Analyse statique de code : Réduction des fausses alertes par apprentissage machine

par

Nathan HUBERT

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE DES TECHNOLOGIES DE
L'INFORMATION
M. Sc. A.

MONTRÉAL, LE 30 JUILLET 2018

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M. Jean-Marc Robert, directeur de mémoire
Département de génie logiciel et des TI à l'École de technologie supérieure

M. Patrick Cardinal, président du jury
Département de génie logiciel et des TI à l'École de technologie supérieure

M. Sègla Jean-Luc Kpodjedo, membre du jury
Département de génie logiciel et des TI à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 26 JUILLET 2018

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je souhaite exprimer toute ma gratitude envers les personnes et les organismes qui ont contribué à la réalisation de ce mémoire.

Je tiens en particulier à remercier mon directeur de recherche, M. Jean-Marc Robert, directeur du département de génie logiciel et des TI et professeur à l'ÉTS, pour m'avoir offert l'opportunité de mener à bien ce projet, ainsi que pour le partage de son expertise et de son temps.

J'exprime ma reconnaissance envers l'organisme MITACS et l'entreprise GoSecure pour leur confiance et leur soutien financier. Je remercie tout particulièrement Philippe Arteau, chercheur en cyber sécurité chez GoSecure, pour son aide précieuse, ses conseils avisés, et l'incroyable quantité de connaissances qu'il a pu m'apporter, et plus généralement toute l'équipe de recherche et développement à GoSecure, pour leur accueil bienveillant dans un environnement de travail chaleureux.

Enfin, je tiens à remercier mes parents pour leurs encouragements et leur soutien infaillible depuis l'autre côté de l'Atlantique, Marion et Valentin pour nos moments d'évasion dans cette merveilleuse place de cuisine japonaise, et Taïna, pour tout le reste.

ANALYSE STATIQUE DE CODE : RÉDUCTION DES FAUSSES ALERTES PAR APPRENTISSAGE MACHINE

Nathan HUBERT

RÉSUMÉ

Les vulnérabilités de sécurité présentes dans les applications Web sont extrêmement dangereuses, à l'heure où n'importe quel attaquant ayant un minimum de connaissances et d'outils peut exploiter avec succès l'une d'entre elles. Les conséquences d'un tel exploit peuvent être très graves : vol de données sensibles, déni de service du site, etc. Il est donc fondamental pour un développeur de détecter ces failles avant la mise en production. Pour ce faire, de nombreuses méthodes existent, l'analyse statique étant probablement la plus utilisée. Le principe est de parcourir le code sans l'exécuter, afin de trouver des modèles caractéristiques de vulnérabilités : cette méthode, en plus d'être très peu coûteuse en temps et ressources, est très efficace. Cependant, elle génère en général de nombreuses fausses alertes, appelées faux positifs : il en résulte que le développeur, recevant le rapport d'analyse, doit faire manuellement le tri entre ces alertes, ce qui peut être extrêmement coûteux en temps et donc mener à des erreurs.

Ce mémoire a pour but de pallier ce problème en ajoutant la notion d'apprentissage machine : automatiser le tri des vulnérabilités permettrait au développeur de gagner un temps considérable, tout en limitant le risque d'erreurs. Notre étude est basée ici sur l'outil Find Security Bugs, un analyseur statique de code Java. Dans un premier temps, nous avons sélectionné des caractéristiques représentant avec précision chaque alerte, afin que l'algorithme d'apprentissage automatique puisse différencier une vraie vulnérabilité d'un faux positif. Par la suite, à partir du rapport d'alertes de l'outil, ainsi que d'une représentation graphique des variables en jeu, nous avons pu extraire ces caractéristiques pour chaque vulnérabilité potentielle, et entraîner différents algorithmes afin d'éliminer le maximum de fausses alertes.

Notre solution a été testée sur la suite Juliet, contenant de nombreuses vulnérabilités déjà étiquetées. Les résultats ont été très satisfaisants : plus de 85% des fausses alertes ont pu être détectées par l'apprentissage machine. Nous avons également mis en évidence qu'il n'existe pas réellement d'algorithme plus performant que les autres dans notre contexte, mais que ceux-ci peuvent être utilisés conjointement pour optimiser les résultats. Enfin, notre solution est flexible, et s'intègre parfaitement avec Find Security Bugs, sans ajout de complexité : le développeur est libre de l'utiliser comme il l'entend.

Mots clés : Vulnérabilité de sécurité, Analyse statique de code, Apprentissage machine, Faux positifs

STATIC CODE ANALYSIS: REDUCTION OF FALSE ALERTS USING MACHINE LEARNING

Nathan HUBERT

ABSTRACT

Security vulnerabilities in web applications are extremely dangerous, especially nowadays, when anyone with a bit of knowledge and the right tools can successfully exploit them. This can have bad consequences: theft of sensitive data, denial of service, etc. Therefore, it is crucial for a developer to detect these vulnerabilities before committing changes. Among the methods used to do that, static code analysis is the most common. The concept is to analyze the code without executing it, in order to find specific vulnerability patterns: overall, this process is very effective and efficient. However, this method generates a lot of false reports, called false positives: the developer has no other choice but to manually sort these alerts, which can be very time-consuming, and thus lead to errors.

The goal here is to overcome this problem by using machine learning technic: by automating the sorting of reports, the developer will save a considerable amount of time, and the risk of errors will be reduced. Our study is based on Find Security Bugs, a static analyzer for Java code. First, we selected some features that represent precisely each alert, so that the machine learning algorithm can make a difference between true vulnerabilities and false positives. Then, we extracted those features from the analyzer's reports and a graphic representation of the code, and we trained different algorithms to eliminate false positives.

We tested our solution on the Juliet test suite, which contains a lot of already labelled vulnerabilities. The results were good: more than 85% of false reports were detected by the machine learning algorithms. We also showed that there is not an optimal algorithm in our context, but it is possible to combine some of them to optimize the results. Finally, our solution is flexible, and integrates seamlessly with Find Security Bugs, without additional complexity: the developer is free to use it the way he wants.

Keywords: Security Vulnerability, Static Code Analysis, Machine Learning, False Positives

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative
FSB	Find Security Bugs
SAT	Static Analysis Tool
API	Application Programming Interface
IDE	Integrated Development Environment

INTRODUCTION

Coder une application Web n'est pas une chose facile. Le développement, bien que systématique, peut très vite devenir un casse-tête, à mesure que l'application grandit. En effet, des milliers de lignes de code peuvent être écrites, et vérifier chacune d'entre elles devient une tâche ardue et fastidieuse, d'autant plus si le développeur est seul. Dans la majorité des cas, pendant la relecture et vérification du code, celui-ci va corriger les erreurs qui font que l'application ne fonctionne pas, ou seulement partiellement. Néanmoins, un problème encore plus important est bien souvent laissé de côté : les vulnérabilités de sécurité. Celles-ci n'ont en effet pas d'impact direct sur le comportement de l'application, mais ouvrent une porte à un attaquant voulant réaliser une tâche malicieuse. Exploiter une vulnérabilité peut par exemple entraîner un déni de service d'un site Web, causant une perte financière pour l'entreprise en question, ou encore permettre à l'attaquant de subtiliser de nombreuses données sensibles, dont il pourra lui-même tirer profit.

De nos jours, l'Internet s'étant démocratisé, et les moyens technologiques le permettant, on dénombre de plus en plus d'attaques, car n'importe quelle personne avec un minimum de connaissance et de matériel peut exploiter avec succès une application Web contenant des failles. Par ailleurs, de nombreux logiciels de source libre facilitent le travail de l'attaquant : ces logiciels, créés à la base pour qu'un développeur teste sa propre application (afin d'y détecter les vulnérabilités), sont finalement utilisés sur des applications par des utilisateurs malveillants désireux d'y réaliser une attaque.

Certaines solutions automatiques existent pour détecter ces vulnérabilités dans le code. La plupart du temps, celles-ci utilisent de l'analyse statique, à savoir parcourent le code pour y détecter des problèmes sans l'exécuter (au contraire de l'analyse dynamique). Bien que ces solutions soient efficaces, et permettent de réduire drastiquement le nombre de failles dans une application, elles sont loin d'être parfaites, autant dans les résultats que dans leur utilisation. Il apparaît en effet qu'elles sont relativement peu utilisées (B. Johnson, Song, Murphy-Hill, & Bowdidge, 2013). Cela semble être principalement dû au fait que la plupart des développeurs

ne sont pas formés à la sécurité, et n'ont surtout pas d'obligation : l'utilisation d'un outil d'analyse statique apporte donc une charge de travail supplémentaire. En effet, beaucoup de développeurs se plaignent du nombre de faux positifs qui ressortent des analyses, et qui doivent être triés individuellement à la main. Lorsque l'on parle de faux positifs dans le cadre de la détection de vulnérabilités, on fait référence à des cas bénins identifiés comme étant malveillants (par exemple un morceau de code jugé dangereux alors qu'il ne l'est pas en réalité). Par ailleurs, Baca, Petersen, Carlsson, et Lundberg (2009) ont montré que les années d'expérience d'un développeur influaient énormément sur sa capacité à utiliser efficacement un outil d'analyse statique, à savoir trier correctement les alertes renvoyées par ce dernier. Une véritable vulnérabilité cachée au milieu de nombreux faux positifs aura donc de fortes chances de passer inaperçue pour un développeur moyen ou devant gérer de grands projets (milliers de lignes de code).

Ce mémoire a pour objectif de proposer une amélioration d'un de ces outils d'analyse statiques, Find Security Bugs (Arteau, 2012), en y introduisant la notion d'apprentissage machine. L'idée derrière ce concept est d'entraîner la machine sur un jeu de données déjà labélisées, pour lui permettre par la suite de classifier automatiquement des données non labélisées. Chaque donnée, appelée instance, possède des attributs : ces attributs (ou caractéristiques) sont communs à toutes les instances, mais prennent évidemment des valeurs différentes pour chacune. C'est sur ces attributs que la machine va effectuer ses calculs (différemment suivant l'algorithme utilisé), afin de faire le lien entre les attributs d'une instance et sa classe : le but étant par la suite de classifier correctement une nouvelle instance grâce à ses attributs. Un algorithme d'apprentissage machine réalisant cette tâche est appelé un classifieur. Son utilisation devrait avoir pour conséquence de rendre le classement des alertes plus performant, de limiter le nombre de faux positifs, et donc de grandement faciliter la tâche du développeur.

L'approche considérée dans ce travail est la suivante. Tout d'abord, en partant du code source d'une application et d'un outil d'analyse statique, deux actions vont être réalisées. La première est de générer le rapport d'erreurs de l'outil sur le code en question, et de labéliser celui-ci : chaque alerte va être triée manuellement en deux catégories, bogue réel ou faux positif, afin de

construire un jeu de données de confiance, ou « ground truth ». La deuxième est d'identifier pour chaque alerte le code concerné, à savoir tous les éléments qui poussent l'outil d'analyse statique à générer une alerte (méthode, classe, variables, etc.). À partir de cela, des caractéristiques représentatives de vulnérabilités vont être extraites, et servir à l'algorithme d'apprentissage machine. En effet, celui-ci va s'entraîner sur les données déjà labélisées, en se basant sur les caractéristiques extraites pour chaque alerte : l'objectif étant ensuite de pouvoir distinguer les vraies alertes des faux positifs, afin de mettre ces derniers de côté pour soulager le travail du développeur. Enfin, une évaluation de différents algorithmes d'apprentissage machine va être effectuée, afin d'aboutir à des résultats les plus performants possible.

Le mémoire est structuré comme suit. Le chapitre 1 présente une revue de la littérature dans le domaine de l'analyse statique et de l'apprentissage machine, puis le chapitre 2 explique le contexte dans lequel est effectuée l'approche décrite ci-dessus, notamment en présentant les principaux outils utilisés. Le chapitre 3 commence par une description détaillée de l'approche utilisée, et présente la solution retenue ainsi que son développement, et enfin le chapitre 4 précède la conclusion en analysant les résultats, et en discutant sur les limites de cette solution.

CHAPITRE 1

REVUE DE LITTÉRATURE

1.1 L'analyse statique

Ces dernières années, l'utilisation de l'analyse statique de code dans la recherche de vulnérabilités s'est largement démocratisée. En effet, la présence de bogues est récurrente dans les applications, et leur exploitation de plus en plus simple en fait des dangers potentiels. Mitropoulos, Gousios, et Spinellis (2012) ont montré que leur nombre augmentait considérablement au fil des versions d'une application : la plupart d'entre eux proviennent de librairies externes qui ne sont pas mises à jour. Il est donc fondamental de corriger ces vulnérabilités rapidement. De plus, leur correction au fil du développement par une personne expérimentée peut nous en apprendre énormément sur leur importance et leur priorité (Kim & Ernst, 2007). En effet, bien que ce résultat soit intuitif, celles qui sont corrigées rapidement sont en général les plus graves.

L'analyse statique est donc utilisée pour aider le développeur à trouver ces bogues. Le principe est d'analyser le code avant son exécution, pour y détecter d'éventuels problèmes. Livshits et Lam (2005) ainsi que Schwartzbach (2008) développent les concepts fondamentaux sur lesquels repose cette analyse et donnent des exemples concrets des principales vulnérabilités (Injection SQL, XSS, etc.). Ils y détaillent notamment les différentes méthodes de parcours de code afin d'y détecter des modèles caractéristiques de vulnérabilités (« ou patterns ») : cela peut être une méthode dépréciée, une erreur de configuration, ou bien souvent la propagation d'une variable souillée (dont la valeur est entrée par l'utilisateur et qui n'est pas filtrée). La recherche et la découverte de ces variables potentiellement dangereuses sont appelées « taint analysis ». Komiya, Paik, et Hisada (2011) donnent en guise d'introduction un exemple concret de ce qu'un attaquant peut réaliser grâce à l'une de ces variables : une injection SQL. Cette vulnérabilité, très connue et très dangereuse, permet de réaliser une action malveillante sur la

base de données d'un serveur (vol de données, update, suppression de la base, etc.). Pour réaliser cela, l'idée est de passer une chaîne de caractères sous un format irrégulier au niveau d'une entrée utilisateur, qui va par la suite totalement changer le sens de la requête SQL envoyée par le serveur à la base de données. Par exemple, considérons la requête suivante :

*SELECT * FROM user_table WHERE id='user_id' AND password='pass';*

Cette requête récupère l'identifiant et le mot de passe qu'entre l'utilisateur, et demande à la base de données de renvoyer la ligne de la table correspondante (contenant par exemple des informations à afficher). Or, si l'utilisateur malveillant ne rentre pas son identifiant, mais la chaîne suivante : *' OR 1=1; --* , la requête SQL envoyée se transformera en :

*SELECT * FROM user_table WHERE id=' ' OR 1=1; -- AND password='pass';*

Or, cette requête renverra tous les champs de la table pour lesquels l'identifiant vaut une chaîne vide OU que 1 = 1, ce qui est évidemment toujours vrai. Ainsi, toute la table sera retournée par l'application. Halfond, Viegas, et Orso (2006) apportent des explications plus avancées sur cette vulnérabilité, ainsi que des moyens de défense. Néanmoins, on voit qu'il est fondamental pour une application de vérifier les champs qui sont fournis par l'utilisateur, pouvant contenir des valeurs souillées : l'analyse statique est une des méthodes permettant de détecter cela.

Néanmoins, le concept d'analyse statique n'est pas aussi récent, et a fait son apparition avec les premiers ordinateurs (fin des années 1940). Le premier outil conçu spécialement dans ce but est *Lint*, un analyseur de langage C (S. C. Johnson, 1977). Cet outil est encore utilisé aujourd'hui, ayant été amélioré afin d'élargir ses capacités de détection : une des extensions importantes est notamment *LCLint* (Evans, Guttag, Horning, & Tan, 1994). Par ailleurs, l'analyse statique s'est énormément développée vers la fin des années 1990 : la plupart des compilateurs actuels l'ont intégrée à leur fonctionnement, de façon plus ou moins poussée. L'un des déclencheurs de sa popularité a probablement été l'accident du vol inaugural de la fusée Ariane 5, qui s'est autodétruite quelques dizaines de secondes après son décollage en

raison d'un dépassement d'entier présent dans les calculateurs électroniques du pilotage automatique. Ce bogue est encore aujourd'hui considéré comme l'un des plus chers de l'histoire. De ce fait, de nombreux outils se sont développés à cette période, comme PREfix et ITS4, qui se concentrent sur des vulnérabilités de sécurité dans du code C et C++ (Bush, Pincus, & Sielaff, 2000; Viega, Bloch, Kohno, & McGraw, 2000), ou UNO, qui analyse statiquement du code C en respectant des propriétés définies par l'utilisateur (Holzmann, 2002). La plupart de ces outils se concentrent sur un langage de programmation précis (Java, C++) et peuvent se baser sur le code source directement, ou bien sur le code compilé (Louridas, 2006). C'est le cas de FindBugs (Hovemeyer & Pugh, 2004), renommé récemment SpotBugs, qui se concentre sur du pseudo-code Java (ou *bytecode Java*), qui est un code binaire compilé présentant une instruction par ligne. Il fonctionne classiquement grâce à des détecteurs de modèles de bogues, à savoir des morceaux de codes ne suivant pas la bonne pratique (dans le cas de l'utilisation d'une librairie externe par exemple). Le simple fait de parcourir le code déclenche ces détecteurs en cas de présence d'une irrégularité. C'est pourquoi FindBugs est si souvent utilisé, encore à l'heure actuelle : comme l'explique Ayewah, Hovemeyer, Morgenthaler, Penix, et Pugh (2008), c'est sa simplicité qui fait sa force. En contrepartie, cet outil ne détecte pas tout, comme certaines exceptions de pointeurs nuls, dépassements de tableaux, etc. D'autres le font, mais sont plus lourds et ressortent également plus de fausses alertes.

Certaines études ont d'ailleurs pour but d'évaluer les outils d'analyse et de comparer leurs résultats : c'est le cas de Velicheti, Feiock, Peiris, Raje, et Hill (2014), qui se basent sur la suite de test Juliet, un ensemble de classes java contenant des vulnérabilités déjà labélisées, pour en évaluer deux d'entre eux (non cités). Le but de leur étude est de créer un environnement python visant à prévoir quel outil sera le plus performant en fonction du code analysé (code potentiellement vulnérable). Goseva-Popstojanova et Perhinschi (2015), qui évaluent trois outils d'analyse statiques anonymes, se concentrent, sur l'estimation de la couverture de l'outil sur un code en C++ donné (combien de vulnérabilités ont été manquées). Ils en concluent que même si les trois quarts sont trouvés, les outils pourraient être plus performants. Enfin, Charest, Rodgers, et Wu (2016) utilisent également la suite Juliet pour comparer quatre outils

(FindBugs, Code Pro Analytix, JLint, et VisualCodeGrepper). Bien qu'ils détectent tous environ la même quantité de bogues, il apparaît que chaque outil est plus performant que les autres sur la détection de certains types de vulnérabilités, et moins performant sur d'autres : il semble donc qu'à l'heure actuelle, il n'existe pas d'outil parfait.

Enfin, l'analyse statique est parfois combinée à une analyse de graphe. Un graphe est avant tout un modèle abstrait permettant de mettre des objets en relation, en les reliant entre eux. De manière générale, utiliser un graphe est un avantage non négligeable pour un développeur : il est en effet parfois très complexe de comprendre du code, notamment lorsque celui-ci contient beaucoup de classes totalisant des milliers de lignes, et de nombreuses méthodes s'appelant les unes les autres. Même en utilisant un environnement de développement intégré (IDE), il peut être très complexe de faire un suivi effectif de variables par exemple, ou plus généralement de suivre la trace du programme. Un graphe aide à faire cela, et il en existe de nombreux types permettant de représenter du code. L'un des plus utilisés est l'arbre syntaxique abstrait : il est important de rappeler qu'un arbre est un type de graphe particulier, qui n'est pas toujours orienté (les arrêtes n'ont pas forcément de sens), qui n'est pas cyclique (il n'existe pas de circuit fermé permettant de revenir à un point en partant de ce même point), et qui est connexe (d'un seul tenant). Les nœuds de cet arbre sont des opérateurs, et les feuilles sont les variables, ou constantes du programme. Ainsi, celui-ci est très utile pour représenter des morceaux uniques du code : Baxter, Yahin, Moura, Sant'Anna, et Bier (1998) l'ont par exemple utilisé pour trouver des répétitions dans un programme. Un autre type de graphe est le graphe de flot de contrôle, qui représente tous les chemins possibles de l'exécution du programme : cela peut être très utile à un développeur voulant déboguer son code. Par ailleurs, ces différents graphes combinés peuvent être utilisés pour mettre en évidence des modèles de vulnérabilités et aider à leur détection (Yamaguchi, Golde, Arp, & Rieck, 2014).

Pour conclure, bien que la représentation du code sous forme de graphe ne soit qu'une simple méthode de visualisation, elle peut être très intéressante à combiner avec l'analyse statique dans le but de découvrir des vulnérabilités. Des modèles peuvent plus facilement être mis en évidence, notamment grâce à certains systèmes permettant d'effectuer des requêtes sur les

graphes, simplifiant grandement les recherches. C'est par exemple le cas de Neo4j, un système de gestion de base de données par graphe (Webber, 2012), qui est décrit plus précisément dans la section 2.2.

1.2 Classification des alertes

Néanmoins, l'outil en lui-même n'est pas le seul problème : la gestion et la classification des alertes post-analyse sont également un point central de l'analyse statique, car c'est ce qui va directement impacter sur l'action prise par le développeur une fois le rapport émis par l'outil. Muske et Serebrenik (2016) ont réalisé une étude d'ensemble sur 79 systèmes de gestion d'alertes issues d'analyse statique. Ceux-ci ont été classés en cinq catégories principales en fonction de la méthode utilisée :

- le « clustering », qui consiste à effectuer un regroupement des alertes par similarité;
- le « ranking », qui consiste à prioriser les alertes selon certains critères, comme le retour du développeur;
- le « pruning », qui regroupe différentes méthodes de classification binaire pour éliminer les fausses alertes, comme l'apprentissage machine;
- la simplification d'analyse, qui s'effectue notamment par ajout d'une interface utilisateur pour aider le développeur dans sa gestion des alertes;
- l'utilisation d'un LSAT (*Light Weight Static Analysis Tool*), méthode qui ne génère que des alertes réelles en se basant sur des modèles très précis, mais ne détecte donc pas toutes les vulnérabilités.

Ces cinq méthodes sont globalement les plus utilisées, notamment car elles facilitent le travail du développeur sans être extrêmement compliquées à mettre en place. Cependant, il existe également deux autres méthodes, plus complexes.

La première consiste à éliminer le plus de faux positifs possible, de façon certaine, par des méthodes précises comme l'analyse symbolique. L'idée derrière une telle analyse est de parcourir le code et remplacer les variables par des valeurs symboliques et non des valeurs

réelles, ce qui permet de créer un arbre représentatif de l'exécution du programme, avec tous les cas possibles. Chaque nœud correspond à une ligne où différents cas sont possibles (une condition if par exemple). Chaque cas est représenté par une branche menant à un autre nœud, etc. (dans le cas d'une condition if, deux branches seraient créées, pour le cas où la condition est vérifiée ou non). Par la suite, il suffit de remplacer les valeurs symboliques par des valeurs réelles dans l'arbre pour avoir le schéma d'exécution du programme pour une situation donnée. Cadar et Sen (2013) expliquent de façon plus détaillée, et avec des exemples concrets, les fondements de l'analyse symbolique, et son utilisation à l'heure actuelle.

La seconde méthode est de combiner l'analyse statique avec une analyse dynamique pour affiner les résultats. L'analyse dynamique, au contraire de l'analyse statique, consiste à exécuter réellement le programme avec des valeurs variées afin d'étudier son fonctionnement en direct, et détecter d'éventuels problèmes. Ainsi, certaines vulnérabilités manquées par l'analyse statique peuvent être détectées. Néanmoins, cette analyse est beaucoup plus coûteuse à réaliser, notamment en temps : plus il y a de valeurs testées, plus la chance de trouver une vulnérabilité est grande. On peut par exemple penser à un outil d'analyse dynamique ayant pour but de trouver une injection SQL sur une application Web vulnérable, qui va tester un nombre important de chaînes de caractère pour trouver la bonne qui permettra l'attaque. Ainsi, les deux méthodes peuvent se combiner, et peuvent être efficaces si bien utilisées : c'est ce que démontre Ernst (2003).

Chaque méthode ayant ses avantages et ses défauts, la combinaison de plusieurs d'entre-elles peut se révéler judicieuse, comme le regroupement d'alertes par modèles semblables de bogues et l'utilisation d'apprentissage machine pour éliminer les faux positifs (Hanam, Tan, Holmes, & Lam, 2014). Shen, Fang, et Zhao (2011) ont également pris en compte le retour de l'utilisateur pour améliorer le classement d'alertes de FindBugs, outil d'analyse statique évoqué auparavant, qui ne se basait que sur des modèles de vulnérabilités. Cela permet notamment à un développeur de choisir lesquelles présentent le plus de risques pour son application. Par ailleurs, certaines recherches visent uniquement à identifier ces modèles, par différents moyens, pour constituer une documentation importante que les outils d'analyse

statique pourraient utiliser pour obtenir des résultats meilleurs : par exemple, Lam, Martin, Livshits, et Whaley (2008) ont utilisé un langage particulier appelé PQL (*Program Query Language*), permettant de traquer la provenance d'une variable dans du code, pour détecter 30 nouveaux modèles de vulnérabilités dans une dizaine d'applications réelles. Plus récemment, Reynolds et al. (2017) se sont basés sur la suite de test Juliet en utilisant de la réduction de code, et en ont trouvé 14 nouveaux, qui concernaient des vulnérabilités très largement répandues.

1.3 L'apprentissage machine

Enfin, l'une des méthodes intéressantes ayant fait son apparition ces dernières années est l'apprentissage machine. Le principe est d'automatiser le tri des alertes et de détecter le plus de faux positifs possible, sans que le développeur n'ait à fournir d'effort. En effet, un certain nombre d'alertes vont être triées à la main, et données à la machine pour s'entraîner. Par la suite, en se basant sur ces données, appelées vérité terrain, elle va trier elle-même les nouvelles alertes en se basant sur des caractéristiques définies par le développeur : plus les caractéristiques sont précises et décrivent au mieux un modèle de vulnérabilité, plus la classification effectuée par la machine va être précise. De nombreux algorithmes d'apprentissage machine existent, et en fonction du problème et de ses caractéristiques, certains seront plus performants que d'autres : Yuksel et Sozer (2013) ont réalisé une étude dans laquelle ils comparent 34 algorithmes différents. Le but est de classer les alertes d'un outil d'analyse statique dans le cas d'une application réelle, en l'occurrence ici une application de télévision offrant des vidéos à la demande en ligne. Les caractéristiques retenues sont principalement le temps de vie des alertes, à savoir le temps passé entre le moment où l'alerte fait son apparition dans le code et le moment où elle est corrigée par le développeur, le niveau de priorisation de l'outil d'analyse, et l'avis du développeur sur l'importance de telle ou telle alerte. L'algorithme d'apprentissage machine est donc entraîné sur les premières versions de l'application, avec des alertes triées à la main, et classifie les alertes ultérieures automatiquement. Au final, les meilleurs algorithmes d'apprentissage machine pour ce problème obtiennent un excellent taux de bonne classification (plus de 90%), alors que les

moins bons sont à 60% environ. Cependant, sur d'autres problèmes, les résultats pourraient être inversés : en effet, les algorithmes ne sont pas tous efficaces sur les mêmes formats de données.

D'autres études ont également utilisé l'apprentissage machine pour classifier les alertes d'un outil d'analyse statique, chacune d'entre-elles considérant des caractéristiques différentes, ou des types de vulnérabilités différentes. Komiya et al. (2011) se sont concentrés exclusivement sur les injections SQL, en analysant avec précision chaque terme des requêtes SQL qui transitent du serveur à la base de données. Leurs caractéristiques considérées ne sont donc relatives qu'aux chaînes de caractères, et les résultats sont excellents : les requêtes malicieuses sont identifiées à 98%. Néanmoins, la mise en place d'un tel système est plus délicate qu'une simple analyse statique, car elle implique d'intercepter toutes les requêtes envoyées vers la base de données. Dans un registre plus proche de l'analyse statique, de nombreuses autres caractéristiques peuvent être utilisées pour entraîner les algorithmes d'apprentissage machine, comme la décomposition du code en symbole API, composé de différents éléments (noms de méthodes, types des variables, valeurs castées, etc.) décrivant le code ayant levé l'alerte (Yamaguchi, Lindner, & Rieck, 2011; Yamaguchi, Lottmann, & Rieck, 2012). Ces deux études, réalisées dans des environnements réels (logiciel de source libre de traitement de flux vidéo), ont permis la découverte de deux failles de sécurité graves, dont une totalement inconnue des développeurs de l'application. Dans une autre étude, Tripp, Guarnieri, Pistoia, et Aravkin (2014) impliquent directement l'humain, en créant un outil appelé ALETHEIA : ce dernier utilise l'apprentissage machine pour classer des alertes en se basant sur quelques caractéristiques définies (nom de la méthode source, ligne de l'appel, etc.), mais surtout sur ce que le développeur veut prioriser (mettre l'accent sur tel type de bogue, élimination des faux positifs ou conservation absolue de toutes les vraies alertes, etc.). Cet outil offre d'excellents résultats puisqu'il permet de réduire d'un facteur 2,8 le nombre de faux positifs. De leur côté, Medeiros, Neves, et Correia (2014) se sont concentrés principalement sur deux types classiques de vulnérabilités, les injections SQL et XSS. Ainsi, les caractéristiques décrivant chaque alerte ont pu être adaptées à ces vulnérabilités : il est principalement question de regarder s'il y a, dans le code correspondant à l'alerte, des fonctions de manipulations de

chaînes de caractères, ou s'il y a des fonctions permettant de filtrer les entrées de l'utilisateur, qui sont par définition souillées. En fonction de la présence ou non de telles fonctions, l'algorithme prendra sa décision, bonne dans 92% des cas selon les chercheurs. Koc, Saadatpanah, Foster, et Porter (2017) ont une approche bien différente des précédentes : après avoir réalisé pour chaque alerte une découpe précise du code malicieux, ils utilisent les différentes instructions du pseudo-code Java comme caractéristiques. L'idée étant d'avancer que certaines instructions seraient représentatives d'une vraie vulnérabilité, ou au contraire d'un faux positif. Ainsi, chaque caractéristique, correspondant à une instruction du pseudo-code, a pour valeur 0 ou 1, en fonction de si l'instruction est présente ou non dans le code malicieux. La taille de ce vecteur de caractéristique est donc le nombre de toutes les instructions de pseudo-code présentes dans le code des vulnérabilités du jeu de données d'entraînement. Les résultats sont très bons, avec près de 90% de bonne décision par l'algorithme d'apprentissage machine, et 98% de véritables vulnérabilités détectées.

Enfin, il est important de noter que certaines études ont essayé de complètement supprimer la partie d'analyse statique pour ne garder que l'apprentissage machine comme moyen de détection de vulnérabilités. C'est le cas de Chappelly, Cifuentes, Krishnan, et Gevay (2017), qui ont réalisé leur étude sur code écrit en langage C. Les caractéristiques choisies pour leur algorithme se basent sur le code opération, code d'instruction précisant à la machine la nature des opérations à effectuer (load, store, etc.). De plus, l'algorithme est entraîné séparément sur chaque type de bogue. Néanmoins, les résultats sont nettement moins bons que lorsqu'une analyse statique est effectuée au préalable : de nombreuses failles ne sont pas détectées, et à l'inverse beaucoup de faux positifs sont à déplorer. Il apparaît donc de manière évidente que l'apprentissage machine n'est pas fait pour trouver lui-même les vulnérabilités, mais est plutôt un outil à utiliser après coup pour confirmer les alertes ressortant d'une analyse statique, ou au contraire pour éliminer les faux positifs.

CHAPITRE 2

CONTEXTE DE DÉVELOPPEMENT ET OUTILS

Ce chapitre a pour but d'expliciter le contexte de l'étude, à savoir les outils existants qui sont déjà utilisés par les développeurs.

2.1 Un outil d'analyse statique : Find Security Bugs

2.1.1 SpotBugs

SpotBugs (SpotBugs, 2016) est le successeur de FindBugs, un outil d'analyse statique (SAT) très populaire. Son but est de détecter des bogues en tous genres dans des programmes Java, en se basant sur des modèles existants. Chaque modèle correspond au code idiomatique d'un bogue, et est appelé détecteur. SpotBugs base sa détection sur le pseudo-code Java (ou *bytecode*) : celui-ci est un code binaire compilé, composé d'instructions exécutables par la machine virtuelle Java. Il est contenu dans les fichiers à l'extension .class. Son format, une instruction binaire par ligne, en fait un code idéal à analyser statiquement pour un outil, car les modèles de bogues peuvent être clairement reconnus.

SpotBugs fonctionne de la façon suivante : les détecteurs sont utilisés les uns à la suite des autres sur le pseudo-code Java. Lorsqu'un détecteur pense trouver un bogue, à savoir constate un morceau de code ressemblant à son propre code idiomatique, il lève une alerte. Une fois que le code entier est vérifié, les alertes sont collectées pour être présentées au développeur.

Au total, SpotBugs est capable de détecter plus de 400 modèles de bogues différents. Ceux-ci sont répartis en différentes catégories leur attribuant une description :

- **Mauvaise pratique (BAD_PRACTICE)** : violation de principes essentiels à une bonne manière de coder, ce qui peut correspondre par exemple à une mauvaise gestion d'exceptions, ou un problème de sérialisation, ou encore le mauvais usage d'une méthode délicate, comme *equals(), hashCode()* ou *finalize()*;
- **Exactitude (CORRECTNESS)** : Bogue potentiel causé par une erreur dans le code, probablement non intentionnelle de la part du développeur (variable non nulle qui n'est pas initialisée par exemple);
- **Code malicieux vulnérable (MALICIOUS_CODE)** : morceau de code vulnérable pouvant être attaqué par un autre code (variable publique au lieu de privée, méthode sensible non restreinte à une permission particulière, etc.);
- **Exactitude d'un programme en chapelet (MT_CORRECTNESS)** : problème de code ayant un rapport avec la gestion de fils (ou « threads ») par exemple à la création d'un fil grâce à la méthode *run()* vide par défaut;
- **Performance (PERFORMANCE)** : code qui n'est pas forcément incorrect, mais inefficace;
- **Sécurité (SECURITY)** : utilisation d'une variable non fiable entrée par l'utilisateur, pouvant causer une vulnérabilité de sécurité (injection SQL ou XSS par exemple);
- **Code douteux (STYLE)** : code confus pouvant amener à des erreurs d'exécution, comme une variable initialisée qui n'est pas utilisée par la suite (appelée *dead store*), ou bien un switch mal initialisé pouvant causer l'exécution de tous les cas (oubli de l'instruction *break;*);
- **Expérimental (EXPERIMENTAL)** : modèles de bogues non validés, encore en expérimentation (l'outil étant de source libre, chacun peut soumettre les siens, qui sont par la suite en période de validation).

On voit donc que SpotBugs détecte des bogues en tous genres, il n'y a pas que des vulnérabilités de sécurité. De plus, chaque alerte vient avec deux informations, quel que soit son type : la confiance et le classement.

La confiance est une indication quant à la probabilité qu'une alerte soit un bogue réel. Si la confiance pour une alerte est élevée, cela montre que l'outil est quasiment certain d'avoir détecté un vrai problème.

Le classement, en revanche, indique la gravité du bogue détecté. Cette information est à la base un nombre entre 1 et 20 (1 pour le plus grave et 20 pour le moins grave), mais est représentée par 4 catégories facilement appréhendables par le développeur :

- *Scariest* (1-4);
- *Scary* (5-9);
- *Troubling* (10-14);
- *Of concern* (15-20).

Ainsi, une alerte peut être priorisée par le développeur selon ces deux critères, en fonction de s'il préfère corriger des bogues dont l'outil est certain, afin ne pas perdre de temps, ou s'il préfère être prudent et corriger les alertes graves, même si leur confiance est basse.

Pour conclure, SpotBugs est un outil simple et efficace, et c'est pourquoi il est utilisé par de nombreux développeurs voulant une vérification rapide de leur code. De plus, étant un logiciel de source libre, tout le monde peut soumettre de nouveaux détecteurs, ou créer des modules d'extension qui se concentrent sur certains types de bogues : c'est le cas de Find Security Bugs.

2.1.2 Find Security Bugs : explication et fonctionnement

Find Security Bugs est un module d'extension de l'outil d'analyse statique SpotBugs, qui se concentre sur les vulnérabilités de sécurité (Arteau, 2012). Il couvre 125 types d'entre elles, et contient presque 800 signatures uniques de fonctions dans sa base de données.

Il fonctionne de la même manière que SpotBugs, en scannant le pseudo-code Java grâce à des détecteurs, afin identifier des morceaux de code affaiblissant l'application et pouvant amener

à une exploitation de la part d'un attaquant. Il permet notamment de détecter de nombreuses failles de sécurité graves, présentes dans le Top 10 d'OWASP.

OWASP (*Open Web Application Security Project*) est un projet de source libre visant à promouvoir les bonnes pratiques de sécurité, en mettant à disposition de tous conseils et outils adaptés (OWASP, 2001). Tous les ans, un nouveau Top 10 des types de vulnérabilités de sécurité les plus dangereuses est publié afin de sensibiliser les développeurs. Find Security Bugs permet non seulement de détecter une grande partie de ces vulnérabilités, mais fournit également de la documentation et des références pour chaque modèle de bogue, avec notamment des indications de correction. Le développeur examinant une alerte est donc clairement informé de ce à quoi il fait face et comment le corriger.

Par ailleurs, tout comme SpotBugs, Find Security Bugs s'intègre très bien dans de nombreux environnements de développement intégré (Eclipse, Intellij IDEA, Android Studio, etc.), afin de proposer au développeur une interface simple d'utilisation et claire, directement dans son code. Il peut également s'exécuter en ligne de commande, ce qui est pratique notamment lors de l'utilisation d'un outil de gestion et d'automatisation de production de projets Java, comme Maven ou Ant. Enfin, il est également conçu pour être utilisé avec des systèmes d'intégration continue, comme Jenkins ou SonarQube (l'utilisation de ce dernier sera explicitée en section 2.2.). Cela est notamment dû à son format de données en sortie, facilement traitable : toutes les alertes générées sont stockées dans un unique fichier XML. Ce format est très pratique, car il permet de représenter les données de façon arborescente, grâce à l'utilisation de balises. Ainsi, chacune des alertes de Find Security Bugs correspond à un nœud de l'arborescence : autrement dit, elle est encadrée par des balises spécifiques <*BugInstance*>. Les sous-nœuds de chaque instance, correspondant à d'autres balises imbriquées au sein de celle-ci, contiennent des informations sur la vulnérabilité (son type, la ligne de code, la méthode, etc.). La figure 2.1 montre la représentation d'une alerte au format XML : on voit que les informations présentes sont très faciles à récupérer par un autre outil de gestion d'alertes par exemple.

Figure 2.1 Représentation d'une alerte au format XML

En un mot, Find Security Bugs est un outil très performant dans la découverte de vulnérabilités de sécurité, et simple d'utilisation, s'intégrant très bien avec de nombreux outils. Néanmoins, comme tout outil d'analyse statique, celui-ci a ses limites.

2.1.3 Limitations

D'une manière générale, l'analyse statique a ses limites. Parcourir le code sans l'exécuter permet de détecter à l'avance de nombreux problèmes, mais ne permet pas réellement de prévoir avec exactitude le comportement d'un programme, en fonction de la multitude de variables qui peut lui être passée. Cela a deux conséquences directes. D'une part, il est possible que l'outil ne détecte pas certains défauts du programme, qui se révéleront à l'exécution de celui-ci. D'autre part, et c'est le principal inconvénient d'un outil d'analyse statique, ce dernier va bien souvent générer de nombreux faux positifs. Un faux positif est une alerte générée par l'outil qui ne correspond pas à une vraie vulnérabilité. Le but principal est en effet de trouver le plus de vulnérabilités possible, notamment lorsque l'on comprend les dommages que peut causer seulement l'une d'entre elles (déni de service, vol de données sensibles, etc.). Ainsi, l'outil va en général ratisser large, et lever des alertes sur tout morceau de code pouvant hypothétiquement être malicieux dans un certain contexte, même si dans de nombreux cas, le code en question est sûr.

Find Security Bugs ne déroge pas à la règle. En effet, il a été évalué en utilisant un projet de test : OWASP Benchmark (Benchmark, 2015). Ce dernier a été créé pour évaluer des outils d'analyse statique sur huit types de vulnérabilités principales. Il est composé d'un peu plus d'un millier de classes de tests : certaines vulnérabilités sont réelles et d'autres sont fausses. Cependant, tous les cas sont déjà labélisés. Le but est d'évaluer la couverture de l'outil, à savoir sa capacité à détecter le plus de vulnérabilités diverses, et à ne pas considérer un faux positif comme l'une d'entre elles. Benchmark est conçu de manière à ce qu'une fois l'outil d'analyse statique passé sur l'ensemble des cas, un rapport soit généré, avec un graphique permettant de déterminer quels types de vulnérabilités ont été les mieux détectées, etc. En résumé, des statistiques sont ressorties automatiquement et permettent de donner un premier aperçu de l'efficacité de l'outil d'analyse statique. Ci-dessous, la figure 2.2 présente les résultats pour Find Security Bugs :

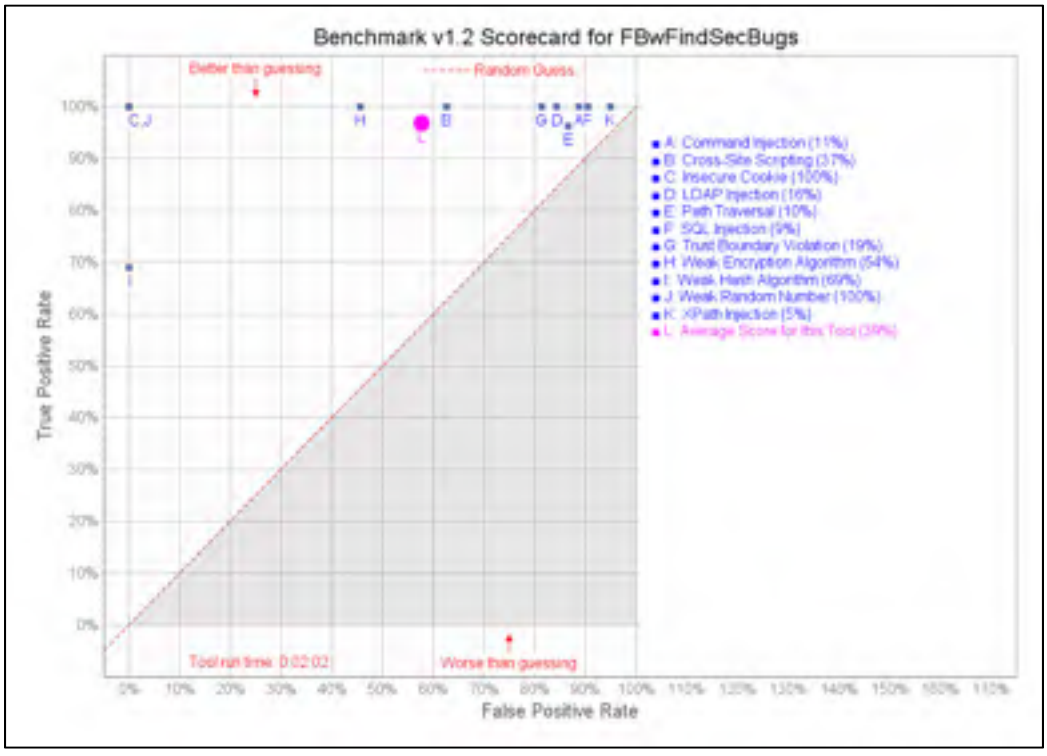

Figure 2.2 Graphique des résultats de Find Security Bugs sur OWASP Benchmark
Tirée de (FSB results on Benchmark, 2018)

Le score affiché pour chaque vulnérabilité à droite du graphique correspond au taux de vrais positifs (vraies vulnérabilités détectées comparées à l'ensemble de toutes les vraies vulnérabilités) moins le taux de faux positifs (fausses vulnérabilités détectées comme étant vraies comparées à l'ensemble des fausses vulnérabilités). Ces deux taux sont par ailleurs retrouvés sur le graphique lui-même : ainsi, plus l'outil est performant pour détecter une vulnérabilité sans générer de faux positifs, plus le point concerné se trouvera proche du coin supérieur gauche. Il ressort du graphique que Find Security Bugs est très bon pour détecter toutes les vulnérabilités réelles. En revanche, il a un taux de faux positifs très important. Les résultats globaux comprenant toutes les vulnérabilités sont présentés dans le Tableau 2.1 :

Tableau 2.1 Résultats globaux de l'analyse d'OWASP Benchmark par Find Security Bugs
Adapté de (FSB results on Benchmark, 2018)

TPR	**FPR**	**Score**	**TP**	**FN**	**TN**	**FP**	**Total**
96,84%	57,74%	39,10%	1370	45	622	703	2740

En plus du taux de vrais positifs (TPR), de faux positifs (FPR) et du score, les éléments suivants sont présentés :

- Vrais positifs (TP) : nombre de vulnérabilités réelles qui ont été correctement détectées;
- Faux négatifs (FN) : nombre de vulnérabilités réelles qui ont échappé à la détection;
- Vrais négatifs (TN) : nombre de fausses vulnérabilités qui ont été correctement évitées par l'outil et n'ont donc pas causé d'alertes;
- Faux positifs (FP) : nombre de fausses vulnérabilités qui ont été considérées comme réelles par l'outil, créant donc de fausses alertes.

Néanmoins, il est important de garder à l'esprit que ces résultats ne sont pas exactement représentatifs du comportement de l'outil d'analyse statique sur un projet réel. En effet, les cas de tests mis à disposition par Benchmark sont principalement du code auto généré. Ainsi, de nombreux cas sont similaires, et certains sont même très peu représentatifs d'un code écrit par

un développeur dans un projet d'application quelconque : par exemple, certaines conditions (if) sont toujours vraies, etc.

Il est donc difficile de juger de l'efficacité d'un outil d'analyse statique en se basant sur un projet spécialement conçu pour ça, car celui-ci ne donnera pas un aperçu très représentatif de la réalité. Par ailleurs, il est difficile de faire des tests sur des projets réels conséquents, car cela impliquerait trier toutes les alertes à la main avant, ce qui peut vite devenir un travail de titan. Néanmoins, Benchmark permet de mettre en évidence l'essentiel : Find Security Bugs est très efficace dans la découverte de vulnérabilités, mais il génère de nombreux faux positifs.

Pour terminer quant aux limitations de l'outil, on voit que les faux positifs sont responsables d'un paradoxe : l'analyse statique est supposée être simple et surtout rapide, car l'outil ne fait que parcourir le code, mais en réalité le temps que peut y passer un développeur est faramineux. Un gros projet générant de nombreuses alertes va obliger le développeur à vérifier chacune d'entre elles afin de faire le tri entre les vulnérabilités à corriger et les faux positifs. Cette tâche oblige le développeur à naviguer dans son code entre les classes, examiner les variables, etc. Cela est particulièrement chronophage, et à la longue il peut perdre en concentration et se tromper dans son tri, à savoir mettre de côté des problèmes importants en les prenant pour des erreurs de l'outil.

Afin d'essayer de pallier ce problème, des outils de gestion d'alertes existent à l'heure actuelle : ils disposent souvent d'une interface pour faire gagner du temps au développeur. C'est le cas de SonarQube.

2.2 Un outil de gestion d'alertes : SonarQube

SonarQube est un logiciel de source libre permettant de contrôler la qualité du code en continu. Il dispose d'un scanner permettant de parcourir le code à la recherche de bogues, de mauvaises pratiques d'écriture, de morceaux de code dupliqués, etc. De plus, il supporte un peu plus de vingt langages de programmation.

Il est notamment utilisé au sein d'environnement de développement (Eclipse, IntelliJ IDEA, etc.), ou avec des outils de production de projets Java, comme Maven ou Ant, avec lesquels il s'intègre parfaitement. Le développeur a simplement besoin de lancer l'application, et à se connecter à l'interface Web de SonarQube. À partir de celle-ci, il va pouvoir faire de nombreuses choses : gérer ses projets, avoir une vision globale du nombre de bogues pour chacun d'entre eux, générer des graphes d'évolution du projet en termes de qualité de code, et surtout examiner au cas par cas chaque alerte. Celle-ci sera présentée directement avec le code malicieux en question, et le développeur pourra la trier : en d'autres mots, il pourra lui attribuer un flag, en fonction de si l'alerte est confirmée, si c'est un faux positif, si elle est en cours d'analyse, etc. Il pourra de plus ajouter des commentaires, ce qui peut être utile si un autre développeur travaille sur le sujet. La figure 2.3 ci-dessous donne un aperçu de l'interface de gestion des alertes.

Figure 2.3 Interface de gestion d'alertes de SonarQube

Par ailleurs, ce qui fait la force de ce logiciel est qu'il supporte l'ajout de modules d'extension (ou *plugins*), permettant d'augmenter ses fonctionnalités. L'un de ces modules est celui de Find Security Bugs. En effet, le scanner intégré dans SonarQube n'est pas spécialisé dans la détection de vulnérabilités de sécurité. Néanmoins, il est possible de récupérer les alertes émises par Find Security Bugs pour les traiter au sein même de SonarQube afin de bénéficier

de son interface, ainsi que des différents outils de graphes et statistiques qu'il procure, notamment pour le suivi d'un projet sur le long terme.

Pour résumer, Find Security Bugs s'intègre très bien à SonarQube, et ce dernier apporte une solution intéressante à la gestion des alertes post analyse statique. Il permet au développeur de gagner du temps, même s'il n'influe en rien sur l'analyse en elle-même. Il est cependant possible de préciser cette dernière, grâce à un support visuel : la représentation du code sous forme de graphe.

2.3 Un outil de gestion de données par graphe : Neo4j

Neo4j est un logiciel de source libre de gestion de base de données, basé sur des graphes, et développé en Java (Neo4j, 2007).

Au sein de ce système, toutes les données sont représentées soit par des nœuds, soit par des arcs (arêtes), soit par des attributs. Les différents nœuds sont reliés par des arcs afin d'établir des relations, et chacun d'eux (nœuds ou arcs) peut avoir un nombre illimité d'attributs. Un attribut est une paire ('clé', 'valeur'), bien souvent de simples chaînes de caractères ou valeurs numériques. De plus, il est possible d'attribuer à ces éléments un label, pour faciliter les recherches. Ce format simple de représentation des données permet donc une recherche et un système de requêtes rapides à prendre en main et efficaces.

La base de données est d'ailleurs construite de façon à être très performante : les jointures entre les éléments sont précalculées lors de l'écriture de la base, et non calculées lors de la lecture comme avec des bases de données relationnelles. Des grands ensembles peuvent donc être traités par Neo4j. De plus, un langage de requête spécial a été créé avec ce système : Cypher. Ce langage, de source libre depuis 2015 et utilisé également en dehors de Neo4j, a été créé spécialement pour les bases de données graphiques. Il permet de réaliser plus efficacement que le langage SQL les opérations de parcours nécessaire à la recherche ou la modification d'un graphe de propriétés. En plus de se concentrer sur les éléments classiques d'un graphe (nœuds

et arcs, qui forment une relation), ce langage va également utiliser les attributs et les labels pour accéder plus rapidement au résultat voulu. De plus, la syntaxe est très simple à comprendre : elle est articulée autour de trois termes, MATCH, WHERE and RETURN. MATCH est utilisé pour décrire la structure de la donnée cherchée, en se basant sur sa relation (à savoir type de nœuds et arcs), WHERE apporte des contraintes additionnelles (notamment au niveau des attributs et labels), et RETURN explicite la valeur à renvoyer. Enfin, au sein de la requête, les nœuds sont indiqués entre parenthèses, et les arcs entre crochets. De ce fait, Cypher est un langage simple à comprendre et utiliser, mais très puissant.

Par ailleurs, l'interface de visualisation de graphe de Neo4j est très conviviale pour l'utilisateur. Elle est accessible facilement par navigateur, et les requêtes Cypher peuvent y être effectuées en direct. La figure 2.4 ci-dessous en montre un aperçu.

Figure 2.4 Interface de visualisation de Neo4j

Ainsi, on voit que Neo4j est spécialement conçu pour créer, parcourir, et analyser des réseaux de données connectées entre elles, et pour trouver des informations cachées, ou tout du moins difficiles à mettre en évidence, au sein de ces réseaux. C'est dans cette optique que ce système va être utilisé conjointement à Find Security Bugs, afin de pouvoir récupérer efficacement et

rapidement de nombreuses informations sur le code correspondant à chaque alerte (comme le suivi des variables impliquées par exemple). Il est en effet possible de représenter le code sous forme de graphe. Ceci est explicité plus spécifiquement en section 3.2.

CHAPITRE 3

PRÉSENTATION DE LA SOLUTION

3.1 Méthodologie

Cette section détaille la méthodologie suivie ici lors de l'élaboration de la solution. Celle-ci se retrouve dans de nombreuses études traitant d'apprentissage machine, et s'inspire notamment de Koc et al. (2017), qui ont effectués une étude très similaire concernant l'amélioration du triage des alertes issues de l'analyse statique par l'apprentissage automatique. L'idée est de scinder le problème en trois parties. Dans un premier temps, le but est de choisir et d'extraire des propriétés intéressantes décrivant les alertes, afin de construire le vecteur de caractéristiques. Celui-ci est la clé de voûte de l'apprentissage machine : l'algorithme va utiliser ces caractéristiques pour construire son modèle. Dans un second temps, on s'intéresse à la phase d'apprentissage à proprement parler : quel(s) algorithme(s) choisir, sur quelles données baser son apprentissage, etc. Enfin, la dernière étape est la phase d'évaluation et de validation de la solution : en d'autres termes, le but est d'interpréter les résultats et de discuter quant aux éventuelles limites de ceux-ci. Un schéma complet de l'approche réalisée ici est présenté ci-dessous en Figure 3.1. Les différentes parties qui y sont représentées sont détaillées dans les sections suivantes.

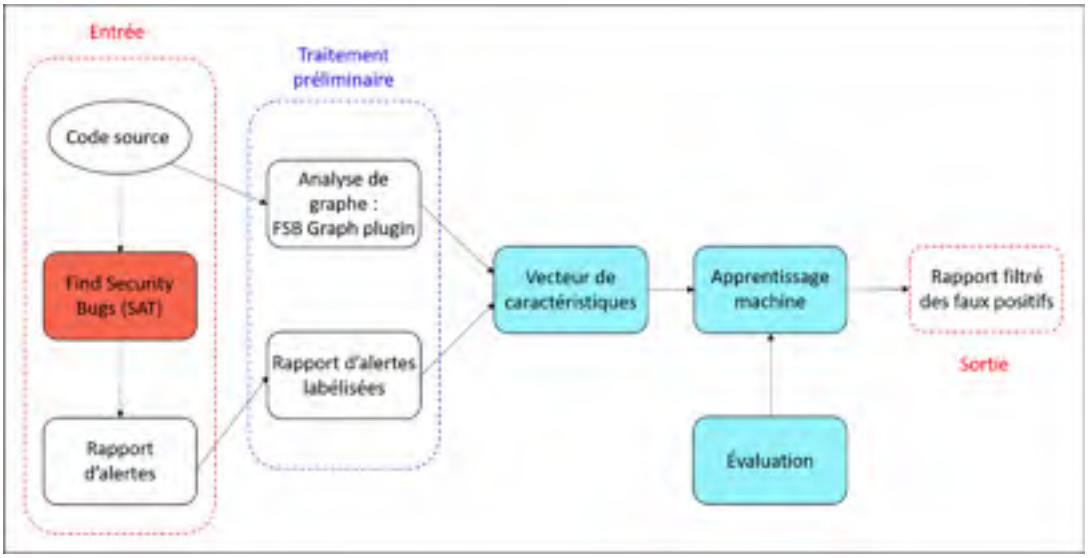

Figure 3.1 Schématisation de la méthodologie suivie
Adaptée de Koc et al. (2017)

3.1.1 Extraction du vecteur de caractéristiques

Comme annoncé précédemment, le vecteur de caractéristiques est l'un des points les plus importants à considérer lors de l'utilisation d'apprentissage machine. En effet, si les attributs choisis ne sont pas suffisamment représentatifs de l'élément à classifier, les résultats ne pourront pas être bons, et ce peu importe l'algorithme choisi.

Afin de trouver les meilleures caractéristiques possible, l'interrogation est la suivante : comment un développeur fait-il pour classifier manuellement une alerte (en tant que vulnérabilité réelle ou faux positif) ? En d'autres termes, quel est son cheminement de pensée, et sur quels critères se base-t-il pour rendre son verdict ? Par la suite, et pour apporter plus de précisions à cette recherche de caractéristiques, il est également intéressant d'examiner de nombreux cas de fausses alertes levées par Find Security Bugs, en essayant de comprendre pour chacune d'entre elles pourquoi l'outil l'a considéré comme une vulnérabilité.

Enfin, après mise en évidence des caractéristiques potentielles, il est nécessaire de comprendre en profondeur les outils du contexte déjà existants (cf. Chapitre 2) afin de trouver un moyen

d'extraire ces caractéristiques pour chaque alerte. En l'occurrence, celles-ci sont récupérées grâce au rapport d'alertes de Find Security Bugs d'une part, et à une analyse du graphe représentant le code source d'autre part. La construction du graphe est détaillée en section 3.2.

3.1.2 Phase d'apprentissage

Une fois le vecteur de caractéristiques extrait, il est alors temps de passer à la phase d'apprentissage, lors de laquelle l'algorithme va créer son modèle. Ici, modèle est synonyme d'algorithme entraîné : concrètement, lorsque la machine s'est entraînée sur un jeu de données labélisées, il en résulte ce modèle qui va pouvoir être utilisé par la suite sur un jeu de données non labélisées cette fois-ci, afin d'effectuer une classification, ou qui va pouvoir être mis à jour en incorporant de nouvelles données classifiées, afin de gagner en précision.

Néanmoins, de nombreux algorithmes d'apprentissage machine existent. Contrairement à la cryptographie, où certains algorithmes sont prouvés comme étant plus robustes que d'autres et donc fortement recommandés à utiliser, les algorithmes d'apprentissage machine ont différentes méthodes de fonctionnement, ce qui va les amener à être plus ou moins performants en fonction du type de données à traiter, et de leurs caractéristiques. Par exemple, certains vont être très bons pour réaliser une classification binaire, d'autres seront plus performants pour traiter des attributs au format numérique, etc. De ce fait, un réel travail de compréhension, d'évaluation et de comparaison de différents algorithmes doit être effectué : cela est en effet impératif dans le développement de la solution.

3.1.3 Évaluation, validation et discussion

En tout dernier a lieu la phase d'évaluation : il est crucial de valider la solution, car cela prouve l'efficacité de l'algorithme en même temps que les caractéristiques choisies. Ainsi, il est fondamental de trouver une méthode d'évaluation correcte, et cela apporte son lot de défis.

Tout d'abord, le choix du jeu de données labélisées est important : celui-ci doit être suffisamment grand pour que les résultats soient relativement précis, et surtout il faut que les

données soient labélisées, ce qui peut être un énorme défi si cette labélisation doit être effectuée manuellement. Ensuite, il est nécessaire de trouver des mesures d'évaluation intéressantes : autrement dit, quelles valeurs peuvent être mesurées pour valider la solution ? Il est intuitif d'avancer qu'il suffit simplement de regarder le pourcentage d'éléments correctement classifiés, mais est-ce vraiment le cas ? Un mauvais choix des mesures d'évaluation peut mettre en danger la validité de la solution. Enfin, une fois le jeu de données d'entraînement et les mesures d'évaluation choisis, il ne manque plus qu'à analyser les résultats pour chaque modèle (chaque algorithme entraîné).

De cette analyse découlent deux interrogations. La première est de savoir s'il existe, dans notre cas, un algorithme optimal apportant de meilleurs résultats que les autres à tous les niveaux. Ceci est fortement lié à l'intégration de la solution dans le contexte existant. Le second est une discussion concernant les limites de la solution : les expliciter est une part importante de l'étude, car cela ouvre la voie à des améliorations, et potentiellement d'autres recherches dans le domaine.

Pour conclure, le reste du chapitre 3 présente la solution, en détaillant sa construction, tandis que le chapitre 4 se concentre sur l'analyse des résultats ouvrant à discussion.

3.2 Représentation graphique du code

Comme expliqué précédemment, l'une des premières étapes de résolution du problème a été de se mettre à la place d'un développeur triant des alertes. Autrement dit, il a été question de comprendre sur quels critères celui-ci se base pour faire sa classification, et notamment éliminer les faux positifs.

Il n'existe en l'occurrence pas de critère universel sur lequel se baser pour confirmer, ou au contraire infirmer, de façon certaine une prévision de l'outil d'analyse statique. Ce que le développeur va faire va beaucoup dépendre du contexte auquel il est confronté. Parfois, il suffit de regarder le code autour de la potentielle vulnérabilité et comprendre certaines conditions,

parfois il faut se concentrer sur les appels à des interfaces de programmation applicatives (API) particulières : en fonction du contexte, telle ou telle API peut se révéler dangereuse. Enfin, il est très souvent utile de réaliser un suivi des différentes variables en jeu, qui peuvent provenir de la méthode contenant la vulnérabilité, mais aussi d'autres méthodes se trouvant dans d'autres classes. Ces variables sont possiblement souillées (entrées par l'utilisateur) et donc dangereuses : cependant, dans certains cas, elles peuvent également être saines. L'outil d'analyse statique a du mal à faire la différence, notamment si les variables se propagent entre différentes classes.

Un exemple simple est développé sur la Figure 3.2.

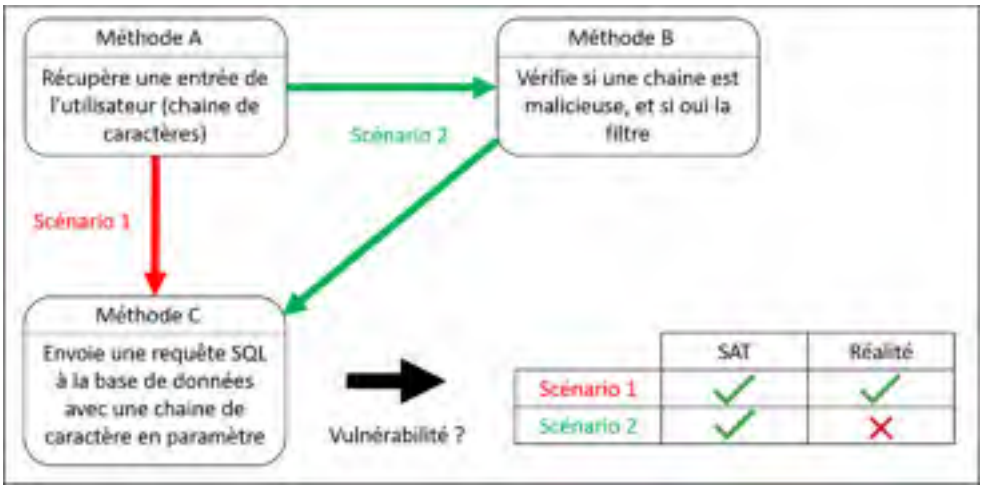

Figure 3.2 Exemple de cas classique causant un faux positif par l'outil d'analyse statique

Ici, l'outil d'analyse statique (SAT) détecte une vulnérabilité dans la méthode C, car celle-ci exécute une requête en prenant en paramètre une variable entrée par l'utilisateur. Dans le premier scénario (rouge), la variable vient directement de la méthode A, qui demande à l'utilisateur d'entrer une information quelconque. Ce que l'utilisateur entre n'est pas filtré, et la requête construite dans la méthode C peut mener à une injection de code malicieux. Or, dans le second scénario (vert), la variable entrée par l'utilisateur, depuis la méthode A, est envoyée dans la méthode B (située possiblement dans une autre classe), qui filtre la variable et la rend saine, peu importe ce qu'elle contient. Ainsi, la requête envoyée par la méthode C ne sera dans

ce cas pas dangereuse. Néanmoins, ces deux cas ne sont pas différents pour l'outil d'analyse statique, qui va se concentrer uniquement sur la méthode C et y détecter un modèle de code dangereux.

Ainsi, on voit que le développeur va dans de nombreux cas devoir sortir du cadre de la vulnérabilité (tel qu'il est présenté dans le rapport de vulnérabilités) pour suivre des variables et vérifier différents appels à des API qui auraient échappé à l'outil d'analyse statique. En pratique, cela n'est pas si compliqué à faire : les environnements de développement intégré offrent de nos jours des fonctionnalités permettant de simplifier grandement cette tâche, même si cela prend un certain temps et peut devenir fastidieux si le nombre d'alertes est élevé.

C'est de cette observation qu'est née l'idée de représenter le code sous forme d'un graphe, en utilisant l'outil Neo4j, afin de simuler un suivi des variables et des appels à différents API un peu à la façon d'un développeur dans son environnement de développement intégré. Les avantages de cette représentation sont multiples : l'aspect visualisation est tout d'abord très pratique, et peut venir faciliter l'analyse réalisée dans l'IDE. De plus, il est relativement simple d'accéder à des informations précises dans un graphe, grâce au système de requêtes : récupérer un attribut précis d'une variable, provenant elle-même d'une méthode donnée, est très rapide en questionnant le graphe via une requête Cypher. Ainsi, l'extraction de caractéristiques représentatives d'une potentielle vulnérabilité, et utiles à l'algorithme d'apprentissage machine, va être grandement simplifiée.

Le graphe est logiquement construit autour d'un type d'élément : les variables. C'est en effet celles-ci qui vont être traquées, et dont vont découler les caractéristiques extraites. Concrètement, chaque variable fait partie d'un des trois types suivants :

- Paramètre (P) : la variable est simplement l'un des paramètres d'une méthode;
- Valeur de retour (R) : la variable est une valeur de retour d'une méthode;
- Champ, ou « field » (F) : la variable est une variable *globale*, elle est extérieure à une méthode ou un constructeur, mais interne à la classe, et est utilisée pour stocker l'état d'un objet.

De plus, chacune d'entre elles va se voir attribuer un état : *TAINTED*, *SAFE* ou *UNKNOWN*. Par défaut, les paramètres sont initialisés avec la valeur *UNKNOWN*. Cependant, si la variable provient directement d'une entrée utilisateur, son état sera *TAINTED*, tandis que si elle provient au contraire d'une méthode connue jugée saine, son état sera *SAFE*. Par la suite, les états se propagent : on peut prendre l'exemple d'une méthode A retournant une chaîne de caractères, qui va être prise en paramètre dans une autre méthode B. L'état de la variable correspondant à la chaîne de caractères retournée par la méthode A va être transféré au paramètre de la méthode B (car c'est en soit la même variable, mais qui correspond à 2 nœuds dans le graphe).

Ainsi, chaque nœud dans le graphe correspond à une variable dans un de ses états : une variable étant transmise entre différentes méthodes résultera en plusieurs nœuds (comme indiqué dans l'exemple précédent). Chaque nœud possède par ailleurs trois attributs, listés ci-dessous.

- Son Nom : signature de la méthode (ou classe dans le cas d'un champ) impliquant la variable. La signature d'une méthode (ou fonction) comprend le nom sa classe, suivi du nom de la méthode, suivi du type des paramètres (et du type de la valeur de retour s'il y a lieu). Plus de détails sur la signature d'une méthode seront donnés en section 3.3.3.
- Son Type : paramètre, retour, ou field.
- Son État : *TAINTED*, *SAFE* ou *UNKNOWN*.

Les différents nœuds sont reliés entre eux par des arêtes, unidirectionnelles, symbolisant le transfert d'une variable. En reprenant l'exemple précédent, on aurait dans le graphe deux nœuds reliés par une arête, schématiquement tel que suit : [variable_retour_methodeA] → [parametre_methodeB].

Plus précisément, considérons le code Java ci-après.

```
1  public void bad(HttpServletRequest request, HttpServletResponse response) throws Throwable {
2
3      String data = request.getParameter("name");
4
5      Connection dbConnection = null;
6      Statement sqlStatement = null;
7
8      try {
9          dbConnection = IO.getDBConnection();
10         sqlStatement = dbConnection.createStatement();
11
12         Boolean result = sqlStatement.execute("insert into users (status) values ('updated') where name='"+data+"'");
13     }
14
15 [...]
```

Figure 3.3 Exemple de code Java vulnérable à une injection SQL

Ce code est vulnérable à une injection SQL. En effet, à la ligne 3, on lit une donnée provenant d'un *getParameter*, donc pouvant avoir été modifié par l'utilisateur. Par la suite, à la ligne 12, on exécute la requête SQL sans avoir filtré la donnée. Lorsque l'on créé le graphe, l'outil d'analyse statique détecte que la variable *data* retournée est souillée à la ligne 3. Ainsi, cet état sera transmis au paramètre du *sqlStatement.execute* : on aura deux nœuds dans le graphe, relié par un arc symbolisant le transfert de variable. La figure 3.4 présente le graphe résultant, avec les attributs de chaque nœud. La représentation n'est pas celle de l'outil Neo4j présenté en section 2.3, car cette dernière n'était pas suffisamment explicite visuellement, notamment au niveau des attributs de chaque nœud : cependant, les données sont exactement les mêmes.

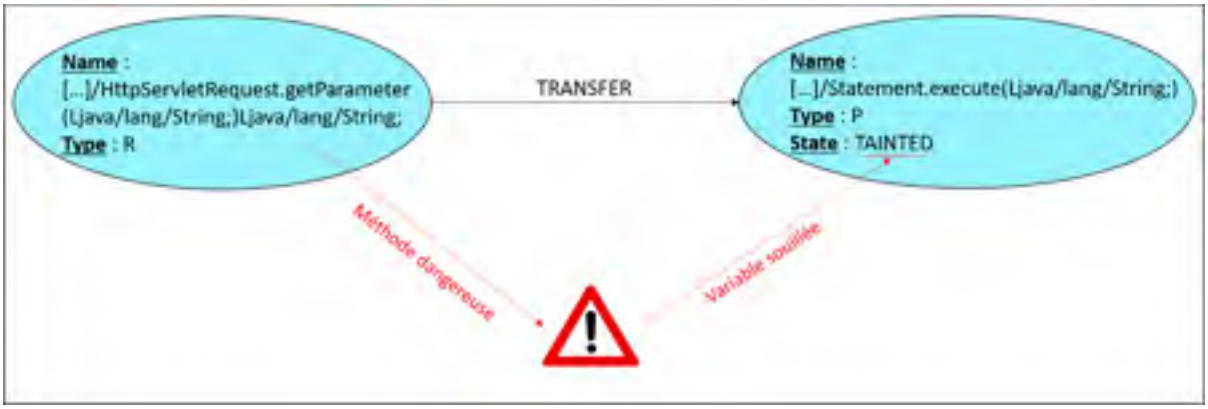

Figure 3.4 Exemple de graphe avec transfert d'état d'une variable

Grâce à cette représentation, traquer l'état d'une variable, pour affirmer ou non la vraisemblance d'une vulnérabilité, devient beaucoup moins complexe : en recherchant le nom de la méthode dont le code est jugé dangereux par l'outil d'analyse statique, on peut trouver les variables impliquées et vérifier leur état, qui aura été transmis par leurs parents dans le graphe. Ainsi, en reprenant l'exemple de la figure 3.2, il sera possible de savoir que dans le cas du second scénario, la variable pouvant causer une injection SQL est en réalité *SAFE*.

Pour conclure, analyser le graphe ainsi construit va permettre de simuler un suivi précis des variables : nous allons pouvoir extraire des caractéristiques, en lien avec l'état de ces dernières, qui vont être utiles à l'algorithme d'apprentissage machine, celui-ci ayant pour tâche d'éliminer les faux positifs. Ces caractéristiques et leur extraction seront précisées en section 3.3.3.

3.3 L'apprentissage machine

L'ajout d'apprentissage machine à l'analyse statique réalisée par Find Security Bugs est le point central, et la principale contribution apportée par ce mémoire. La section suivante définit et détaille avec précision la mise en place de ce processus.

3.3.1 Définition

Formellement, le terme apprentissage machine fait référence à l'apprentissage automatisé d'un modèle statistique : à partir de données, la machine tente de résoudre une tâche spécifique. Si les performances sur cette tâche s'améliorent avec les données d'entraînement, on dit que la machine apprend. L'aspect très systématique de ce processus permet de réaliser des tâches complexes ou redondantes, qu'un autre algorithme moyen ou un être humain auraient des difficultés à résoudre.

Concrètement, un problème d'apprentissage machine comporte toujours les éléments spécifiques suivant :

- des données;
- un algorithme d'apprentissage;
- une tâche à accomplir;
- une mesure des performances.

Cependant, deux types différents d'approche sont régulièrement utilisés : l'approche supervisée et l'approche non supervisée.

L'apprentissage supervisé est utilisé lorsque les classes à prédire sont connues. En d'autres termes, on sait ce que l'on cherche. Cette approche nécessite un jeu de données d'entraînement, qui ont été au préalable labélisées (par un expert par exemple). Le processus se passe en deux temps : la première phase, dite phase d'entraînement, consiste à entraîner l'algorithme d'apprentissage machine sur les données labélisées. Ainsi, celui-ci va trouver lui-même un lien entre les caractéristiques des données, et leur classe à prédire (classe qui est renseignée dans ces données d'entraînement). Les méthodes pour y parvenir sont diverses, et varient en fonction de l'algorithme utilisé. La seconde phase, dite phase de classification (ou test), consiste à donner à l'algorithme entraîné (ou modèle) une nouvelle donnée non labélisée : le but étant que celui-ci détermine automatiquement la classe à laquelle cette donnée appartient. Un schéma de cette approche supervisée est présenté en Figure 3.5:

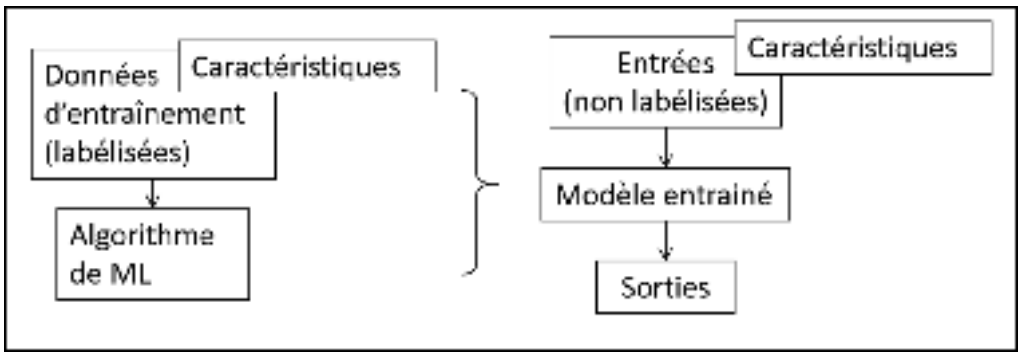

Figure 3.5 Schéma d'apprentissage supervisé

L'approche non supervisée, en revanche, est utilisée lorsque l'on ne sait pas réellement ce que l'on cherche : on parle de *clustering*. Concrètement, on dispose d'un ensemble de données, et l'algorithme va essayer par lui-même de diviser les données en groupes homogènes en fonction de leurs caractéristiques. Cela peut être très utile, car certaines ressemblances sont cachées, et l'ensemble de données peut être très grand, rendant complexe pour un humain la mise en évidence d'un regroupement efficace. Cependant, une fois les données groupées, ce sera à lui de trouver un sens à cette répartition, et éventuellement déterminer une classe pour chaque groupe.

Dans notre cas d'étude, nous utilisons une approche supervisée : le but est en effet de réaliser une classification des alertes levées par Find Security Bugs. Comme vu en section 2.1, celui-ci est stable : il permet de détecter efficacement la majeure partie des vulnérabilités, même si les résultats sont parasités par des faux positifs. Le but ici n'est donc pas d'améliorer la détection (on ne fait rien contre les faux négatifs, à savoir les vulnérabilités n'ayant pas été trouvées), mais de rendre les résultats plus cohérents en mettant de côté les fausses alertes. Ainsi, pour chaque alerte, le résultat de l'algorithme est une valeur discrète, une catégorie (Vraie vulnérabilité ou Faux positif). De plus, nous entraînons l'algorithme sur un jeu de données labélisées (détaillé en section 4.1.1).

3.3.2 Librairie Weka

Afin d'utiliser de l'apprentissage machine pour notre problème, nous avons dû trouver une implémentation accessible et utilisable de différents algorithmes adéquats. Pour cela, nous avons utilisé la librairie Weka (Weka, 1997).

Weka est un logiciel de source libre proposant une large collection d'algorithmes d'apprentissage machine, ainsi que des méthodes de visualisation. Il est entièrement implémenté en Java, ce qui est très pratique au niveau de la portabilité (il fonctionne sur tous les systèmes d'exploitation et plateformes actuels). De plus, il dispose d'une interface de

programmation (API) permettant de l'utiliser directement dans du code Java, ce qui en a fait l'outil idéal pour notre problème : Find Security Bugs étant aussi codé en Java.

Il propose également de nombreuses fonctionnalités et outils d'exploration et de prétraitement des données, de classification, de régression, et de regroupement (*clustering*). Malgré cela, il reste très simple d'utilisation : toutes les données considérées ne proviennent que d'un fichier de type CSV (*Comma-separated values*), où une ligne correspond à une instance avec ses différents attributs. Ces derniers peuvent d'ailleurs être filtrés ou modifiés à la guise de l'utilisateur, avant d'appliquer les algorithmes d'apprentissage machine, ce qui permet une flexibilité des tests de performance, et notamment la mise en évidence de l'importance de chaque attribut sur la classification globale. Par ailleurs, il dispose d'une interface graphique intuitive, composée de différents onglets en fonction de l'étape à laquelle l'utilisateur se trouve dans son problème : traitement préliminaire des données, sélection d'attributs, classification, visualisation, etc. Il est important de préciser que toutes ces options sont également accessibles via l'API directement dans du code Java.

Ainsi, la librairie Weka correspondait parfaitement à notre problème, aussi bien grâce à la diversité des algorithmes, et des méthodes de visualisation qu'elle propose, que grâce à son implémentation et interface de programmation qui s'intègrent parfaitement à notre cas d'étude.

3.3.3 Vecteurs de caractéristiques

3.3.3.1 Explications

Le vecteur de caractéristiques est, comme annoncé auparavant, l'élément le plus important d'un problème d'apprentissage machine.

Chaque instance est représentée par un certain nombre de caractéristiques, ou attributs. Ces caractéristiques seront les mêmes pour chaque donnée passée à l'algorithme, que ce soit lors de la phase d'entraînement ou de test (classification d'une instance non labélisée). En effet, l'algorithme les utilise pour construire son modèle : si une nouvelle donnée n'ayant pas les

bons attributs est envoyée vers l'algorithme entraîné, le modèle n'est plus valide et la classification ne sera pas possible. De la même façon, les caractéristiques doivent représenter précisément chaque instance : on parle de sous-apprentissage, dès lors que l'algorithme n'a pas assez d'informations pour réaliser une classification efficace.

Ainsi, pour une instance *a* donnée, le vecteur de caractéristique est de forme :

$$(X_{1a}, X_{2a}, \ldots, X_{na})$$

avec n le nombre total d'attributs, et X_{ia} la valeur de l'attribut X_i pour l'instance *a*. Il est important de préciser que si les valeurs de tous les attributs d'une instance lui sont uniques, et que ce phénomène se répète sur la quasi-totalité des données d'entraînement, un problème se posera : le surentraînement, ou *overfitting*. Cela se produit lorsque l'algorithme n'arrive pas à trouver de points communs entre les différentes instances, et ne fait en quelque sorte qu'apprendre par cœur. Il n'y a en réalité pas d'apprentissage à proprement parler, mais plutôt de la reconnaissance de signatures. Ainsi, l'algorithme n'arrivera pas à construire un modèle cohérent, et se retrouvera coincé lorsqu'une nouvelle donnée, aux valeurs de caractéristiques inconnues, lui sera fournie.

On voit donc qu'il est fondamental de choisir avec attention les caractéristiques sur lesquelles va se baser l'algorithme d'apprentissage machine. La suite de la section présente celles qui ont été choisies pour cette étude, ainsi que leur extraction lors du processus d'analyse du code.

3.3.3.2 Choix des caractéristiques

Comme dit précédemment, les caractéristiques choisies dans notre étude ont pour but de représenter au mieux des propriétés que le développeur irait récupérer (notamment grâce à son environnement de développement intégré) pour déterminer si une vulnérabilité est réelle ou non.

Ces caractéristiques sont au nombre de sept, et proviennent de trois catégories principales : la localisation de la vulnérabilité dans le code, la source ou provenance de la vulnérabilité, et les appels aux interfaces de programmation (API).

La **localisation de la vulnérabilité** dans le code, tout d'abord, peut être un indice non négligeable dans le processus de validation d'une vulnérabilité. Le code de la vaste majorité des applications modernes étant divisé en modules, il n'est pas inconcevable d'avancer que certains d'entre eux sont plus susceptibles d'héberger des vulnérabilités que d'autres. Un module contenant du code basé sur des fichiers statiques de confiance sera logiquement moins vulnérable que le module gérant le frontal de l'application (*front end*). De plus, les caractéristiques liées à cette catégorie, comme le module, le nom de classe, le répertoire, etc., sont évidentes et donc très faciles à extraire : c'est en effet ce qui saute immédiatement aux yeux du développeur lisant le rapport d'alertes. Néanmoins, ces caractéristiques ne sont pas suffisantes pour classifier les alertes de façon certaine.

La **source de la vulnérabilité** est ce que le développeur va investiguer dans un deuxième temps. Cela n'est en général pas aussi évident que pour les caractéristiques concernant la localisation, mais les informations qu'il va en tirer seront les plus importantes, et orienteront définitivement son jugement dans la plupart des cas. Concrètement, la source est l'endroit d'où provient le flux de données ayant mené à la vulnérabilité (potentielle). Bien souvent, le développeur va se demander d'où vient la variable passée en arguments de la méthode concernée. Un exemple simple est une requête SQL vers la base de données via la méthode *executeQuery(req)*. Dans ce cas, la variable *req*, correspondant à la requête, provient peut-être d'une autre classe, et est probablement composée d'entrées non filtrées saisies par l'utilisateur. Ou dans le cas contraire, la variable peut être totalement saine, et la vulnérabilité n'en est pas vraiment une. Ainsi, en apprendre plus sur la source (état des variables concernées, méthodes appelantes, etc.) est essentiel à la classification, et des caractéristiques concernant cette source doivent être extraites pour espérer avoir des résultats cohérents.

Enfin, les **appels à des interfaces de programmation** (API) particulières fournissent également des informations importantes au développeur. Find Security Bugs possède déjà dans sa base de données les signatures de différentes méthodes provenant d'API connues. Par exemple, certaines méthodes sont reconnues comme renvoyant des valeurs souillées (dangereuses), tandis que d'autres au contraire renvoient toujours une valeur saine. Cependant, il y a toujours des méthodes qui sont inconnues de l'outil d'analyse statique, car elles ne sont pas dans sa base de données, ou simplement, car elles proviennent d'APIs privées (codées par un développeur indépendant). Récupérer des caractéristiques sur ce genre de méthodes (parfois inconnues), impliquées dans les alertes, permettrait notamment à l'algorithme d'accroître sa précision si le développeur annonce leur faire confiance : autrement dit, s'il détermine qu'une alerte est un faux positif alors que le code concerné était un appel à une de ses propres méthodes (inconnues de Find Security Bugs), ce choix sera pris en compte par l'algorithme si elle est réutilisée par la suite. En effet, l'algorithme va s'entraîner sur des données qui vont être fournies par le développeur lui-même : ses choix se refléteront donc dans les résultats. De ce fait, extraire des caractéristiques concernant les appels à des APIs, même inconnues, nous a semblé être un atout non négligeable.

En prenant en compte les trois catégories ci-dessus, qui correspondent à ce qu'un développeur utiliserait pour classifier des alertes, nous avons choisi les caractéristiques présentées ci-après, au nombre de sept.

Type de Bogue : c'est la première qui vient à l'esprit, et ne peut pas être laissée de côté. Le type de la vulnérabilité est intuitivement en lien avec toutes les autres caractéristiques : en fonction de celui-ci, la source, ou bien les appels aux méthodes concernées par exemple, pourraient avoir un sens totalement différent. Au niveau de la forme, cette caractéristique prend une valeur nominale, correspondant à la vulnérabilité détectée (par exemple XSS_SERVLET).

Méthode source : cette caractéristique est relativement simple, et correspond logiquement à la signature de la méthode contenant la vulnérabilité. En Java, une méthode est une fonction, ou un sous-programme, contenant du code pouvant s'exécuter par simple appel à cette fonction

dans le corps du programme principal. La signature d'une méthode comprend alors le nom de sa classe, son nom à elle, et les types de ses différents paramètres. Ainsi, cet attribut donne également des informations sur la localisation de la vulnérabilité (le nom du package ou répertoire est présent dans le nom de classe).

Méthode *sink* : une méthode dite *sink* est une méthode possédant au moins un paramètre injectable, en d'autres termes qui peut causer une injection. Concrètement, cela va correspondre à des paramètres recevant des requêtes, des expressions de code, ou encore des chemins d'accès à un fichier. Si une méthode de ce type est présente dans le code de la vulnérabilité, la caractéristique aura pour valeur la signature de cette méthode. Or dans certains cas, la vulnérabilité ne correspond pas à une injection, mais simplement à l'utilisation d'un algorithme déprécié par exemple. Il n'y a donc pas de méthode *sink* liée à la vulnérabilité. De ce fait, la valeur de cet attribut sera une chaîne arbitraire constante, « *NO_VALUE* ». Il est en effet impossible de laisser le champ vide, car cela serait interprété par l'algorithme comme une valeur manquante, ce qui est faux : le fait qu'il n'y ait pas de méthode *sink* est une caractéristique en soi.

Source Inconnue : cet attribut renseigne sur le fait qu'il y ait eu ou non un appel à une méthode provenant d'une interface de programmation inconnue de l'outil d'analyse statique (comme dit précédemment, cela peut être une API privée, considérée comme bénigne par le développeur). Si un appel à une méthode de ce genre a lieu, la valeur de cet attribut sera la signature de la méthode en question. Dans le cas contraire, si aucune méthode inconnue de l'outil n'est appelée, l'attribut prendra pour valeur la « *NO_VALUE* » : comme pour la méthode *sink*, il est impératif de mettre une valeur constante plutôt que rien du tout, ce qui serait considéré comme une valeur manquante par l'algorithme.

Possède une source souillée : cet attribut est la plupart du temps une valeur binaire (*vrai* ou *faux*), en fonction de si le code de la vulnérabilité implique une donnée souillée qui se serait propagée, ou non. Cette propriété est extrêmement importante, et correspond directement à un suivi de variables pour en étudier la source. Dans la plupart des cas, si la vulnérabilité implique

une variable souillée (provenant directement de l'utilisateur), elle est confirmée. Cependant, il arrive que pour certains types de vulnérabilités, une analyse des variables ne soit pas nécessaire : on peut par exemple penser au cas d'utilisation de l'algorithme DES pour chiffrer les données (déprécié depuis un moment au profit de l'AES), ou à l'utilisation d'un mot de passe codé en dur dans le code. Dans les deux cas, la mauvaise pratique est évidente et ne nécessite pas un suivi d'une quelconque variable pour être vérifiée. La valeur de la caractéristique est alors la chaîne de caractère « *IRRELEVANT* », pour symboliser le fait que cette caractéristique n'est pas nécessaire dans ce cas spécifique.

Possède une source saine : cet attribut est, comme le précédent, souvent une valeur binaire (*vrai* ou *faux*), ou la chaîne « IRRELEVANT ». Cette fois-ci, on regarde s'il y a une donnée que l'on sait être saine avec certitude, impliquée dans la vulnérabilité. Ceci peut être détecté encore une fois grâce à un suivi de la variable : certaines méthodes sont connues pour renvoyer des valeurs toujours saines. Cependant, avoir une variable saine ne suffit pas à réfuter totalement la présence de la vulnérabilité.

Possède une source inconnue : tout comme les deux précédents, cet attribut peut prendre les valeurs *vrai*, *faux* ou « *IRRELEVANT* ». Certaines variables peuvent provenir directement d'une source que l'on ne connaît pas : on ne peut pas savoir directement si elles sont saines ou souillées. Il est par ailleurs important de préciser que cette caractéristique n'est pas directement liée avec Source Inconnue. En effet, cette dernière est extraite du rapport de Find Security Bugs (plus de détails dans la section suivante), et concerne juste le code de la vulnérabilité. La caractéristique dont il est question ici, en revanche, implique un suivi des variables au fil des classes et méthodes : elle peut donc avoir comme valeur *vrai*, alors que l'attribut Source Inconnue a pour valeur « *NO_VALUE* ».

C'est donc sur ce vecteur de caractéristiques que l'algorithme d'apprentissage machine va s'entraîner pour construire son modèle. Cependant, avant de parler des algorithmes en eux-mêmes, il est important de savoir comment ces caractéristiques ont été extraites.

3.3.3.3 Extraction des caractéristiques

Nous avons vu dans la partie précédente que les caractéristiques retenues étaient diverses, certaines étant plus facilement accessibles que d'autres. Il s'avère qu'en effet, elles ne sont pas toutes extraites de la même manière : une partie provient directement du rapport d'erreur XML de Find Security Bugs, tandis que l'autre partie résulte de l'analyse de graphe (dont la construction a été présentée précédemment en section 3.2).

Le rapport d'erreur de Find Security Bugs contient en effet de nombreuses informations, notamment celles liées à la localisation de la vulnérabilité (nom de la classe et méthode en question) ainsi que son type (cf. section 2.1.2). De plus, lorsque la vulnérabilité est directement un appel à une méthode *sink* ou à une méthode inconnue, caractéristiques importantes de notre modèle, le rapport le précise également. L'extraction de ces attributs n'est donc pas un problème, le format XML étant facilement manipulable.

En revanche, les trois autres caractéristiques (*Possède une source souillée/saine/inconnue*) n'ont pas pu être extraites aussi facilement. Il a en effet fallu trouver un moyen de simuler le suivi des variables que ferait un développeur effectuant de la revue de code grâce à son environnement de développement. Nous avons donc créé un graphe, dont chaque nœud correspond à une variable possédant différents attributs (nom, type, état). Les détails sont explicités en section 3.2. Ainsi, pour extraire les caractéristiques restantes, il suffit d'effectuer des requêtes sur le graphe. Le nom de la méthode contenant la vulnérabilité étant récupéré, les variables associées à celle-ci peuvent être tracées dans le graphe, afin d'examiner leur provenance. On peut donc voir si elles deviennent souillées à un moment de leur parcours, en les traçant depuis leur entrée, jusque dans la méthode *sink* par exemple. De cette manière, les caractéristiques concernant les sources (souillées, saines ou inconnues) peuvent être extraites pour chaque alerte, chose qui ne serait pas possible uniquement avec le rapport d'alertes de Find Security Bugs.

Une fois ces caractéristiques choisies et extraites, et avant de pouvoir commencer l'expérience, il ne manque plus qu'à s'intéresser aux algorithmes d'apprentissage machine en eux-mêmes.

3.3.4 Choix des différents algorithmes d'apprentissage machine

Il existe de nombreux algorithmes d'apprentissage automatique, tous fonctionnant d'une manière différente, et donc apportant des résultats différents. Aussi, il nous a paru nécessaire de ne pas se restreindre à un seul d'entre eux, mais bien d'en tester plusieurs afin d'évaluer quels types étaient les plus performants, et de voir s'il en existait un bien meilleur que les autres, considérant les caractéristiques décrites précédemment.

Les algorithmes évalués ici sont relativement classiques, et appartiennent à différentes catégories : modèle graphique (arbres), modèle probabiliste, modèle linéaire, etc. Tous sont implémentés en Java dans la librairie Weka (explicitée ci-dessus) : ils ont donc été facilement intégrables au sein du contexte. Le fonctionnement de chacun est explicité ci-dessous, sans pour autant trop rentrer dans la complexité mathématique de leur implémentation.

3.3.4.1 Classification naïve bayésienne

L'algorithme bayésien naïf est un classifieur probabiliste. Cela signifie qu'il va déterminer une probabilité pour chaque classe à laquelle une instance peut appartenir, ici *vraie vulnérabilité* ou *faux positif*. La somme des probabilités pour chaque classe sera égale à 1, et l'algorithme classifiera logiquement l'instance dans la classe ayant la plus haute probabilité. Pour réaliser ce calcul de probabilité, l'algorithme va se baser sur le théorème de Bayes, en faisant la supposition que les caractéristiques sont indépendantes les unes des autres.

En réalité, cela est rarement le cas, d'où le terme « naïf ». C'est d'ailleurs la principale limitation de cet algorithme, qui ne reflète pas vraiment la réalité dans ses calculs. Néanmoins, les résultats sont en général assez surprenants. Cette efficacité inattendue est notamment due à l'application de la règle du « maximum a posteriori », à savoir que la classe ayant la plus grande probabilité sera choisie. Les probabilités n'ont donc pas à être extrêmement précises

pour permettre une bonne classification, notamment dans le cas d'une classification binaire : la classe adéquate a juste à avoir une probabilité supérieure à l'autre, et non une probabilité précise.

3.3.4.2 K plus proches voisins

L'algorithme des K plus proches voisins est une des méthodes d'apprentissage les plus connues, également probabiliste. Ce classifieur réalise un apprentissage basé sur la mémoire : il ne va pas effectuer une généralisation explicite des données d'entraînement afin de déterminer une loi permettant de classifier une nouvelle instance, mais va directement comparer cette dernière avec celles vues lors de la phase d'entraînement.

Concrètement, l'algorithme va assigner à une instance la classe de la majorité de ses K plus proches voisins dans l'espace. Cet espace dimensionnel est construit à partir du nombre de caractéristiques présent dans le vecteur (on parle d'espace vectoriel). Pour savoir quels voisins sont les plus proches, il est nécessaire d'avoir une mesure de distance : dans le cas classique, la simple distance euclidienne est utilisée (plus court chemin d'un point A à un point B dans un espace). Enfin, le nombre de voisins choisis est un paramètre de l'algorithme : il n'y a pas de loi permettant de trouver la valeur K optimale, celle-ci est souvent dépendante du problème, et est trouvée par validation croisée (qui sera explicitée en section 4.1.2). Dans le cas où K=1, l'instance à classifier se verra attribuée la classe de son voisin le plus proche.

Les résultats de cet algorithme peuvent être variables en fonction de l'étude, mais c'est un algorithme très classique, et simple d'utilisation et de compréhension : c'est pourquoi nous avons décidé de l'évaluer ici.

3.3.4.3 Arbre de décision (C4.5)

L'algorithme C4.5 est utilisé pour générer un arbre de décision. Il est considéré comme étant l'un des algorithmes d'arbres les plus utilisés à l'heure actuelle. Son fonctionnement est relativement simple : il va considérer l'ensemble des données d'entraînement pour construire

un arbre unique, et chaque nouvelle instance n'aura qu'à effectuer un trajet dans l'arbre correspondant à ces caractéristiques pour être classifié.

Plus précisément, la construction de l'arbre est réalisée comme suit : à chaque nœud (intersection), l'algorithme choisit l'attribut qui divise au mieux le jeu de données d'entraînement, considérant les différentes classes possibles. Pour déterminer l'attribut ayant le plus d'influence, il va se baser sur le *ratio de gain d'information*. Cette mesure est liée à la notion d'entropie : l'entropie permet de déterminer, en fonction d'une distribution de probabilités, la quantité d'information nécessaire pour prévoir un évènement. En l'occurrence, on s'intéresse ici à la probabilité qu'une instance appartienne à une classe en fonction de la valeur d'un attribut : c'est le gain d'information. Cependant, ce dernier ne suffit pas : *le ratio de gain d'information* permet de prendre en considération le nombre de valeurs possibles pour chaque attribut. En effet, si un attribut possède énormément de valeurs (une pour chaque instance dans le cas extrême), la probabilité qu'une instance appartienne à une classe pour telle valeur de l'attribut est très élevée (et même égale à 1 dans le cas extrême) : le gain d'information sera très élevé, sans pour autant que l'attribut ne soit très utile, d'où la nécessité d'utiliser le ratio. Un exemple illustré de cette mesure, et comment la calculer, est explicité dans le chapitre 4 de la documentation de Weka (WEKA Machine Learning, 2016).

Chaque nœud sera ainsi traité de cette manière, et les feuilles (tout en bas de l'arbre) correspondront à une classe. Enfin, une dernière étape est effectuée : les branches qui n'apportent aucun intérêt à la classification sont taillées (retirées de l'arbre) et remplacées par des feuilles : c'est le *pruning*. Cela permet de réduire le temps et coût de calcul et gagner en performance, sans impacter les résultats (un exemple étant une caractéristique inutile qui n'apporte aucune information).

3.3.4.4 Forêt aléatoire

La classification par forêt aléatoire (ou forêt d'arbres décisionnels) est probablement la plus populaire, car apportant généralement de bons résultats, quel que soit le contexte d'étude. L'algorithme va créer des arbres de décisions à partir du jeu de données d'entraînement, et l'ensemble de ces arbres vont être utilisés pour réaliser la classification.

Chaque arbre créé ne va s'intéresser qu'à un sous ensemble du jeu de données. Plus précisément, chaque sous-ensemble va être créé grâce à un tirage avec remise (ou *bootstrap*) : en général, la taille de chaque sous-ensemble sera égale à celle du jeu de données d'entraînement au complet, mais pas sa composition, de par le tirage avec remise qui autorise l'algorithme à piocher plusieurs fois la même instance. Ainsi, chaque arbre sera construit sur un sous-ensemble différent du jeu de données d'entraînement.

Par la suite, la forêt va être créée de la manière suivante : pour chaque nœud de chaque arbre, l'algorithme va sélectionner un échantillon aléatoire d'attributs parmi ceux du vecteur de caractéristiques, et divisera les données suivant le meilleur des attributs de cet échantillon, en utilisant la même mesure que le C4.5 (le ratio de gain d'information). Le nombre d'attributs à choisir aléatoirement pour chaque nœud est paramétrable, mais reste constant pour l'ensemble des arbres. Lorsqu'une nouvelle instance est passée dans le modèle, elle traverse l'ensemble des arbres, et la classe retenue sera celle présente chez la majorité des arbres (pour l'instance en question).

Cet algorithme est très polyvalent, et apporte en général des résultats supérieurs à ceux d'autres algorithmes d'arbres de décision. De plus, l'une des particularités de la classification par forêt aléatoire est que celle-ci est très résistante au surapprentissage (*overfitting*). Cela est dû au fait que contrairement à un simple arbre de décision, qui en gagnant en profondeur va apprendre des modèles de caractéristiques très irréguliers, la forêt aléatoire va faire une moyenne de différents arbres de décisions, entraînés sur des parties différentes du jeu de données : le risque d'erreur va être légèrement augmenté, mais la variance entre les instances va être diminuée, et

le modèle général en bénéficiera. Les performances seront en effet améliorées, en comparaison à un modèle avec une grande variance entre les instances : celles-ci seront trop uniques pour l'algorithme, qui sera perdu lorsqu'il en rencontrera une nouvelle.

De fait, la classification par forêt d'arbres décisionnels était impérativement à tester pour notre étude.

3.3.4.5 SVM

L'algorithme SVM, pour Machines à Vecteur de Support, est un classifieur linéaire binaire. En effet, il n'est utile que dans les problèmes où il n'y a que deux classes possibles, en faisant une méthode de classification de choix pour notre étude.

Son fonctionnement est relativement simple à comprendre : le principe est de représenter chaque instance du jeu de données d'entraînement dans un espace de dimension n (avec n le nombre d'attributs). Par la suite, l'algorithme va tenter de trouver un hyperplan de cet espace séparant au mieux les deux classes d'éléments, avec une marge la plus grande possible. Un hyperplan à un espace de dimension n est un espace de dimension n-1 : par exemple, si les données ont deux caractéristiques, l'algorithme SVM va représenter virtuellement ces données dans un plan (espace de dimension 2), puis va essayer de trouver une droite (espace de dimension 1) séparant les données. La Figure 3.6 illustre cet exemple : le schéma de droite montre une séparation non optimale (petite marge), tandis que le schéma de gauche montre la séparation optimale. Dans les deux cas, la droite pointillée correspond à l'hyperplan.

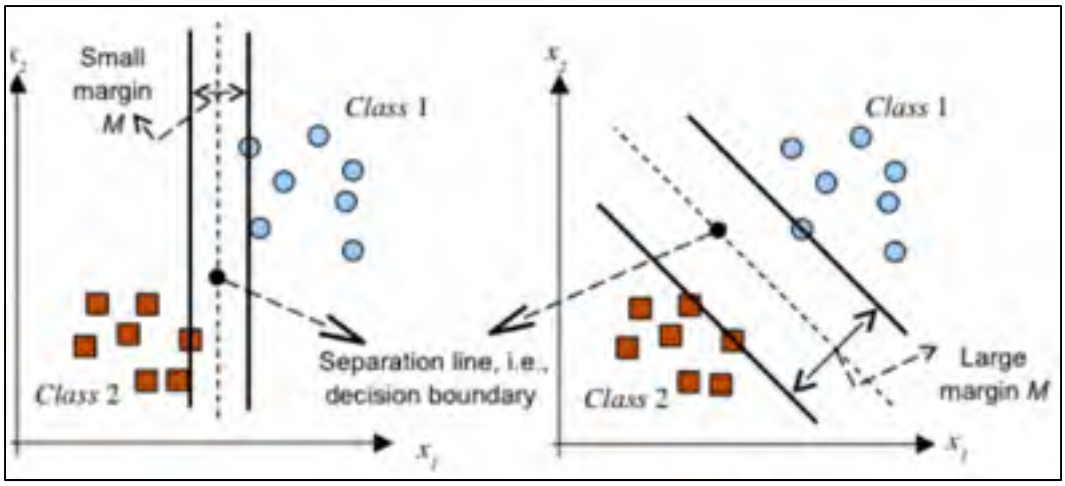

Figure 3.6 Fonctionnement de l'algorithme SVM
Adaptée de (Analytics, 2011)

Ainsi, lorsqu'une nouvelle donnée va être passée dans le modèle, sa position dans le plan par rapport à la séparation déterminera directement la classe à laquelle elle appartient. Il est par ailleurs important de noter que la séparation choisie par l'algorithme va dépendre directement du type de noyau utilisé par celui-ci : ce paramètre est central dans la conception d'un modèle d'apprentissage basé sur SVM. Concrètement, celui-ci peut être linéaire, polynomial ou Gaussien. La différence va venir de la façon dont la séparation va être tracée pour séparer au mieux les données : en prenant l'exemple d'un problème en dimension 2, un noyau linéaire va correspondre à la figure ci-dessus (la séparation sera une droite), un noyau polynomial pourra tracer une courbe (de type parabole) qui s'affinera avec le degré associé au polynôme, tandis qu'un noyau Gaussien pourra optimiser la séparation en créant des sous-espaces englobant une classe spécifique. Cependant, les deux derniers types de noyau évoqués ont en général un temps de calcul très long, contrairement au noyau linéaire, c'est pourquoi nous avons choisi de conserver celui-ci pour notre problème (le nombre de caractéristiques étant de plus peu élevé, nous sommes restés sur un modèle simple).

Par ailleurs, il est également possible de paramétrer la complexité C de l'algorithme : plus cette valeur sera haute, plus l'algorithme cherchera un hyperplan séparant au mieux l'ensemble des données, au détriment de la marge. Il n'existe pas de valeur optimale pour ce paramètre, car

une plus haute complexité n'impliquera pas forcément une meilleure classification de nouvelles données, la marge étant réduite : en revanche, les données d'entraînement seront mieux décrites. En reprenant la Figure 3.6, le modèle de gauche a une plus grande complexité que son homologue de droite.

En conclusion, cet algorithme est prouvé comme étant très efficace : c'est pourquoi nous avons choisi de l'évaluer avec les autres dans notre étude.

3.4 Intégration de l'apprentissage machine à Find Security Bugs

Avant de passer à l'évaluation de notre approche (présente dans le chapitre suivant), il convient de présenter une rapide vue d'ensemble de l'intégration de l'apprentissage machine au sein du contexte existant, principalement Find Security Bugs. Nous ne rentrerons pas ici dans les détails du code, mais allons plutôt nous concentrer sur la façon dont le développeur va pouvoir utiliser notre modèle, et surtout les informations qu'il va pouvoir en retirer.

Find Security Bugs est relativement simple d'utilisation pour un développeur : il peut être intégré directement à l'environnement de développement sous la forme d'une extension, ou il peut être utilisé en ligne de commande avec des outils de gestion de production de projets comme Maven. Concrètement, à l'issue de l'analyse statique, un fichier XML contenant la totalité des alertes est généré, appelé *findbugs-result.xml* (cf. section 2.1.2). Ces alertes peuvent apparaître dans l'IDE, ou le développeur peut choisir de les gérer grâce à d'autres outils, comme SonarQube (cf. section 2.2). Dans tous les cas, ce rapport est la seule information qui est fournie au développeur, celui-ci devant ensuite effectuer le triage des alertes.

Avec l'ajout du triage automatique basé sur l'apprentissage machine, notre approche n'est pas d'effectuer seulement cette tâche sans renvoyer de retour au développeur : au contraire, le but est de le tenir au courant des différentes étapes du processus, des alertes qui ont été triées, des résultats immédiats estimés de l'algorithme, etc. De cette manière, il a toutes les informations

pour juger s'il veut faire confiance au triage automatique ou s'il préfère trier lui-même les alertes : il est en effet possible qu'avec une application jeune, peu d'alertes ressortent de l'analyse statique, et donc que les résultats de l'algorithme d'apprentissage machine ne soient pas assez bons, faute de données. Par ailleurs, l'utilisation de notre solution est simple, et très semblable à Find Security Bugs : tout s'exécute simplement en ligne de commande. La figure 3.7 ci-après présente une vue d'ensemble des différents fichiers et informations que le développeur pourra consulter à la suite de l'utilisation de celle-ci.

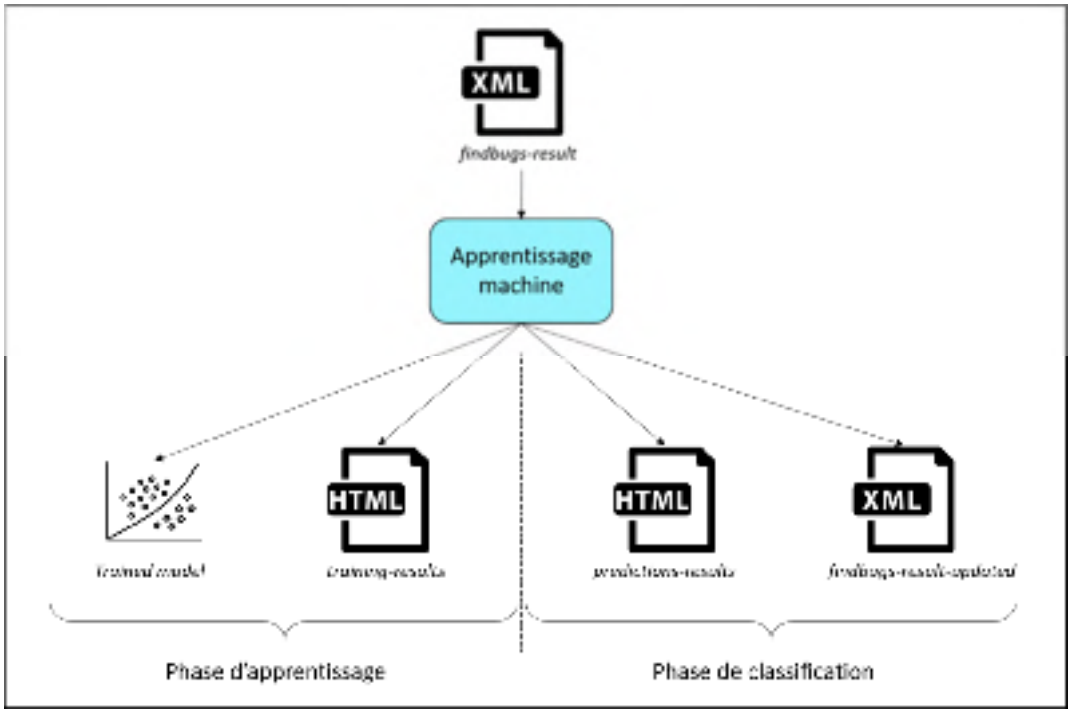

Figure 3.7 Informations de retour de l'apprentissage machine

Les quatre éléments apportés (en bas de la figure) sont décrits dans les paragraphes suivants.

Tout d'abord, il y a le modèle entraîné : lorsque l'algorithme a construit son modèle, qu'il s'est donc entraîné sur les données qu'on lui a fournies, ce modèle est sauvegardé. Cela permet notamment au développeur de pouvoir le réutiliser plus tard sur de nouvelles alertes, sans avoir à entraîner l'algorithme à nouveau.

De plus, un résumé d'évaluation de l'algorithme est fourni : à l'issue de la phase d'entraînement, une synthèse des principales mesures d'évaluation de l'algorithme est sauvegardée au format HTML (*training-results.html*). Ces mesures d'évaluation seront détaillées dans le chapitre suivant. Avoir ces données permet notamment au développeur de comprendre si l'algorithme fonctionne bien et s'il peut faire confiance au modèle concernant le triage automatique.

Ensuite, une liste d'alertes prioritaires va être enregistrée, au format HTML également *(predictions-results.html*), à l'issue de la phase de classification cette fois-ci. Cette liste est créée de la façon suivante : lorsque l'algorithme choisit une classe pour une alerte, il se base sur un score de confiance (entre 0 et 1). En fonction de son type, ce score est calculé différemment : par exemple, si c'est un algorithme probabiliste, ce score sera la probabilité que l'alerte appartienne à la classe choisie. De ce fait, l'utilisateur peut paramétrer un seuil à partir duquel il choisira de ne plus faire totalement confiance à l'algorithme : si une alerte est classifiée avec un score de confiance inférieur à ce seuil, alors elle finira dans cette liste (ce seuil étant réglé par défaut à 0,85). Il est important de préciser que celle-ci ne contient pas les alertes les plus graves, mais seulement celles que l'algorithme a eu plus de mal à classifier. Par la suite, libre au développeur d'agir comme il l'entend. Une manière intéressante de procéder est par exemple de classifier à la main les alertes dont l'algorithme n'est pas certain, puis d'entraîner à nouveau celui-ci. Ainsi, il y a un retour sur l'apprentissage, et l'algorithme peut apprendre de ces erreurs.

Enfin, un rapport d'alertes mises à jour, au format XML, sera également fourni au développeur (*findbugs-result-updated.xml*). Ce rapport sera le même que son homologue résultant de l'analyse statique, à l'exception du fait qu'il sera allégé des faux positifs que l'algorithme d'apprentissage machine a détecté. Le seuil de confiance défini par le développeur est également pris en compte ici : si l'algorithme juge une alerte comme étant un faux positif, mais avec un score inférieur au seuil défini, cette alerte ne sera pas supprimée du rapport. Le principe

est en effet d'éliminer les faux positifs sans impacter sur la détection des vulnérabilités réelles, auquel cas notre solution n'aurait que peu d'intérêts.

Ainsi, notre approche se veut être la plus transparente possible pour le développeur effectuant sa revue de code. Celui-ci reste donc très libre de faire ce qu'il juge être le mieux, en fonction des informations qui lui sont données. Il peut faire confiance à l'algorithme, il peut trier une partie des alertes à la main afin d'améliorer le modèle, ou même ne pas du tout faire confiance au triage automatique : le rapport initial issu de l'analyse statique n'est en aucun cas supprimé.

Pour conclure, notre solution s'intègre parfaitement avec Find Security Bugs : il n'y a pas de complexité supplémentaire liée à son utilisation, et le développeur est assez libre de faire ce qu'il veut. Désormais, il ne reste plus qu'à évaluer son efficacité : c'est ce qui est présenté dans le chapitre suivant.

CHAPITRE 4

ÉVALUATION, ANALYSE DES RÉSULTATS ET DISCUSSIONS

4.1 Évaluation des différents algorithmes

Nous avons présenté, dans le chapitre précédent, la construction de notre modèle : choix des différentes caractéristiques, extractions de celles-ci par analyse de graphe (principalement), choix d'algorithmes d'apprentissage automatique pertinents, et intégration au sein du contexte (analyse statique via Find Security Bugs).

Ce chapitre est, de son côté, entièrement centré sur l'évaluation de ce modèle. Nous nous poserons ici les questions suivantes : sur quelles données pouvons-nous entraîner nos algorithmes ? Par quels moyens ou mesures pouvons-nous évaluer leur efficacité ? Est-ce que l'un d'entre eux est supérieur aux autres dans notre cas d'étude ? Et enfin, tout simplement, est-ce que le modèle proposé ici est viable ?

Le but de cette évaluation est de savoir si notre solution peut réellement être utilisé par un développeur effectuant de la revue de code, pour lui faire gagner du temps, et ce sans prendre le risque de passer à côté de vulnérabilités graves.

4.1.1 Jeu de données d'entraînement : la suite de test Juliet

Le choix du jeu de données d'entraînement pour les algorithmes n'est pas une chose simple, tout du moins dans notre domaine. En effet, il doit répondre à deux critères :

- les données doivent évidemment être labélisées. Dans notre cas, les vulnérabilités doivent déjà avoir été triées (vulnérabilité réelle ou faux positif);
- il doit y avoir un nombre suffisamment élevé de données pour ne pas être en sous-apprentissage, et avoir des résultats corrects.

Ces deux critères ne semblent pas insurmontables, mais réussir à avoir les deux en même temps est une tâche difficile. La raison principale est que trier des alertes à la main est très long pour une personne effectuant de la revue de code, et un grand nombre de données (de vulnérabilités) signifie analyser de grands projets, et donc y passer énormément de temps. De plus, l'erreur humaine est un facteur non négligeable : à la longue, certaines vulnérabilités vont être mal triées (pour cause de fatigue, redondance, etc.), or les données d'entraînement se doivent d'être parfaites, sans quoi le modèle résultant ne sera pas précis : c'est la règle du « garbage in, garbage out ».

Ainsi, il nous a fallu trouver un compromis entre jeu de données suffisamment grand, et données étiquetées avec soin. Nous avons tout d'abord essayé avec des projets moyens, contenant environ une centaine de vulnérabilités, telle que WebGoat, une application volontairement vulnérable proposée par OWASP afin de tester des outils de pénétration (WebGoat, 2016). Ces vulnérabilités ont été triées à la main avec soin, mais les résultats n'ont pas été concluants : il n'y avait visiblement pas assez de données pour que l'algorithme d'apprentissage machine puisse construire un modèle efficace. Le temps nous empêchant de traiter manuellement des milliers de vulnérabilités venant d'autres projets, nous nous sommes tournés vers une autre solution : Juliet.

Juliet (Boland Jr & Black, 2012) est une suite de test composée d'une collection importante de programmes Java ou C++ contenant des vulnérabilités. Au total, la partie Java comporte environ 24 000 cas de potentielles vulnérabilités, de tous types. Mise à disposition par le NIST (National Institute of Standards and Technology), ce code est entièrement de source libre, et est spécialement conçu pour tester ses outils de détection. En effet, toutes les instances (vulnérabilités réelles et faux positifs) sont déjà étiquetées.

Le fonctionnement de Juliet est très simple : chaque cas de test contient une ou plusieurs classes Java, et se concentre uniquement sur une vulnérabilité, indiquée dans le nom de la classe (et du répertoire). De plus, les méthodes présentes dans ces classes ont toujours un préfixe *good* ou *bad*, indiquant si le flux de données est malicieux ou non. En d'autres termes, lorsque l'outil

d'analyse statique détecte une vulnérabilité, il suffit de vérifier que le type de celle-ci soit bien en accord avec le répertoire dans lequel se trouve la classe en question, et si oui, il suffit de jeter un œil à la méthode contenant la vulnérabilité : si elle a un préfixe *bad*, c'est un cas réel, si elle a un préfixe *good*, c'est un faux positif.

Ainsi, un jeu de données d'entraînement suffisamment grand et correctement étiqueté a été obtenu : après avoir analysé Juliet avec Find Security Bugs, il a fallu conserver uniquement les alertes correspondant bien aux cas de test. En effet, l'outil détectait parfois des vulnérabilités ne correspondant pas au type testé dans une certaine classe : par exemple, un code malicieux correspondant à un cas d'injection SQL (cette information étant présente dans le nom de la classe et du package) pouvait faire ressortir des alertes d'autres types que celui-ci. Ne pouvant pas déterminer avec certitude si ces alertes *parasites* étaient vraies ou fausses (le code en question se concentrant sur un seul type de vulnérabilité), nous les avons mises de côté. Par la suite, labéliser les alertes correspondant au bon type testé par chaque classe n'a pas été difficile : il a suffi de regarder le nom de la méthode pour déterminer si la vulnérabilité était réelle ou non. Au total, environ 10 000 vulnérabilités potentielles ont pu être étiquetées. Le Tableau 4.1 indique la distribution de celles-ci en fonction de leur type. Chaque type est explicité en détail, avec exemples, dans la documentation de Find Security Bugs (FSB Bugs Patterns, 2017).

Cette suite de test a donc été d'une grande aide pour établir un jeu de données d'entraînement correct. Néanmoins, cela ne vient pas sans certains inconvénients : même si les classes de test de Juliet tentent de se rapprocher au mieux de la réalité, le code est principalement autogénéré, et est donc différent d'un projet réel. Sa construction est très systématique, et cela peut avoir des impacts sur le modèle global et les résultats. Pour plus de précisions, une discussion concernant ses limitations est en section 4.2.1.

Tableau 4.1 Distribution des vulnérabilités potentielles par type

Nombre	**Type de vulnérabilité**
3240	SQL_INJECTION_JDBC
1944	HTTP_RESPONSE_SPLITTING
1872	XSS_SERVLET
648	XPATH_INJECTION
624	LDAP_INJECTION
624	COMMAND_INJECTION
468	UNVALIDATED_REDIRECT
358	UNENCRYPTED_SOCKET
270	UNENCRYPTED_SERVER_SOCKET
98	ECB_MODE
51	HARD_CODE_PASSWORD
34	DES_USAGE
34	PREDICTABLE_RANDOM
34	WEAK_MESSAGE_DIGEST_MD5
19	PT_RELATIVE_PATH_TRAVERSAL
17	HARD_CODE_KEY
17	WEAK_MESSAGE_DIGEST_SHA1
16	PT_ABSOLUTE_PATH_TRAVERSAL
16	STATIC_IV
15	DMI_CONSTANT_DB_PASSWORD

4.1.2 Méthodes et mesures d'évaluation

Un classifieur est logiquement évalué sur sa capacité à bien classifier les instances qui lui sont données. Néanmoins, cette mesure, bien que représentative de la performance globale de l'algorithme, ne permet pas à elle seule d'évaluer correctement le modèle.

Par exemple, prenons le cas d'un ensemble de données très hétérogène au niveau des classes : 90% de vraies vulnérabilités et 10% de faux positifs. Un algorithme très mauvais, classant l'intégralité des instances comme étant des vraies vulnérabilités, aura dans ce cas précis correctement classifié 90% des instances, ce qui est un bon score. Néanmoins, le modèle n'est pas bon : si les proportions étaient inversées, il obtiendrait un score très mauvais.

Ainsi, différentes mesures sont généralement utilisées pour évaluer les performances d'un algorithme. Elles dépendent toutes des quatre éléments suivants, qui ont été expliqués dans la section 2.1.3 :

- nombre de vrais positifs (TP),
- nombre de faux positifs (FP),
- nombre de vrais négatifs (TN),
- nombre de faux négatifs (FN).

Nous avons choisi quatre d'entre elles pour notre étude, chacune permettant d'évaluer un aspect différent des performances de l'algorithme.

- **L'exactitude** (ou *Accuracy*) : c'est la mesure la plus classique, décrite au-dessus, qui correspond au pourcentage d'instances classifiées correctement par l'algorithme. Sa formule est la suivante : $\frac{TP+TN}{TP+FP+TN+FN}$
- **Le rappel** : il mesure la proportion de vraies vulnérabilités qui ont été *classifiées comme telles* par l'algorithme. Ainsi, si sa valeur est égale à 1, cela signifie que toutes les vraies vulnérabilités ont été correctement classifiées. Sa formule est la suivante : $\frac{TP}{TP+FN}$

- **La précision** : elle mesure quant à elle la proportion d'instances classifiées comme vraies vulnérabilités qui *sont de vraies vulnérabilités*. Autrement dit, la précision va donner une indication sur le nombre de faux positifs (attention, on parle ici des faux positifs de l'algorithme d'apprentissage machine, et non des « fausses vulnérabilités » ressorties par l'outil d'analyse statique). Si sa valeur est égale à 1, cela signifie que l'algorithme n'a fait aucun faux positif : toutes les instances classifiées comme vulnérabilités le sont réellement. Sa formule est la suivante : $\frac{TP}{TP+FP}$
- **F-mesure** : cette mesure est la *moyenne harmonique* entre le rappel et la précision. De ce fait, elle prend en considération à la fois les faux positifs et les faux négatifs. Elle est donc très utilisée, car c'est une bonne alternative à l'exactitude, tout en étant mieux adapté à une distribution inégale des classes. Notre jeu de données d'entraînement contient environ deux fois plus de vraies vulnérabilités que de fausses alertes (distribution deux tiers/un tiers), donc cette mesure sera intéressante. Sa formule est la suivante : $2 \times \frac{(Rappel \times Précision)}{(Rappel + Précision)}$

Ces quatre mesures permettront donc une évaluation assez complète des performances du modèle créé. Les attentes sont doubles : même s'il paraît logique d'espérer les meilleurs scores possible pour toutes les mesures, avoir un bon rappel et une bonne précision est essentiel dans notre étude. Pour le premier, il ne faut pas que des vraies vulnérabilités soient mal classifiées par l'algorithme, car cela pourrait avoir des conséquences graves pour l'application si l'une d'entre elles est exploitée, car non corrigée. Pour la seconde, le but même de l'étude est de réduire le nombre de faux positifs lors de l'analyse statique : si l'apprentissage machine apporte lui aussi des faux positifs, on ne résout pas grand-chose au final.

Ayant sélectionné les mesures d'évaluation, la question est ensuite de savoir comment pouvons-nous récupérer les éléments nécessaires à leur calcul, à savoir le nombre de vrais et faux positifs, etc. Autrement dit, sur quelles données allons-nous donc tester nos algorithmes d'apprentissage machine ? Seul notre jeu de données d'entraînement est labélisé.

Pour répondre à ce problème, l'idée est de réaliser une *validation croisée* (à dix échantillons). Cela consiste à diviser aléatoirement le jeu de données en dix sous-ensembles, et à en utiliser neuf pour entraîner l'algorithme, et le dernier pour le tester. On réitère cette opération neuf autres fois, afin que chaque sous-ensemble ait servi d'entraînement et de tests. À l'issue de cette validation croisée, on aura donc les éléments nécessaires au calcul des différentes mesures précédentes, et le modèle peut être entraîné sur l'ensemble des données. Par ailleurs, il est important de noter que même si l'algorithme s'entraîne et teste sur le même jeu de données, il n'essaye pas de classifier une instance qu'il a utilisée pour s'entraîner (ce qui apporterait un biais aux résultats), car le modèle est réinitialisé à chaque tour. Ainsi, cette méthode est très largement utilisée pour mesurer les performances d'un algorithme sans avoir à créer un ensemble de données spécialement pour le test.

La méthodologie d'évaluation étant mise en place, il ne reste plus qu'à lancer les algorithmes et observer les résultats.

4.1.3 Analyse des résultats

4.1.3.1 Résultats de classification par les algorithmes

Avant de détailler les résultats, il est important de parler des valeurs choisies pour les différents paramètres évoqués lors de la description des algorithmes (section 3.3.4). Tout d'abord, la classification bayésienne et l'arbre de décision ont été utilisés directement depuis Weka sans modification : ces deux algorithmes étant très simples, aucun paramètre n'influence réellement la classification (à l'exception peut-être de la suppression du *pruning* pour le C4.5, mais qui a été évidemment conservée ici, car faisant la particularité de cet arbre de décision). Pour l'algorithme des K plus proches voisins, la valeur K=2 a donné les résultats les plus intéressants après validation croisée : nous avons donc choisi de conserver cette valeur.

De son côté, l'algorithme de forêts d'arbres décisionnels a été le plus complexe, car il possède le plus de paramètres : premièrement, le nombre d'arbres de la forêt a été conservé à 100 (valeur par défaut dans Weka). Augmenter ce nombre n'a en effet pas eu d'influence sur les

résultats, à l'exception du temps de calcul légèrement plus long. Ensuite, la taille de chaque sous-ensemble a été également conservée à sa valeur par défaut, à savoir 100% du jeu de données d'entraînement : cela permet de considérer le maximum d'instances lors de l'entraînement. Enfin, le nombre d'attributs tirés aléatoirement à considérer pour chaque nœud a été fixé à 2 : il est en effet conseillé que cette valeur soit inférieure à la racine carrée du nombre d'attributs (Breiman, 2001).

Pour finir, le choix de la complexité liée au noyau linéaire de l'algorithme SVM a été fixé à 1. En règle général, cette valeur est comprise entre 10^{-6} et 10^{3} en fonction des implémentations : plus elle est petite, plus la marge sera grande, mais moins l'hyperplan séparera les deux classes de façon précise. Ne pouvant pas prévoir à l'avance le comportement du jeu de données, nous avons réalisé des tests par validation croisée, et avons abouti à une valeur de 1 pour la complexité, moyennant un compromis entre la précision de l'hyperplan, la grandeur de marge, et le temps de calcul de l'algorithme.

Ainsi, ces cinq algorithmes, se basant sur les caractéristiques détaillées dans le chapitre 3, ont été testés sur le jeu de données récupéré grâce à la suite de test Juliet. Au total, celui-ci contient 7098 vraies vulnérabilités, et 3302 faux positifs. Une validation croisée à 10 échantillons a donné les mesures suivantes :

Tableau 4.2 Résultats d'évaluation des cinq algorithmes choisis

	Rappel	**Précision**	**F-mesure**	**Exactitude (%)**
CNB	0,852	0,919	0,884	84.73
K-PPV (k = 2)	0,860	0,911	0,885	84.70
Arbre de décision	0,883	0,876	0,892	85.68
Forêt aléatoire	0,808	0,913	0,858	81.66
SVM	0,821	0,916	0,866	82.62

Avant de s'intéresser directement aux valeurs des différentes mesures pour chaque algorithme, il est nécessaire de comprendre que celles-ci correspondent à la moyenne de ces mesures pour les 10 itérations de la validation croisée. Afin de vérifier si cette moyenne est représentative de notre jeu de données, à savoir si celui-ci est suffisamment homogène pour que les résultats ne varient que très peu en fonction des données choisies pour s'entrainer et pour tester, nous avons calculé la variance entre les valeurs des 10 itérations, pour chaque mesure et chaque algorithme. En l'occurrence, celle-ci a toujours été extrêmement faible, et ce pour chaque mesure : de l'ordre de 10^{-5}. Ainsi, nos valeurs moyennes présentes dans le tableau sont donc validées, et une interprétation peut en être faite.

En s'intéressant désormais aux valeurs affichées, le Tableau 4.2 présente à première vue des résultats relativement similaires pour les différents algorithmes. L'exactitude est autour de 83%, ce qui signifie qu'un peu plus de quatre instances sur cinq sont correctement classifiées. La F-mesure est quant à elle légèrement plus élevée, ce qui est de bonne augure pour le rappel et la précision : la distribution des classes dans notre jeu de données n'étant pas égale, cette mesure a plus de valeur que l'exactitude.

Cependant, en y regardant de plus près, on remarque des différences de force entre les algorithmes. L'arbre de décision, par exemple, a le plus haut rappel, avec 0,88 : cela signifie que c'est le meilleur algorithme pour classifier les vraies vulnérabilités. En revanche, la classification naïve bayésienne remporte le prix de la meilleure précision : cet algorithme ne fera que peu de faux positifs, toute instance prédite comme une vulnérabilité en sera très probablement une.

Le classifieur par forêt aléatoire, de son côté, n'obtient pas d'excellents résultats, ce qui peut paraître étonnant, car c'est en général celui qui a les meilleures performances. La réponse se cache dans le jeu de données, et tout particulièrement un attribut : la méthode Source. Comme décrit dans la section 3.3.3.2, la valeur de celle-ci est la signature de la méthode contenant la vulnérabilité : or, avec les données provenant de Juliet, le classifieur est perturbé pour deux raisons. Tout d'abord, les chaînes de caractères sont extrêmement longues, car les noms des

répertoires indiquent de quelle vulnérabilité il s'agit. De plus, ayant une vulnérabilité par classe, on a presque autant de valeur nominale possible pour ce champ que d'instance dans le jeu de données. Sauf qu'il s'avère que l'implémentation Java de cet algorithme (librairie Weka) est légèrement différente des autres : lorsqu'un nœud considère un attribut nominal pour diviser les données et créer des branches, la division n'est pas binaire comme souvent, mais correspond à une branche par valeur nominale de l'attribut. Ainsi, dans le cas de notre jeu de données particulier, pour chaque arbre de la forêt, des milliers de branches vont partir du nœud qui correspond à l'attribut méthode source. Cela ne sera très probablement pas le cas si cet algorithme est utilisé pour un autre projet, mais cela explique pourquoi le classifieur par forêt d'arbres décisionnels est moins performant ici (et très long à l'exécution).

Ainsi, il ne semble pas y avoir d'algorithme réellement meilleur que les autres. Tout d'abord, étant en train de les tester sur un projet particulier, il est difficile de prévoir avec exactitude quelles pourraient être leurs performances sur un autre projet plus commun (une application Web quelconque par exemple). Certains algorithmes sont meilleurs pour confirmer les vraies vulnérabilités, tandis que d'autres arrivent mieux à éliminer les faux positifs. Il n'est donc pas insensé de combiner des algorithmes entre eux, afin de tirer profit des avantages de chacun. De plus, la plupart de ces algorithmes peuvent indiquer une information utile : le niveau, ou seuil de confiance attribué à chaque classification. Dans le cas d'un algorithme probabiliste, comme le classifieur naïf bayésien par exemple, cette valeur correspondra à la probabilité que l'instance appartienne à la classe choisie. Cette probabilité est évidemment la plus forte, mais avoir la valeur exacte est intéressant, car l'utilisateur peut déterminer un seuil en dessous duquel il ne fera plus totalement confiance à l'algorithme, et vérifiera manuellement les alertes : c'est une option que nous avons implémentée, avec un seuil fixé par défaut que nous avons obtenu en traçant la courbe ROC pour chaque algorithme, dont un exemple apparait en Figure 4.1 pour l'arbre de décision.

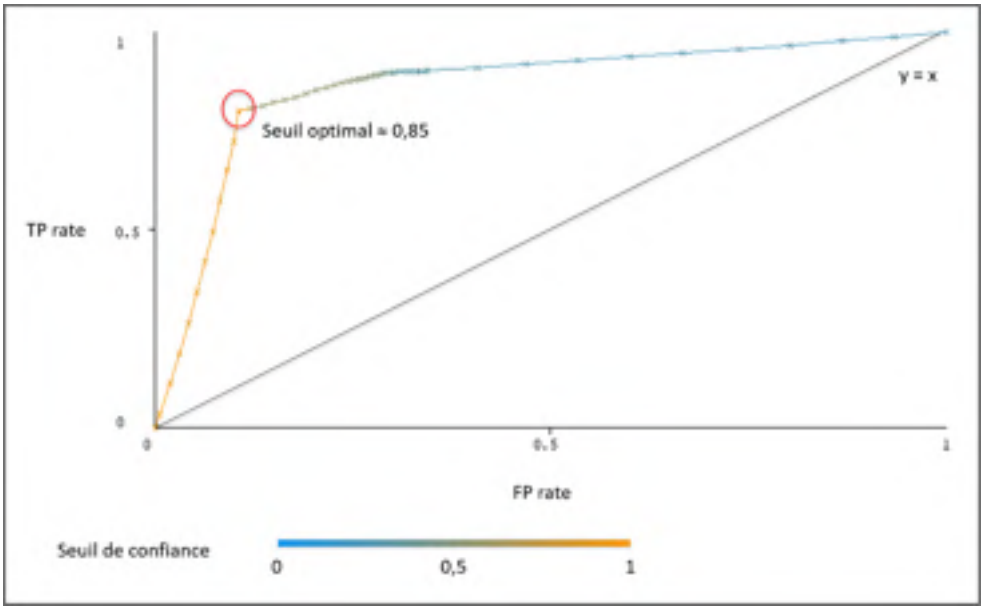

Figure 4.1 Courbe ROC pour l'algorithme d'arbre de décision

La courbe ROC est souvent représentative des résultats d'un algorithme d'apprentissage machine. La classification parfaite impliquant de passer par le point en haut à gauche, de coordonnées (0 ; 1) qui maximiserait le taux de vraies vulnérabilités détectées et minimiserait le taux de faux positifs, l'aire sous la courbe (AUC) nous donne notamment une indication de comment l'algorithme performe (la valeur maximale étant de 1). Dans notre cas, cette aire est de 0,86, ce qui est très correct, et rejoins les résultats précédemment présentés. Par ailleurs, on voit qu'en changeant le seuil de confiance de l'algorithme (à savoir on ne va considérer comme bien classifiées seulement les alertes prédites avec un score de confiance supérieur à ce seuil), on modifie le taux de vrais et faux positifs. Le seuil optimal, ici de 0,85, va être choisi comme celui maximisant le rapport entre le taux de vrais positifs et faux positifs, à savoir le point de la courbe qui s'éloigne le plus de la droite y = x, et qui s'approche le plus du point de coordonnées (0 ; 1), correspondant à la classification parfaite. Comme dit précédemment, cette valeur va être implémentée par défaut comme seuil de confiance en dessous duquel les alertes vont être présentées au développeur pour vérification. Cela permet ainsi, en plus de réduire drastiquement le nombre d'alertes à trier, de prioriser celles qui restent, pour que celles dont l'algorithme est le moins sûr passent en premier.

Enfin, il convient de parler des performances des algorithmes au niveau du temps d'exécution. Concrètement, présenter les résultats sous forme de tableau ne nous a pas paru pertinent, dans la mesure où tous les algorithmes testés construisent leur modèle et apportent des résultats extrêmement rapidement, en moins de trente secondes, à l'exception du SVM qui est un peu plus long (aux alentours de cinq minutes). Ce temps est évidemment négligeable dans la mesure où les tests ont été effectués sur plus de 10 000 alertes (qui auraient mises un temps fou à être classifiées à la main). De plus, une fois le modèle construit, classifier de nouvelles alertes (même quelques centaines à la fois) ne prend que quelques millisecondes, la construction du modèle, ou apprentissage, étant la phase la plus coûteuse en temps.

Pour conclure sur ces résultats de classification, on voit que ceux-ci sont plutôt bons. De plus, l'utilisateur a beaucoup de flexibilité sur son utilisation de l'apprentissage machine : il peut combiner les algorithmes pour gagner en performance, et il peut définir un seuil en dessous duquel il ne fera plus confiance à leurs décisions, lui permettant de prioriser les alertes restantes à sa manière. L'apprentissage machine ne doit pas être vu ici comme un moyen magique de trier toutes les alertes, mais plutôt comme une aide au développeur, pouvant lui faire gagner énormément de temps si elle est utilisée à bon escient.

4.1.3.2 Classement par type de vulnérabilité

Outre les résultats généraux des classifieurs, il est intéressant de s'intéresser aux types de vulnérabilités au cas par cas : cela peut en effet permettre d'effectuer un autre niveau de priorisation des alertes à corriger, mais également d'aiguiller les recherches futures d'améliorations de l'outil en se concentrant sur les vulnérabilités que l'algorithme d'apprentissage machine a le plus de mal à classifier.

La Figure 4.2 présente graphiquement la répartition des vulnérabilités entre les deux classes BAD et GOOD (respectivement vraie alerte et faux positif) avant le passage de l'apprentissage machine. Chaque point sur le graphique correspond donc à une alerte levée par Find Security Bugs : à gauche en bleu, l'alerte est réelle, et à droite en rouge, c'est un faux positif.

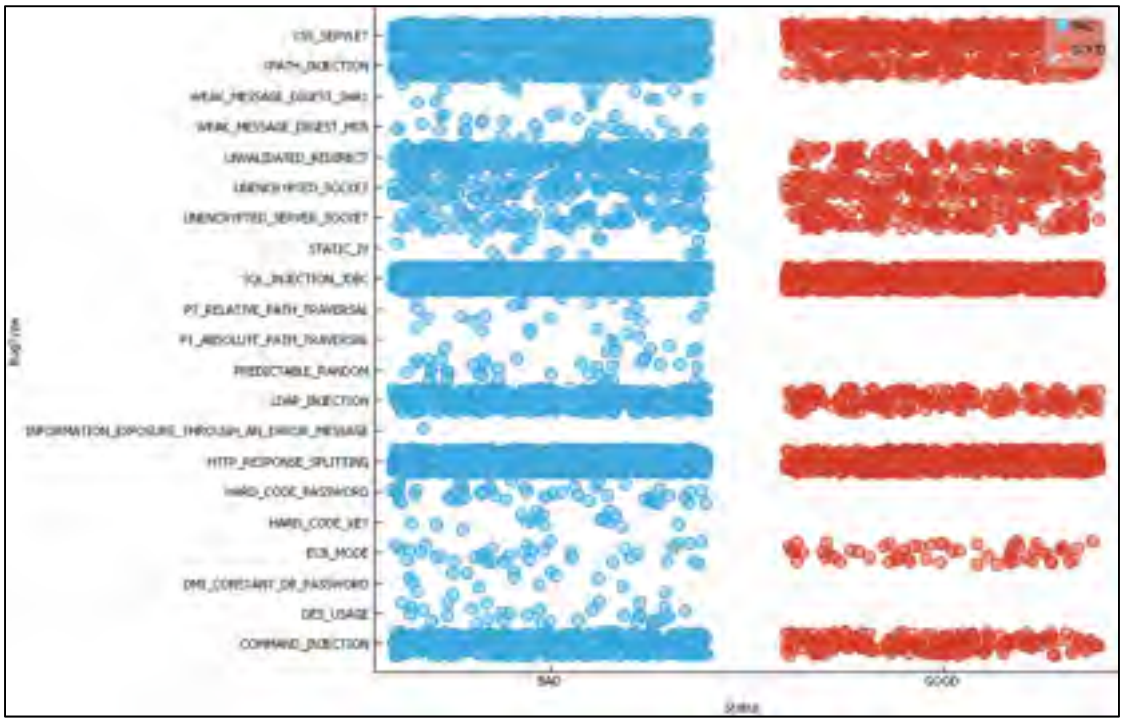

Figure 4.2 Classification des vulnérabilités par type avant apprentissage machine

Deux choses intéressantes sont à voir sur ce graphique : premièrement, et comme annoncé en section 2.1.3, de nombreux faux positifs sont à déplorer pour la majorité des vulnérabilités (colonne de droite, rouge). Le but va donc être d'éliminer le maximum d'entre eux avec l'apprentissage machine. Deuxièmement, on remarque que de nombreuses vulnérabilités ne génèrent aucun faux positif : la raison à cela est qu'elles ne correspondent pas à des injections, autrement dit elles n'impliquent pas des variables pouvant avoir un état varié. Par exemple, la vulnérabilité DES_USAGE implique seulement l'utilisation d'un algorithme de cryptographie jugé mauvais en matière de sécurité, car ayant été cassé : Find Security Bugs va donc juste avoir à détecter l'utilisation de l'algorithme en question, par une méthode caractéristique notamment, sans se soucier de ce que ce dernier prend en entrée. De la même manière, HARD_CODE_PASSWORD indique simplement l'utilisation d'un mot de passe écrit en dur dans le code, ce qui est très facile à voir. Cela explique donc pourquoi certaines vulnérabilités ne génèrent pas de faux positifs dans un premier temps. L'utilisation d'apprentissage

automatique va donc servir pour les vulnérabilités correspondant à des injections (principalement).

La Figure 4.3 présente un graphique très semblable au précédent, mais après le passage de l'algorithme d'apprentissage ayant classifié les alertes (celui choisi ici est le k plus proches voisins).

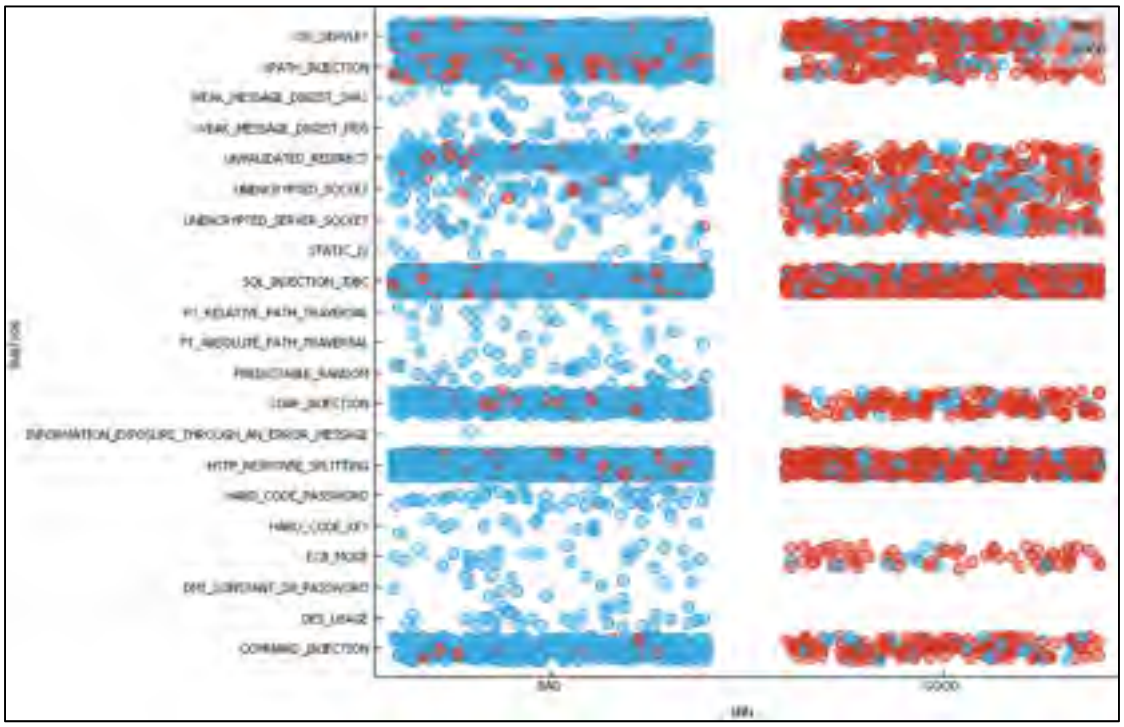

Figure 4.3 Classification des vulnérabilités par type après apprentissage machine

Sur ce graphique, on peut voir les prédictions de l'algorithme d'apprentissage machine : les points bleus correspondent toujours aux vraies alertes, et les points rouges, aux faux positifs. En revanche, ce sont les colonnes BAD et GOOD sur l'axe des abscisses qui changent par rapport au graphique précédent : ici, elles représentent les prédictions de l'algorithme d'apprentissage automatique, respectivement vraie vulnérabilité et faux positif. Ainsi, toutes les alertes bleues (donc les alertes réelles) présentes sur la colonne de gauche (BAD) ont été correctement prédites par l'algorithme, de même que les alertes rouges (les fausses alertes) dans la colonne de droite (GOOD). En revanche, les points rouges dans la colonne BAD

correspondent à des fausses alertes prédites comme étant réelles (pas d'amélioration par rapport à l'analyse statique simple), tandis que les points bleus dans la colonne GOOD correspondent à des vulnérabilités réelles prédites comme étant fausses (faux négatifs).
Globalement, les résultats que l'on peut observer sont les suivants. Tout d'abord, la plupart des faux positifs ont été éliminés, ce qui est une excellente nouvelle, car c'est notre objectif principal. Cela est en effet très visible : sur le premier graphique, tous les points rouges dans la colonne de gauche étaient des fausses alertes. Sur le second graphique en revanche, on voit que la très grande majorité des points rouges sont dans la colonne GOOD, donc ont été correctement prédits comme étant des fausses alertes par l'algorithme d'apprentissage automatique. La seule vulnérabilité posant un réel problème à l'algorithme à ce niveau est l'injection xPath (XPATH_INJECTION). Par ailleurs, et sur une note moins positive, de nombreux faux négatifs ont été ajoutés, et cela est problématique : un développeur ne veut pas que des alertes réelles soient écartées par l'algorithme qui les a considérées comme fausses, car cela pourrait être dangereux pour son application. Cependant, en regardant attentivement les différents types de vulnérabilités, il ressort que seulement deux d'entre elles donnent du mal à l'algorithme d'apprentissage automatique : les deux vulnérabilités liées aux interfaces de connexion, *UNENCRYPTED_SOCKET* et *UNENCRYPTED_SERVER_SOCKET*. Il apparaît donc que les caractéristiques choisies ne sont pas suffisantes pour caractériser ces types d'alertes, l'algorithme ayant du mal à classifier correctement une vraie vulnérabilité d'un de ces deux types.

On voit donc qu'analyser les résultats par type de vulnérabilités peut se révéler être une option extrêmement judicieuse pour le développeur : en effet, cela va l'aider à adapter son tri, en faisant notamment plus confiance à l'algorithme sur les vulnérabilités où il est le plus performant (la plupart des injections notamment). De la même manière, il pourra décider de classifier à la main les trois types d'alertes vus précédemment pour éviter les faux négatifs pouvant être dangereux pour son application. Enfin, concernant les résultats de l'apprentissage automatique en eux même, ils sont cohérents avec ce que nous avons vu dans la section précédente : une très grande majorité des faux positifs a été éliminée (plus de 85%), tandis que relativement peu d'alertes réelles ont été mal classifiées.

4.1.3.3 Importance de chaque attribut

Enfin, un dernier résultat intéressant est l'importance de chaque attribut dans la classification finale. En effet, les caractéristiques n'ont pas toutes le même « poids », certaines influent beaucoup plus que d'autres sur le choix de l'algorithme. Connaître cette information peut être un atout pour le développeur faisant de la revue de code : il aura une meilleure compréhension des principales causes de vulnérabilités (ou de faux positifs) et pourra gagner du temps sur le triage éventuel d'alertes qu'il pourrait avoir à faire par la suite.

En général, en fonction de l'algorithme utilisé, les attributs n'ont pas la même importance dans la classification. Cependant, le *ratio de gain d'information*, qui a déjà été détaillé en section 3.3.4.3, et qui est utilisé par la plupart des algorithmes d'arbres décisionnels (dont l'algorithme C4.5), nous donne une estimation précise de cette information, notamment car son calcul est indépendant de l'algorithme utilisé : il est calculé en se basant uniquement sur des probabilités tirées du nombre de valeurs pour chaque attribut, et des classes associées à chacune de ces valeurs. La figure 4.4 ci-dessous représente sur un diagramme les différents attributs et leur ratio de gain d'information :

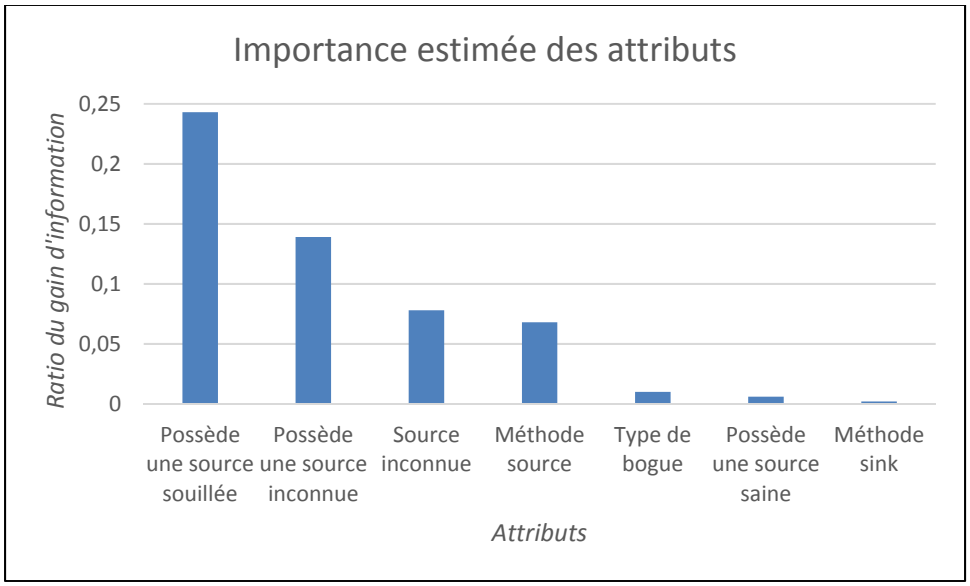

Figure 4.4 Ratio du gain d'information pour chaque caractéristique

On voit que la caractéristique *Possède une source souillée* est nettement devant les autres. Ce résultat n'est en soit pas surprenant : la plupart des vulnérabilités du jeu de données de Juliet étant des injections (cf. Tableau 4.1), il est logique que de nombreuses variables souillées soient présentes dans le code et aident fortement à déterminer si la vulnérabilité est réelle ou pas. Dans le cas d'une injection, c'est en effet très simple : si une variable souillée est passée en paramètre d'une méthode *sink*, la vulnérabilité est confirmée. Si au contraire, la variable est saine, c'est un faux positif. On note également que l'attribut *Méthode source* a un gain d'information élevé, même si cela est beaucoup moins représentatif au vu des données utilisées : cet attribut est en effet presque unique pour chaque instance. Néanmoins, lorsque l'on voit ces résultats en considérant le jeu de données actuel, on peut imaginer que cet attribut deviendrait très important sur un projet standard, où certaines méthodes pourraient être mal écrites et porteuses de vulnérabilités.

Pour résumer, le ratio de gain d'information pour chaque attribut enrichit la compréhension et l'évaluation du modèle d'apprentissage automatique, car il permet de quantifier l'importance des caractéristiques une à une. Dans notre cas, il permet notamment de confirmer que les caractéristiques issues de l'analyse du graphe (et notamment la mise en évidence des variables souillées) jouent un rôle extrêmement important dans notre modèle. Par ailleurs, cela se ressent lorsque l'on revient à la Figure 4.3, à savoir la classification en fonction du type de vulnérabilité : celles correspondant à des injections (SQL, XSS, commande, etc.), et donc très sensibles aux variables souillées, ont été très bien classifiées de manière générale.

4.2 Validité, limitations et pistes d'améliorations de la solution

Malgré une évaluation positive de notre modèle, il est important de parler de ses limitations, qui sont de plusieurs types.

4.2.1 Limitations quant aux données d'entraînement

Tout d'abord, il convient de discuter du jeu de données d'entraînement. Entraîner un modèle de manière efficace demande un grand nombre de données labélisées. Dans le cas de la

détection de vulnérabilités, labéliser une grande quantité de données (milliers d'alertes) à la main est trop coûteux en temps : c'est pour cette raison que la plupart des études se tournent vers des suites de test comme Juliet, que nous avons utilisé ici.

Ces suites contiennent un grand nombre de données labélisées, et favorisent donc la mise en place et l'utilisation des algorithmes afin de répondre vite au problème. Néanmoins, cela vient avec un inconvénient : le code utilisé est créé spécialement pour ça, et ne correspond pas vraiment à la réalité. Dans le cas de Juliet, la plupart des classes sont même autogénérées, ce qui signifie que des morceaux de codes semblables vont se retrouver un peu partout, et introduire un biais dans les performances de l'outil d'apprentissage automatique.

Pour démontrer cela, nous avons essayé d'introduire un nouvel attribut dans le vecteur de caractéristiques : le numéro de ligne où la vulnérabilité est découverte. Il est hautement improbable que cette caractéristique soit représentative d'une vulnérabilité, ou au contraire d'un faux positif, dans le code d'une application réelle : sa valeur est beaucoup trop sensible au reformatage, aux nombres de commentaires présents au sein du code, etc. De plus, un développeur ne prend jamais ceci en ligne de compte lorsqu'il effectue une classification manuelle. Nous avons donc rajouté cet attribut pour chaque instance (en conservant bien sur les autres ayant servi à l'évaluation réelle), et avons évalué ce nouveau modèle exactement de la même manière que précédemment, à savoir avec les mêmes mesures, et par validation croisée de 10 échantillons. Les résultats sont présentés dans le Tableau 4.3 :

Tableau 4.3 Résultats de l'évaluation des algorithmes en rajoutant le *Numéro de ligne*

	Rappel	**Précision**	**F-mesure**	**Exactitude (%)**
CNB	0,908	0,913	0,911	87.24
K-PPV (k = 2)	0,997	0,977	0,987	98.12
Arbre de décision	0,992	0,959	0,975	96.37
Forêt aléatoire	0,992	0,966	0,979	96.96
SVM	0,951	0,953	0,952	93.15

Ceux-ci sont stupéfiants : plus de 95% d'exactitude pour la plupart des algorithmes testés, on frôle la perfection. Cependant, il est évident que cela ne peut pas arriver dans un cas réel : il est peut-être possible que cet attribut influence très légèrement la classification, mais en principe une vulnérabilité peut se retrouver à peu près n'importe où, les classes ne font pas toutes la même taille, et ne contiennent pas forcément le même nombre de méthodes. En d'autres termes, il est impossible qu'un seul attribut change les résultats et augmente de façon drastique les performances d'un classifieur, d'autant plus si cet attribut ne semble pas du tout pertinent à première vue. L'unique raison pour laquelle c'est le cas ici est que la majorité du code est autogénéré, et donc toutes les classes ont la même structure, avec les vulnérabilités globalement au même endroit.

On voit donc qu'il n'est pas évident de trouver un jeu de données d'entraînement qui permet d'évaluer avec grande précision le modèle d'apprentissage automatique choisi. Une suite de test comme Juliet est d'une grande aide, et permet la création et l'évaluation complète d'un modèle, mais elle ne suffit pas pour prévoir des cas plus concrets. En d'autres termes, nous ne pouvons pas garantir à l'heure actuelle que les performances mesurées en section 4.1.3.1 se retrouveront exactement si un développeur décide d'utiliser Find Security Bugs avec notre modèle afin de trier ses alertes, sur sa propre application. Cela peut aller dans les deux sens : les performances peuvent être moins bonnes (le code n'étant pas autogénéré, l'algorithme ne pourra probablement pas repérer les mêmes ressemblances qu'avec les données de Juliet), ou elles peuvent peut-être être meilleures (les attributs correspondant à des signatures de méthode, tels que *Methode source* ou *Unknow source*, auront beaucoup moins de valeurs nominales différentes au sein du jeu de données, et des ressemblances pourront être faites à ce niveau).

Il n'est donc pas possible de prévoir avec certitude le comportement du modèle en dehors du jeu de données connu (Juliet), même si les caractéristiques choisies ont fait leurs preuves. C'est pour cette raison que les données d'entraînement sont extrêmement importantes, et se doivent d'être le plus possible en adéquation avec les futurs cas concrets d'utilisation.

4.2.2 Limitations de l'apprentissage machine en général

Par ailleurs, la qualité du jeu de données d'entraînement n'est pas la seule limitation qu'un modèle d'apprentissage rencontre. Un modèle a en effet besoin de temps pour se construire. Grâce à Juliet, nous avons eu des milliers de données d'un coup, ce qui était parfait pour la phase d'entraînement : le modèle (algorithme entraîné) a donc été rapidement prêt à classifier des nouvelles instances. Or, ce ne sera probablement pas le cas d'un développeur gérant son application : à moins que celle-ci ne soit très conséquente, il va falloir un certain temps, correspondant à l'évolution de l'application, avant que l'algorithme ait assez de données pour réaliser une classification correcte. Comme dit précédemment, une dizaine ou même une centaine d'alertes ne suffisent pas toujours à avoir un modèle suffisamment intéressant.

Un moyen de contourner ce problème et d'utiliser un modèle existant (algorithme déjà entraîné) pour classifier ses alertes. Ce modèle pourrait être celui que l'on a évalué sur la suite Juliet, mais il est difficile de ne pas douter de la pertinence des résultats que l'on pourrait obtenir. En effet, chaque modèle est construit dépendamment de son contexte : même si les attributs ne changent pas (le développeur qui fait de la revue de code ne va pas changer sa façon de trier une alerte suivant le projet), un contexte différent de celui dans lequel s'est entraîné l'algorithme peut fausser les résultats sortis par celui-ci.

On voit donc que l'apprentissage machine a ses limites, et que celles-ci sont principalement liées aux données servant à entraîner l'algorithme, et à la différence de contexte entre deux applications, obligeant un réentraînement du modèle d'apprentissage. Les principaux défis sont de trouver des données d'entraînement complètes et de confiance, ainsi que des caractéristiques représentant au mieux le problème. Malgré tout, il permet un gain de temps considérable, notamment dans le domaine du triage d'alertes, et n'est pas si compliqué à implémenter, notamment grâce aux solutions existantes (Weka par exemple).

4.2.3 Validité de la solution

Après avoir analysé les résultats, et discuter des limitations, il est temps de s'interroger sur la validité de la solution. En d'autres termes, est-ce que celle-ci est viable ? Peut-elle être utilisée efficacement ? Et surtout, répond-elle aux objectifs établis au début de l'étude ?

Tout d'abord, les résultats sont très corrects. Il est vrai qu'il y a très probablement moyen d'y apporter des améliorations (cf. section suivante), mais globalement le modèle établi ici réussit ce pour quoi il a été créé : éliminer les faux positifs après l'analyse statique par Find Security Bugs. En effet, plus de 85% des faux positifs peuvent être éliminés, ce qui est un gain de temps énorme pour un développeur. Par ailleurs, même si un peu plus de 10% des alertes sont au final mal classifiés, ceci n'est pas un grand nombre en soi, d'autant plus que la mise en place du seuil de confiance permet de mettre en évidence une grande partie de ces mauvais choix de l'algorithme : en effet, la très grande majorité des vulnérabilités incorrectement classifiées le sont avec une probabilité inférieure à 0,8. De ce fait, les résultats ne sont pas du tout en dessous de nos espérances, et notre implémentation permet au développeur de garder une marge de manœuvre concernant le triage : à la fois grâce au seuil de confiance, mais également au type de vulnérabilité (cf. section 4.1.3.2).

Ensuite, l'un des points forts de notre solution est qu'elle est entièrement fonctionnelle. Dans la littérature (cf. Chapitre 1), de nombreuses études ont proposé des solutions impliquant de l'apprentissage machine pour trier des alertes. Cependant, aucune ne semblait réellement être implémentée, seul l'aspect théorique était abordé (identification des caractéristiques, évaluation d'un ou plusieurs algorithmes). Dans notre cas, utiliser l'apprentissage automatique est aussi simple que d'utiliser Find Security Bugs : notre solution y est totalement intégrée. Une fois l'analyse statique effectuée, l'utilisateur a juste à lancer quelques lignes de commande supplémentaires, qui sont évidemment documentées, et il pourra entraîner son algorithme, sauvegarder le modèle entraîné, et classifier de nouvelles vulnérabilités présentes dans son application : le tout sans réelle complexité d'utilisation. On ajoute donc ici un cadre concret à

l'approche théorique souvent présente dans la littérature : notre solution a donc réellement une utilité.

Il résulte donc, après cette analyse de fin d'étude, que notre solution répond correctement aux objectifs initialement définis, et peut être un réel atout dans la réduction de fausses alertes. Il ne semble alors pas aberrant de confirmer sa validité, même si certains points pourraient être améliorés.

4.2.4 Pistes d'améliorations

Pour finir, il est important de parler des améliorations qui pourraient être apportées concernant ce modèle d'apprentissage machine, pour d'éventuelles futures études dans le domaine.

Tout d'abord, la perfection est une notion étrangère à l'apprentissage automatique. Il est toujours possible de faire mieux et d'augmenter les performances de son modèle. L'un des principaux moyens pour cela est de trouver des nouvelles caractéristiques pertinentes à ajouter au vecteur, qui permettraient de décrire encore plus précisément les modèles de vulnérabilités et de faux positifs. Par exemple, on pourrait s'intéresser à des expressions dynamiques (évaluées lors de l'exécution), qui peuvent mener à une injection XSS : la syntaxe particulière de certains fichiers, comme les fichiers JSP, peut être dangereuse. D'autres caractéristiques intéressantes pourraient également être extraites en étendant le graphe : on pourrait par exemple rajouter des nœuds pour les classes et interfaces, reliées entre elles par des liens d'héritage et d'implémentation. Cela apporterait de la complexité au modèle, mais permettrait entre autres de traquer les appels à des superclasses, et ainsi élargir notre champ de vision en ce qui concerne l'analyse de variables souillées.

De plus, une autre piste d'amélioration serait de s'intéresser aux différents types de vulnérabilités, indépendamment les uns des autres. En ce qui concerne les attributs choisis pour notre modèle, l'étude a été générale. Notre raisonnement a été de trouver des caractéristiques les plus proches de ce qu'un développeur effectuant de la revue de code irait chercher pour

trier les alertes. Mais il est possible d'effectuer un raisonnement différent, en examinant les types de vulnérabilités un par un, et d'essayer de mettre en évidence des caractéristiques qui leur sont propres. En effet, nous avons vu, dans la section 4.1.3.2, que l'algorithme avait plus de mal à classifier certaines vulnérabilités. De fait, trouver de nouveaux attributs pour les décrire de façon unique pourrait grandement améliorer la détection. Cela a l'inconvénient de devoir entraîner différents modèles d'apprentissage, mais certains types d'alertes devraient être mieux couverts par l'outil. Par ailleurs, l'inverse s'applique également : on a vu que Find Security Bugs était très efficace sur certains d'entre eux et ne commettait quasiment pas de faux positifs. Chercher des caractéristiques sur ces vulnérabilités serait donc une perte de temps, l'apprentissage automatique n'ayant pas lieu d'être dans ce cas.

Enfin, une dernière piste d'amélioration du modèle pourrait être de le tester sur un projet réel conséquent. Même si cela semble plus facile à dire qu'à réaliser, et que cela prendrait beaucoup de temps pour construire le jeu de données de confiance, les performances pourraient être mesurées à leur juste valeur. En poussant l'idée plus loin, on pourrait imaginer mettre en place des études expérimentales contrôlées, lorsque notre outil sera utilisé par de nombreux développeurs, afin de tester son efficacité dans différents contextes.

.

CONCLUSION

La recherche menée ici s'est focalisée sur le triage de vulnérabilités ressortant de l'analyse statique d'un code logiciel par un outil réputé : Find Security Bugs. L'objectif était de trouver un moyen de réduire le nombre de faux positifs qu'engendre une telle analyse, afin de permettre au développeur effectuant une revue de code de gagner un temps précieux. Pour cela, nous avons décidé d'utiliser de l'apprentissage machine.

La méthodologie suivie a été semblable à celle de Koc et al. (2017). Après une prise en main du contexte et des outils mis en jeu, il a fallu extraire des caractéristiques importantes représentant au mieux des vulnérabilités, et surtout permettant de faire la différence entre ces dernières et les faux positifs rapportés par l'outil d'analyse statique. Après identification de telles caractéristiques, une représentation du code sous forme de graphe a permis de compléter le rapport d'alertes de Find Security Bugs pour extraire celles-ci. La notion d'apprentissage machine est alors entrée en jeu : le principe était d'entraîner un algorithme sur un jeu de données labélisées, provenant d'une suite de test conçue dans ce but (Juliet), en se basant sur ces caractéristiques : le modèle obtenu (correspondant à l'algorithme entraîné) permettrait donc de classifier automatiquement toute nouvelle alerte correspondant à une potentielle vulnérabilité.

Différents algorithmes ont été évalués, et les résultats ont été très concluants : dans notre contexte, environ 85% des faux positifs issus de l'analyse statique ont pu être éliminés grâce à l'apprentissage machine. Nous avons montré, grâce à différentes mesures, qu'il n'existait pas d'algorithme parfait, et que chacun avait ses points forts et faibles : combiner les algorithmes pour tirer parti du meilleur de chacun est une solution recommandée. Par ailleurs, il est important de préciser que l'algorithme d'apprentissage automatique pouvait également causer des faux négatifs, à savoir considérer une vulnérabilité réelle comme étant bénigne. Même si cela arrive dans peu de cas, les conséquences peuvent être graves. Aussi, nous avons décidé de tracer la probabilité à laquelle l'algorithme classifiait une instance dans la classe choisie, et de permettre au développeur d'établir un seuil en dessous duquel la classification effectuée par

l'algorithme ne serait plus certaine. Autrement dit, le développeur est libre de choisir à quel niveau il décide de faire confiance à l'algorithme d'apprentissage machine : toute alerte étant classifiée avec une probabilité en dessous du seuil minimal sera aussitôt placée dans une liste d'alertes à prioriser (pour un triage manuel). Cela permet entre autres à l'utilisateur d'améliorer le modèle, à savoir entraîner à nouveau l'algorithme sur des données mises à jour, qui ont pu être mal classifiées par celui-ci dans un premier temps : il y a donc un retour sur l'apprentissage. Par ailleurs, notre étude a permis de mettre en évidence certains types de vulnérabilités plus ou moins faciles à classifier par les algorithmes évalués, en utilisant les caractéristiques choisies : cela permet au développeur de mettre en place, s'il le souhaite, un autre niveau de priorisation des alertes.

Malgré des résultats très concluants, il est fondamental de comprendre que l'apprentissage machine se heurte à des limitations qui lui sont propres : la principale est liée aux données d'entraînement. Il est très complexe de trouver des données déjà labélisées sur lequel l'algorithme peut s'entraîner, et cela est encore plus vrai quand il est question de détection de vulnérabilité. Afin de pallier ce problème, la plupart des études dans le domaine, y compris la nôtre, utilisent des jeux de données conçus dans ce but, comme la suite de test Juliet. Néanmoins, ce choix a un prix : le modèle obtenu n'est pas totalement vraisemblable, dans la mesure où la plupart du code est autogénéré. Les résultats sont donc à considérer avec précaution, car ils peuvent ne pas être exactement les mêmes si la solution est testée sur un autre projet, d'autant plus qu'il faut un ensemble de données suffisamment grand dès le départ pour entraîner l'algorithme et créer un modèle fiable.

De ce fait, la recherche dans ce domaine n'en est probablement qu'à ses débuts : de nombreuses améliorations sont possibles. Des nouvelles caractéristiques plus précises peuvent par exemple être extraites de différentes manières (en étendant le graphe créé par exemple), ou des études peuvent se concentrer sur certains types particuliers de vulnérabilités, pour trouver des attributs représentatifs qui leur seraient propres : concernant ce point, notre étude est restée ici relativement générale. De plus, labéliser à la main un grand jeu de données correspondant à un ensemble de vulnérabilités d'un réel projet de source libre, bien que travail titanesque, pourrait

permettre d'évaluer notre modèle dans un environnement plus concret, et donc fournir des résultats plus précis.

En conclusion, malgré le fait qu'il y ait matière à discussion (ce qui est bien souvent le cas en apprentissage machine), les objectifs de ce mémoire ont bien été atteints : la solution est fonctionnelle, intégrée totalement à Find Security Bugs, et n'importe quel développeur est désormais en mesure de l'utiliser simplement pour son projet. De plus, comme convenu, elle permet de réduire drastiquement le nombre de faux positifs généré par l'outil, sans perte réelle de performance.

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Analytics, Y. (2011). Analytics: The Machine Learning Advantage. Repéré à https://yottamine.com/machine-learning-svm

Arteau, P. (2012, 7 August 2017). Find Security Bugs. Repéré à https://find-sec-bugs.github.io/

Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J., & Pugh, W. (2008). Using static analysis to find bugs. *IEEE software, 25*(5).

Baca, D., Petersen, K., Carlsson, B., & Lundberg, L. (2009). Static code analysis to detect software security vulnerabilities-does experience matter? Dans *Availability, Reliability and Security, 2009. ARES'09. International Conference on* (pp. 804-810). IEEE.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998). Clone detection using abstract syntax trees. Dans *Software Maintenance, 1998. Proceedings., International Conference on* (pp. 368-377). IEEE.

Benchmark. (2015). OWASP Benchmark Project. Repéré à https://www.owasp.org/index.php/Benchmark#tab=Main

Boland Jr, F. E., & Black, P. E. (2012). The Juliet 1.1 C/C++ and Java test suite. *Computer (IEEE Computer), 45*(Computer (IEEE Computer)).

Breiman, L. (2001). Random forests. *Machine learning, 45*(1), 5-32.

Bush, W. R., Pincus, J. D., & Sielaff, D. J. (2000). A static analyzer for finding dynamic programming errors. *Software-Practice and Experience, 30*(7), 775-802.

Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM, 56*(2), 82-90.

Chappelly, T., Cifuentes, C., Krishnan, P., & Gevay, S. (2017). Machine learning for finding bugs: An initial report. Dans *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Workshop on* (pp. 21-26). IEEE.

Charest, T., Rodgers, N., & Wu, Y. (2016). Comparison of static analysis tools for java using the juliet test suite. Dans *11th International Conference on Cyber Warfare and Security* (pp. 431-438).

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. Dans *WODA 2003: ICSE Workshop on Dynamic Analysis* (pp. 24-27).

Evans, D., Guttag, J., Horning, J., & Tan, Y. M. (1994). LCLint: A tool for using specifications to check code. *ACM SIGSOFT Software Engineering Notes, 19*(5), 87-96.

FSB Bugs Patterns. (2017, 7 August 2017). Find Security Bugs - Bugs Patterns. Repéré à https://find-sec-bugs.github.io/bugs.htm

FSB results on Benchmark. (2018). OWASP Benchmark Scorecard for FBwFindSecBugs v1.4.6 (SAST). Repéré à https://rawgit.com/OWASP/Benchmark/master/scorecard/Benchmark_v1.2_Scorecard_for_FBwFindSecBugs_v1.4.6.html

Goseva-Popstojanova, K., & Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology, 68*, 18-33.

Halfond, W. G., Viegas, J., & Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. Dans *Proceedings of the IEEE International Symposium on Secure Software Engineering* (Vol. 1, pp. 13-15). IEEE.

Hanam, Q., Tan, L., Holmes, R., & Lam, P. (2014). Finding patterns in static analysis alerts: improving actionable alert ranking. Dans *Proceedings of the 11th Working Conference on Mining Software Repositories* (pp. 152-161). ACM.

Holzmann, G. J. (2002). Static source code checking for user-defined properties. Dans *Proc. IDPT* (Vol. 2).

Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *Acm sigplan notices, 39*(12), 92-106.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? Dans *Proceedings of the 2013 International Conference on Software Engineering* (pp. 672-681). IEEE Press.

Johnson, S. C. (1977). *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill.

Kim, S., & Ernst, M. D. (2007). Which warnings should i fix first? Dans *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 45-54). ACM.

Koc, U., Saadatpanah, P., Foster, J. S., & Porter, A. A. (2017). Learning a classifier for false positive error reports emitted by static code analysis tools. Dans *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (pp. 35-42). ACM.

Komiya, R., Paik, I., & Hisada, M. (2011). Classification of malicious web code by machine learning. Dans *Awareness Science and Technology (iCAST), 2011 3rd International Conference on* (pp. 406-411). IEEE.

Lam, M. S., Martin, M., Livshits, B., & Whaley, J. (2008). Securing web applications with static and dynamic information flow tracking. Dans *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (pp. 3-12). ACM.

Livshits, V. B., & Lam, M. S. (2005). Finding Security Vulnerabilities in Java Applications with Static Analysis. Dans *USENIX Security Symposium* (Vol. 14, pp. 18-18).

Louridas, P. (2006). Static code analysis. *IEEE software, 23*(4), 58-61.

Medeiros, I., Neves, N. F., & Correia, M. (2014). Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. Dans *Proceedings of the 23rd international conference on World wide web* (pp. 63-74). ACM.

Mitropoulos, D., Gousios, G., & Spinellis, D. (2012). Measuring the occurrence of security-related bugs through software evolution. Dans *Informatics (PCI), 2012 16th Panhellenic Conference on* (pp. 117-122). IEEE.

Muske, T., & Serebrenik, A. (2016). Survey of approaches for handling static analysis alarms. Dans *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on* (pp. 157-166). IEEE.

Neo4j. (2007). Neo4j Graph Platform. Repéré à https://neo4j.com/

OWASP. (2001). Repéré à https://www.owasp.org/index.php/Main_Page

Reynolds, Z. P., Jayanth, A. B., Koc, U., Porter, A. A., Raje, R. R., & Hill, J. H. (2017). Identifying and documenting false positive patterns generated by static code analysis tools. Dans *Proceedings of the 4th International Workshop on Software Engineering Research and Industrial Practice* (pp. 55-61). IEEE Press.

Schwartzbach, M. I. (2008). Lecture notes on static analysis. *Basic Research in Computer Science, University of Aarhus, Denmark*.

Shen, H., Fang, J., & Zhao, J. (2011). Efindbugs: Effective error ranking for findbugs. Dans *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on* (pp. 299-308). IEEE.

SpotBugs. (2016). SpotBugs - Find bugs in Java Programs. Repéré à https://spotbugs.github.io/

Tripp, O., Guarnieri, S., Pistoia, M., & Aravkin, A. (2014). Aletheia: Improving the usability of static security analysis. Dans *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (pp. 762-774). ACM.

Velicheti, L. M. R., Feiock, D. C., Peiris, M., Raje, R., & Hill, J. H. (2014). Towards modeling the behavior of static code analysis tools. Dans *Proceedings of the 9th Annual Cyber and Information Security Research Conference* (pp. 17-20). ACM.

Viega, J., Bloch, J.-T., Kohno, Y., & McGraw, G. (2000). ITS4: A static vulnerability scanner for C and C++ code. Dans *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference* (pp. 257-267). IEEE.

Webber, J. (2012). A programmatic introduction to neo4j. Dans *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity* (pp. 217-218). ACM.

WebGoat. (2016, 3 Janvier 2018). OWASP WebGoat Project. Repéré à https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

Weka. (1997, 23 Janvier 2017). Weka 3: Data Mining Software in Java. Repéré à https://www.cs.waikato.ac.nz/ml/weka/

WEKA Machine Learning. (2016, 2016). Data Mining Part 4. Repéré à https://www.cs.waikato.ac.nz/~tcs/DataMining/Short/CH4_2up.pdf

Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. Dans *Security and Privacy (SP), 2014 IEEE Symposium on* (pp. 590-604). IEEE.

Yamaguchi, F., Lindner, F., & Rieck, K. (2011). Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. Dans *Proceedings of the 5th USENIX conference on Offensive technologies* (pp. 13-13). USENIX Association.

Yamaguchi, F., Lottmann, M., & Rieck, K. (2012). Generalized vulnerability extrapolation using abstract syntax trees. Dans *Proceedings of the 28th Annual Computer Security Applications Conference* (pp. 359-368). ACM.

Yuksel, U., & Sozer, H. (2013). Automated classification of static code analysis alerts: A case study. Dans *Software Maintenance (ICSM), 2013 29th IEEE International Conference on* (pp. 532-535). IEEE.