 6 du pseudo code. Les nœuds v_8 et v_9 à droite de la figure correspondent à la fonction $g()$. Notez que les numéros des lignes du pseudo code ne correspondent directement aux numéros des nœuds du CFG en partie parce deux nœuds sont nécessaires pour représenter le site d'un appel à une fonction (un nœud de type *call_begin* et un autre de type *call_end*).

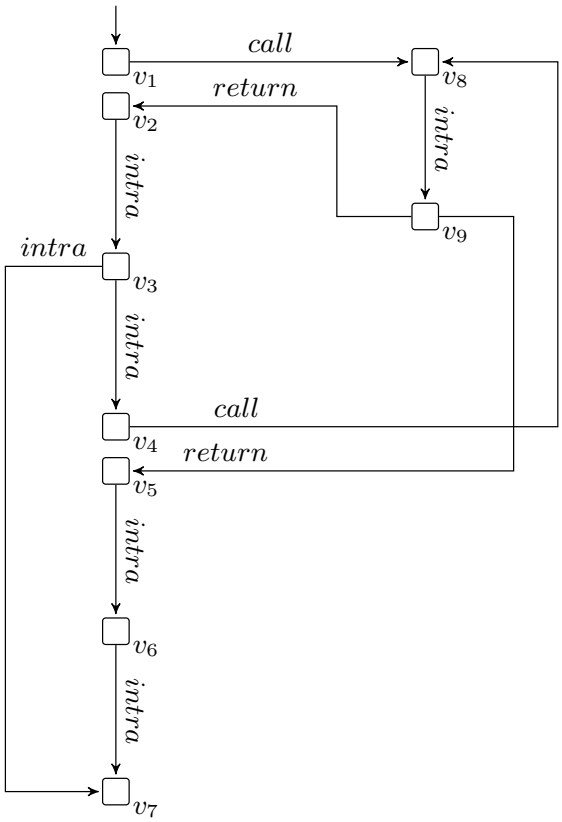

Figure 7.3 Représentation du CFG inter-procédural du pseudo code de la figure précédente. Les arcs inter-procéduraux sont identifiés *call* et *return*.

La difficulté majeure avec cette représentation simpliste des transferts inter-procéduraux est qu'il existe des chemins valides dans CFG qui ne correspondent pas à des traces d'exécutions possibles dans le pseudo code. Le chemin (path) $\pi_1 = \langle v_1, v_8, v_9, v_2, v_3, v_7 \rangle$ est un chemin valide dans le CFG qui correspond aussi à une exécution possible du pseudo code. Par contre le chemin $\pi_2 = \langle v_1, v_8, v_9, v_5, v_6, v_7 \rangle$ est un chemin valide dans le CFG mais ne correspond pas à une exécution possible dans le pseudo code. En effet, il ne devrait pas être possible d'emprunter l'arc (v_9, v_5) de type *return* que s'il on a auparavant emprunté l'arc *call* (v_4, v_8). Dans le cas de π_2 l'arc *call* est (v_1, v_8) et l'arc *return* est (v_9, v_5) ce qui ne devrait pas être permis. Une analyse de flux naïve basée sur cette représentation pourrait conclure que en utilisant π_2 il est possible de rejoindre l'instruction *print* dans la Figure 7.2 à la ligne 4 sans avoir effectué la vérification de sécurité à la ligne 2 ce qui n'est pourtant pas le cas.

Pour être plus représentative, cette représentation simpliste des transferts inter-procéduraux doit être augmenté d'un mécanisme pour traiter différemment les ars inter-procéduraux (*call* et *return*) des autres types d'arcs. Il ne devrait pas être possible d'emprunter un arc *return* si

l'arc *call* n'a pas été emprunté au bon moment auparavant. Plusieurs stratégies peuvent être utilisés pour résoudre ce problème. Souvent des graphes de flux de contrôle intra-procédural seulement sont construits et ils sont traités individuellement. Les graphes intra-procéduraux peuvent être reliés ensemble en utilisant un super-graphe comme décrit dans [92] utilisant des hyper-arêtes pour relier les CFG intra-procéduraux entre eux. Une autre possibilité utilisée dans [53, 93] est de construire des sommaires pour les arcs inter-procéduraux. Et aussi les automates à pile (pushdown automaton) peuvent être employés pour résoudre ce problème dans le domaine du model checking. Notre solution consiste à construire qu'un seul graphe simple qui représente tout le programme et d'utiliser le principe des assignations, des conditions et des variables tel qu'il est usuel en model checking pour s'assurer du respect de l'ordre des *call* et des *return*.

7.2.3 Patrons d'autorisations

Nous définissons les patrons d'autorisations comme un sous-ensemble des arcs dans le CFG tel que les arcs *positive_authorization* ou *negative_authorization* dans le patron concèdent un niveau d'autorisation spécifique aux traitements subséquent à de tels transitions.

Les patrons d'autorisations sont dépendant de l'application. Un exemple typique d'un patron concédant la sécurité dans phpBB est visible à la ligne 7 de la Figure 7.4. Le script pagestart.php s'arrête subrepticement (die) si le niveau d'autorisation de l'exécution est différent du niveau administrateur de système. La transition $(7,\ 8)$ garantie que l'exécution continue au niveau administrateur du système. La transition $(7,\ 8)$ est du type *positive_authorization*, tandis que la transition $(7,\ v_{out})$ est du type *negative_authorization*.

```
1  //
2  // Load default header
3  //
4  $no\_page_header = TRUE;
5  $phpbb_root\_path = "../";
6  require($phpbb_root_path . 'extension.inc');
7  require('pagestart.' . $phpEx);
8  //
9  // Do the job ...
10 //
```

Figure 7.4 Authorization Granting Pattern

Les accès à la base de données qui sont vulnérables peuvent être définis comme des énoncés qui requièrent un niveau de sécurité $sLev_i$ pour leur exécution mais qui peuvent être rejoint

par une exécution dont le niveau d'autorisation $aLev_j$ est plus petit que le niveau de sécurité requis $sLev_i$. Une discussion des niveaux de sécurité requis et effectifs est disponible dans [19, 20, 22, 24, 94]. Aussi, la section suivante 7.2.4 décrit avec plus de détails cette propriété de sécurité.

7.2.4 Le problème des privilèges des rôles

Supposons qu' un chemin dans le CFG soit $p = (v_0, v_1, ..., v_{p-1}, v_p),\ v_i \in V$ et qu'une simple valeur booléenne est utilisée pour représenter l'autorisation administrateur de système que cette autorisation assume un sens complémentaire relatif au niveau d'autorisation d'un usager.

Le niveau d'autorisation effectif de l'exécution dans p juste avant l'exécution de v_p est alors une valeur booléenne $gav(p)$ (granted authorization value), ce qui signifie que v_p possède ou non l'autorisation d'exécuter des opérations $admin$. Si ces opérations privilégiés sont exécutés immédiatement après v_{p-1} dans p. Un niveau d'autorisation $user$ sera représenté comme une valeur $false$ pour $gav(p)$.

De cette manière, le niveau d'autorisation effectif $gav(p)$ représente le niveau d'autorisation entrant dans v_p avant son exécution et après l'exécution de v_{p-1} dans p.

Le niveau d'autorisation effectif $gav(p)$ peut être calculé d'une manière récursive selon la taille de p comme décrit dans l'équation (7.1) où le type de l'arc (v_{p-1}, v_p) détermine $gav(v_p)$. Initialement, $\forall (v_0, v_j) \in E, gav(v_0, v_j) = false$ ce qui signifie que la valeur d'autorisation initiale concédée par le système d'exploitation pour n'importe quelle exécution est $false$ et que $gav(v_p)$ est déterminé selon :

$$gav(v_p) = \begin{cases} T & (positive\_authorization\ edge) \\ F & (negative\_authorization\ edge) \\ gav(v_0, ..., v_{p-1}) & (generic\ edge) \end{cases} \tag{7.1}$$

Le problème de sécurité peut être exprimé comme le problème d'évaluer les propriétés suivantes sur les nœuds du CFG :

$$\begin{gathered} \forall v_p \in V, \\ \forall p = (v_0, ..., v_p), gav(p) \geq required\_security(v_p) \end{gathered} \tag{7.2}$$

Les valeurs d'autorisations sont ordonnées en suivant un ordre partiel conventionnel entre des valeurs booléennes

$$\begin{gathered} false \leq false \\ false \leq true \\ true \leq true \end{gathered} \tag{7.3}$$

tel que les privilèges concédés à la valeur d'autorisation $false$ sont plus petits dans le domaine d'application que ceux qui sont concédés à la valeur $true$.

L'équation (7.2) peut être exprimée en logique temporelle comme montrée dans l'équation (7.4) où $granted\_authorization$ est la valeur courante de sécurité de l'automate de sécurité et $required\_security$ est une propriété de chaque nœud du CFG. À chacun des points de n'importe quelle exécution cette condition de sécurité devra être respectée.

$$\Box(granted\_authorization \geq required\_security) \tag{7.4}$$

En référence au patron d'autorisation de la Figure 7.4, le script pagestart.php arrête subrepticement l'exécution si l'exécution ne possède pas des privilèges élevés alors que la transition (7, 8) garantit que la poursuite de l'exécution possède des privilèges élevés.

7.3 Équations

Les contextes inter-procéduraux correspondent au nombre d'instances des procédures qui peuvent être rejointes en exécutant le programme sous analyse. Puisque ce nombre est habituellement infini, en pratique plusieurs stratégies de réduction des contextes ou de fusionnement des contextes doivent considérées. Souvent le processus de réduction du nombre de contextes introduit une distorsion conservative dans les résultats de l'analyse.

Une autre façon complémentaire de traiter le problème est de définir un ensemble réduit de contextes indistinguables du point de vue de l'analyse et d'évaluer la distorsion induite.

Le modèle formel de sécurité est un automate qui peut être décrit comme un ensemble d'état et un ensemble de transitions orientées joignant deux états. La construction de l'automate est décrite en deux étapes. La première étape génère un ensemble fini d'états en réécrivant l'ensemble des nœuds du CFG et la deuxième étape génère les transitions à partir de l'ensemble des arcs du CFG.

L'automate de sécurité peut être décrit par l'ensemble de propriétés élémentaires $Prop$ et le 8-tuple $\mathcal{A}$

$$Prop \ = \ \{P_1, \ P_2, \ ... \} \tag{7.5}$$

$$\mathcal{A} \ = \ (Q_{\mathcal{A}}, \ \Sigma_{\mathcal{A}}, \ T_{\mathcal{A}}, \ q_0, \ L_{\mathcal{A}}, \ V_{\mathcal{A}}, \ G_{\mathcal{A}}, \ A_{\mathcal{A}}) \tag{7.6}$$

où Q est un ensemble fini d'états ; Σ est un ensemble fini d'étiquettes de transitions ;
$T_{\mathcal{A}} \ \subseteq \ Q_{\mathcal{A}} \times \ \Sigma \times \ Q_{\mathcal{A}}$ est un ensemble de transitions ; q_0 est l'état initial ; $L_{\mathcal{A}}$ associe $L_{\mathcal{A}} : \ Q_{\mathcal{A}} \leftrightarrow \mathcal{P}(Prop)$ pour chaque état dans $Q_{\mathcal{A}}$ avec un sous-ensemble de propositions

atomiques qui sont vraies dans cet état ; $V_{\mathcal{A}}$ est l'ensemble des variables utilisé dans les « conditions » et les « assignations » ; $G_{\mathcal{A}}$ est un ensemble "conditions" qui sont des propositions logiques utilisant $V_{\mathcal{A}}$ et qui sont associés avec une transition ; et $A_{\mathcal{A}}$ est un ensemble d'assignations qui modifient la valeur d'une variable et qui sont aussi associées avec une transition.

Dans notre problème de sécurité, l'ensemble des étiquette de transitions Σ est vide, alors $T_{\mathcal{A}}$ est simplement $T_{\mathcal{A}} \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{A}}$, l'ensemble des propriétés élémentaire est $Prop = \{true,\ false\}$ qui représente la vulnérabilité d'un état correspondant à une exécution concédant un niveau de sécurité plus petit que celui requis pour cet état. Pour notre problème ceci correspond à des accès à la base de données à partir de fichiers sources situés dans le répertoire '/admin' et qui peuvent être possiblement exécutés avec un niveau de sécurité *user*. Alors, puisque l'ensemble des étiquettes de transitions est vide et que nous choisissons d'identifier l'état initial par $q_{0,0,0}$ selon notre nomenclature des nœuds , l'automate de sécurité peut être considéré simplement comme

$$\mathcal{A} = (Q_{\mathcal{A}},\ T_{\mathcal{A}},\ q_{0,0,0},\ L_{\mathcal{A}},\ V_{\mathcal{A}},\ G_{\mathcal{A}},\ A_{\mathcal{A}}) \tag{7.7}$$

7.3.1 Nœuds

Pour chaque nœud du CFG dans notre modèle, il y quatre états d'exécution possibles : un nœud du CFG peut être rejoint par un chemin inter-procédural dont la dernier appel de fonction était fait à partir d'un nœud au niveau de sécurité *user* (0) ou bien au niveau *admin* (1). Cette valeur est appelée le « niveau de sécurité du contexte d'appel ». En plus, au niveau intra-procédural, un nœud du CFG peut passer à un niveau de sécurité différent de celui du niveau inter-procédural selon la topologie intra-procédurale et la disposition des patrons concédant la sécurité dans le CFG. Les états de l'automate sont donc identifiés par un n-tuple de trois nombres dont un fait référence à un nœud du CFG et les deux autres font respectivement référence au contexte de sécurité « inter » et « intra » procédural.

Pour chaque nœud v_x du CFG découle donc quatre état possible dans l'automate de sécurité il sont identifiés $q_{x,0,0}$ pour l'état correspondant au nœud v_x avec le niveau de sécurité du contexte d'appel *user* et le niveau de sécurité intra-procédural *user*, $q_{x,0,1}$ pour le niveau de sécurité du contexte d'appel *user* et le niveau de sécurité intra-procédural *admin*, $q_{x,1,0}$ pour le niveau de sécurité du contexte d'appel *admin* et le niveau de sécurité intra-procédural *user* et $q_{x,1,1}$ où les niveaux de sécurités inter et intra-procédural sont *admin*.

Le modèle formel de sécurité est un automate où chacun des états $q_{i,j,k}$ est identifié par un 3-tuple i, j, k où i est le nœud correspond à un nœud du CFG ; j est le niveau de sécurité 0 ou 1 du contexte d'appel qui est le niveau de sécurité effectif du nœud du CFG qui a

effectué l'appel de la procédure en cours (ou le niveau de sécurité délégué par le système d'exploitation si la procédure en cours est le point d'entré principal du programme) ; k est le niveau de sécurité concédé intra-procédural 0 ou 1. Comme indiqué dans la sous-section 7.3.1 pour chaque nœud v dans le CFG quatre états q sont générés dans l'automate de sécurité.

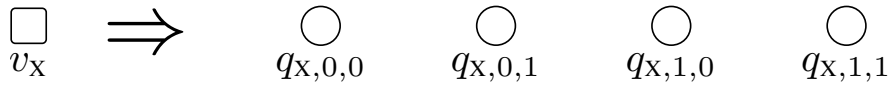

Figure 7.5 Réécriture des nœuds .

7.3.2 Arcs intra-procéduraux

Les transitions dans l'automate sont générées à partir de l'ensemble des arcs du CFG. Les arcs du CFG sont traités un par un et la règle de réécriture appropriée est appliquée pour ajouter les transitions aux états déjà existants de l'automate. La règle appropriée est choisie selon le type de l'arc dans le CFG.

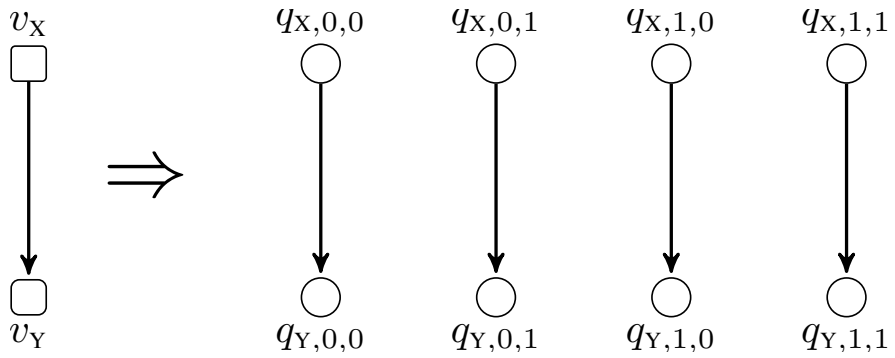

Figure 7.6 Réécriture des arcs, cas général intra-procédural.

La Figure 7.6 présente les transitions dans l'automate de sécurité qui correspondent à un arc intra-procédural du CFG. Ces arcs débutent d'un nœud de type *intra*, *entry*, ou *callreturn*. Notons qu'un arc débutant d'un nœud *callreturn* est situé après le retour d'appel et ne participe pas à l'appel inter-procédural et ne peux donc pas changer le niveau de sécurité du contexte d'appel. Dans tous ces cas, la transition est générique et elle ne change en aucune façon le niveau de sécurité intra-procédural ni le niveau de sécurité du contexte d'appel. Ces transitions sont alors directes et explicites comme montrés dans la Figure 7.6.

Les arcs qui effectuent les changements des niveaux de sécurité qui sont dus aux patrons de concessions de la sécurité changent le niveau de sécurité intra-procédural. La Figure 7.7

montre l'arc $(v_{\text{X}},\ v_{\text{Y}}) \in E_{CFG}$ de type *positive_authorization* qui correspond à un patron concédant la sécurité ainsi que sa réécriture en quatre transitions $(q_{\text{X},i,j},\ q_{\text{Y},m,n}) \in T_{\mathcal{A}}$ de l'automate de sécurité. L'arc $(v_{\text{X}},\ v_{\text{Y}})$ ne participe pas à un appel inter-procédural alors le contexte d'appel ne peux pas être changé c'est à dire $i = m$ dans les transitions générés. Cependant puisque sans tenir compte du niveau de sécurité intra-procédural au départ, à l'arrivée le niveau intra-procédural sera toujours 1 à cause du type de la transition *positive_authorization*. Alors $n = 1$ pour tous les états d'arrivés de ces transitions.

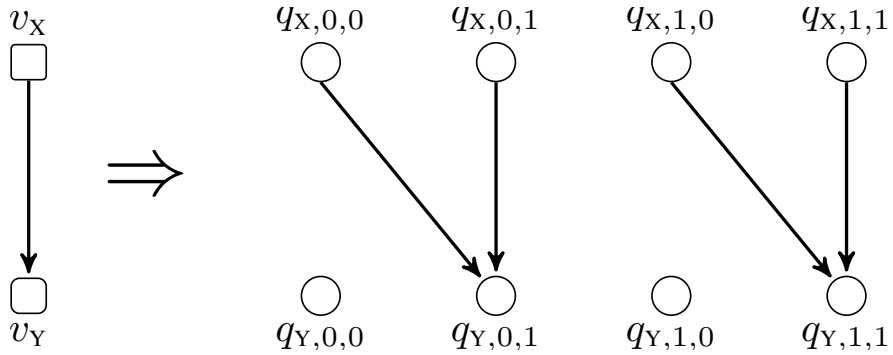

Figure 7.7 Réécriture d'un arc concédant la sécurité.

La figure 7.8 montre l'arc $(v_{\text{X}},\ v_{\text{Y}}) \in E_{CFG}$ correspondant à un patron révoquant la sécurité dans le CFG ainsi que sa réécriture dans les transitions $(q_{\text{X},i,j},\ q_{\text{Y},m,n}) \in T_{\mathcal{A}}$. Comme pour le cas *positive_authorization* dans la Figure 7.7 le niveau de sécurité du contexte d'appel ne change pas donc $i = m$ dans les transitions générés mais désormais le niveau de sécurité intra-procédural sera toujours *user* donc $n = 0$ pour tous états terminant ces transitions.

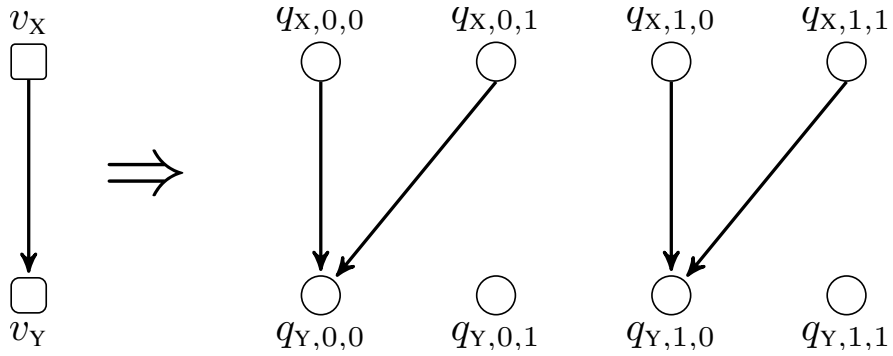

Figure 7.8 Réécriture d'un arc révoquant la sécurité.

7.3.3 Réécriture des arcs d'appel inter-procéduraux

Les arcs inter-procéduraux du CFG sont un peu plus difficiles à réécrire. La Figure 7.9 montre la règle de réécriture pour l'arc $(v_{\mathrm{X}},\ v_{\mathrm{Y}}) \in E_{CFG}$ de type *call*. Le nœud v_{X} est de type *call_begin* et le nœud v_{Y} de type *entry* représente le point dans la procédure appelée où le contrôle est transféré.

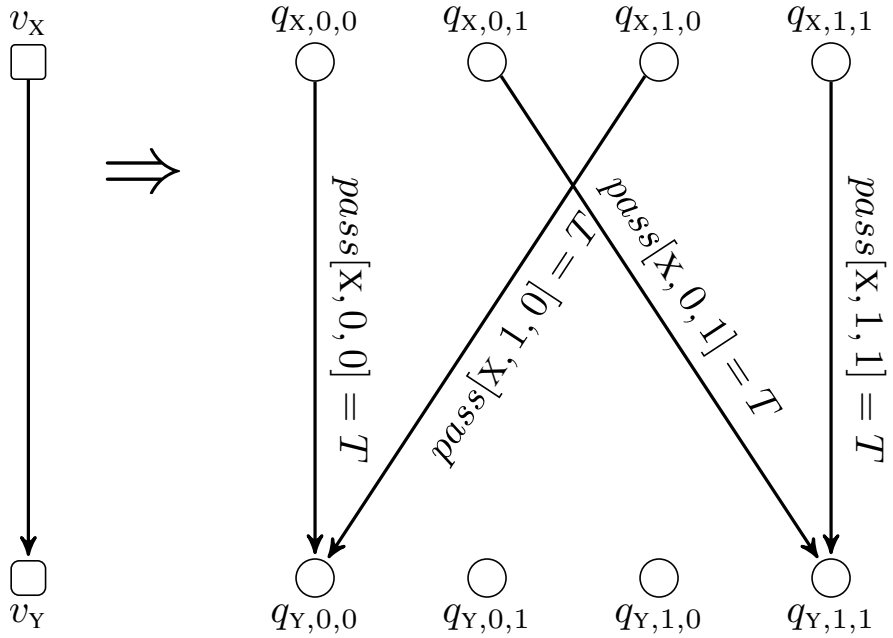

Figure 7.9 Réécriture d'un arc d'appel inter-procédural.

Le mécanisme d'appels et de retours inter-procéduraux ne change pas le niveau de sécurité intra-procédural alors $j = n$ dans les transitions générés $(q_{\mathrm{X},i,j},\ q_{\mathrm{Y},m,n}) \in T_{\mathcal{A}}$.

Par contre, des changements peuvent survenir pour le niveau de sécurité du contexte d'appel. Le nouveau niveau de sécurité du contexte d'appel reflétera le mécanisme des appels inter-procéduraux, donc il sera égal au niveau de sécurité intra-procédural de la fonction appelante c'est à dire $j = m$ pour toutes les transitions générées. Puisque $j = n$ et $j = m$ il en découle que $m = n$ pour les transitions générés ainsi seulement deux états cibles seront possibles $q_{\mathrm{X},0,0}$ pour $m = n = 0$ et $q_{\mathrm{X},1,1}$ pour $m = n = 1$.

En plus pour chacune des transitions générées une affectation à un variable dans $V_{\mathcal{A}}$ est effectuée. Il existe quatre variables pour chacun des nœuds du CFG de type *call_begin* et elles sont identifiées par un index 3-tuple de la même manière que les états de l'automate de sécurité. Ce tableau de variables booléenne est nommée *pass*. Il mémorise qu'un chemin d'exécution en cours d'évaluation est déjà passé par ce *call_begin* avec un contexte de sécurité inter-procédual et intra-procédural spécifique.

La Figure 7.10 présente la règle de réécriture associée à un arc $(v_{\mathrm{X}},\ v_{\mathrm{Y}}) \in E_{CFG}$ type *call_return*. Le nœud $v_{\mathrm{X}} \in V_{CFG}$ est de type *exit* et le nœud $v_{\mathrm{Y}} \in V_{CFG}$ est de type *call_end*. Le nœud v_{Y} représente le nœud du CFG qui reçoit le retour d'appel à la fin de l'exécution d'une procédure. Comme dans le cas de lar réécriture de l'arc *call* le mécanisme de retour ne change pas le niveau de sécurité intra-procédural alors $j = n$ dans le transitions générées $(q_{\mathrm{X},i,j},\ q_{\mathrm{Y},m,n}) \in T_{\mathcal{A}}$.

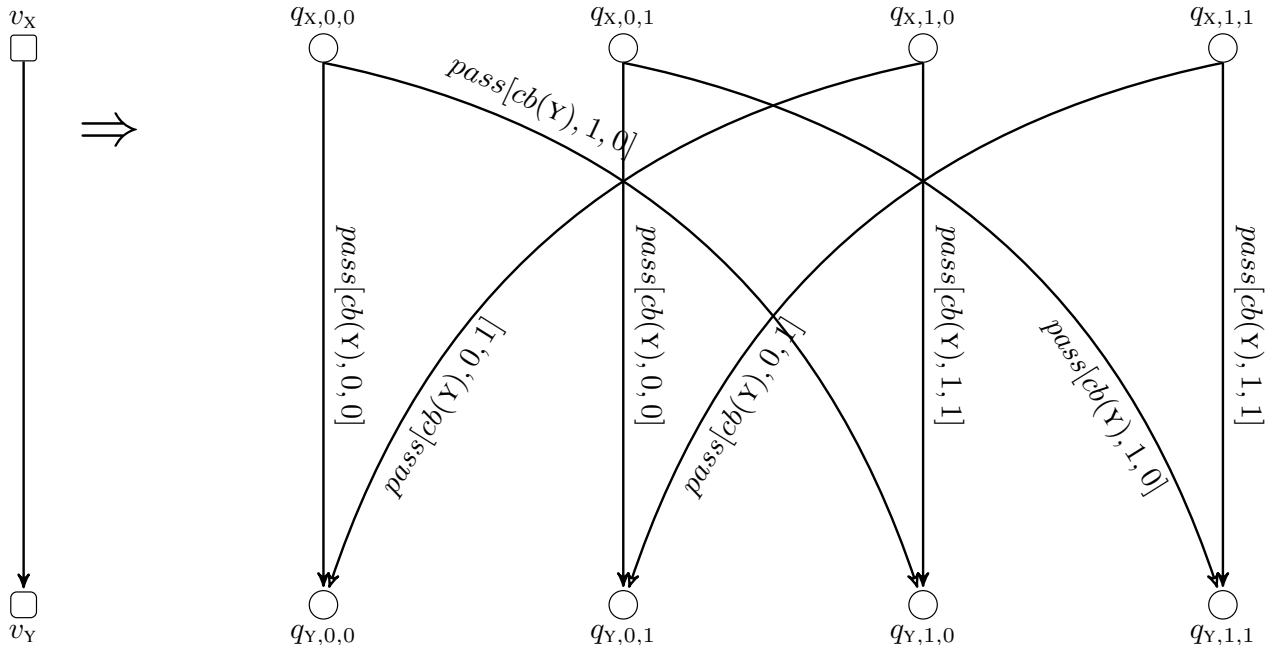

Figure 7.10 Réécriture d'un arc de retour inter-procédural.

L'état $q_{\mathrm{X},i,j}$ de l'automate de sécurité correspondant à un nœud de type *exit* du CFG a été rejoint par une exécution qui a appelé la procédure courante en cour de terminaison (exiting) à partir d'un nœud de type *entry* dont le niveau de sécurité inter et intra-procédural était égal à i. L'état $q_{\mathrm{X},i,j}$ va aussi propager son propre niveau de sécurité intra-procédural j correspondant à un éventuel changement de niveau de sécurité à l'intérieur de la procédure appelée. Le niveau de sécurité j doit être propagé à tous les états de l'automate correspondant à des nœuds du CFG associés au *call_end* terminant le *call_begin* avec le niveau de sécurité du contexte d'appel i. Ainsi, $q_{\mathrm{X},i,j}$ est connectée à $q_{\mathrm{Y},0,j}$ et à $q_{\mathrm{Y},1,j}$ créant les transitions $(q_{\mathrm{X},i,j},\ q_{\mathrm{Y},0,n})$ et $(q_{\mathrm{X},i,j},\ q_{\mathrm{Y},1,n})$ comme montré dans la Figure 7.10.

Un nœud *call_end* du CFG ne devrait être rejoint que si le nœud *call_begin* associé a été rejoint auparavant. Quand un état $q_{\mathrm{X},i,j}$ de l'automate de sécurité correspondant à un *call_end* du CFG est rejoint, une condition $L_{\mathcal{A}}$ s'assure de faire respecter cette contrainte en utilisant une variable du tableau *pass*.

Les assignations et les conditions impliquant les variables *pass* sont nécessaires pour conserver la précision du modèle. Si elles n'étaient présentes des faux positifs pourraient être induits. Supposons par exemple dans le cas où une fonction commençant à *entry* et se terminant à *exit* et qui ne contient aucun changement de niveau de sécurité intra-procédural. Cette fonction est appelée par deux points d'appels différents l'un des *call_begin* étant au niveau de sécurité *user* et l'autre *call_begin* au niveau de sécurité *admin*. Dans le cas de l'automate sans l'usage des variables il ne pourra pas distinguer quels niveaux de sécurité il doit propager au *call_end*, s'il les propageaient tous, il introduirait un faux positif si le reste de l'exécution après le *call_end* contient un accès à la base de données. L'utilisation des variables *pass* permet de contrôler correctement les retours à *call_end* en conservant la précision du modèle.

L'approche proposée modélise le calcul des sommaires inter-procéduraux des niveaux de sécurités en respectant l'information du contexte d'appel de l'appelant et de l'appelé et des patrons concédant la sécurité. Les sommaires peuvent être calculés parce que le mécanisme d'appel inter-procédural *call-return* n'affecte pas les changements du niveau de sécurité. Les valeurs du niveau de sécurité et niveau de sécurité du contexte d'appel d'un état *entry* peuvent être déterminés seulement à partir du nœud *call_begin* de l'appelant, similairement les valeurs du niveau de sécurité et niveau de sécurité du contexte d'appel d'un état *call_end* peuvent être déterminés seulement à partir du nœud *exit* de l'appelé.

La Figure A.3 présente un exemple d'un automate de sécurité pour un petit fragment de code. Étant donné la taille de la figure générée la présentation est situé dans l'Annexe A.

7.4 Algorithmes

La Figure 7.11 présente l'algorithme utilisée pour extraire le modèle de sécurité à partir d'un CFG qui représente le code source en PHP. Initialement, les ensemble des nœuds $Q_{\mathcal{A}}$, des transitions $T_{\mathcal{A}}$, des variables $V_{\mathcal{A}}$, des conditions $G_{\mathcal{A}}$ et des actions $A_{\mathcal{A}}$ sont vides (ligne 1). Tous les nœuds dans le CFG sont visités et comme décrit dans la Figure 7.11 quatre états de l'automate sont créés (ligne 2-3) pour chacun des nœuds du CFG.

Les lignes 7 à 18 et 27 à 30 ajoute les transitions appropriés à $T_{\mathcal{A}}$ conformément au type de l'arc du CFG comme indiqué dans les figures 7.6, 7.7, 7.8, 7.9 et 7.10. Les lignes 21 à 25 créent des actions dans $A_{\mathcal{A}}$ qui assignent la valeur T (true) aux variables *pass* et cette action est associée à une transition de l'automate comme indiqué dans la Figure 7.9. Les lignes 31 à 39 génèrent de nouvelles conditions (guards) dans $G_{\mathcal{A}}$, ces conditions testent des valeurs dans le tableau de variables *pass*, ces condtitions sont aussi associés avec une transition comme indiquée dans la figure Figure 7.10.

function $modelExtract(V_{CFG},\ E_{CFG})$

1: $Q_{\mathcal{A}} = T_{\mathcal{A}} = V_{\mathcal{A}} = G_{\mathcal{A}} = A_{\mathcal{A}} = \emptyset$
2: **for all** $\mathrm{x} \mid v_{\mathrm{x}} \in V_{CFG}$ **do**
3: $\quad Q_{\mathcal{A}} \cup= \{q_{\mathrm{x},0,0},\ q_{\mathrm{x},0,1},\ q_{\mathrm{x},1,0},\ q_{\mathrm{x},1,1}\}$
4: **end for**
5: **for all** $\mathrm{x}, \mathrm{y} \mid (v_{\mathrm{x}}, v_{\mathrm{y}}) \in E_{CFG}$ **do**
6: $\quad type = edge\_type(v_{\mathrm{x}}, v_{\mathrm{y}})$
7: $\quad$ **if** $type == generic$ **then**
8: $\quad\quad T_{\mathcal{A}} \cup= \{(q_{\mathrm{x},0,0},\ q_{\mathrm{y},0,0}),\ (q_{\mathrm{x},0,1},\ q_{\mathrm{y},0,1}),$
9: $\quad\quad\quad (q_{\mathrm{x},1,0},\ q_{\mathrm{y},1,0}),\ (q_{\mathrm{x},1,1},\ q_{\mathrm{y},1,1})\}$
10: $\quad$ **else if** $type == positive\_authorization$ **then**
11: $\quad\quad T_{\mathcal{A}} \cup= \{(q_{\mathrm{x},0,0},\ q_{\mathrm{y},0,1}),\ (q_{\mathrm{x},0,1},\ q_{\mathrm{y},0,1}),$
12: $\quad\quad (q_{\mathrm{x},1,0},\ q_{\mathrm{y},1,1}),\ (q_{\mathrm{x},1,1},\ q_{\mathrm{y},1,1})\}$
13: $\quad$ **else if** $type == negative\_authorization$ **then**
14: $\quad\quad T_{\mathcal{A}} \cup= \{(q_{\mathrm{x},0,0},\ q_{\mathrm{y},0,0}),\ (q_{\mathrm{x},0,1},\ q_{\mathrm{y},0,0}),$
15: $\quad\quad (q_{\mathrm{x},1,0},\ q_{\mathrm{y},1,0}),\ (q_{\mathrm{x},1,1},\ q_{\mathrm{y},1,0})\}$
16: $\quad$ **else if** $type == call$ **then**
17: $\quad\quad T_{\mathcal{A}} \cup= \{(q_{\mathrm{x},0,0},\ q_{\mathrm{y},0,0}),\ (q_{\mathrm{x},0,1},\ q_{\mathrm{y},1,1}),$
18: $\quad\quad (q_{\mathrm{x},1,0},\ q_{\mathrm{y},0,0}),\ (q_{\mathrm{x},1,1},\ q_{\mathrm{y},1,1})\}$
19: $\quad\quad V_{\mathcal{A}} \cup= \{pass[\ \mathrm{x}, 0, 0\ ],\ pass[\ \mathrm{x}, 0, 1\ ],$
20: $\quad\quad\quad pass[\ \mathrm{x}, 1, 0\ ],\ pass[\ \mathrm{x}, 1, 1\ ]\}$
21: $\quad\quad A_{\mathcal{A}} \cup= \{$
22: $\quad\quad \{(q_{\mathrm{x},0,0},\ q_{\mathrm{y},0,0}) \mapsto pass[\ \mathrm{x}, 0, 0\ ] = T\},$
23: $\quad\quad \{(q_{\mathrm{x},0,1},\ q_{\mathrm{y},1,1}) \mapsto pass[\ \mathrm{x}, 0, 1\ ] = T\},$
24: $\quad\quad \{(q_{\mathrm{x},1,0},\ q_{\mathrm{y},0,0}) \mapsto pass[\ \mathrm{x}, 1, 0\ ] = T\},$
25: $\quad\quad \{(q_{\mathrm{x},1,1},\ q_{\mathrm{y},1,1}) \mapsto pass[\ \mathrm{x}, 1, 1\ ] = T\}\ \}$
26: $\quad$ **else if** $type == return$ **then**
27: $\quad\quad T_{\mathcal{A}} \cup= \{(q_{\mathrm{x},0,0},\ q_{\mathrm{y},0,0}),\ (q_{\mathrm{x},0,0},\ q_{\mathrm{y},1,0}),$
28: $\quad\quad (q_{\mathrm{x},0,1},\ q_{\mathrm{y},0,1}),\ (q_{\mathrm{x},0,1},\ q_{\mathrm{y},1,1}),$
29: $\quad\quad (q_{\mathrm{x},1,0},\ q_{\mathrm{y},1,0}),\ (q_{\mathrm{x},1,0},\ q_{\mathrm{y},0,0}),$
30: $\quad\quad (q_{\mathrm{x},1,1},\ q_{\mathrm{y},1,1}),\ (q_{\mathrm{x},1,1},\ q_{\mathrm{y},0,1})\}$
31: $\quad\quad G_{\mathcal{A}} \cup= \{$
32: $\quad\quad \{(q_{\mathrm{x},0,0},\ q_{\mathrm{y},0,0}) \mapsto pass[\ cb(\mathrm{y}), 0, 0\ ]\},$
33: $\quad\quad \{(q_{\mathrm{x},0,0},\ q_{\mathrm{y},1,0}) \mapsto pass[\ cb(\mathrm{y}), 1, 0\ ]\},$
34: $\quad\quad \{(q_{\mathrm{x},0,1},\ q_{\mathrm{y},0,1}) \mapsto pass[\ cb(\mathrm{y}), 0, 0\ ]\},$
35: $\quad\quad \{(q_{\mathrm{x},0,1},\ q_{\mathrm{y},1,1}) \mapsto pass[\ cb(\mathrm{y}), 1, 0\ ]\},$
36: $\quad\quad \{(q_{\mathrm{x},1,0},\ q_{\mathrm{y},1,0}) \mapsto pass[\ cb(\mathrm{y}), 1, 1\ ]\},$
37: $\quad\quad \{(q_{\mathrm{x},1,0},\ q_{\mathrm{y},0,0}) \mapsto pass[\ cb(\mathrm{y}), 0, 1\ ]\},$
38: $\quad\quad \{(q_{\mathrm{x},1,1},\ q_{\mathrm{y},1,1}) \mapsto pass[\ cb(\mathrm{y}), 1, 1\ ]\},$
39: $\quad\quad \{(q_{\mathrm{x},1,1},\ q_{\mathrm{y},0,1}) \mapsto pass[\ cb(\mathrm{y}), 0, 1\ ]\}\ \}$
40: $\quad$ **end if**
41: **end for**
42: **return** $Q_{\mathcal{A}},\ T_{\mathcal{A}},\ V_{\mathcal{A}},\ G_{\mathcal{A}},\ A_{\mathcal{A}}$

Figure 7.11 Algorithme pour extraire le model formel de sécurité à partir du CFG

Pour chacun des nœuds du CFG quatre états sont ajoutés dans l'automate et pour chacun des arcs dans le CFG quatre ou huit transitions sont ajoutées dans l'automate de sécurité. La complexité de l'algorithme de la Figure 7.11 est $\mathcal{O}(|V_{CFG}| + |E_{CFG}|)$. Si la cardinalité de V_{CFG} est similaire à la cardinalité de E_{CFG}comme il est souvent normal, la complexité de l'algorithme de la Figure 7.11 peut être considéré de $\mathcal{O}(|V_{CFG}|)$.

La Figure 7.12 présente l'algorithme pour vérifier le modèle de sécurité extrait. L'algorithme utilise la fonction $cb()$ associant les nœuds *call_end* avec leurs nœuds *call_begin* correspondants tel que décrit dans la section 7.2.

La complexité de l'algorithme de la Figure 7.12 est $\mathcal{O}(|Q_{\mathcal{A}}| + |T_{\mathcal{A}}|)$. Comme $Q_{\mathcal{A}}$ est directement dérivé de V_{CFG} avec un facteur constant de 4 tel $|Q_{\mathcal{A}}| = 4|V_{CFG}|$ et $T_{\mathcal{A}}$ est directement dérivé de E_{CFG} avec un facteur entre 4 et 8 selon la répartition du type des arcs, on peut ainsi exprimer la complexité de l'algorithme 7.12 en fonction des nœuds du CFG par $\mathcal{O}(|V_{CFG}| + |E_{CFG}|)$ et ainsi de la même manière que pour la Figure 7.11 la complexité de de la Figure 7.12 est $\mathcal{O}(|V_{CFG}|)$.

7.5 Expérimentation et résultats

L'approche d'extraction et de validation proposée a été évaluée d'une manière préliminaire sur phpBB qui est une application PHP de taille moyenne implémentant un système de babillard électronique. Les résultats de l'exécution et les temps d'exécution de l'approche proposée sont présentés et commentés.

Pour évaluer notre approche, nous avons pris la perspective d'une étude de cas visant de la détection des vulnérabilités des privilèges des rôles. L'application phpBB [79] est bien connue pour l'abondance présence de violations de sécurité bien documentées dans ces versions antérieures. En effet, durant les dernières années et les dernières versions un effort concerté a été effectué pour réduire le nombre de vulnérabilités.

Pour l'expérience décrite, nous avons choisi la version de phpBB 2.0.0 pour laquelle plusieurs failles de sécurités ont été rapportées. Le Tableau 7.1 donne la taille de phpBB en terme de nombre de fichiers (file), de répertoire (path) et de lignes de code (LOC). Notre objectif est de prendre une ancienne version déficiente de phpBB et de détecter automatiquement les vulnérabilités qui peuvent être atteintes par des données externes venant de l'usager ou possiblement par du code introduit par un attaquant interne.

L'analyse syntaxique a été effectuée en utilisant une version étendue d'une grammaire PHP disponible sur le web [63] implémenté par Satyam en JavaCC [64]. La taille de la grammaire étendue est de 73 règles et 1221 lignes de codes.

Les fonctions externes incluant les bibliothèques de fonctions (library) ont étés approximés

Tableau 7.1 phpBB and CFG Features

Chemin	Fichiers	LOC
/	16	11 919
/admin	18	7 884
/contrib	2	1 057
/db	8	3 149
/includes	25	8 213
/language	4	1 879
Total	73	34 101

en assumant que les appels systèmes et les appels à des bibliothèques de fonctions propagent les niveaux de sécurité auxquelles ils ont été appelés. PHP permet les appels variables à des fonctions en permettant d'effectuer des appels à une fonction dont le nom est stocké dans une chaîne de caractères arbitraire. Cette technique joue un rôle similaire aux pointeurs de fonctions disponible dans certains autres langages. Pour le moment, une approche conservative est employée en relation avec l'approximation des appels de fonction variables en supposant qu'elles peuvent contenir le nom de n'importe laquelle fonction appelable dans le système incluant les appels externes. Toutes les fonctions rejoignable dans le système peuvent ainsi recevoir l'information de flux au niveau de sécurité de l'appel de fonction variable. En outre dans phpBB 2.0.0 les appels variables à des fonctions sont peu utilisés et on pu être résolus manuellement ainsi n'affectent pas les résultats présentés.

Une mise en œuvre a été effectuée pour évaluer les caractéristiques et la performance du modèle proposée et son extraction.

Le Tableau 7.2 présente les propriétés de l'automate de sécurité obtenue correspondant à phpBB 2.0.0 ainsi qu'à la performance de son extraction et de sa validation tel que décrit dans la section 7.2.4. Les algorithmes décrits dans les figures 7.11 et 7.12 sont utilisés.

Tableau 7.2 Sommaire des résultats.

États	Q	296 340
Transitions	T	406 304
États nécessitant d'être sécurisé	Q_{rs}	536
Variables		33 148
Temps de construction du modèle		27.9 s
Temps d'exécution du modèle		21.8 s
Temps total		49.7 s
Mémoire utilisée		10.2 MB
Violations du modèle	MV	15

Puisque comme énoncé dans la définition du problème, les accès à la base de données effectués à partir d'un script dans le répertoire '/admin' qui sont rejoignable avec le niveau de sécurité *user* sont considérés comme des vulnérabilités. Une analyse plus détaillée a été effectuée fichier par fichier pour les fichiers du répertoire '/admin'. Cette expérimentation a été effectuée en considérant chaque fichier du répertoire '/admin' comme une cible d'une requête web. Se faisant le nœud qui est le point d'entrée de chaque fichier est accédé au niveau de sécurité *user*. Même si cette méthode n'est pas l'usage usuel ou escompté de phpBB, quoi qu'il en soit elle est possible en utilisant un navigateur web et peut conduire à des vulnérabilités additionnelles.

Le Tableau 7.3 présente les des violations identifiées. Les résultats de l'identification des états protégés et vulnérables associés avec les accès à la base de données sont identifiés dans le Tableau 7.3, ils ont été comparés avec ceux obtenues par l'approche précise par analyse de flux dans [6]. Les résultats de la vérification par model checking identifie aussi les mêmes 15 vulnérabilités que l'analyse de flux correspondante.

7.6 Discussion

Une application PHP de taille moyenne a été analysée en utilisant l'approche formelle proposée. Les résultats de cette analyse ont été comparés avec ceux obtenus avec une analyse de flux antérieurement développée pour analyser le même problème de privilège des rôles.

L'objectif de ce chapitre est d'exprimer le problème de la détection des énoncés vulnérables en utilisant le paradigme du model checking pour vérifier les modèles extraits. Cette approche a facilité la réflexion et le raisonnement formel à propos des analyses inter-procédurales du code source. L'approche correspondante développée antérieurement en utilisant l'analyse de flux plutôt que le model checking était algorithmique et ces propriétés logiques étaient plus difficiles à évaluer.

Il y a une correspondance entre l'analyse de flux et le model checking [57]. Ce chapitre présente une simple expérience anecdotique à propos de l'étude d'un problème de vulnérabilité des énoncés résout par l'analyse statique de flux et aussi par la vérification formelle.

L'algorithme de la Figure 7.12 est nécessaire parce que les outils communs essayés pour la vérification du modèle n'étaient pas capables de résoudre le modèle de sécurité généré. Nous croyons que le nombre d'états et le nombre de variables *pass* fait exploser l'évaluation de ces modèles. Par exemple, le model checker Spin [95] a souvent été arrêté après plusieurs jours d'exécution sans avoir terminé l'évaluation du modèle.

Les vulnérabilités identifiées sont conforme avec celles produites par l'analyse de flux et

Tableau 7.3 Violations de sécurité

	Q_{rs}	MV
admin_board.php	8	0
admin_db_utilities.php	4	0
admin_disallow.php	12	0
admin_forum_prune.php	4	0
admin_forumauth.php	8	0
admin_forums.php	120	0
admin_groups.php	52	0
admin_mass_email.php	8	0
admin_ranks.php	24	0
admin_smilies.php	36	0
admin_styles.php	60	15
admin_ug_auth.php	72	0
admin_user_ban.php	36	0
admin_users.php	52	0
admin_words.php	16	0
index.php	24	0
page_footer_admin.php	0	0
page_header_admin.php	0	0
pagestart.php	0	0

sont causés par une erreur connue de la classe « Input Validation » publiée dans les archive de phpBB [81, 82].

L'extraction du modèle présentée est principalement indépendante du langage utilisé au moins pour la classe des langages impératifs. Tout langage avec une sémantique de style impérative pour lequel un parseur est disponible peut en principe être analysé par l'approche proposée. L'extension des expérimentations à d'autres langage par exemple Java est néanmoins intéressante et requise pour étendre les perspectives de l'approche d'extraction du modèle.

Les niveaux de sécurité dans l'application étudiée sont concédés par des patrons décrits dans la section 7.2.3. Les patrons concédant la sécurité sont dépendant de l'application et ne sont pas susceptibles d'être réutilisés par plusieurs applications. Néanmoins, lorsque l'analyse de l'évolution de la sécurité d'une application est voulue [10] par exemple pour valider des versions successives d'un système, les patrons concédant la sécurité démontrent une certaine stabilité d'une version à l'autre. L'analyse d'applications autre que phpBB requerrait le codage de nouveaux patrons spécifiques dans le parseur utilisé pour produire le CFG. Une étude plus poussée de systèmes plus grands et plus diversifiés est requise pour mieux évaluer les faux positifs, la performance de l'extraction du modèle de sécurité et la performance de la vérification.

L'approche d'extraction du modèle de sécurité est basée sur l'hypothèse que les niveaux de sécurités sont binaires comme les niveaux *user* et *admin*. D'autres arrangements des privilèges et de la sécurité sont basés des nombres entiers et parfois peuvent être attribués selon différentes dimensions des privilèges. Par exemple, différents niveaux d'autorisations peuvent être attribués pour installer l'application sur un serveur, pour changer les paramètres de configurations de l'application, pour certains opérateurs qui doivent réagir à des situations au moment de l'exécution de l'application ou bien par exemple aussi pour des usagers qui modifient des paramètres personnels. Ces privilèges peuvent être ou non strictement hiérarchique. Par exemple, les opérateurs peuvent changer les paramètres d'exécution mais pas les préférences des usagers. D'autre part, les usagers peuvent modifier leurs préférences mais ne peuvent pas modifier les paramètres d'exécution. Plus de travaux sont requis pour étendre l'approche de l'extraction du modèle selon cette perspective.

Un algorithme original pour extraire un modèle de sécurité des privilèges des rôles booléen à partir d'une perspective inter-procédurale du code source en PHP a été présenté. L'approche d'extraction proposée a été évaluée sur phpBB une application PHP de taille moyenne qui implémente un système de babillard électronique. Le modèle extrait a été vérifié pour détecter la présence de violations des privilèges des rôles en utilisant un nouvel algorithme décrit dans ce chapitre.

L'extraction et la validation du modèle de sécurité démontrent une consommation de mémoire raisonnable en pratique. Le temps requis pour l'analyse de phpBB est d'environ 50 s en tout incluant l'extraction du modèle à partir du CFG jusqu'à la vérification du modèle inclusivement.

Les vulnérabilités identifiées correspondent à celle déjà trouvées par une analyse de flux publiée antérieurement [6]. Les vulnérabilités *admin* sont causées par une erreur connue publiée dans les archives de phpBB [81, 82].

Plus de travaux sont requis pour étendre la base expérimentale à des systèmes plus diversifiés et de plus grande envergure afin de mieux évaluer la performance de l'approche. Aussi, l'approche devrait être étendue de manière à permettre la représentation de modèles de sécurités plus complexes.

```
function reach(Q_A, T_A, q_{0,0,0}, V_A)
 1: w = ∅
 2: for all i,j,k | q_{i,j,k} ∈ Q_A do
 3:    reachable[i,j,k] = false
 4:    pending[i,j,k] = ∅
 5: end for
 6: for all i,j,k | pass[ i,j,k ] ∈ V_A do
 7:    pass[i,j,k] = false
 8: end for
 9: w.push(q_{0,0,0})
10: while ¬ (w.empty()) do
11:    (i,j,k) = w.pop()
12:    if ¬ (reachable[i,j,k]) then
13:       reachable[i,j,k] = true
14:       for all x,y,z | (q_{i,j,k}, q_{x,y,z}) ∈ T_A do
15:          type = edge_type(q_{i,j,k}, q_{x,y,z})
16:          if type == call then
17:             if ¬ (pass[i,j,k]) then
18:                pass[i,j,k] = true
19:                w.append(pending[i,j,k])
20:             end if
21:             w.push(x,y,z)
22:          else if type == return then
23:             if pass[cb(x),y,z)] then
24:                w.push(x,y,z)
25:             else
26:                pending[cb(x),y,z)].append(x,y,z)
27:             end if
28:          else
29:             w.push(x,y,z)
30:          end if
31:       end for
32:    end if
33: end while
34: return reachable
```

Figure 7.12 Algorithme de joingnabilité du model

CHAPITRE 8

CONCLUSION

Les logiciels évoluent constamment tout au long de leur cycle de vie, soit à cause des besoins changeants de leurs utilisateurs ou soit parce que le contexte dans lequel le logiciel évolue change. Un exemple de changement dans le contexte est l'apparition d'une nouvelle menace de sécurité qui n'avait pas été prise en compte lors de la conception d'un logiciel. Des outils spécialisés peuvent être utiles pour aider à réaliser les tâches de maintenances rendues nécessaires par l'évolution des logiciels.

Cette thèse a présenté des approches pour aider aux tâches de maintenance liées à la détection, la protection, l'évolution et aux tests de défaillances dans un logiciel. Nous avons décrit ces approches en les appliquant à l'étude d'attaques de type SQL-injection pouvant provenir d'un attaquant interne ou externe et la génération de cas de tests pour l'optimiseur de la base de données DB2.

Ces approches ont été testées avec succès dans le cadre d'une étude de cas en les employant sur une ou plusieurs versions de l'application phpBB. Cette application écrite en PHP permet de gérer un système de babillard électronique. La base de données DB2 a aussi été utilisé afin de tester la génération de tests.

Les travaux découlant dans cette thèse ont fait l'objet de communications scientifique sous forme d'un rapport technique [1], de posters [2, 3] et d'articles avec comités de lecture dans des conférences [4, 5, 6, 7, 8, 9, 10, 11]

Une des principales contributions de cette thèse est la présentation d'une analyse de flux utile pour la détection et pour le suivi dans un cadre évolutif des vulnérabilités de type SQL-injection dans un logiciel écrit en PHP. Cette analyse considère les niveaux de sécurité effectifs dans l'application et les niveaux de sécurités requis pour accéder à la base de données. Une caractéristique avantageuse de cette analyse de flux est sa rapidité théorique et pratique démontrant un passage à l'échelle asymptotiquement linéaire.

Une autre contribution est la reprise, en utilisant le model checking, de l'analyse de flux utilisée pour la détection et l'évolution des vulnérabilités. Les résultats offerts par le model checking ont reproduits la précision, le rappel et la vitesse d'exécution de l'analyse de flux tout en offrant un cadre théorique plus formel et plus simple pour étendre cette analyse.

Cette thèse présente aussi une approche de réingénierie pour protéger automatiquement une application contre les SQL-injection en insérant automatiquement des barricades de sécurités dans le code source de l'application. Les barricades sont générés automatiquement

par une analyse dynamique de l'application qui va produire un ensemble de requêtes SQL légitimes qui seront analysées pour produire des patrons de requêtes légitimes utilisés par les barricades.

Aussi un algorithme génétique pilotant la génération de requêtes SQL a été présenté, la fonction d'adéquation est basée sur un modèle d'enchaînement des règles et une rétroaction lors de l'analyse de la requête par l'optimiseur de la base de données.

Ces approches ont été vérifiées expérimentalement en utilisant une perspective d'étude de cas avec deux applications ayant une taille significative soit phpBB et l'optimiseur de requêtes de la base de données DB2.

Des travaux futurs devraient permettre de mieux valider ces approches en effectuant des expérimentations sur des logiciels plus gros et plus variés que phpBB. De plus le modèle de sécurité booléen pourrait être étendu à des modèles de sécurité plus complexes.

Références

[1] Ettore Merlo, Dominic Letarte, and Giuliano Antoniol. Insider threat resistant SQL-injection prevention in PHP. Technical Report EPM-RT-2006-04, Ecole Polytechnique de Montreal, April 2006. URL `http://www.polymtl.ca/biblio/epmrt/rapports/rt2006-04.pdf`.

[2] Dominic Letarte, Ettore Merlo, Fahad Javed, Nattavut Sutyanyong, and Calisto Zuzarte. Meta-heuristic based tests of DB2 optimizer. Poster at the Conference of the Center for Advanced Studies on Collaborative Research, 2010.

[3] Dominic Letarte, Ettore Merlo, and Giuliano Antoniol. SQL-injection vulnerability analysis and protection in PHP applications. Poster at the conference of the center for advanced studies on Collaborative research, 2007.

[4] Dominic Letarte. Model checking graph representation of precise boolean inter-procedural flow analysis. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 511–516, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0116-9. URL `http://doi.acm.org/10.1145/1858996.1859099`.

[5] Dominic Letarte. Conversion of fast inter-procedural static analysis to model checking. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–2, 2010. ISBN 978-1-4244-8630-4. URL `http://dx.doi.org/10.1109/ICSM.2010.5609537`.

[6] Ettore Merlo, Dominic Letarte, and Giuliano Antoniol. Insider and ousider threat-sensitive SQL injection vulnerability analysis in PHP. *Reverse Engineering, Working Conference on*, 0 :147–156, 2006. ISSN 1095-1350. URL `http://doi.ieeecomputersociety.org/10.1109/WCRE.2006.33`.

[7] Ettore Merlo, Dominic Letarte, and Giuliano Antoniol. Automated protection of PHP applications against SQL-injection attacks. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 191–202, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2802-3. URL `http://portal.acm.org/citation.cfm?id=1251979.1252785`.

[8] Dominic Letarte and Ettore Merlo. Extraction of inter-procedural simple role privilege models from PHP code. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 187–191, Washington, DC, USA, 2009. IEEE Computer

Society. ISBN 978-0-7695-3867-9. doi : http://dx.doi.org/10.1109/WCRE.2009.32. URL http://dx.doi.org/10.1109/WCRE.2009.32.

[9] Francois Gauthier, Dominic Letarte, Thierry Lavoie, and Ettore Merlo. Extraction and comprehension of Moodle's access control model : A case study. In *Privacy, Security and Trust (PST), 2011 Ninth Annual International Conference on*, pages 44 –51, july 2011. doi : 10.1109/PST.2011.5971962.

[10] Ettore Merlo, Dominic Letarte, and Giuliano Antoniol. SQL-injection security evolution analysis in PHP. In *Proceedings of the 9th IEEE International Workshop on Web Site Evolution*, pages 45–49, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1450-5. URL http://portal.acm.org/citation.cfm?id=1524880.1525464.

[11] Dominic Letarte, Francois Gauthier, and Ettore Merlo. Security model evolution of PHP Web applications. In *IEEE Fourth International Conference on Software Testing Verification and Validation (ICST)*, pages 289 –298, march 2011. URL http://dx.doi.org/10.1109/ICST.2011.36.

[12] William G. J. Halfond and Alessandro Orso. AMNESIA : Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Automated Software Engineering (ASE)*. Association for Computing Machinery (ACM), Nov 2005.

[13] William G. J. Halfond and Alessandro Orso. Combining static analysis and runtime monitoring to counter SQL-injection attacks. In *Proc. of the 3rd International ICSE Workshop on Dynamic Analysis (WODA)*. IEEE Computer Society Press, May 2005.

[14] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, 2005.

[15] C. Gould, Z. Su, and P. Devanbu. JDBC checker : A static analysis tool for SQL/JDBC applications. In *Proc. of the 26th International Conference on Software Engineering (ICSE) - Formal Demos*, pages 697–698. IEEE Computer Society Press, 2004.

[16] R. McClure and I. Kruger. SQL DOM : Compile time checking of dynamic SQL statements. In *Proc. of the 27th International Conference on Software Engineering (ICSE)*, pages 88–96. IEEE Computer Society Press, 2005.

[17] W. R. Cook and S. Rai. Safe query objects : Statically typed objects as remotely executable queries. In *Proc. of the 27th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 2005.

[18] Ke Wei, M. Muthuprasanna, and Suraj Kothari. Preventing sql injection attacks in stored procedures. *Software Engineering Conference, Australian*, 0 :191–198, 2006. ISSN 1530-0803. doi : http://doi.ieeecomputersociety.org/10.1109/ASWEC.2006.40.

[19] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *International Symposium on Secure Software Engineering (ISSSE)*, pages 87–96. IEEE Computer Society Press, March 2006.

[20] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5) :236–243, 1976.

[21] A. C. Myers, N. Nystrom, L. Zhebg, and S. Sdanewic. Jif : Java information flow. *http ://www.cornell.edu/jif*.

[22] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3) :167–187, 1996.

[23] E. Merlo, G. Antoniol, and P. L. Brunelle. Fast flow analysis to compute fuzzy estimates of risk levels. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 351–360. IEEE Computer Society Press, 2003.

[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. of the 12th International World Wide Web Conference (WWW)*, May 2004.

[25] A. S. Christensen, A. Moller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. of the 10th International Static Analysis Symposium, SAS*, pages 1–18. Springer-Verlag, June 2003.

[26] Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *5th international workshop on Software engineering and middleware, SIGSOFT : ACM Special Interest Group on Software Engineering*, pages 106 – 113. ACM Press, 2005.

[27] M. M. Lehman. Programs life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9) :1060–1076, September 1980.

[28] M. Mattsson and J. Bosch. Observations on the evolution of an industrial oo framework. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 139–145, Oxford UK, Aug-Sept 1999.

[29] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 160–166, Bari Italy, Oct 1997.

[30] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 208–217, Bethesda MD, November 16-20 1998.

[31] E. Burd and M. Munro. Investigating component-based maintenance and the effect of software evolution : a reengineering approach using data clustering. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 199–207, Bethesda MD, November 16-20 1998.

[32] R. Holt and J. Y. Pak. Gase : Visualizing software evolution-in-the-large. In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 163–166, Monterey CA, November 1996.

[33] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Maintaining traceability links during object-oriented software evolution. *Software - Practice and Experience*, 31 :1–25, 2001.

[34] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44 :755–765, October 2002.

[35] E. Merlo, G. Antoniol, M. Di Penta, and F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analysis. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 412–416, Chicago, USA, Sept 11-17 2004. IEEE Computer Society Press.

[36] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. *Principles of Software Evolution, International Workshop on*, 0 :31–40, 2004. ISSN 1550-4077. doi : http://doi.ieeecomputersociety.org/10.1109/IWPSE.2004.1334766.

[37] Donald R. Slutz. Massive stochastic testing of sql. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 618–622, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. ISBN 1-55860-566-5. URL `http://portal.acm.org/citation.cfm?id=645924.671199`.

[38] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB*, pages 1243–1251. ACM, 2007. ISBN 978-1-59593-649-3.

[39] Hicham G. Elmongui, Vivek Narasayya, and Ravishankar Ramamurthy. A framework for testing query transformation rules. In *Proceedings of the 35th SIGMOD internatio-*

nal conference on Management of data, SIGMOD '09, pages 257–268, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi : http://doi.acm.org/10.1145/1559845.1559874. URL `http://doi.acm.org/10.1145/1559845.1559874`.

[40] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 499–510, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi : http://doi.acm.org/10.1145/1376616.1376668. URL `http://doi.acm.org/10.1145/1376616.1376668`.

[41] Daniele Romano, Massimiliano Di Penta, and Giuliano Antoniol. An approach for search based testing of null pointer exceptions. In *ICST*, pages 160–169. IEEE Computer Society, 2011.

[42] S. Wong, M. Aaron, J. Segall, K. Lynch, and S. Mancoridis. Reverse engineering utility functions using genetic programming to detect anomalous behavior in software. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 141 –149, oct. 2010. doi : 10.1109/WCRE.2010.23.

[43] Massimiliano Di Penta, Gerardo Canfora, Gianpiero Esposito, Valentina Mazza, and Marcello Bruno. Search-based testing of service level agreements. In *GECCO '07 : Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1090–1097, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-697-4. doi : http://doi.acm.org/10.1145/1276958.1277174.

[44] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9 :263–282, 1999.

[45] C.C. Michael, G. Mcgraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12) :1085–1110, 2001. ISSN 0098-5589. doi : http://doi.ieeecomputersociety.org/10.1109/32.988709.

[46] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14) :841–854, 2001.

[47] Nashat Mansour and Miran Salame. Data generation for path testing. *Software Quality Control*, 12(2) :121–136, 2004. ISSN 0963-9314. doi : http://dx.doi.org/10.1023/B:SQJO.0000024059.72478.4e.

[48] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA '07 : Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83,

New York, NY, USA, 2007. ACM. ISBN 978-1-59593-734-6. doi : http://doi.acm.org/10.1145/1273463.1273475.

[49] Ahmed S. Ghiduk, Mary Jean Harrold, and Moheb R. Girgis. Using genetic algorithms to aid test-data generation for data-flow coverage. In *APSEC '07 : Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 41–48, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3057-5. doi : http://dx.doi.org/10.1109/APSEC.2007.100.

[50] James H. Andrews, Felix C. H. Li, and Tim Menzies. Nighthawk : a two-level genetic-random unit test data generator. In *ASE '07 : Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 144–153, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi : http://doi.acm.org/10.1145/1321631.1321654.

[51] Gregory M. Kapfhammer and Mary Lou Soffa. Database-aware test coverage monitoring. In *ISEC '08 : Proceedings of the 1st conference on India software engineering conference*, pages 77–86, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-917-3. doi : http://doi.acm.org/10.1145/1342211.1342228.

[52] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *In Proc. ACM PLDI*, pages 203–213. ACM Press, 2001.

[53] Thomas Ball and Sriram K. Rajamani. Bebop : a path-sensitive interprocedural dataflow engine. In *PASTE '01 : Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 97–103, New York, NY, USA, 2001. ACM. ISBN 1581134134. doi : 10.1145/379605.379690. URL `http://dx.doi.org/10.1145/379605.379690`.

[54] Javier Esparza and Stefan Schwoon. A bdd-based model checker for recursive programs. In *In Proc. CAV'01, LNCS 2102*, pages 324–336. Springer-Verlag, 2001.

[55] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for java, 2002. URL `citeseer.ist.psu.edu/700455.html`.

[56] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *SSYM'05 : Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[57] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98 : Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi : http://doi.acm.org/10.1145/268946.268950.

[58] C. Anley. Advanced SQL injection. In *Technical report*. NGSSoftware Insight Security Research, 2002.

[59] C. Anley. Advanced SQL injection in SQL server applications. In *Technical report*, 2002.

[60] mySql. `http://dev.mysql.com/doc`.

[61] SQL. `http://www.iso.org`.

[62] G. Ollmann. Second-order code injection attacks. In *Technical report*. NGSSoftware Insight Security Research, 2004.

[63] PHP grammar, . `https://javacc.dev.java.net/files/documents/17/14269/php.jj`.

[64] JavaCC. `https://javacc.dev.java.net`.

[65] T. Holz, S. Marechal, and F. Raynal. New threats and attacks on the world wide web. *IEEE Security and Privacy Magazine*, 4(2) :72–75, 2006.

[66] National Infrastructure Security Co-ordination Centre. `http://www.uniras.gov.uk/niscc/index-en.html`.

[67] U.S. Department of Energy. `http://www.ciac.org/ciac/CIACHome.htm`.

[68] Michelle Keeney, Dawn Cappelli, Eileen Kowalski, Andrew Moore, Timothy Shimeall, and Stephanie Rogers. Insider threat study : Computer system sabotage in critical infrastructure sectors. Technical report, United States Secret Service and CERT Coordination Center/SEI, May 2005.

[69] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *ACM Workshop on Rapid Malcode (WORM)*, pages 76–82. ACM Press, 2003.

[70] Marisa Reddy Randazzo, Dawn Cappelli, Michelle Keeney, Andrew Moore, and Eileen Kowalski. Insider threat study : Illicit cyber activity in the banking and finance sector. Technical report, United States Secret Service and CERT Coordination Center/SEI, August 2004.

[71] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12) :701–726, 1998.

[72] D.C. Atkinson and W.G. Griswold. The design of whole-program analysis tools. *Proc. of the Int. Conf. on Software Engineering*, pages 16–27, 1996.

[73] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[74] K. B. Gallagher and J. R. Lyle. Using program slicing for software maintenance. *IEEE Transactions on Software Engineering*, 17(8) :751–761, 1991.

[75] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, 1999.

[76] Christian Hammer and Gregor Snelting. An improved slicer for java. In *ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 17 – 22. ACM Press, 2004.

[77] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3 (3) :121–189, 1995.

[78] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. *Proc. of the Int. Conf. on Software Engineering*, pages 433–443, 1997.

[79] phpBB, . `http://www.phpbb.com`.

[80] R. C. Holt, Andreas Winter, and Andy Schürr. Gxl : Toward a standard exchange format. *Proceedings 7th Working Conference on Reverse Engineering (WCRE 2000)*, 2000.

[81] phpBB archive, . `http://www.phpbb.com/phpBB/viewtopic.php?t=113826`.

[82] phpBB security, . `http://www.securityfocus.com/bid/7932`.

[83] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer-Verlag.

[84] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology*, 14(1) :1–41, January 2005.

[85] Salah Bouktif, Giuliano Antoniol, and Ettore Merlo. A feedback based quality assessment to support open source software evolution : the grass case study. In *Proceedings of IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 2006 (to appear).

[86] XPath. `http://www.w3.org/TR/xpath`.

[87] Php, . `http://www.php.net/manual`.

[88] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. *SIGMOD Rec.*, 21 :39–48, June 1992. ISSN 0163-5808. doi : http://doi.acm.org/10.1145/141484.130294. URL `http://doi.acm.org/10.1145/141484.130294`.

[89] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3) :19–29, 1995.

[90] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen : generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 341–352, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi : http://doi.acm.org/10.1145/1247480.1247520. URL `http://doi.acm.org/10.1145/1247480.1247520`.

[91] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Trans. on Knowl. and Data Eng.*, 18 :1721–1725, December 2006. ISSN 1041-4347. doi : http://dx.doi.org/10.1109/TKDE.2006.190. URL `http://dx.doi.org/10.1109/TKDE.2006.190`.

[92] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural data flow analysis via graph reachability. In *Proc. of POPL'95*, pages 49–61, 1995.

[93] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp : Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[94] D. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, Lecture Notes on Computer Science vol. 1214, pages 607–621. Springer-Verlag, April 1997.

[95] Gerard Holzmann. *Spin model checker, the : primer and reference manual*. Addison-Wesley Professional, first edition, 2003. ISBN 0-321-22862-6.

ANNEXE A

Exemple de la réécriture d'un CFG en un automate de sécurité.

Cette annexe présente la transformation d'un exemple de code en un CFG et en un automate de sécurité. Pour des raisons d'espace occupé par l'automate de sécurité, l'exemple de code et son CFG sont réduit le plus possible. Néanmoins l'exemple est complet et non trivial. La Figure A.1 présente l'exemple de code, il y a deux fonctions, deux appels de fonction, un patron de sécurité et aucune structure de contrôle et de boucles. Ce choix a été fait pour représenter les mécanismes de transfert de contrôle inter-procéduraux tout en minimisant le nombre de nœuds et d'états qui seront nécessaires par le CFG et par l'automate de sécurité.

```
-: void fA(){
1:   fB();
2:   PATRON DE SÉCURITÉ
3:   fB();
4: }

5: void fB(){
6: }
```

Figure A.1 Exemple de code.

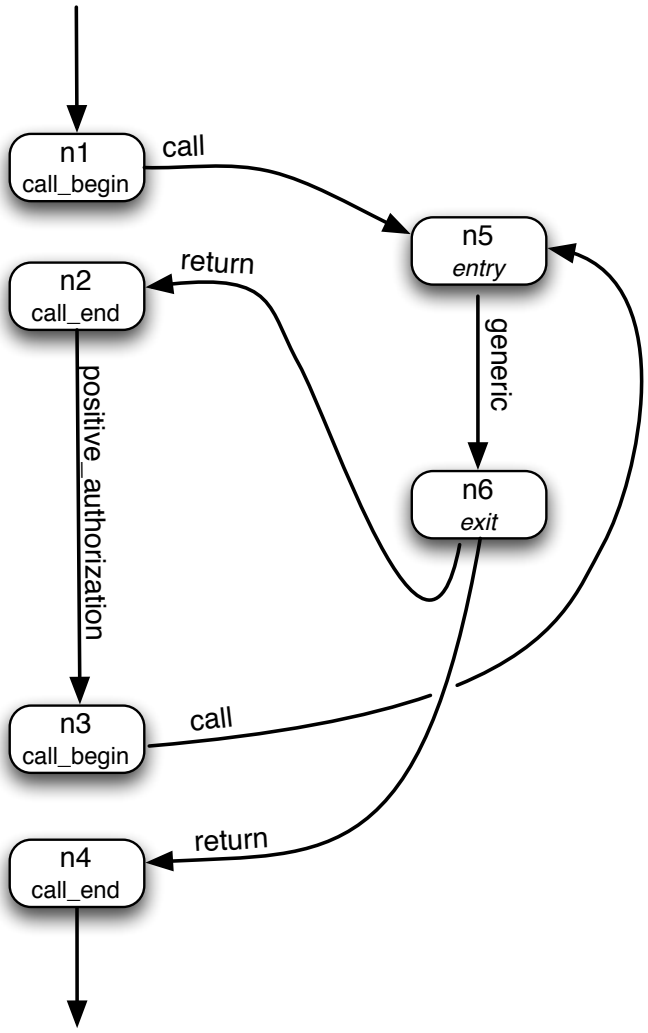

Figure A.2 CFG du code de la figure précédente.

La Figure A.2 présente de le CFG qui correspond au code de la figure précédente. Normalement le graphe de flux de contrôle est produit automatiquement en utilisant un parseur. Dans ce cas-ci, le graphe a été produit à la main afin de minimiser au maximum le nombre de nœuds nécessaire pour représenter le graphe. Conformément à la nomenclature de la section 7.2, les noeuds sont de type *call_begin*, *call_end*, *entry* ou *exit* et les arcs sont de type *generic*, *call*, *return* ou *positive_authorization*. Pour la simplicité de la lecture les identificateurs des nœuds (n1, n2, n3, n4, n5 et n6) correspondent au numéro des lignes dans l'exemple de code.

La Figure A.3 présente l'automate de sécurité produit en appliquant les règles de réécriture décrite dans la section 7.3 sur le CFG de la Figure A.2. L'identification des états dans l'automate diffère de la description faite dans section 7.3, l'identificateur d'état $q_{i,j,k}$

correspondra ici à ni,cj,vk où n réfère au nœud du CFG, c au contexte inter-procédural et v à la valeur du niveau de sécurité intra-procédural. Le même principe s'applique pour l'identification des variables *pass* sur les transitions de l'automate.

L'automate produit contient 24 états et 32 transitions. Ce qui peut sembler beaucoup venant d'un CFG qui contient 6 nœuds et 5 arcs mais c'est conforme au modèle décrit. Pour chacun des nœuds du CFG quatre états sont produits et pour chaque arc du CFG soit quatre ou huit transitions sont produites. On peut remarquer aussi que les nœuds et arcs du CFG possède des types différents selon leurs rôles dans le programme tandis que tous les états de l'automate sont semblables et ne possède pas d'information de type. L'automate produit peut être traité avec n'importe lequel algorithme ou outil régulier de model checking sans avoir de connaissance sur la sémantique des types employés par le CFG.

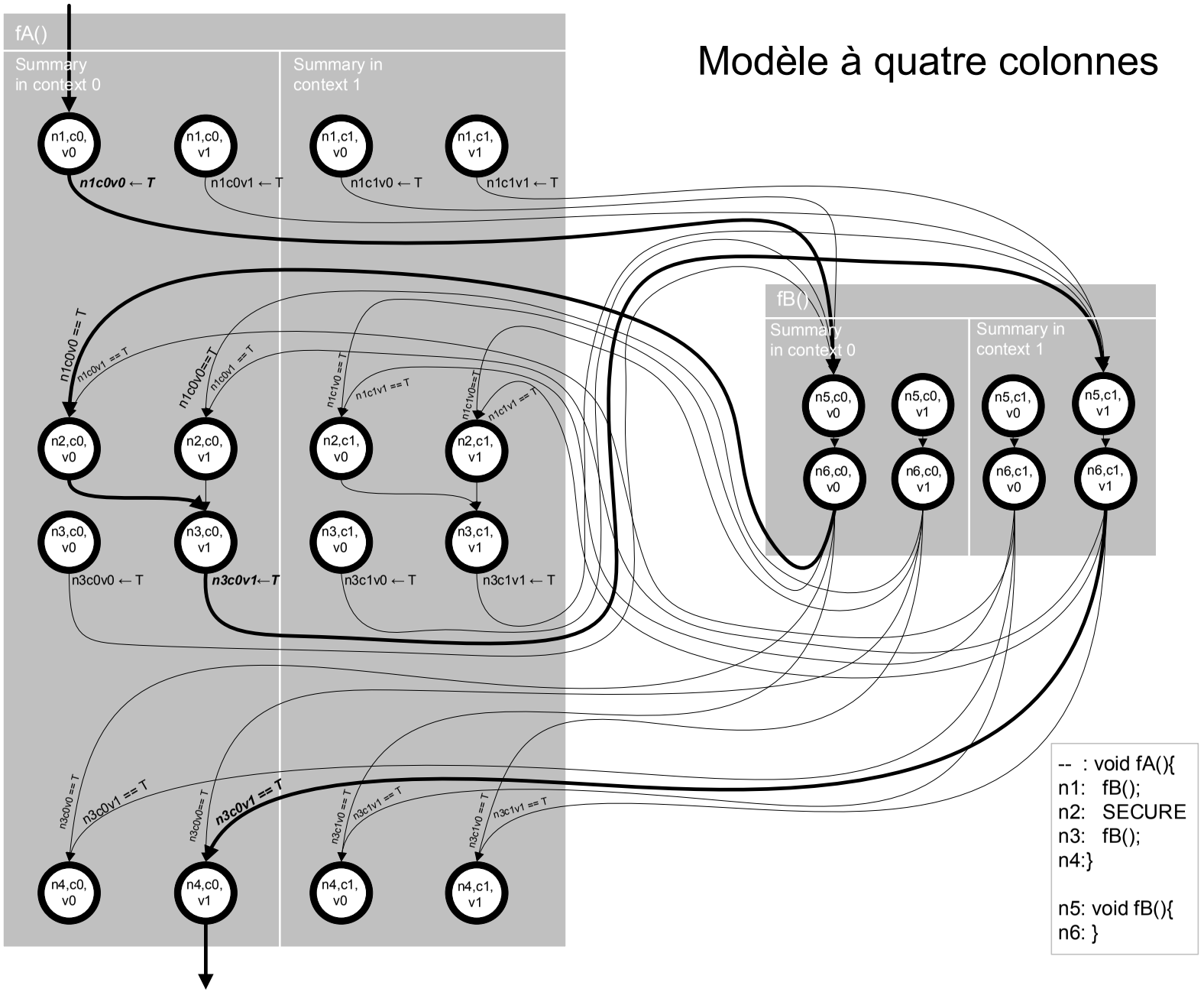

Figure A.3 Exemple d'un automate de sécurité.

ANNEXE B

Produit cartésien du modèle inter-procédural à quatre colonnes

Lors de la conception de l'automate de sécurité à partir de l'analyse de flux correspondante, plusieurs alternatives de représentation équivalentes s'offraient. Lorsqu'un automate est composé d'états et de variables il est possible d'effectuer un produit cartésien entre ceux-ci pour transformer un variable en état supplémentaires ou inversement. Les figures B.1 et B.2 considèrent que l'information sur le contexte inter-procédural et sur la valeur du niveau de sécurité intra-procédural sont des variables et peuvent être manipulés avec le produit cartésien. Les figures A.3, B.1 et B.2 sont équivalents en tout points. On peut noter qu'à mesure que le nombre d'états dans l'automate diminue, la complexité des conditions sur les transitions augmente.

Nous avons choisi d'écrire des règles réécriture dans la section 7.3 pour produire le modèle à quatre colonne de la Figure A.3 parce que c'est le modèle qui nous semblait le plus intuitif pour quelqu'un qui a déjà une connaissance de l'analyse statique inter-procédurale.

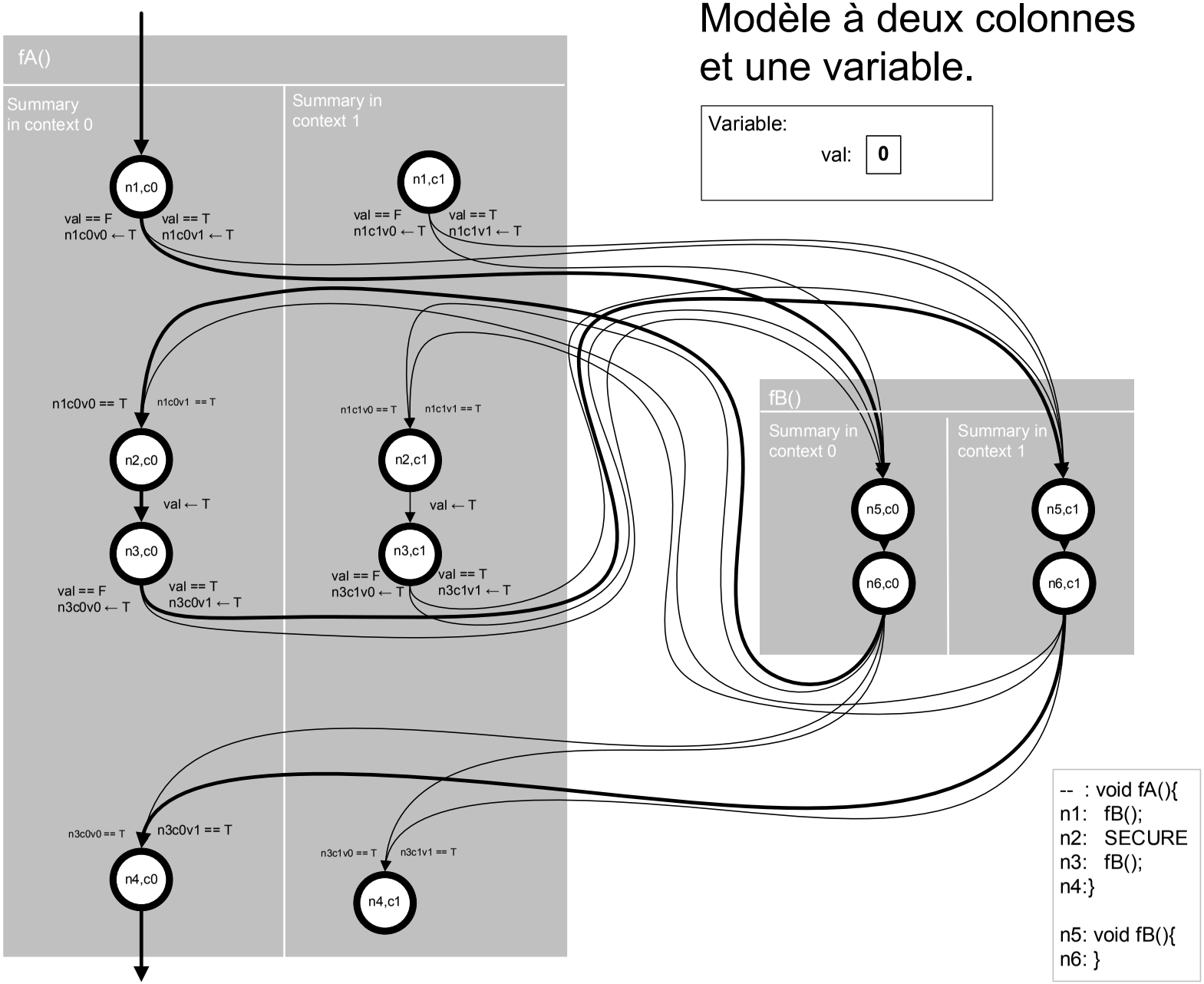

Figure B.1 Automate de sécurité alternatif appelé *Modèle à deux colonnes*.

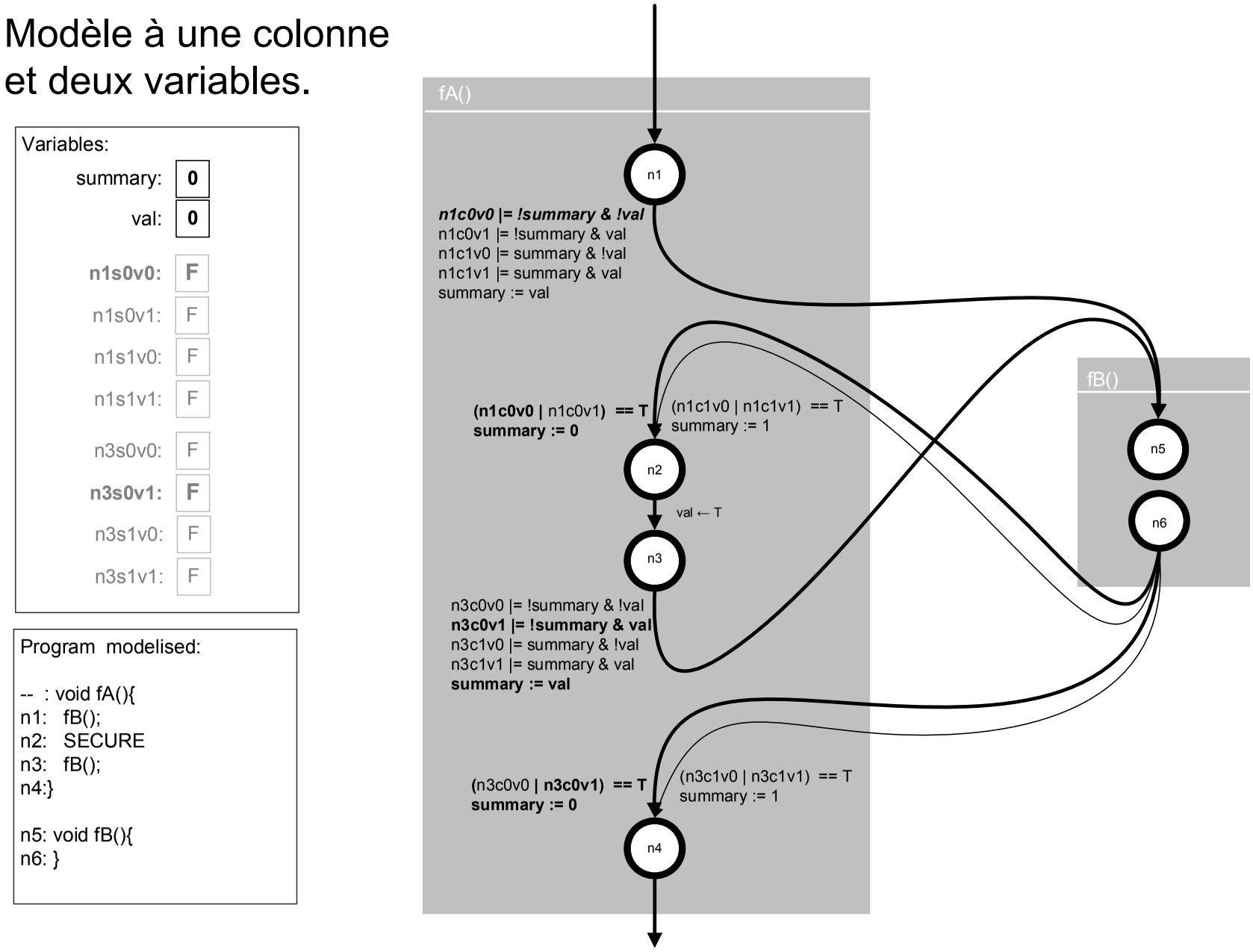

Figure B.2 Automate de sécurité alternatif appelé *Modèle à une colonne*.

ANNEXE C

Élimination possible d'une partie des variables *pass* dans le modèle inter-procédural

Cette annexe explore la possibilité qu'une partie des variables *pass* soit redondante et qu'il soit possible de diminuer le nombre de variables *pass* sans modifier d'aucune manière les résultats obtenus par l'automate construit. Dans l'automate décrit quatre variables *pass* sont définies pour chacun des points d'appels i dans le programme. Soit $pass[i, 0, 0]$, $pass[i, 0, 1]$, $pass[i, 1, 0]$ et $pass[i, 1, 1]$ qui intuitivement représente le fait d'être passé par un point d'appel i avec un tel contexte de sécurité et une telle valeur du niveau de sécurité.

Dans la Figure C.1 (et dans la Figure B.2 aussi) on peut s'apercevoir que les variables `n3c0v0` et `n3c0v1` sont toujours utilisés conjointement autant pour leurs assignations que pour leurs évaluations. Et ainsi de suite pour les couples de variables (`n3c1v0`, `n3c1v1`), (`n1c0v0`, `n1c0v1`) et (`n1c1v0`, `n1c1v1`). Il en résulte que l'ensemble de ces huits variables devrait pouvoir être réduite à un ensemble de seulement quatre variables `n1c0`, `n1c1`, `n1c0` et `n1c1` sans perdre d'expressivité. Intuitivement cela signifierait qu'il n'est pas nécessaire de conserver la valeur du niveau de sécurité intra-procédural dans les variables *pass*. Les figures C.2 et C.3 explorent cette transformation possible en modifiant le nombre de variables *pass*. Une démonstration plus formelle serait cependant nécessaire.

Le gain espéré par cette opération de réduction du nombre de variables *pass* est de divisé par deux le nombre de variables manipulés. Comme la complexité de la résolution de l'automate est déjà linéaire en fonction de la taille du programme étudié, la division de la taille de l'automate par un nombre constant ne va pas modifier la complexité de la résolution du système.

Modèle à quatres variables.

Variables:

summary:	**0**
val:	**0**
n1s0v0:	**F**
n1s0v1:	F
n1s1v0:	F
n1s1v1:	F
n3s0v0:	F
n3s0v1:	**F**
n3s1v0:	F
n3s1v1:	F

Program modelised:

```
--  : void fA(){
n1:   fB();
n2:   SECURE
n3:   fB();
n4:}

n5: void fB(){
n6: }
```

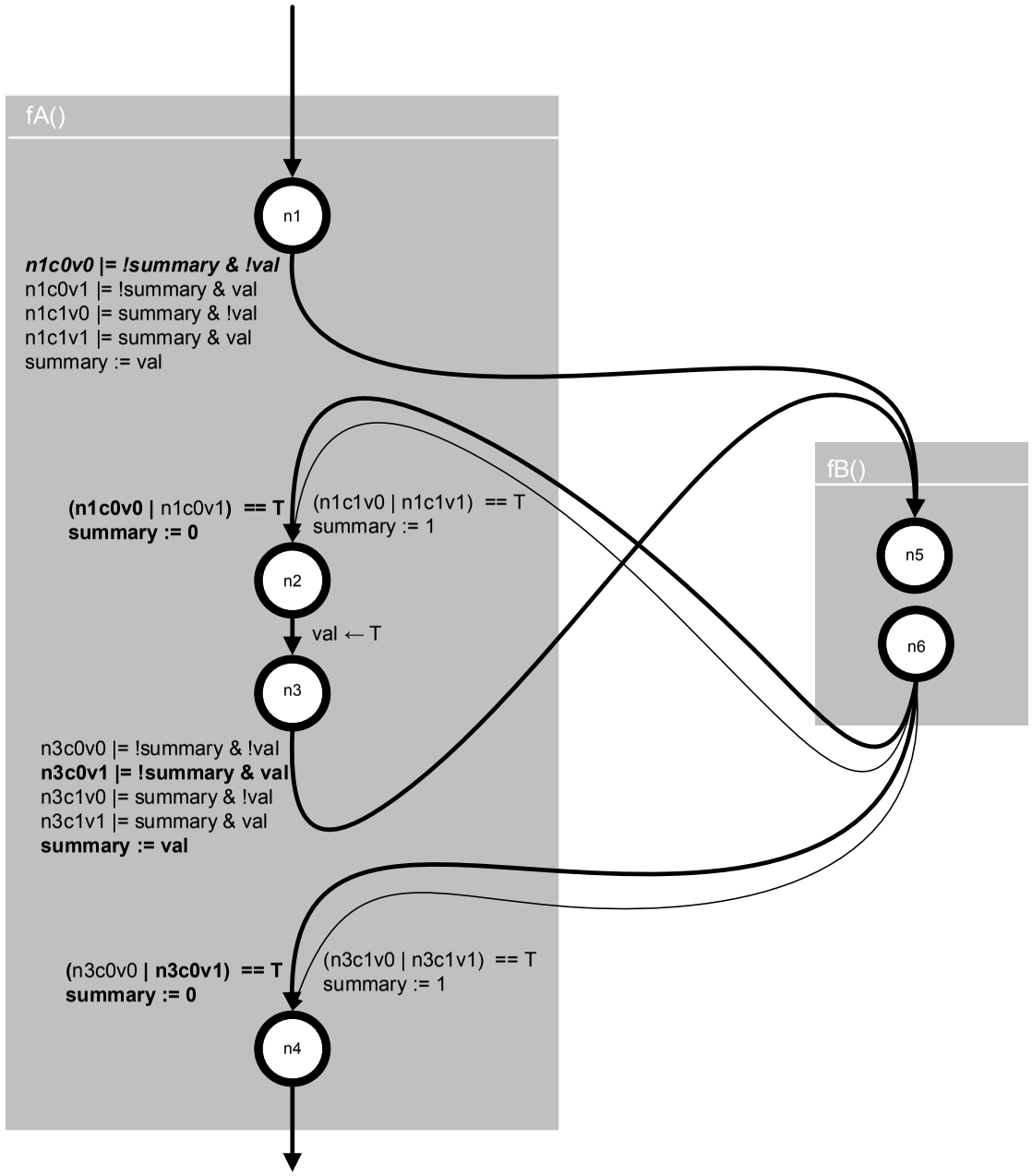

Figure C.1 Automate de sécurité alternatif appelé *Modèle à quatre variables.*

Modèle à quatre variables dans une matrice.

Variables:

summary: **0**

val: **0**

pass_n1:

val \ summary	0	1
0	**F**	F
1	F	F

pass_n3:

val \ summary	0	1
0	F	F
1	**F**	F

Program modelised:

```
--  : void fA(){
n1:   fB();
n2:   SECURE
n3:   fB();
n4:}

n5: void fB(){
n6: }
```

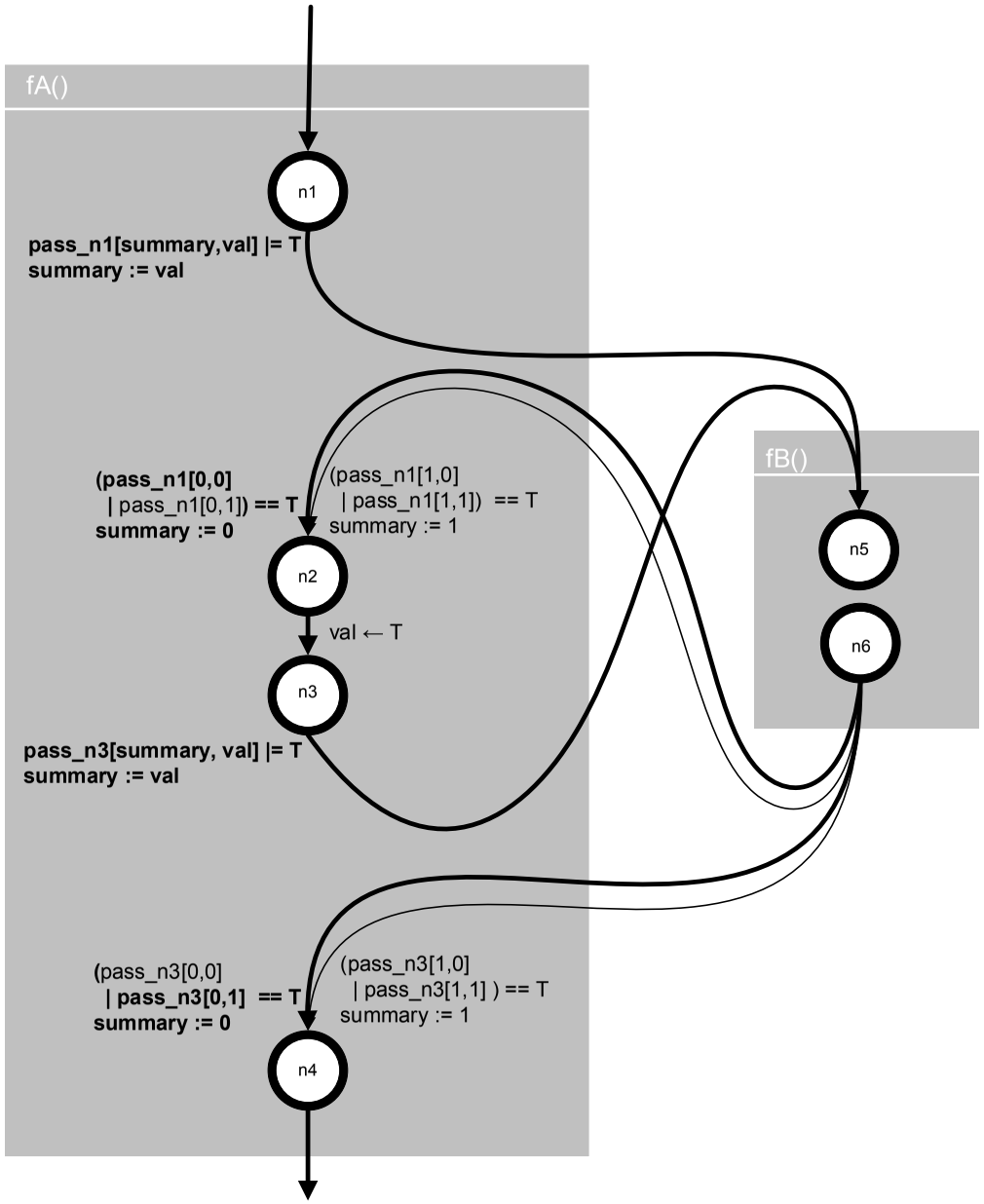

Figure C.2 Automate de sécurité alternatif appelé *Modèle avec une matrice de variables.*

Modèle à deux variables dans un vecteur.

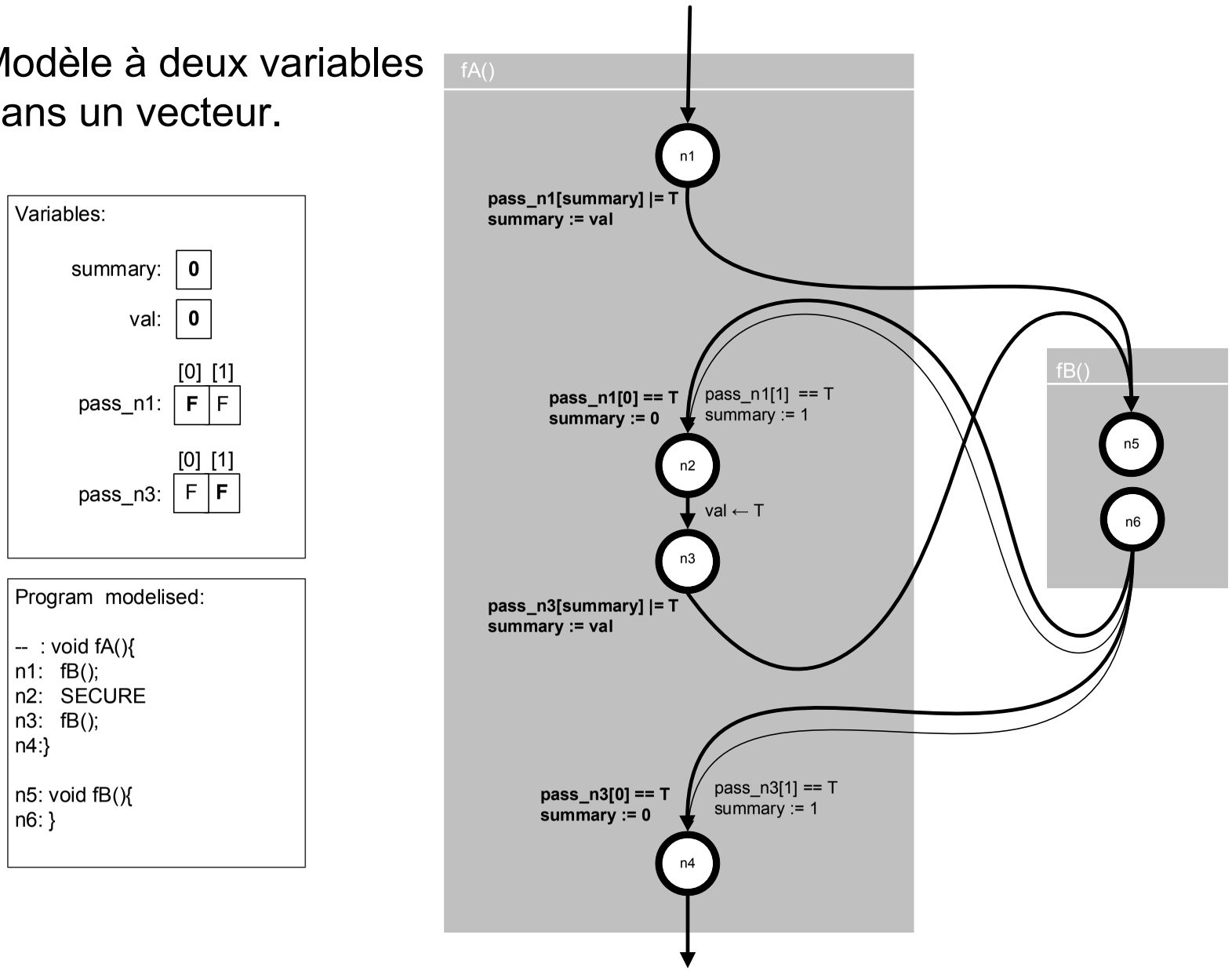

Figure C.3 Automate de sécurité alternatif appelé *Modèle à deux variables*.