Inférence automatique de modèles d'applications Web et protocoles pour la détection de vulnérabilités

Karim Hossen

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Karim Hossen

Thèse dirigée par **le professeur Roland Groz**
et co-encadrée par **le docteur Catherine Oriat**

préparée au sein **Laboratoire d'informatique de Grenoble**
et de **Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Inférence automatique de modèles d'applications Web et protocoles pour la détection de vulnérabilités

Thèse soutenue publiquement le **15 Décembre 2014**,
devant le jury composé de :

Mme. Marie-Laure Potet
Professeur, Grenoble INP, Présidente
M. Yves Le Traon
Professeur, Université du Luxembourg, Rapporteur
M. Jean-Christophe Janodet
Professeur, Université d'Evry, Rapporteur
M. Frédéric Dadeau
MCF, Université de Franche-Comté, Institut FEMTO-ST, Examinateur
M. Keqin Li
Docteur, SAP Labs, Invité
M. Roland Groz
Professeur, Grenoble INP, Directeur de thèse
Mme. Catherine Oriat
MCF, Grenoble INP, Co-Encadrante de thèse

Remerciements

Durant ma thèse, j'ai eu l'occasion de côtoyer et travailler avec de nombreuses personnes qui m'ont aidé à accomplir ce travail et je souhaitais les remercier.

Mon directeur de thèse, Roland Groz, ma co-encadrante Catherine Oriat et Jean-Luc Richier qui font tous partie de l'équipe VASCO et qui ont su me guider durant ces quelques années que ce soit sur mon sujet ou sur les diverses présentations et réunions auxquelles j'ai participé. Naeem Irfan et Mohammed Amine Labiadh pour avoir été des collègues de bureau intéressants et sympathiques.

Keqin Li et Alexandre Petrenko pour m'avoir donné la chance de participer à leurs travaux. J'ai pu aussi travailler dans le projet SPaCIoS où même étant un des derniers arrivants, j'ai été particulièrement bien accueilli et cela m'a permis de progresser avec la langue anglaise qui n'était pas forcément mon fort auparavant.

Tout les membres du projet SPaCIoS avec particulièrement Luca Vigano pour nous avoir su gérer un projet avec succès, Marius Minea et Petru Florin Mihancea pour avoir collaboré sur les aspects d'inférence, Johan Oudinet et Matthias Buechler sur la détection de vulnérabilités et Gabriele Costa pour avoir aidé à l'intégration de l'outil dans Eclipse.

Akram Idani, Roland Groz, Florent Autreau et Marie Laure Potet pour m'avoir permis de donner des cours et TP pendant ma thèse.

Marie Laure Potet et Laurent Mounier pour m'avoir accueilli à Verimag le temps de finir ma thèse et pour m'avoir permis de travailler dans le projet BinSec [BinSec 2013].

Les différents membres du club sécurité de l'Ensimag qui ont su faire vivre le club quand j'avais moins le temps de le faire. Avec en particulier François Desplanques, Josselin Feist et Guillaume Jeanne pour en avoir assuré la gestion.

Inférence de modèles d'applications Web et protocoles pour la détection automatique de vulnérabilités

Résumé : Les approches de tests basées sur les modèles (MBT) ont su montrer leur efficacité pour le test logiciel, mais elles nécessitent de disposer au préalable d'un modèle formel du logiciel. Dans la plupart des cas, ce modèle n'est pas disponible pour des raisons de coût, de temps ou encore de droits. Dans le cadre du projet SPaCIoS qui vise à développer une plate-forme pour le test de la sécurité basé sur les modèles, l'objectif est de fournir un outil d'inférence automatique de modèle pour les applications Web ainsi que des méthodes de détection de vulnérabilités à partir de ce modèle.

Nous avons conçu pour cela un algorithme d'inférence adapté aux applications Web et à leurs caractéristiques. Cette méthode prend en compte les données et leurs impacts sur le flot de contrôle. À l'aide d'algorithmes de fouille de données, le modèle est complété par des gardes ainsi que des fonctions de sortie. Nous avons aussi travaillé sur l'automatisation de la procédure d'inférence. Dans les approches d'inférence active, il est généralement nécessaire de connaître l'interface complète du système à inférer pour pouvoir communiquer avec l'application. Cette étape a été rendue automatique par l'utilisation d'un collecteur qui parcourt l'application pour en extraire les informations nécessaires et optimisées pour l'inférence. Ce collecteur est aussi utilisable par des méthodes d'inférence tierces.

Dans la version complète de l'algorithme, nous avons fusionné l'algorithme d'inférence et celui d'extraction d'interfaces pour obtenir une méthode automatique. Nous présentons ensuite l'outil libre SIMPA qui implémente ces différents algorithmes ainsi que divers résultats obtenus sur les cas d'études du projet SPaCIoS ainsi que des protocoles.

Mots clés : Inférence de modèle, test de la sécurité, rétro-ingénierie, application Web, protocole

Model inference of Web applications and protocols for automatic vulnerability detection

Abstract : In the last decade, model-based testing (MBT) approaches have shown their efficiency in the software testing domain but a formal model of the system under test (SUT) is required and, most of the time, not available for several reasons like cost, time or rights. The goal of the SPaCIoS project is to develop a security testing tool using MBT approach. The goal of this work, funded by the SPaCIoS project, is to develop and implement a model inference method for Web applications and protocols. From the inferred model, vulnerability detection can be done following SPaCIoS model-checking method or by methods we have developed.

We developed an inference algorithm adapted to Web applications and their properties. This method takes into account application data and their influence on the control flow. Using data mining algorithms, the inferred model is refined with optimized guards and output functions. We also worked on the automation of the inference. In active learning approaches, it is required to know the complete interface of the system in order to communicate with it. As this step can be time-consuming, this step has been made automatic using crawler and interface extraction method optimized for inference. This crawler is also available as standalone for third-party inference tools.

In the complete inference algorithm, we have merged the inference algorithm and the interface extraction to build an automatic procedure. We present the free software SIMPA, containing the algorithms, and we show some of the results obtained on SPaCIoS case studies and protocols.

Keywords : Model inference, security testing, reverse-engineering, Web applications, protocols

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Ce chapitre d'introduction présente le contexte de cette thèse en commençant par le projet SPaCIoS qui a financé les travaux présentés dans les chapitres suivants, puis les raisons du besoin en nouvelles méthodes de test de la sécurité des applications Web avec une description de l'Internet des Services (IoS) et des problèmes liés à la sécurité Web. Enfin, nous présentons les diverses contributions ainsi que la structure du document de thèse.

Sommaire

1.1 Contexte

On estime en 2013 à environ 2.7 milliards [Sanou 2013], le nombre de personnes utilisant Internet soit 39% de la population mondiale. D'un autre côté, Netcraft [Netcraft 2014] a pu recenser en 2014 près d'un milliard de sites Web actifs et autant d'applications Web potentiellement vulnérables. Le développement des technologies liées au Web, des services et moyens d'accès comme les smartphones ont contribué à ce succès, mais en contrepartie, le Web est devenu un lieu où la moindre vulnérabilité peut être revendue au plus offrant puis exploitée à des fins financières, politiques ou simplement pour montrer ce dont est capable le pirate.

Les réseaux sociaux, comptes en ligne et mails sont devenus les cibles privilégiées des pirates. Les besoins de test de la sécurité de ces applications sont de plus en

plus grands, mais aussi compliqués à cause des technologies toujours plus évoluées. Le projet SPaCIoS [Vigano 2013] tente de répondre à cette problématique.

1.1.1 Le projet SPaCIoS

Le projet SPaCIoS (Secure Provision and Consumption in the Internet of Services) est un projet européen constitué de 7 partenaires, 5 provenant du monde académique (l'université de Vérone, ETH Zurich, l'Institut Polytechnique de Grenoble, l'université de Gênes, l'Institute e-Austria de Timisoara et l'université technique de Munich) et 2 du monde industriel (Siemens et SAP).

L'internet des services (IoS) a complètement changé la façon dont les applications sont pensées, implémentées, déployées et utilisées : l'application n'est plus le résultat de la programmation d'un seul composant effectuée par un seul groupe, mais la composition de plusieurs composants distribués au travers d'Internet et utilisés à la demande et de manière flexible selon les besoins. Cependant, les nouvelles opportunités offertes par ce type d'architecture ne pourront être utilisées pleinement que si les concepts, outils et techniques existent pour en assurer la sécurité.

Ce défi est le principal objectif du projet SPaCIoS. Il devra poser les fondations technologiques pour une nouvelle génération d'analyseurs pour la validation automatique de la sécurité et ainsi permettre l'amélioration de la sécurité de l'IoS. Pour arriver à ce but, l'outil SPaCIoS combine les dernières technologies et méthodes pour le test de pénétration, test de la sécurité, model-checking et apprentissage automatique. Cet outil a été ensuite mis en oeuvre sur des applications provenant du monde des logiciels libres, mais aussi de l'industrie. Cela permettra entre autres de convertir les résultats du projet en pratiques utilisées par l'industrie.

Dans la figure 1.1, nous avons la représentation de l'architecture de l'outil SPaCIoS.

La clé de l'approche SPaCIoS est l'utilisation de modèles pour la représentation du système et de multiples méthodes pour la génération de cas de test pour le test de la sécurité (tests par mutations, détection de violation de propriétés, simulation de tests de pénétration). Comme dans la plupart des cas, aucun modèle formel de l'application n'est disponible et le testeur doit lui même écrire ce modèle. Le but de l'inférence de modèle est justement de résoudre ce problème en fournissant à l'utilisateur un moyen de générer un modèle de l'application à tester de manière automatique. Si ce modèle n'est pas forcément équivalent à un modèle écrit manuellement, il fournit néanmoins un modèle de départ que l'utilisateur pourra compléter.

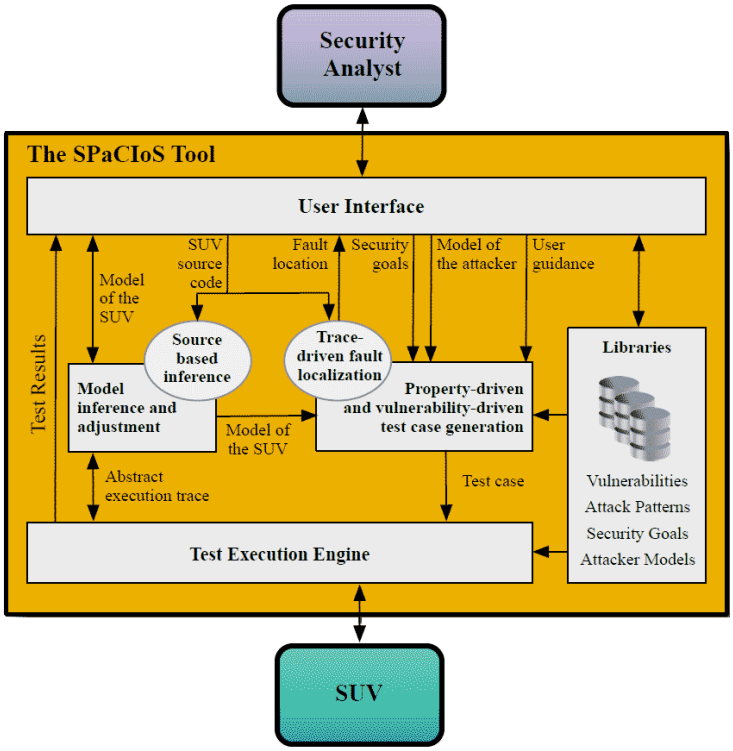

FIGURE 1.1 – Architecture de l'outil SPaCIoS

1.1.2 Inférence de modèle

La méthode de génération de cas de test guidée par des propriétés considère que chaque composant du système à tester a été modélisé en ASLan++ [Oheimb 2012] [D3.2 2011], le langage de haut niveau de spécification de modèle du projet, mais cela demande au préalable d'avoir une certaine connaissance du comportement interne du système, de ses caractéristiques, mais aussi une certaine expertise dans la modélisation avec le langage ASLan++. Dans la majeure partie des cas, les testeurs n'auront pas les connaissances suffisantes pour effectuer cette modélisation surtout quand l'application est une combinaison de plusieurs composants développés séparément et par des équipes différentes.

L'inférence de modèle permet d'automatiser cette partie de l'approche en créant un modèle du système sous forme d'automate qui sera ensuite converti en ASLan++. L'outil SPaCIoS inclut deux approches d'inférence pour construire automatiquement ces modèles :

— Une méthode en boite noire, utilisant des interactions avec le système, qui est constituée des travaux présentés dans cette thèse et développée conjointement avec SAP.

— Une méthode en boite blanche à partir du code source de l'application. Cette méthode sera présentée plus en détail dans la section 2.2.1.1.

Le modèle peut aussi être écrit à la main par le testeur, mais utiliser les approches d'inférence permet de gagner du temps en créant au moins un patron de modèle qui pourra être ensuite modifié et complété. Nous pouvons par exemple le compléter avec des informations sémantiques selon les parties de l'application à vérifier ou les vulnérabilités à considérer.

De plus, les deux approches d'inférence sont complémentaires et il est possible de combiner les informations des deux modèles. Les techniques en boite noire trouveront plus facilement les chemins faisables de l'application et cela permettrait de réduire l'effet de la sur-approximation des modèles extraits en boite blanche. En analysant le code source, il est plus facile de repérer les entrées importantes du système, celle qui aura un effet sur le flot de contrôle. Combiné avec l'inférence en boite noire qui demande justement de connaître ces entrées permettrait d'améliorer le modèle inféré comme présenté dans [D2.2.2 2013].

Les méthodes d'inférence en boite noire fonctionnent sur des exemples de comportement obtenus par du test actif, mais elles ne peuvent pas capturer toutes les informations sémantiques qui peuvent être spécifiées dans un modèle en ASLan++. Les modelés générés pour le projet SPaCIoS utilisent la même modélisation, une machine à états finie étendue, et le même format de fichier pour pouvoir être compatibles.

1.1.3 Les vulnérabilités Web

Les vulnérabilités considérées ici sont des faiblesses dans les applications qui peuvent permettre à un attaquant de détourner l'application de son fonctionnement normal. Cela peut porter atteinte à la confidentialité des données, l'intégrité, mais aussi à la disponibilité de l'application. Une faiblesse peut être due à un défaut au niveau de la conception [Armando 2012], configuration [Eshete 2013] ou de la programmation. Elles sont critiques selon leur impact ou l'application attaquée et elles sont en générale corrigées au fur et à mesure de leur découverte d'où le besoin de garder à jour ces applications. Certains travaux permettent de détecter les intrusions passées à partir d'un correctif [Kim 2012].

L'OWASP [OWASP 2014a] recense et classe les différentes vulnérabilités liées aux applications Web. En 2013, il existe 23 classes pour 169 types de vulnérabilité différents. De la qualité du code aux vulnérabilités liées à la cryptographie, chaque vulnérabilité est détaillée avec les moyens de prévention correspondants. L'OWASP

[OWASP 2013] maintient aussi un classement des dix vulnérabilités les plus importantes. Voici la liste du Top10 2013 ainsi qu'un exemple.

A1 - Injection de code : injection SQL, commandes systèmes, LDAP,

A2 - Authentification et gestion de session vulnérable : vols de cookies de session,

A3 - Cross-Site Scripting (XSS) : injection de code JavaScript permettant de voler des informations de la victime, comme un cookie de session,

A4 - Référence à des objets non sécurisés : liens directs vers des documents protégés,

A5 - Configuration non sécurisée : compte par défaut toujours actif,

A6 - Exposition de données sensibles : fichiers de configuration,

A7 - Manque de contrôle d'accès : pas de niveaux d'accès différents,

A8 - CrossSite request forgery (CSRF) : exécution d'actions malicieuses par l'intermédiaire d'un lien ou formulaire vulnérable,

A9 - Utilisation de composants vulnérables : utilisation de logiciel dont la version est connue pour être vulnérable,

A10 - Redirection non sécurisée : hameçonnage.

Dans les chapitres suivants, nous nous concentrerons sur les vulnérabilités qui sont détectables à partir du flot de contrôle de l'application (A2, A4, A6, A7) et des propriétés sur les données (A3, A8). Avec une approche de test basée sur les modèles, il devient nécessaire de définir le type de modèle adéquat pour les applications Web et les caractéristiques de l'algorithme d'inférence associé.

1.2 Problématique

Il existe assez peu de travaux sur l'inférence de modèle d'applications Web pour la détection de vulnérabilité. Les derniers travaux existants seront présentés dans le chapitre 2. Vouloir détecter des vulnérabilités à partir d'un modèle pose plusieurs questions auxquelles nous essayerons de répondre dans les chapitres suivants.

1.2.1 Quels types de modèles ?

Nous avons besoin d'un modèle qui peut exprimer suffisamment de caractéristiques pour pouvoir y détecter des vulnérabilités. Pour définir un type de modèle qui correspond, nous devons observer les caractéristiques des applications Web.

1.2.1.1 Gestion des entrées et sorties

Les applications Web fonctionnent sur le protocole HTTP qui fonctionne avec le principe de requête et de réponse. Dès lors, nous devons au moins considérer un modèle capable d'exprimer les entrées et sorties efficacement. L'algorithme d'Angluin, nommé L* [Angluin 1987], permet d'inférer un modèle sous la forme d'automate fini déterministe (AFD). C'est un algorithme qui a fortement été utilisé dans l'ingénierie logicielle et notamment pour du test.

Il est possible d'inférer un système avec entrées et sorties (I/O) avec un AFD comme dans l'adaptation de L* de [Hungar 2003] pour les systèmes réactifs. Mais cela multiplierait le nombre d'états en essayant d'exprimer ces relations d'I/O avec des états. De plus, l'approche SPaCIoS utilise un model-checker pour la détection de vulnérabilité et un grand nombre d'états ne ferait que perturber cette détection. Il y a aussi une raison pratique : les modèles seront transformés dans le langage ASLan++ et ce langage a été conçu pour garder une certaine simplicité dans la définition des systèmes. Avoir un modèle avec de nombreux états ne permettrait pas à un testeur de faire comprendre rapidement ce modèle pour pouvoir ensuite l'enrichir avec des informations sémantiques ou des propriétés de sécurité.

Nous devons au moins considérer un modèle comme la machine de Mealy, définie formellement dans le chapitre 2, des AFD ayant sur chaque transition, une entrée et une sortie associées. Avec les machines de Mealy, nous pouvons lier chaque requête et réponse HTTP à son symbole correspondant. Il reste une caractéristique des applications Web à prendre en compte : les paramètres. Contrairement aux protocoles ou aux systèmes embarqués qui sont des domaines dans lesquels les interfaces et les différents messages de l'application sont parfaitement définis (en taille et format), les applications Web permettent au développeur de créer et gérer des entrées et données de n'importe quel format et taille et cela rend plus compliqué de lier un symbole à chaque requête ou réponse.

Nous avons besoin d'un modèle à entrée et sortie qui prennent en compte les paramètres. Chaque entrée et sortie sera associée à un nombre variable de paramètres suivant l'application.

1.2.1.2 Adapté aux applications Web

Afin de supporter les mécanismes d'authentification comme ils sont présentés à l'utilisateur dans la page Web, certaines transitions dans le modèle ne seront franchissables que suivant certaines valeurs de paramètres. Il faut aussi prendre en compte le fait qu'une trace de ce modèle sera ensuite utilisée pour générer des cas de test concrets et que les valeurs des paramètres devront y être valides. Cela peut

se faire en ajoutant des prédicats basés sur la valeur des paramètres pour chaque transition.

Une des vulnérabilités sur lesquelles nous nous concentrerons, les XSS, nécessitent d'avoir une idée sur la valeur des paramètres des sorties. Un paramètre qui serait égal à une certaine entrée, comme dans le cas des réflexions XSS doit être détectable sur le modèle.

Nous obtenons ce qui ressemble une machine à états finie étendue et déjà utilisée auparavant dans [Ramalingom 2003] [Li 2006b] [Petrenko 2004a].

1.2.2 Une méthode automatique

Les algorithmes d'inférence basés sur l'algorithme d'Angluin [Angluin 1987] nécessitent de connaître l'interface complète du système à inférer (SUI). Dans le cas des applications et de l'infinité des possibilités en ce qui concerne les entrées et sorties et leurs paramètres, nous avons besoin d'une méthode en boite noire pour récupérer automatiquement ces informations. Nous devons aussi définir ce qu'est une entrée et une sortie pour une application Web. Cette méthode devra couvrir un maximum de fonctionnalités de l'application.

En plus des I/O, les méthodes d'inférence actives doivent pouvoir communiquer concrètement avec l'application. Les symboles d'entrées doivent être convertis en requêtes concrètes et les réponses concrètes en symboles de sortie. Cette étape est faite par l'adaptateur de test qui fait le lien entre le niveau concret et abstrait. Pour automatiser la méthode d'inférence, nous devons aussi être capables de générer ces adaptateurs de test automatiquement.

Cela pose une autre question sur le lien entre réponse HTTP et symboles de sortie. Dans le cas des entrées, nous pouvons associer chaque entrée à un symbole en fonction de son adresse et de ses paramètres. Mais lier une page Web à un symbole est plus difficile d'autant qu'une même page affichant différents paramètres devra être considérée comme un même symbole de sortie si nous utilisons I/O paramétrées.

En suivant l'approche SPaCIoS, à la fin d'une vérification, le model-checker peut renvoyer une trace d'attaque potentielle qui sera testée sur l'application. Les symboles d'entrées de cette trace ainsi que les paramètres devront être facilement convertis en requêtes HTTP. Ainsi, l'adaptateur de test jouera aussi le rôle de moteur d'exécution de tests.

1.2.3 Algorithme d'inférence adapté

Une fois le modèle étendu défini, l'algorithme d'inférence doit lui aussi correspondre aux caractéristiques des applications Web. En particulier, l'usage d'algorithmes basés sur L* soulève une question sur l'inférence. En partant du fait qu'un outil de type collecteur, qui parcourt le plus de pages possibles de l'application, a récupéré les entrées et sorties d'une application, nous savons qu'il est possible que cet outil n'ait pas couvert toutes les entrées de l'application. Cela implique que l'algorithme d'inférence partira d'un sous-ensemble des entrées, mais aussi qu'il soit possible que pendant l'inférence un nouveau comportement émerge avec de nouvelles entrées et sorties potentielles. Or les algorithmes de type L* ne peuvent pas prendre en compte une nouvelle définition de l'interface durant l'inférence.

Comme nous nous intéressons à des problèmes des sécurités, les différentes protections mises en place dans l'application doivent pouvoir être supportées par l'algorithme d'inférence. Si la plupart des mécanismes d'authentification peuvent être passés en fournissant les bonnes valeurs, l'utilisation de jetons aléatoires, de plus en plus présents dans les applications Web, empêchera les I/O découvertes précédemment de fonctionner. L'algorithme d'inférence doit donc détecter ces jetons et pouvoir les utiliser ultérieurement.

1.3 Contributions

La principale contribution est décrite dans le chapitre 5. C'est un algorithme d'inférence adapté pour les applications Web et qui combine les avantages des algorithmes existants pour en faire un outil de test de la sécurité.

Dans le chapitre 4, nous avons aussi développé une méthode de génération automatique d'adaptateur de test pour les applications Web. Cette méthode utilisée dans le projet SPaCIoS, mais aussi utilisable par des outils tiers, a permis d'automatiser l'inférence de modèle et aussi de servir de moteur d'exécution de test dans le projet.

Dans le chapitre 7 Nous avons développé l'outil SIMPA qui a permis d'évaluer les algorithmes d'inférence. En particulier, nous avons pu montrer que l'inférence de modèle pouvait effectivement être utilisée dans le contexte de la rétro-ingénierie avec la découverte de subtiles différences dans deux implantations du même protocole, mais aussi du test de la sécurité et de la détection de vulnérabilité pour les applications Web.

Nous avons découvert diverses vulnérabilités XSS sur des sites en production

notamment sur des sites sportifs, d'assurance et d'école qui sont fortement visités.

1.4 Structure de la thèse

Ce document de thèse est organisé de la façon suivante : le chapitre 1 présente le contexte de la thèse dont le projet SPaCIoS et les raisons de ces travaux, puis le chapitre 2 présente en détail le fonctionnement des applications Web et les principales vulnérabilités qui seront considérées dans les chapitres suivants. Puis, les derniers travaux en inférence de modèles pour la sécurité y seront détaillés avec une partie abordant les divers problèmes liés à l'inférence de modèle et à la détection de vulnérabilité à partir de modèle.

Le chapitre 3 présente deux algorithmes d'inférence différents développés avant et pendant la thèse et auxquels lesquels j'ai participé. Leurs avantages ainsi que leurs inconvénients y seront discutés. Puis, le chapitre 4 présente une méthode de création automatique d'adaptateurs de test pour de l'inférence. Il est suivi par le chapitre 5 qui présente une méthode d'inférence adaptée pour les applications Web en combinant diverses approches. Le chapitre 6 présente différentes méthodes de détection de vulnérabilité à partir du modèle inféré.

Enfin, le chapitre 7 présente l'outil libre nommé SIMPA qui implémente les algorithmes présentés dans les chapitres précédents. Le chapitre 8 termine en résumant les travaux présentés précédemment et aborde différentes perspectives.

Chapitre 2

Sécurité Web et inférence de modèles

Ce chapitre présente le fonctionnement et la construction des applications Web en se concentrant sur les aspects liés à la sécurité. Ensuite, nous avons une sélection des principales vulnérabilités et de leurs exploitations décrites plus en détail. Dans une seconde partie, nous abordons les derniers travaux d'inférence de modèles liée à la découverte de vulnérabilités pour les protocoles et les applications Web. Nous verrons ainsi différents modèles et méthodes d'inférence en boite blanche et noire.

Sommaire

2.1 Sécurité des applications Web

Les applications Web sont devenues un des moyens les plus utilisés pour fournir des informations aux utilisateurs. Avec presque un milliard de sites Web recensés en avril 2014 par Netcraft [Netcraft 2014] dont 39 millions de nouveaux sites rien qu'au mois de mars, les cibles potentielles pour les pirates ne manquent pas. L'arrivée des blogs et des pages personnelles mises à disposition par les fournisseurs d'accès ont grandement contribué à ce nombre. Mais ce n'est pas le nombre de ces sites qui crée des vulnérabilités, mais plutôt les technologies qui sont utilisées et qui ont évolué au fil du temps. Si les premières applications Web étaient faites en HTML seulement, de nombreuses technologies se sont ajoutées à ce standard qui a lui-même évolué pour fournir du contenu toujours plus complexe et plus rapidement.

2.1.1 Fonctionnement des applications Web

Pour pouvoir aborder les différents problèmes de sécurité des applications Web, il est nécessaire de rappeler leur fonctionnement tant au niveau de l'architecture, du protocole que du traitement des informations provenant de l'utilisateur.

2.1.1.1 Statique vs. Dynamique

Les premières applications étaient statiques puisque le serveur Web ne faisait que renvoyer le document que l'utilisateur demande sans traitements particuliers. Un mode de fonctionnement qui est parfait quand les données de l'application ne changent que rarement. Dans ce cas, il n'y a pas vraiment de vulnérabilités au niveau de l'application elle-même. Les requêtes de l'utilisateur ne contiennent pas de paramètres, mais uniquement le nom de la page directement. Bien qu'on puisse encore trouver quelques sites Web créés de cette façon, la quasi-totalité des sites Web actuels est dynamique.

Les sites Web dynamiques ont un fonctionnement légèrement différent comme illustré dans l'image 2.1. Le serveur ne renvoie plus directement les pages Web, mais il procède à un traitement sur la page en fonction des paramètres reçus dans la requête.

FIGURE 2.1 – Site Web dynamique

Les applications dynamiques se servent de paramètres qui proviennent de l'utilisateur ou de mécanismes de gestion de sessions. Suivant leurs utilisations, cela peut introduire plusieurs types vulnérabilités.

2.1.1.2 Requêtes paramétrées

Il y a deux principaux moyens de passer des paramètres à une requête. Ces méthodes sont définies dans le protocole HTTP.

— La méthode GET
Les paramètres sont ajoutés à la fin de l'adresse de la page Web correspon-

dante après l'ajout du caractère *?* pour distinguer la fin de l'adresse. Elles sont sous forme *clé=valeur* séparées par des '&' comme dans l'exemple 2.1.

Listing 2.1– Méthode GET

```
GET /webapp?arg1=hi&arg2=bob HTTP/1.1
Host: test.com
```

Cela à l'avantage de pouvoir être géré par des caches, historiques et les fonctions *précédent/suivant,* mais ils ont une longueur limitée et ils ne doivent surtout pas être utilisés pour des données sensibles puisque ces données sont visibles dans l'adresse.

— La méthode POST
Les paramètres ne sont plus passés dans l'adresse de la requête, mais dans le corps. Comme dans l'exemple 2.2, ils nécessitent l'ajout de deux métadonnées pour préciser la taille des données et leurs encodages. L'encodage permet donc d'éviter toutes ambiguïtés entre les données et les requêtes. (Ex. : $\backslash n \backslash n$ dans les données qui sert à délimiter l'entête du corps de la requête).

Listing 2.2– Méthode POST

```
POST /transfer.php HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 40

source=42424242&dest=12345678&value=123
```

Contrairement aux données GET, les données passées en POST ne sont pas mises en cache ni accessibles depuis un historique. Mais la méthode POST permet de travailler avec de grand volume de données. Du point de vue de la sécurité, les requêtes de type POST sont plus discrètes puisque non visibles dans l'adresse, mais elles sont tout de même transmises en clair.

Les paramètres des requêtes proviennent des utilisateurs et donc de potentiels attaquants. C'est le principal vecteur d'attaque, c'est pourquoi leur vérification avant leur traitement est indispensable pour assurer la sécurité de l'application.

2.1.1.3 HTTP : Un protocole sans état

Le protocole HTTP est un protocole sans état, c'est-à-dire qu'il ne conserve pas de lien entre les précédentes requêtes et les nouvelles. Seules les informations de la requête sont utilisées pour générer la page de sortie. Du point de pratique, cela empêche tout mécanisme d'authentification. C'est pourquoi plusieurs moyens ont été utilisés pour sauvegarder l'identité des émetteurs des requêtes et ainsi suivre un utilisateur sur les diverses pages du site Web.

— L'adresse IP identifie la machine qui a envoyé la requête. Elle n'est pas censée être contrôlable par l'utilisateur, c'est pourquoi elle est parfois utilisée pour identifier un utilisateur. Si elle est suffisante pour les sites Web simples, elle ne l'est plus si nous considérons aussi les utilisateurs derrières un proxy ou un NAT (Network Address Translation) puisque plusieurs utilisateurs réels auront la même adresse IP pour le site Web. Les NAT permettent à plusieurs ordinateurs d'un réseau à partager la même adresse IP publique. De plus avec les proxys gratuits disponibles sur Internet et les réseaux d'anonymisation comme TOR, l'adresse IP n'identifie pas forcément la machine émettrice.

— Le champ Referrer est rempli automatiquement par le navigateur Web et il contient l'adresse de la dernière page consultée par l'utilisateur. On peut donc s'en servir pour savoir d'où l'utilisateur provient et, par exemple, interdire les requêtes à une ressource spécifique si elle ne provient pas de la bonne adresse. Mais ce champ vient du côté client et il est possible, s'il le souhaite, de le changer pour passer cette protection.

— Le cookie est un ensemble de valeurs sauvegardées sous forme de texte côté client. Il correspond à un certain domaine ou site Web et il possède une durée limitée. Le serveur est capable de créer et modifier la valeur de ce cookie par l'intermédiaire des réponses HTTP. Suivant l'entête *Set-Cookie*, le navigateur met à jour la valeur du cookie. Cette valeur est ensuite automatiquement transmise au serveur à travers les requêtes HTTP et l'entête *Cookie* comme indiqué dans la figure 2.2. Il est possible de stocker n'importe quel type de valeur si elle est encodée correctement. Plutôt utilisés pour des données de petite taille, comme des identificateurs, ils sont notamment utilisés dans les sessions. En effet, le site Web peut conserver côté client des données concernant l'état de connexion du client. Le plus souvent, cela correspond à un identificateur unique qui identifie l'ensemble de données du client côté serveur.

Figure 2.2 – Gestion des cookies

Le cookie peut conserver des données critiques du point de vue sécurité et il est lisible depuis un script JavaScript. Pour éviter au JavaScript de lire ce cookie, il est possible de restreindre l'accès à ce cookie aux requêtes HTTP en spécifiant une valeur *httponly* dans le cookie.

— La session est un mécanisme fortement utilisé par les sites Web pour gérer et suivre les différents utilisateurs en même temps. Il se base sur un identificateur unique généré côté serveur une fois qu'un utilisateur s'est connecté au site Web. Côté serveur, cet identificateur est lié à une structure contenant différents champs, choisis par le développeur. Par exemple, le nom, prénom et le niveau d'accès. Côté client, seul l'identificateur est stocké en utilisant le cookie. Là encore le cookie a une grande importance dans la sécurité de l'application. Les langages tels que PHP ou ASP fournissent des API de gestion des sessions.

Bien qu'efficaces et utiles, les mécanismes de gestions de sessions sont vulnérables si le cookie et les champs associés à la session sont divulgués ou connus par un attaquant. Le secret du cookie est donc indispensable pour sécuriser ces informations.

2.1.1.4 Traitement des paramètres

Nous pouvons classer les paramètres en trois catégories selon leur utilisation par le serveur Web.

— Les paramètres statiques qui sont utilisés pour les valeurs qui sont statiquement liées à des données (par exemple un tableau de correspondance entre un symbole de langue et fichier contenant la traduction) ou actions. Pour le test de la sécurité, nous cherchons à éviter ce genre de paramètres puisque nous avons aucun contrôle sur elles.

— Les paramètres réfléchis qui sont directement utilisés dans la page de sortie sans aller chercher des données correspondantes dans les bases de données de l'application. Ce genre de paramètre permet de contrôler la page de sortie grâce à des requêtes spécialement construites et peut amener à une vulnérabilité.

— Les paramètres dynamiques qui sont utilisés pour récupérer les données liées à ce paramètre dans les bases de données, comme l'identificateur d'une page. Ces paramètres permettent potentiellement de contrôler les requêtes SQL correspondantes et rompre l'intégrité et la confidentialité des données stockées.

Les paramètres réfléchis et dynamiques sont des vecteurs de vulnérabilités et ce sont ces paramètres que l'on cherche en priorité lors des audits.

2.1.2 Les problèmes de sécurité

Dans la section précédente, nous avons décrit le fonctionnement des applications Web et mis en valeur certains aspects liés à la sécurité comme les cookies et les sessions. Ces aspects sont inhérents au protocole HTTP, au langage HTML et à l'architecture de l'application. Sans vérification, il est possible pour un attaquant d'utiliser ces mécanismes avec des valeurs spécialement formées pour effectuer des actions frauduleuses telles qu'une escalade de privilèges ou un vol d'information.

Dans la figure 2.3, nous avons les différentes vulnérabilités classées par occurrences de la liste du projet WHID (Web Hacking Incident Database) qui recense les incidents de sécurité liés aux applications Web. On peut voir que les attaques de type XSS occupent une grande partie des attaques connues. Les attaques sont classées selon leur pourcentage.

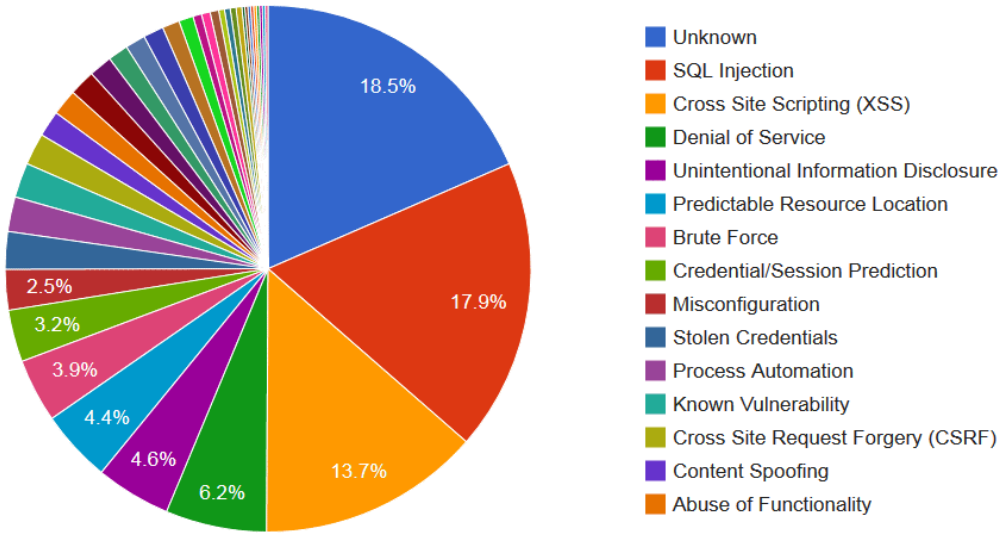

FIGURE 2.3 – Statistiques sur les incidents Web (WHID)

Dans les sections suivantes, nous avons une description des vulnérabilités qui sont considérées dans notre approche basée sur les modèles. Chaque vulnérabilité ainsi que son exploitation et sa prévention seront expliquées.

2.1.2.1 CrossSite Scripting (XSS)

En octobre 2005, le ver *Samy* a été le premier vers XSS à faire parler de lui. En seulement une nuit, il a pu infecter un million de profils sur le site *www.myspace.com*. En plus d'afficher un message sur les profils, il déclenchait des requêtes de demande d'amis avec le code du ver ce qui a permis d'accélérer l'infection. La solution adoptée par MySpace fut de couper les serveurs pour empêcher la propagation. Dans le cas de *Samy*, le code injecté par le ver était inoffensif et conçu uniquement pour la propagation. Mais si la même vulnérabilité avait existé sur des sites comme Google ou Yahoo, les conséquences auraient pu être bien plus importantes.

Les vulnérabilités de type XSS ou Cross-Site Scripting sont dans le Top 3 du classement OWASP 2013 [OWASP 2013]. Elles font partie des vulnérabilités de type injection dans lesquels du code HTML où JavaScript est injecté, par l'attaquant, dans la page Web renvoyée par le serveur à la victime. Ce code est ensuite injecté dans le navigateur de la victime et il permet au minimum de récupérer et envoyer des informations à l'attaquant, comme des identificateurs de session. Plus dangereux encore, il peut permettre de déclencher d'autres vulnérabilités au niveau du navigateur comme des dépassements de tampon ce qui entraîne une exécution de code arbitraire sur la machine de la victime.

Comparées aux autres types de vulnérabilité, les injections XSS sont celles qui se propagent le plus vite et sans nécessité d'actions de la part de l'attaquant. Si le code injecté exécute une requête Web, il serait possible d'effectuer une attaque de type déni de service distribué (DDOS) [Mirkovic 2004] contre n'importe quel site Web en se servant des utilisateurs du site attaqué. L'attaquant peut injecter du code qui exécutera des requêtes Web (chargement d'image par exemple) ce qui entraînera un grand nombre de requêtes sur le site visé. De plus, cette vulnérabilité est indépendante du système d'exploitation et du navigateur. En contrepartie, il est assez rapide de la stopper puisqu'il suffit de trouver la page vulnérable qui contient le script malicieux et de la désactiver.

Il existe deux types de vulnérabilité XSS qui ne diffèrent pas sur leurs impacts, mais sur la propagation :

— Les XSS réfléchis proviennent une vulnérabilité au niveau du filtrage des entrées de l'utilisateur. Cette vulnérabilité se produit lorsqu'une entrée de l'utilisateur se retrouve dans la réponse du serveur. L'entrée est réfléchie, mais ce script doit être placé dans les paramètres d'une requête. Comme il ne se trouve pas sur le site, l'attaquant doit fournir un lien (ou une forme différente) à la victime. Une fois que la victime a cliqué dessus, le lien chargera la page dans son navigateur, le script contenu dans les paramètres y sera injecté et il sera exécuté comme tout autre script qui provient du site

Web. Il suffit qu'une seule entrée ne soit pas vérifiée et l'attaquant peut y placer du code JavaScript qui sera exécuté par la victime. Cette attaque est illustrée dans la figure 2.4.

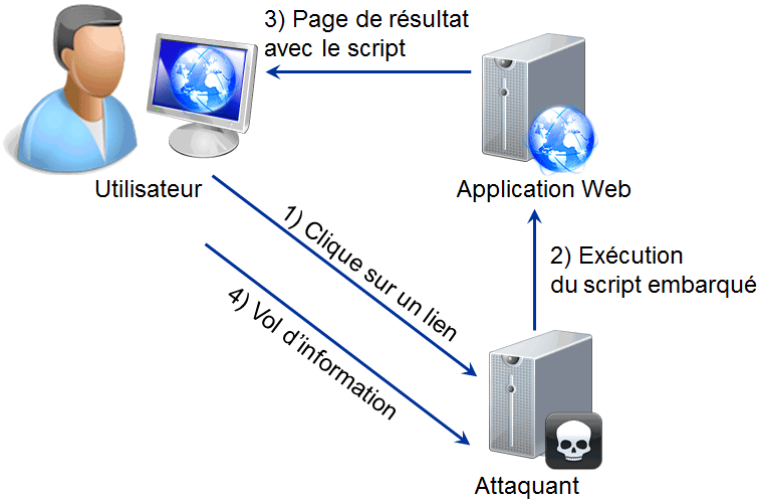

FIGURE 2.4 – Attaque utilisant une XSS réfléchie

— Les XSS persistants diffèrent dans le moyen de propagation. Le code injecté n'est plus dans l'adresse d'une requête, mais stockée sur le site Web. La raison est la même que pour les XSS réfléchis, un défaut de filtrage des entrées. Le script malicieux sera stocké et distribué dans chaque page vulnérable qui sera consultée par les utilisateurs et sera ensuite injecté dans leur navigateur. Pour un site avec un trafic soutenu, le nombre de personnes infectées sera très important (cf. ver Samy). Comme la page qui contient le script peut-être différente de la page vulnérable qui va le stocker, retrouver la source de l'infection est plus difficile.

Pour les XSS réfléchis, en une seule requête d'injection, l'attaquant peut infecter un utilisateur seulement (l'attaquant utilise généralement un email piégé envoyé à la victime) alors que dans le cas des XSS persistant, en une seule requête, l'attaquant peut infecter un grand nombre d'utilisateurs en même temps suivant le trafic du site Web. Une fois le code malicieux injecté côté serveur, chaque visiteur du site sera une victime potentielle.

Dans le code PHP 2.3, nous avons un exemple de page Web qui ne filtre pas les entrées provenant de l'utilisateur (GET). La valeur fournie en entrée par le paramètre *motcle* est directement réfléchie en sortie.

Listing 2.3– Page vulnérable aux attaques XSS réfléchie

```
<?php
  if (isset($_GET['motcle']))
```

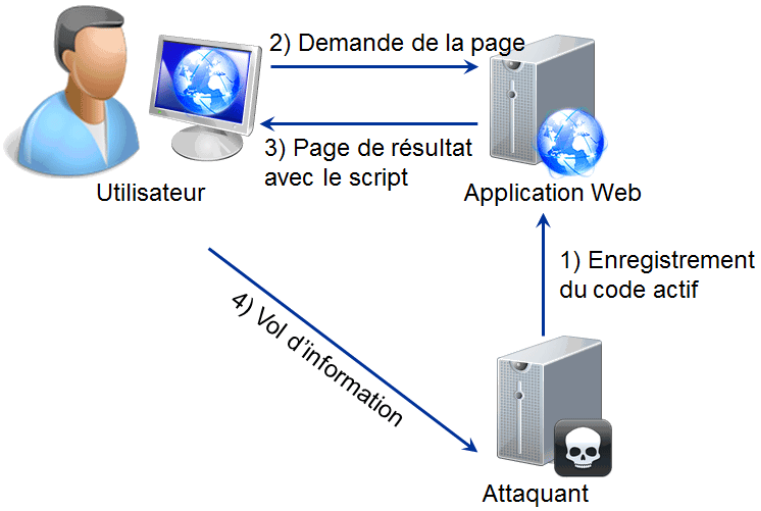

FIGURE 2.5 – Attaque utilisant une XSS persistante

```
  {
  echo "Vous recherchez : ".$_GET['motcle'];
  }
?>

Recherche:
<form method="get" action="search.php">
  <input type="text" name="motcle"/>
  <input type="submit" name="Submit" value="Go" />
</form>
```

En envoyant un lien contenant du code JavaScript comme valeur du champ *motcle*, comme dans la requête 2.4, l'attaquant fera exécuter son script sur le navigateur de la victime comme illustré dans la figure 2.6.

Listing 2.4– Attaque XSS

```
<a href="search.php?motcle=<script>alert('XSS')</script>
```

Les vulnérabilités XSS peuvent paraître moins critiques que les autres à cause des limitations des scripts JavaScript (par exemple pas de lecture ou d'écriture du client), mais leur vitesse de propagation et la facilité d'exploitation en font une attaque redoutable. Selon le rapport de WhiteHat Security en 2007, 80% des applications Web qui ont été testées étaient vulnérables.

2.1.2.2 CrossSite Request Forgery (CSRF)

En 2008, les utilisateurs de routeurs ADSL à Mexico ont été victimes d'une attaque CSRF via une balise image dans un email. Cette balise image exécutait une

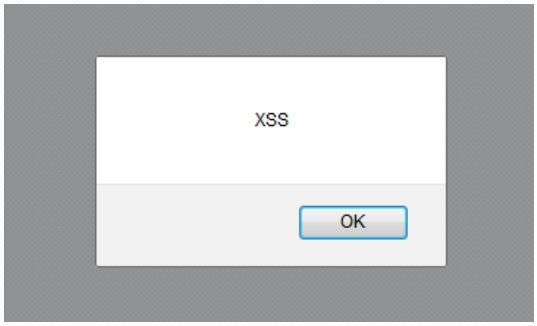

FIGURE 2.6 – Script exécuté par le client

requête sur le routeur ADSL et permettait de modifier la configuration DNS (qui lie les adresses IP et les noms de domaine) des sites liés à la banque de Mexico par un serveur contrôlé par l'attaquant. Un an plus tôt, ce sont 18 millions de clients d'eBay qui se sont fait voler leurs informations personnelles au travers d'une vulnérabilité de type CSRF sur la version coréenne du site.

Les vulnérabilités de type CSRF (appelé également XSRF) se placent à la huitième place sur le classement OWASP 2013 [OWASP 2013]. Cette vulnérabilité pourrait permettre de faire exécuter à la victime, non pas du code comme pour les XSS, mais une action sur une application Web. Très utilisée sur les routeurs personnels pour en modifier la configuration, elle peut aussi permettre de créer des comptes pour un attaquant ou encore altérer les données de l'application. Comme la requête est exécutée par un utilisateur légitime de l'application, une application Web vulnérable va exécuter cette requête comme si elle provenait vraiment de l'utilisateur comme illustré dans la figure 2.7.

La vulnérabilité se trouve dans la vérification de l'origine de la requête. Quand la victime aura cliqué sur le lien provenant de l'attaquant, l'application considérera l'action comme provenant de la victime et non de l'attaquant. Cependant, cette attaque possède quelques limitations :

— l'attaquant doit trouver une action qui lui permettra de récupérer ou modifier des informations même indirectement,
— la victime doit être authentifiée au moment de l'attaque CSRF si l'application nécessite une authentification.

Les attaques de type CSRF peuvent être combinées avec des attaques XSS comme dans le cas du ver Samy pour créer dynamiquement les charges utiles CSRF depuis l'attaque XSS. En 2009, un nouveau vecteur d'attaque CSRF a été présenté au BlackHat Briefing. Ce nouveau vecteur se sert des informations récupérées indi-

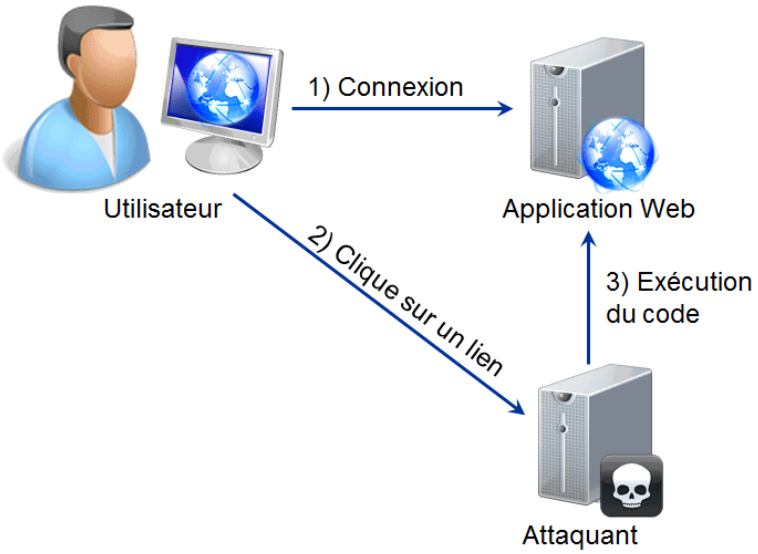

FIGURE 2.7 – Attaque CSRF

rectement pour contourner les méthodes de protection CSRF courantes.

Dans le code 2.5, nous avons un exemple de formulaire d'application Web pour le transfert d'argent entre comptes bancaires. Nous avons un champ pour le compte destinataire et un champ pour le montant à transférer.

Listing 2.5– Formulaire de transfert d'argent entre comptes

```
<form name="transfer" method="post"
  action="/Home/Transfer">
  <input type="text" name="destinationAccountId" value="1" />
  <input type="text" name="amount" value="1000" />
</form>
```

Un attaquant pourrait pré-remplir les différents champs et faire exécuter cette action par la victime grâce à une vulnérabilité CSRF. Dans le code 2.6, l'attaquant a pré-rempli les valeurs en mettant son compte en destinataire. Pour que la victime ne voie pas les champs falsifiés, l'attaquant a défini ces champs comme cachés. De plus, un script JavaScript va automatiquement envoyer la requête dès que la victime aura chargé la page Web ou l'email contenant le formulaire malicieux.

Listing 2.6– Formulaire pour l'attaque CSRF

```
<form name="transfer" method="post"
  action="http://mybank.com/Home/Transfer">
  <input type="hidden" name="destinationAccountId" value="1337" />
  <input type="hidden" name="amount" value="1000" />
</form>
```

```
<script type="text/javascript">
  document.transfer.submit();
</script>
```

Si la page ne vérifie pas de qui provient cette action, elle sera exécutée comme provenant de la victime et de l'argent sera transféré à l'attaquant.

2.1.2.3 Prévention et protection contre les vulnérabilités

Il existe des méthodes de protection pour chaque vulnérabilité présentée dans les sections précédentes. L'OWASP fournit notamment des guides de développement pour aider les développeurs à construire des applications Web plus sécurisées. Le but de cette partie n'est pas de lister de manière exhaustive les moyens de protection contre les vulnérabilités Web, mais plutôt de généraliser ces moyens. Au vu des vulnérabilités présentées précédemment, il apparaît que celles-ci proviennent toujours d'un manque de vérification sur les données utilisateurs, que ce soit au niveau du format de la donnée (XSS) ou de sa provenance (CSRF).

Pour contrer les attaques XSS, il faut vérifier que les entrées utilisateurs ne contiennent pas de texte pouvant être interprété comme script JavaScript ou balise HTML non autorisée. Le développeur peut autoriser certaines balises de formatage, notamment dans les messages de forums, mais ces balises devront être aussi vérifiées. La plupart des langages de développement d'application fournissent des fonctions pour encoder correctement les entrées pour qu'elles ne soient pas mal interprétées (par exemple, htmlspecialchars en PHP). De plus, il existe un certain nombre de bibliothèques spécialement conçues pour la prévention des vulnérabilités XSS, par exemple, PHP AntiXSS ou xss_clean.php filter.

Concernant les attaques CSRF, il faut vérifier la provenance de la requête sur le point d'être exécutée. Pour cela, la principale méthode, définie dans [Zeller 2008], consiste à générer une valeur pseudo-aléatoire et cryptographiquement sûre du côté du serveur et de l'inclure dans les formulaires renvoyés à l'utilisateur, mais aussi dans les cookies. La requête est considérée comme valide uniquement si la valeur transmise en paramètre de la requête correspond à celle contenue dans le cookie et que cette valeur est bien la valeur générée par le serveur.

2.2 Inférence de modèles et sécurité

Avec la multiplication de leur nombre, de leur complexité et de leurs fonctionnalités (gestion de compte bancaire, réseaux sociaux, achat en ligne ...), les applica-

tions Web sont devenues la cible privilégiée des pirates. Il suffit souvent d'une seule vulnérabilité dans une application pour mettre en danger toutes les informations des utilisateurs. Cela a aussi provoqué une hausse des besoins en test de la sécurité des applications Web.

Une des approches les plus simples pour tester la sécurité d'une application est d'utiliser un des nombreux scanners de vulnérabilités disponibles sur Internet [OWASP 2014b]. Ces scanners ont un fonctionnement en boite noire, c'est-à-dire qu'ils fonctionnent sans informations sur le code source ou le fonctionnement interne de l'application. Ces outils partent d'une ou plusieurs adresses de départ pour lesquelles ils vont récupérer chaque lien et formulaire. Chacune de ses actions est ensuite explorée avec différentes valeurs suivant les vulnérabilités à tester. Le scanner s'arrête de fonctionner quand chaque page de l'application a été explorée. Cette méthode a l'avantage d'être rapide et indépendante de l'application à tester. De plus, elle nécessite peu d'effort du point de vue du testeur, ce qui a participé à la popularité de ce genre d'outils.

Comme étudié dans [Doupé 2010], les scanners ont des limitations, principalement sur la couverture de l'application, mais aussi au niveau des vulnérabilités qu'ils peuvent détecter comme montré dans [Chen 2014] [Suto 2007] [Suto 2010] [Fonseca 2007]. Les XSS persistants sont un des cas plus difficiles à repérer puisqu'il est nécessaire de trouver où la réflexion du paramètre a lieu. C'est pourquoi nous avons besoin d'avoir plus d'informations sur l'application.

Avoir un modèle de l'application est une solution, mais la construction manuelle de ce genre de modèle peut s'avérer complexe et coûteuse. Durant la phase de développement de l'application, il est rare d'avoir et de maintenir un modèle à jour. Même si certaines méthodes à base de modèle créé à la main existent [Lebeau 2013], l'inférence de modèle permet de générer un automate plus au moins détaillé du comportement de l'application. Encore peu utilisée dans le domaine de la sécurité, elle était déjà utilisée pour le test logiciel [Raffelt 2009], [Meinke 2010], mais aussi dans la vérification et le test des composants en boite noire [Niese 2003], [Shahbaz 2008], [Shu 2007].

L'inférence de modèle peut se diviser en deux catégories :

— L'inférence passive qui utilise des traces positives (valides sur l'application) ou négatives (invalides) pour construire le modèle de l'application. Comme la méthode d'inférence ne peut pas demander d'autres traces, la principale limitation se trouve dans la qualité et la quantité de ces traces. Les techniques d'inférence développées dans [Biermann 1972], [Cook 1998] ou [Lorenzoli 2008] supposent qu'un nombre suffisant de traces du système

soient disponibles.

— L'inférence active qui utilise des interactions avec le système à inférer. Au lieu d'utiliser un ensemble limité de traces de l'application, les méthodes d'inférence actives vont demander leurs propres séries de traces en générant des séquences d'entrées qui vont être testées sur le système et dont les sorties seront observées. Progressivement, le modèle exact de l'application sera construit jusqu'à ce qu'un oracle nous assure que le modèle inféré est bien le modèle du système comme dans [Angluin 1987]. D'autres méthodes actives sans oracle ne s'assurent pas de la complétude du modèle inféré [Doupé 2010] [Duchene 2013].

Dans les sections suivantes, nous présentons différentes méthodes d'inférence pour la détection de vulnérabilité dans les applications Web. Nous commençons par une méthode en boite blanche puis en boite noire. Pour des soucis de place, nous nous sommes restreints aux méthodes d'inférence liées à la sécurité uniquement.

2.2.1 Boite blanche et inférence passive

Les méthodes suivantes se basent sur des informations comme le code source [Mihancea 2014] ou des traces de l'application [Pellegrino 2014].

2.2.1.1 Inférence depuis le code source

Développé dans le cadre du projet SPaCIoS en parallèle aux méthodes en boite noire, l'outil jModex [Mihancea 2014] permet d'extraire le comportement d'une application Web en analysant le code source. Les méthodes utilisant le code source ont déjà été utilisées auparavant pour détecter des manipulations de paramètres [Bisht 2011] ou encore des fautes logiques [Felmetsger 2010].

L'approche utilisée par jModex utilise la méthode définie dans le projet SPaCIoS avec l'utilisation d'un model-checker pour la découverte de vulnérabilité. Cette approche est détaillée dans la section 6.5. En ce qui concerne la modélisation, jModex analyse le code source des applications Web/Servlet développées en JSP. Une des limitations ce type d'outil est justement le nombre de langages supportés.

La première étape de la modélisation consiste à capturer le comportement de chaque composant de l'application sous forme de machine à états finie étendue. Ce type de machine est décrit formellement dans la section 3.1.2. Cela correspond à un automate fini avec entrée, sortie et prédicat pour chaque transition. Les nœuds de l'automate sont les points d'entrées de sorties des fonctions ou boucle et les

transitions représentent un chemin d'exécution. Pour chaque transition, l'analyse va générer une garde correspondante ainsi que les mises à jour des variables nécessaires.

L'outil utilise la bibliothèque WALA pour extraire un graphe d'appel et le flot de contrôle de chaque composant. Ensuite le graphe d'appel est parcouru en profondeur et chaque fonction est transformée en automate puis les automates des fonctions appelés sont fusionnés avec celui de la fonction appelante. À la fin de l'analyse, un seul automate global est retourné.

La transformation de chaque fonction en automate s'effectue en parcourant le flot de contrôle en profondeur, mais en partant du point de sortie pour remonter au point d'entrée. Afin de réduire le nombre d'états, chaque étape successive du flot de contrôle est fusionnée avec les autres si nécessaire en utilisant l'encodage LBE (large-block encoding) [Beyer 2009]. Pour différencier les étapes à garder dans le modèle, jModex possède une liste de fonctions liées aux bases de données ou aux mécanismes de sécurité.

Cette méthode possède quelques limitations liées à l'analyse du code source comme la gestion des boucles ou des constructions particulières du langage Java (exception, appel polymorphique). De plus, l'analyse du code source pourra trouver des chemins qui ne sont pas forcément satisfiables en pratique. L'inférence depuis du code source a tendance à produire une sur-approximation de l'application. Ce type de modèle est complémentaire avec les modèles obtenus en boite noire qui ne contiennent pas forcément tous les chemins possibles dans l'application (sous-approximation) [D2.2.2 2013].

2.2.1.2 Inférence passive depuis des traces

Dans [Pellegrino 2014], la méthode d'inférence de modèle cible les vulnérabilités logiques dans les applications Web. Cette méthode utilise un ensemble de traces valides provenant d'utilisateurs pour générer un modèle de l'application. Le modèle contient les actions valides effectuées par les différents utilisateurs. Suivant l'organisation des actions dans ce modèle, par exemple, une action est exécutée qu'une seule fois ou une action A est toujours exécutée avant B, des motifs d'attaques sont générés et testés sur le système. Afin de réduire la taille du modèle inféré, les traces se limitent aux fonctionnalités ciblées par le testeur comme un achat pour une boutique en ligne. La modélisation permet de créer un graphe de navigation. Ce graphe ne représente pas un modèle complet de l'application comme pourrait le faire jModex, mais des séries de chemins possibles. Le but étant d'avoir les comportements types des utilisateurs. Formellement, un graphe de navigation $G = (C \cup I, F, E)$ où C est l'ensemble de groupes, I le nœud d'origine, F le nœud final et E un ensemble

de transitions. Une transition (u, v) est créée s'il existe une trace π dans laquelle une ressource $r' \in u$ précède immédiatement une ressource $r'' \in v$. Alors pour chaque r_j au début de chaque trace (c'est à dire $\pi = \langle r_j, \ldots \rangle$), une transition (I, u) est placée où $r_j \in u$ et pour chaque r_j situé à la fin de chaque trace (c'est à dire $\pi = \langle \ldots, r_j \rangle$) une transition (u, F) où $r_j \in u$. Finalement, chaque nœud u est associé à l'ensemble de tous les éléments pour tout $r \in u$.

La première étape est nommée abstraction des ressources. Dans cette étape, les paires de requêtes et réponses HTTP sont abstraites de leurs paramètres et seule une information sur le type est gardée. Pour les requêtes HTTP, les différents paramètres GET ou POST sont identifiés et un type leur est attribué. Pour les réponses HTTP, le contenu de la réponse est abstrait à son tour pour ne garder que les informations pertinentes comme les différents liens, formulaires et leurs paramètres. Dans la figure 2.8, un paramètre a été identifié dans la requête HTTP, le paramètre *action* est de type *mot*. Pour la réponse HTTP, l'élément *item1* contient une des données de la page. Ces données sont constituées d'un *prix* et d'une *taxe* qui sont tous les deux de type *décimal*.

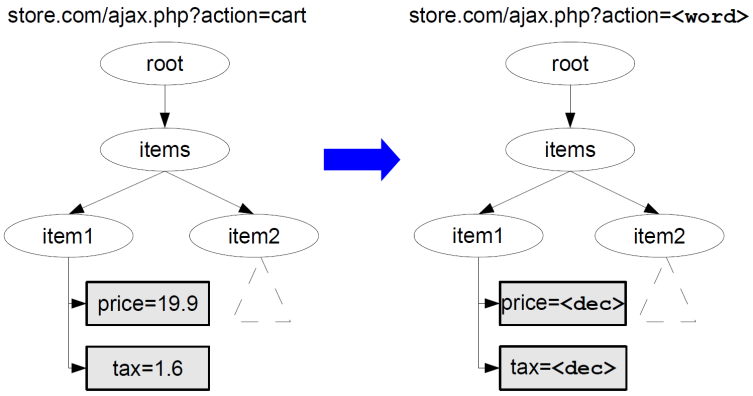

FIGURE 2.8 – Étape d'abstraction des ressources

L'autre partie consiste à séparer les groupes qui ont été considérés comme équivalents à cause de l'abstraction des ressources, mais qui sont en fait plusieurs actions bien différentes. Dans la première étape, chaque valeur de paramètre a été remplacée par un type correspondant. Si cette méthode se prête bien pour les paramètres qui influent sur les données à récupérer, c'est différent pour les paramètres qui permettent de sélectionner l'action à faire.

Dans la figure 2.9, nous avons en (a) une trace après l'abstraction des ressources. Nous pouvons voir que la dernière action AJAX peut revenir à l'état précédent pour exécuter la page *do.php*. Mais l'analyse des paramètres montre que le paramètre *action* permet de choisir entre des actions bien distinctes sur le système, c'est pourquoi

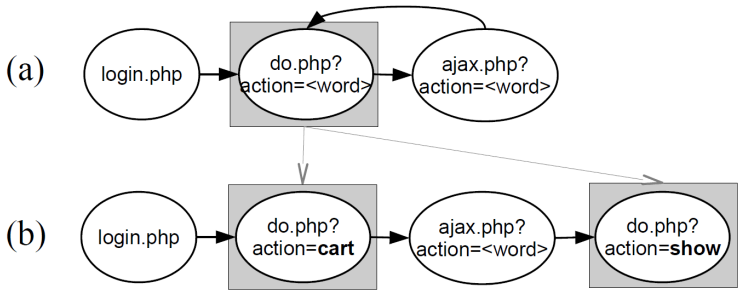

FIGURE 2.9 – Étape de séparation des actions différentes

nous pouvons voir en (b) que le deuxième état a été séparé en deux pour pouvoir refléter ces deux actions distinctes. On obtient au final une trace compacte et abstraite des actions des utilisateurs.

La détection de vulnérabilité, inspiré par [Wang 2011] et [Wang 2012], se fait en analysant les traces pour en distinguer des motifs. Dans [Pellegrino 2014], deux types de motif sont définis. Les motifs de traces qui représentent ce que l'utilisateur fait et les motifs de modèle qui représente ce que permet de faire le modèle. Il y a trois motifs de trace qui sont définis :

— Les nœuds singleton qui sont visités au plus une fois.
— Les opérations multi-étapes qui sont une suite de nœuds visités toujours dans le même ordre.
— Les nœuds repères qui jouent un rôle important dans la communication avec l'application et qui existent dans chaque trace.

Ainsi que deux motifs de modèle :

— Les opérations repérables qui sont un ensemble de nœuds qui font partie d'une boucle et qui pourraient être répétées un certain nombre de fois.
— Les repères de modèle qui sont des nœuds qui sont forcément visités, quel que soit le chemin parcouru entre le nœud d'origine et le nœud final.

En se basant sur ces différents motifs, des attaques ont été créées pour essayer de casser ces motifs. Par exemple, exécuter plusieurs fois un nœud singleton.

Le modèle inféré possède quelques limitations sur la couverture de l'application. Mais l'approche ne vise pas à avoir un modèle complet, mais une série de chemins types qu'un utilisateur ferait. En générant des cas de test spécialement conçus pour casser ces motifs cela permet de tester des séquences qui n'ont pas forcément été testées pendant le développement puisqu'elles sont en quelque sorte incohérentes. En pratique, cette méthode s'est révélée particulièrement efficace.

2.2.2 Inférence active en boite noire

Les méthodes d'inférence actives en boite noire communiquent avec le système à inférer en testant des séquences d'entrées et en observant les réponses. Dans les sections suivantes, nous allons présenter en premier certaines méthodes basées sur l'algorithme d'Angluin [Angluin 1987] et ses évolutions pour différents types de modèles. Dans une seconde partie, nous allons voir d'autres méthodes actives en boite noire.

2.2.2.1 Algorithme d'Angluin - L*

La méthode d'apprentissage active avec des requêtes faites à un *professeur* qui devait répondre si oui on non la requête était valide. Elle fut introduite dans [Angluin 1981] et illustrée dans la figure 2.10. Dans [Angluin 1987], il fut prouvé que les langages rationnels pouvaient s'apprendre avec un nombre polynomial de requêtes et cet algorithme d'inférence fut nommé L*.

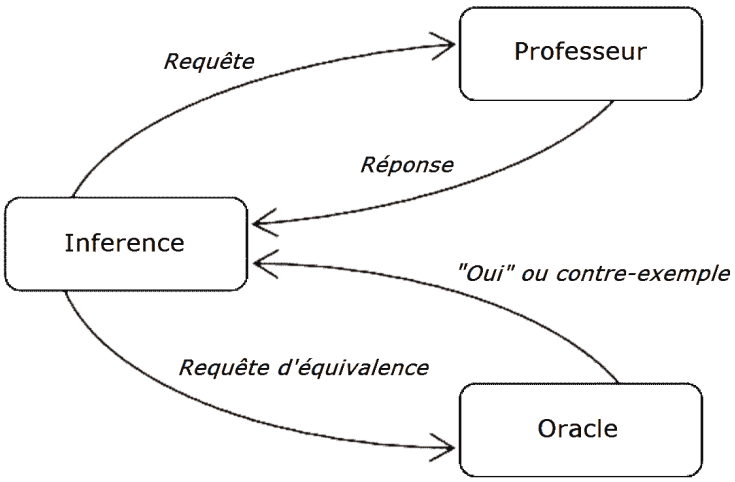

FIGURE 2.10 – Méthode d'apprentissage active

L'algorithme L* va explorer progressivement le système à inférer en envoyant des requêtes et en observant les réponses du professeur. Les réponses booléennes sont stockées dans une *table d'observation* 2.11 qui une fois qu'elle aura certaines propriétés, pourra être traduite en automate. Dans le cas de L*, c'est un automate fini déterministe (AFD) défini en 1 et l'algorithme suppose que le modèle formel du système à inférer (SUI) soit sous forme d'AFD.

Definition 1 *Un automate fini déterministe (AFD) est un quintuplet* $(Q\,;\,\Sigma\,;\,\delta\,;\,F\,;\,q_0)$*, où :*

— Q *est un ensemble fini non vide d'états,*
— $q_0 \in Q$ *est l'état initial,*
— Σ *est l'ensemble fini des lettres (l'alphabet),*
— $F \subseteq Q$ *est l'ensemble des états finals,*
— $\delta : Q \times \Sigma \to Q$ *est la fonction de transition.*

Cet automate n'est pas forcément équivalent au système, c'est là qu'intervient l'oracle qui recevra une *requête d'équivalence* et qui sera chargé de répondre si le modèle inféré correspond bien au système. Dans le cas contraire, il devra répondre par un contre-exemple qui sera ensuite utilisé par l'algorithme pour améliorer le modèle jusqu'à qu'une requête d'équivalence soit positive.

Même si c'est une méthode en boite noire, il est nécessaire que les entrées du système Σ soient connues et qu'il soit possible de réinitialiser la machine après chaque requête.

La table d'observation contient des lignes servant à l'exploration du système et des colonnes servant à distinguer les différents états. Comme nous allons le voir après, les lignes qui sont closes par préfixe et les colonnes closes par suffixe. Dans la figure 2.11, nous avons une table d'observation remplie avec l'alphabet $\Sigma = a, b$ au début de l'algorithme d'inférence.

Les lignes sont divisées en deux parties, une pour les états et représenté par le symbole S et une pour l'exploration des entrées suivantes notées $S.\Sigma$. La colonne contient uniquement le suffixe vide ε puisque nous sommes au début de l'algorithme et qu'aucun suffixe en plus n'est nécessaire pour distinguer les états.

		E
		ϵ
S	ϵ	0
	a	1
$S \cdot \Sigma$	b	1
	aa	1
	ab	0

FIGURE 2.11 – Exemple de table d'observation avec $\Sigma = a, b$

Pour qu'une conjecture (un modèle) puisse être construite à partir de cette table d'observation, elle doit valider deux propriétés :

— Close : chaque ligne dans $S.\Sigma$ doit avoir un équivalent dans S. Sinon cela signifie qu'un nouvel état a été trouvé et qu'il est nécessaire de le déplacer dans S puis d'explorer ses transitions sortantes.
— Compatible : toutes les lignes dans S sont considérées comme des états différents. L'exploration, une entrée plus loin doit donc avoir au moins une différence. Dans le cas contraire, l'entrée $e \in \Sigma$ qui différencie les deux lignes est ajoutée comme colonne puisqu'elle sert à différencier deux états.

Une fois ces propriétés validées, une conjecture est construite et une requête d'équivalence est envoyée à l'oracle. Si la conjecture n'est pas équivalente au SUI, l'oracle répond avec un contre-exemple CE dont tous les préfixes seront ajoutés comme colonne à la table d'observations qui est ensuite complétée. Ces étapes sont effectuées jusqu'à que l'oracle réponde que la conjecture est équivalente au système.

2.2.2.2 Inférence de machine de Mealy

La principale optimisation de l'algorithme L* se fait sur le nombre de requêtes faites au système. Une série de filtres prenant en compte le domaine des applications est proposée dans [Hungar 2003]. Afin de pourvoir l'appliquer aux systèmes réactifs et mieux correspondre au test logiciel, l'algorithme a été adapté et nommé Lm* pour pouvoir inférer des machines de Mealy [Niese 2003] [Li 2006a] [Shahbaz 2009], des automates à entrée et sortie (I/O) définis en 2. Bien qu'il était possible de modéliser indirectement des machines de Mealy avec L* [Hungar 2003], le fait de le faire directement permet de réduire considérablement le nombre d'états puisque seuls des états supplémentaires pouvaient représenter les I/O.

Definition 2 *Une machine de Mealy est un sextuplet $(Q\,;\, I\,;\, O.\,;\, \delta\,;\, \lambda\,;\, q_0)$, où :*
— *Q est un ensemble fini non vide d'états,*
— *$q_0 \in Q$ est l'état initial,*
— *I est l'ensemble fini des symboles d'entrées,*
— *O est l'ensemble fini des symboles de sorties,*
— *$\delta : Q \times I \to Q$ est la fonction de transition qui fait correspondre les entrées à l'état suivant pour chaque état,*
— *$\lambda : Q \times I \to O$ est la fonction de sortie qui fait correspondre les entrées à la sortie pour chaque état.*

L'adaptation est faite aussi au niveau de la table d'observation. Au lieu de résultat booléen sur le fait qu'une séquence d'entrées était acceptée ou non, la sortie [Shu 2007] est utilisée pour remplir la table. Au niveau de l'initialisation, les colonnes ne sont plus initialisées avec ε mais Σ. Quant aux suppositions, le seul ajout est l'alphabet de sortie qui doit être aussi connu.

Dans la figure 2.12, nous avons une table d'observations Lm* initialisée avec $I = \{a, b\}$ et $O = \{x, y\}$.

		E_M	
		a	b
S_M	ϵ	x	y
$S_M \cdot I$	a	x	y
	b	x	y

FIGURE 2.12 – Exemple de table d'observation Lm* avec $I = \{a, b\}$ et $O = \{x, y\}$

L'algorithme en lui même ne change pas, il y a toujours une phase de remplissage de la table jusqu'à qu'elle soit close et compatible. Ensuite une boucle de requête d'équivalence et de traitement du contre-exemple est exécutée jusqu'à que la conjecture soit considérée équivalente au système par l'oracle. La recherche de contre-exemple par marche aléatoire comme proposée dans [Angluin 1987] ou ses variantes [Cho 2010] [Irfan 2010b] [Irfan 2010a] a été aussi étudiée dans [Howar 2010] et le traitement des contre-exemples non-optimaux a été étudiée dans [Rivest 1993], [Maler 1991], [Shahbaz 2009] ou encore [Irfan 2013].

Un autre modèle d'automate, nommé automate à registres, est présenté dans [Howar 2012]. Ce modèle est un des premiers à prendre en compte des paramètres et leur influence sur le flot de contrôle. Les automates à registres fonctionnent avec des paramètres sur un domaine infini et les registres fonctionnent comme un ensemble de valeurs dans lesquelles l'algorithme peut piocher pour trouver des correspondances avec les paramètres. Cela permet d'inclure, par exemple, l'effet des couples nom d'utilisateur/mot de passe dans un mécanisme d'authentification.

2.2.2.3 Inférence de protocole de commande et contrôle d'un botnet

Dans [Cho 2010], nous avons l'une des premières utilisations de l'algorithme d'inférence Lm* [Shahbaz 2009] dans le contexte de la sécurité. Le système à inférer était celui du protocole SMTP du botnet MegaD [MacDonald] qui fut responsable de près d'un tiers du spam mondial [Caballero 2009]. Inférer un botnet réel et non une simulation demande de prendre quelques précautions pour ne pas se faire attaquer en retour, c'est pourquoi ils ont décidé d'utiliser le réseau d'anonymisation Tor [Dingledine 2004] même si cela réduit le nombre de requêtes faisables dans un temps raisonnable puisqu'une requête prenait 6.8 secondes en moyenne.

L'algorithme Lm* possède les mêmes restrictions quant au système, il doit être déterministe et il doit être possible de réinitialiser le système. En étudiant le pro-

tocole, ils n'ont trouvé qu'une seule source de non-déterminisme quand le botnet recevait le message m et qu'il pouvait répondre y_1 mais aussi attendre quelques secondes puis répondre y_2. Afin de résoudre ce problème, le message m fut divisé en deux messages distincts m_1 et m_2. Pour les applications Web et de nombreux protocoles réseau, la réinitialisation s'effectue en démarrant une nouvelle session soit par une commande particulière soit en effaçant toute information sur la session. Grâce au travail de [Caballero 2009], [Cui 2007] et [Cui 2008], ils ont pu identifier le message qui permettaient de réinitialiser le système.

Dans la figure 2.13, l'architecture utilisée par la plate-forme d'inférence. Nous pouvons voir qu'elle est constituée d'un mécanisme de prédiction basé sur l'analyse des boucles sur le même état, d'un cache pour éviter les requêtes redondantes au système et d'une série d'émulateurs de robot qui sont en fait les adaptateurs de test pour le botnet. L'adaptateur de test est responsable de la communication entre le niveau abstrait utilisé par l'algorithme Lm* et le niveau concret utilisé par le botnet.

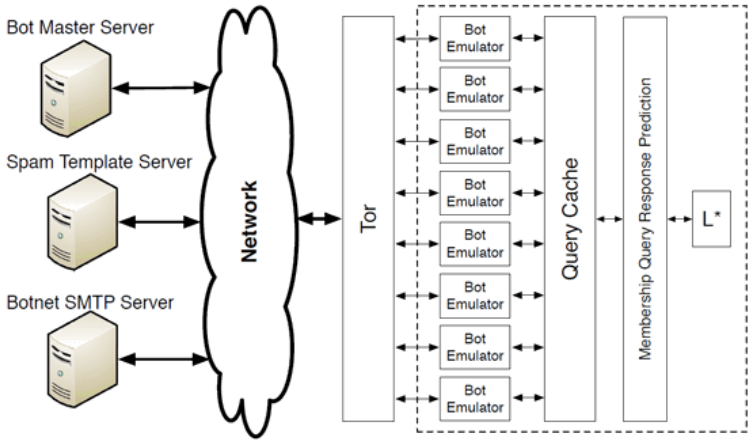

FIGURE 2.13 – Architecture de la plateforme d'inférence dans [Cho 2010]

L'expérimentation a pris près de 3 semaines pour obtenir un modèle à 17 états.

Les travaux présentés dans [Cho 2010] ont permis de montrer que l'inférence de modèle pouvait produire un modèle utile dans le contexte de la sécurité. Ils ont montré que le modèle pouvait être utilisé pour identifier des vulnérabilités dans le protocole, mais aussi d'identifier des différences entre différentes versions du même protocole.

2.2.2.4 Scanner de vulnérabilité en boite noire

En plus de l'algorithme d'Angluin, d'autres algorithmes en boite noire et basés sur des heuristiques ont été développés pour l'inférence d'application Web et la détection de vulnérabilités. Dans cette section, nous présentons les travaux de [Doupé 2012] qui ont pour but la détection de vulnérabilités.

Dans [Doupé 2010], l'auteur avait déjà étudié les performances des scanners de vulnérabilité et il avait montré que le seul fait de parcourir l'application page par page était trop limité et que les scanners avaient besoin d'avoir plus d'informations sur les systèmes pour pouvoir trouver plus de vulnérabilités.

Dans [Doupé 2012], il décide de créer un scanner de vulnérabilité qui tient compte de l'état du système. Il y a donc une étape d'inférence de modèle en parallèle de la détection de vulnérabilité. À la différence des algorithmes basés sur [Angluin 1987], seul un modèle partiel est construit et il est ensuite utilisé par une étape de frelatage (fuzzing) pour la découverte de vulnérabilités. L'application Web est modélisée sous forme d'une machine de Mealy symbolique définie dans [Berg 2008].

Pour pouvoir inférer un modèle de l'application, il faut pouvoir distinguer que deux états sont différents ou au moins que l'état courant à changé. Comme dans les algorithmes basés sur [Angluin 1987] qui se basent sur les différentes réponses pour distinguer les états, la méthode définie dans [Doupé 2010] effectue une requête envoyée précédemment et vérifie que la réponse est la même ou non (le système est supposé être déterministe). La différence se situe dans le fait que les requêtes ne se font pas sur l'ensemble des entrées, mais une partie seulement.

Le scanner va commencer par parcourir l'application aléatoirement en faisant des requêtes selon les entrées apprises lors des réponses précédentes. Chaque réponse HTML est transformée en arbre de page abstrait contenant les différents liens et formulaires ainsi que les paramètres. L'algorithme suppose que les états n'ont pas changé. Si une entrée déjà parcourue est rencontrée, l'algorithme s'en sert pour savoir si l'état a changé ou pas. Néanmoins pour construire un modèle, il faut être capable de localiser l'action qui a provoqué le changement d'état, mais aussi de grouper les états similaires puisque seuls les changements d'état sont détectés.

La détection de la requête qui a provoqué le changement d'état se fait en utilisant une heuristique. Quand un changement d'état est détecté, il est évident qu'une des requêtes précédentes en est responsable. L'algorithme considère en premier les requêtes les plus récentes, les requêtes de type POST plutôt que GET et aussi le fait

qu'une requête a pu déjà être utilisée pour changer d'état. Ces différentes propriétés sont utilisées dans une fonction de score qui permet de déterminer la requête qui a le plus de chance d'avoir changé l'état.

Le regroupement d'états est réduit au problème de coloration de graphe sur un graphe non planaire [Jensen 1995]. En partant du graphe obtenu pendant le parcours de l'application, deux nœuds a et b qui sont connectés sont deux états différents si au moins une des deux conditions suivantes est vraie :

— Une requête R a été effectué dans a et b et les arbres de page abstraits sont différents.
— Les deux nœuds a et b n'ont aucune page en commun.

À la fin de la coloration, tous les nœuds possédant la même couleur font partie du même état. Les résultats montrent que la couverture de l'application a été améliorée comparée aux outils de parcours de site Web classiques (wget) et d'autres scanners (w3af, skipfish).

Une des limitations vient du support de la technologie AJAX qui est supportée partiellement par la plate-forme HtmlUnit [Bowler 2002]. D'autres techniques tels que dans [Mesbah 2008] visent à convertir un site Web dynamique avec utilisation de requêtes AJAX en site Web statique et certaine [Choudhary 2013b] sont spécialement conçues pour les applications Web 2.0.

D'autres approches d'inférence en boite noire similaires à celle de [Doupé 2010] se concentrent sur la détection de vulnérabilité XSS [Duchene 2013] ou encore sur la modélisation des mécanismes de contrôle d'accès [Li 2014].

Chapitre 3

Inférence d'automates à état fini

Ce chapitre présente deux algorithmes d'inférence développés en collaboration avec *Keqin Li* et *Alexandre Petrenko*. Ils ont été utilisés dans le projet SPaCIoS pour générer des modèles en $ASLan++$ ainsi que sur des fournisseurs de service SIP sur Internet.

Sommaire

3.1 Inférence d'automates avec variables et paramètres non déterministes

Les applications Web utilisent un système de requête/réponse principalement basé sur le protocole HTTP donc les considérer comme des systèmes avec entrées et sorties puis les modéliser sous forme d'une machine de Mealy semble être une bonne solution. Ce type de modélisation se concentre surtout sur la partie contrôle de l'application en laissant de côté la partie données qui a pourtant au moins autant d'importance : dans le cas des applications Web, les requêtes et réponses contiennent

des paramètres qui peuvent influer sur le flot de l'application. Et en plus de simples chaînes de caractères, les paramètres peuvent être définis sur des domaines plus complexes tels que les dates ou des structures de données sérialisées. Plusieurs approches ont été proposées précédemment pour inférer des machines de Mealy ([Niese 2003], [Shahbaz 2009]) ou des automates finis paramétrés ([Li 2006c], [Berg 2006]).

En considérant que les entrées/sorties sont paramétrées et que chaque état a la possibilité d'accéder et de modifier un ensemble de variables, nous pouvons réduire la taille du modèle inféré tout en exprimant plus précisément les relations entre les entrées/sorties et leurs paramètres.

La sécurité des applications Web passe par l'utilisation de protocoles de sécurité qui nécessitent de générer puis utiliser des valeurs non déterministes. Ces valeurs ne doivent être utilisées qu'une seule fois par opération et ne doivent pas pouvoir être déduites de précédentes entrées ou sorties. Appelées *nonces*, elles sont en général générées de façon pseudo-aléatoire côté client ou serveur pour par exemple signer ou chiffrer cryptographiquement des communications ([Zenner 2009]) ou encore prévenir les attaques par rejeu ([Syverson 1994]). Pour les applications Web, les nonces sont générés côté serveur et se retrouvent dans les cookies du client pour représenter des identificateurs de session.
Avoir la possibilité de détecter la génération de nonces ainsi que leur valeur permet de les utiliser pendant l'inférence pour améliorer la couverture et la précision du modèle inféré.

La méthode présentée dans cette section permet d'inférer une machine à états finie déterministe avec des *entrées et sorties paramétrées* et associées à un ensemble de *variables* dont la valeur peut être *non déterministe*. Contrairement aux méthodes utilisant des machines de Mealy, cette méthode utilise deux types d'états : les états de l'automate et les états des variables. Pour gérer les états des variables, nous distinguons deux types de table d'observation, la *table de contrôle* qui enregistre les relations entre les symboles d'entrées et de sortie (équivalant à la table d'observation pour les machines de Mealy), mais aussi une *table de données* qui conservera les relations entre les paramètres des symboles d'entrées, les variables et les paramètres des symboles de sortie.

3.1.1 Test par apprentissage

Pour trouver des erreurs dans une application, nous utilisons une méthode de test basée sur de l'apprentissage. C'est une méthode itérative qui commence par l'apprentissage d'un premier modèle de l'application, ce modèle n'est généralement qu'une approximation du modèle exact. Bien que partiellement complet, ce modèle permet quand même de générer des cas de test. Ceux-ci sont ensuite exécutés sur le modèle ainsi que sur l'application pour obtenir une trace et vérifier que le comportement du modèle est bien équivalent à celui de l'application. Une différence dans

les traces peut indiquer soit une erreur dans le modèle et dans ce cas le modèle est complété par cette trace, soit une erreur dans l'application et dans ce cas le testeur peut ensuite approfondir les tests pour en trouver la cause. Par cette approche itérative, nous évitons un grand nombre de cas de test basiques qui n'auraient pas aidé à trouver des erreurs et nous nous concentrons sur des fautes plus subtiles à trouver.

Étant basée sur l'algorithme d'Angluin ([Angluin 1987]), la méthode présentée ci-après a besoin d'un oracle pour pouvoir à la fois exécuter des cas de test pour la construction du modèle, mais aussi pour comparer la conjecture à l'application. Pour cette comparaison, il faut pouvoir exhiber les contre-exemples, mais dans la pratique un tel oracle n'est pas envisageable. C'est pourquoi il est ici remplacé par les réponses de l'application. Le raffinement du modèle inféré s'effectuera tant qu'il existe au moins un cas de test qui produit une trace différente sur le système et sur le modèle jusqu'à qu'une convergence des deux ait lieu.

3.1.2 Machine à états finie étendue

On s'intéresse aux machines à états finies étendues avec des paramètres pour chaque symbole et un ensemble de variables pouvant être utilisé sous forme de garde pour sélectionner la bonne transition à effectuer, mais aussi pour définir la sortie ou ses paramètres. Après analyse de différentes applications Web et notamment de leur partie sécurité, nous pouvons voir que des valeurs de certains paramètres d'entrée peuvent être utilisées dans la partie contrôle de l'application. Il est donc nécessaire de garder ces valeurs en mémoire. Quant aux paramètres de sorties, leurs valeurs peuvent être nécessaires pour de futures entrées. Là aussi, nous devons garder une trace de ces valeurs.

Nous faisons deux suppositions en ce qui concerne la gestion de ces variables :
- il y a autant de variables que de paramètres (une variable d'automate associée à chaque paramètre I/O) ;
- seule la dernière valeur des paramètres I/O sera conservée. L'affectation d'une variable ne se fera qu'au moment où la transition est utilisée.

Ces suppositions permettent de guider l'apprentissage et de restreindre la classe des modèles à considérer. De plus, on montrera qu'elles sont réalistes dans le cadre des applications Web. Une fois ces deux suppositions établies, nous pouvons utiliser une restriction du modèle d'EFSM défini dans [Petrenko 2004b] et l'étendre avec le concept de paramètre de sortie non déterministe.

Dans les définitions suivantes, soit X et Y des ensembles finis de symboles d'entrée et de sortie, P et V des ensembles finis disjoints de n paramètres et variables tels que $|P| = |V| = n$. On peut poser $P = \{p_1, \ldots, p_n\}$ et $V = \{v_1, \ldots, v_n\}$, ainsi nous pouvons associer à chaque paramètre une variable de même indice. Pour $z \in$

$X \cup Y$, nous notons $p(z) \subseteq P$ l'ensemble de ses paramètres associés et D_z le domaine des valuations des paramètres dans l'ensemble $p(z)$. Nous étendons ces notations aux séquences σ d'entrées et de sorties donc $p(\sigma)$ est une séquence d'ensemble de paramètres, et D_σ est une séquence de valeurs de paramètre. De même, D_V est un ensemble de valeurs des variables V. Nous avons donc bien nos entrées et sorties qui possèdent chacune un ensemble fini de paramètres qui eux-mêmes possèdent un ensemble infini de valeurs. De cette façon, nous avons un modèle bien plus expressif que les machines de Mealy et il devient plus facile de modéliser les entrées et sorties d'un système.

Une machine à états finie étendue (EFSM) M, définie sur X, Y, P, V et la fonction associée p, est un couple (S, T) d'un ensemble fini d'états S, qui contient l'état initial appelé s_0, et un ensemble fini de transitions T entre les états dans S, tels que chaque transition $t \in T$ est un tuple (s, x, G, op, y, up, s'), où :

- $s, s' \in S$ sont respectivement l'état initial et final de la transition ;
- $x \in X$ est le symbole d'entrée de la transition ;
- $y \in Y$ est le symbole de sortie de la transition ;
- G, op, et up sont les fonctions, définies sur les paramètres d'entrée et les variables V, telles que,
  - $G : D_x \times D_V \to \{True, False\}$ est la garde de la transition ;
  - $op : D_x \times D_V \to D_y$ est le paramètre de sortie de la transition ;
  - $up : D_x \times D_V \to D_V$ est la fonction de mise à jour des variables de la transition.

Dans un état $s \in S$, tous les symboles d'entrée ne sont pas forcément valides et acceptés par l'EFSM. Soit un symbole d'entrée $x \in X$ qui n'est pas accepté par l'EFSM, nous utilisons une transition spéciale avec comme symbole de sortie $\Omega \in Y$ pour représenter le fait que ce symbole d'entrée n'est pas accepté. Selon la définition, il est aussi possible qu'un symbole d'entrée ou de sortie n'ait pas de paramètres. Dans ce cas nous introduisons une autre notation spéciale pour représenter le fait que ce symbole n'ait pas de paramètres. Soit un symbole d'entrée ou de sortie $z \in X \cup Y$ qui ne possède pas de paramètres, c'est à dire, $p(z) = \{\}$, nous utilisons ω pour représenter $d_z(p(z))$.

Considérant les suppositions faites précédemment à propos des variables, soit d la valeur du paramètre x, v la valeur de la variable V, et $d' = op(d, v)$ la valeur des paramètres de y selon la fonction op, la fonction up met à jour les valeurs respectives des variables des paramètres x et y à d et d', mais conserve la valeur des autres variables. Si x et y partagent un paramètre commun, la variable correspondante est elle aussi mise à jour avec la valeur du paramètre de sortie. Formellement, pour $d \in D_x$, $v \in D_V$, alors pour toutes les variables v_i :

- si $p_i \in p(y)$ alors $up(d, v)(v_i) = op(d, v)(p_i)$;
- sinon si $p_i \in p(x)$ alors $up(d, v)(v_i) = d(p_i)$;

— sinon $up(d, v)(v_i) = v(v_i)$;
(dans ce cas la valeur de v_i n'est pas modifiée).

Nous définissons ensuite une *entrée paramétrée*, c'est-à-dire un symbole d'entrée avec son ensemble de paramètres qui ont chacun une valeur, de la façon suivante : $x(d(p(x)))$ où x est le paramètre d'entrée, et $d(p(x))$ les valeurs pour chaque paramètre de x. Une *séquence d'entrées paramétrées* est une liste constituée d'entrées paramétrées. De la même façon, nous définissons une *sortie paramétrée* ainsi qu'une *séquence de sorties paramétrées*.

Selon la définition de la fonction *op* de l'EFSM, les valeurs des paramètres de sortie dépendent uniquement des valeurs des paramètres de l'entrée ainsi que des variables qui représentent globalement l'état interne du système. Mais nous avons vu aussi que dans le contexte de la sécurité, certains paramètres de sortie (générés côté serveur) pouvaient être non déterministes et donc indépendants des entrées et du système : ce sont les nonces. Ce type de paramètre rend la définition de la fonction *op* invalide, c'est pourquoi, et afin de rester cohérent avec la définition, nous introduisons pour les paramètres une nouvelle valeur, notée *ndv*, qui représente ces valeurs non déterministes. De cette façon, *op* reste une fonction. Nous verrons par la suite comment détecter et se servir de ce genre de valeur durant l'inférence.

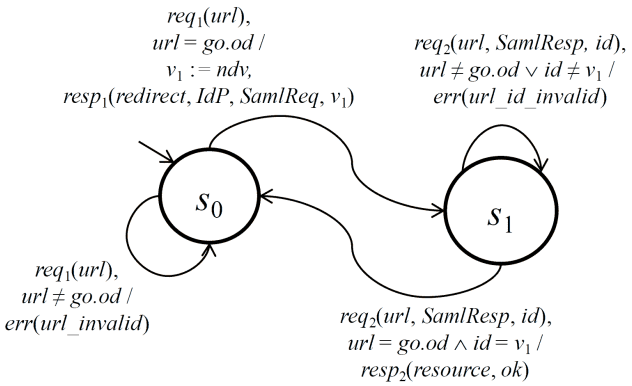

FIGURE 3.1 – Exemple d'EFSM représentant SAML

Un exemple d'EFSM est donné dans la Figure 3.1. Certains éléments des transitions ont été omis pour des questions de lisibilité. Cet EFSM représente le *Service Provider* (SP), un des trois composants du protocole d'authentification et d'autorisation SAML [Consortium 2008] (Security Assertion Markup Language) qui est un standard de l'OASIS (Organization for the Advancement of Structured Information Standards). Dans ce protocole, l'*User Agent* (UA) représente l'utilisateur qui souhaite accéder à une ressource du SP. En premier, l'UA demande directement la ressource au SP, mais comme l'UA n'a pas encore prouvé qu'il est autorisé à accéder à cette ressource, le SP renvoie à l'UA une requête d'authentification (*SAMLRequest*) contenant des informations sur la ressource, le SP ainsi qu'un nonce identifiant la

requête puis le redirige vers le troisième composant qui est l'*Identity Provider* (IP) : le composant qui connaît les utilisateurs et ce à quoi ils peuvent accéder. L'IP demande ensuite à l'UA des informations pour s'authentifier, par exemple sous forme d'un couple nom d'utilisateur/mot de passe et génère ensuite une réponse, appelé *assertion d'authentification* (*SAMLResponse*), qui est donnée ensuite au SP par l'UA. Si cette réponse est positive et que le nonce présent dans la réponse est bien le même que celui de la requête, l'UA peut accéder à la ressource.

Dans l'état S_0, le SP vérifie qu'il possède bien la ressource et génère un *ndv* qu'il renvoie à l'UA. La procédure d'authentification n'est pas présente puisqu'elle concerne uniquement l'UA et l'IP. Puis dans l'état S_1, le SP vérifie la réponse ainsi que le *ndv* pour ensuite renvoyer la ressource à l'UA.

3.1.3 Méthode d'inférence pour EFSM

L'algorithme d'inférence 1 suivant peut être considéré comme une extension de l'algorithme L_M* [Li 2006c] qui est lui-même une extension de l'algorithme d'Angluin ([Angluin 1987]). La partie qui concerne la gestion des contre-exemples est présentée dans la sous-sous-section 3.1.3.4. Quant aux tables de contrôle et de données, ce sont des extensions de la table d'observation classique dans ce type d'algorithme et elles seront définies formellement dans la section suivante.

Algorithme 1 : Inference_EFSM

1 Initialisation des tables de contrôle et de données avec $S = \varepsilon$, $E = X$, et R obtenu en associant chaque symbole d'entrée avec chaque valeur de paramètre (si le symbole en possède);
2 Construction des séquences d'entrées paramétrées et test sur le système à inférer;
3 Enregistrement des observations dans les tables de contrôle et de données;
4 Si un *ndv* est identifié, construire et appliquer les nouvelles séquences d'entrées paramétrées utilisant la valeur du *ndv*;
5 **while** *la table de contrôle n'est pas équilibrée, précisée et fermée* **do**
6 équilibrer la table, c'est à dire, pour tout $s, t \in S \cup R$, s et t sont équilibré;
7 éliminer les ambiguïtés de la table, c'est à dire, toutes les lignes $s \in S$ doivent être précisées;
8 rendre la table fermée, c'est à dire, pour chaque $t \in R$, il existe une ligne $s \in S$ telle que $s \cong t$;
9 **end**
10 Construire la conjecture à partir des tables de contrôle et de données;

3.1.3.1 Table de contrôle et de données

Pour l'inférence d'EFSM, nous utilisons deux types de tables. Cela permet de séparer proprement les relations sur les symboles et les paramètres. La table de contrôle contient les symboles d'entrée et de sortie qui sont en grande partie responsables de la partie contrôle tandis que la table des données contient les paramètres d'entrée, de sortie et les variables utilisées pour les gardes et l'identification de *ndv*. Bien que les informations stockées soient de nature différente, leur structure est bien la même.

Soit U l'ensemble de toutes les combinaisons de valeurs d'entrées paramétrées possibles. La structure de la table de contrôle (S, R, E, C) et de la table de données (S, R, E, D) sont définies de la façon suivante :

— $S \subseteq U$ et $R \subseteq U$ sont des ensembles finis non vides d'entrées paramétrées qui constituent les lignes des tables. S est un ensemble fermé par préfixe qui identifie les états potentiels dans la conjecture et R est utilisé pour explorer le système d'une transition suivante.

— $E \subseteq X^*$ sont les colonnes des tables et elles sont utilisées pour distinguer les états potentiels de la conjecture. Notons que dans la procédure d'inférence décrite plus haut, E est initialement égal à X. Mais plus généralement, un élément de E pourrait être une séquence.

Dans la table de contrôle, chaque cellule $C(s, x)$ indexée par $s \in S \cup R$ et $x \in E$ est un ensemble d'éléments de la forme $(d(p(x)), y)$, tandis que dans la table des données, chaque cellule $D(s, x)$ est un élément de la forme de $(d(p(x)), v(V) \rightarrow d'(p(y)))$, où $d \in D_x$, $y \in Y$, $v \in D_V$, et $d' \in D_y$. Pendant la procédure d'inférence, la valeur de la variable est automatiquement mise à jour selon la dernière valeur des ses paramètres d'entrées et de sortie.

Au début de la procédure d'inférence, S, E, et R sont initialisées de la manière suivante : $S = \{\varepsilon\}$, $E = X$, R est un ensemble de $|X|$ entrées paramétrées obtenues en associant chaque symbole d'entrée avec une valeur par paramètre. S et R seront progressivement étendus durant l'inférence.

Pour illustrer cette construction, nous avons dans la Figure 3.2 les tables de contrôle ainsi que de données du *Service Provider* de SAML. Dans cette figure, la notation Ω est utilisée pour représenter une entrée qui n'a pas été acceptée par l'application, celle-ci ayant renvoyé une erreur ou même aucune réponse. Quant au symbole $\perp$, il représente la valeur initiale des variables quand aucune transition n'a utilisé le paramètre associé.

Nous avons vu dans l'algorithme que trois critères étaient requis pour pouvoir construire une conjecture à partir des tables. Ces tables doivent être :

— *Fermées* : cette propriété est similaire à celle de L* avec les tables d'observations. Elle permet de s'assurer qu'un état potentiel découvert dans R est bien

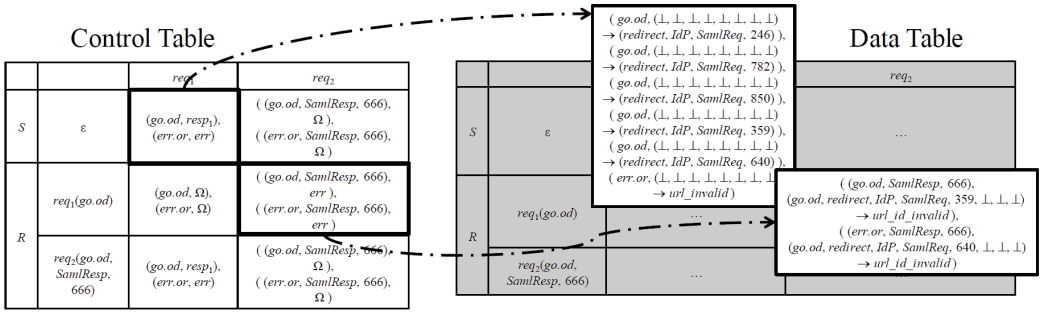

FIGURE 3.2 – Extrait des tables de contrôle et de données pour SAML SP

considéré en tant que état potentiel et exploré comme il se doit. À chaque fois que R contient une ligne dont les symboles de sortie pour chaque entrée n'a pas d'équivalent dans S, cette ligne sera déplacée dans S puis R sera étendu pour explorer une transition suivante. Formellement, la table de contrôle est dite *fermée*, si pour chaque $t \in R$, il existe un $s \in S$ tel que $s \cong t$. Pour chaque $t \in R$ qui empêcherait de rendre la table fermée, nous déplaçons t de R vers S et étendons R avec $|X|$ chaînes $\{t \cdot x_i(d_i(p(x_i)))|d_i \in D_{x_i}, 1 \leq i \leq k\}$ en notant x_i les éléments de X.

— *Équilibrées* : cette propriété s'assure que chaque état ou état potentiel a été exploré de la même façon, avec les mêmes entrées et mêmes valeurs de paramètres. Cette étape est nécessaire puisqu'elle permet de rendre la comparaison entre les différents états possible. Elle permet aussi de propager automatiquement les valeurs des *ndv* à travers chaque ligne et de découvrir de nouveaux comportements. Formellement, les lignes s_1 et s_2 sont dites *équilibrées*, si $C(s_1, x)$ et $C(s_2, x)$ contiennent les mêmes valeurs de paramètre pour chaque $x \in E$. Une table est dite *équilibrée*, si pour tout $s, t \in S \cup R$, s et t sont équilibrés. Quand deux lignes s_1 et s_2 ne sont pas équilibrées, il suffit d'appliquer les valeurs manquantes aux différents paramètres pour les rendre à nouveau équilibrées.

— *Non ambiguës* : pour les EFSM, il est possible d'obtenir un symbole de sortie différent même si on a utilisé la même séquence d'entrées paramétrées puis le même symbole d'entrée. En effet, les paramètres du symbole d'entrée peuvent aussi influer sur le symbole de sortie. Dans ce cas, la table de contrôle se retrouve avec une cellule contenant un seul symbole d'entrée et plusieurs symboles de sortie ce qui rendrait la conjecture non déterministe. Nous devons donc explorer séparément la transition avec le symbole d'entrée et chaque valeur de paramètres. Formellement, dans la table de contrôle, une ligne $s \in S$ est *ambiguë*, s'il existe un $x \in E$ tels qu'au moins deux symboles de sortie se trouvent dans la cellule $C(s, x)$ indexée par s et x. Une ligne ambiguë est *précisée* si pour chaque $y \in Y$ se trouvant dans la cellule $C(s, x)$, il existe

$t = s \cdot x(d(p(x))) \in R$, tels que $(d(p(x)), y) \in C(s, x)$. Pour créer les transitions de la conjecture, quand une ligne ambiguë s n'est pas précisée parce qu'il n'existe pas un tel t dans R pour un certain y, nous étendons R avec t. Une table de contrôle est dite *précisée* si toutes les lignes dans S sont elles même précisées.

Deux lignes s_1 et s_2 sont dites *équivalentes*, noté $s_1 \cong s_2$, si elles sont équilibrées et qu'elles contiennent le même ensemble de symboles de sortie pour tout $x \in E$. Pour les lignes $s \in S \cup R$, nous notons $[s]$ la classe d'équivalence des lignes qui inclut s.

3.1.3.2 Identification et utilisation des *ndv*

L'application est considérée comment étant déterministe : pour la même séquence d'entrées paramétrées avec les mêmes valeurs de paramètres, les sorties paramétrées devraient être exactement les mêmes. Mais dans le cas de paramètres non déterministes, la même séquence d'entrée paramétrée peut produire le même symbole de sortie, mais avec une ou plusieurs valeurs de paramètres qui seront différentes. Dans ce cas nous pouvons dire que ces paramètres ne dépendent pas des entrées, ce sont donc des *ndv*. Plus formellement, si pour une même cellule $D(s, x)$, deux des éléments de cette cellule ont les mêmes valeurs de paramètres d'entrées $d(p(x))$ ainsi que les mêmes valeurs de variables, mais qu'un ou plusieurs paramètres ont des valeurs différentes alors ces paramètres sont des *ndv*.

Nous avons donc identifié un paramètre non déterministe après avoir exécuté une certaine séquence d'entrée. Nous pouvons raisonnablement supposer que sa valeur doit servir à une prochaine entrée pour déclencher un comportement particulier. Il faut donc tester cette valeur pour toutes les séquences d'entrée ayant le même préfixe, et ce pour chacun des paramètres. En pratique, nous avons juste à nous occuper de l'entrée sur lequel le *ndv* a été détecté. Les autres tests sont assurés par la propriété d'équilibre de la table de contrôle qui spécifie que chaque paramètre d'entrée doit être testé avec le même ensemble de valeurs.

Les modifications sur les tables sont les suivantes :

— Dans la cellule de la table de données, nous fusionnons les éléments qui n'ont que la valeur du paramètre non déterministe de différent. Ces valeurs sont remplacées par un identificateur *ndv*.

— Pour toutes les prochaines séquences d'entrée qui ont $s \cdot x(d(p(x)))$ comme préfixe et pour tout leurs paramètres, nous utiliserons la valeur courante du *ndv* en plus des valeurs existantes du paramètre. En ce qui concerne l'enregistrement des observations, les valeurs concrètes du paramètre seront remplacées par l'identificateur ndv_i, dans lequel i représente l'index du paramètre de sortie étant un *ndv*.

3.1.3.3 Construction de la conjecture

Une fois que les tables respectent les propriétés d'équilibre, non-ambiguïté et fermeture, nous pouvons construire une conjecture du système $Q = (S_Q, T_Q)$ à partir de ces tables. Deux étapes distinctes sont nécessaires pour créer cette conjecture, la première se sert de la table de contrôle C et crée un EFSM partiel proche des machines de Mealy avec entrées/sorties seulement. Ensuite, la seconde étape utilise la table des données D pour générer les gardes, variables et fonctions de sortie de l'automate grâce à des algorithmes de fouille de données. Nous obtenons à la fin un EFSM.

Construction de l'EFSM partiel Cette étape construit un EFSM partiel de $Q = (S_Q, T_Q)$. Les classes d'équivalence des séquences d'entrées dans S deviennent les états dans Q, c'est-à-dire $S_Q = \{[s] | s \in S\}$ avec $s_{0Q} = [\varepsilon]$ l'état initial.

L'ensemble des transitions T_Q est défini tel que pour chaque $[s]$ avec $(s \in S)$, $x \in X$, une ou plusieurs transitions sont définies, suivant le nombre de symboles de sortie différents dans la cellule $C(s, x)$. Supposons qu'un de ces symboles de sortie soit y, la transition est $([s], x, G, op, y, up, s')$, dans laquelle :

— Quand plusieurs symboles de sortie sont dans une même cellule $C(s, x)$, une garde G est ajoutée comme ceci : dans chaque cellule de la table des données $D(t, x)$ $(t \in [s])$, il y a un ensemble d'éléments $(d(p(x)), v(V) \rightarrow d'(p(y)))$ qui correspond au symbole de sortie y, et $(d(p(x)), v(V) \rightarrow True)$ respectivement. Pour tous les autres éléments, nous construisons $(d(p(x))$, $v(V)$ $\rightarrow False)$ respectivement ; la garde G est composée de l'ensemble de ces éléments construits ;

— Dans chaque cellule de la table des données $D(t, x)$ $(t \in [s])$, il y a un ensemble d'éléments correspondant au symbole de sortie y. La fonction pour les paramètres de sortie op est l'ensemble contenant ces éléments ;

— La fonction de mise à jour des variables up est définie comme dans la définition des EFSM. Cette fonction remplace la valeur de la variable associée à chaque paramètre à chaque fois qu'une nouvelle valeur est appliquée ;

— Comme les tables d'observation sont non-ambiguës, il existe $t = s{\cdot}x(d(p(x)))$, $t \in S \cup R$, tel que $(d(p(x)), y) \in C(s, x)$. Nous définissons l'état cible $s' = [t]$.

En plus de ces transitions logiques, s'il existe un élément (ndv_i, y) dans une cellule $C(s, x)$ de la table de contrôle, nous devons prendre en compte le ndv de cette façon :

— S'il existe déjà une transition de $[s]$ avec le symbole de sortie y, nous ajoutons

"$\vee p(x) = v_i$" dans la garde de la transition.

— Sinon, nous créons une transition avec comme garde "$p(x) = v_i$" ; puis, dans la garde de toutes les autres transitions depuis $[s]$, nous ajoutons "$\wedge p(x) \neq v_i$".

Inférence des gardes, variables et fonctions de sortie Pour le moment, les gardes et les fonctions associées aux transitions ne sont représentées que par des ensembles de relations. Ces relations proviennent directement de la table de données et sont donc sous cette forme $d(p(x)), v(V) \rightarrow d'(p(y))$. Si elles sont utiles pour avoir une idée des gardes et fonctions, elles ne sont valides que pour les valeurs qui ont été testées durant l'inférence et du point de vue du développeur ou du testeur, elles sont plus difficiles à appréhender et pas adaptées pour un modèle.

Nous généralisons ces ensembles pour obtenir des gardes sous forme normale conjonctive et des fonctions plus compactes en utilisant des outils de fouilles de données tels que Daikon [Ernst 2001] ou Weka [Witten 2011] qui contiennent des algorithmes de classification et de régression.

3.1.3.4 Gestion des nouvelles observations

Dans l'algorithme d'Angluin original, un oracle est utilisé pour déterminer si la conjecture est équivalente au SUI (système sous inférence). Dans le cas où une différence apparaît, l'oracle répond avec la séquence d'entrées qui produit une différence de comportement, appelé contre-exemple. Cette séquence produit toujours à un nouvel état dans la conjecture suivante.

Dans ce nouvel algorithme pour EFSM, le concept de contre-exemple est étendu à une nouvelle observation qui est une paire composée d'une séquence d'entrée paramétrée et de la séquence de sortie paramétrée correspondante obtenue sur le SUI. Cette séquence de sortie n'a jamais été exécutée pendant la procédure d'inférence sinon la conjecture contiendrait déjà cette observation. Par contre, le fait de considérer cette nouvelle observation ne conduit pas forcément à l'ajout d'un nouvel état. En effet, dans cette version, une nouvelle observation peut aussi compléter les gardes et fonctions.

Supposons que la séquence de symboles d'entrée de la nouvelle observation est $\alpha \in X^*$, la séquence d'entrée paramétrée est $\alpha(d(p(\alpha)))$, la séquence de symboles de sortie $\beta \in Y^*$ et la séquence de sortie paramétrée est $\beta(d(p(\beta)))$. Notons $\mathit{suffix}^j(\alpha)$ le suffixe de α de longueur j.
En adaptant la méthode appelée "Suffix1by1" et décrite dans [Irfan 2010b], la méthode de gestions des nouvelles observations peut se définir de cette façon :

1. Diviser α comme $\gamma \cdot \sigma$, où γ est le plus long préfixe de α tel qu'il existe $d'(p(\gamma)) \in D_\gamma$ et $\gamma(d'(p(\gamma))) \in S \cup R$;

2. Ajouter $\gamma(d(p(\gamma)))$ dans $S \cup R$ s'il n'est pas déjà présent ;
3. Ajouter les suffixes de σ un par un en partant de celui de longueur 1 jusqu'à E ;
4. Compléter les tables d'observations et rendre la table équilibrée ;
5. Si la table de contrôle n'est pas fermée, arrêter d'ajouter les suffixes et rendre la table fermée, équilibrée et non-ambiguë ;
6. Produire une nouvelle conjecture ;

3.1.4 Expérimentation

Nous avons implémenté notre algorithme d'inférence en Java et nous l'avons appliqué sur diverses applications Web et protocole de sécurité tels que WebGoat (`http://code.google.com/p/webgoat/`), une application contenant volontairement des failles de sécurité, et le protocole de sécurité SAML défini dans ([Consortium 2008]). De plus, pour évaluer les performances de l'algorithme, nous l'avons utilisé pour inférer des systèmes générés aléatoirement. Nous avons effectué des mesures en faisant varier la complexité et la taille du système grâce au nombre d'états, d'entrées et gardes.

Nous voulons ici créer des systèmes avec gardes et variables. Les configurations des expérimentations sont décrites ci-dessous :

— La taille du système à inférer, le nombre d'états et le nombre d'entrées sont configurables. Les valeurs des paramètres sont définies uniquement sur les entiers naturels, plus précisément, et ce pour mieux observer les différents types de paramètres, les valeurs simples sont comprises entre 0 et 1000 tandis que les valeurs des paramètres non déterministes sont comprises entre 10000 et plus. En utilisant différentes plages de valeurs pour chaque type de paramètre, il est plus simple de suivre leur utilisations et de détecter les éventuelles problèmes. Les systèmes générés sont compatibles avec les suppositions nécessaires pour l'algorithme.

— Chaque couple d'état est ensuite parcouru et une transition est créee selon le pourcentage de transitions spécifié dans les options de la génération. Par exemple, si le pourcentage est défini à 50, pour chaque couple d'état, il y a une chance sur deux qu'une transition soit créée. Si une transition doit être ajoutée, une entrée et une sortie y sont associées. Une dernière étape consiste à détecter les états qui ne contiennent pas de transition sortante ainsi que ceux qui ne sont atteignables. L'automate est corrigé en ajoutant les transitions nécessaires. Chaque paramètre est défini par un minimum et un maximum.

— Nous avons défini deux types de gardes : "simple" et "variable". La proportion de ces gardes est aussi configurable. Une garde dite *simple* est une conjonction d'inégalités ou d'égalités entre un paramètre et une valeur constante (par exemple $url = go.od$) tandis qu'une garde dite *variable* est une comparaison d'une valeur (qui peut être un *ndv*) et d'une valeur stockée précédemment dans une variable (par exemple $id = v_1$). Comme la détection et l'utilisation de *ndv* entraîne de nouvelles requêtes au système, le fait de contrôler ces types de gardes permettra de mesurer l'impact des *ndv*.

— Pour chaque expérimentation, nous avons un ensemble de données de test prédéfini, c'est à dire, les valeurs des paramètres d'entrée. C'est valeurs sont choisies pour que les conditions des gardes puissent être satisfaites avec au moins une combinaison de valeurs. De cette façon, nous simulons une stratégie de sélection des données qui pourrait être utilisée par le testeur.

— L'algorithme d'inférence est utilisé jusqu'à la première conjecture qui, dans la plupart des cas, est assez proche du système lui-même. La première conjecture est obtenue quand les tables sont fermées, équilibrées et non ambiguës. Cela permet d'avoir une bonne indication quant à la complexité nécessaire à l'inférence du système.

Ensuite, pour chaque cas, comme un nombre donné d'états du SUI, nous avons généré et inféré 50 de ces différents systèmes. Dans les graphiques suivants, nous avons mesuré le nombre de requêtes moyen nécessaire pour inférer la première conjecture. Voici les résultats :

— Pour un nombre d'états variant de 2 à 14, nous définissons $|X| = |Y| = 3$, 25% des transitions ont une garde *simple*, et 25% ont des gardes *variables*. L'influence du nombre d'états est présentée dans la Figure 3.3 ;

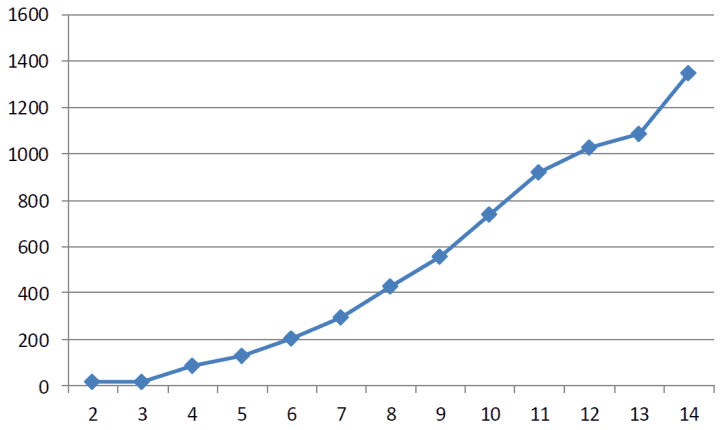

FIGURE 3.3 – Complexité en nombre de requêtes suivant le nombre d'états

— Pour un nombre de symboles d'entrée variant de 2 à 16, nous définissons le nombre d'états à 6 et $|Y| = 3$, 25% des transitions ont une garde *simple*, et 25% ont des gardes *variables*. L'impact du nombre de symboles d'entrée est présenté dans la Figure 3.4 ;

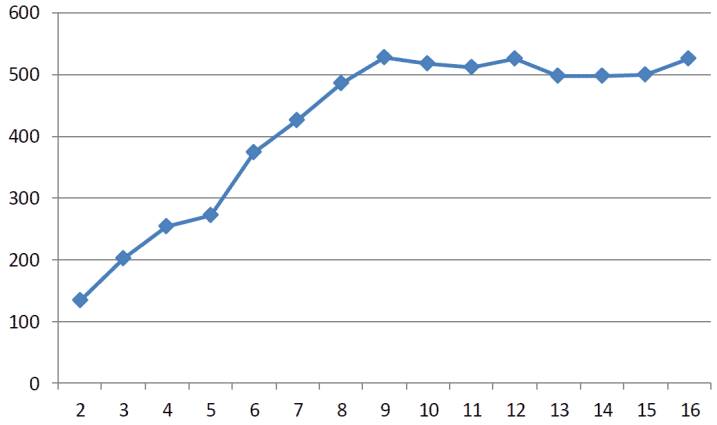

FIGURE 3.4 – Complexité en nombre de requêtes suivant le nombre de symboles d'entrée

— Pour des pourcentages de garde *simple* et *variable* allant de 0 à 100% , nous définissons $|X| = |Y| = 3$, et un nombre d'états à 10. Les résultats sont présentés dans la Figure 3.5.

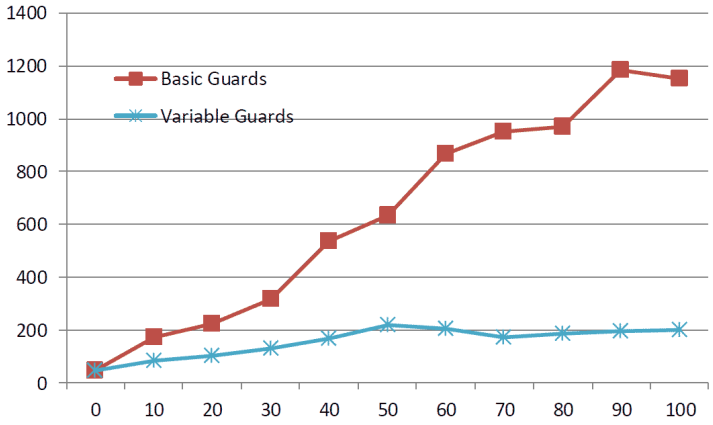

FIGURE 3.5 – Complexité en nombre de requêtes suivant le pourcentage de garde *simple* et *variable*

3.1.5 Avantages et inconvénients

Cet algorithme permet d'inférer un modèle d'un système sous forme d'EFSM. Ce type de modèle a été choisi pour mieux représenter les applications Web et plus généralement les services. Il est désormais plus facile de faire correspondre le système au modèle, chaque requête et réponse du système peut se traduire en entrées et sorties paramétrées. Les applications Web sont très consommatrices de requêtes dont la forme peut varier, ce qui nécessite d'être capable de gérer des entrées avec plusieurs dizaines de paramètres aussi bien que des entrées à un seul paramètre. De la même façon, les sorties et les paramètres eux-mêmes sont sujets à ces variations, nous pouvons rencontrer des paramètres entiers aussi bien que des données textuelles encodées et de grandes tailles. Le modèle EFSM correspond bien à ce type d'applications. L'algorithme d'inférence a lui aussi été adapté au contexte avec comme principales améliorations, par rapport à [Li 2006b], la détection des nonces. Qu'ils soient utilisés pour les cookies ou pour la détection de CSRF, les nonces sont présents dans la plupart des applications et pouvoir les détecter et les utiliser pour l'inférence doit améliorer le modèle inféré ainsi qu'aider à la détection de vulnérabilités.

Cependant même avec quelques améliorations, l'algorithme présente des limitations quand il est appliqué aux applications Web. La principale étant la création de l'adaptateur de test. Cet adaptateur est nécessaire pour permettre la communication entre l'inférence à haut niveau et l'application à bas niveau. L'écriture de cet adaptateur peut prendre un certain temps selon les applications et nécessiter certaines compétences en fonction des technologies utilisées par l'application. Le testeur doit identifier toutes les actions et les pages de l'application puis fournir les fonctions de traduction manuellement ce qui rend la méthode moins applicable automatiquement. De plus, une action peut avoir un grand nombre de paramètres et c'est au testeur de spécifier quels sont les paramètres à garder pour l'inférence. Utiliser tous les paramètres conduirait à allonger considérablement le temps d'exécution de l'algorithme et le nombre de requêtes envoyé au système. Même si une approche automatique de récupération des entrées est envisagée, par exemple avec un collecteur qui navigue dans l'application en suivant les liens, il se peut qu'il n'ait pas découvert toutes les actions possibles ni les pages. Les fonctions de traduction seraient alors incomplètes et le modèle produit perdrait en précision. Au niveau de l'algorithme lui-même, il peut y avoir un problème quand plusieurs *ndv* sont détectés. La stratégie à employer n'est pas évidente, si nous considérons toutes les combinaisons possibles, nous trouverons exactement à quoi sert le *ndv*, mais nous aurons en même temps un grand nombre de requêtes à envoyer. Il y a une dernière limitation dans la gestion des variables. Pour chaque paramètre, seule la dernière valeur de la variable est enregistrée, mais il se peut que certains protocoles utilisent une valeur précédente. La partie de l'inférence utilisant des outils de fouille de données pourrait tirer parti de ces valeurs précédentes et ainsi améliorer le modèle.

En conclusion, cet algorithme permet d'inférer un modèle adéquat au domaine des applications Web, mais la mise en pratique reste encore compliquée dans certains cas.

3.2 Inférence Z-Quotient

La méthode Z-Quotient permet d'inférer une machine à états finie (FSM) d'un système (un ou plusieurs de ces composants) en le testant directement. C'est une méthode de type boîte noire parce qu'elle ne nécessite pas d'avoir l'exécutable ou son code source pour fonctionner par contre, les composants doivent être accessibles (par exemple par Internet). Cette méthode n'utilise que les entrées et sorties du système [Aarts 2010a] ce qui correspond à une situation classique dans le test en boite noire où le testeur envoie une requête au système et sa réponse est analysée.

Pour cette méthode, nous supposons donc que le système peut être modélisé par une machine à états finie et uniquement à partir de ses entrées et sorties.

3.2.1 Z-Quotient

Il est possible de modéliser un système sous forme de FSM (le système est équivalent au FSM) à l'aide de séquences d'entrée. L'algorithme présenté dans cette partie diffère du précédent parce qu'il est basé sur la méthode de test d'automates de [Vasilevskii 1973], c'est-à-dire avec un ensemble de caractérisation noté ensemble-W, constitué de séquences d'entrées.

La notion de quotient initial d'un FSM associé à un ensemble-W partiel $Z \in W$ est à la base de cette approche. Z-quotient ne représente qu'une approximation du système dans lequel certains états ne pourront peut-être pas être distingués.

La plupart des méthodes d'inférence d'automates sont basées sur de l'inférence grammaticale avec oracle comme l'algorithme d'inférence d'EFSM (3.1.3). Notre algorithme peut utiliser le système comme un oracle de deux façons, en envoyant une séquence d'entrées pour obtenir la séquence de sortie correspondante et en envoyant une conjecture pour obtenir une réponse booléenne sur le fait que la conjecture est équivalente au système ou non. Dans ce dernier cas, l'oracle fournit le contre-exemple montrant la divergence entre le modèle et le système. Notre méthode va utiliser ce principe pour construire petit à petit l'ensemble-W correspondant au système à partir des contre-exemples.

Cette méthode se veut plus adaptée et efficace pour le test logiciel : même si elle est, comme les méthodes basées sur l'algorithme d'Angluin, une méthode dite *active*, elle peut néanmoins utiliser dès le départ un ensemble de traces, choisies par le testeur et ayant pour but d'accélérer l'inférence. Il est aussi possible de définir

dès le départ un ensemble-W partiel. Dans la pratique, cette méthode nécessite en moyenne moins de requêtes que les méthodes basées sur l'algorithme d'Angluin.

3.2.1.1 Définitions

Une machine à états finie (FSM) A est un 5-uplet (S, s_0, I, O, h_A), où :
— S est un ensemble fini d'états contenant l'état initial s_0 ;
— I et O sont des ensembles finis disjoints et non vides d'entrées et de sorties, respectivement ;
— h_A est la fonction de transition $h_A : S \times I \to 2^{S \times O}$, où $2^{S \times O}$ est l'ensemble des parties de $S \times O$.

Selon les propriétés de la fonction de transition, certains FSM particuliers peuvent être définis. Soit un FSM $A = (S, s_0, I, O, h_A)$, il est appelé :
— *trivial* si $h_A(s_0, a) = \varnothing, \forall (s_0, a) \in S \times I$;
— *complet* si $h_A(s, a) \neq \varnothing, \forall (s, a) \in S \times I$;
— *partiellement défini* (un FSM partiel) si $\exists (s, a) \in S \times I$ tel que $h_A(s, a) = \varnothing$;
— *déterministe* si $|h_A(s, a)| \leq 1, \forall (s, a) \in S \times I$;
— *non déterministe* si $\exists (s, a) \in S \times I$ tel que $|h_A(s, a)| > 1$;
— *observable* si l'automate $A_\times = (S, s_0, I \times O, \delta)$, où $\delta(s, ab) \ni s'$ si et seulement si $(s', b) \in h_A(s, a)$, est déterministe.

Nous considérons ici uniquement des machines observables. De plus, nous supposons que toutes les machines sont *connectées initialement*, c'est-à-dire chaque état est atteignable depuis l'état initial. Nous utiliserons a, b, c pour les symboles d'entrées et de sorties, α, β, γ pour les séquences d'entrées et de sorties, s, t, p, q pour les états et u, v, w pour les traces.

Étant donné un FSM $A = (S, s_0, I, O, h_A)$, voici quelques notations :
— (s_1, ab, s_2) est une transition si $s_1, s_2 \in S$ et $(s_2, b) \in h_A(s_1, a)$.
— Un chemin de l'état s_1 à s_{n+1} est une séquence de transitions $(s_1, a_1b_1, s_2)(s_2, a_2b_2, s_3) \dots (s_n, a_nb_n, s_{n+1})$ telles que $(s_{i+1}, b_i) \in h_A(s_i, a_i)$, où $1 \leq i \leq n$ et n est la longueur du chemin.
— Une séquence $u \in (I \times O)^*$ est appelée une *trace* de A dans l'état $s_1 \in S$, s'il existe un chemin $(s_1, a_1b_1, s_2)(s_2, a_2b_2, s_3) \dots (s_n, a_nb_n, s_{n+1})$ tel que $u = a_1b_1a_2b_2 \dots a_nb_n$. Remarque : une trace de A dans l'état s_0 est un mot de l'automate $A_\times$.
— Soit $inp(u) \subseteq I$ l'ensemble des entrées apparaissant dans une trace u. Nous notons $Tr(s)$ l'ensemble de toutes les traces de A dans l'état s et $Tr(A)$ l'ensemble des traces de A dans l'état initial.
— L'opérateur de projection $\downarrow_B$, qui projette des séquences dans $(I \times O)^*$ sur l'ensemble $B \subseteq I \cup O$, est récursivement définis comme étant $\varepsilon \downarrow_B = \varepsilon$, $(ua) \downarrow_B = u \downarrow_B a$ si $a \in B$, et $(ua) \downarrow_B = u \downarrow_B$ sinon, où $u \in (I \times O)^*$ et

$a \in I \cup O$.

— Étant donnée une séquence $u \in (I \times O)^*$, la séquence $u \downarrow_I$ est une projection de l'entrée de u. La séquence d'entrées $\alpha \in I^*$ est une séquence d'entrées définie dans l'état s de A s'il existe $u \in Tr(s)$ tel que $\alpha = u \downarrow_I$. Nous utilisons $\Omega(s)$ pour noter l'ensemble de toutes les séquences d'entrées définies pour l'état s.

— Étant donnés deux états $s, t \in S$ du FSM A et un ensemble de séquences d'entrées $Z \subseteq \Omega(s) \cap \Omega(t)$, s et t sont *Z-équivalent*, si pour tout $a \in Z$, $\{u \in Tr(s) \mid u \downarrow_I = a\} = \{u \in Tr(t) \mid u \downarrow_I = a\}$.
— Les états Z-équivalent sont *k-équivalent*, si Z contient toutes les séquences d'entrées de longueur k.
— Les états s et t sont équivalent s'ils sont Z-équivalents et $Z = \Omega(s) = \Omega(t)$, c'est-à-dire $Tr(s) = Tr(t)$.
— Les états s et t qui ne sont pas Z-équivalent sont *Z-distincts*.
— Une séquence d'entrées $a \in \Omega(s) \cap \Omega(t)$ telle que $\{u \in Tr(s) \mid u \downarrow_I = a\} \neq \{u \in Tr(t) \mid u \downarrow_I = a\}$ est appelée une séquence *discriminante* s et t.
— Les états s et t sont *(k-)distinct*, s'il existe une séquence qui les distingue (de longueur k).
— Un ensemble de séquences d'entrées Z tel que chaque paire d'états distincts est Z-distinct est appelé *ensemble de caractérisation* du FSM A. Un FSM complet qui n'a pas d'états équivalents est dit *minimal*.

Ces relations d'équivalence et de distinguabilité sur les états sont étendues aux états de différentes machines.

3.2.2 Z-Quotient initial

À partir de maintenant, nous supposons que le système sous inférence (SUI) se comporte comme un FSM auquel nous pouvons envoyer des entrées et en observer les réponses. En supposant que le système se comporte bien de façon déterministe, qu'il contient un nombre fini d'états et que son ensemble de caractérisation est connu, l'inférence peut être exécutée assez facilement. Pour cela, nous devons au minimum connaître un élément de l'alphabet d'entrée pour exécuter le premier test puis recevoir d'autres entrées grâce à l'oracle (le système).

L'ensemble de caractérisation sert essentiellement à différencier les états qui ne sont pas équivalents. Pour inférer un FSM du système, nous pouvons utiliser une version légèrement modifiée de la méthode-W [Vasilevskii 1973]. Cependant, quand le nombre d'états ainsi que l'ensemble de caractérisation du système ne sont pas connus, il nous faut toujours un moyen d'inférer une approximation du système avec une précision contrôlable. Cela est possible en utilisant un ensemble Z de séquences d'entrées prédéfinies au lieu de l'ensemble de caractérisation. En effet,

une relation de Z-équivalence a été définie sur les états de la machine, donc elle peut être identifiée suivant cette relation. De cette idée, nous pouvons établir les définitions et l'algorithme d'inférence suivants.

Soit un FSM complet $A = (S, s_0, I, O, h_A)$ et un ensemble fini non vide de séquences d'entrées $Z \in I^*$, soit π_Z la partition sur l'ensemble des états S générée par la relation de Z-équivalence. Pour un état s, les états qui sont Z-équivalent à l'état s constituent la classe d'équivalence $\pi_Z(s)$. L'idée que nous utilisons ici et qui peut être relié au travail de [Biermann 1972] et [Nerode 1958] est de fusionner tous les états Z-équivalent (k-équivalent dans [Biermann 1972]) en gardant toutes les transitions. Le modèle ainsi obtenu conserve toutes les traces du modèle original, mais en possède de nouvelles.

Si l'on veut pouvoir inférer un modèle, nous devons conserver uniquement un seul élément pour chaque classe d'équivalence π_Z parce qu'une fois un état Z-distinct identifié, nous devons l'inclure dans le modèle inférer qui ne doit contenir que des états différents (Z-distinct).

Définition 1 *Soit un FSM complet $A = (S, s_0, X, O, h_A)$ ainsi qu'un ensemble de séquences d'entrées $Z \subseteq I^*$, $I \subseteq X$, un FSM $K = (Q, q_0, I, O, h_K)$ est un (Z, I)-*quotient initial *de A, s'il existe une injection f de Q à S tel que :*

- *$f(q_0) = s_0$;*
- *pour chaque paire d'états distincts $q_1, q_2 \in Q$, $f(q_1)$ et $f(q_2)$ sont Z-distinct ;*
- *pour tout $q \in Q$, il existe un chemin $(s_0, a_1b_1, s_1) \dots (s_{n-1}, a_nb_n, s_n)$, tel que $s_i = f(q_i)$, $q_i \in Q$, $1 \leq i \leq n$ et $s_n = f(q)$;*
- *pour tout $q \in Q$ et $a \in I$, $b \in O$, $(p, b) \in h_K(q, a)$ si et seulement si $s \in S$, tel que $(s, b) \in h_A(f(q), a)$ et s et $f(p)$ sont Z-équivalent.*

Remarque : nous utilisons ici le terme quotient *initial* pour insister sur le fait qu'il représente uniquement une partie atteignable du FSM donné, depuis l'état initial, et ce, modulo la Z-équivalence.

Nous n'avons pas besoin d'utiliser toutes les entrées pour chaque état du système à inférer. En pratique, que ce soit pour un protocole ou une application Web, nous observons que pour chaque état, seul un sous-ensemble des entrées peut être utilisé de façon valide, le reste des entrées peuvent être considérées comme *équivalentes* et conduisant à une action non appropriée pour l'état ou à une erreur. Par exemple, l'action d'authentification est bien souvent la seule action possible avant de pouvoir utiliser pleinement l'application. Utiliser à chaque fois toutes les entrées entraînerait un grand nombre de requêtes inutile pour l'inférence.

Si $I = X$ alors nous nous referons au *Z-Quotient initial* au lieu (Z, X)-quotient initial, ou simplement au *quotient* quand l'ensemble Z est clairement défini dans le contexte. Par exemple, si l'on considère le FSM A dans la Figure 3.6 et son Z-quotient initial dans la Figure 3.7, où $Z = \{a, b\}$, $f(\varepsilon) = 0$, $f(a_1) = 1$ et $f(a_1a_2) = 2$.

Notons que le Z-quotient initial est déterministe.

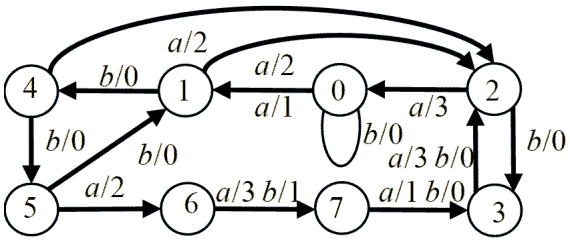

FIGURE 3.6 – FSM A

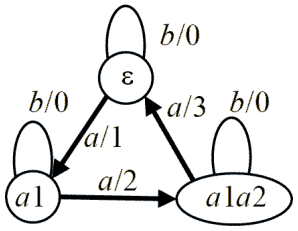

FIGURE 3.7 – $\{a, b\}$-quotient initial du FSM A

Il y a un cas spécial où le Z-quotient initial est équivalent au FSM A : l'ensemble Z est un ensemble de caractérisation du FSM A. Si c'est le cas, alors chaque paire d'états de A peut être différencié à l'aide des séquences de Z ce qui implique que chaque état du système est représenté par un état distinct dans le quotient initial. Rappelons que pour la Figure 3.6 où $Z = \{a, b\}$ n'est pas égal à l'ensemble de caractérisation de A, le Z-quotient initial n'est donc pas équivalent à A. Cette observation nous conduit au théorème suivant à propos du Z-quotient initial.

Théorème 1 *Soit un Z-quotient initial K d'un FSM complet A, si Z est l'ensemble de caractérisation de A, alors les FSM A et K sont équivalent sinon, si A possède des états distincts, mais* Z-équivalent*, alors A et K sont distincts. Ce théorème est extrait de l'article [Petrenko 2014].*

Au début de cette section, nous voulions une méthode d'inférence avec une précision contrôlable. Cette précision peut être contrôlée par la qualité de l'ensemble Z initial. Plus l'ensemble Z initial contient de séquences de caractérisation, plus précise sera l'approximation du Z-quotient du FSM. L'ensemble Z est appelé le paramètre de l'inférence.

Étant donné un entier naturel k, un Z-quotient initial est appelé un *k-quotient* si $Z = I^k$ [Groz 2008]. L'ensemble I^k contient les séquences d'entrées discriminantes

pour toutes les paires d'états de tous les FSM sur l'alphabet d'entrée I avec au plus $n = k + 1$ états, c'est donc l'ensemble de caractérisation de ces machines.

Le cas $k = n - 1$ correspond au pire scénario, cela se produit essentiellement avec des machines particulières, tel que les verrous de Moore [Moore 1956] où les plus longues séquences d'entrées sont nécessaires pour identifier certains états.

Pour les autres cas, la k-équivalence des états entraîne l'équivalence des états pour des valeurs plus petites que k. Il est donc plus approprié d'utiliser une caractérisation asymptotique du paramètre de l'inférence qu'une limite haute. Dans [Trakhtenbrot 1973], il est dit que pour un FSM complet avec n états, m entrées, et l sorties la longueur des séquences d'entrées atteignant tous les n états est asymptotiquement égale à $\log_m(n)$ et pour les états discriminants à $\log_m \log_l(n)$. Ces résultats nous montrent que même si la valeur de k est bien inférieure au nombre d'états du FSM, le k quotient devrait être suffisant précis pour être utilisé en pratique. Choisir de longues séquences d'entrées pour Z n'est pas nécessaire tandis que de courtes séquences ciblant certaines fonctionnalités du système devraient être plus efficaces.

3.2.3 Méthode d'inférence d'un Z-quotient pour FSM

Pour la suite, nous supposons que le système sous inférence peut être utilisé en boite noire, c'est-à-dire envoyer des séquences d'entrées et observer des séquences des sorties, et qu'il peut être modélisé sous la forme d'une machine à états finie complète et déterministe sur les entrées X et les sorties O. De plus, une opération de remise à zéro du système (retour à l'état initial) peut être utilisée entre différentes séquences d'entrées.

De plus, nous supposons avoir un ensemble d'entrées $I \subseteq X$ et un ensemble de séquences d'entrées $Z \subseteq I^*$ pour inférer un (Z, I)-quotient de A en testant le système. Avec la définition du (Z, I)-quotient, on peut avoir une idée de l'algorithme d'inférence :

— construire le (Z, I)-quotient initial en incluant un état initial qui représente celui de A ;
— explorer les états de A à partir de l'état initial en utilisant les entrées connues de I ;
— pour chaque état visité, s'il est Z-équivalent à un état déjà visité, alors ne plus explorer depuis cet état.

Pour chaque état, les transitions sont définies selon la Z-équivalence.

Cela représente les grandes étapes de l'algorithme qui sera décrit complètement dans les paragraphes suivants. Pour conserver les traces observées, nous utiliserons un FSM sous forme d'arbre.

Définition 2 *Soit un ensemble U de traces closes par préfixe, observé sur le FSM et définis sur l'ensemble d'entrées I et de l'ensemble de sorties O, l'*arbre d'ob-

servation *est le FSM* $(U, \varepsilon, I, O, h_U)$*, où l'ensemble des états est* U *et* $h_U(u, a) = \{(uab, b) \mid \exists b \in O \ (uab \in U)\}$.

Nous utilisons U pour représenter l'ensemble clos par préfixe de traces du FSM, c'est-à-dire les états, mais aussi le FSM $(U, \varepsilon, I, O, h_U)$.

La méthode d'inférence de quotient contient deux phases :
— construire l'arbre d'observation en identifiant les différents états ;
— déterminer les transitions.

Dans la première phase, nous envoyons différentes séquences entrées au système puis nous stockons les sorties observées dans l'arbre U, initialisé avec $\{\varepsilon\}$. Ensuite pour identifier les états, nous parcourons en largeur l'arbre. Si un nœud, c'est-à-dire un état u, se trouve être Z-distinct des autres nœuds déjà parcourus, alors c'est un nouvel état. Nous ajoutons alors u à l'ensemble des états du quotient. Dans le cas contraire, il existe un état w, précédemment parcouru, qui est Z-équivalent à u. Nous étiquetons l'état u avec w, c'est-à-dire $label(u) = w$; u n'est donc pas inclus dans l'ensemble des états du quotient et le comportement du FSM A ne sera pas exploré à partir de u puisqu'il est déjà exploré par w. Une fois que l'arbre a fini de grandir, nous avons identifié tous les états du (Z, I)-quotient.

Dans la deuxième étape, nous utilisons les transitions présentes dans l'arbre pour définir les transitions du quotient. Seules les transitions à partir d'un état non étiqueté et vers un état non étiqueté sont considérées. Pour les autres transitions, elles sont redirigées vers l'état correspondant à l'étiquette.

Nous avons séparé l'algorithme d'inférence du quotient en deux parties définies ci-dessous.

Soit un arbre d'observation U, et un FSM inconnu A avec un alphabet d'entrée X, la procédure *Etendre_nœud(A, u,* Σ*)* permet de découvrir le FSM A depuis un état atteint par la séquence d'entrées u en appliquant les séquences d'entrées de l'ensemble Σ des *requêtes* et retourne un arbre d'observation étendu. Cette procédure est décrite dans l'algorithme 2.

Algorithme 2 : Etendre_nœud (A, u, Σ) où $u \in U$, $\Sigma \subseteq X^*$

```
1 while ∃ a1a2...ak ∈ Σ / a1a2...ak ≠ v ↓I, ∀ v ∈ Tr(u) do
2 |   remettre A à son état initial;
3 |   appliquer u ↓I à A;
4 |   appliquer a1a2...ak à A et obtenir b1b2...bk en sortie;
5 |   ajouter la trace ua1b1a2b2..akbk et tous ses préfixes ua1b1, ua1b1a2b2, ...,
6 |       ua1b1a2b2..akbk à U;
7 end
```

Soit un arbre d'observation U, et un FSM A inconnu sur l'ensemble d'entrée X et de sortie O. La procédure *Construire_Quotient(A, I, Z, U)*, où $I \in X$, $Z \in I^*$, construit le FSM $K = (Q, q_0, I, O, h_K)$ qui est un (Z, I)-quotient de A, et retourne un arbre d'observation étendu. Cette procédure est détaillée dans l'algorithme 3.

Remarque : La procédure *Construire_Quotient* peut prendre n'importe quel arbre d'observation U, y compris l'arbre trivial $U = \{\varepsilon\}$.

Algorithme 3 : Construire_Quotient (A, I, Z, U_0)

Result : Un arbre d'observation U étiqueté et le FSM $K = (Q, \varepsilon, I, O, h_k)$ comme étant le (Z, I)-quotient du FSM A

$U = U_0$;
foreach *état u de U parcouru durant le BFS tel que u n'a pas de prédécesseur étiqueté* **do**
  Étendre_nœud(A, u, Z);
  if *u est Z-équivalent à un nœud déjà traversé w de U* **then**
    étiquette u avec w, c'est-à-dire $label(u) = w$;
  end
  else
    ajouter u dans Q;
    Étendre_nœud(A, u, I);
  end
end
K = FSM vide (sans état);
foreach *transition (u, ab, v), tel que ni l'état u ni ses prédécesseurs ne soient étiquetés* **do**
  if *v n'est pas étiquette* **then**
    ajouter la transition (u, ab, v) à K;
  end
  else
    ajouter la transition (u, ab, w) à K, où $w = label(v)$;
  end
end
foreach *nœud u' étiqueté avec u* **do**
  étiqueter les successeurs de u' tel que ;
  foreach *transition (u', ab, v)* **do**
    if *il existe une transition (u, ab, w) dans K* **then**
      étiquette v avec w;
    end
  end
end
return U, K;

Pour illustrer cet algorithme sur un exemple concret, nous allons essayer d'inférer un (Z, I)-quotient du FSM A de la Figure 3.6. L'ensemble d'entrées X de A est $\{a, b\}$, et nous construisons un (Z, I)-quotient avec l'ensemble d'entrées $Z = I = \{a, b\}$. Au départ, $U = \{\varepsilon\}$, $Z = \{a, b\}$. À la première exécution de la première boucle *for*, seul le nœud ε est parcouru. Après avoir appelé Étendre_nœud(A, ε, Z), les nœuds $a1$ et $b0$ sont ajoutés à U, et ε est ajouté à Q.
Dans la seconde exécution, le nœud $a1$ est parcouru. Cette fois, les séquences $a1a2$ et $a1b0$ sont ajoutés à U. Comme le nœud $a1$ n'est pas Z-équivalent au nœud ε, $a1$ est aussi ajouté à Q.
Dans la suite de l'exécution, Étendre_nœud$(A, b0, Z)$ est appelé, les nœuds $b0a1$ et $b0b0$ sont ajoutés à U. Comme le nœud $b0$ est Z-équivalent au nœud ε, il est étiqueté avec ε.
La procédure continue, les nœuds $a1a2$, $a1b0$, $a1a2a3$, et $a1a2b0$ sont parcourus, le nœud $a1a2$ est ajouté à Q, $a1b0$ est étiqueté avec $a1$ et $a1a2a3$ avec ε, $a1a2b0$ avec $a1a2$.
À la fin de la première boucle, $Q = \{\varepsilon, a1, a1a2\}$.

Pour l'exécution de la seconde boucle *for*, les transitions suivantes sont ajoutées à K : $(\varepsilon, a1, a1)$, $(\varepsilon, b0, \varepsilon)$, $(a1, a2, a1a2)$, $(a1, b0, a1)$, $(a1a2, a3, \varepsilon)$, et $(a1a2, b0, a1a2)$. Avec ces transitions, nous obtenons le quotient initial de la Figure 3.7.

Finalement, la troisième boucle *for* est exécutée. Par exemple, le nœud $b0$ est étiqueté avec ε, dans U il existe une transition $(b0, a1, b0a1)$, dans K il existe une transition $(\varepsilon, a1, a1)$, donc, le nœud $b0a1$ est étiqueté avec $a1$. L'arbre d'observations final U est représenté dans la Figure 3.8.

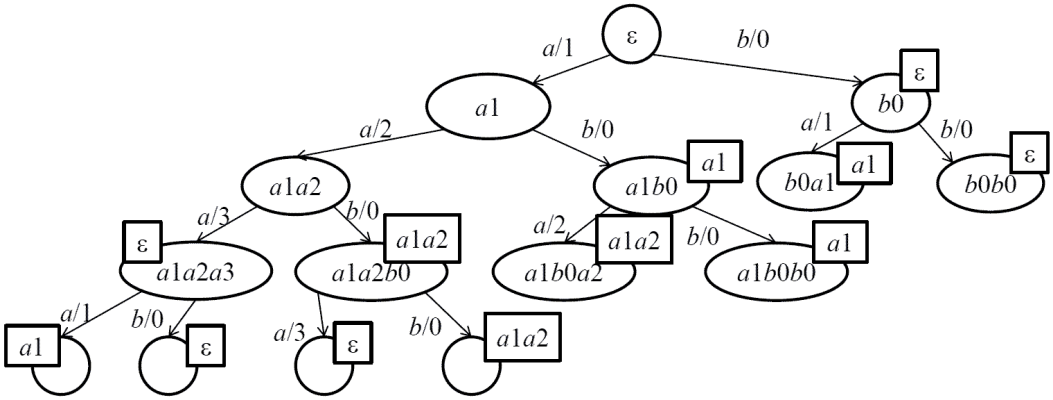

FIGURE 3.8 – Arbre d'observation U

La procédure Construire_Quotient(A, I, Z, U) partitionne les nœuds de l'arbre d'observation U selon leurs comportements avec les séquences d'entrées de Z de telle façon que chaque classe d'équivalence devient un état du quotient. Mais elle ne garantit pas que le quotient préserve toutes les traces qui pourraient les différencier. C'est pourquoi le quotient retourné par cette procédure a besoin d'une vérification

ou mise à jour.

Soit $a_1a_2...a_k$ une séquence d'entrée de X^* et K un FSM (Q, q_0, I, O, h_K) où $I \subseteq X$. Une trace $a_1b_1a_2b_2...a_kb_k$ est dite *incompatible* avec l'état q de K si et seulement s'il n'existe pas de trace $c_1b_1c_2b_2...c_kb_k$ de l'état q tel que pour tout i, si $a_i \in I$ alors $c_i = a_i$.

Notons que pour un (Z, X)-quotient, c'est-à-dire $I = X$, la définition ci-dessus nécessite seulement qu'une trace donnée ne soit pas dans $Tr(q)$.

Nous présentons une version plus générale dans les sections suivantes pour gérer les contre-exemples.

Étant donné un arbre d'observation U et un (Z, I)-quotient $K = (Q, q_0, I, O, h_K)$ obtenu à partir de U, un nœud de l'arbre U étiqueté avec $q \in Q$ possède une étiquette *contradictoire*, s'il a une trace qui est incompatible avec l'état q du (Z, I)-quotient K ; cette trace est appelée une *trace contradictoire* pour q.

Une façon de résoudre ces contradictions est d'étendre l'ensemble Z avec les projections des entrées de la trace contradictoire, puis de répéter la procédure Construire_Quotient(A, I, Z, U) jusqu'à ce que l'arbre d'observation ne contienne plus d'étiquette contradictoire. Cette idée est développée dans la procédure nommée Rendre_Cohérent de l'algorithme 4.

Algorithme 4 : Rendre_Cohérent(A, I, Z, U, K)

Result : Un arbre d'observation correctement étiqueté et son quotient

while *Il existe un nœud d'une trace contradictoire non parcouru w pour un état q tel que la projection de ses entrées ne soit pas dans Z* **do**
  if *Il n'existe pas de trace $v \in Tr_U(q)$ telle que $v \downarrow_I = w \downarrow_I$* **then**
    appliquer $(qw) \downarrow_I$ à A dans l'état initial pour obtenir la trace $v \in Tr_U(q)$;
  end
  if $w \neq v$ **then**
    $Z' = Z \cup \{w \downarrow_I\}$ et $I' = I \cup set_of_input(w)$;
    U, K = Construire_Quotient(A, I', Z', U);
    $Z = Z'$ et $I = I'$;
  end
  marquer w comme parcouru;
end
return I, Z, U, K;

Si l'on reprend l'exemple précédent, l'arbre d'observation U dans la Figure 3.8 ne contient aucune étiquette incohérente.

Le théorème suivant dit que la méthode présentée ci-dessus peut être utilisée

pour inférer un Z-quotient initial d'un FSM à partir des traces récoltées durant des phases de tests.

Théorème 2 *Les procédures Construire_Quotient et Rendre_Cohérent appliquées à un FSM A déterministe se terminent. Le FSM K résultant de ces procédures est un (Z, I)-quotient initial du FSM A, de plus, c'est un FSM minimal. Ce théorème est extrait de l'article [Petrenko 2014].*

3.2.4 Gestion des contre-exemples

Une fois le (Z, I)-quotient initial obtenu ainsi que le FSM K correspondant, K n'est pas forcément équivalent à A, donc il peut exister des traces qui sont incompatibles avec l'état initial du (Z, I)-quotient. Ces traces sont appelées *contre-exemples* ou CE.

La méthode d'inférence inclut une partie pour la gestion des CE : soit un FSM inconnu A avec un alphabet d'entrée X, un alphabet de sortie O et $I \subseteq X$, $Z \subseteq I^*$. La procédure *Infère(A, I, Z)* retourne des ensembles I' et Z' mis à jour ainsi qu'un FSM qui correspond au (Z', I')-quotient initial de A. Cette procédure est détaillée dans l'algorithme 5.

Algorithme 5 : Infère(A, I, Z)

```
Result : I, Z et le quotient correspondant
1 U, K = Construire_Quotient(A, I, Z, {ε});
2 Rendre_Cohérent(A, I, Z, U, K);
3 while Il existe un contre-exemple CE pour K do
4 |   U = U ∪ CE;
5 |   I, Z, U, K = Rendre_Cohérent(A, I, Z, U, K);
6 end
7 return I, Z, U, K;
```

Si l'on reprend encore une fois notre exemple, nous avons effectué les étapes 2 et 3 pour obtenir le (Z, I)-quotient décrit dans la Figure 3.8. Puis dans l'étape 4, supposons que nous obtenons le contre-exemple suivant : $a1a2b0b0b0a3b0b0a3$. Dans l'étape 6, nous mettons à jour l'arbre U avec le contre-exemple et nous obtenons l'arbre de la Figure 3.9.

À ce moment, la procédure Rendre_Cohérent est appelée. Dans la première exécution de la boucle *while*, nous remarquons que dans U, l'état $a1a2b0$ possède comme étiquette $a1a2$ ce qui est incohérent, puis la trace $b0b0a3b0b0a3$ est incompatible avec l'état $a1a2$ dans le (Z, I)-quotient, comme cet état ne possède pas ces traces. Donc, la trace incohérente w est $b0b0a3b0b0a3$. Nous appliquons en premier $a1a2$ suivi par $bbabba$ à A dans l'état initial et nous obtenons la trace

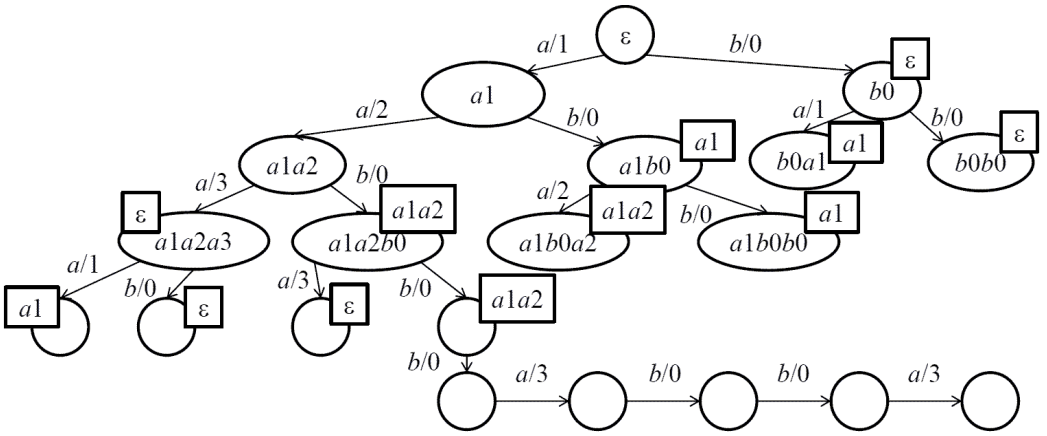

FIGURE 3.9 – Arbre d'observations U mis à jour avec le contre-exemple

$b0b0a3b0b0a1$, qui est différente de $w = b0b0a3b0b0a3$. Ensuite, nous étendons Z avec $\{a, b, bbabba\}$ et exécutons la procédure Construire_Quotient. Le nouveau (Z, I)-quotient est représenté dans la Figure 3.10.

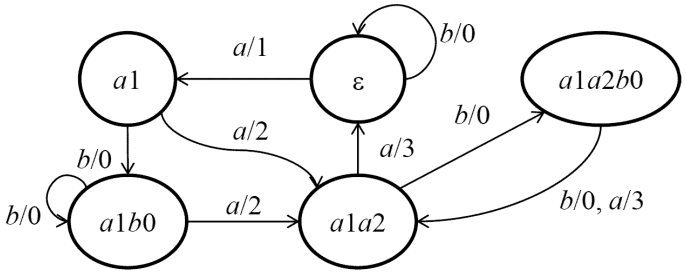

FIGURE 3.10 – FSM K_1, un $\{a, b, bbabba\}$-quotient initial du FSM A

Dans la seconde exécution de la boucle *while*, nous remarquons que dans U, l'état $a1b0b0$ étiquette avec $a1b0$ est incohérent parce que la trace $a2b1$ est incompatible avec l'état $a1b0$ du (Z, I)-quotient de la Figure 3.10. Dans ce cas, la trace contradictoire w est $a2b1$. L'état $a1b0$ a une trace $a2b0$, qui a comme projection ab et qui est différent de w. Nous étendons alors Z à $\{a, b, bbabba, ab\}$ puis exécutons Construire_Quotient encore une fois.

Le (Z, I)-quotient obtenu à cette étape est équivalent au FSM que nous voulions inférer.

3.2.4.1 Stratégies de gestion des contre-exemples

Dans la procédure Rendre_Cohérent, la recherche de contradiction peut s'implanter de différentes façons. Par exemple, chacun des suffixes du contre-exemple pourrait être vérifié avec une longueur croissante. En effet, une stratégie de bas en haut, comme dans la méthode Suffix1by1 [Irfan 2010b], permet d'identifier le suffixe de taille minimale du contre-exemple qui provoque la contradiction.

Contrairement aux méthodes basées sur les tables d'observations telles que [Irfan 2010b, Steffen 2011], où la longueur de la trace est importante, dans notre méthode il est possible de ne faire qu'une recherche de haut en bas qui, dans la plupart des cas (dont les verrous de Moore et les compteurs) est plus efficace. En utilisant la méthode de Rivest et Schapire [Rivest 1993], il est aussi possible d'identifier la plus courte trace par dichotomie sur le contre-exemple.

3.2.5 Expérimentations

Pour évaluer les performances de l'algorithme proposé ainsi que son adéquation avec des systèmes concrets et existants, nous avons effectué divers tests sur des machines générées aléatoirement puis sur deux fournisseurs de voix sur IP (VOIP) utilisant chacun une implantation différente du protocole SIP. En utilisant notre algorithme d'inférence, nous pourrons comparer ces deux implantations.

La génération des machines aléatoires se fait en créant les états en premier. Puis un certain nombre de symboles d'entrées et de sorties sont générés et une passe est effectuée sur chaque couple d'états pour créer les transitions selon un pourcentage prédéfini. Pour chaque transition créée, une entrée et une sortie y sont associées. Chaque paramètre est défini par un minimum et un maximum.

3.2.5.1 Machines aléatoires

Nous avons donc généré un grand nombre de machines aléatoirement. Même si nous connaissons exactement les spécifications de ces machines et que nous pouvons calculer à chaque fois un contre-exemple de taille minimale, nous avons choisi de simuler une recherche de CE comme elle serait faite en pratique, c'est-à-dire en générant des séquences d'entrées aléatoires, puis en observant la réponse du système inféré et réel à cette séquence. Cette marche aléatoire sera reproduite un certain nombre de fois, une fois cette limitée dépassé, la machine inférée sera considérée comme équivalente au système et l'inférence sera terminé. Dans ces expérimentations, nous avons déterminé le nombre moyen de requêtes (tests) nécessaire pour que l'inférence se termine. Les machines générées ont un nombre d'états variant de 50 à 1000, 10 entrées et 10 sorties.

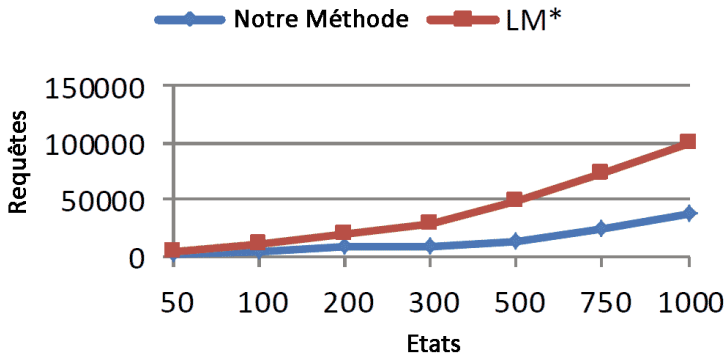

FIGURE 3.11 – Comparaison du nombre de requêtes moyen avec l'algorithme LM^*

La Figure 3.11 montre le nombre de requêtes émis pendant l'inférence et la méthode basée sur les tables, l'algorithme LM^* [Shahbaz 2009] combiné avec la méthode *Suffix1by1* de gestion des contres exemples [Irfan 2010b]. Avec la nouvelle méthode, le nombre de requêtes augmente moins vite que pour LM^*.

Nous avons aussi comparé cette méthode à un autre algorithme basé sur des tables, l'algorithme L_m^*. Nous avons utilisé les exemples fournis par les auteurs dans [Steffen 2011]. Le FSM à inférer a 6 états, 4 entrées et 3 sorties. Pour chaque méthode, nous avons considéré uniquement deux contre-exemples. L_m^* infère la machine avec 155 requêtes. Avec un ensemble d'entrées I vide, notre méthode nécessite le même nombre de requêtes ; par contre, quand nous considérons les entrées déjà connues (comme le fait L_m^*), le nombre de requêtes nécessaire n'est plus que de 124.

3.2.6 Avantages et inconvénients

L'algorithme proposé permet d'inférer un modèle d'un système sous forme de FSM avec des performances légèrement meilleures que les approches tabulaires basées sur l'algorithme L* d'Angluin. L'approche incrémentale est un des avantages sur ces derniers. En effet, nous n'avons pas besoin de connaître toutes les entrées du système pour commencer l'inférence et l'algorithme peut utiliser les entrées qui sont découvertes au fur et à mesure. Au-delà de cette différence de fonctionnement, elle permet d'utiliser des méthodes de récupérations d'entrées et sorties automatiques pour applications Web. Même si ces méthodes ne couvrent pas l'application en entier, l'inférence peut toujours s'adapter aux nouvelles entrées et sorties. Pour aider l'inférence, le testeur peut fournir un certain nombre de séquences d'entrées connues pour déclencher certains comportements. D'un point de vue plus pratique, l'utilisation d'un arbre au lieu de tables contenant des structures complexes facilite son implantation.

Cette méthode d'inférence infère un modèle sous forme de FSM. Ce type de modèle est adéquat si l'on s'intéresse seulement à la partie logique du système, mais dans le cas des applications Web, une grande partie des vulnérabilités se situe au niveau des données échangées. Il est toujours possible de considérer ces données en gardant un modèle sous forme de FSM, mais cela produirait un grand nombre d'états et rendrait le modèle plus difficile à analyser. Comme pour l'approche présentée dans la section 3.1, l'écriture de l'adaptateur doit se faire manuellement. Cela comprend l'identification des différentes entrées et sorties, de leurs paramètres et l'écriture des fonctions de traduction.

Chapitre 4

Création d'adaptateurs de test

La création d'un adaptateur de test est une étape essentielle dans l'utilisation de l'inférence de modèle. Selon le type du système, sa taille et les bibliothèques utilisées, il est plus ou moins difficile de construire cet adaptateur. Dans ce chapitre, nous présentons ce qu'est un adaptateur de test ainsi que sa construction manuelle. Ensuite nous montrons comment générer de façon automatique ces adaptateurs pour les applications Web.

Sommaire

4.1 Adaptateur de test

Comme nous l'avons vu précédemment, les méthodes d'inférence actives doivent de communiquer avec le système sous inférence (SUI) contrairement à l'inférence passive où des traces de communications avec le système sont utilisées. Un des problèmes de l'inférence active est donc de définir comment communiquer avec le système. Le problème étant que l'algorithme d'inférence fonctionne à un niveau abstrait (symboles et variables) alors que l'application fonctionne à un niveau concret (requête HTTP, requêtes formatées).

Généralement, ces méthodes ont besoin de pouvoir envoyer des requêtes (entrées) et d'observer les réponses (sorties). De plus, il est nécessaire pour la plupart des algorithmes de connaître toutes les entrées et sorties possibles avant même le début de l'inférence. Selon le modèle que l'on cherche à inférer, les entrées peuvent être un simple symbole ou un symbole paramétré.

L'adaptateur de test se place entre le système à inférer et l'outil d'inférence. Il sert à convertir les requêtes et réponses entre les niveaux abstrait et concret. Nous pouvons schématiser le fonctionnement d'un adaptateur de test avec la figure 4.1 :

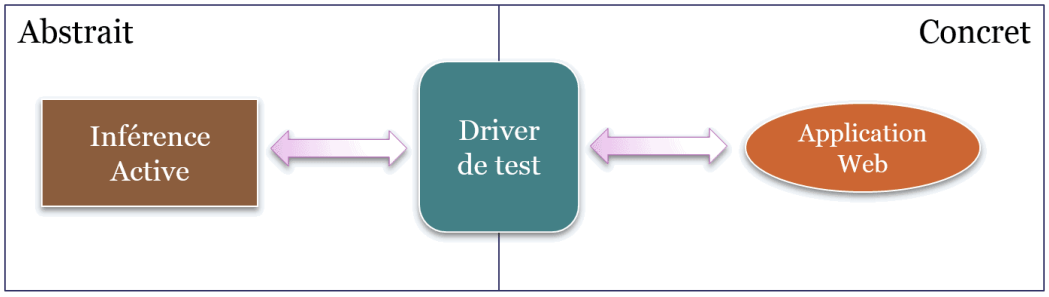

FIGURE 4.1 – Adaptateur de test

À l'aide de la figure 4.1, nous pouvons présenter le fonctionnement de l'adaptateur de test en quatre étapes :

— Supposons qu'il existe un symbole d'entrée *login* à deux paramètres (nom d'utilisateur et mot de passe). L'inférence veut envoyer une requête abstraite : $login(root, toor)$
— L'adaptateur de test convertit la requête abstraite en concrète. Par exemple, il peut s'agir d'une requête *POST* vers la page *login.php* avec comme paramètres $user = root\&password = toor$.
— L'application va traiter cette requête puis répondre par une réponse HTTP avec, par exemple, le code 200 qui correspond à une exécution de la requête sans problème. Puis nous avons quelques entêtes concernant la réponse puis le code source HTML de la page.
— L'adaptateur de test récupère la réponse HTTP et construit puis envoie une réponse abstraite. Dans notre cas, cela peut être : $homepage(root)$. Ce symbole correspond à la page d'accueil de l'application et il contient un paramètre qui est le nom de l'utilisateur.

On peut remarquer que la première transformation nécessite d'avoir une certaine connaissance de l'application cible notamment le format des messages et la page responsable du traitement de la requête. De plus, la seconde transformation est loin d'être simple à réaliser puisqu'il faut, en quelque sorte, connaître la sémantique de la page renvoyée pour trouver le symbole de sortie correspondant.

Dans le cadre du projet SPaCIoS, nous avons dans un premier temps construit des adaptateurs de test manuellement pour pouvoir ensuite tester les algorithmes d'inférence. L'effort nécessaire étant conséquent, nous avons ensuite développé une méthode de génération automatique d'adaptateur de test pour les applications Web, l'une des cibles du projet SPaCIoS.

4.2 Écriture manuelle d'adaptateur de test

Pour tester l'algorithme d'inférence de Z-Quotient, nous avons créé un adaptateur de test pour le protocole SIP. Nous avons aussi créé un adaptateur de test pour une application Web dans le but de tester l'algorithme d'inférence du projet SPaCIoS.

4.2.1 Pour le protocole SIP (non basés sur HTTP)

Dans la section 7.3.2, nous présenterons les résultats de l'inférence de deux fournisseurs de service SIP. Pour cela, nous avons d'abord créé un adaptateur de test pour le protocole SIP. Dans cette partie nous décrivons comment écrire un adaptateur de test pour un protocole où les messages sont précisément définis dans des RFC (Request for Comments) et différents de HTTP. Nous présentons les avantages et inconvénients de ce type de système.

4.2.1.1 Choix de la bibliothèque

Le protocole SIP (Session Initiation Protocol) est un protocole de communication souvent utilisé dans le domaine de la VOIP et des télécommunications en général. C'est un protocole texte, dans le sens où les messages sont transmis sous forme de texte. SIP est par ailleurs basé sur HTTP et SMTP, c'est pourquoi nous retrouvons le même format de message.

Listing 4.1– Exemple de requête SIP

```
INVITE sip:1000@iptel.org SIP/2.0
Call-ID: STARGATE@82.233.118.237
CSeq: 7 INVITE
From: <sip:user1test@iptel.org>;tag=1250168047
To: <sip:1000@iptel.org>
Via: SIP/2.0/UDP 192.168.0.10:5060;branch=z9hG4bK372632
Max-Forwards: 70
Contact: <sip:user1test@82.233.118.237:5070>
User-Agent: SIMPA/SIPClient
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, INFO, REFER, NOTIFY
Supported: replaces
Content-Type: application/sdp
```

```
Content-Length: 358
```

Dans le message décrit dans le listing 4.1, on peut reconnaître certains champs existants aussi dans le protocole HTTP comme $Content-Type$, $Content-Length$ ou encore $User-Agent$. Par contre, les méthodes GET et $POST$ ont été remplacées par $INVITE$, une méthode propre à ce protocole. Puis diverses autres entêtes ont été ajoutées.

SIP est indépendant de la couche de transport, c'est-à-dire qu'il peut très bien fonctionner sur TCP ou même UDP. SIP fonctionne par signaux qui sont émis aussi bien par le client que par le serveur quand un événement est arrivé. Il se peut qu'il n'y ait aucun signal pendant un certain temps ce qui rend l'utilisation de TCP un peu plus efficace puisqu'il permet de réduire le trafic alors que sur UDP, des signaux sont échangés pour maintenir l'état du client ou serveur à jour. En pratique, c'est UDP qui est le plus utilisé et c'est celui qui est accepté par tout les fournisseurs de services SIP gratuits que nous avons essayés. Comme peu de fournisseurs acceptaient SIP sur TCP, nous avons utilisé UDP pour développer notre adaptateur de test.

Comme l'outil d'inférence ainsi que le projet SPaCIoS est développé principalement en Java, il a été normal de choisir Java pour développer notre adaptateur. Il existe bien des bibliothèques en Java pour créer des clients SIP. Il a fallu trouver une bibliothèque qui ne travaille pas seulement à un haut niveau, par exemple, avec des méthodes comme *passerUnAppel* ou *raccrocher* mais aussi au niveau des messages SIP pour pouvoir créer soi-même des messages personnalisés en utilisant les différents types de messages SIP.

Nous avons choisi finalement une pile SIP appelée JAIN-SIP [Java.net 2010] qui est bien documentée et possède une grande communauté d'utilisateurs.

4.2.1.2 Lister les entrées

La RFC 3261 définit un certain nombre de types de requêtes SIP décrites dans le tableau 4.1. S'il existe 14 types de requêtes différentes, seul un sous-ensemble de 4 requêtes est nécessaire pour émettre un appel complet (émission et terminaison). Dans le but d'écrire un adaptateur de test dans un temps raisonnable, nous avons choisi d'utiliser uniquement ces quatre méthodes qui sont : *INVITE*, *ACK*, *REGISTER*, *BYE*. Ainsi nous pourrons modéliser le comportement du serveur quand un client passe un appel.

4.2.1.3 Lister les sorties

La RFC 3261 définit un certain nombre de types de réponses SIP décrites dans le tableau 4.2. En plus de ces réponses, nous avons ajouté un symbole de sortie

Table 4.1 – Liste des différentes requêtes SIP

Méthode	Description	RFC
INVITE	Un client commence un appel.	RFC 3261
ACK	Confirmation d'une requête INVITE.	RFC 3261
BYE	Termine un appel.	RFC 3261
CANCEL	Annule les requêtes en attente.	RFC 3261
OPTIONS	Demande les options du serveur.	RFC 3261
REGISTER	S'enregistre auprès du serveur SIP.	RFC 3261
PRACK	Réponse provisionnelle.	RFC 3262
SUBSCRIBE	Souscrit à un événement.	RFC 6665
NOTIFY	Notifie un client d'un événement.	RFC 6665
PUBLISH	Publie un événement sur le serveur.	RFC 3903
INFO	Demande d'information de mi-session.	RFC 6086
REFER	Demande au destinataire d'un appel	RFC 3515
MESSAGE	Envois de message texte sur SIP.	RFC 3428
UPDATE	Modifie l'état d'une session SIP	RFC 3311

Table 4.2 – Liste des différentes réponses SIP

Code	Description
Provisional (1xx)	La requête est en cours de traitement.
Success (2xx)	La requête a été bien reçue, comprises et traitée.
Redirection (3xx)	D'autres actions sont nécessaires côté émetteur.
Client Error (4xx)	Erreur de syntaxe ou impossible de compléter la requête.
Server Error (5xx)	Requête valide, mais erreur au niveau du serveur.
Global Failure (6xx)	Impossible de traiter la requête.

nommé $TIMEOUT$ dans les cas où le serveur n'émet aucune réponse après un certain temps. Cette éventualité peut arriver quelquefois puisque nous utilisons le protocole SIP en mode UDP et que les serveurs ciblés sont de réels serveurs qui ont un grand nombre d'utilisateurs et qui risquent de ne plus vouloir répondre si on exécute des requêtes invalides trop de fois. Nous avons donc fixé le délai maximum à 8 secondes.

Au lieu de définir un symbole par réponse, nous avons directement utilisé le code de sortie comme symbole de sortie. Pour ce genre de protocole, le contenu du message n'est pas nécessaire pour avoir la signification du message. Toutes les différentes possibilités de message possèdent un code spécifique.

4.2.1.4 Les étapes d'une communication

Pendant l'inférence, l'adaptateur sera amené à envoyer les différents messages qui composent une communication comme décrite dans la figure 4.2. Il faudra donc

que ces messages soient valides pour être pris en compte par le serveur correctement. Étant basé sur HTTP, SIP hérite de sa particularité d'être un protocole sans état. Pour conserver un état, divers entêtes des messages sont utilisés comme l'entête *CSeq* qui sert à retrouver l'ordre des messages. Nous avons aussi créé un compte sur le site du fournisseur SIP. Pour le destinataire, chaque fournisseur possède un numéro attribué au service *echo* qui permet de tester si on est capable de passer une communication.

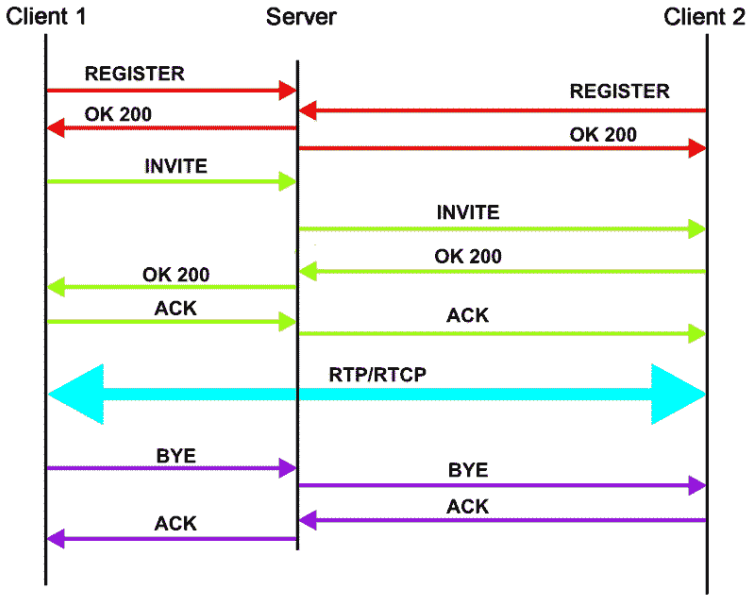

FIGURE 4.2 – Étapes d'une communication avec SIP

1. La première étape consiste à s'enregistrer auprès du proxy SIP pour que les autres utilisateurs puissent nous retrouver ensuite. Comme sur beaucoup de serveurs SIP, cette requête doit être authentifiée et nécessite deux messages $REGISTER$. Le premier sert à récupérer le nonce qui servira à produire un second message qui sera authentifié.
2. Maintenant nous pouvons commencer à passer un appel en utilisant le message $INVITE$. Comme pour le message $REGISTER$, ce message doit être authentifié.
3. Aprés avoir obtenu une réponse favorable pour l'appel, nous devons valider notre appel en envoyant un message ACK.
4. Dans un client classique, la télécommunication se fait maintenant sur un autre port qu'elle soit vidéo ou audio seulement. Nous pouvons couper cette communication à tout moment en envoyant le message BYE qui lui aussi doit être authentifié.

4.2.1.5 Développement d'un adaptateur

Une fois que nous avons la liste des entrées, sorties et que nous connaissons comment effectuer un appel, nous pouvons développer notre adaptateur de test. Nous pouvons remarquer ici, que SIP comme la plupart des protocoles, est bien défini, tant au niveau des entrées-sorties que sur les différentes valeurs que peuvent prendre un paramètre. Ce n'est pas le cas du protocole Web que nous verrons dans la section suivante où le nombre d'entrées, de sorties ainsi que le format des messages ne dépendent que des choix de l'utilisateur.

À chaque mise à zéro par l'adaptateur, il faut incrémenter le champ $CSeq$ des requêtes SIP pour que le message soit pris en compte comme faisant partie d'une nouvelle session. Pour les messages qui nécessitent d'être authentifiés, nous devons sauvegarder à chaque fois le nonce reçu dans la première requête.

Au final, le code de l'adaptateur contient trois principales méthodes :

— abstractToConcrete, qui crée une requête SIP pour chaque type de message.
— concreteToAbstract, qui renvoie le code de statut de la réponse SIP.
— execute, qui envoie et attend un certain temps la réponse du serveur SIP.

Développer un adaptateur de test pour SIP nécessite de bien connaître le protocole avant de pouvoir commencer une inférence. Nous sommes aidés par le choix fait pour SIP d'utiliser HTTP comme base pour le transport des messages. Mais pour la plupart des autres protocoles, cela n'est pas le cas. Comme la plupart des protocoles de l'Internet, SIP est entièrement définis dans une RFC.

Dans le cas des FSM, il n'y a pas de valeurs à fournir par l'utilisateur. A la place, l'adaptateur est construit directement avec les paramètres permettant de le rendre fonctionnel, comme l'adresse du destinataire de l'appel ou l'adresse du serveur SIP. Ce choix des valeurs peut se faire en étudiant les échanges réseaux entre un autre client et le serveur.

4.2.2 Pour les applications Web (sur HTTP)

Pour les applications Web, en plus de l'application sur le serveur, nous disposons aussi de l'interface graphique du côté client, la page Web. On peut donc définir deux types d'entrées :

— Au niveau HTML par des actions sur la page Web, par exemple, clics, envois de formulaire.
— Au niveau HTTP en construisant la requête HTTP correspondante à chaque action.

Dans le cadre du projet SPaCIoS, nous avons développé un adaptateur de test pour l'application SimpleSAMLphp [UNINETT 2007] au niveau HTML. Ensuite, pour l'application WebGoat, nous avons utilisé le niveau HTTP pour l'adaptateur.

Dans le cas du niveau HTTP, les actions correspondent aux actions disponibles sur la pages HTML mais nous les appelons HTTP parce que chacune de ces actions correspondent à exactement une requête HTTP, comme un formulaire ou un lien. Les actions du niveau HTML sont constituées d'une suite d'événements utilisateurs comme un clic ou une sélection. Elles correspondent à une action logique comme le fait de s'authentifier mais elle peut générer plusieurs requêtes HTTP.

Nous allons voir dans les sections suivantes les différents avantages ainsi que les inconvénients et raisons pour lesquels nous avons choisi un type et pas l'autre.

4.2.2.1 Niveau HTML

Au niveau HTML, nous utilisons les actions de la page Web pour représenter les entrées. Cela présente l'avantage de ne pas avoir besoin de comprendre exactement comment fonctionne l'application puisqu'on se sert de l'application elle-même pour générer les entrées. D'un autre côté, nous perdons en précision puisque nous considérons uniquement les actions à très haut niveau et laissons peut-être des paramètres gérés par l'application.

4.2.2.2 Choix de la bibliothèque

SimpleSAMLphp est une application écrite en PHP qui gère des mécanismes d'authentification. Elle implémente notamment SAML [Consortium 2008] en tant que fournisseur d'identité et de service. SAML est un protocole d'échange d'information d'authentification et d'autorisation basé sur des messages XML. SAML définit trois entités dans son protocole :

- un fournisseur de service qui détient la ressource à laquelle les utilisateurs peuvent accéder ;
- un fournisseur d'identité qui s'occupe de vérifier si l'utilisateur peut accéder à une certaine ressource ;
- un client qui souhaite accéder à une ressource.

Comme décrit dans l'image 4.3, il y a plusieurs étapes pour que le client puisse accéder à la ressource demandée.

- la première étape consiste à ce que le client demande la ressource au fournisseur de service ;
- le fournisseur de service propose ensuite une liste de fournisseurs d'identité qui s'occupe de la ressource ;

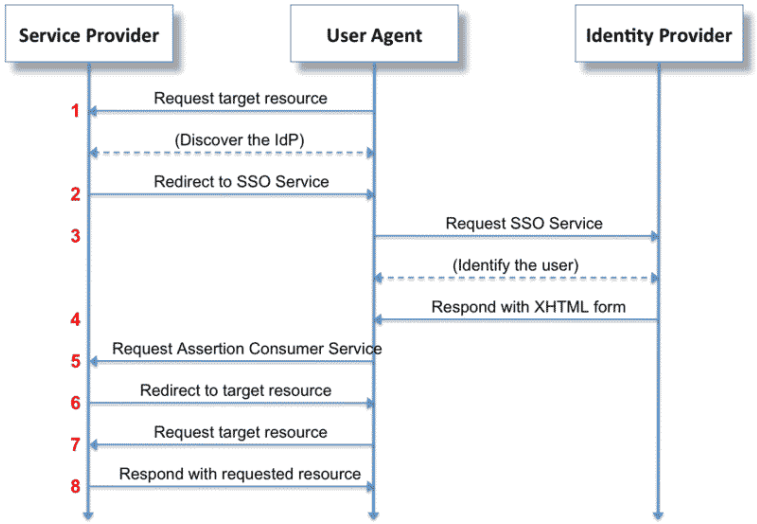

FIGURE 4.3 – Diagramme de fonctionnement de SAML

— le client est ensuite redirigé vers le fournisseur d'identité pour qu'il s'authentifie, quelque soit le moyen (nom d'utilisateur/mot de passe, LDAP, ...) ;
— le fournisseur d'identité répond au client avec ses informations d'autorisation ;
— le fournisseur de service vérifie que l'autorisation est valide et qu'elle correspond bien à la ressource ;
— le fournisseur de service renvoie la ressource au client.

Dans SimpleSAMLphp, la plupart des étapes intermédiaires sont cachées à l'utilisateur qui a l'impression de se connecter sur un site classique alors que l'authentification se passe sur un autre service. Du point de vue utilisateur, il ne voit que le formulaire de connexion et la réponse du fournisseur de services. Les messages XML sont encodés et transmis par l'application dans des champs cachés.

Nous avons utilisé Java, et plus particulièrement HTMLUnit [Bowler 2002], pour créer les adaptateurs de type HTML. HTMLUnit est un navigateur sans interface graphique qui peut se manipuler depuis du code Java. Nous avons donc accès à toutes les actions que l'utilisateur pourrait faire sur la page comme cliquer sur un bouton ou encore remplir un champ d'un formulaire. De plus, HTMLUnit fournit un moteur JavaScript qui prend en charge une bonne partie du langage.

Listing 4.2– Soumettre un formulaire avec HTMLUnit

```
public void submittingForm() throws Exception {
    final WebClient webClient = new WebClient();
```

```
        final HtmlPage page1 = webClient.getPage("http://some_url");

        final HtmlForm form = page1.getFormByName("myform");
        final HtmlSubmitInput button = form.getInputByName("submitbutton")
            ;
        final HtmlTextInput textField = form.getInputByName("userid");

        textField.setValueAttribute("root");

        final HtmlPage page2 = button.click();
        webClient.closeAllWindows();
    }
```

Dans le listing 4.2, nous avons un exemple d'utilisation de HTMLUnit pour soumettre un formulaire. Il suffit de récupérer le contenu d'une page, puis de chercher le formulaire ainsi que les différents champs disponibles. Ensuite, pour chaque champ, nous lui attribuons une valeur et soumettons finalement le formulaire.

Nous pouvons voir avec ce morceau de code qu'il est assez facile de construire un adaptateur avec des actions HTML.

4.2.2.3 Lister les entrées

Au niveau des entrées, il faut parcourir l'application que l'on souhaite inférer pour avoir les formulaires disponibles. Nous avons distingué quatre entrées :

— start : l'URL de base de l'application qui correspond à la page d'accueil ;
— selectIdP : le formulaire de choix du fournisseur d'identité ;
— login : le formulaire d'authentification ;
— logout : le formulaire de déconnexion.

Dans l'image 4.4, nous avons la page d'authentification de SimpleSAMLphp avec les deux champs pour le nom d'utilisateur et le mot de passe.

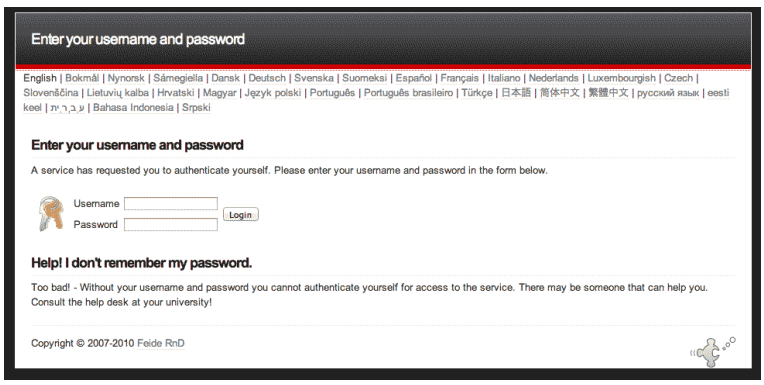

FIGURE 4.4 – Page d'authentification de SimpleSAMLphp

Comme dans l'exemple 4.2, pour chaque action il faut récupérer le formulaire, remplir les champs puis soumettre le formulaire.

4.2.2.4 Lister les sorties

Les sorties sont les différentes pages de l'application. Nous avons distingué quatre sorties :

— credentialsPage : la page d'accueil ;
— idpList : la page présentant les différents fournisseurs d'identité ;
— profilePage : la page de profil une fois que l'on est correctement identifié ;
— loggedOutPage : la page une fois l'utilisateur déconnecté.

Pour pouvoir générer le bon symbole de sortie suivant le contenu de la page, nous pouvons nous servir d'un élément de la page pour la détection. Dans SimpleSAMLphp, le titre de la page suffit pour détecter la page. On peut remarquer que se servir du titre de la page ne sera pas toujours un bon choix. Bon nombre d'applications gardent le même titre suivant la page.

Nous pouvons nous servir aussi de n'importe quel autre contenu de la page comme un ou plusieurs mots clés, mais cela nécessite de comparer chaque page manuellement pour en trouver un élément qui les distingue. Nous montrons dans la section 4.3 comment automatiser cette partie.

4.2.2.5 Les étapes d'une communication

Quand on est au niveau HTML, on est limité par la page que l'application nous fournit. C'est pourquoi essayer de soumettre le formulaire de connexion sur la page de profil n'a aucun sens. Seules les actions valides pour la page pourront être utilisées.

— en premier, nous devons récupérer la page d'accueil pour avoir le formulaire de choix du fournisseur d'identité ;
— puis, nous pouvons choisir un des fournisseurs d'identité de la liste ;
— sur la page d'authentification, nous pouvons essayer plusieurs couples d'utilisateur/mot de passe ;
— une fois connectés, nous pouvons utiliser la déconnexion.

4.2.2.6 Développement de l'adaptateur

Comme précédemment, nous avons trois principales fonctions :

— abstractToConcrete, qui vérifie que la page courante permet d'exécuter l'action puis l'envoi sinon une erreur est générée. On remarque ici que la fonction comprend aussi l'exécution de l'entrée.
— concreteToAbstract, qui se sert du titre de la page pour générer le symbole de sortie correspondant.

Développer un adaptateur de test pour une application WEB au niveau HTML est sûrement la méthode manuelle la plus rapide. Elle ne nécessite pas de connaître forcément l'application puisqu'il suffit de la parcourir pour en repérer les différents formulaires. Cependant, suivant la taille de l'application, même cette étape peut se révéler assez longue.

De plus, utiliser uniquement les actions du navigateur limite les entrées que l'on utilise pour une inférence. Dans SimpleSAMLphp, la phase d'authentification comprend plusieurs requêtes et redirections qui sont finalement cachées à l'utilisateur et donc à l'adaptateur. La précision du modèle s'en trouve réduite.

SimpleSAMLphp a très peu de requêtes, mais un grand nombre de paramètres qui sont encodés ou chiffrés selon un format spécifique. Ces transformations sont assez complexes pour que l'écriture d'un adaptateur niveau HTTP se révèle aussi compliqué puisqu'il nécessite de comprendre comment l'application forme ses paramètres d'entrées.

4.2.2.7 Niveau HTTP

Dans la section 3.1.4, nous avons présenté quelques expérimentations effectuées avec l'algorithme d'inférence du projet SPaCIoS. Parmi les applications inférées, nous avons choisi WebGoat, une plateforme d'apprentissage des vulnérabilités Web qui fournit un ensemble d'applications vulnérables (une par vulnérabilité).

Nous avons choisi l'application de démonstration des failles XSS de type persistante. Les requêtes produites par l'application étant relativement simples (comparé à SimpleSAMLphp), nous avons décidé d'écrire un adaptateur de type HTTP.

4.2.2.8 Choix de la bibliothèque

WebGoat fait partie des projets de l'OWASP pour apprendre la sécurité des applications Web en étudiant les failles. Chaque leçon est composée d'une application vulnérable, d'un tutoriel et de la solution (certaines en vidéos). La plupart des applications sont bâties sur la même base, mais chacune sert à montrer un type de vulnérabilités en particulier.

C'est une application de gestion des ressources humaines avec des actions simples telles que voir, créer, supprimer et éditer un profil. Chaque action est limitée aux

utilisateurs qui en ont le droit. Dans les applications WebGoat, l'utilisateur ayant le moins de droits est *Larry* qui a un statut d'employé. Celui qui a le plus de droits est *John* qui est administrateur. Le but de cette leçon est pour *Larry* de pouvoir effectuer des actions qui requièrent le niveau administrateur de *John*.

Chaque action envoie une requête HTTP de type POST vers une page unique qui gère les différentes actions grâce au paramètre *action*. Pour générer ces requêtes HTTP, nous avons créé un client Web léger en Java pour pouvoir contrôler tous les éléments des requêtes. Ce client est plus léger que celui d'HTMLUnit et permet de réaliser un grand nombre de requêtes plus rapidement.

4.2.2.9 Lister les entrées

L'application ne comporte que quelques pages différentes. Sur la page d'accueil, l'utilisateur est invité à s'authentifier pour avoir accès à son compte. Cette action sera représentée par le symbole *login*. Puis sur la page suivante, nous avons les actions de base décrites plus haut. Ces actions seront représentées par les symboles : *logout*, *viewProfile*, *editProfile*, *updateProfile*.

Bien que l'action de création de profil soit désactivée, un attaquant pourrait très bien la réactiver, mais la création de profil ne sera pas considérée par l'adaptateur. En effet, si WebGoat a désactivé cette action c'est parce qu'elle n'intervient pas dans la détection de la vulnérabilité de cette leçon.

4.2.2.10 Lister les sorties

Comme pour les adaptateurs de type HTML, nous devons lister les différentes pages de l'application. Chaque page correspond à un symbole abstrait pour l'algorithme d'inférence. Comme nous travaillons sur des EFSM, nous devons aussi localiser les paramètres importants des pages.

Une fois connecté au système, on peut voir que le nom de l'utilisateur apparaît en haut. C'est donc un paramètre de la page qui peut potentiellement représenter une faille XSS. Sur la page de profil, nous avons les différents champs du profil d'un employé. Chaque champ représente un paramètre de la page de profil, mais aussi un endroit potentiel où se trouve une faille XSS.

Nous avons donc défini quatre symboles de sortie pour les quatre pages rencontrées pendant l'exploration de l'application :

— listing : la page qui contient la liste des profils accessibles ;
— home : la page d'accueil qui demande de s'authentifier ;
— profilePage : la page de profil avec les différents champs concernant l'employé ;
— editionPage : la page d'édition d'un profil.

Contrairement à SimpleSAMLphp, les pages de l'application ont toujours le même titre. Nous avons donc cherché des éléments du code qui permettent de distinguer chaque page. Cela peut être un texte dans la page comme *Staff Listing Page* pour distinguer la page qui contient la liste de profils ou encore un élément de formulaire comme *Credit Card Limit* pour distinguer la page de profil d'un employé.

Même si la procédure pour lister les différentes sorties peut paraître simple, trouver les paramètres qui sont nécessaires l'est un peu moins. De plus, plus l'application contient un grand nombre de pages, plus cette étape sera longue.

4.2.2.11 Les étapes d'une communication

Avec un adaptateur de type HTTP, nous ne sommes plus limités par les actions que propose la page Web. Nous pouvons très bien envoyer une requête d'authentification sur la page d'un profil. Pour chaque action, nous avons recréé la requête HTTP correspondante en modifiant uniquement certains paramètres.

Évidemment, certaines actions seront invalides et généreront de la part du serveur un code d'erreur. La plupart du temps, l'application ne se sert pas de ce code, mais redirige l'utilisateur vers la page principale du site.

Par exemple, la transformation d'une requête abstraite, $login(john, john)$ sera la requête HTTP contenue dans le listing 4.3.

Listing 4.3– Requête concrete HTTP de login WebGoat

```
POST /WebGoat/attack?Screen=96&menu=900 HTTP/1.1
Cookie: JSESSIONID=4568B5BCBAFE21434958AFAB2DDB3E6B; Path=/WebGoat/
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=

employee_id=111&password=john&action=Login
```

A l'inverse, la réponse concrète du listing 4.4 sera transformée en réponse abstraite $listing(larry)$ grâce au texte $StaffListingPage$ qui distingue cette page.

Listing 4.4– Réponse concrete HTTP de WebGoat

```
<div class="lesson_title_box">
  Welcome Back<span>Larry</span> - Staff Listing Page
</div>
<p>Select from the list below</p>
<form id="form1" name="form1" method="post" action="attack?>
  <table width="60%" border="0" cellpadding="3">
  <tr>
    <td>
    <label>
      <select name="employee_id" size="11">
      <option selected value="101">Larry Stooge (employee)</option>
```

```
        </select>
      </label>
      </td>
      <td>
      <input type="submit" name="action" value="SearchStaff" />
      <br>
      <input type="submit" name="action" value="ViewProfile" />
      <br>
      <input type="submit" name="action" value="Logout" />
      </td>
    </tr>
    </table>
  </form>
```

4.2.2.12 Développement de l'adaptateur

Maintenant nous avons tous les éléments nécessaires à la construction de l'adaptateur : les entrées, sorties et les fonctions qui permettent de passer du niveau abstrait au niveau concret et vice versa.

Nous pouvons développer notre adaptateur de test en Java en suivant le même modèle que les précédents adaptateurs. L'adaptateur sera découpé en trois fonctions principales :

— abstractToConcrete, qui sert à convertir une requête abstraite en enquête HTTP ;
— concreteToAbstract, qui sert à convertir une réponse concrète HTTP en réponse abstraite ;
— exécute, qui sert à envoyer une requête HTTP et retourne la réponse HTTP.

Le développement d'un adaptateur au niveau HTTP demande plus d'effort qu'au niveau HTML puisqu'il faut aussi s'occuper de construire les requêtes. Mais nous gagnons en précision parce que nous avons le contrôle sur chaque paramètre de la requête, y compris celui contenu dans les entêtes comme les cookies.

Là encore, selon la taille de l'application, trouver toutes les actions et les pages peut prendre un certain temps. De plus, ajouter tous les paramètres pourrait ralentir la procédure d'inférence parce que seul un sous-ensemble des paramètres a un impact sur le flot de contrôle de l'application.

Contrairement au FSM où les paramètres sont intégrés à l'adaptateur, nous devons ici fournir un ensemble de valeurs pour les paramètres de chaque symbole d'entrée. Nous ne devons pas être exhaustif pour le choix des valeurs mais plutôt choisir un sous-ensemble réduit de valeurs permettant à l'exécution d'avoir le plus de comportements différents. Cela améliorera la découverte de l'application ainsi que les résultat de l'étape de fouille de données effectuée à la fin. Par exemple, pour

un couple de pseudo/mot de passe, nous devons fournir au minimum des crédentiels valides et invalides.

4.3 Génération automatique d'adaptateur de test pour application Web

Dans la section 4.2.2 nous avons vu comment construire les adaptateurs de test manuellement. Travailler avec des EFSM implique de devoir non seulement trouver les entrées et sorties de l'application, mais aussi de trouver les différents paramètres qui les composent. Pour les applications Web, ces paramètres sont dépendants du choix du développeur tant par leur nombre que par leur format. S'il est possible de parcourir l'application manuellement pour les trouver, cela devient vite complexe quand la taille de l'application, les techniques de développement ou le langage évoluent. Le temps gagné par l'inférence de modèle est compensé par le temps perdu à la création de l'adaptateur de test.

Dans le cadre du projet SPaCIoS, nous avons développé une méthode pour générer automatiquement des adaptateurs pour les applications Web. Elle comprend une procédure d'extraction des entrées et sorties paramétrées ainsi que d'un collecteur qui est chargé d'appliquer cette procédure sur le plus de pages possible du site. Bien que la méthode soit entièrement automatique, elle nécessite au préalable que l'utilisateur fournisse les données que le collecteur ne pourra deviner ou générer, comme des crédentiels valides ou des valeurs particulières pour les formulaires.

4.3.1 Extraction des entrées paramétrées

Les applications Web récentes sont la plupart du temps composées de plusieurs parties. Une partie back-end permet d'administrer l'application et qui est accessible à un nombre restreint de personnes ; une partie front-end qui est publique. Quelque soit le langage utilisé par ces deux parties, du point de vue de l'utilisateur, l'application est toujours disponible en tant que page Web, c'est-à-dire qu'il s'agit de code HTML. Le fait d'analyser le code HTML est donc suffisant pour en repérer les différents formulaires et en extraire les entrées correspondantes.

En écrivant l'adaptateur manuellement, nous avons déjà eu une intuition quant à la définition d'une entrée paramétrée. Ces entrées sont les différentes actions qui sont rendues disponibles à l'utilisateur par l'application. On les retrouve généralement sous forme de liens avec la balise HTML *a* ou formulaires *form*.

Nous pouvons donc définir une entrée comme étant un tuple (M, A, P) où M est la méthode HTTP utilisée, A l'adresse de la page qui gère la requête et P un ensemble fini de couples $(nom, valeurs)$ où nom est le nom d'un paramètre et

valeurs un ensemble de chaînes de caractères représentant les différentes valeurs possibles pour ce paramètre.

Pour les liens hypertextes, la méthode est fixée à GET, A est la page sur laquelle pointe le lien et P sont les différents paramètres de la requête que l'on retrouve dans l'adresse du lien. On remarque qu'un paramètre, même s'il est utilisé dans un lien, peut avoir plusieurs valeurs possibles. C'est le cas dans le listing 4.5 pour un lien et dans le listing 4.6 pour un élément de formulaire.

Listing 4.5– Paramètre à plusieurs valeurs pour un lien

```
http://test/index.php?id=1&id=2&id=3&id=4
```

Listing 4.6– Paramètre à plusieurs valeurs pour un formulaire

```
<select>
  <option value="1">A</option>
  <option value="2">B</option>
  <option value="3">C</option>
  <option value="4">D</option>
</select>
```

Le fait de retrouver plusieurs valeurs pour un paramètre dans un lien a d'ailleurs été la source d'attaque nommée pollution de paramètre HTTP [Balduzzi 2011].

Étant donnée une entrée sous forme (M, A, P), nous pouvons reconstruire la requête HTTP concrète correspondante facilement.

En pratique, il peut arriver que les pages ne soient pas correctement formées ou qu'elles ne respectent pas totalement le standard. Cela peut avoir un impact sur la détection des formulaires avec certains parseurs HTML. Pour prendre en compte le maximum d'applications Web, nous avons fait en sorte de détecter aussi les formulaires non conformes au standard en nous basant uniquement sur les balises HTML correspondant au formulaire, indifféremment de leur emplacement.

Dans le listing 4.7, nous avons un exemple d'entrée qui est équivalente à ($POST$, *buy.php*, $[price = [10, 20, 30]]$).

Listing 4.7– Exemple d'entrée

```
<form method="post" action="buy.php">
  <select name="price">
    <option value="10">A</option>
    <option value="20">B</option>
    <option value="30">C</option>
  </select>
</form>
```

Les applications récentes proposent un grand nombre de ressources dans une page grâce à des listes ou des tableaux (par exemple une galerie d'images). En

plus de la stratégie de parcours de l'application, nous avons besoin d'une stratégie pour éviter de parcourir des éléments équivalents qui n'auront pas d'intérêt pour le collecteur.

Une galerie d'images peut présenter sous forme de tableau un grand nombre de liens avec la même adresse, mais avec un paramètre différent, comme dans le listing 4.8.

Listing 4.8– Exemple de galerie

```
<li><a href="image.php?id=1">Image 001</a></li>
<li><a href="image.php?id=2">Image 002</a></li>
<li><a href="image.php?id=3">Image 003</a></li>
<li><a href="image.php?id=4">Image 004</a></li>
```

Pour l'adaptateur, visiter plus d'un lien n'est pas efficace et n'apportera aucune sortie ou entrée en plus. Dans cet exemple, nous avons qu'une entrée avec un paramètre à quatre valeurs qui conduit à une sortie paramétrée par l'adresse de l'image. Pour éviter de parcourir chaque valeur du paramètre *id*, nous avons défini une heuristique qui prend en compte les différences entre les entrées et les sorties. Si au moins 3 requêtes ayant la même adresse, un seul paramètre x de différent et conduisant à la même sortie, les autres valeurs de x ne seront pas testées et ces requêtes sont appelées requêtes équivalentes.

Dans l'exemple du listing 4.8, seuls les trois premiers liens seront parcourus. Plus généralement, cela permet d'éviter de parcourir les listes d'éléments des applications.

La méthode d'extraction des entrées peut se décrire avec l'algorithme 6.

Algorithme 6 : Extraction_Entrées(P, I_0) où P est une page web et I_0 est l'ensemble initial d'entrées.

```
1 I = I_0;
2 foreach action a ∈ P do
3 |   p_a = extraire_paramètres(a);
4 |   i_a = créer_entrée(générer_nouveau_symbol(), p_a);
5 |   if i_a ∉ I ∧ i_a ≢ i ∈ I then
6 |   |   I = I + i_a;
7 |   end
8 end
```

4.3.2 Extraction des sorties

Définir une sortie est plus compliqué puisque nous ne pouvons pas assigner un symbole de sortie pour chaque page dont l'adresse serait différente. Dans la

création d'adaptateurs manuelle, nous avons choisi de retenir uniquement les pages distinctes, celles qui sont censées représenter un seul ensemble d'informations qui sont ces paramètres, par exemple la page servant à afficher la liste des profils ou la page de connexion). Nous devons donc nous concentrer sur la structure de la page et non son contenu textuel ou son adresse puisqu'on peut appeler deux fois la même page avec des paramètres différents (deux adresses différentes) ce qui renverra la même page avec deux contenus différents.

Pour identifier les différentes pages distinctes, nous devons donc conserver uniquement les informations liées à la structure de la page ainsi que les entrées. Deux pages ayant la même structure et les mêmes entrées seront considérées comme identiques. Pour cela, nous avons défini une représentation de page sous forme d'arbre, appelée arbre de page (PageTree). Ce PageTree diffère de celui de [Doupé 2012] dans les éléments qui constituent l'arbre.

Un PageTree est un arbre dont les nœuds sont les balises de la page liée à la structure ou à des entrées. Un grand nombre de balises sont donc supprimées de la page comme les balises pour choisir la police ou la couleur du texte. La liste des balises non prise en compte est définie dans le listing 4.9.

Listing 4.9– Liste des balises non prises en compte pour le PageTree

```
"#document", "#text", "span", "font", "a", "center", "bold", "italic",
     "style", "base", "param", "script", "noscript", "b", "i", "tt", "
    sub", "meta", "title", "head", "tbody", "sup", "big", "small", "
    img", "br", "tr", "td", "option", "#comment", "#data", "strong", "
    li"
```

On peut remarquer que les éléments de listes et de tableaux ne sont pas pris en compte. En effet, une même page peut afficher des listes de différentes tailles. Même si ici, la structure de la page s'en trouve modifiée, cela provient plus des paramètres qui définissent la liste que de la page elle-même.

Dans la figure 4.5, nous avons une représentation simplifiée d'un PageTree pour une des pages de l'application WebGoat de la figure 4.6.

Sur la page Web de la figure 4.6, l'organisation est faite avec un tableau (< *table* >) qui contient à gauche un élément de type *label* pour présenter l'élément de type *select* et à droite un ensemble de champs d'entrée de formulaire regroupés dans un élément de type *div*.

L'idée derrière le PageTree est de pouvoir faire de la classification de PageTree pour savoir si une page est une nouvelle sortie ou pas. En visitant les pages d'une application, nous pouvons ainsi construire petit à petit l'ensemble des différentes pages du site. Cette partie sera développée dans la section 4.3.4.1.

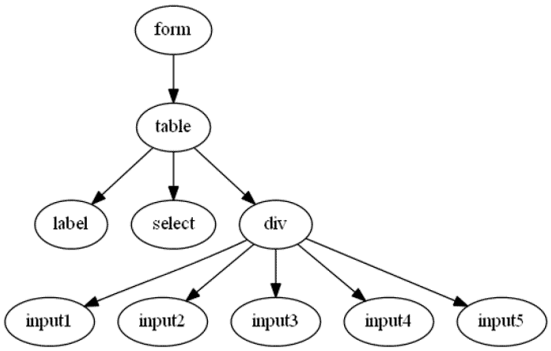

FIGURE 4.5 – Exemple de PageTree

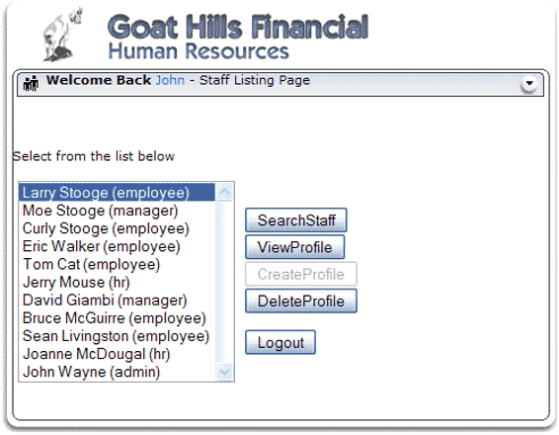

FIGURE 4.6 – Page correspondante au PageTree

4.3.3 Extraction des paramètres des sorties

Pour les entrées, les paramètres sont relativement simples à extraire puisqu'ils sont donnés dans la balise représentant l'entrée. En ce qui concerne les sorties et les pages, une seule page n'est pas suffisante pour identifier les paramètres.

Tout d'abord, nous pouvons définir un paramètre d'une page comme étant le contenu texte d'une balise qui peut changer pour une même page. Nous avons vu précédemment comment reconnaître une même page même si le contenu texte (les paramètres) change.

À chaque fois qu'une requête conduit à une page, nous allons envoyer plusieurs fois cette requête avec différents paramètres d'entrée pour récupérer différentes versions de la même page. Ensuite il suffit de vérifier que nous avons bien eu la même page, si c'est le cas, comme nous avons la même structure de page, il suffit de comparer les différentes versions des pages pour localiser la position des paramètres comme dans la figure 4.7.

```
<div id="#profile">
   <span id="#username">Alice</span>
   <span id="#money">200</span>
</select>
```

```
<div id="#profile">
   <span id="#username">Bob</span>
   <span id="#money">100</span>
</select>
```

FIGURE 4.7 – Deux versions de la même page pour localiser les paramètres

Nous pouvons résumer la procédure d'identification des paramètres dans l'algorithme 7 qui prend en entrée une entrée I, le préfixe d'accès (une séquence d'entrée) I_{prev} et une page P. L'inférence de type utilise certaines heuristiques pour détecter,

par exemple, les nombres et chaînes. La génération de valeur se fait par mutation pour les chaînes de caractères (modification aléatoire d'un caractère) et par incrément pour les nombres.

Algorithme 7 : Detection_Paramètres(I_{prev}, I, P)

```
Result : Une liste L de localisation des paramètres
1  L = ∅;
2  foreach paramètre p_i de I do
3      t = inferType(p_i);
4      for n = 0 ; n <NB_FUZZ ; n++ do
5          tmpVal = generateVal(t);
6          I' = I;
7          mettre_à_jour(I', i, tmpVal);
8          P' = exécuter(I_prev.I');
9          if PageTree(P') == PageTree(P) then
10             ajouteLesChampsDifférents(L, P, P');
11         end
12         else
13             ajouteValeurPourParamEntrée(tmpVal, I, p_i);
14         end
15     end
16 end
```

Nous avons donc l'entrée I qui, après la séquence d'entrées I_{prev}, conduit à la page P. Pour chaque paramètre de I, nous allons essayer plusieurs valeurs. Si le PageTree obtenu avec une nouvelle valeur est différent de celui de P alors nous avons une valeur utile pour l'inférence qui peut conduire à une nouvelle page et un nouvel état potentiel. Dans l'autre cas, nous localisons les différences dans les champs et nous les ajoutons à L.

4.3.4 Parcours de l'application

Maintenant que nous avons défini une procédure pour extraire les entrées d'une page ainsi qu'identifier si une page Web est une nouvelle sortie ou non, il nous manque un moyen d'appliquer ces méthodes sur les pages de l'application. Pour cela, nous avons développé un collecteur qui prend en compte les états du site.

Le but du collecteur est de parcourir le plus de pages possible d'un site Web donné. Les collecteurs classiques partent d'une adresse puis explorent chaque lien trouvé dans la page. Ils s'arrêtent quand tous les liens ont été explorés avec un parcours en profondeur ou en largeur. Cette méthode serait efficace si le but du collecteur était d'indexer le contenu des pages comme le font les moteurs de recherche par exemple. Mais dans notre cas, nous devons seulement visiter le plus de pages

distinctes différentes.

L'exploration classique ne fonctionne plus si, durant l'exploration, une requête a changé l'état de l'application, rendant le reste de l'exploration impossible. L'ordre de visite des pages est donc important. Par exemple, cela arrive lorsque le collecteur est authentifié sur le site puis qu'il visite une page de déconnexion. Le reste des adresses à visiter n'est plus forcément disponible puisque le collecteur s'est déconnecté. C'est pourquoi nous devons considérer l'état de l'application.

Nous avons donc choisi de parcourir le site par séquence de requêtes avec un parcours en largeur. Au lieu de visiter une adresse suivante à chaque fois, nous visitons le chemin complet. Le fait de parcourir en largeur réduit la longueur moyenne des chemins. De plus, il y aura forcément des pages qui seront demandées plusieurs fois, ces pages serviront à la détection de paramètres de sortie.

Dans le listing 4.10, nous avons un exemple de parcours d'une application contenant cinq pages : *index*, *login*, *view*, *search*, *logout*. Le fait de parcourir par chemin permet de pouvoir visiter la page *edit* même si le collecteur a visité la page *logout* précédemment.

Listing 4.10– Exemple de parcours du collecteur

```
1. index
2. index ; login
3. index ; login ; view
4. index ; login ; search
5. index ; login ; view ; logout
5. index ; login ; view ; edit
```

4.3.4.1 Génération des symboles de sorties

À chaque visite d'une page, son PageTree est construit et comparé à l'ensemble des PageTree distincts déjà visités. Si aucun élément ne correspond, cette page est considérée comme nouvelle et un nouveau symbole de sortie lui sera attribué.

Nous pouvons résumer la procédure de génération des symboles de sortie par l'algorithme 8 qui prend en entrée une page P et l'ensemble $Pages$ des pages déjà identifiées et qui retourne un booléen selon si la page est nouvelle ou pas.

4.3.5 Optimisation pour l'inférence

Le collecteur a été conçu pour générer une abstraction dans le but d'effectuer une inférence sur l'application. Pendant le parcours, il peut arriver certains cas où nous avons des informations qui peuvent servir à cette inférence.

Algorithme 8 : Générer_Symbole_Sortie($Pages$, P)

```
Result : Vrai ou faux selon si la page est nouvelle ou pas
P' = supprime_données(P);
Page_tree = PageTree(P');
foreach page_i de Pages do
    if compareDFS(Page_i, Page_tree) == Vrai then
        retourne Faux;
    end
end
retourne Vrai;
```

— Durant la recherche de paramètres de sorties, il est possible qu'une variation des paramètres de la requête conduise à une nouvelle page (nouvelle sortie). Dans ce cas, la combinaison de paramètres utilisés est sauvegardée pour pouvoir être réutilisée pendant l'inférence pour potentiellement découvrir un nouveau comportement.

— Dans le but de réduire le nombre de paramètres pour réduire le temps nécessaire à l'inférence, nous considérons uniquement les paramètres de sortie déterministes, c'est-à-dire uniquement les paramètres qui dépendent des entrées. Cela permet ainsi de réduire le nombre de paramètres de sortie, mais aussi de garder uniquement les paramètres pour lesquels on a un certain contrôle sur la valeur ce qui sera utile notamment lors de la détection d'attaques par injections de code.

— La même page peut être appelée avec différents ensembles de paramètres. Dans le but d'éviter de multiplier les combinaisons de paramètres, il est possible de forcer la fusion des paramètres qui sont gérés par la même page. Cela réduit le nombre de paramètres et rend l'inférence plus rapide.

4.4 Génération de l'adaptateur de test

Une fois le parcours du site terminé, nous avons l'ensemble des entrées et l'ensemble des PageTree avec la position de leurs paramètres respectifs. Ces informations sont ensuite sauvegardées dans un fichier XML et peuvent être utilisées par n'importe quel outil supportant ce format, et ainsi que par SIMPA, l'outil d'inférence du projet SPaCIoS, qui fournit un adaptateur générique qui permet de charger ce fichier XML.

Chapitre 5

Inférence d'automates étendus d'applications Web

Ce chapitre présente un algorithme d'inférence de modèle pour les applications Web et dans le but de tester leur sécurité. Il a été pensé pour combiner les différentes qualités des algorithmes présentés dans les chapitres précédents, tels que l'inférence sur les données et une certaine souplesse au niveau des contraintes sur les entrées. De plus, il se sert dynamiquement des méthodes de génération d'adaptateurs de test pour communiquer avec l'application.

Sommaire

Les méthodes existantes d'inférence décrites dans le chapitre 3 sont efficaces dans certaines conditions selon le type d'application ou les entrées/sorties, mais elles ne sont pas totalement adaptées aux applications Web. De plus, les méthodes de test de la sécurité basées sur les modèles s'adressent à un public pas forcément formé à la modélisation des applications d'où le besoin d'une méthode la plus automatisée possible sans pour autant perdre en qualité de modélisation et en expressivité pour

pouvoir, ensuite, y détecter des vulnérabilités.

L'algorithme présenté dans cette section est construit en reprenant les avantages des précédentes méthodes tout en l'améliorant pour le contexte de la sécurité et des applications Web. Le but est de fournir un modèle comprenant les divers mécanismes de sécurité tout en restant compréhensible et proche de l'application pour pouvoir y faire référence et la corriger si nécessaire.

5.1 Motivations

Nous avons vu précédemment que la modélisation des applications Web est assez complexe. La méthode d'inférence de la section 3.1 utilise un modèle adapté aux applications Web, mais nécessite de connaître l'interface complète de l'application avant l'inférence. Pour une application Web, même en s'aidant de techniques de collection (ou crawling) avancées, il n'est pas toujours possible de garantir que toute l'application ait été explorée que ce soit en terme d'entrées ou de sorties. L'inférence produira donc un modèle partiel de l'application. Bien qu'il soit possible ensuite à l'utilisateur de détecter les régions de l'application inexplorées et d'ajouter manuellement les informations permettant d'accéder à ces régions, il devra ensuite recommencer toute la procédure d'inférence puisque les tables d'observations seront devenues obsolètes.
L'algorithme de la section 3.1 ne considère que la dernière valeur pour chaque paramètre uniquement alors que certaines pages Web peuvent être réutilisées avec différents paramètres et l'association des différents résultats peut conduire à un nouveau comportement. L'algorithme devra donc être étendu autant au niveau de l'inférence que de la fouille de données pour pouvoir supporter les m-dernières valeurs de chaque paramètre. De plus, le nombre de valeurs des paramètres grandissant, l'algorithme doit avoir un moyen de considérer les meilleures valeurs en premier.

L'algorithme Z-Quotient est par construction plus souple que celui pour EFSM de la section 3.1. Il permet notamment de ne pas avoir toutes les entrées pour commencer l'inférence puisqu'il est capable de gérer de nouvelles entrées découvertes pendant l'inférence. Ces nouvelles entrées sont susceptibles de mener à de nouvelles sorties qui pourront être créées à la volée avec des techniques de partitionnement de pages Web. Par contre, le modèle utilisé actuellement, la machine de Mealy, n'est pas assez expressif pour les applications Web. L'application fonctionnant avec des requêtes et réponses paramétrées, il est préférable d'utiliser une machine à états finie étendue. Cet algorithme sera donc étendu aux EFSMs.
Du côté des paramètres, les applications Web peuvent en comporter un grand nombre dont certains qui ne sont pas nécessaires pour la partie contrôle de l'appli-

cation. En plus du nombre de paramètres et contrairement aux systèmes tels que les protocoles, où les différents champs constituant les requêtes sont d'un format fixe, les applications Web permettent d'utiliser toutes sortes de structures de données de différentes tailles et formes. La méthode d'inférence devra être adaptée pour détecter et utiliser ces types de données.

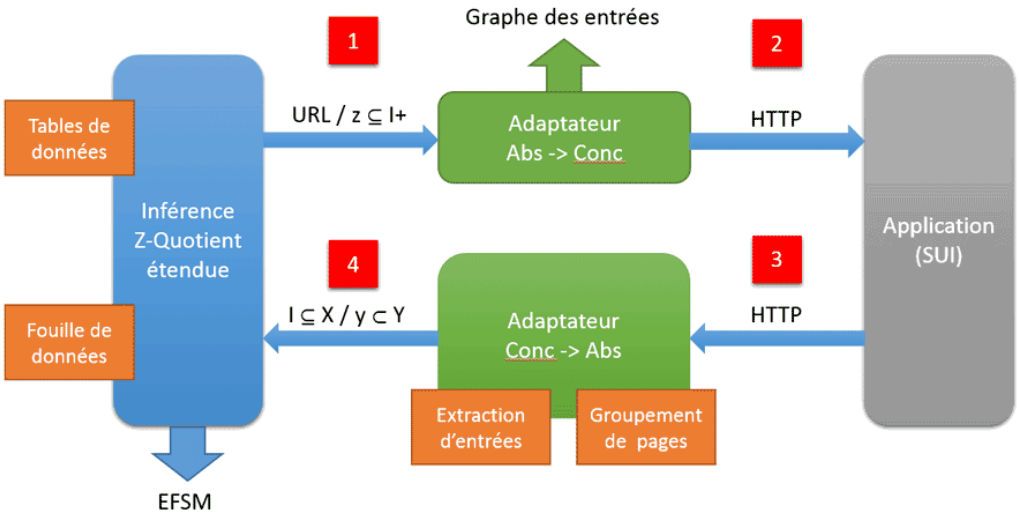

FIGURE 5.1 – Schéma de l'algorithme Z-Quotient étendu

L'algorithme se sert de l'adaptateur pour explorer les URL et concrétiser les entrées paramétrées (1). Une requête HTTP est ensuite émise pour l'application (2) et la réponse HTTP est renvoyée à l'adaptateur (3). Les nouvelles entrées et la sortie sont extraites et des symboles correspondants sont générés à la volée puis renvoyés à l'algorithme d'inférence. Nous obtenons un EFSM en sortie ainsi qu'un graphe des entrées correspondant à une carte de l'application et qui sera utilisée dans le chapitre 6.

5.2 Le modèle EFSM à m-variables par paramètre

Dans la section 3.1, la machine à états finie étendue s'est montrée suffisamment expressive pour représenter les applications Web en gardant une taille de modèle réaliste. Les EFSM considérés ici ont aussi des paramètres pour chaque symbole, que ce soit d'entrée ou de sortie. En ce qui concerne l'ensemble de variables qui représente l'état courant du système, nous considérons plusieurs valeurs pour chaque paramètre.

Dans chaque état, il y a donc un ensemble de variables indexées d'une taille

dépendante de l'état de l'automate. En effet, il est possible que certaines variables représentant des paramètres de sorties ne soient accessibles qu'à partir d'un certain état du système et non pas dès le début. Au final, chaque état dispose de son ensemble de variables dans lequel il peut puiser pour sélectionner la bonne transition à exécuter. D'un côté cela permet de ne pas utiliser, dans les gardes et fonctions de sorties, des valeurs qui n'auraient pas été utilisées, ce que l'on pouvait retrouver avec le précédent algorithme de la section 3.1. D'un autre côté, cela facilite l'inférence des gardes et fonctions de sortie en réduisant le nombre d'attributs.

Nous pouvons voir apparaître des cas où plusieurs valeurs de paramètres de sortie d'un symbole sont utilisées dans la partie contrôle. Imaginons par exemple un mécanisme d'authentification qui utiliserait une série de questions. Chaque page demande une question puis dirige vers la prochaine page qui est, du point de vue symbole de sortie, la même. Au bout de m questions, si toutes les précédentes réponses sont vérifiées et si elles sont toutes correctes, l'accès est autorisé. Dans ce cas, le symbole de sortie de la dernière page de question utilise les m dernières valeurs d'un paramètre d'une certaine entrée. De ce fait, garder les m dernières valeurs devient nécessaire. Évidemment, la partie inférence de gardes et de fonction de sortie est adaptée pour ces multiples valeurs.

Nous faisons donc quelques suppositions en ce qui concerne la partie de gestion de ces variables :

— il y a n variables associées à chaque paramètre ;
— une variable qui n'a pas été valuée a initialement comme valeur le symbole $\perp$;
— seules les m dernières valeurs des paramètres sont conservées ;
— l'affectation d'une variable ne se fait qu'au moment où le paramètre correspondant est utilisé.

Une fois ces suppositions établies, nous pouvons reprendre le modèle d'EFSM défini dans 3.1 et adapter le concept de paramètre de sortie non déterministe (nonces) avec plusieurs valeurs pour chaque paramètre. Nous considérons bien des valeurs de paramètres de sorties non déterministes et non pas les entrées.

Dans les définitions suivantes, soit X et Y des ensembles finis de symboles d'entrée et de sortie tels que $|X \cup Y| = n$, P un ensemble fini de paramètres et tels que $|P| = n$ et V un ensemble de n vecteurs de m éléments. Nous avons donc :

$$P = \{p_0, \ldots, p_{n-1}\} \text{ et } V = \begin{vmatrix} p_0^{-m-1} & p_1^{-m-1} & \cdots & p_{n-1}^{-m-1} \\ \vdots & \vdots & \cdots & \vdots \\ p_0^{-1} & p_1^{-1} & \cdots & p_{n-1}^{-1} \\ p_0^{0} & p_1^{0} & \cdots & p_{n-1}^{0} \end{vmatrix}.$$

Ainsi nous pouvons associer à chaque paramètre P_i un ensemble de m variables $P_i^0, P_i^{-1}, \cdots, P_i^{-m}$ qui auraient le même indice et un indice représentant la m-ième

dernière valeur. Pour $z \in X \cup Y$, nous notons P_z l'ensemble de ses paramètres associés, $P_{z,i}$ le i-ième paramètre du symbole z, D_z l'ensemble des valuations des ses paramètres et $D_{z,i}$ les m valeurs du paramètre z et enfin $D_{z,i,m}$ la m-ième valeur du i-ième paramètre du symbole z. Nous avons donc bien nos entrées et sorties qui possèdent chacune un ensemble fini de paramètres qui eux-mêmes possèdent un ensemble fini de valeurs.

Une machine à états finie étendue (EFSM) M, définie sur X, Y, P, V et la fonction associée p, est une paire (S, T) d'un ensemble fini d'états S, qui contient l'état initial appelé s_0, et un ensemble finis de transitions T entre les états dans S, tels que chaque transition $t \in T$ est un tuple (s, x, G, op, y, up, s'). La définition de l'EFSM est la même que celle définie dans la section 3.1.2. Le seul changement se situe au niveau de la fonction de mise à jour des paramètres, la fonction up sert uniquement à décaler les valeurs associées à un paramètre puis modifier la dernière valeur.

Dans un état $s \in S$, tous les symboles d'entrée ne sont pas forcément valides et acceptés par l'EFSM. Par exemple, se déconnecter d'un site peut n'être possible que si l'on s'est préalablement connecté, ou une page peut ne pas contenir toutes les actions possibles sur le site. Soit un symbole d'entrée $x \in X$ qui n'est pas accepté par l'EFSM, nous utilisons une transition spéciale avec comme symboles de sortie $\Omega \in Y$ pour représenter le fait que ce symbole d'entrée ne soit pas accepté.

Selon la définition, il est aussi possible qu'un symbole d'entrée ou de sortie ne possède pas de paramètre. Dans ce cas nous introduisons une autre notation spéciale pour représenter le fait que ce symbole n'ait pas de paramètre. Soit un symbole d'entrée ou de sortie $z \in X \cup Y$ qui ne possède pas de paramètres, c'est-à-dire $P_z = \{\}$, nous utilisons ω pour représenter D_z.

Considérant les suppositions faites précédemment à propos des variables, soit d la valeur du paramètre x, v les valeurs des variables V, et $d' = op(d, v)$ la valeur des paramètres de y selon la fonction op, la fonction up met à jour les plus récentes valeurs respectives des paramètres x et y à d et d' et supprime la plus ancienne. Si x et y partagent un paramètre commun, la variable correspondante est elle aussi mise à jour avec la valeur du paramètre de sortie. La fonction up est définie dans l'algorithme 9. La liste des valeurs des paramètres est une liste FIFO (First In First Out).

Nous pouvons aussi imaginer stocker uniquement les valeurs de paramètres différentes pour éviter d'avoir des doublons dans les listes. La gestion de ces listes serait plus complexe notamment pour connaître quand la valeur a été utilisée (on ne pourra plus se baser sur l'indice dans la liste).

Nous définissons ensuite une *entrée paramétrée*, c'est-à-dire un symbole d'entrée avec son ensemble de paramètres qui ont chacun une valeur, de la façon suivante : $x(d_x)$ où x est le paramètre d'entrée, et d_x les valeurs pour chaque paramètre de

Algorithme 9 : Update

```
Soit z ∈ (X ∪ Y) et i l'index d'un paramètre de z;
for v = m − 1 à 0 do
    D_{z,i,v+1} = D_{z,i,v};
end
D_{z,i,0} = op(d, v);
```

x. Une *séquence d'entrées paramétrées* est une liste d'entrées paramétrées. De la même façon, nous définissons une *sortie paramétrée* ainsi qu'une *séquence de sorties paramétrées*.

Comme pour l'algorithme Z-Quotient, nous utilisons ici un ensemble Z constitué de séquences d'entrées au lieu de l'ensemble de caractérisation. Pour distinguer deux états, nous utilisons une relation de Z-équivalence qui a été définie sur les états. Bien que nous utilisons ici des entrées et sorties paramétrées, les séquences de Z ne comporteront uniquement des symboles non paramétrés, autrement dit, pour chaque Z_i appartenant à $Z, Z_i \subseteq I^*$.

Un EFSM M est dit *déterministe*, si chaque paire de transitions sortantes d'un même état et ayant le même symbole d'entrée, ont des gardes mutuellement exclusives ; *observable* si, pour chaque état, toutes les transitions sortantes ayant la même entrée doit avoir une sortie distincte. Nous considérons dans cette méthode uniquement des EFSM déterministes et observables à inférer .

5.3 Z-Quotient initial étendu

Nous supposons, comme dans l'algorithme de la section 3.1 que le système se comporte comme un EFSM et que nous sommes capable de lui envoyer des requêtes et d'en observer les réponses. L'application Web se comporte comme tout programme de façon déterministe, mais elle peut, dans ses paramètres, renvoyer des valeurs non déterministes (générées aléatoirement) du côté serveur.

Comme pour l'algorithme de la section 3.2, nous n'avons pas forcément besoin de connaître un sous-ensemble non vide des entrées pour commencer l'inférence. Si nous suivons l'algorithme d'inférence Z-Quotient dans le cas où nous ne possédons pas d'information sur les entrées, c'est-à-dire $I = \emptyset$, alors nous passons directement à la boucle de recherche et de traitement des contre-exemples.

Dans le cas des applications Web et comme nous considérons des EFSM, la recherche de contre-exemples peut être assez coûteuse et nécessite de nombreuses requêtes Web avec plusieurs valeurs de paramètres. Pour construire le Z-Quotient

initial étendu, nous allons aussi explorer l'application en même temps pour en extraire des entrées et sorties grâce aux méthodes présentées en 4.3. Le but étant d'utiliser le maximum d'information sans avoir recours aux contre-exemples. Les pages explorées à $I = \emptyset$ ne sont pas des entrées qui effectuent une action, mais uniquement des points d'entrée de l'application, c'est pourquoi elles ne sont pas dans I initialement.

Nous avons donc plusieurs configurations selon la valeur de l'ensemble I au début de l'inférence :

— D'un point de vue pratique, prendre $I = \emptyset$ permet de nous affranchir d'une exploration manuelle de l'application Web et de construire ainsi un outil comparable aux autres outils du domaine du test de la sécurité Web qui partent de peu d'information.

— Suivant les connaissances de l'utilisateur, il peut connaître une ou plusieurs entrées, c'est-à-dire $I \subset X$. Fournir certaines entrées peut améliorer l'exploration de l'application et accélérer la convergence vers un Z-Quotient initial étendu. D'un autre côté, l'utilisateur doit connaître ce sous-ensemble, ce qui peut nécessiter un certain effort.

— Avoir $I = X$ implique de connaître entièrement l'interface de l'application ainsi que les différentes valeurs pour chaque paramètre. C'est justement une des limitations de l'approche SPaCIoS et de l'algorithme pour EFSM section 3.1 qui se basait sur un collecteur pour récupérer les entrées et sorties alors que le collecteur ne pouvait pas forcément tous les récupérer. Il est toujours possible, pour certaines applications, que le testeur connaisse entièrement I et qu'il guide le collecteur, en lui fournissant les valeurs de paramètres nécessaires, jusqu'à avoir $I = X$ mais là encore, cela nécessite un certain effort.

L'algorithme Z-Quotient est paramétrable selon l'ensemble Z. Dans le reste de cette section, nous allons considérer que $Z = \emptyset$ au début de l'algorithme. Nous verrons ensuite dans le chapitre 6 comment nous pouvons cibler certaines vulnérabilités en définissant Z avec différentes séquences d'entrées.

Cet algorithme a été pensé pour fonctionner avec la première configuration, c'est-à-dire $I = \emptyset$ et avec $Z = \emptyset$. De cette façon, toute la partie de l'adaptateur de test est laissée à l'algorithme et non à l'utilisateur qui devra uniquement remplir le pool de valeurs de paramètres à utiliser pendant l'exploration.

5.4 Inférence d'un Z-Quotient initial étendu

L'algorithme général reprend celui du Z-Quotient, mais plusieurs algorithmes provenant de la partie génération d'adaptateur de test ou de l'algorithme SPaCIoS viennent le compléter. Nous supposons que le système peut être utilisé en boite noire, c'est-à-dire envoyer des requêtes et observer les réponses, et peut se modéliser sous la forme d'un EFSM. Il est aussi possible de remettre à zéro le système avant chaque exécution de séquence d'entrées.

5.4.1 Arbre d'observation étendu

Nous pouvons déjà définir l'arbre d'observation étendu qui contiendra les traces ainsi que l'EFSM inféré :

Définition 3 *Soit un ensemble U de traces clos par préfixe, observé sur l'EFSM et définis sur l'ensemble d'entrées paramétrées I et de l'ensemble de sorties paramétrées O, l'*arbre d'observation étendu *est l'EFSM $(U, \varepsilon, I, O, T_U)$, où l'ensemble des états est U et chaque $t \in T_U = (U_s, I, G(P, V, o), (P, V, S), O, U_f)$. Avec U_s et U_f les états initial et final, I et O sont les symboles d'entrée et de sortie et $(P, V et S)$ une structure de données qui représente les paramètres d'entrées, les précédents paramètres et ceux de sorties ainsi que o un symbole de sortie.*

$$P = \{p_0, p_1, \ldots, p_{n-1}\},\ V = \begin{vmatrix} p_0^{-m-1} & p_1^{-m-1} & \ldots & p_{n-1}^{-m-1} \\ \vdots & \vdots & \ldots & \vdots \\ p_0^{-1} & p_1^{-1} & \ldots & p_{n-1}^{-1} \\ p_0^{0} & p_1^{0} & \ldots & p_{n-1}^{0} \end{vmatrix},\ S = y_0, y_1, \ldots, y_{n-1}.$$

$G(P, V, o)$ la garde de la transition et $S = Op(P, V)$ avec Op la fonction des paramètres de sortie qui se sert des entrées et de l'état. Avec n en fonction du nombre de paramètres des entrées et sorties de l'état correspondant au nœud et m un paramètre de l'algorithme.

Ces données seront mises à jour automatiquement à chaque nouvelle observation. Nous utilisons U pour représenter l'ensemble clôt par préfixe de traces de l'EFSM, c'est-à-dire les états, mais aussi l'EFSM $(U, \varepsilon, I, O, T_U)$.

Par exemple, prenons un nœud ayant deux transitions par "a", une avec en sortie "x" et une autre avec "y". Pour simplifier, prenons $m = 1$. Si nous sommes dans une authentification, nous pourrions avoir les éléments suivants associés à U_s :

$$P = \{root, toor\},\ V = \begin{vmatrix} admin & admin & 401 \end{vmatrix},\ o = \text{“}x\text{”}.$$
$$P = \{root, root\},\ V = \begin{vmatrix} root & toor & 401 \end{vmatrix},\ o = \text{“}x\text{”}.$$

$$P = \{root, x!jklfiz@\},\ V = \begin{vmatrix} root & root & 401 \end{vmatrix},\ o = \text{“}y\text{”}.$$
$$P = \{root, x!jklfiz@\},\ V = \begin{vmatrix} root & x!jklfiz@ & 200 \end{vmatrix},\ o = \text{“}y\text{”}.$$

De ces éléments, nous pouvons en déduire que pour avoir y en sortie, nous devons envoyer (root, x !jklfiz@) comme paramètres de “a”. De la même façon, S, les valeurs de paramètres de sorties, est décidées par les éléments associés au nœud. Chaque valeur différente de chaque paramètre sera décidée suivant les valeurs des P et V.

5.4.2 Procédure d'inférence générale

La procédure d'inférence 10 est réduite à la construction du Z-Quotient initial puisque la recherche de contre-exemples paramétrés sur le système serait trop coûteuse en temps et requêtes. En effet, en plus d'essayer des séquences d'entrées de façon aléatoire, il faut aussi tester les différentes combinaisons de valeurs pour chaque paramètre. Il peut arriver aussi que certaines valeurs critiques ne soient pas spécifiées par le testeur, comme des informations de connexion, rendant la recherche de contre-exemple difficilement réalisable en pratique.

Néanmoins, il est possible d'adapter l'algorithme pour gérer les contre-exemples comme c'est le cas pour l'algorithme SPaCioS présentée dans la section 3.1. Même si certains contre-exemples seront difficilement identifiables, il est toujours possible de trouver des réponses ou comportements différents.

En ce qui concerne le traitement de ces contre-exemples, nous pouvons adapter la méthode de gestion des contre-exemples de l'algorithme Z-Quotient à notre arbre d'observation étendu. Cette méthode consiste à ajouter le contre-exemple à l'arbre puis d'appeler la procédure Rendre_Cohérent_Etendu pour localiser et traiter la ou les contradictions.

La procédure prend en paramètre E, le système en tant qu'EFSM, $I = \emptyset$, l'ensemble vide des entrées, $Z = \emptyset$, l'ensemble vide des séquences discriminantes, P un ensemble de couples *nom*/*valeur* qui représente le pool de paramètres et U un ensemble fini d'URL qui représentent les points d'entrées de l'application. L'algorithme retourne I' et Z' qui sont les ensembles I et Z mis à jour ainsi que K, le (Z', I')-quotient initial étendu correspondant à l'EFSM E. Les méthodes d'inférence des gardes et des fonctions de sorties sont décrites dans la section 5.5.

La fonction "Explorer_URL" récupère la page Web correspondant aux URLs fournit dans U. Une étape d'extraction des entrées, comme celle de l'algorithme 6 de la section 4.3 est ensuite effectuée pour retourner l'ensemble I. L'inférence des gardes et des fonctions de sorties sont décrites dans la section 5.5 grâce à une version modifiée de l'algorithme ID3 appliquée sur les données de U. Une fois inférées, les gardes et fonctions de sortie sont ajoutées à K pour obtenir un EFSM.

Algorithme 10 : Infère$(E, I, Z, Urls, P)$

Result : I, Z et le Z-quotient étendu correspondant

```
1 I = Explorer_URL(Urls);
2 U = Construire_Quotient_Etendu(E, I, Z, {ε}, P);
3 K = (Q, q0, I, O, hK);
4 Rendre_Cohérent_Etendu(E, I, Z, U, K, P);
5 Inférer_Gardes(U, K);
6 Inférer_Fonctions_Sorties(U, K);
7 return K, Z, I;
```

Comme nous partons de $I = Z = \emptyset$, la procédure *Construire_Quotient_Etendu* définie dans l'algorithme 13 va renvoyer un arbre à un seul nœud. C'est pourquoi, nous explorons en premier les points d'entrées de l'application pour mettre à jour I.

Soit un arbre d'observation étendu U, et un EFSM inconnu E avec un alphabet d'entrée paramétrée X, la procédure *Étendre_nœud(E, u, Σ)* permet de découvrir l'EFSM E depuis un état atteint par la séquence d'entrées paramétrées u en appliquant les séquences d'entrées de l'ensemble Σ des *requêtes* et retourne un arbre d'observation étendu. La différence avec la procédure pour les Mealy est que nous avons des paramètres que nous devons prendre en compte pour l'exploration.

5.4.3 Stratégie d'utilisation des paramètres

Pour chaque entrée $x \in \Sigma$, nous avons un ensemble de paramètres correspondant. L'algorithme peut puiser dans le pool de valeurs P pour récupérer les valeurs possibles pour chaque paramètre. Par exemple, pour un paramètre *password* nous pourrions trouver dans P les valeurs *toto* ou *root*. Chaque paramètre obtient donc un ensemble de valeurs possibles. Dans le cas où aucune valeur n'est fournie, l'algorithme génère une chaîne de caractères aléatoire à l'aide de lettres et chiffres et d'une longueur aléatoire également sauf si le paramètre provient d'un formulaire où la taille min ou max sont précisées.

La stratégie pour utiliser ces valeurs est d'utiliser toutes les combinaisons possibles de valeurs. Cette stratégie est appliquée uniquement la première fois qu'une entrée paramétrée est utilisée. Elle vise à classer ces combinaisons en fonctions des pages en sorties pour ensuite n'utiliser qu'un élément de cette classe. Par exemple, un formulaire d'authentification demande un nom d'utilisateur et un mot de passe. Plusieurs mots de passe et noms d'utilisateur peuvent être contenus dans P. Toutes les combinaisons sont testées pour obtenir à la fin deux classes, les crédentiels valides et les non valides. Dans la suite de l'algorithme, uniquement deux combinaisons

(une valide et une invalide) seront testées sur le système.

Algorithme 11 : Étendre_Nœud_Etendu (E, u, Σ, P) où $u \in U$, $\Sigma \subseteq X^*$

```
1  while ∃ a1a2...ak ∈ Σ / a1a2...ak n'appartient pas à v ↓I, ∀ v ∈ Tr(u) do
2      foreach ai do
3          if ai est utilisé pour la première fois then
4              Définir_Classe(E, u, ai, P);
5          end
6          foreach classe ci de paramètre associée à ai do
7              remettre E à son état initial;
8              appliquer u ↓I à E;
9              Sélectionner xi une combinaison associée à ci;
10             appliquer xi sur E et obtenir di en sortie;
11             ajouter la trace à U en créant le nœud v;
12         end
13     end
14 end
```

La procédure définie dans l'algorithme 12 permet de classer les combinaisons de paramètres selon le symbole de sortie obtenu. En utilisant toutes les combinaisons, plusieurs seront équivalentes et conduiront au même symbole de sortie. Dans le but d'éviter d'avoir à tester toutes ces combinaisons à chaque fois, elles sont classées selon leur résultat et seule une combinaison par classe sera utilisée ultérieurement.

Algorithme 12 : Définir_Classe (E, u, x, P) où $u \in U$, $x \in X$

```
1  foreach combinaison xi de paramètres pour x do
2      remettre E à son état initial;
3      appliquer u ↓I à E;
4      appliquer xi sur E et obtenir yi en sortie;
5      associer xi à la classe nommée yi;
6  end
```

5.4.4 Arbre d'observations étendu

Soit un arbre d'observation étendu U, et un EFSM E inconnu sur l'ensemble d'entrée X et de sortie O. La procédure *Construire_Quotient_Etendu(A, I, Z, U, P)*, où $I \in X$, $Z \in I^*$, construit l'EFSM $K = (Q, q_0, I, O, h_K)$ qui est un (Z, I)-quotient étendu de E, et retourne un arbre d'observation étendu. Cette procédure est similaire à celle pour les Mealy, seule la fonction d'extension de nœud est remplacée par la procédure 11.

Algorithme 13 : Construire_Quotient_Etendu (A, I, Z, U, P)

Result : Un arbre d'observation U étiqueté et l' EFSM K comme étant le (Z, I)-quotient étendu de l'EFSM A

foreach *état u de U parcouru durant le BFS tel que u n'a pas de prédécesseur étiqueté* **do**
  Étendre_nœud(A, u, Z, P);
  if *u est Z-équivalent à un nœud déjà traversé w de U* **then**
    étiqueter u avec w, c'est-à-dire $label(u) = w$;
  end
  else
    ajouter u dans Q;
    Étendre_nœud(A, u, I, P);
  end
end
foreach *transition (u, ab, v), tel que ni l'état u ni ses prédécesseurs soient étiquetés* **do**
  if *v n'est pas étiqueté* **then**
    ajouter la transition (u, ab, v) à K;
  end
  else
    ajouter la transition (u, ab, w) à K, où $w = label(v)$;
  end
end
foreach *nœud u' étiqueté avec u* **do**
  étiqueter les successeurs de u' tel que ;
  foreach *transition (u', ab, v)* **do**
    if *il existe une transition (u, ab, w) dans K* **then**
      étiqueter v avec w;
    end
  end
end

Étant donné un arbre d'observation étendu U et un (Z, I)-quotient étendu $K = (Q, q_0, I, O, h_K)$ obtenu à partir de U, un nœud de l'arbre U étiqueté avec $q \in Q$ possède une étiquette *contradictoire*, s'il a une trace qui est incompatible avec l'état q du (Z, I)-quotient étendu K ; cette trace est appelée une *trace contradictoire* pour q.

Une façon de résoudre ces contradictions est d'étendre l'ensemble Z avec les projections des entrées de la trace contradictoire, puis de répéter la procédure

Construire_Quotient_Etendu(A, I, Z, U, K, P) jusqu'à ce que l'arbre d'observation ne contienne plus d'étiquette contradictoire. Cette procédure est similaire à celle pour les Mealy, *Rendre_Cohérent_Etendu* mais elle prend en compte les nonces dans une seconde partie.

Algorithme 14 : Rendre_Cohérent_Etendu(A, I, Z, U, K, P)

Result : Un arbre d'observation étendu correctement étiqueté et son quotient

est_incohérent = Il existe une séquence d'entrée S_i présente dans U et un nœud w étiqueté u tel que $w\downarrow_{S_i} \neq u\downarrow_{S_i}$;
nouveau_nonce = Il existe un nœud état $u \in U$ tel que les données de P_i indiquent un nouveau nonce;
while *est_incohérent ou nouveau_nonce* **do**
    if *est_incohérent* **then**
        $Z' = Z \cup \{w\downarrow_I\}$ et $I' = I \cup set_of_input(w)$;
        Construire_Quotient_Etendu(A, I', Z', U, P);
        $Z = Z'$ et $I = I'$;
    end
    if *nouveau_nonce* **then**
        remettre A à son état initial;
        appliquer $u\downarrow_I$ à E;
        n = valeur du nonce;
        foreach $i \in I$ *et* P_i *les paramètres* i **do**
            if $type(P_i) = type(n)$ **then**
                remettre A à son état initial;
                appliquer $u\downarrow_I$ à E;
                n = valeur du nonce;
                appliquer i à A avec comme valeur n;
                ajouter la trace à U;
            end
        end
    end
    est_incohérent = Il existe une séquence d'entrée S_i présente dans U et un nœud w étiqueté u tel que $w\downarrow_{S_i} \neq u\downarrow_{S_i}$;
    nouveau_nonce = Il existe un nœud état $u \in U$ tel que ses données contiennent un nouveau nonce;
end

Pendant l'exploration, si une nouvelle page est découverte, nous avons atteint un nouvel état potentiel. Pour en être sûr, nous allons étendre les nouveaux nœuds crées avec I (comme si c'était de nouveaux états) et relancer ensuite la procédure

de construction du quotient en ajoutant la racine de l'arbre d'observations étendu au parcours en largeur.

5.4.5 Détection et réutilisation des nonces

Les valeurs non déterministes (nonces) sont détectées sur les données associées à chaque nœud. Soit un nœud $u \in U$. Un paramètre de sortie est un nonce s'il existe au moins deux éléments des données de u qui contiennent les mêmes valeurs en entrées, mêmes m dernières valeurs de paramètres, mais différentes valeurs pour le paramètre en sortie. Dans ce cas, le paramètre est uniquement marqué comme étant un *ndv* et sa valeur est utilisée dans les traces partant de u pour mettre à jour l'arbre. Si un nouveau comportement est identifié, la procédure *Rendre_Cohérent_Etendu* se chargera de mettre à jour l'arbre d'observations étendu.

Pour limiter le nombre de requêtes produites par la réutilisation de la valeur d'un nonce, nous utilisons une inférence de type basique sur la valeur du nonce et des autres paramètres pour limiter les paramètres qui réutiliseront cette valeur. Seul un paramètre du même type pourra réutiliser le nonce. Plus l'inférence de type est précise, plus la méthode est efficace.

Nous avons défini quelques heuristiques de détection de nonce :
- les nombres : des valeurs numériques limitées dans $[1, 1000]$. Ces nombres correspondent à des identificateurs (pages.php ?id=40) ;
- les identificateurs numériques $[1001, +\infty]$: des valeurs numériques assez grandes pour représenter tout type d'objets. Ces identificateurs sont notamment utilisés par Facebook ;
- les chaînes de caractères ;
- les valeurs encodées : ce sont des chaînes de caractères particulières et suivant un format spécial (hash, base64, ...).

5.5 Inférence des gardes et fonctions de sorties

La méthode d'inférence décrite précédemment permet d'obtenir un modèle sous forme d'EFSM. Mais il peut exister pour un même état S plusieurs transitions ayant le même symbole d'entrée x mais différents symboles de sorties et menant à des états différents. L'information qu'il manque se trouve en fait dans les paramètres du symbole d'entrée. L'exemple courant est la requête d'authentification à un site, la même requête peut conduire à deux résultats différents suivant la valeur des crédentiels fournis par l'utilisateur.

L'étape d'inférence des gardes permet de transformer l'information des données en gardes pour l'EFSM. Nous allons nous servir des valeurs des paramètres d'entrées et de sorties qui sont associées à ces états pour inférer des gardes qui seront mutuellement exclusives. Elles pourront aussi servir à inférer les fonctions de sortie.

5.5.1 Les données brutes

Dans la conjecture brute, les gardes et fonctions sont représentées sous forme de relations entre les entrées, l'état et les sorties. Ces informations sont de la forme :

$$x_0, x_1, \ldots, x_{n-1}, \begin{vmatrix} p_0^{-m-1} & p_1^{-m-1} & \cdots & p_{n-1}^{-m-1} \\ \vdots & \vdots & \cdots & \vdots \\ p_0^{-1} & p_1^{-1} & \cdots & p_{n-1}^{-1} \\ p_0^{0} & p_1^{0} & \cdots & p_{n-1}^{0} \end{vmatrix}, y_0, y_1, \ldots, y_{n-1}.$$

Les valeurs des paramètres des symboles d'entrée et de sortie sont respectivement les x_i et y_i. L'état du système est ici représenté par une matrice $n * m$ avec n le nombre de paramètres qui peut varier suivant le nombre d'entrée et de sortie découvertes dans l'état courant et m le nombre des dernières valeurs de ce paramètre (m étant un paramètre de l'algorithme).

5.5.2 Une version modifiée d'ID3

Nous allons utiliser une version modifiée de l'algorithme de création d'arbres de décision ID3 [Quinlan 1986]) sur ces valeurs pour trouver les différentes relations entre les entrées, valeurs des paramètres et symboles de sorties. ID3 fait partie des algorithmes de classification supervisés, il se sert d'instances déjà classées pour générer un arbre de décision qui servira à classer de nouvelles instances. À chaque nœud de l'arbre, nous avons différents choix selon les valeurs d'un des attributs. Quand une feuille de l'arbre est atteinte, nous avons la classe à laquelle appartient la nouvelle instance.

La plupart des algorithmes de ce type ont été faits pour travailler sur plusieurs millions d'instances et leur but est de trouver le meilleur arbre avec le moins d'erreurs de classification. Des améliorations d'ID3 ont été créées dans ce sens comme C4.5 [Quinlan 1993]. Évidemment, il est rare d'avoir une classification entièrement correcte. Alors que dans notre cas, nous avons très peu d'éléments, de l'ordre de la centaine, mais surtout nous savons qu'il existe un arbre qui doit classer parfaitement les données. L'application étant déterministe, elle se sert des entrées et de l'état pour décider quelles transitions exécuter. Il peut y avoir des erreurs de classification dans le cas où une relation de plus haut niveau existe entre certains

attributs, par exemple, $p_1 = sha1(v_1^0)$ ($sha1$ [NIST 2002] étant un algorithme de hachage produisant une valeur sur 160 bits correspondant à son paramètre).

Le principe d'ID3 est de sélectionner l'attribut qui permettra de classer le maximum d'éléments. Pour cela, il parcourt chaque attribut et calcule un gain d'information basé sur la différence d'entropie avant et après la sélection de l'attribut. Ensuite, il choisit l'attribut ayant le maximum de gain et découpe ses valeurs en sous-ensemble qui correspondront chacun à une ou plusieurs classes. Pour celle ayant encore plusieurs classes, le travail est effectué encore une fois sur les attributs restants.

Définition 4 *Soit $H(S)$ l'entropie d'un ensemble de données. Cette valeur mesure la quantité d'information présente dans les données.*
$H(S) = \sum_{x \in X} P(x) log_2 P(x)$ avec S l'ensemble des données, X l'ensemble des classes de données et $P(x)$ la proportion des éléments de S appartiennent à la classe x. On remarque que si $H(S) = 0$ alors tous les éléments de S font partie de la même classe.

Définition 5 *Soit $IG(A, S)$ le gain d'information après avoir utilisé l'attribut A pour classer les éléments de S. Plus ce gain est élevé, plus nous allons classer d'éléments de S en utilisant l'attribut A. Ce gain d'information est calculé en faisant la différence d'entropie avant et après l'utilisation de l'attribut A.*
$IG(A, S) = H(S) - (\sum_{t \in T} P(t)H(t)) * C(A)$ avec $H(S)$ l'entropie de l'ensemble des données de S, T le sous-ensemble de données crée en découpant les données selon l'attribut A, $P(t)$ la proportion des éléments de S appartement à la classe t, $H(t)$ l'entropie du sous-ensemble t et $C(A)$ un coefficient selon l'âge de la valeur.

La modification du calcul portera sur la meilleure donnée à utiliser puisque chaque paramètre contient plusieurs valeurs. Il n'est pas rare d'avoir la même valeur de paramètres pour les m dernières valeurs, ou des valeurs moins récentes qui ont un gain équivalent a des valeurs récentes. Comme nous avons maintenant deux critères, la valeur et son âge, il nous faut un moyen de sélectionner celui qui a priorité. Pour cela nous allons pondérer les variables, comme dans [Guan 2011], avec des poids plus faibles plus la valeur est ancienne pour privilégier les valeurs récentes qui sont supposées avoir le plus d'impact sur le classement comme décrit dans l'algorithme 15.

Algorithme 15 : ID3_Pondéré($S(A), T$) où S est un ensemble de données avec A attributs et T l'arbre de décision correspondant.

```
1  Créer le nœud racine;
2  Soit X l'ensemble des classes;
3  if ∃ x ∈ X tel que tout s ∈ S, classe(s) = x then
4  |  Renvoyer le nœud racine avec la classe x;
5  end
6  else
7  |  Soit a ∈ A tel que IG(a, S) = max(IG(c, S)∀c ∈ A);
8  |  A = A − {a};
9  |  foreach valeur v de a do
10 |  |  créer une nouvelle branche U_v;
11 |  |  soit S_v les éléments ayant comme valeur v;
12 |  |  U_v− > fils = ID3_Pondéré(S(A) − S_v, T);
13 |  end
14 end
```

5.5.2.1 Inférence des gardes

Pour chaque état ayant au moins deux transitions différentes et ayant des symboles de sorties différents, toutes les informations sont regroupées et seront traitées par fouille de données. Ces informations contiennent les paramètres d'entrées, l'état du système et le symbole et ses paramètres de sortie. Le symbole d'entrée étant le même, il n'est pas nécessaire ici. Par contre, les symboles de sortie sont considérés comme les classes de l'instance. Nous avons donc nos données classées et prêtes à être traitées. En ce qui concerne l'état du système, chaque variable est utilisée en tant qu'attribut des données.

Pour chaque paramètre de sortie $y_i \cup Y$, nous définissons les instances à traiter de cette façon :

$$[x_0, x_1, \ldots, x_n], [p_0^0, p_1^0, \ldots, p_n^0], \ldots, [p_0^{-m}, p_1^{-m}, \ldots, p_n^{-m}], [D_{y_i}].$$

Les différentes valeurs pour chaque paramètre sont considérées comme attributs des données. Ces attributs seront ensuite pondérés suivant m. Nous avons donc nos instances constituées des valeurs des paramètres de l'entrée et des variables pour les attributs, et de la valeur d'un paramètre de sortie comme classe. Ci-dessous, un exemple de données classées, certaines instances correspondent au symbole *list* et d'autres au symbole *home*.

— 101, larry, submit, 0, test, *list* ;
— 110, john, submit, 1, test, *list* ;

— 101, john, submit, 1, test, *home* ;
— 110, larry, submit, 1, test, *home*.

Il se peut qu'il y ait un attribut constant qui fausse le résultat de l'arbre. Suivant le nombre d'instances dans les classes, l'algorithme peut considérer que cet attribut a un gain suffisant pour classer toutes les instances. C'est pourquoi un prétraitement des données sera effectué pour enlever tous les attributs (paramètre du symbole d'entrées et les variables d'états) qui seront constants.

Concernant les problèmes liés à des relations de haut niveau liées à la sécurité, nous pouvons ajouter une de ces relations avant le traitement par ID3. Ces relations peuvent être définies manuellement, par exemple, égalité, relations de hachage. Pour chaque relation de haut niveau, nous pouvons spécifier une valeur de support minimum pour laquelle un nouvel attribut sera créé. Un prétraitement est donc effectué sur les données pour trouver ces relations. Par exemple, si nous trouvons que le fait d'avoir deux attributs égaux à chaque fois permet de classer toutes les données, un nouvel attribut nommé *attribut1 = attribut2* sera créée et chaque donnée sera complétée par la valeur booléenne correspondante. L'algorithme ID3 détectera que ce nouvel attribut possède le plus grand gain d'information et produira un arbre de décision optimal.

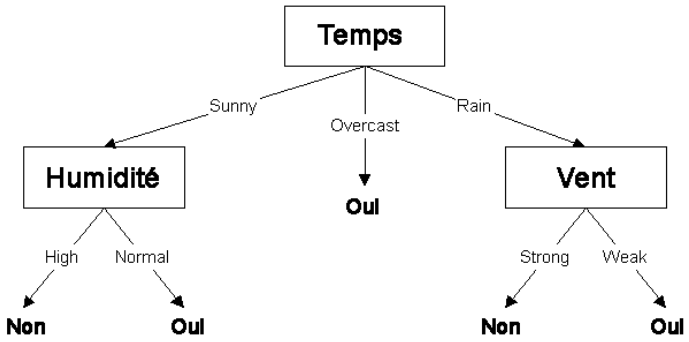

FIGURE 5.2 – Exemple d'arbre de décision

Une fois l'arbre de décision obtenu, nous pouvons en extraire les règles associant les données aux symboles de sortie dans une expression sous forme normale disjonctive. Il suffit de parcourir l'arbre en profondeur pour chaque classe, puis de fusionner chaque règle.

Par exemple, pour l'arbre de décision de la figure 5.2 qui représente s'il faut prendre son parapluie selon le temps, le vent et l'humidité, on obtient ces prédicats :
— (Temps = Sunny $\wedge$ Humidité = Normal) $\vee$ (Temps = Couvert) $\vee$ (Temps = Rain $\wedge$ Vent = Weak) $\Rightarrow$ Oui ;

— (Temps = Sunny ∧ Humidité = High) ∨ (Temps = Rain ∧ Vent = Strong) ⇒ Non ;

5.5.2.2 Inférence des fonctions de sortie

Les fonctions de sortie permettent de définir les valeurs des paramètres de sortie. Nous pouvons utiliser un arbre de décision J48 (implantation libre du C4.5 de [Quinlan 1993]). Le principe est le même que pour les gardes, les valeurs de chaque paramètre sont tour à tour considérées comme des classes.

En analysant des applications Web ainsi que leurs mécanismes de sécurité, il s'est avéré qu'il n'existe pratiquement aucun cas où la valeur d'un paramètre de sortie est une fonction linéaire des données. Un algorithme avec régression comme le M5P [Wang 1997] ne serait pas efficace. Les paramètres de type nombre sont le plus souvent le code de statut de la réponse HTTP ou des champs de données associés à un paramètre, par exemple, champs code postal d'un profil. De plus, les résultats de l'algorithme de régression avec les données produiront le plus souvent des résultats qui ne sont pas utilisables dans le modèle.

Dans notre approche, comme les paramètres de type nombre sont au final comparables aux paramètres des autres types, nous les considérons aussi comme des classes ce qui permet de retrouver des relations du genre :

$$ID = 42 \rightarrow \begin{cases} nom & = & jack \\ prenom & = & oneill \\ age & = & 64 \end{cases}, ID = 43 \rightarrow \begin{cases} nom & = & daniel \\ prenom & = & jackson \\ age & = & 43 \end{cases}$$

De cette façon, nous pouvons, pendant la détection de vulnérabilité, observer une requête avec comme valeur en entrée *Jackson* et trouver qu'il faut envoyer une autre requête avec le paramètre d'entrée *43* pour voir la réflexion. À part en cas d'égalité ou d'autres relations de haut niveau, l'algorithme J48 trouvera la plupart du temps des relations du type *paramètre* → *ensemble de valeurs*.

5.6 Expérimentations

Nous avons appliqué notre algorithme sur une des leçons de l'application WebGoat qui représente un site Web de gestion de ressources humaines. Cette application est décrite en détail dans le chapitre 4 lors de la création manuelle d'adapta-

teurs.

Configuration de départ :
— $I = \emptyset$;
— $Z = \emptyset$;
— $URL =$ "http ://localhost :8080/WebGoat/" ;
— $P =$ ["manager" : "110", "employee_id" : "111, 110", "password" : "john, larry", "search_name" : "Bruce"].

1. Nous créons un nœud racine pour l'arbre d'observation étendu ;

FIGURE 5.3 – nœud racine de l'arbre d'observation étendu

2. La première étape consiste à explorer les points d'entrée. La seule adresse contenue dans URL est la page de connexion représentée en figure 5.4.

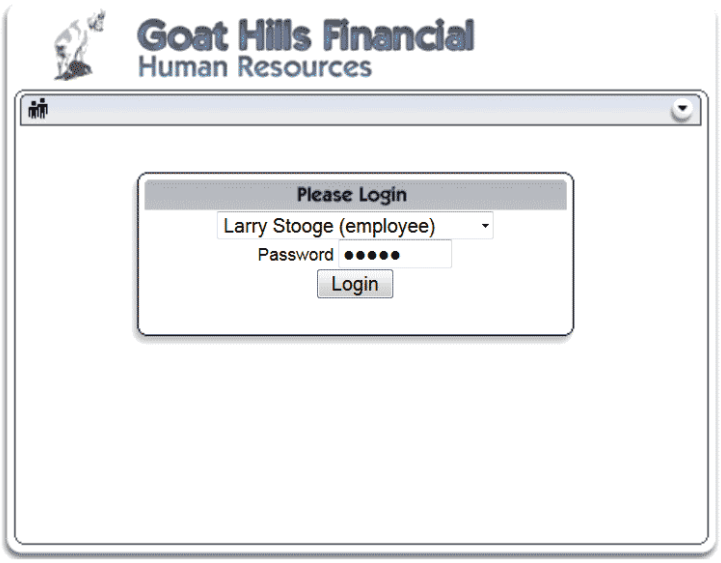

FIGURE 5.4 – Page de connexion de l'application

La seule entrée obtenue est celle du formulaire de connexion. Une entrée à deux paramètres, *employee_id* pour l'utilisateur et *password* son mot de passe. Nous mettons à jour $I = [LOGIN]$. Dans cet exemple, nous appelons cette entrée $LOGIN$ mais un nom générique est utilisé par l'algorithme. Du

coté des sorties, nous avons une sortie en plus, la page de connexion. L'arbre de page correspondant est ajouté à O.

3. La procédure *Construire_Quotient_Etendu* est ensuite exécutée. La procédure *Étendre_nœud_étendu* avec Z n'a pas d'effet puisque $Z = \emptyset$. Le seul nœud existant est détecté et marqué comme étant un nouvel état et il est donc étendu avec les éléments de $I = [LOGIN]$. Comme c'est la première fois que l'entrée $LOGIN$ sera utilisée, nous explorons $LOGIN$ avec toutes les combinaisons de valeurs de paramètres :
   — ("111", "john") → Liste de employées (Nouvelle page) ;
   — ("111", "larry") → Page de connexion ;
   — ("110", "john") → Page de connexion ;
   — ("110", "larry") → Page de connexion.

   Pendant cette exploration, nous avons découvert une nouvelle sortie correspondant à la liste des employées (figure 5.5) qui a été ajoutée à O. Nous avons aussi obtenu deux classes de valeurs :
   — ("111", "john") → Crédentiels valides ;
   — ("111", "larry"), ("110", "john"), ("110", "larry") → Crédentiels invalides.

   Les prochaines explorations de $LOGIN$ se feront avec un élément de chaque classe uniquement. Nous étendons l'arbre avec l'entrée $LOGIN$ avec les paramètres ("111", "john") et ("111", "larry").

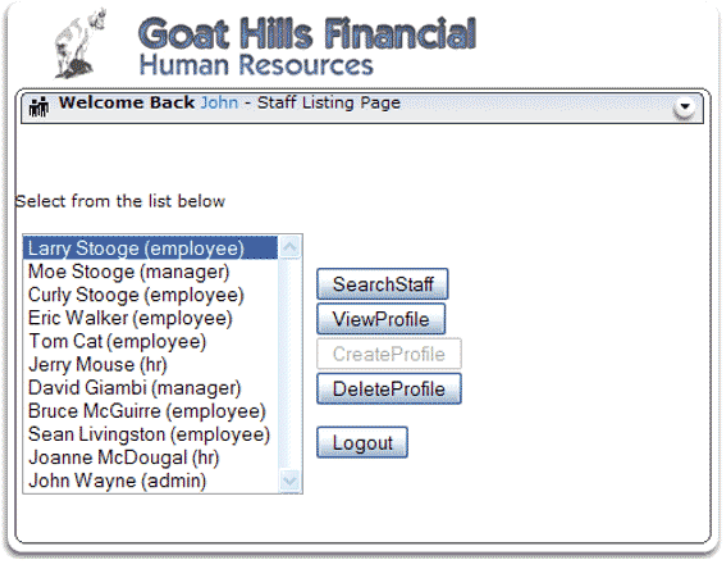

FIGURE 5.5 – Liste des employées

4. Nous obtenons l'arbre décrit dans la figure 5.6 où les deux nœuds fils sont

étiquetés avec $U0$. Nous avons aussi obtenu une nouvelle page donc de nouvelles entrées à ajouter à I. I = [LOGIN, VIEW, CREATE, DELETE, SEARCH, LOGOUT].

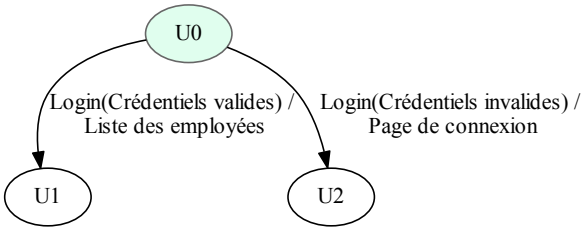

FIGURE 5.6 – Arbre d'observation étendu avec $I = [LOGIN]$

5. Comme l'exploration du symbole $LOGIN$ a entraîné la découverte d'une nouvelle page (ou symbole de sortie), nous avons atteint un nouvel état potentiel. Comme pour un nouvel état, les nouveaux nœuds sont étendus avec I et la racine de l'arbre est ajouté à la liste des nœuds à parcourir, pour reconstruire le quotient. Nous obtenons l'arbre d'observations décrit dans la figure 5.7.

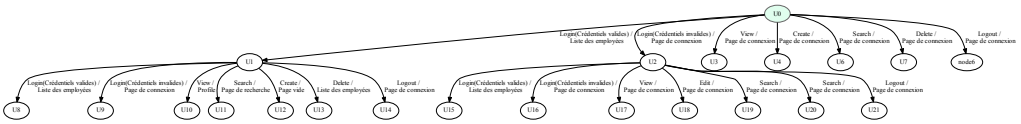

FIGURE 5.7 – Arbre d'observation étendu avec I = [LOGIN, VIEW, CREATE, DELETE, SEARCH, LOGOUT].

Cette fois-ci, chaque nœud a été exploré avec toutes les entrées disponibles. Les nœuds $U1$ et $U2$ sont toujours étiquetés comme étant équivalents au nœud (à l'état) $U0$.

6. La procédure *Rendre_Cohérent_Etendu* est exécutée. Elle trouve une trace contradictoire (à cet instant l'arbre d'observation en contient plusieurs) : $VIEW$. En effet, la séquence d'entrée composée uniquement du symbole $VIEW$ a un résultat différent dans $U0$ et $U1$ alors que $U1$ est étiqueté comme équivalent à $U0$. Cette trace est donc ajoutée à l'ensemble Z qui vaut maintenant $Z = [VIEW]$. L'ensemble I ne change pas puisqu'il contient déjà $VIEW$.
7. La procédure *Construire_Quotient_Etendu* est ensuite exécutée une dernière fois. Elle va permettre d'identifier $U1$ comme un nouvel état. Aucune trace contradictoire n'est trouvée ensuite. L'arbre d'observations étendu final se trouve en figure 5.8 et contient deux états.

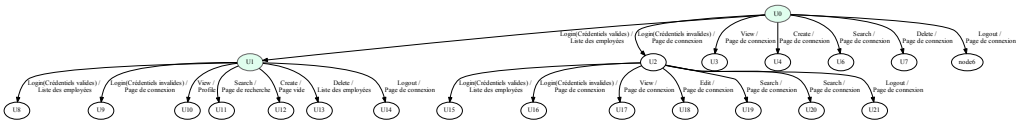

FIGURE 5.8 – Arbre d'observation étendu avec I = [LOGIN, VIEW, CREATE, DELETE, SEARCH, LOGOUT], Z = [VIEW].

8. Les fonctions de recherche et d'édition font aussi apparaître de nouvelles pages. Elles seront explorées comme ci-dessus, mais aucun nouvel état n'est trouvé.

Du côté des fonctions de sortie, nous trouvons les relations liant l'identificateur de profil, comme *111* pour l'utilisateur *John*, avec les différents champs de son profil affiché grâce à l'entrée $VIEW$. Par exemple :

$$ID = 111 \rightarrow \begin{cases} Firstname &= John \\ Lastname &= Wayne \\ Street &= 129\ Third\ St \end{cases} .$$

Ce type de relations sont utiles pour la détection de vulnérabilités de type XSS. Elles sont utilisées dans le chapitre 6. Dans la figure 5.9, nous avons une représentation simplifiée du modèle EFSM obtenu après inférence avec notamment la réflexion du paramètre *Street* qu'il fallait trouver dans cette leçon.

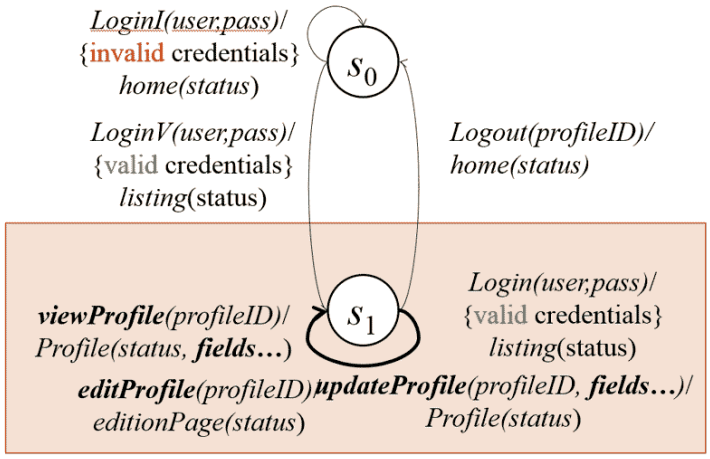

FIGURE 5.9 – Modèle EFSM inféré de la leçon WebGoat Stored XSS.

5.7 Limitations

Une des limitations de cette approche est due aux technologies utilisées par le site Web. Ces dernières années, l'émergence du JavaScript et de l'AJAX (Asynchronous JavaScript and XML) a permis de créer des applications plus dynamiques et réactives. Ces applications dites *riches* (RIA) ont changé la façon dont les pages sont utilisées. Les liens ne conduisent plus directement à une page, mais au déclenchement d'un événement, puis à l'exécution d'un script qui effectuera la requête. De plus, sans changer de page, l'application est capable de communiquer et de modifier la page avec les réponses d'autres services sur Internet. Pour notre approche, cela limite l'extraction automatique des entrées puisque les informations nécessaires ne sont plus dans le code HTML, mais dans le code JavaScript. Étant donné que l'on peut générer les requêtes d'une infinité de façons, il est impossible au collecteur de récupérer automatiquement les adresses des requêtes ainsi que leurs paramètres.

```
var webservice = "http://uri/services/";
$('#logout').click(function(){
    $.ajax({
      type: "GET",
      dataType: "json",
      url: webservice + "logout",
      success: function(data){
       alert(data);
      }
    });
});
```

Listing 5.1– Association d'une requête HTTP à un événement *click*

Depuis 2008, les méthodes de collection (crawling) d'applications riches se sont développées principalement en fuzzant les différents éléments de la page Web pour déclencher les événements et ainsi récupérer les différentes requêtes et réponses générées. Dans [Mesbah 2008], ils ont développé un outil nommé *Crawljax* qui permet de générer un miroir statique d'un site Web dynamique. L'outil génère divers événements comme un clic ou un passage de souris sur les différents éléments disponibles sur la page Web, puis les modifications sur le DOM sont enregistrées pour générer progressivement une machine à états qui représente les différents états de la page ainsi que les actions utilisées. À partir de cette machine, une copie statique du site est générée. Plus récemment [Benjamin 2011], le but est aussi de construire une machine à états contenant les différents états du DOM qui peuvent être atteints par les différents événements JavaScript, mais la méthode génère une stratégie de collection optimale, appelée *hypercube*, en fonction d'un modelé supposé de l'application. En utilisant ce modèle, cette méthode permet de sélectionner les meilleures actions à exécuter sur l'application pour construire la machine à état

correspondantes. Dernièrement dans [Choudhary 2013a], la méthode décrite dans [Benjamin 2011] est amélioré en remplaçant la stratégie *hypercube* qui s'est avéré assez complexe à utiliser en pratique par *menu*. Ce nouveau modèle considère certains événements comme étant indépendants de l'état dans lequel se trouve l'application. Le déclenchement de cette action aboutira toujours sur le même résultat, comme un menu.

Bien que nous construisons aussi une machine à états, ici elle représente les états du DOM alors que nous recherchons les états de l'application elle-même. Le but est aussi différent. La plupart des méthodes indiquées précédemment sont faites pour les moteurs de recherche où l'énumération du maximum de pages et de leur contenu est la plus importante. Dans notre contexte, nous n'avons besoin que des différentes actions ou services fournis par l'application.

Dans le cas des Web services (SOAP, SOAP-RPC), les entrées que nous recherchons sont souvent décrites dans un fichier WSDL (*Web Services Description Language*) où se trouve la localisation du service, les méthodes disponibles et le format des messages requis pour communiquer avec le service. SOAP et SOAP-RPC définissent un protocole complet pour communiquer avec les services, mais il existe aussi un autre type d'architecture : REST (Representational State Transfer) dans lequel l'interface est bien plus simple et libre.

Les services basés sur REST sont appelés RESTful et utilisent directement les URI pour accéder aux services. Par exemple *GET http ://example.com/res/17* permet de récupérer l'élément d'index 17. Les autres méthodes HTTP ont aussi un sens comme DEL pour supprimer, POST pour créer et PUT pour modifier. Cette architecture étant assez simple, il n'y a pas de fichier dédié pour la description des services. On peut trouver des fichiers WADL pour *Web Application Description Language* qui représentent l'équivalent REST des fichiers WSDL. Cependant, ce langage de description a été soumis au W3C en 2009 par Sun Microsystems, mais il n'est pas prévu qu'il devient un standard.

Pour notre approche d'inférence, nous pouvons utiliser les fichiers *WSDL* ou *WADL* pour récupérer la liste complète des interfaces. En ce qui concerne les services RESTful sans *WADL*, notre approche peut utiliser une liste d'URI de ressource en entrée.

Chapitre 6

Détection de vulnérabilités Web

Nous avons vu dans les chapitres précédents comment modéliser une application Web sous forme de machine à états finie étendue. Ce type de modèle a été choisi pour pouvoir représenter le maximum de fonctionnalités, d'un point de vue de la sécurité, tout en gardant un nombre d'états raisonnable pour que le modèle puisse être analysé assez rapidement. Dans ce chapitre, nous allons voir quels sont les types de vulnérabilités que l'ont peut détecter à partir de ce genre modèle, mais aussi comment les détecter et générer les cas de tests correspondant. Comme nous voulons garder notre approche la plus automatique possible, ces étapes seront elles aussi automatiques.

Sommaire

Les applications sont conçues pour fournir un certain service aux clients selon des spécifications. Il peut arriver qu'une défaillance entraîne un changement de comportement de l'application qui fournit désormais un service différent. Cette défaillance est produite par une ou plusieurs erreurs. Une erreur est un état atteint de l'application qui n'a pas été spécifié au préalable ou que l'application n'était pas censée atteindre. Une erreur est déclenchée par une faute, généralement un problème dans la conception de l'application.

Toutes les méthodes de détection de vulnérabilités présentées dans ce chapitre

se basent sur un modèle EFSM inféré, au préalable, par l'algorithme présenté dans le chapitre 5. Elles sont exécuté séparément, à la suite de l'inférence.

6.1 Détection d'actions non autorisées

Cette méthode utilise le modèle inféré d'une application ainsi que le graphe des entrées du site obtenu lors de l'inférence par le collecteur. Ce graphe ainsi que le modèle sont deux visions de l'application. En les comparant, nous pouvons exhiber certains chemins qui sont des vulnérabilités potentielles.

Les applications Web sont une façon de fournir à l'utilisateur un moyen d'utiliser certaines actions sur un système. Le plus souvent, cet ensemble d'actions est contrôlé au niveau de l'application et il dépend du rôle de l'utilisateur ou de son niveau d'accès. Il y a une étape d'authentification où l'utilisateur prouve qu'il est bien un certain utilisateur et ensuite l'application lui attribue son niveau d'accès avec les différentes actions qui sont liées. Typiquement, pour une application de gestion d'emails, un utilisateur s'authentifie sur un site avec son couple nom d'utilisateur/mot de passe et il est ensuite autorisé à consulter ses mails.

À partir d'un modèle EFSM inféré d'une application, nous avons développé une méthode de détection des actions qui sont permises par l'application, mais qui ne sont pas forcément voulues par les spécifications. Nous les appelons : *actions non autorisées*. Pour reprendre l'exemple précédent, il est possible que, connaissant la requête qui permet de consulter son courrier électronique, un utilisateur qui n'est pas authentifié puisse quand même consulter son courrier. Le plus souvent ce sont des erreurs liées à une mauvaise vérification de session ou d'authentification où seuls les paramètres de la requête sont vérifiés. Il peut s'agir aussi de mauvaises pratiques de développement dans lesquels seul l'élément HTML qui est responsable de la requête a été désactivé ou caché, mais la gestion de la requête correspondante reste toujours valide. Comme nous utilisons uniquement le flot de contrôle de l'application, il n'est pas possible de dire avec certitude que l'action non autorisée pourra être utilisée à des fins malicieuses. Mais cela indique une faute dans la conception du site qui peut entraîner une erreur.

6.1.1 Modèle inféré simplifié

Dans chaque état du modèle inféré, nous avons le résultat de toutes les actions trouvées sur le système. C'est sur ce principe que nous pouvons distinguer les différents états. Nous pouvons d'ailleurs simplifier la représentation du modèle EFSM défini dans le chapitre 3.1 pour ne garder que les composants d'entrée et

sortie. Nous obtenons donc une machine simplifiée M_s représentée par le tuple (S, X, Y, T) ou S est l'ensemble des états du système, X l'ensemble des entrées, Y l'ensemble de sorties et T un ensemble de transitions représentées par le tuple (s_i, x, y, s_f) dans lequel $s_i, s_f \in S$ sont, respectivement, les états initial et final de la transition, $x \in X$ l'entrée et $y \in Y$ la sortie correspondante.

Dans la figure 6.1, nous avons le modèle simplifié de l'application de gestion des ressources humaines utilisée par la plate-forme WebGoat. Dans ce modèle, nous n'avons que deux états. L'état S_0 correspond à un utilisateur qui n'est pas connecté sur le système tandis que dans l'état S_1, l'utilisateur s'est connecté et les actions deviennent valides (restent dans l'état S_1).

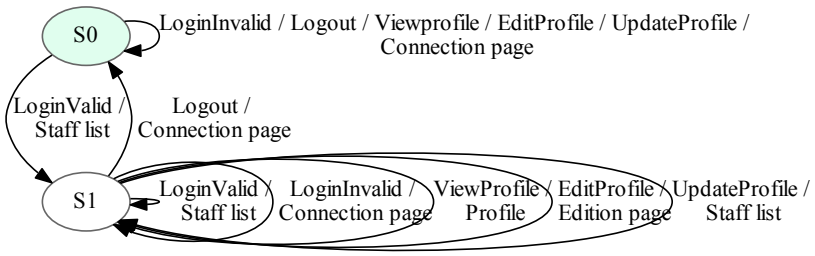

FIGURE 6.1 – Exemple du modèle pour l'application WebGoat

Nous avons donc exclu les informations sur les données telles que les prédicats et les fonctions de sortie.

6.1.2 Graphe des entrées

D'un autre côté, pendant la découverte progressive de l'application par l'adaptateur, les différentes séquences de requêtes qui mènent aux différentes entrées sont conservées. Nous pouvons ainsi reconstruire le graphe des entrées du site dans lequel chaque nœud est une page et chaque transition est une requête fournie par l'application. Ce graphe des entrées est en quelque sorte la carte du site puisqu'il indique où se trouvent les actions et comment on passe d'une page à une autre.

Nous obtenons le graphe W représenté par le tuple (P, X, T) dans lequel P est l'ensemble des pages de l'application, X l'ensemble des entrées et T l'ensemble des transitions qui permettent de passer d'une page à une autre. Contrairement à la machine M_s, chaque page $p \in P$ ne contient que les actions dans X qui sont fournies par la page.

Dans la figure 6.2, nous avons le graphe des entrées de l'application de gestion des ressources humaines utilisée par la plate-forme WebGoat.

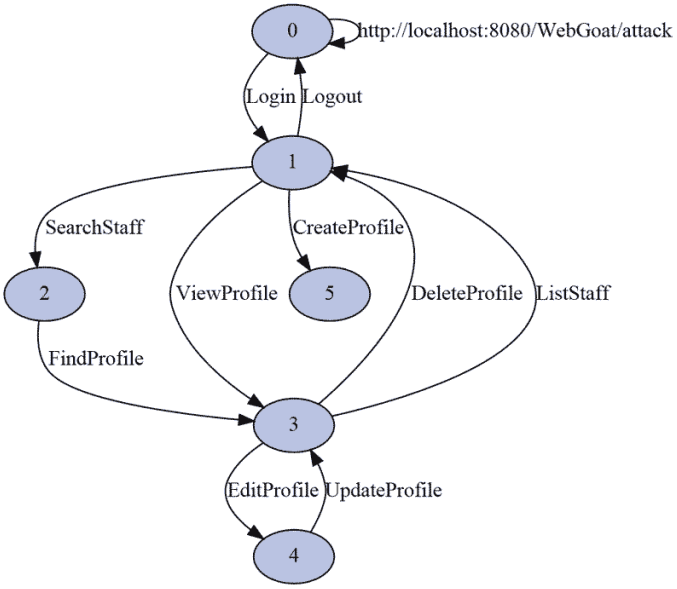

FIGURE 6.2 – Exemple du graphe des entrées pour l'application WebGoat

Nous pouvons déjà constater que la carte du site semble être cohérente avec le modèle simplifié de la figure 6.1. En effet, nous avons accès en premier lieu uniquement à l'action de *login*, qui une fois validée, nous permet d'utiliser toutes les autres actions de l'application telles que voir un profil ou le modifier.

6.1.3 Détection de vulnérabilités

Nous avons à notre disposition deux visions du site. Celle du point de vue navigateur (générée par l'adaptateur) et celle inférée par l'inférence de modèle. Pour détecter les actions non autorisées, nous allons comparer le modèle et le graphe des entrées pour en extraire des cas de test.

Voici les différentes étapes de cette méthode :

— La première étape consiste à colorer les pages du graphe d'entrée en fonction de l'état du modèle correspondant. Cela permet de faire correspondre les pages aux états pour pouvoir ensuite les comparer. Cette étape est similaire à celle de [Doupé 2012] à la différence que dans notre cas, nous savons déjà quels sont les états grâce au modèle inféré et nous cherchons uniquement à copier cette information sur une autre représentation. Sur la figure 6.3, le graphe d'entrée coloré en fonction du modèle. Notons P_{s_i} les pages correspondants à l'état S_i dans le modèle.

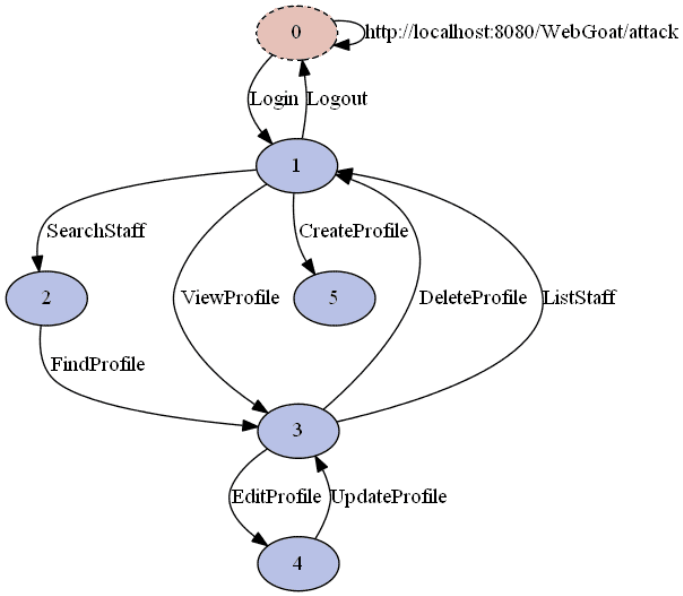

FIGURE 6.3 – Graphe des entrées coloré pour l'application WebGoat

On peut voir que la première page, quand l'utilisateur n'est pas connecté, a changé de couleur pour correspondre à l'état S_0 du modèle.

Seuls les symboles d'entrée sont représentés sur la figure 6.2 mais les requêtes complètes sont aussi sauvegardées. Pour la coloration du graphe, nous effectuons un parcours en profondeur du graphe des entrées. Une fois un nœud coloré, il est marqué pour éviter de le parcourir une nouvelle fois.
L'algorithme de coloration part de l'état S_0 du modèle inféré et du nœud initial du graphe. Pour chaque transition du graphe, une des requêtes associées est sélectionnée puis ses paramètres sont comparés aux prédicats dans le modèle pour identifier la transition correspondante dans le modèle. Le nœud est coloré suivant l'état d'arrivée dans le modèle et il est aussi marqué. Les requêtes conservées avec le graphe des entrées sont les mêmes que celles utilisées par l'inférence, nous trouverons toujours une transition correspondante dans le modèle. La coloration est terminée quand tous les nœuds ont été marqués.

— Ensuite, nous allons faire la liste des actions qui sont découvertes dans un certain état.

S_0	S_1
login	logout
	create
	create
	view
	edit
	update
	delete

Dans notre cas, seule l'action de *login* est découverte dans l'état S_0. Les autres le seront ensuite.

— Maintenant, nous pouvons comparer les actions fournies par le site et les actions qui sont possibles. Pour chaque état, nous allons vérifier pour les actions qui ne sont pas disponibles, si le résultat dans le modèle est différent d'un état à l'autre, ce qui signifierait que l'action a bien été traitée comme non autorisée par l'application, ou alors qu'il existe un état dans le modèle tels que l'action produit le même résultat alors que l'action n'est pas disponible sur le site.

Soit un modèle simplifié M et un graphe d'entrée G, nous recherchons les actions $x \in X$ telles que pour un état $s \in S$, $x \notin P_s$ et qu'il existe un autre état s' tels que le résultat de x dans s' soit égal à celui dans s.

De cette manière, nous recherchons une entrée qui n'est pas disponible sur le site, mais active au niveau de l'application. De plus, cette entrée a une sortie égale à la même entrée dans un état différent. Ce qui pourrait signifier que le résultat de cette action ne devrait être accessible que dans l'état s' et non s. C'est une vulnérabilité potentielle.

On peut résumer le fonctionnement de la méthode de détection par l'algorithme 16.

L'action peut très bien renvoyer le même résultat sans pour autant être une vulnérabilité exploitable à des fins malicieuses. Mais elle peut au moins signifier un problème de construction ou de méthodologie dans le développement de l'application. Les résultats sont à vérifier par le testeur pour décider si les informations accessibles sont exploitables pour un attaquant.

6.2 XSS (Cross-Site Scripting)

Durant la phase d'inférence définie dans la section 5.5.2.2, les réflexions des paramètres sont utilisées pour générer des fonctions des paramètres de sortie. Nous

Algorithme 16 : Detect_Non_Autorise(M, G) où M est un modèle simplifié et G le graphe d'entrée.

```
1 Coloration de G selon les états de M;
2 Construction de la table des pages depuis G;
3 foreach état s ∈ S de M do
4 |   foreach action x ∉ P_s do
5 |   |   if ∃ s' ≠ s telle que le résultat de x dans s est égale à celui de x dans
  |   |   s' then
6 |   |   |   Afficher x en tant qu'action potentielle non autorisée;
7 |   |   end
8 |   end
9 end
```

avons associé certaines valeurs en entrée à des valeurs en sortie. Ces associations sont présentes dans le modèle sous forme de fonctions de sortie. En parcourant les entrées utilisées dans les gardes et en comparant ces valeurs avec les paramètres des sorties, nous allons pouvoir détecter où se font les réflexions.

Grâce au modèle inféré, nous allons voir que détecter une vulnérabilité de type XSS persistant n'est pas plus compliqué que celle de type réfléchi puisque nous connaissons où se situent les réflexions ainsi que la requête qui est à l'origine de la valeur réfléchie.

Nous utilisons des comparaisons de chaînes pour détecter les réflexions, c'est pourquoi nous ne détectons que les réflexions totales de chaînes. Il peut arriver dans certains cas où les chaînes sont filtrées justement pour éviter de créer une vulnérabilité de type XSS. Dans la section suivante, nous discutons du problème des filtres et des réflexions partielles et nous verrons pourquoi, du point de vue du modèle, ces réflexions ne sont pas prises en compte, mais aussi pourquoi leur impact sur les résultats de la détection de vulnérabilité est limité.

La méthode est donc la même et elle peut se résumer avec la procédure définie dans l'algorithme 17. Un paramètre est une réflexion si une partie de sa valeur est égale à la valeur d'un paramètre d'une entrée.

Pour la recherche de chemin, nous pouvons nous aider du model-checker comme décrit dans la section 6.4.

6.2.1 Filtres et réflexions partielles

Il existe de nombreuses façons d'empêcher une application d'être vulnérable aux injections de type XSS. La plus simple étant de filtrer les entrées de l'application

Algorithme 17 : Detect_XSS(M) où M est un modèle inféré.

```
1 foreach état s ∈ S et chaque entrée x ∈ X de M do
2     if un des paramètres de sortie est une réflexion then
3         Identifier l'état s', l'entrée x' et son paramètre p' qui sera réfléchi;
4         Trouver un chemin partant de s_0 passant par s' puis s;
5         Générer un cas de test avec une valeur de p' différente pour
          l'ensemble des charges utiles XSS;
6     end
7 end
```

pour empêcher toute exécution de code. Ce filtre peut se faire en échappant certains caractères, mais aussi en les encodant d'une façon différente telle que les entités HTML, par exemple, <script> devient <script>. En fait, chaque langage de programmation d'application Web fournit plusieurs méthodes différentes pour encoder/filtrer de façon sécurisée les entrées de l'utilisateur. Avec ces méthodes, nous ne retrouvons plus exactement la même chaîne une fois la valeur réfléchie. Pour l'utilisateur, la valeur affichée est bien la même, mais au niveau du code source de la page, la valeur est différente.

De plus, les valeurs que nous envoyons à l'application peuvent être traitées de multiples façons différentes. On peut afficher directement cette valeur, ou ajouter un suffixe ou un préfixe, voire effectuer un traitement sur toute la chaîne comme un chiffrement ou un encodage. Toutes ces transformations permettent potentiellement d'injecter du code JavaScript en prévoyant comment sera la chaîne après transformation. Mais elle ne sera pas détectée comme réflexion par comparaison de chaînes.

A priori ces filtres ou transformations auront un impact sur le taux de détections des vulnérabilités XSS par l'outil. Mais durant l'inférence, nous utilisons uniquement des chaînes valides comme valeurs de paramètres. Elles sont valides du point de vue du format attendu par l'application justement pour ne pas déclencher de filtres qui pourraient transformer la chaîne après réflexion. Ces valeurs de paramètre proviennent soit du testeur qui sait comment utiliser l'application, soit de la génération automatique de valeurs qui se sert du type du paramètre et qui générera dans la plupart des cas des valeurs valides également.

Au final, nous avons le plus souvent des réflexions totales de chaînes qui seront détectées par l'algorithme. En ce qui concerne la détection de vulnérabilité, c'est-à-dire de réflexions de code JavaScript, nous utilisons une série de charges utiles XSS qui est spécialement conçue pour pouvoir contourner les différents filtres mis en place comme dans [Snake 2014]. Ces listes de charges utiles pourront être complétées par le testeur pour gérer les transformations effectuées par l'application. D'autres

approches [Duchene 2014] se basent sur des algorithmes génétiques et la grammaire HTML pour trouver des entrées qui contourneront les filtres.

6.2.2 Réduction de l'ensemble de code à injecter

Il existe plus d'une centaine de morceaux de code JavaScript à injecter pour détecter des éventuelles vulnérabilités XSS. Notre algorithme de détection d'XSS sépare la phase de détection de paramètre réfléchi de la phase de test de l'exploitabilité de cette réflexion. Pour chaque réflexion, il serait coûteux de tester l'ensemble de ses codes surtout que certains sont utilisables que dans un contexte précis.

Pour réduire le nombre de scripts à tester, nous devons considérer où se trouve la réflexion. Cela peut être dans un attribut particulier, comme *style* ou *value* pour les formulaires, dans un script existant en tant que contenu d'une variable ou un commentaire ou encore dans un élément de base comme une balise *div*. Pour les attributs, nous pouvons utiliser uniquement les codes utilisant les événements ou le chargement de ressources pour déclencher le script tandis que pour les éléments de base, nous pouvons nous limiter aux différents encodages des balises *script*.

6.3 CSRF (Cross-Site Request Forgery)

Ces vulnérabilités sont plus rares que les injections XSS et aussi moins puissantes puisqu'il y a quelques restrictions dans son fonctionnement :

— la victime doit être authentifiée sur le système (session valide) ;
— la victime doit elle même exécuter l'action fournie par l'attaquant ;
— la victime peut retrouver l'action qui a été faite à son insu.

Malgré ces restrictions, ce type de vulnérabilité s'est récemment montré très efficace contre différents routeurs. En effet, ces routeurs utilisent une application Web comme interface et il est possible de modifier les paramètres du routeur par une attaque de type CSRF. Cela a notamment été le cas récemment pour une majeure partie des 300000 routeurs qui ont été découverts avec une configuration DNS modifiée [Cymru 2014].

6.3.1 Détection de paramètres aléatoires dans les entrées

Pour cette méthode, nous allons nous servir de la détection des nonces effectuée lors de l'inférence. Ces nonces sont utilisés dans des mécanismes de sécurité prévenant les attaque de type CSRF. En recherchant les entrées qui n'utilisent pas de nonces, nous aurons des vulnérabilités potentielles.

Parmi les moyens de prévention de ce type d'attaque, la plus simple est de demander confirmation à l'utilisateur. Mais elle alourdit également l'application et son utilisation. De plus, dans le domaine de la sécurité, les confirmations sont souvent inefficaces puisque l'utilisateur aura tendance à confirmer automatiquement une action sans y prêter attention comme étudié dans [Krol 2012].

La méthode de prévention de CSRF la plus utilisée est de vérifier l'origine de la requête. Le seul fait de vérifier l'entête *HTTP_REFERRER* n'est pas suffisant puisqu'elle peut être forgée facilement. La vérification se fait par l'ajout d'une valeur aléatoire dans ses paramètres, cette valeur est alors appelée *jeton* de sécurité. C'est grâce à ce jeton et à sa particularité d'être aléatoire que nous allons pouvoir détecter les actions protégées du CSRF.

Lors de l'inférence, l'algorithme va déterminer les paramètres des différentes pages obtenues en sortie. Une fois que ce paramètre sera détecté comme aléatoire, il sera associé à l'entrée correspondante s'il s'agit bien d'un jeton présent dans un lien ou un formulaire. Dans le cas contraire, il sera simplement associé à la page comme les autres paramètres. Une fois le modèle inféré, nous allons parcourir les différentes entrées qui ont été trouvées et vérifier qu'elle possède une valeur aléatoire nécessaire à son bon fonctionnement. Un mauvais fonctionnement se traduit par l'observation d'un résultat différent. Nous pouvons nous concentrer principalement sur les requêtes qui permettent de changer d'état comme celles liées à l'authentification.

Le fonctionnement de l'algorithme de détection de CSRF est décrit dans l'algorithme 18.

Algorithme 18 : Detection_Chemin(M, S_i, S_f, p) où M est un modèle inféré, S_i l'état de départ (S_0 par défaut) et S_f l'état d'arrivée.

```
1 foreach entrée x ∈ X do
2 |   foreach paramètre p_i ∈ P_x, si p_i est marqué comme aléatoire do
3 |   |   Trouver un état s ∈ S tel qu'il existe une transition pour x;
4 |   |   Trouver un chemin passant commençant par S_0 et passant par s;
5 |   |   Afficher la trace correspondant au chemin;
6 |   end
7 end
```

Comme pour la détection de vulnérabilité XSS, nous pouvons nous aider de la recherche de chemin par model-checking décrite dans la section 6.4. Nous obtenons au final une liste d'actions qui, n'ayant pas de jeton aléatoire, ne sont pas protégées d'une attaque de type CSRF.

Framework	Format	Entropie (bits)
Django	[a-zA-Z0-9]32	190.53
Ruby on Rails	32 octets (base64)	256
Spring Security	Python UUID4 (16 octets)	122
OSASP PHP CSRF Guard	[a-z0-9]128	661.75
OWASP J2EE CSRF Guard	[A-Z0-9]32	165.44

TABLE 6.1 – Tableau des formats de jetons CSRF

6.3.1.1 Faux négatifs

L'utilisation de jeton aléatoire dans une requête pour son bon fonctionnement est une condition nécessaire pour une prévention de vulnérabilité de type CSRF, mais pas suffisante. Il peut arriver qu'un paramètre soit aussi aléatoire ou considéré comme aléatoire, mais sans qu'il n'intervienne dans la protection de vulnérabilité. Cela peut être le cas pour les dates par exemple.

Dans le but de réduire ces faux négatifs, un paramètre aléatoire sera considéré comme jeton CSRF uniquement s'il correspond à un format utilisé pour ce genre de valeurs. Dans le tableau 6.1, nous avons listé différents formats de jetons utilisés par des plates-formes de développement d'application Web. Il apparaît que le jeton doit toujours faire une certaine longueur pour avoir assez d'entropie et qu'il est souvent utilisé dans sa version alphanumérique.

Nous avons retenu comme format de jetons les chaînes qui sont matchées par l'expression régulière suivante :

```
^[a-zA-Z0-9+/]{16,}(={0,2})?$
```

Nous avons choisi cette expression pour pouvoir détecter les différents formats décrits dans le tableau 6.1. Globalement, cela correspond à une chaîne encodée en base 64 d'une longueur 16 minimum. Nous pouvons aussi vérifier l'entropie de cette chaîne. Par exemple, la chaîne composée de 28 caractères A est dans le bon format mais du fait de sa faible entropie, elle n'est pas un jeton.

6.4 Recherche de chemin

Que ce soit pour les vulnérabilités XSS ou encore CSRF, nous avons besoin de trouver un chemin passant par un ou plusieurs états du modèle pour pouvoir générer une trace d'attaque potentielle. Dans le cas des XSS, s'il s'agit d'une injection réfléchie, il nous faut seulement un chemin passant par la page qui contient la réflexion alors que pour les injections persistantes, nous pouvons être amené à rechercher un chemin passant d'abord par un état où l'entrée vulnérable sera donnée à l'application, puis par un autre état qui peut être le même, où la réflexion aura lieu. Pour les attaques de type CSRF, cela revient à trouver un chemin jusqu'à l'entrée vulnérable comme dans le cas des injections XSS réfléchies.

Cette recherche de chemin faisable à partir du modèle peut se faire avec l'algorithme de Dijkstra [Dijkstra 1959] de recherche du plus court chemin dans un graphe. L'algorithme utilise les poids positifs de chaque transition pour déterminer le plus court chemin.

Nous pouvons aider l'algorithme pour qu'il considère en priorité les transitions dans le même état en jouant sur les poids des transitions. Pour les attaques où seule l'arrivée du chemin compte, l'effet de différents poids n'a pas d'impact. Par contre, pour les chemins d'attaques XSS persistantes potentielles, nous avons constaté en pratique qu'il n'est pas nécessaire de changer d'état et que la page qui contient la réflexion est le plus souvent accessible depuis le même état.

Comme nous utilisons un EFSM, chaque transition nécessite une certaine combinaison de paramètres pour qu'elle soit valide. Les prédicats étant calculés durant la phase de fouille de données à partir de valeurs concrètes, nous n'avons pas besoin d'une autre étape pour résoudre chaque prédicat puisque les valeurs validant le prédicat sont déjà connues.

Nous pouvons résumer le fonctionnement de l'algorithme de détection de chemin par la procédure de l'algorithme 19 qui prend en paramètre un modèle inféré, l'état intermédiaire du chemin (S_0 par défaut), l'état d'arrivée et un poids $p > 1$ pour les transitions entre états différents. Nous recherchons ici les chemins faisables, ce qui implique que l'algorithme doit tenir compte des prédicats.

Le poids $p > 1$ est utilisé pour augmenter le coût des transitions inter-états et ainsi privilégier les transitions qui restent dans le même état. Le choix de la valeur de p doit se faire selon le modèle. Le but étant de garder une longueur de chemin raisonnable tout en privilégiant l'état dans lequel on se trouve.

Algorithme 19 : Detection_Chemin(M, S_i, S_f, p) où M est un modèle inféré, S_1 l'état de départ (S_0 par défaut) et S_1 l'état d'arrivée.

```
foreach transition t(s1, s2) ∈ T do
    if s1 = s2 then
        Affecter le poids 1 à t;
    end
    else
        Affecter le poids p à t;
    end
end
Appliquer Dijkstra sur M en partant de S0 pour aller à Si;
Appliquer Dijkstra sur M en partant de Si pour aller à Sf;
```

Nous disposons aussi d'un autre moyen de trouver des chemins dans le modèle en utilisant le model-checker du projet SPaCIoS [SPaCIoS 2010]. Cette méthode est décrite dans la section suivante 6.5.

6.5 Model-checking - approche SPaCIoS

Une des approches de détection de vulnérabilité du projet SPaCIoS inclut un modèle EFSM inféré provenant soit du composant d'inférence en boite noire, soit le composant d'inférence depuis le code source. Dans la figure 6.4, nous avons la représentation de l'approche en boite noire.

Une fois le modèle inféré sous forme d'EFSM, il est traduit en ASLan++ [Oheimb 2012] qui est le langage haut niveau de définition de modèle du projet SPaCIoS. Il est ensuite possible de définir des propriétés de sécurité [D3.2 2011] notamment avec des propriétés sur les paramètres, des formules LTL [Pnueli 1977] ou en choisissant des canaux de communication avec différents niveaux de sécurité.

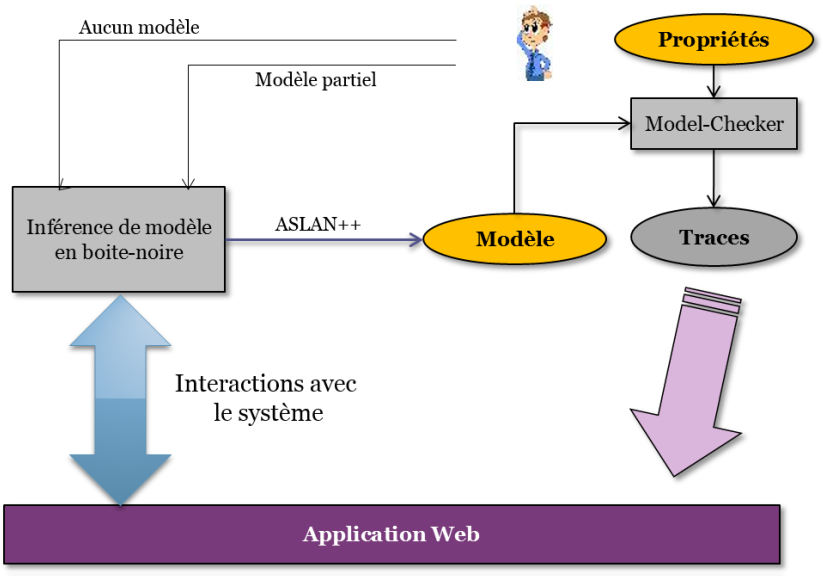

FIGURE 6.4 – Utilisation du modèle inféré dans l'approche SPaCIoS

Chapitre 7

Outils et expérimentations

L'outil SPaCIoS ainsi que tous ses composants sont mis à disposition des utilisateurs sous licence GPLv3 [Foundation² 2007]. Nous avons développé une plate-forme d'inférence pour pouvoir à la fois tester différents algorithmes d'inférence, mais aussi de s'affranchir des problèmes liés aux licences. Ce chapitre présente la plate-forme nommée SIMPA (**S**impa **I**nfers **M**odels **P**retty **A**utomatically) qui implémente plusieurs algorithmes d'inférences et notamment ceux présentés dans les chapitres précédents.

Sommaire

7.1 SIMPA : une plate-forme d'inférence d'automates

La plate-forme a été principalement développée pour implémenter les approches d'inférences en boite noire du projet SPaCIoS. Comme la plupart des autres composants du projet, il a été développé sous Java SE v7 et peut fonctionner à la fois sur les environnements Windows, Linux et Mac. La version courante de SIMPA est constituée de 15000+ lignes de codes réparties dans 120+ classes et 22 paquets. SIMPA est disponible en téléchargement sur la forge IMAG à cette adresse :

https://forge.imag.fr/projects/simpa

En plus des algorithmes d'inférence, SIMPA inclut le générateur d'adaptateurs de test présenté dans le chapitre 4.3 ainsi que plusieurs bibliothèques pour la communication suivant divers protocoles comme HTTP ou SIP. L'architecture de SIMPA ainsi que ses principaux modules sont représentés dans la figure 7.1.

L'architecture a été conçue de façon à ce que la partie inférence soit bien séparée de la partie adaptateur de test. Il est donc possible de modifier et ajouter des algorithmes d'inférence en réutilisant les modèles déjà définis puis dans un second temps, de travailler sur l'adaptateur correspondant ou de réutiliser les adaptateurs existants. Ils peuvent utiliser des bibliothèques externes (non fournies) pour la communication sans être liés aux algorithmes d'inférence.

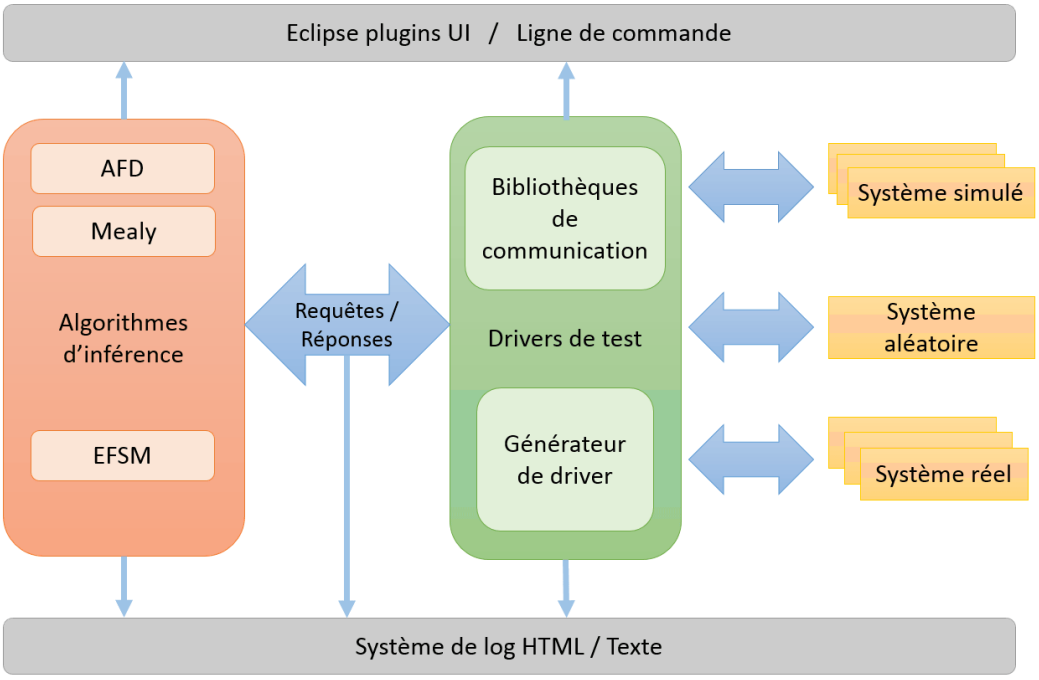

FIGURE 7.1 – Architecture globale de SIMPA

Les différents composants et leurs fonctionnalités sont présentés dans les sections suivantes.

7.1.1 Algorithmes d'inférence

Ce module est implémenté dans les paquets et sous-paquets *learner* et il contient différents algorithmes d'inférence basés sur L* et sur les Z-Quotients. Il contient no-

tamment l'algorithme d'Angluin dans sa version originale [Angluin 1987] ainsi que les algorithmes d'inférence Lm^* [Shahbaz 2009] avec la méthode de gestion des contre-exemples définie dans [Irfan 2010b] et l'algorithme L^*m [Steffen 2011]. Pour l'inférence d'EFSM, nous avons implémenté l'algorithme du projet SPaCIoS ainsi que l'algorithme Z-Quotient étendu du chapitre 5.

Chaque algorithme d'inférence est implémenté dans une classe qui hérite de la classe *Learner*. Cette classe possède les méthodes de base à tous les algorithmes d'inférence, c'est-à-dire *learn*() et *createConjecture*(). Chaque classe peut être ensuite complétée par différentes méthodes suivant le type de modèles et les besoins de l'algorithme.

7.1.2 Adaptateurs de test

Ce module est implémenté dans les paquets et sous-paquets *drivers* et il contient les différents adaptateurs développés dans le cadre du projet SPaCIoS comme celui pour le protocole SIP et l'application Web WebGoat. Il contient aussi un ensemble d'adaptateurs accompagnés de la définition du système correspondant. Cette configuration est utilisée pour tester les exemples et cas particuliers des algorithmes d'inférence. De plus, il contient, pour les FSM et EFSM, un adaptateur de test générique qui se charge de construire aléatoirement un système suivant divers paramètres. Cette classe est utilisée dans les tests de performance des algorithmes.

Chaque type d'adaptateur hérite au moins de la classe *Driver* qui contient les méthodes communes à tous les adaptateurs comme *reset* ou encore *execute*. Chaque classe peut être ensuite complétée par différentes méthodes suivant le type de modèles (sorties, les paramètres) et les besoins de l'algorithme (contre-exemples).

Comme défini dans le chapitre 4, il existe deux principaux types d'adaptateur, ceux basés sur HTML et ceux sur HTTP. Chacun de ses types correspond à une classe abstraite d'adaptateur. Par exemple, pour construire un adaptateur de type HTTP, il faudra que l'adaptateur hérite de la classe $HTTPWebDriver$ et non $HTMLWebDriver$.

7.1.3 Les différents systèmes

Il existe trois principaux types de système implémentés dans SIMPA, les simulés, aléatoires et réels.

7.1.3.1 Les systèmes simulés

Les systèmes simulés sont définis dans une classe séparée en héritant de la classe *Automata*, *Mealy* ou encore *EFSM*. Pour chaque système, il suffit alors de compléter la méthode *getAutomata*() qui est censée renvoyer l'automate correspondant. Dans le listing 7.1, nous créons un nouvel EFSM nommé *NSPK*, puis trois états dont $s1$ qui sera l'état initial. Ensuite, nous définissons une transition entre $s1$ et $s2$.

```
public static EFSM getAutomata() {
  EFSM test = new EFSM("NSPK");
  State s1 = test.addState(true);
  State s2 = test.addState();
  State s3 = test.addState();
  test.addTransition(new EFSMTransition(
      test, s1, s2, "m1", "m2",
      new IOutputFunction() {
        @Override
        public List<Parameter> process(EFSM
            automata,
            List<Parameter> inputParameters
                ) {
          List<Parameter> p = new ArrayList
              <Parameter>();
          int n = new Random().nextInt
              (1000);
          automata.setMemory("n", String.
              valueOf(n));
          p.add(new Parameter(
              inputParameters.get(0).value,
              Types.NUMERIC));
          p.add(new Parameter(String.
              valueOf(n), Types.NUMERIC));
          return p;
        }
      }));
  ...
}
```

Listing 7.1– Extrait du code d'un système simulé (NSPK)

7.1.3.2 Les systèmes aléatoires

Les systèmes aléatoires se configurent depuis la ligne de commandes de SIMPA. Il existe un certain nombre de paramètres permettant de jouer précisément sur le modèle qui sera généré. Chaque modèle ainsi créé pourra être sauvegardé pour être inféré une autre fois si nécessaire. Pour chaque paramètre, nous pouvons définir un minimum et un maximum ce qui permet de générer une grande variété de modèles. Voici la liste de quelques paramètres pour les EFSM :

— Nombre d'états
— % de création de transition entre deux états
— Nombre d'entrées
— Nombre de sorties
— Nombre de paramètres
— Domaine des paramètres
— Type et complexité des prédicats

De plus, pour ce type de systèmes, SIMPA inclut un module de test dédié qui permet de lancer à la suite un grand nombre d'exécutions d'inférence et d'en récupérer les résultats sous forme de statistiques détaillées.

7.1.3.3 Les systèmes réels

Pour SIMPA, les systèmes peuvent être quelconques à condition qu'il soit possible d'envoyer des requêtes et d'observer les réponses. Dans le cas des protocoles non basés sur HTTP, l'adaptateur doit être entièrement défini manuellement alors que pour les applications basées sur HTTP, la création de l'adaptateur est intégrée à la procédure d'inférence.

7.1.4 Systèmes de log

Le système de log permet d'enregistrer chaque requête et réponse du système lors d'une inférence. Le fait de proposer une sortie HTML des log permet d'analyser plus rapidement et efficacement les résultats d'une inférence. Pour des raisons de performances, il est aussi possible de désactiver ce module pour ne conserver que le résultat d'inférence.

7.2 Intégration

Dans le but de faciliter promotion et l'utilisation de l'outil SPaCIoS, chaque composant de l'outil a été intégré en tant que plug-in Eclipse. De plus, l'outil SPaCIoS a aussi été intégré sur la plate-forme de test NESSoS [NESSoS 2010].

7.2.1 Intégration en tant que plug-in Eclipse

SIMPA est utilisable en ligne de commandes, mais aussi à travers Eclipse. Nous avons développé une série de plug-ins qui permet d'intégrer SIMPA dans l'interface graphique d'Eclipse. Comme tous les autres composants de l'outil SPaCIoS, SIMPA est disponible à travers le menu *SPaCIoS* comme montré dans la figure 7.2. Il est

aussi possible de passer par le menu de création de projets pour pouvoir créer un nouvel adaptateur de test comme indiqué dans la figure 7.3.

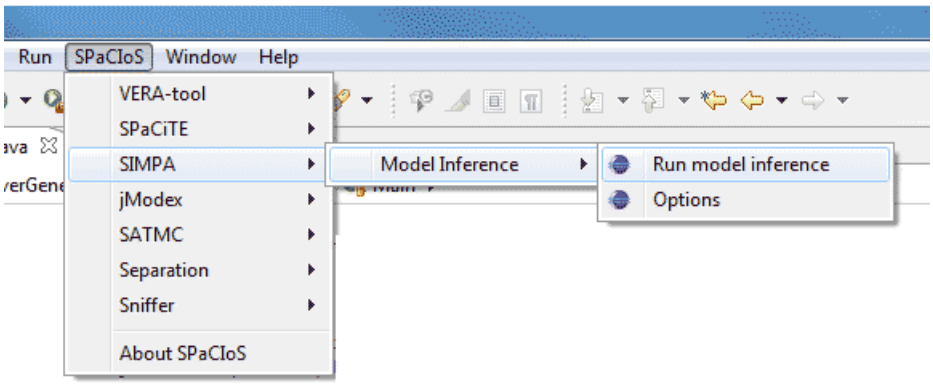

FIGURE 7.2 – Menu SIMPA dans Eclipse

Une fois le projet créé, un ensemble de dossiers est ajouté pour contenir les différents résultats de l'inférence, comme les logs et les modèles. Dans la figure 7.4, nous avons l'arborescence du projet une fois une inférence terminée. Nous distinguons entre autres le modèle sous forme ASLan++.

7.2.2 Intégration dans NESSoS

En plus des plug-ins Eclipse, nous avons aussi intégré SIMPA dans la plate-forme de test de la sécurité NESSoS dans le but de rendre son utilisation possible par un plus grand nombre de personnes. SIMPA est donc disponible sous forme de composant SDE comme illustré dans la figure 7.5.

Il est aussi disponible à travers la palette de la plate-forme comme dans la figure 7.6.

7.3 Cas d'études

Dans cette section, nous présentons divers résultats obtenus avec les algorithmes d'inférence définis dans les chapitres précédents. Nous avons appliqué nos méthodes d'inférence à un cas d'étude du projet SPaCIoS, c'est-à-dire l'application WebGoat ainsi qu'au protocole SIP, protocole souvent pris comme cible pour l'inférence, mais toujours sur une version simulée [Aarts 2010b] et non sur une implantation réelle.

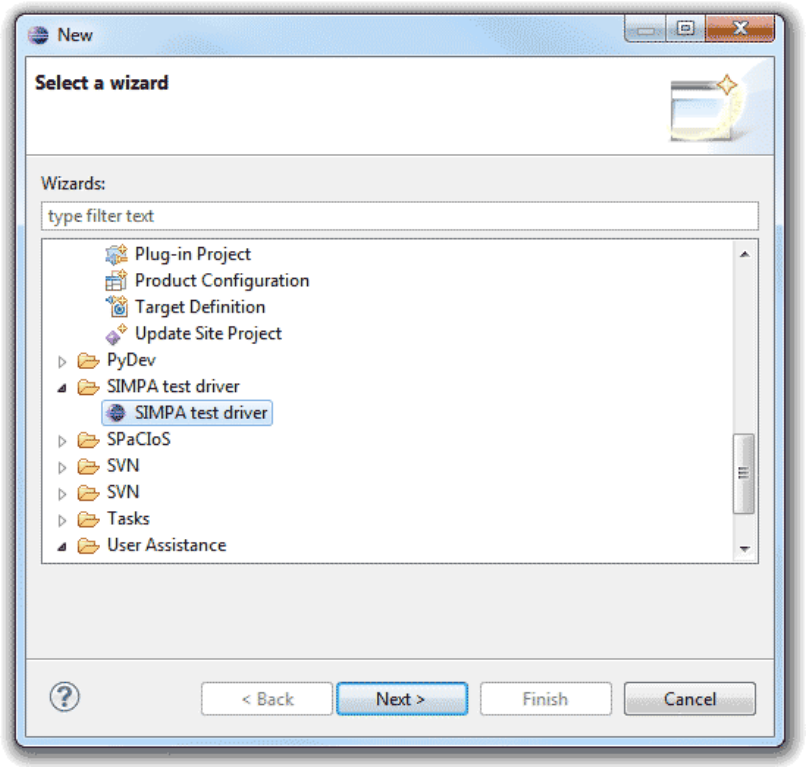

FIGURE 7.3 – Création d'un adaptateur dans SIMPA

7.3.1 Applications de WebGoat

WebGoat [OWASP 2010] est une plate-forme d'apprentissage de la sécurité Web à travers divers exercices. WebGoat contient plusieurs applications Web qui servent à montrer étape par étape les vulnérabilités et comment les exploiter. Créé par l'OWASP, elle se compose d'une archive contenant une application J2EE ou .NET qui permettent de lancer la plate-forme et on y accède localement depuis le navigateur. Chaque application contient les indications nécessaires pour trouver et exploiter la vulnérabilité ainsi que la solution. Une description du fonctionnement des applications se trouve dans la section 4.2.2 du chapitre 4.

Si chaque application exhibe une vulnérabilité particulière, l'application en elle-même est souvent la même. C'est une application de gestion des ressources humaines avec une gestion de profils. Une dizaine d'utilisateurs sont enregistrés et chacun possède un certain niveau d'accès. Selon ce niveau, il est possible à un utilisateur de créer ou supprimer le profil des autres employées. Dans la figure 7.7, nous pouvons voir la page principale de l'application une fois que l'utilisateur s'est authentifié.

L'application étant souvent la même pour un grand nombre de leçons WebGoat,

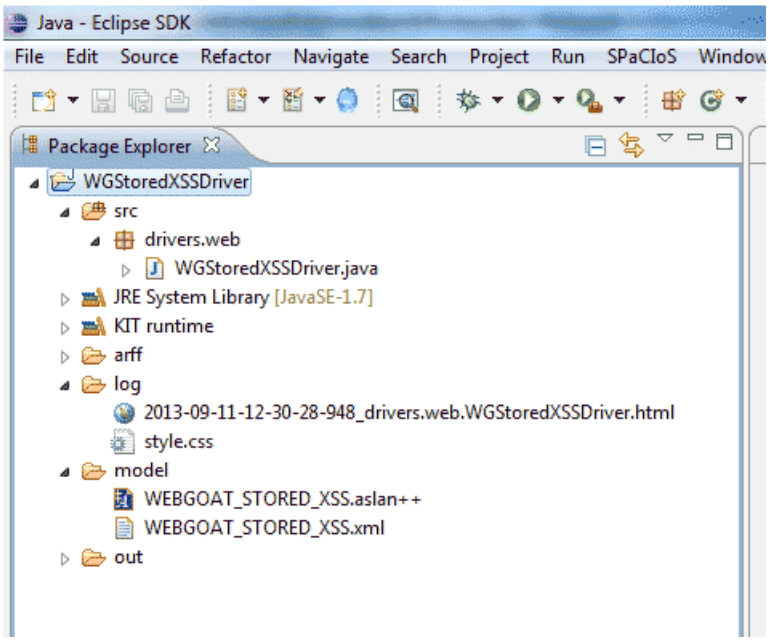

FIGURE 7.4 – Arborescence d'un projet dans SIMPA

les modèles qui en sont inférés sont assez proches.

7.3.1.1 XSS persistant

Comme décrites dans la section 2.1.2, les vulnérabilités de type XSS persistant permettent d'effectuer des attaques puissantes puisqu'elles peuvent toucher un grand nombre d'utilisateurs très rapidement. Cela permet notamment à l'attaquant de pouvoir rediriger la personne vers un site malicieux (hameçonnage), voler des informations sur les clients ou encore rendre la page difficilement consultable.

Modèle

Le modèle de l'application obtenu après inférence est décrit dans la figure 7.8. Nous obtenons donc un modèle à deux états :

— Dans l'état S_0, l'utilisateur n'est pas authentifié au système. La seule action possible et qui est valide est l'action de *login*. S'authentifier avec des informations valides permet de passer dans l'état suivant S_1 tandis que les autres actions n'ont aucun effet puisque le système reste dans le même état. Notons que nous avons fourni comme valeurs de paramètres, les credentiels valides de l'utilisateur *John*, l'administrateur du système et *Larry*, un employé.

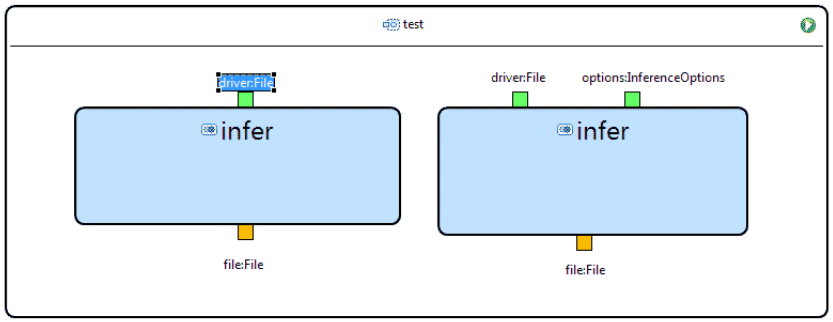

FIGURE 7.5 – Composant SDE SIMPA dans NESSoS

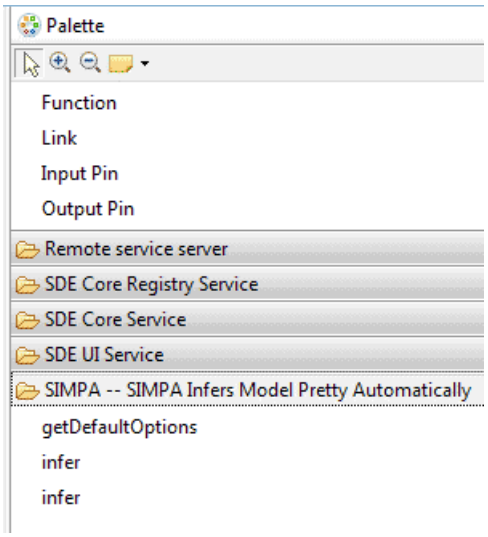

FIGURE 7.6 – Palette SIMPA dans NESSoS

— Dans l'état S_1, l'utilisateur devient capable d'exécuter d'autres actions. Il est au minimum possible de consulter et de modifier son profil. Selon les droits de l'utilisateur, il est aussi possible de faire de même pour des profils que n'appartiennent pas à l'utilisateur. La figure 7.8 est une représentation simplifiée du modèle obtenu. Par exemple, il n'y a que le paramètre qui correspond au champ *street* qui est représenté alors que tous les autres paramètres sont aussi considérés.

L'application est relativement simple dans le sens où le nombre de paramètres reste raisonnable tout comme le nombre d'états. Le modèle obtenu est cohérent avec le comportement de l'application.

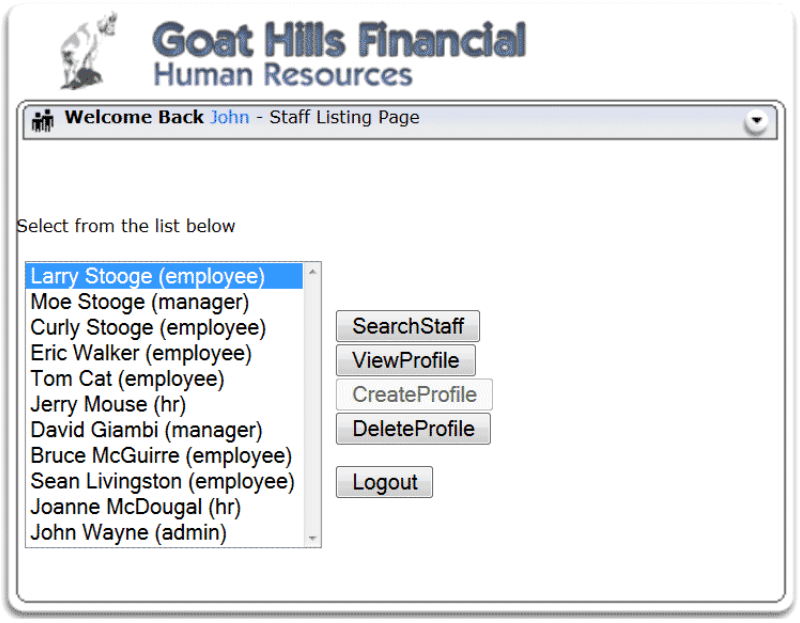

FIGURE 7.7 – Page principale de l'application Stored XSS

Détection de vulnérabilités

Pour la détection de vulnérabilité, nous avons suivi l'approche du projet SPaCIoS définie dans la section 6.5. Nous complétons le modèle $ASLan++$ obtenu en ajoutant des propriétés de sécurité. Pour cette application, nous recherchons de possibles injections XSS, c'est-à-dire les endroits où un ou plusieurs paramètres d'une requête sont affichés sur une page.

Depuis le modèle, on voit clairement que les paramètres passés dans la requête de mise à jour du profil sont affichés lors de l'affichage du profil. Nous allons donc marquer la modification du profil par un fait *attacked* puis l'affichage de ce profil par le fait *view*. Grâce à la formule LTL 7.1, nous recherchons les traces qui permettent à un utilisateur de modifier un profil qui sera vu ensuite par un autre utilisateur. Nous appellerons ces deux utilisateurs *victim* et *attacker*.

$$\forall profil.\Box(victim.view(profil) \rightarrow \neg attacked(profil)) \tag{7.1}$$

Le model-checker du projet SPaCIoS trouve en quelques secondes la trace suivante :

1. $attacker \rightarrow system : login(101, \text{“}larry\text{”})$
2. $system \rightarrow attacker : staffList()$

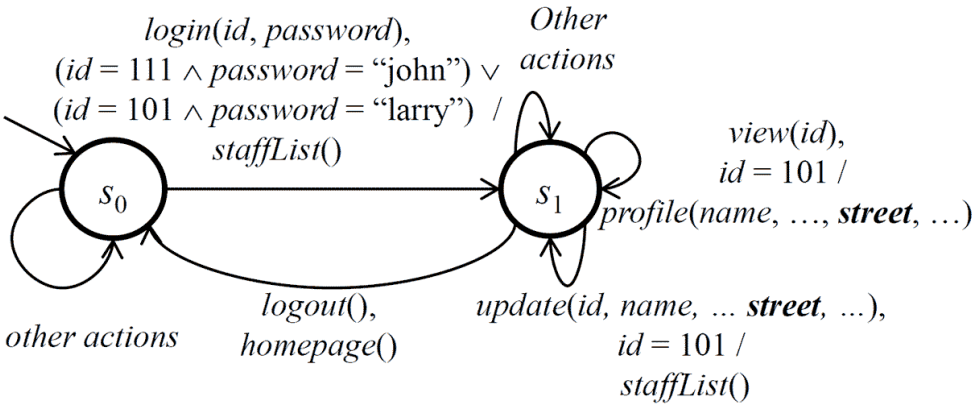

FIGURE 7.8 – Modèle inféré de l'application WebGoat

3. $attacker \rightarrow system : update(101, \text{“}larry\text{”}, fields...)$
4. $system \rightarrow attacker : staffList()$
5. $victim \rightarrow system : login(111, \text{“}john\text{”})$
6. $system \rightarrow victim : staffList()$
7. $victim \rightarrow system : view(101)$
8. $system \rightarrow victim : profil(\text{“}larry\text{”}, fields...)$

Cette trace correspond à l'exploitation d'une vulnérabilité de type XSS persistant. La première étape est effectuée par l'attaquant qui sera *Larry*, un utilisateur de l'application qui ne possède pas de droits particuliers. Une fois connecté, n'ayant pas les droits nécessaires pour voir ou modifier les autres profils, l'attaquant va mettre à jour son propre profil. Ici, l'attaquant peut très bien mettre à jour son profil avec un script JavaScript au lieu des informations normales de son profils. Il n'est pas nécessaire pour l'attaquant de se déconnecter de l'application.

La deuxième partie de l'attaque met en jeu la victime qui est *John*, l'administrateur de l'application. Une fois connectée, il lui suffit de visiter le profil de l'attaquant pour voir les différents champs de son profil. Évidemment, si l'application est réellement vulnérable à une attaque de type injection XSS, le script JavaScript qui est mis dans le profil, sera exécuté par l'administrateur ce qui peut entraîner un vol de cookie qui permettra à l'attaquant de se faire passer pour l'administrateur et donc de réussir son attaque.

Depuis le modèle de l'application, il était assez simple de trouver les paramètres qui sont réfléchis dans les pages, qu'ils soient, directement réfléchis (XSS réfléchi) ou indirectement (XSS persistant). Ici, le model-checker est particulièrement adapté aux injections de type XSS persistant puisqu'il permet de trouver les chemins dans

l'application qui généreront une attaque.

Confirmation de la vulnérabilité

Les traces produites par le model-checker ne sont que des traces potentielles d'attaques puisqu'il n'est pas certain que le système ne filtre pas le paramètre comme défini dans la section 2.1.2.3. Nous pouvons utiliser maintenant le moteur d'exécution de test du projet SPaCIoS qui permet, à partir d'une trace et de la sémantique des éléments de cette trace, de l'exécuter sur une instance de l'application.

Pour vérifier s'il y a réellement une vulnérabilité, nous pouvons rejouer la trace en utilisant un script JavaScript à la place des informations du profil. Si le script est retranscrit tel quel une fois le profil affiché, nous aurons bien détecté une vulnérabilité. De plus, l'application WebGoat nous signalera si nous avons correctement exploité la vulnérabilité.

Après les tests sur l'application, il apparaît que le champ *street* du profil est vulnérable à une attaque de type XSS. Une fois le profil visité par l'utilisateur *John*, nous avons bien un message de confirmation de réussite de la part de l'application WebGoat. Bien qu'ils n'aient pas d'impacts sur le succès de l'exercice, les tests montrent que plusieurs autres champs comme *City/State* sont aussi vulnérables.

7.3.1.2 Contournement du contrôle d'accès - couche donnée

Nous avons aussi essayé d'appliquer notre méthode de détection de vulnérabilité sur un autre type d'application. Nous avons choisi une vulnérabilité dans la couche de données d'une application. Dans cette application, nous sommes censés être uniquement un employé avec aucun droit particulier et nous devons trouver un moyen d'afficher le profil d'un autre employé.

Modèle

Comme précédemment nous avons modélisé l'application et obtenu le modèle décrit dans la figure 7.9. Nous obtenons un modèle équivalent à celui précédemment inféré. En effet, bien que la vulnérabilité soit différente, l'application en elle-même reste la même.

Sur la figure 7.9, nous avons uniquement représenté les parties intéressantes pour trouver la vulnérabilité, celle qui permet de voir un profil.

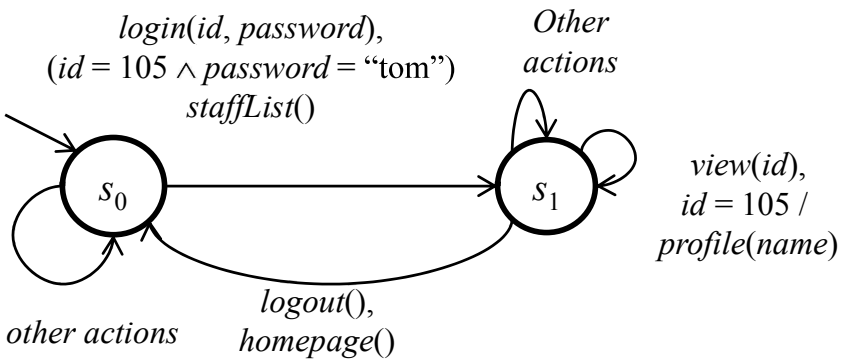

FIGURE 7.9 – Modèle inféré - Contournement de la couche donnée

Détection de vulnérabilités

Avec la formule LTL 7.2, nous allons chercher les traces qui permettent de voir un autre profil que celui que nous sommes censés voir, le nôtre uniquement.

$$\forall profil.\Box(attacker.view(profil) \rightarrow attacker.isOwn(profil)) \tag{7.2}$$

Le fait *isOwn* est défini pour que seul le profil correspondant à la personne qui est connecté puisse renvoyer *vrai*.

Le model-checker du projet SPaCIoS trouve la trace suivante :

1. $attacker \rightarrow system : login(105, \text{“}tom\text{”})$
2. $system \rightarrow attacker : staffList()$
3. $attacker \rightarrow system : view(110)$
4. $system \rightarrow attacker : profil(\text{“}john\text{”})$

Cette trace nous indique qu'un attaquant peut s'authentifier sur l'application puis faire une requête d'affichage sur un profil qui ne lui appartient pas. C'est exactement le type d'attaque que l'on souhaite effectuer. La vulnérabilité potentielle se situe ici dans la vérification de l'ID de profil que l'utilisateur envoie. Si aucune vérification que l'ID de profil envoyé correspond bien à l'utilisateur ou que l'utilisateur possède bien les droits de voir ce profil, alors il y a une attaque possible.

Confirmation de la vulnérabilité

Nous pouvons utiliser maintenant le moteur d'exécution de test pour exécuter cette trace sur une instance de l'application pour vérifier si on obtient bien le profil d'une autre personne ou si l'application va nous en empêcher.

Nous obtenons bien le profil de *John* qui correspond à l'administrateur de l'application alors que nous sommes connectés sous le compte de *Tom* qui n'a aucun droit particulier. WebGoat nous confirme aussi que nous avons réussi cet exercice.

7.3.1.3 Contournement du contrôle d'accès - couche business

Dans cet exercice, la vulnérabilité ne se trouve plus dans la vérification des valeurs de l'ID du profil, mais plutôt dans l'action que l'on souhaite faire sur les profils. Un utilisateur simple n'est pas censé pouvoir créer ni supprimer un profil. Ces actions sont réservées aux utilisateurs ayant les droits administrateur ou manager. Le but de cet exercice est de trouver un moyen de supprimer un profil qui ne nous appartient pas.

Modèle

Dans la même façon que précédemment, nous avons inféré un modèle de l'application dont les parties intéressantes sont décrites dans la figure 7.10. Nous obtenons un modèle équivalent à celui précédemment inféré.

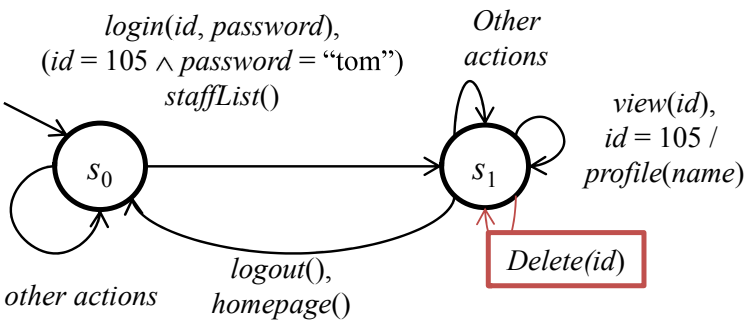

FIGURE 7.10 – Modèle inféré - Contournement de la couche business

Le modèle obtenu est cohérent avec le comportement de l'application. Nous pouvons remarquer qu'il n'y a pas d'actions de suppression du profil.

Détection de vulnérabilités

Pour cette vulnérabilité, nous ne pouvons pas utiliser de formule LTL comme dans le cas précédent. Comme l'action de suppression ne fait pas partie du modèle, il n'existe aucun chemin dans ce modèle qui permet d'arriver à une suppression. La seule solution est d'ajouter cette action au modèle.

Nous pouvons maintenant utiliser la formule LTL 7.3 pour trouver les chemins menant à la suppression.

$$\forall profil.\Box(!attacker.delete(profil)) \tag{7.3}$$

Le model-checker du projet SPaCIoS trouve la trace suivante :

1. $attacker \rightarrow system : login(105, \text{“}tom\text{”})$
2. $system \rightarrow attacker : staffList()$
3. $attacker \rightarrow system : delete(profil(1))$
4. $system \rightarrow attacker : staffList()$

Confirmation de la vulnérabilité

Même si l'action n'est pas disponible au niveau de l'interface de l'application, la trace trouvée suggère qu'il est quand même possible de supprimer un profil et que cette action de suppression ne serait pas entièrement désactivée.

Nous obtenons bien la suppression du profil de notre choix et WebGoat nous confirme que nous avons réussi l'exercice.

7.3.2 SIP (Session Initiation Protocol)

Le protocole *SIP* [Rosenberg 2002] est largement utilisé dans le domaine des télécommunications et spécialement pour gérer les appels audio et vidéos. Ce protocole a déjà été utilisé pour valider des méthodes d'inférence, mais ces expériences [Aarts 2010a] ont été faites sur un simulateur de réseaux : NS-2 [Mccanne]. Dans nos expérimentations, l'algorithme d'inférence interagit directement avec l'implantation du protocole *SIP* sur Internet.

Pour cela, nous avons créé un nouveau compte sur le site *iptel.org* qui fournit un service *SIP* gratuit. D'après la documentation sur le site, ce service sert aussi de

plate-forme de test d'interopérabilité logiciel/matériel. Nous allons inférer le serveur SIP implémenté par *iptel.org*. Les entrées sont paramétrées de façon à passer un appel au service écho. Ce service sert uniquement à tester si un client est bien configuré pour accéder au service. Nous utilisons la version UDP (User Datagram Packet) du protocole, la plus courante.

SIP contient 14 types de requêtes différentes, dans notre cas, nous allons nous concentrer sur les quatre types de base, les plus utilisés pour passer un appel : *Register*, *Invite*, *Ack* et *Bye*. Les sorties seront les codes de réponses ou l'identificateur *Timeout* quand aucune réponse n'est reçue. Comme nous n'avons aucune information sur les séquences discriminantes de ce protocole *SIP*, l'ensemble Z sera initialement vide. Les contre-exemples seront fournis manuellement.

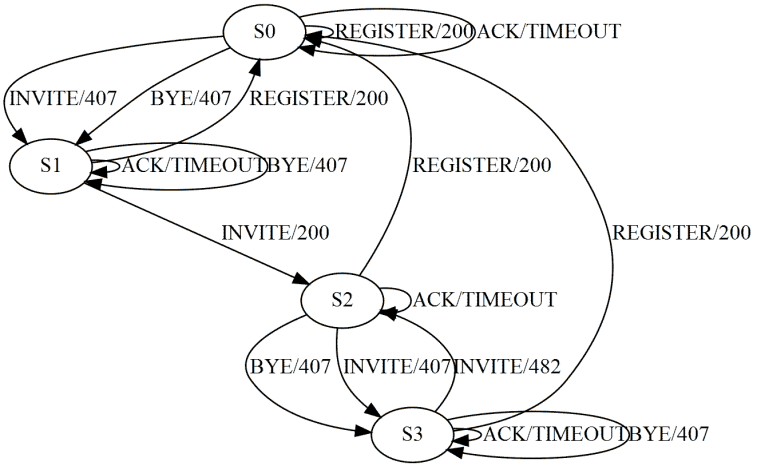

FIGURE 7.11 – Protocole SIP inféré de iptel.org

La Figure 7.11 montre le modèle inféré du protocole *SIP* implémenté par *iptel.org*, qui est obtenu en 26 requêtes et 10 secondes. Ce modèle correspond aux spécifications du protocole SIP. Seul un contre-exemple est nécessaire, il s'agit de *Invite.Invite* : pour pouvoir passer un appel en utilisant *iptel.org*, nous devons être authentifié avant. Ici, le premier *Invite* envoyé par le client n'est pas suffisant, nous recevons une réponse avec le code 407 qui signifie "Proxy authentication required" ainsi que le nonce nécessaire à l'authentification. Ensuite, nous pouvons envoyer notre requête *Invite* authentifiée et une réponse "OK" (code 200) est obtenue.

Nous avons obtenu assez rapidement un modèle de cette implantation du protocole SIP, mais l'algorithme peut facilement être adapté pour d'autres implantations. En fait, il suffit de changer l'adresse du fournisseur de service ainsi que les détails sur le compte utilisé. *SIP2SIP* est un autre fournisseur de service *SIP* gratuit offert par AG Projects. Même si ce protocole est très utilisé et bien défini dans la RFC3261, il se peut que les implantations diffèrent sensiblement. *SIP2SIP* utilise la pile *SIP* OpenSIPS XS alors que *iptel.org* utilise *SIP* Express Router (SER). Nous

avons donc inféré un modèle du protocole *SIP* de la plate-forme *SIP2SIP*. Comme précédemment, les entrées sont paramétrées pour passer un appel au service écho.

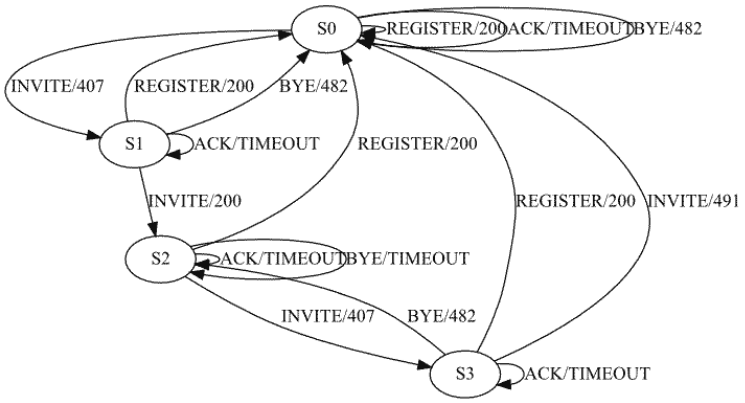

FIGURE 7.12 – Protocole SIP inféré de SIP2SIP

La Figure 7.12 présente le modèle du serveur *SIP* de *SIP2SIP*, qui est obtenu avec 26 requêtes et en une minute. Nous avons utilisé le même contre-exemple que précédemment et le modèle obtenu correspond spécifications du protocole.

Bien que ce soit le même protocole qui soit implémenté par ces deux fournisseurs, ces deux implantations présentent quelques différences. Passer un appel nécessite toujours deux requêtes *Invite*, mais si l'on tente de passer un autre appel pendant un appel, *SIP2SIP* retourne le code 491 qui correspond à une mise en attente, puis le système retourne à son état initial alors que pour *iptel.org*, nous avons un code 482 correspondant à *boucle détectée* puis le système revient à son état précédent. Il y a donc une petite différence entre ces deux protocoles dans la gestion des appels ; *SIP2SIP* préfère mettre l'appel en attente alors que *iptel.org* détecte qu'un appel est déjà en cours et considère cette nouvelle tentative comme une boucle.

Il y a encore une différence dans la gestion de fin des appels (requête *BYE*). Dans l'état initial, envoyer une requête de fin est invalide parce qu'il n'y a pas encore d'appel en cours. Pour *iptel.org*, cette requête est tout de même possible et nous recevons la demande d'authentification (code 407) alors que *SIP2SIP* détecte que l'on essaye de quitter un appel non existant et considère cela comme une boucle (code 482).

Ces expérimentations montrent que la méthode d'inférence proposée peut être utile pour détecter de subtiles variations dans différentes implantations d'un même système, par exemple, vérifier leur interopérabilité.

Chapitre 8

Conclusion

Ce chapitre conclut la thèse en rappelant les contributions et résultats présentés dans les chapitres précédents, puis il présente quelques publications effectuées durant cette thèse et termine par différentes idées d'améliorations, directions et d'applications de l'inférence de modèles.

Sommaire

8.1 Résumé

L'inférence de modèle en boite noire permet à un testeur logiciel ou un spécialiste en sécurité de profiter, à faible coût, de toute la panoplie de méthodes de test basé sur les modèles qui ont montré leur efficacité ces dernières années. Cette thèse se concentre sur l'inférence d'application Web pour fournir un modèle aux outils de test de la sécurité basé sur l'utilisation de modèles pour de la détection de vulnérabilités. Pour cela nous avons développé diverses techniques visant à automatiser, adapter et améliorer les algorithmes d'inférence.

— Dans le but d'automatiser l'inférence, nous avons développé une méthode de génération d'adaptateur de test automatique. Cette méthode utilise un collecteur qui permet de parcourir une application Web et d'en extraire les entrées, sorties et les paramètres correspondants à l'aide d'algorithmes de partitionnement des données. Le collecteur analyse les paramètres détectés pour ne conserver que ceux ayant un intérêt pour l'inférence. Dans le but

d'encourager l'utilisation d'inférence pour les applications Web, le collecteur est aussi disponible sous forme d'outil séparé et peut être utilisé uniquement comme moteur d'exécution de tests si besoin.

— Nous avons développé une méthode d'inférence basée sur L* qui permet de gérer les paramètres non déterministes, étape indispensable pour pouvoir inférer correctement des applications Web qui sont maintenant pratiquement toutes pourvues de mécanismes de sécurité incluant des valeurs pseudo-aléatoires. L'algorithme ajoute une table d'observation spécialement pour les données. Cette table de données permet la détection des paramètres aléatoires qui seront ensuite réutilisés dans les requêtes suivantes.

— Nous avons développé une méthode d'inférence de machine de Mealy basé sur les Z-Quotients. Cette méthode a l'avantage d'être plus souple que les méthodes de type Angluin. En effet, elle permet d'ajouter des entrées pendant l'exécution de l'algorithme. Utilisant des séquences d'entrées pour distinguer les états, elle ajoute la possibilité d'utiliser des séquences d'attaques pour guider l'inférence.

— Nous avons développé une approche combinant les avantages des deux algorithmes précédents et spécialement conçue pour les applications Web. Nous utilisons les méthodes du générateur d'adaptateur de test pour l'exploration de l'application pendant l'inférence. La gestion du non-déterminisme de certains paramètres de l'application et la possibilité de prendre en compte des entrées découvertes pendant l'inférence permettent de mieux automatiser la procédure. Les algorithmes de fouille de données ont été étendus pour considérer plusieurs paramètres dans les gardes de chaque transition tout en privilégiant les plus importants.

— Nous avons implanté ces algorithmes dans la plate-forme d'inférence libre nommée SIMPA et testé sur les cas d'études du projet SPaCIoS ainsi que des sites Web existants et nous avons découvert quelques vulnérabilités XSS.

8.2 Publications liées à la thèse

1. Alexandre Petrenko, Keqin Li, Roland Groz, **Karim Hossen**, Catherine Oriat. Inferring Approximated Models for Systems Engineering. 15th IEEE International Symposium on High Assurance Systems Engineering (HASE 2014), :249-253, Miami, Florida, USA, jan 2014. [Petrenko 2014]

   Construire des systèmes sécurisés et fiables demande des approches rigoureuses comme les méthodes formelles utilisant des modèles. Dans la plupart

des cas, ce type de modèle n'est pas disponible, mais il est possible d'en inférer un depuis le système. Cet article définit l'approche d'inférence des systèmes à entrées et sorties qui sont à la base de la théorie du test par machine à états finie. La notion de quotient initial avec un ensemble de caractérisation partielle se trouve au cœur de cette approche. Cet ensemble permet de contrôler la précision du modèle inféré. En tant que méthode d'inférence active, elle construit progressivement un modèle de plus en plus précis en utilisant des contre-exemples. Diverses expérimentations notamment sur le protocole SIP ont montré la faisabilité et l'efficacité de cette méthode,

2. Matthias Büchler, **Karim Hossen**, Petru Florin Mihancea, Marius Minea, Roland Groz, Catherine Oriat. Model Inference and Security Testing in the SPaCIoS Project. IEEE Working Conference on Reverse Engineering, CSMR-WCRE 2014, :411-414, Antwerp, Belgium, feb 2014. [Büchler 2014]

   Le projet SPaCIoS a pour but la validation et le test de la sécurité des services et des applications Web. Nous proposons une méthodologie et un outil centré sur l'utilisation de modèles décrits dans un langage de spécification spécialisé et supportant l'inférence de modèle, le test par mutation et le model-checking. Le projet a développé deux approches de reverse engineering de modèle depuis l'implantation de l'application. La première, en boite noire, est basée sur les interactions (typiquement HTTP) pour observer les différents comportements et construire progressivement un modèle. La seconde est basée sur l'analyse du code source quand il est disponible. Cet article présente la partie de rétro-ingénierie du projet puis les méthodes de détection de vulnérabilité utilisables avec l'outil SPaCIoS.

3. **Karim Hossen**, Catherine Oriat, Roland Groz, Jean-Luc Richier. Automatic Model Inference of Web Applications for Security Testing. SecTest2014, co-located with ICST2014, Cleveland, Ohio, USA, apr 2014. [Hossen 2014]

   Dans l'Internet des services (IoS), les applications Web sont le moyen le plus commun pour fournir l'accès à des ressources aux utilisateurs. La complexité de ces applications a grandi en même temps que le nombre de techniques et technologie de développement. Le test basé sur les modèles (MBT) a prouvé son efficacité dans le domaine du test logiciel, mais construire le modèle d'une application reste encore une tâche non triviale. Dans cet article, nous introduisons un algorithme d'inférence d'applications Web automatique et guidé par les vulnérabilités. Cette approche construire un modèle en combinant un collecteur Web et de l'inférence.

4. **Karim Hossen**, Roland Groz, Catherine Oriat, Jean-Luc Richier. Automa-

tic generation of test drivers for model inference of web applications. Fourth International Workshop on Security Testing (SECTEST 2013), Workshop of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST 2013), :441-444, Luxembourg, mar 2013. [Hossen 2013]

Les applications Web sont développées en tant que services en utilisant les standards du Web. Le test basé sur les modèles combiné avec une inférence de modèle active est une des méthodes de test des applications automatisées, en particulier pour la recherche de vulnérabilités. Mais une partie nécessaire à l'inférence reste à construire manuellement, l'adaptateur de test. Il contient une abstraction de l'application contenant les différentes entrées et sorties ainsi que leurs paramètres, mais aussi des fonctions de conversion pour faire le lien entre le niveau abstrait de l'algorithme et celui concret de l'application. Dans cet article, nous proposons une méthode générique d'abstractions des applications Web en utilisant un collecteur Web optimisé pour l'inférence de modèle.

5. **Karim Hossen**, Roland Groz, Jean-Luc Richier. Security Vulnerabilities Detection Using Model Inference for Applications and Security Protocols. Second International Workshop on Security Testing (SECTEST 2011), Workshop of the IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST2011), :534-536, Berlin, Germany, mar 2011. [Hossen 2011]

   L'Internet des services (IoS) représente une vision future de l'Internet dans laquelle les applications sont construites en combinant des services fournis par différents agents à travers Internet. Elles sont déployées à la demande et consommées à l'exécution selon la demande et d'une manière flexible. Le test basé sur les modèles est une méthode pour tester la sécurité des applications, mais elle nécessite d'avoir un modèle formel de l'application et la plupart du temps, ce type de modèle n'est pas disponible. L'inférence de modèle est une solution adaptée à ce problème et au test de la sécurité. Cet article présente quelques idées pour combiner une inférence de modèle améliorée avec du test pour assurer la sécurité des services.

8.3 Perspectives

Nous avons travaillé à créer une méthode d'inférence pour les applications Web et le test de la sécurité. Chacune des composantes de cette approche peut être étendue dans certaines directions. Il est aussi envisageable d'étendre les types des vulnérabilités détectables.

8.3.1 Couverture et non-régression

Généralement, l'inférence de modèle en boite noire permet de construire une sous-approximation du système. Le modèle reflète le comportement du système suivant les différentes entrées et valeurs de paramètres utilisés, mais tous les chemins possibles ne sont pas forcément étudiés. Un moyen d'attester de la complétude du modèle serait de mesurer la couverture du système avec une inférence. Cette méthode ne serait plus totalement en boite noire, mais permettrait d'avoir une idée des parties de l'application non testées. Combiné à une inférence en boite blanche pour la génération d'entrées permettant de tester ces parties, nous devrions obtenir une meilleure couverture et donc un modèle plus précis. D'autres approches de tests se concentrent sur la couverture [Alshahwan 2012] [Li 2008].

L'inférence de modèle pourrait être aussi utilisée comme test de non-régression. À chaque étape du développement, un modèle peut être généré avec toujours le même ensemble de valeurs de paramètres et ensuite comparé au modèle obtenu précédemment pour vérifier que les fonctionnalités existantes le sont toujours. Certains travaux permettent des tests de non-régression sur les applications utilisant la technologie AJAX [Roest 2010] ou encore en générant un modèle [Schur 2013].

8.3.2 Stratégies pour le collecteur

La construction d'un collecteur pour applications Web est un problème très étudié. En ce qui concerne la limitation due aux technologies AJAX, leurs parcours et leur indexation ont été étudiés dans [Choudhary 2012]. Il existe des travaux sur leurs parcours comme [Mesbah 2012] [Zhang 2012] qui déclenchent les évènements liés aux éléments de la page Web pour analyser leurs effets sur le DOM. Une autre méthode propose un collecteur basé sur un modèle [Choudhary 2013a]. Ces méthodes pourraient être appliquées dans la construction de l'adaptateur pour améliorer le nombre d'entrées extraites en ajoutant, par exemple, les appels AJAX liés à certains événements JavaScript.

8.3.3 Modélisation détaillée des paramètres

Certaines vulnérabilités, comme les XSS, nécessitent de comprendre exactement comment les valeurs en entrées sont utilisées dans les sorties. Pour chaque paramètre, il pourrait être utile de modéliser la façon dont il est utilisé sur les pages en sortie ou sur le code du côté serveur (SQL injection). Une inférence sur les filtres associés à chaque paramètre puis une comparaison avec les filtres existants ou une génération d'attaques suivant une grammaire d'attaques [Duchene 2012] pourrait permettre d'améliorer l'efficacité de la détection de vulnérabilité XSS.

8.3.4 Génération d'adaptateur de tests pour protocoles

La génération d'adaptateur de test pour les protocoles ne fonctionnant pas sur HTTP n'est pas encore supportée. Cependant, il existe des travaux sur l'analyse des messages échangés entre les différents agents du protocole pour en identifier la structure (taille, format) et ainsi identifier les entrées et sorties comme dans [Bossert 2012].

8.3.5 Inférence appliquée sur les binaires

L'inférence de modèle peut être aussi appliquée sur des binaires pour la recherche de vulnérabilités ou la rétro-ingénierie de certains composants. Avec de l'instrumentation de binaire, il serait possible de modéliser certains mécanismes comme la gestion des caches [Rueda Cebollero 2013] ou les allocateurs mémoires dont la connaissance du comportement est nécessaire pour l'exploitation [Feist 2014]. En se concentrant sur un ensemble restreint de fonctions, il serait possible d'en dégager le comportement à haut niveau pour guider les analyses ultérieures. En particulier, pour l'exploitation de certaines vulnérabilités liées à la réutilisation d'espace mémoire libéré (use-after-free), il est nécessaire de connaître le comportement de l'allocateur mémoire pour exploiter la vulnérabilité.

Bibliographie

[Aarts 2010a] Fides Aarts, Bengt Jonsson et Johan Uijen. *Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction*. In Alexandre Petrenko, Adenilso da Silva Simão et José Carlos Maldonado, editeurs, ICTSS, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010. (Cité en pages 50 et 143.)

[Aarts 2010b] Fides Aarts, Bengt Jonsson et Johan Uijen. *Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction*. In ICTSS, pages 188–204, 2010. (Cité en page 134.)

[Alshahwan 2012] Nadia Alshahwan et Mark Harman. *State Aware Test Case Regeneration for Improving Web Application Test Suite Coverage and Fault Detection*. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pages 45–55, New York, NY, USA, 2012. ACM. (Cité en page 151.)

[Angluin 1981] Dana Angluin. *A note on the number of queries needed to identify regular languages*. Inf. Control, vol. 51, pages 76–87, 1981. (Cité en page 28.)

[Angluin 1987] Dana Angluin. *Learning Regular Sets from Queries and Counterexamples*. Inf. Comput., vol. 75, no. 2, pages 87–106, 1987. (Cité en pages 6, 7, 24, 28, 31, 33, 37, 40 et 131.)

[Armando 2012] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, David Oheimb, Giancarlo Pellegrino, SerenaElisa Ponta, Marco Rocchetto, Michael Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani et Luca Viganò. *The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures*. In Cormac Flanagan et Barbara König, editeurs, Tools and Algorithms for the Construction and Analysis of Systems, volume 7214 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin Heidelberg, 2012. (Cité en page 4.)

[Balduzzi 2011] Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti et Engin Kirda. *Automated discovery of parameter pollution vulnerabilities in web applications*. In NDSS 2011, 18th Annual Network and Distributed System Security Symposium, 6-9 February 2011, San Diego, CA, USA, San Diego, ÉTATS-UNIS, 02 2011. (Cité en page 81.)

[Büchler 2014] Matthias Büchler, Karim Hossen, Petru Florin Mihancea, Marius Minea, Roland Groz et Catherine Oriat. *Model Inference and Security Testing in the SPaCIoS Project*. In IEEE Working Conference on Reverse Engineering, CSMR-WCRE 2014, pages 411–414, Antwerp, Belgium, feb 2014. (Cité en page 149.)

[Benjamin 2011] Kamara Benjamin, Gregor Von Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan et Iosif Viorel Onut. *A Strategy for Efficient Crawling of Rich Internet Applications*. In Proceedings of the 11th International Conference on Web Engineering, ICWE'11, pages 74–89, Berlin, Heidelberg, 2011. Springer-Verlag. (Cité en pages 112 et 113.)

[Berg 2006] Therese Berg, Bengt Jonsson et Harald Raffelt. *Regular Inference for State Machines with Parameters*. In FASE, LNCS 3922, pages 107–121, Vienna, Austria, 2006. (Cité en page 36.)

[Berg 2008] Therese Berg, Bengt Jonsson et Harald Raffelt. *Regular Inference for State Machines Using Domains with Equality Tests*. In FASE, pages 317–331, 2008. (Cité en page 33.)

[Beyer 2009] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu et Roberto Sebastiani. *Software Model Checking via Large-Block Encoding*. 2009. (Cité en page 25.)

[Biermann 1972] A. W. Biermann et J. A. Feldman. *On the Synthesis of Finite-State Machines from Samples of Their Behavior*. IEEE Transactions on Computers, vol. 21, no. 6, pages 592–597, 1972. (Cité en pages 23 et 53.)

[BinSec 2013] BinSec. *Binary Code Analysis for Security, 2013*. http://binsec.gforge.inria.fr, 2013. (Cité en page i.)

[Bisht 2011] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky et V. N. Venkatakrishnan. *WAPTEC : Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction*. In Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pages 575–586, New York, NY, USA, 2011. ACM. (Cité en page 24.)

[Bossert 2012] Georges Bossert, Frédéric Guihéry et Guillaume Hiet. *Netzob : un outil pour la rétro-conception de protocoles de communication*. In Actes du Symposium sur la sécurité des technologies de l'information et des communications, page 43, Rennes, France, Juin 2012. (Cité en page 152.)

[Bowler 2002] Mike Bowler. *HTMLUnit*. http://htmlunit.sourceforge.net/, 2002. (Cité en pages 34 et 73.)

[Caballero 2009] Juan Caballero, Pongsin Poosankam, Christian Kreibich et Dawn Song. *Dispatcher : Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering*. In Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, pages 621–634, New York, NY, USA, 2009. ACM. (Cité en pages 31 et 32.)

[Chen 2014] Shay Chen. *WAVSEP Web Application Scanner Benchmark 2014*. https://code.google.com/p/wavsep/, 2014. (Cité en page 23.)

[Cho 2010] Chia Yuan Cho, Domagoj Babić, Richard Shin et Dawn Song. *Inference and Analysis of Formal Models of Botnet Command and Control Protocols*. In CCS'10 : Proceedings of the 2010 ACM Conference on Computer and Communications Security, pages 426–440. ACM, 2010. (Cité en pages ix, 31 et 32.)

[Choudhary 2012] Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M. Mirtaheri, Ali Moosavi, Gregor von Bochmann, Guy-Vincent Jourdan et Iosif Viorel Onut. *Crawling Rich Internet Applications : The State of the Art.* In Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '12, pages 146–160, Riverton, NJ, USA, 2012. IBM Corp. (Cité en page 151.)

[Choudhary 2013a] Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M. Mirtaheri, Guy-Vincent Jourdan, Gregor v. Bochmann et Iosif Viorel Onut. *Building Rich Internet Applications Models : Example of a Better Strategy.* In Proceedings of the 13th International Conference on Web Engineering, ICWE'13, pages 291–305, Berlin, Heidelberg, 2013. Springer-Verlag. (Cité en pages 113 et 151.)

[Choudhary 2013b] Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M. Mirtaheri, Guy-Vincent Jourdan, Gregor von Bochmann et Iosif-Viorel Onut. *Building Rich Internet Applications Models : Example of a Better Strategy.* In ICWE, pages 291–305, 2013. (Cité en page 34.)

[Consortium 2008] OASIS Consortium. *Security Assertion Markup Language V2 Technical Overview.* https://wiki.oasis-open.org/security/Saml2TechOverview, 2008. (Cité en pages 39, 46 et 72.)

[Cook 1998] Jonathan E. Cook et Alexander L. Wolf. *Discovering Models of Software Processes from Event-based Data.* ACM Trans. Softw. Eng. Methodol., vol. 7, no. 3, pages 215–249, Juillet 1998. (Cité en page 23.)

[Cui 2007] Weidong Cui, Jayanthkumar Kannan et Helen J. Wang. *Discoverer : Automatic Protocol Reverse Engineering from Network Traces.* In Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, SS'07, pages 14 :1–14 :14, Berkeley, CA, USA, 2007. USENIX Association. (Cité en page 32.)

[Cui 2008] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang et Luis Irun-Briz. *Tupni : Automatic Reverse Engineering of Input Formats.* In Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08, pages 391–402, New York, NY, USA, 2008. ACM. (Cité en page 32.)

[Cymru 2014] Team Cymru. *SOHO Pharming : The Growing Exploitation of Small Office Routers Creating Serious Risk.* https://www.team-cymru.com/ReadingRoom/Whitepapers/2013/TeamCymruSOHOPharming.pdf, February 2014. (Cité en page 123.)

[D2.2.2 2013] SPaCIoS D2.2.2. *Combined whitebox and blackbox model inference.* http://www.spacios.eu/deliverables/spacios-d2.2.2.pdf, 2013. (Cité en pages 4 et 25.)

[D3.2 2011] AVANTSSAR D3.2. *ASLan++ specification and tutorial.* www.avantssar.eu/pdf/deliverables/avantssar-d2-3_update.pdf, 2011. (Cité en pages 3 et 127.)

[Dijkstra 1959] E.W. Dijkstra. *A note on two problems in connexion with graphs.* Numerische Mathematik, vol. 1, no. 1, pages 269–271, 1959. (Cité en page 126.)

[Dingledine 2004] Roger Dingledine, Nick Mathewson et Paul Syverson. *Tor : The Second-generation Onion Router.* In Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association. (Cité en page 31.)

[Doupé 2010] Adam Doupé, Marco Cova et Giovanni Vigna. *Why Johnny Can't Pentest : An Analysis of Black-Box Web Vulnerability Scanners.* In Christian Kreibich et Marko Jahnke, editeurs, Detection of Intrusions and Malware, and Vulnerability Assessment, volume 6201 of *Lecture Notes in Computer Science*, pages 111–131. Springer Berlin Heidelberg, 2010. (Cité en pages 23, 24, 33 et 34.)

[Doupé 2012] Adam Doupé, Ludovico Cavedon, Christopher Kruegel et Giovanni Vigna. *Enemy of the State : A State-aware Black-box Web Vulnerability Scanner.* In Proceedings of the 21st USENIX Conference on Security Symposium, Security'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association. (Cité en pages 33, 83 et 118.)

[Duchene 2012] F. Duchene, R. Groz, S. Rawat et J. Richier. *XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing.* In Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, pages 815–817, April 2012. (Cité en page 151.)

[Duchene 2013] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier et Roland Groz. *LigRE : Reverse-Engineering of Control and Data Flow Models for Black-Box XSS Detection.* In Working Conference in Reverse Engineering (WCRE), pages 252–261, Koblenz-Landau, Germany, Oct 2013. IEEE. (Cité en pages 24 et 34.)

[Duchene 2014] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier et Roland Groz. *KameleonFuzz : Evolutionary Fuzzing for Black-Box XSS Detection.* In CODASPY, San Antonio, Texas, USA, 2014. ACM. (Cité en page 123.)

[Ernst 2001] Michael D. Ernst, Jake Cockrell, William G. Griswold et David Notkin. *Dynamically Discovering Likely Program Invariants to Support Program Evolution.* IEEE Trans. Software Eng., vol. 27, no. 2, pages 99–123, 2001. (Cité en page 45.)

[Eshete 2013] Birhanu Eshete, Adolfo Villafiorita, Komminist Weldemariam et Mohammad Zulkernine. *Confeagle : Automated Analysis of Configuration Vulnerabilities in Web Applications.* In Proceedings of the 2013 IEEE 7th International Conference on Software Security and Reliability, SERE '13, pages 188–197, Washington, DC, USA, 2013. IEEE Computer Society. (Cité en page 4.)

[Feist 2014] Josselin Feist, Laurent Mounier et Marie-Laure Potet. *Statically detecting Use-After-Free on Binary Code*. Journal of Computer Virology and Hacking Techniques, vol. online article, January 2014. (Cité en page 152.)

[Felmetsger 2010] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel et Giovanni Vigna. *Toward Automated Detection of Logic Vulnerabilities in Web Applications*. In Proceedings of the 19th USENIX Conference on Security, USENIX Security'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. (Cité en page 24.)

[Fonseca 2007] J. Fonseca, M. Vieira et H. Madeira. *Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks*. In Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on, pages 365–372, Dec 2007. (Cité en page 23.)

[Foundation² 2007] Free Software Foundation². *GNU General Public License*. http://www.gnu.org/licenses/gpl.html, 2007. (Cité en page 129.)

[Frei 2009] Stefan Frei, Dominik Schatzmann, Bernhard Plattner et Brian Trammell. *Modelling the Security Ecosystem - The Dynamics of (In)Security*. In Workshop on the Economics of Information Security (WEIS), June 2009, Cambridge, UK, Jun 2009. (Cité en page 165.)

[Groz 2008] Roland Groz, Keqin Li, Alexandre Petrenko et Muzammil Shahbaz. *Modular System Verification by Inference, Testing and Reachability Analysis*. In TestCom/FATES, pages 216–233, 2008. (Cité en page 54.)

[Guan 2011] Chun Guan et Xiaoqin Zeng. *An Improved ID3 Based on Weighted Modified Information Gain*. In Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security, CIS '11, pages 1283–1285, Washington, DC, USA, 2011. IEEE Computer Society. (Cité en page 104.)

[Hossen 2011] Karim Hossen, Roland Groz et Jean-Luc Richier. *Security Vulnerabilities Detection Using Model Inference for Applications and Security Protocols*. In Second International Workshop on Security Testing (SECTEST 2011), Workshop of the IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST2011), pages 534–536, Berlin, Germany, mar 2011. IEEE. (Cité en page 150.)

[Hossen 2013] Karim Hossen, Roland Groz, Catherine Oriat et Jean-Luc Richier. *Automatic generation of test drivers for model inference of web applications*. In Fourth International Workshop on Security Testing (SECTEST 2013), Workshop of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST 2013), pages 441–444, Luxembourg, mar 2013. IEEE CS Press. (Cité en page 150.)

[Hossen 2014] Karim Hossen, Catherine Oriat, Roland Groz et Jean-Luc Richier. *Automatic Model Inference of Web Applications for Security Testing*. In Position Statement at SecTest2014, co-located with ICST2014, Cleveland, Ohio, USA, apr 2014. to appear. (Cité en page 149.)

[Howar 2010] Falk Howar, Bernhard Steffen et Maik Merten. *From ZULU to RERS - Lessons Learned in the ZULU Challenge*. In 4th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation, ISoLA 2010, pages 687–704, Heraklion, Grèce, 2010. (Cité en page 31.)

[Howar 2012] Falk Howar, Bernhard Steffen, Bengt Jonsson et Sofia Cassel. *Inferring Canonical Register Automata*. In VMCAI, pages 251–266, 2012. (Cité en page 31.)

[Hungar 2003] Hardi Hungar, Oliver Niese et Bernhard Steffen. *Domain-Specific Optimization in Automata Learning*. In CAV, volume 2725 of *LNCS*, pages 315–327. Springer, 2003. (Cité en pages 6 et 30.)

[Irfan 2010a] M.N. Irfan. *State Machine Inference in Testing Context with Long Counterexamples*. In Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, pages 508–511, April 2010. (Cité en page 31.)

[Irfan 2010b] Muhammad Naeem Irfan, Catherine Oriat et Roland Groz. *Angluin style finite state machine inference with non-optimal counterexamples*. In MIIT, pages 11–19, New York, NY, USA, 2010. ACM. (Cité en pages 31, 45, 62, 63 et 131.)

[Irfan 2013] Muhammad-Naeem Irfan, Catherine Oriat et Roland Groz. *Model Inference and Testing*. Advances in Computers, vol. 89, pages 89–139, 2013. (Cité en page 31.)

[Java.net 2010] Java.net. *JAIN-SIP*. `https://jsip.java.net/`, 2010. (Cité en page 68.)

[Jensen 1995] Tommy R. Jensen et Bjarne Toft. Graph coloring problems. Wiley interscience series in discrete mathematics and optimization. John Wiley and sons, New York, 1995. (Cité en page 34.)

[Kim 2012] Taesoo Kim, Ramesh Chandra et Nickolai Zeldovich. *Efficient Patch-based Auditing for Web Application Vulnerabilities*. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pages 193–206, Berkeley, CA, USA, 2012. USENIX Association. (Cité en page 4.)

[Krol 2012] K. Krol, M. Moroz et M.-A. Sasse. *Don't work. Can't work ? Why it's time to rethink security warnings*. In Risk and Security of Internet and Systems (CRiSIS), 2012 7th International Conference on, pages 1–8, Oct 2012. (Cité en page 124.)

[Lebeau 2013] Franck Lebeau, Bruno Legeard, Fabien Peureux et Alexandre Vernotte. *Model-Based Vulnerability Testing for Web Applications*. In SECTEST'13, 4-th Int. Workshop on Security Testing. In conjunction with ICST'13, 6-th IEEE Int. Conf. on Software Testing, Verification and Validation, pages 445–452, Luxembourg, Luxembourg, Mars 2013. IEEE Computer Society Press. (Cité en page 23.)

[Li 2006a] Keqin Li, R. Groz et M. Shahbaz. *Integration Testing of Components Guided by Incremental State Machine Learning.* In Testing : Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings, pages 59–70, Aug 2006. (Cité en page 30.)

[Li 2006b] Keqin Li, Roland Groz et Muzammil Shahbaz. *Integration Testing of Distributed Components Based on Learning Parameterized I/O Models.* In Elie Najm, Jean-François Pradat-Peyre et VéroniqueViguié Donzeau-Gouge, editeurs, Formal Techniques for Networked and Distributed Systems - FORTE 2006, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450. Springer Berlin Heidelberg, 2006. (Cité en pages 7 et 49.)

[Li 2006c] Keqin Li, Roland Groz et Muzammil Shahbaz. *Integration Testing of Distributed Components Based on Learning Parameterized I/O Models.* In FORTE, pages 436–450, 2006. (Cité en pages 36 et 40.)

[Li 2008] Li Li, Wu Chou et Weiping Guo. *Control Flow Analysis and Coverage Driven Testing for Web Services.* In Proceedings of the 2008 IEEE International Conference on Web Services, ICWS '08, pages 473–480, Washington, DC, USA, 2008. IEEE Computer Society. (Cité en page 151.)

[Li 2014] Xiaowei Li, Xujie Si et Yuan Xue. *Automated Black-box Detection of Access Control Vulnerabilities in Web Applications.* In Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, CODASPY '14, pages 49–60, New York, NY, USA, 2014. ACM. (Cité en page 34.)

[Lorenzoli 2008] Davide Lorenzoli, Leonardo Mariani et Mauro Pezze. *Automatic generation of software behavioral models.* In ICSE, pages 501–510, 2008. (Cité en page 23.)

[MacDonald] Doug MacDonald. *Mega-D botnet analysis.* https://www.fortiguard.com/legacy/analysis/ozdokanalysis.html. (Cité en page 31.)

[Maler 1991] Oded Maler et Amir Pnueli. *On the Learnability of Infinitary Regular Sets.* In Proceedings of the Fourth Annual Workshop on Computational Learning Theory, COLT '91, pages 128–138, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. (Cité en page 31.)

[Mccanne] S. Mccanne, S. Floyd et K. Fall. *ns2 (network simulator 2).* http ://www-nrg.ee.lbl.gov/ns/. (Cité en page 143.)

[Meinke 2010] Karl Meinke et Fei Niu. *A Learning-Based Approach to Unit Testing of Numerical Software.* In Alexandre Petrenko, Adenilso Simão et JoséCarlos Maldonado, editeurs, Testing Software and Systems, volume 6435 of *Lecture Notes in Computer Science*, pages 221–235. Springer Berlin Heidelberg, 2010. (Cité en page 23.)

[Mesbah 2008] Ali Mesbah, Engin Bozdag et Arie van Deursen. *Crawling AJAX by Inferring User Interface State Changes.* In Proceedings of the 2008 Eighth International Conference on Web Engineering, ICWE '08, pages 122–134,

Washington, DC, USA, 2008. IEEE Computer Society. (Cité en pages 34 et 112.)

[Mesbah 2012] Ali Mesbah, Arie van Deursen et Stefan Lenselink. *Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes*. ACM Trans. Web, vol. 6, no. 1, pages 3 :1–3 :30, Mars 2012. (Cité en page 151.)

[Mihancea 2014] Petru Florin Mihancea et Marius Minea. *JMODEX : Model extraction for verifying security properties of web applications*. In CSMR-WCRE, pages 450–453, 2014. (Cité en page 24.)

[Mirkovic 2004] Jelena Mirkovic et Peter Reiher. *A Taxonomy of DDoS Attack and DDoS Defense Mechanisms*. SIGCOMM Comput. Commun. Rev., vol. 34, no. 2, pages 39–53, Avril 2004. (Cité en page 17.)

[Moore 1956] Edward F. Moore. *Gedanken-Experiments on Sequential Machines*. In Claude Shannon et John McCarthy, editeurs, Automata Studies, pages 129–153. Princeton University Press, Princeton, NJ, 1956. (Cité en page 55.)

[Nerode 1958] A. Nerode. *Linear automata transformation*. In American Mathematical Society, volume 9, pages 541–544, 1958. (Cité en page 53.)

[NESSoS 2010] NESSoS. *NESSoS*. http://www.nessos-project.eu, 2010. (Cité en page 133.)

[Netcraft 2014] Netcraft. *April 2014 Web Server Survey*. http://news.netcraft.com/archives/category/web-server-survey/, 2014. (Cité en pages 1 et 11.)

[Niese 2003] Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, 2003. (Cité en pages 23, 30 et 36.)

[NIST 2002] NIST. *FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2*. Rapport technique, DEPARTMENT OF COMMERCE, 2002. (Cité en page 104.)

[Oheimb 2012] David Oheimb et Sebastian Mödersheim. *ASLan++ — A Formal Security Specification Language for Distributed Systems*. In BernhardK. Aichernig, FrankS. Boer et MarcelloM. Bonsangue, editeurs, Formal Methods for Components and Objects, volume 6957 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2012. (Cité en pages 3 et 127.)

[OWASP 2010] OWASP. *WebGoat*. http://code.google.com/p/webgoat/, 2010. (Cité en page 135.)

[OWASP 2013] OWASP. *OWASP Top 10 - 2013*. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2013. (Cité en pages 5, 17 et 20.)

[OWASP 2014a] OWASP. *About OWASP*. https://www.owasp.org/index.php/About_OWASP, 2014. (Cité en page 4.)

[OWASP 2014b] OWASP. *Vulnerability Scanning Tools*. https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools, 2014. (Cité en page 23.)

[Pellegrino 2014] Giancarlo Pellegrino et Davide Balzarotti. *Toward Black-Box Detection of Logic Flaws in Web Applications*. In Network and Distributed System Security (NDSS) Symposium, NDSS 14, February 2014. (Cité en pages 24, 25 et 27.)

[Petrenko 2004a] A. Petrenko, S. Boroday et R. Groz. *Confirming configurations in EFSM testing*. Software Engineering, IEEE Transactions on, vol. 30, no. 1, pages 29–42, Jan 2004. (Cité en page 7.)

[Petrenko 2004b] Alexandre Petrenko, Sergiy Boroday et Roland Groz. *Confirming Configurations in EFSM Testing*. IEEE Trans. Software Eng., vol. 30, no. 1, pages 29–42, 2004. (Cité en page 37.)

[Petrenko 2014] Alexandre Petrenko, Keqin Li, Roland Groz, Karim Hossen et Catherine Oriat. *Inferring Approximated Models for Systems Engineering*. In 15th IEEE International Symposium on High Assurance Systems Engineering (HASE 2014), pages 249–253, Miami, Florida, USA, jan 2014. (Cité en pages 54, 60 et 148.)

[Pnueli 1977] Amir Pnueli. *The temporal logic of programs*. In Foundations of Computer Science, 1977., 18th Annual Symposium on, pages 46–57, Oct 1977. (Cité en page 127.)

[Quinlan 1986] J. R. Quinlan. *Induction of Decision Trees*. Mach. Learn, pages 81–106, 1986. (Cité en page 103.)

[Quinlan 1993] J. Ross Quinlan. C4.5 : programs for machine learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. (Cité en pages 103 et 107.)

[Raffelt 2009] Harald Raffelt, Maik Merten, Bernhard Steffen et Tiziana Margaria. *Dynamic testing via automata learning*. International Journal on Software Tools for Technology Transfer, vol. 11, no. 4, pages 307–324, 2009. (Cité en page 23.)

[Ramalingom 2003] T Ramalingom, K Thulasiraman et A Das. *Context Independent Unique State Identification Sequences for Testing Communication Protocols Modelled As Extended Finite State Machines*. Comput. Commun., vol. 26, no. 14, pages 1622–1633, Septembre 2003. (Cité en page 7.)

[Rivest 1993] Ronald L. Rivest et Robert E. Schapire. *Inference of Finite Automata Using Homing Sequences*. In Machine Learning : From Theory to Applications, pages 51–73, 1993. (Cité en pages 31 et 62.)

[Roest 2010] Danny Roest, Ali Mesbah et Arie van Deursen. *Regression Testing Ajax Applications : Coping with Dynamism*. In Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10, pages 127–136, Washington, DC, USA, 2010. IEEE Computer Society. (Cité en page 151.)

[Rosenberg 2002] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley et E. Schooler. *SIP : Session Initiation Protocol*, 2002. (Cité en page 143.)

[Rueda Cebollero 2013] Guillem Rueda Cebollero. Learning cache replacement policies using register automata. Master's thesis, Uppsala University, Department of Information Technology, 2013. (Cité en page 152.)

[Sanou 2013] Brahima Sanou. *ICTFacts and Figures.* http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013-e.pdf, 2013. (Cité en page 1.)

[Schur 2013] Matthias Schur, Andreas Roth et Andreas Zeller. *Mining Behavior Models from Enterprise Web Applications.* In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 422–432, New York, NY, USA, 2013. ACM. (Cité en page 151.)

[Shahbaz 2008] Muzammil Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing.* Phd thesis, Institut Polytechnique de Grenoble, 2008. (Cité en page 23.)

[Shahbaz 2009] Muzammil Shahbaz et Roland Groz. *Inferring Mealy Machines.* In FM, LNCS 5850, pages 207–222, Eindhoven, The Netherlands, 2009. (Cité en pages 30, 31, 36, 63 et 131.)

[Shu 2007] Guoqiang Shu et David Lee. *Testing Security Properties of Protocol Implementations - a Machine Learning Based Approach.* In ICDCS, Toronto, Ontario, Canada, 2007. (Cité en pages 23 et 30.)

[Snake 2014] R. Snake. *XSS (cross site scripting) cheat sheet.* https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2014. (Cité en page 122.)

[SPaCIoS 2010] SPaCIoS. *SPaCIoS project : Secure Provision and Consumption in the Internet of Services.* www.spacios.eu, 2010. (Cité en page 127.)

[Steffen 2011] Bernhard Steffen, Falk Howar et Maik Merten. *Introduction to Active Automata Learning from a Practical Perspective.* In Marco Bernardo et Valérie Issarny, editeurs, SFM, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011. (Cité en pages 62, 63 et 131.)

[Suto 2007] Larry Suto. *Analyzing the Effectiveness and Coverage of Web Application Security Scanners.* http://ha.ckers.org/files/CoverageOfWebAppScanners.zip, 2007. (Cité en page 23.)

[Suto 2010] Larry Suto. *Analyzing the accuracy and time costs of web application security scanners*, 2010. (Cité en page 23.)

[Syverson 1994] Paul Syverson. *A Taxonomy of Replay Attacks.* In In Proceedings of the 7th IEEE Computer Security Foundations Workshop, pages 187–191. Society Press, 1994. (Cité en page 36.)

[Trakhtenbrot 1973] B. A. Trakhtenbrot et Y. M. Barzdin. Finite automata, behavior and synthesis. North-Holland Pub, 1973. (Cité en page 55.)

[UNINETT 2007] UNINETT. *SimpleSAMLphp.* http://simplesamlphp.org/, 2007. (Cité en page 72.)

[Vasilevskii 1973] M.P. Vasilevskii. *Failure diagnosis of automata.* Cybernetics, vol. 9, no. 4, pages 653–665, 1973. (Cité en pages 50 et 52.)

[Vigano 2013] Luca Vigano. *The SPaCIoS Project : Secure Provision and Consumption in the Internet of Services.* 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, vol. 0, pages 497–498, 2013. (Cité en page 2.)

[Wang 1997] Y. Wang et I. H. Witten. *Induction of model trees for predicting continuous classes.* In Poster papers of the 9th European Conference on Machine Learning. Springer, 1997. (Cité en page 107.)

[Wang 2011] Rui Wang, Shuo Chen, XiaoFeng Wang et Shaz Qadeer. *How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores.* In Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11, pages 465–480, Washington, DC, USA, 2011. IEEE Computer Society. (Cité en page 27.)

[Wang 2012] Rui Wang, Shuo Chen et XiaoFeng Wang. *Signing Me Onto Your Accounts Through Facebook and Google : A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services.* In Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, pages 365–379, Washington, DC, USA, 2012. IEEE Computer Society. (Cité en page 27.)

[Witten 2011] I. H. Witten, E. Frank et M. A. Hall. Data mining : Practical machine learning tools and techniques (third edition). Morgan Kaufmann, 2011. (Cité en page 45.)

[Zeller 2008] William Zeller et Edward W. Felten. *Cross-Site Request Forgeries : Exploitation and Prevention.* https://www.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf, 2008. (Cité en page 22.)

[Zenner 2009] Erik Zenner. *Nonce Generators and the Nonce Reset Problem.* In ISC, pages 411–426, 2009. (Cité en page 36.)

[Zhang 2012] Xuesong Zhang et Honglei Wang. *AJAX Crawling Scheme Based on Document Object Model.* In Proceedings of the 2012 Fourth International Conference on Computational and Information Sciences, ICCIS '12, pages 1198–1201, Washington, DC, USA, 2012. IEEE Computer Society. (Cité en page 151.)

Annexe A

Vulnérabilités XSS découvertes

Dans ce chapitre d'annexe, nous présentons diverses vulnérabilités découvertes sur des sites Web existants et ayant un nombre important d'utilisateurs. Pour chaque vulnérabilité, la date de la découverte est indiquée et les personnes en charge du site ont été averties en respectant le modèle *responsible disclosure* [Frei 2009].

Les vulnérabilités suivantes ont été trouvées en appliquant l'algorithme du chapitre 5 pour les applications Web puis la méthode de détection des XSS de la section 6.2. Pour montrer que cette méthode a été automatisée, nous avons gardé uniquement les vulnérabilités trouvées sans information fournies de la part de l'utilisateur (sauf le point d'entrée de l'application). Nous nous concentrons donc sur les parties publiques et non celles nécessitant de l'authentification.

Pour chaque site, un modèle partiel d'EFSM (uniquement les parties accessibles depuis le point d'entrée) du site a été inféré. Comme l'EFSM possède des fonctions de sortie, nous avons pu détecter qu'un ou plusieurs paramètres d'une requête ont été réfléchis dans une sortie. Un chemin est ensuite calculé en partant de l'état initial de l'automate, passant par la requête contenant le paramètre réfléchi et terminant par la réflexion de ce paramètre. Afin d'éliminer les éventuels faux positifs, des tests sont effectués avec plusieurs chaînes de caractères générées aléatoirement (lettres et chiffres). Si la réflexion est confirmée, une série de scripts XSS est testée sur ce paramètre pour confirmer la vulnérabilité.

A.1 Roland-Garros

La première page du site contient 119 liens et un seul formulaire, celui pour la recherche. Une fois le formulaire parcouru, la sortie obtenue contient une sortie déterministe qui reprend la chaîne de caractère recherchée pour afficher le message *il n'y a pas de résultat pour 'chaîne'. Veuillez effectuer une nouvelle recherche.*.

Le formulaire de recherche du site rollandgarros.com ne filtre pas les entrées des utilisateurs. La réflexion est faite dans un élément de type DIV donc n'importe quel code avec les balises SCRIPT peut être injecté comme dans la figure A.1.

La page contient notamment des liens vers la boutique du site. Un attaquant pourrait modifier ces liens afin de tromper l'utilisateur et récupérer ses informations.

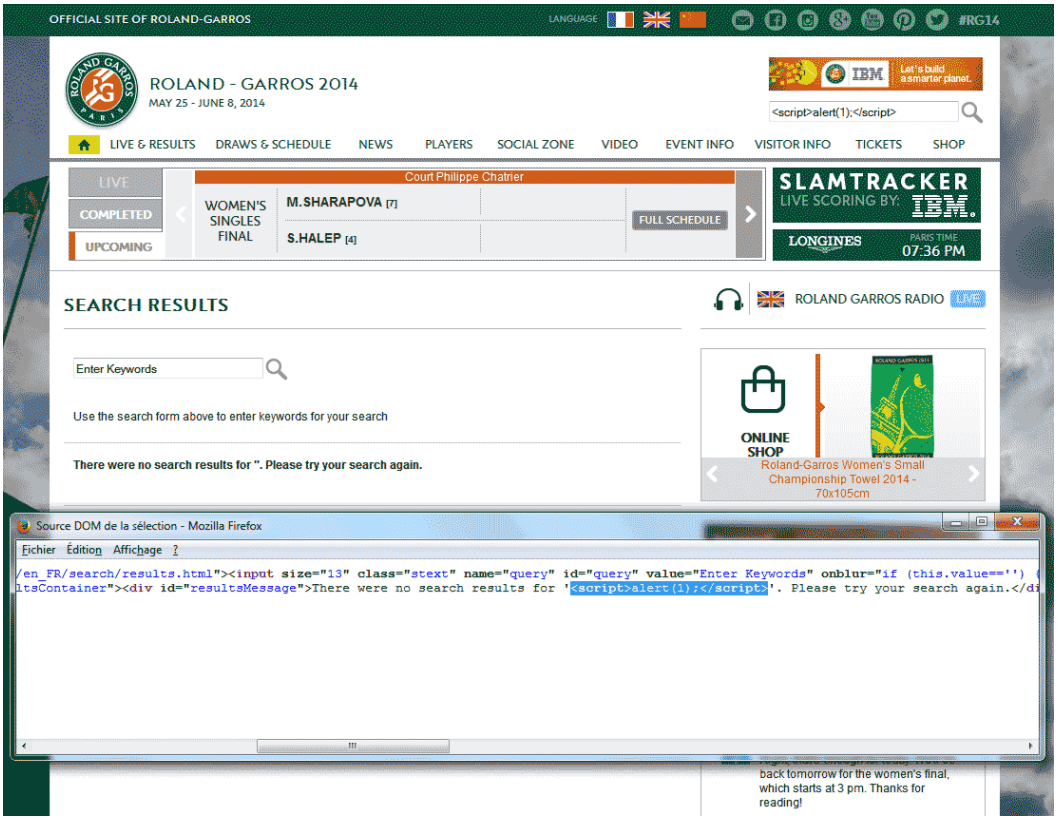

FIGURE A.1 – XSS sur rolandgarros.com

A.2 Ensimag

La première page contient 168 liens et 2 formulaires qui sont identiques. Ce formulaire de recherche possède 12 paramètres : 11 cachés et 1 paramètre pour les mots clés recherchés. Chaque paramètre caché sert à restreindre la recherche à une certaine langue ou à une certaine partie du site. Une fois le formulaire envoyé, la page de sortie contient 7 réflexions de la chaîne recherchée. Seule une réflexion est exploitable directement puisqu'elle est située dans une balise de type H1 dans laquelle on peut directement insérer une balise SCRIPT. Trois réflexions, dont une dans le titre, sont correctement filtrées en empêchant d'ouvrir une nouvelle balise. Les trois dernières réflexions sont aussi exploitables, mais nécessitent de fermer la balise contenant la réflexion au préalable.

Le formulaire de recherche du site ensimag.grenoble-inp.fr ne filtre pas les entrées des utilisateurs. La réflexion est faite dans un élément de type H1 donc n'importe quel code avec la balise SCRIPT peut être injecté comme illustré dans la figure A.2.

La page contient notamment des liens vers les e-services et le formulaire de connexion de Grenoble INP. Un attaquant pourrait modifier ces liens tromper l'uti-

lisateur et récupérer ses informations.

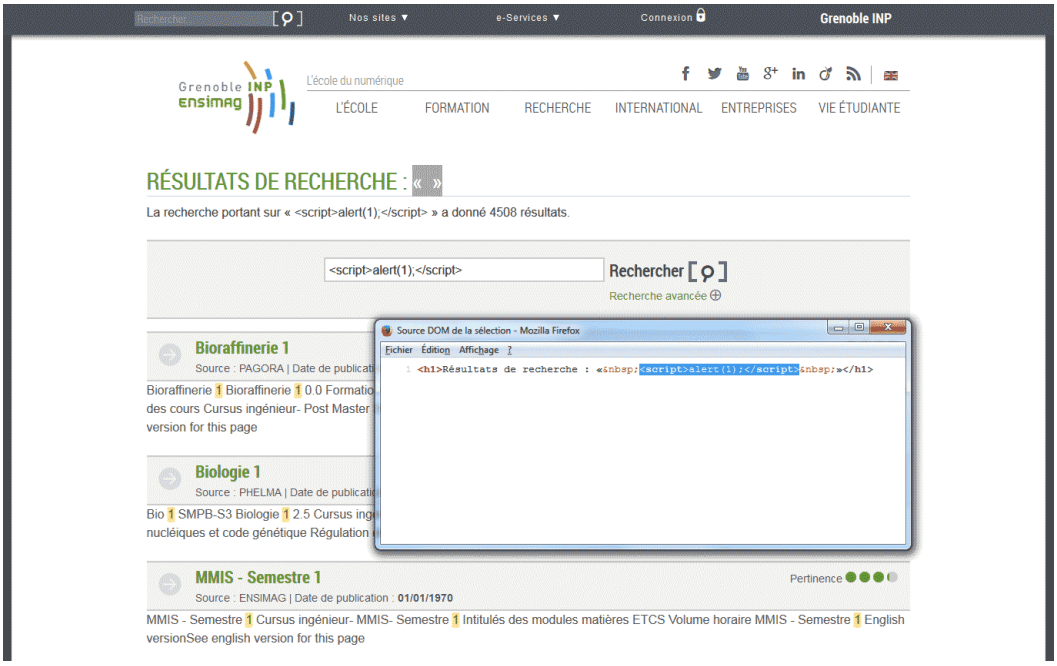

FIGURE A.2 – XSS sur le site de l'Ensimag

A.3 adopteunmec.com

Cette vulnérabilité diffère des précédentes par la façon dont elle est exploitée. La liste de script JavaScript utilisée pour tester si les réflexions sont bien des vulnérabilités n'était pas suffisante. Elle contenait des scripts pour les événements, mais aucun qui permettait une exécution automatique du code au chargement de la page. Nous avons donc ajouté un script à cette liste.

Le site de rencontres www.adopteunmec.com conserve les données envoyées quand un utilisateur effectue une recherche sur le site. Une fois la recherche effectuée, les valeurs des champs du formulaire sont réutilisées dans les attributs *value*.

Fermer la balise INPUT correspondant au champ *pseudo* pour ensuite utiliser une balise SCRIPT ne fonctionne pas, mais nous pouvons utiliser un événement comme *onfocus* qui déclenche un script quand l'élément a le focus. En plus, nous pouvons utiliser le mot clé *autofocus* pour déclencher cet événement après le chargement de la page comme dans la figure A.3.

On peut voir que les caractères des parenthèses ouvrantes et fermantes sont encodés différemment, mais le script est quand même exécuté (testé sur Firefox

30).

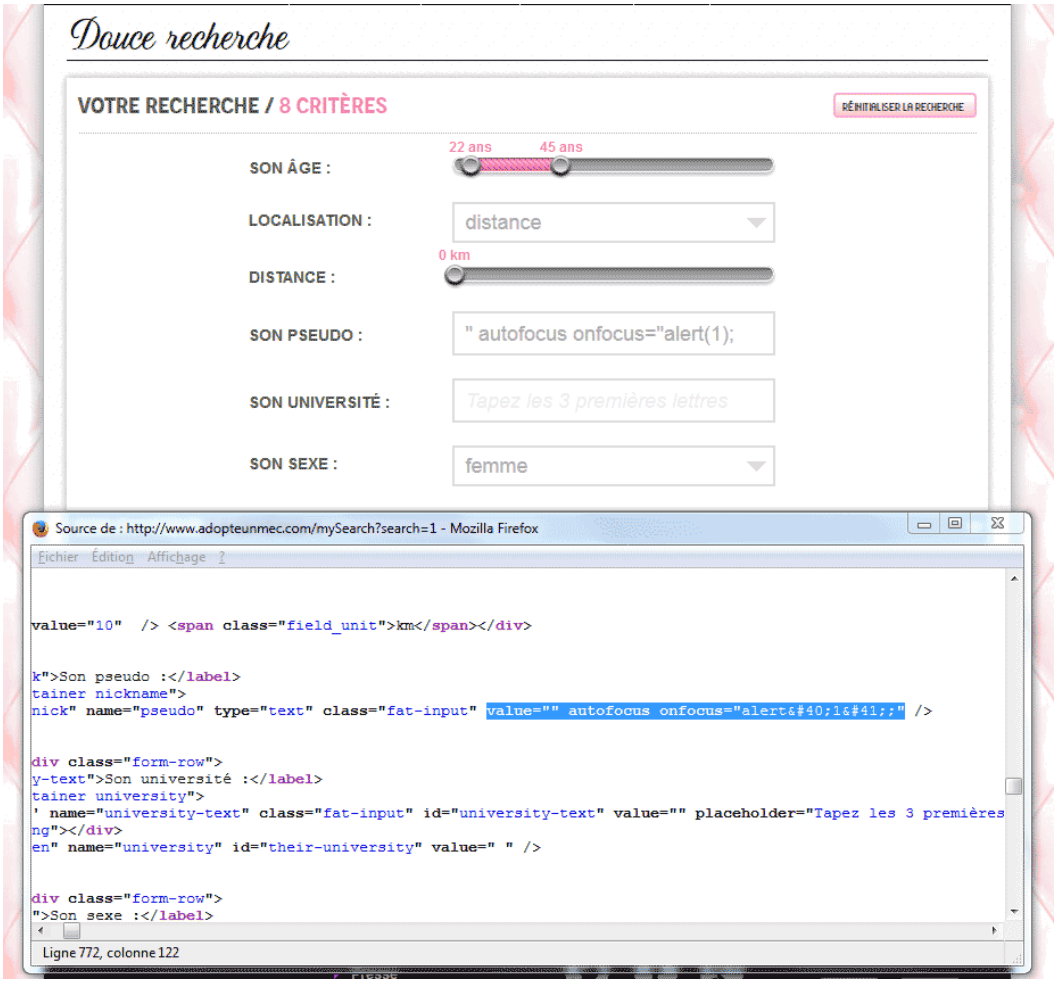

FIGURE A.3 – XSS sur le site adopteunmec.com

A.4 IAE Grenoble

Le site de l'IAE de Grenoble www.iae-grenoble.fr utilise le paramètre GET nommé *candidat* pour connaître l'origine des utilisateurs voulant se connecter au site de gestion des préinscriptions en Master. Sur la page de connexion, le paramètre *candidat* est réfléchi en tant que champ caché du formulaire comme on peut le voir sur la figure A.4.

Comme la valeur est réfléchie dans un attribut d'une balise INPUT, il faut avant tout fermer cette balise et utiliser ensuite la balise SCRIPT.

Un attaquant pourrait modifier le formulaire et récupérer les informations de

connexion.

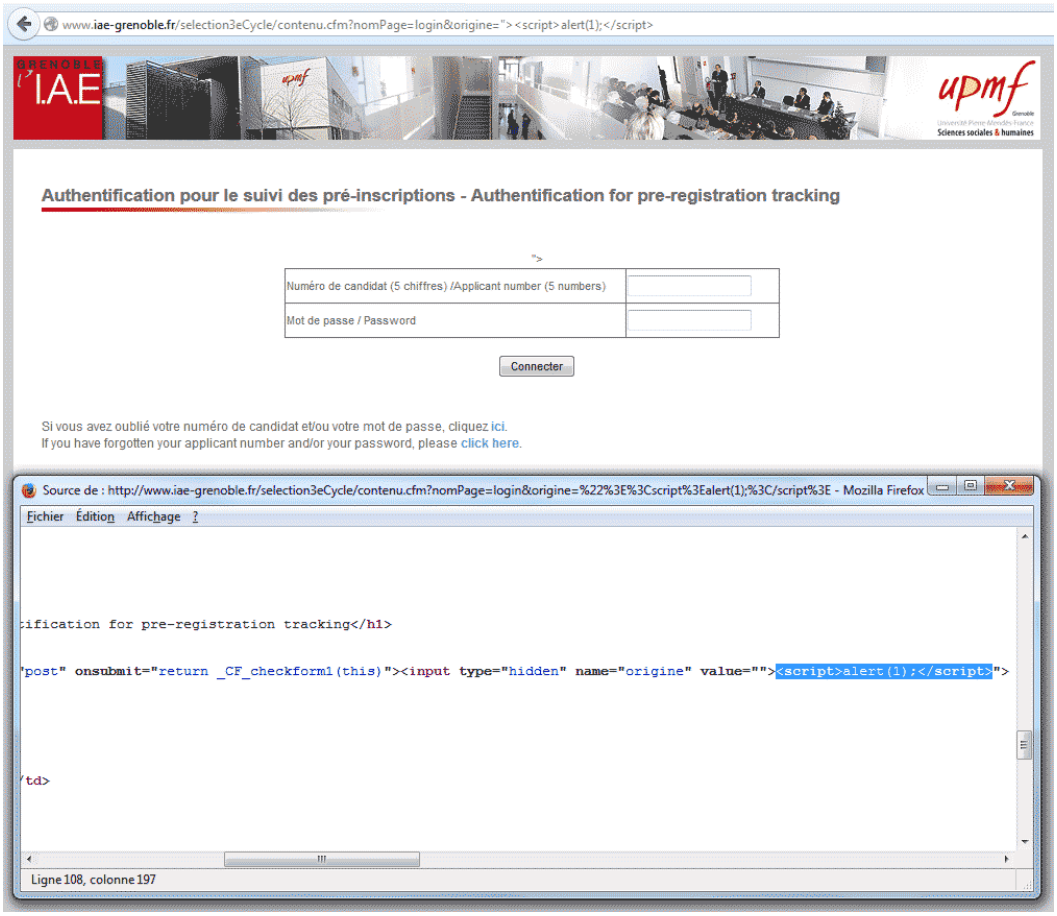

FIGURE A.4 – XSS sur le site de l'IAE Grenoble

A.5 QueChoisir.org

Le site quechoisir.org ne filtre pas les données provenant de la fonction recherche du site. Comme la valeur est réfléchie dans un attribut d'une balise INPUT, il faut avant tout fermer cette balise et utiliser ensuite la balise SCRIPT comme dans la figure A.5.

A.6 Direct-assurance

Les données de la fonction recherche du site de l'assurance direct-assurance.fr sont réfléchies dans un script JavaScript à la fin de la page. Ces données n'étant pas

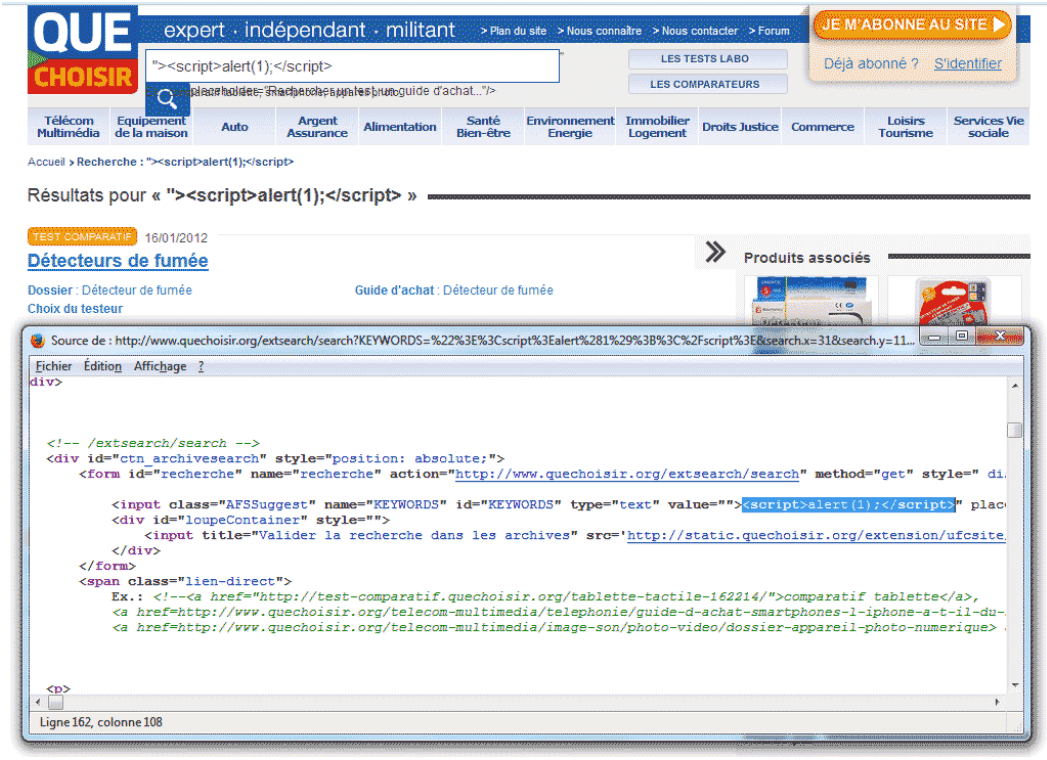

FIGURE A.5 – XSS sur le site de QueChoisir

filtrées correctement, il est possible de fermer le commentaire HTML contenant le script, puis d'ouvrir une nouvelle balise SCRIPT qui permet d'exécuter le code de notre choix comme illustré dans la figure A.6.

A.7 Centerblog.net

Le site centerblog.net ne filtre pas correctement les données provenant du formulaire de recherche. Il est alors possible d'injecter du code via une balise SCRIPT comme dans la figure A.7.

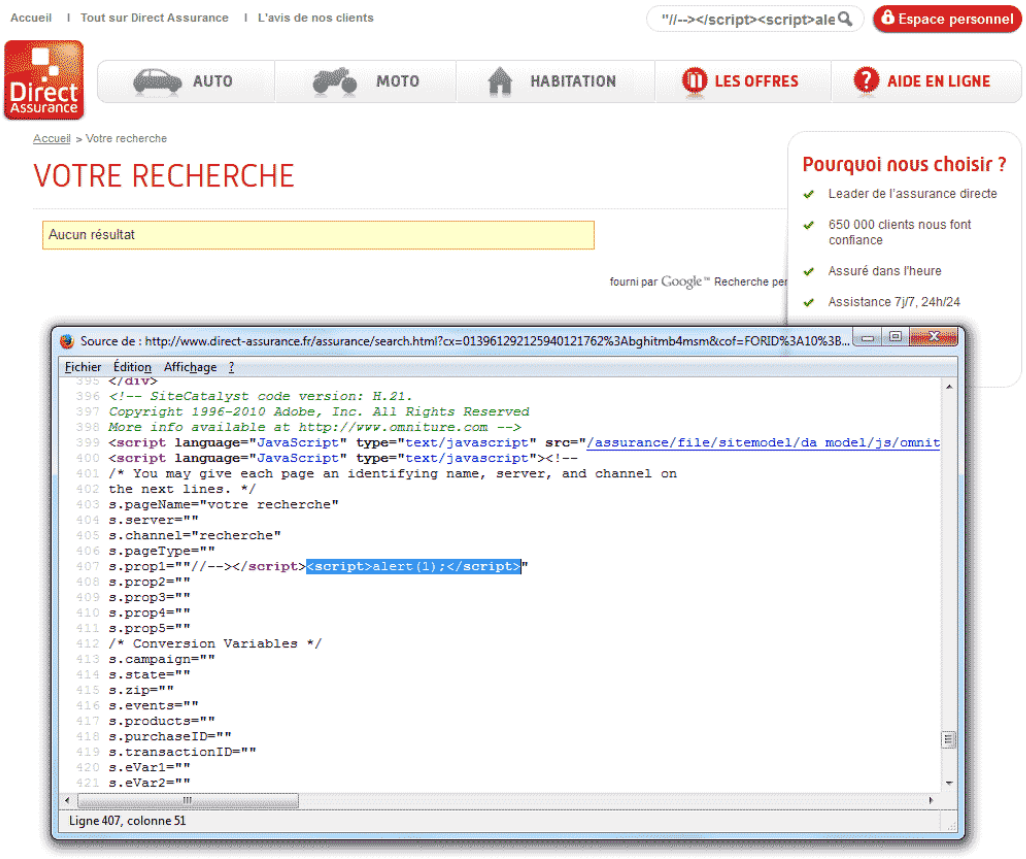

FIGURE A.6 – XSS sur le site de direct-assurance

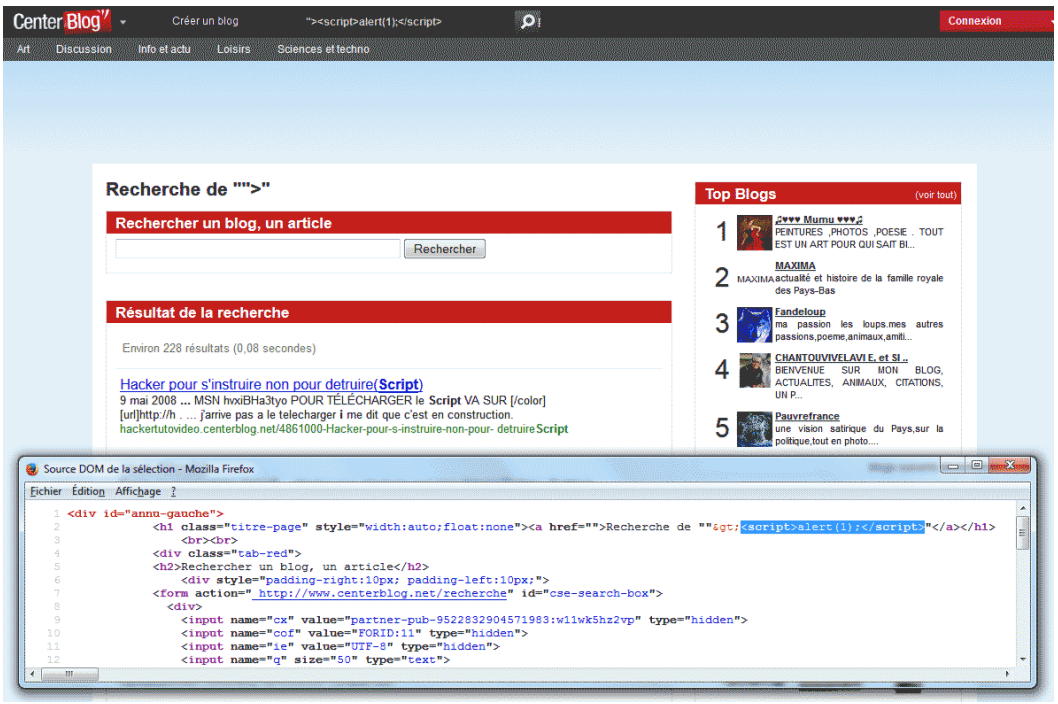

FIGURE A.7 – XSS sur le site de Centerblog.net