Identification automatique des vulnérabilités de sécurité dans les systèmes logiciels

Raounak Benabidallah

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ BRETAGNE SUD
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par **Raounak BENABIDALLAH**

Identification automatique des vulnérabilités de sécurité dans les systèmes logiciels

Thèse présentée et soutenue à « VANNES », le « 30/11/2020 »
Unité de recherche : Institut de Recherche en Informatique et Systèmes Aléatoires, UMR 6074
Thèse N° : 576

Rapporteurs avant soutenance :

Tegawendé F. BISSYANDE	Professeur à l'Université du Luxembourg
Chouki TIBERMACINE	Maître de conférence HDR à l'Université de Montpellier

Composition du Jury :

Président :	Christelle Urtado	Maître de conférences HDR à l'École des Mines d'Alès, France
Examinateurs :	Tegawendé François Bissyandé	Professeur à l'Université du Luxembourg, Luxembourg
	Chouki Tibermacine	Maître de conférences HDR à l'Université de Montpellier, France
Dir. de thèse :	Salah Sadou	Professeur à l'Université Bretagne Sud
Co-dir. de thèse :	Isabelle Borne	Professeur à l'Université Bretagne Sud

Remerciements

Je tiens à remercier sincèrement mes directeurs de thèse Salah Sadou et Isabelle Borne de m'avoir donné l'opportunité de réaliser cette thèse dans une équipe bienveillante et un laboratoire chaleureux. Cette thèse m'a permis de m'initier au monde de la recherche et de découvrir ses nombreuses facettes, qui jusqu'ici, m'étaient inconnues. Je les remercie également pour le sujet qui a fini par m'intéresser fortement. Avec du recul, j'ai réussi à comprendre que cette thèse représentait l'inconnu pour moi et cela a été le problème majeur durant ces dernières années.

Je remercie profondément mes deux rapporteurs de thèse : Tegawendé Bissyandé et Chouki Tibermacine d'avoir accepté d'examiner ma thèse sans la moindre hésitation. Malgré la crise sanitaire actuelle et les procédures mises en place, j'ai eu la chance d'avoir des retours de qualité, comportant à la fois des critiques instructives et de nouvelles pistes de recherche. J'adresse mes remerciements à Christelle Urtado qui a présidé mon jury de thèse avec professionnalisme et beaucoup de bienveillance. Je suis fière d'avoir soutenu devant un tel jury.

Beaucoup de personnes ont également contribué dans cette thèse. Je remercie en premier Gildas Menier qui m'a apporté son aide lorsque j'en avais le plus besoin. Grâce à son implication, j'ai pu réaliser mon dernier travail de thèse en intégrant ses remarques qui ont toujours été précises et pertinentes. Je remercie profondément Eric Coatanea avec qui j'ai eu la chance de collaborer durant ces deux dernières années. Cette collaboration ne s'inscrit pas entièrement dans le cadre de ma thèse mais m'a appris énormément sur les aspects collaboratif et interdisciplinaire de la recherche. Je le remercie également pour les échanges enrichissant et sa bonne humeur. Je remercie aussi l'équipe Diverse qui m'a chaleureusement accueillie et qui a facilité mon intégration. J'espère que je pourrai échanger avec les membres de cette équipe, dès que la covid nous le permet...

Outre l'aspect scientifique d'une thèse, l'aspect social a toute son importance. Je n'aurais certainement pas pu finir ma thèse sans l'implication de plusieurs personnes dans ma

vie. En premier, un grand merci à ma binôme de thèse Lucie avec qui j'ai partagé le bon et le "moins bon" dès le début de nos thèses : stress, doute, déprime, joie et fierté. L'avoir à mes côtés m'a donné beaucoup de courage et de motivation et pour cela, je ne la remercierai jamais assez. J'ai beaucoup de chance de t'avoir comme amie. Je remercie aussi Nan qui m'a également beaucoup soutenu et encouragé. À l'heure actuelle, je suis contente et fière que les piratesses des 3 continents sont toutes docteures. Merci à Jamila pour les longues pauses cafés et les discussions plus que diverses et variées. Nos fous rires me manquent énormément. Merci à Imane qui a été ma bouffée d'oxygène à chaque passage. Nos discussions, rires et bons plats restent mémorables pour moi. Merci à Manal pour son aide incroyable ! Tu as réussi à avoir une place dans ma vie très rapidement. Je remercie aussi tous mes collègues et amis du laboratoire IRISA : Claire, Iris, Jean-Christophe, Behzad, Mathieu, Lionnel, Delphine, Armel, Mael, Fadhlallah, Clément, Elia, Izaskun, Jamal, Paul, Jade, Marie. J'en ai certainement oublié quelques-uns mais je remercie tous ceux qui ont rendu cette thèse agréable. Une pensée à Vanéa qui n'est plus parmi nous !

Je remercie aussi Laure Sadou qui a été une mère et une amie pour moi. Je ne te remercierai jamais assez pour ton accueil, ton accompagnement, tes conseils, ton inquiétude et ton affection. Merci simplement d'avoir été présente dès mon arrivée en France et d'avoir pris toutes les initiatives nécessaires pour rendre ma vie agréables. Merci à Salah et à toi pour tout ce que vous faites pour moi afin de me faire sentir encore en famille.

Je remercie profondément toute ma famille pour son soutien inconditionnel. Je remercie mes merveilleux parents Mohamed et Louisa qui ont toujours cru en moi depuis mon enfance. Je vous remercie de m'avoir inculqué les bonnes valeurs. J'espère que je suivrai vos pas et que je vous rendrai fiers. Mes chères tantes, Naima, Soraya, Farida et Karima, et mon oncle Hacen, qui m'ont élevée et qui ont fait de moi la femme forte et indépendante que je suis. Je vous remercie surtout pour toute la tendresse et l'amour que vous me témoignez tous les jours. Je resterai à jamais votre fille. Mes soeurs Rahil et Randa qui ont toujours veillé sur moi et qui ont clairement eu un rôle maternel plus qu'important dans mon éducation et ma vie. Je vous remercie d'être mes amies en plus d'être mes soeurs. Je remercie mes beaux frères Zineddine et Amine d'avoir été des frères pour moi et ma meilleure amie Sarah pour sa précieuse place dans ma vie. Je vous remercie pour votre amour et bienveillance, pour vos visites plaisantes et apaisantes et surtout de m'avoir

supporté durant toutes ces années. Je n'aurai certainement pas réussi à accomplir cette thèse sans vous.

Je remercie mes neveux et nièces de me rendre tellement heureuse avec le simple fait de les voir. Je vous remercie pour le bonheur que vous me donnez, je vous remercie de me faire rire et sourire à chaque mot, pas, jeu et bêtises. Je ne peux exprimer tout l'amour que j'ai pour vous.

Je voudrais adresser un remerciement spécial à Rymel, ma soeur "jumelle" et amie avec qui j'ai littéralement tout partagé depuis ma naissance. La vie chez nos parents puis nos tantes, le retour à Alger, les études, les confidences, les amitiés, et notre séjour en France. Tu es une personne en or et j'ai beaucoup de chance de t'avoir dans ma vie.

Last but not least, je remercie la personne qui a été mon pilier : Elyes ! Tu as été la personne la plus présente et encourageante pour moi. Je te remercie pour TOUT ce que tu as pu faire pour moi durant toutes ces années : me faire rire, me faire pleurer d'émotion, me faire sortir quand j'en avais envie, me laisser respirer quand j'en avais besoin., me supporter dans mes périodes de doute et de colère. Ta générosité, ta tendresse et ton amour ont fait de moi une nouvelle femme. Je ne te remercierai jamais assez d'être là pour moi.

À mon défunt grand père,
À mes chères et tendres tantes,
À mes merveilleux mes parents,
À mes soeurs chéries.

SOMMAIRE

Première partie

Introduction

Chapitre 1

Introduction

1.1 Contexte

Les cyberattaques sont un fléau d'envergure mondiale et ne cessent d'augmenter à la fois en termes de fréquence, de gravité et d'impact [1]. Face à ce fléau, les entreprises se retrouvent de plus en plus vulnérables. En effet, les attaques de sécurité peuvent entraîner de graves dommages pour les organisations, pouvant aller de l'atteinte à leur image et leur réputation jusqu'à la paralysie de leur système d'information. Ainsi, il devient primordial pour ces organisations de se protéger et de sécuriser leur système d'information afin de limiter les impacts liés aux cyberattaques. L'étape clé pour toute protection est de trouver la source du problème. Dans le domaine de la sécurité informatique, les cyberattaques sont généralement dues à l'existence de vulnérabilités qui sont exploitées par des attaquants pour atteindre leurs objectifs.

L'organisation Mitre [2], qui maintient le dictionnaire CVE (*Common Vulnerabilities and Exposures*) [1], définit la vulnérabilité comme étant :

> *" une faiblesse dans la logique de fonctionnement trouvée dans les composants logiciels et matériels qui, lorsqu'elle est exploitée, a un impact négatif sur la confidentialité, l'intégrité ou la disponibilité."*

Un *exploit*, aussi appelé une vulnérabilité exploitable, est l'intersection de trois éléments : *(i) une vulnérabilité de sécurité, (ii) l'accès de l'attaquant à la faille et (iii) la capacité de l'attaquant à exploiter la faille*. Pour exploiter une vulnérabilité, un attaquant doit disposer d'au moins un outil ou une technique applicable qui peut atteindre une faiblesse du système [3].

En d'autres termes, la vulnérabilité est une faiblesse dans un système logiciel tandis qu'un exploit est une attaque qui exploite cette vulnérabilité. Alors que la vulnérabilité

signifie qu'il existe théoriquement un moyen d'exploiter une certaine faiblesse (c'est-à-dire qu'une vulnérabilité existe), un exploit signifie qu'il existe un chemin précis pour le faire. Naturellement, les attaquants veulent trouver des faiblesses qui sont réellement exploitables. Les défenseurs, quant à eux, priorisent la correction des vulnérabilités exploitables afin de sécuriser les systèmes informatiques.

Les vulnérabilités sont classées en fonction de la classe de ressource à laquelle elles sont liées : matériel, logiciel, réseau, personnel, site physique ou encore organisationnel. Parmi ces différents types de vulnérabilités, celles basées sur le code sont responsables de la majorité des exploits [4]. Par conséquent, cette thèse se focalise particulièrement sur la découverte des vulnérabilités logicielles

Depuis un certain nombre d'années, des efforts conséquents ont été investis pour tenter d'endiguer les problèmes de sécurité et plus particulièrement ceux liés aux systèmes logiciels. Des lignes directrices ont été suggérées depuis les années 70 [5] avec la mise en place de bonnes pratiques en matière de développement logiciel, l'utilisation d'un environnement de conception sécurisé, le choix de langages et de règles de programmation appropriés, etc. En dépit de ces efforts, le nombre de vulnérabilités ne cesse d'augmenter de façon exponentielle. Ce phénomène peut être observé dans la figure 1.1 présentant le nombre de vulnérabilités publiées chaque année depuis 1999. Une tendance à la hausse peut être observée avec une explosion des vulnérabilités signalées au cours des trois dernières années.

Les vulnérabilités logicielles peuvent être très coûteuses lorsqu'elles sont exploitées. Une faille logicielle peut faire exploser des vaisseaux spatiaux [6], faire perdre le contrôle de centrifugeuses nucléaires [7], ou forcer un constructeur automobile à rappeler des milliers de voitures défectueuses [8]. Pire encore, les défauts critiques pour la sécurité ont tendance à être difficiles à détecter, plus difficiles à protéger et jusqu'à cent fois plus coûteux après le déploiement du logiciel [9]. La situation empire avec la taille et la complexité du logiciel qui croissent très rapidement [10]. Les réviseurs de code expérimentés ne peuvent plus analyser des applications comportant des millions de lignes de code pour trouver les failles dans un délai raisonnable. Cette situation rend urgente la mise en place de systèmes de détection automatisés afin de pouvoir trouver les vulnérabilités le plus tôt possible dans le cycle de développement des logiciels. De ce fait, les solutions et les

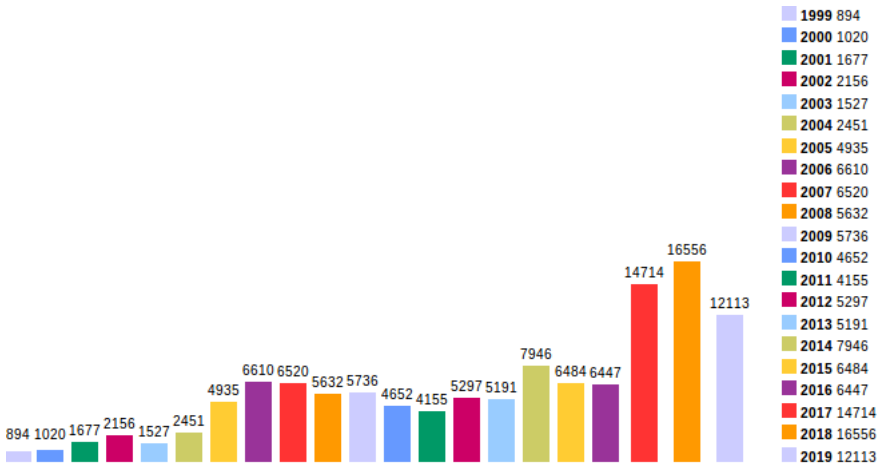

FIGURE 1.1 – Le nombre de vulnérabilités rapportées par an depuis 1999 [1]

outils de détection automatique de vulnérabilités suscitent l'intérêt de différents acteurs du secteur informatique (chercheurs, ingénieurs, développeurs, etc.).

Parmi les diverses solutions proposées, certaines visent à mettre en place de bonnes pratique de développement afin de concevoir des systèmes sûrs le plus tôt possible (Secure by design), tandis que d'autres visent à aider les développeurs à effectuer des inspections de sécurité en leur indiquant sur quoi se concentrer. Dans cette thèse, nous nous intéressons à cette deuxième catégorie, qui est l'identification et la prédiction de la vulnérabilité. Cette catégorie de techniques implique les méthodes d'analyse statique, souvent à travers l'utilisation d'outils basés sur des règles statiques tels que Find Security Bugs [11] et Fortify Static Code Analyzer [12], des outils d'analyse dynamique, tels que les outils de fuzzing [13, 14], des outils d'analyse hybride. [15-17] ou encore l'utilisation de modèles de prédiction. [18-21].

Les techniques de prédiction reposent sur l'exploration de données et l'apprentissage automatique pour deviner où se trouvent les vulnérabilités. En d'autres termes, ces techniques déterminent les composants logiciels susceptibles d'être vulnérables. Ces modèles sont communément appelés modèles de prédiction de la vulnérabilité et sont au centre de cette thèse. Plus particulièrement, dans cette thèse, nous nous intéressons à la prédiction du code vulnérable indépendamment du type de vulnérabilité. Il existe de nombreux autres modèles de prédiction, non pris en compte, comme les modèles qui prédisent le nombre de vulnérabilités subsistant dans un système logiciel [22] et des modèles détermi-

nant si les vulnérabilités sont exploitables [23].

Cependant, les modèles de prédiction de la vulnérabilité sont confrontés à des problèmes très spécifiques, à commencer par la disponibilité des données. Le concept de construction de modèles de prédiction revient à apprendre un modèle à partir d'une grande quantité de données. Dans le cas des vulnérabilités, est problématique car celles-ci sont peu nombreuses. En outre, comme les déclarations de vulnérabilités dans le code restent rares, pour des raisons compréhensibles, il est difficile d'avoir un grand ensemble de données qui soit équilibré. De ce fait, avant d'entamer toute étude sur la prédiction de vulnérabilités, il est nécessaire de collecter des données de qualité ce qui n'est pas une tâche aisée dans le contexte des vulnérabilités logicielles.

La collecte des vulnérabilités logicielles est une activité complexe et coûteuse. Plusieurs techniques sont proposées pour réduire ce coût. La plus intuitive, et probablement la plus efficace, est l'examen du code source [24]. Cependant, comme expliqué précédemment, la taille et la complexité du code rendent le processus long et sujet à des erreurs. De plus, la recherche de vulnérabilités nécessite une expertise en matière de sécurité qui reste rare chez les développeurs. Pour résoudre ce problème, les chercheurs et les industriels ont déployé beaucoup d'efforts pour trouver de nouvelles méthodes. L'une des techniques les plus répandues consiste à exploiter les correctifs des programmes (*patchs*) afin de collecter les vulnérabilités [25]. L'idée ici est de considérer le morceau de code visé par le correctif comme un prototype de la vulnérabilité, et une fois la correction appliquée, le code résultant est considéré comme débarrassé de cette vulnérabilité [26]. Cependant, il existe plusieurs façons de corriger une vulnérabilité et un *patch* ne représente qu'une solution possible. De plus, considérer le code comme vulnérable avant le *patch* et comme non vulnérable après l'application de ce *patch* est une hypothèse incorrecte. En effet, nous n'avons aucune garantie que la vulnérabilité a été correctement corrigée grâce à ce *patch* ni que le code ne contient pas d'autres types de vulnérabilités.

Une autre méthode largement utilisée par les industriels et les chercheurs consiste à exploiter les outils d'analyse statique axés sur la sécurité comme indicateurs de vulnérabilité [27-30]. Ainsi, tout composant contenant au moins un avertissement renvoyé par l'outil serait considéré comme vulnérable. Bien que ces outils soient utiles pour guider les développeurs, les utiliser pour construire une réalité terrain reste discutable, particu-

lièrement dans les études basées sur les modèles de prédiction. Ces derniers apprennent à se rapprocher au mieux des données utilisées dans leur apprentissage. Ceci revient à construire des modèles qui tentent d'imiter un outil au lieu de construire des modèles qui renvoient un réel avertissement de vulnérabilité. Cela est d'autant plus problématique que ces outils ont tendance à produire beaucoup de fausses alertes.

1.2 Défis de la thèse

Les travaux décrits dans ce manuscrit de thèse portent sur l'identification et la prédiction des vulnérabilités logicielles. La découverte de vulnérabilités, contrairement aux bogues, nécessite des compétences et des connaissances spécifiques que les développeurs ne possèdent pas nécessairement. De plus, le processus de recherche de vulnérabilités est très coûteux d'où l'intérêt des chercheurs à réduire la zone de recherche. Les outils d'analyse ainsi que les modèles de prédiction donnent des résultats prometteurs mais soulèvent toutefois quelques problèmes.

Dans ce qui suit, un aperçu des principaux défis abordés dans cette thèse est présenté.

1. **Création d'un corpus de vulnérabilités logicielles :**
   Les études empiriques dans ce domaine de recherche sont souvent confrontées à l'absence de bases de données standards sur la vulnérabilité. Ayant comme objectif de proposer une étude sur l'identification des vulnérabilités logicielles, il nous a été nécessaire de collecter un nombre conséquent de vulnérabilités. Les informations requises sont généralement dispersées dans différents endroits. À cet effet, plusieurs projets doivent être utilisés pour créer un ensemble de données qui permet la généralisation des résultats. De plus, il est primordial que les données soient annotées avec précision d'où l'intérêt de trouver une méthode automatisée avec de bonnes performances d'annotation.

2. **Annotation automatique de données :**
   Annoter des données consiste à attribuer à chaque fragment de code les vulnérabilités qu'il contient. Malheureusement, l'annotation du code en utilisant des outils d'analyse statique produit un nombre important de faux positifs. Ces derniers correspondent à des alertes renvoyées par l'outil alors que le code est non vulnérable. Cette méthode est donc inefficace lorsque les données sont utilisées comme base

d'apprentissage aux modèles de prédiction. D'où l'intérêt de trouver une approche d'annotation efficace.

3. **Création et évaluation de modèles de prédiction du code vulnérable :**
   Un autre frein important dans les études sur les vulnérabilités correspond aux ensembles de données déséquilibrés. En effet, les vulnérabilités sont très rares dans les systèmes logiciels. Exploiter ce système pour construire une base d'apprentissage mène à un déséquilibre important entre la population vulnérable et non vulnérable. Il est donc nécessaire de proposer une méthode d'évaluation adéquate pour permettre la généralisation des résultats.

1.3 Contributions de la thèse

Le travail de cette thèse vise spécifiquement les défis décrits ci-dessus. Nos réponses à ces défis sont matérialisées par les contributions suivantes :

1. **Conception d'un méta-scanner de vulnérabilités logicielles :**
   Tout travail de recherche sur les vulnérabilités a besoin d'un corpus dont les vulnérabilités sont connues et clairement spécifiées. La création d'un ensemble de données est intéressante car elle garantit un contrôle et une compréhension totale des données. Néanmoins, comme de nouvelles vulnérabilités sont découvertes quotidiennement, un ensemble de données peut rapidement être dépassé et fausser les résultats. Pour remédier à cette problématique, nous proposons non seulement un corpus de vulnérabilités mais aussi une approche de conception de méta-scanner permettant l'annotation automatique de code source. La méthode décrite dans ce manuscrit s'appuie sur différents outils existants simultanément afin de produire un méta-scanner qui tire profit des points forts de chaque outil utilisé.

2. **Création d'un corpus de vulnérabilités logicielles :**
   Une fois que l'approche de conception de méta-scanner validée, nous l'utilisons pour construire un corpus de sécurité. Ce processus comprend trois étapes importantes : (i) trouver le bon ensemble d'applications Java, (ii) annoter cet ensemble efficacement et (iii) valider les résultats obtenus. En suivant ce processus, nous obtenons un corpus de sécurité que nous appelons "SecureQualitas". Ce corpus contient un ensemble conséquent de projets libres, écrits en langage Java et annotés avec les

vulnérabilités qu'ils contiennent.

3. **Prédiction du code vulnérable :**
   Le corpus "SecureQualitas" a été utilisé pour construire des modèles de prédiction pour le code vulnérable. Ce corpus est réaliste et reflète parfaitement la rareté des vulnérabilités. Il s'agit d'un ensemble de données déséquilibré où la population vulnérable représente un pourcentage beaucoup moins important que la population non vulnérable. Les modèles que nous voulons construire ont pour objectif de maximiser leurs performances sur les deux populations (vulnérable et non vulnérable). Par conséquent, nous proposons de sélectionner les modèles en utilisant les critères suivants à maximiser : la précision et le rappel sur la population vulnérable et non vulnérable. Il s'agit ici d'une validation multi-objectifs que nous résolvons en utilisant l'optimum de Pareto.

1.4 Structure de la thèse

Cette thèse est composée de 4 parties majeures :

— **Partie 1 : Introduction**
  Cette partie comprend le chapitre présent qui introduit le contexte de la thèse et la problématique induite par les vulnérabilités logicielles.

— **Partie 2 : Contexte de travail et état de l'art**
  Cette partie comprend 3 chapitres qui passent en revue les travaux existants relatifs à la découverte et l'identification de vulnérabilités logicielles :

  — Le chapitre 2 donne plus de détails sur les vulnérabilités logicielles en présentant des exemples de vulnérabilités et des exploits potentiels.

  — Le chapitre 3 présente les approches utilisées pour découvrir les vulnérabilités. Les travaux de recherche liés à ce domaine sont également détaillés.

  — Le chapitre 4 présente le processus de construction de modèles de prédiction des vulnérabilités en mettant l'accent sur les défis liés à la prédiction des vulné-

rabilités. Le processus est suivi par une présentation des travaux de recherche sur les modèles de prédiction de vulnérabilités.

— **Partie 3 : Contributions**
La partie "Contributions" contient trois chapitres :

— Le chapitre 6 décrit la méthode de conception d'un méta-scanner de vulnérabilités logicielles. Les étapes de ce processus sont illustrées à travers un prototype utilisant trois outils d'analyse de vulnérabilités existants.

— Le chapitre 7 décrit notre seconde contribution, à savoir le corpus "SecureQualitas". Ce corpus est décrit par le nombre de vulnérabilités qu'il contient, le nombre d'instances de chaque vulnérabilité et les résultats de notre évaluation quantitative et qualitative.

— Le chapitre 8 présente notre travail sur la construction et l'évaluation des modèles de prédiction du code vulnérable. L'évaluation proposée dans ce chapitre s'adapte au problème des bases de données déséquilibrées.

— **Partie 4 : Conclusion**
Enfin, le dernier chapitre 9 conclut cette thèse en fournissant un résumé des contributions ainsi que des perspectives sur les travaux futurs.

Deuxième partie

Contexte de travail et état de l'art

Chapitre 2

Vulnérabilités logicielles : généralités

Dans ce chapitre, nous commençons par présenter une vue d'ensemble du large domaine de la cybersécurité avant d'y positionner nos travaux. Par la suite, nous présentons des exemples de vulnérabilités logicielles en expliquant comment ces vulnérabilités peuvent être exploitées et les conséquences de tels exploits. Enfin, les principales causes de vulnérabilités et les bonnes pratiques qui aident à les éviter sont détaillées.

Sommaire

2.1 Cybersécurité et vulnérabilités

Le nombre croissant des systèmes informatiques interconnectés, et la dépendance croissante à leur égard, que ce soit de la part des particuliers, des entreprises ou des entités industrielles et gouvernementales, signifie qu'un nombre important de systèmes est à risque. La cybersécurité vise à protéger les systèmes informatiques contre le vol et l'endommagement de leur matériel, de leurs logiciels ou des informations qu'ils contiennent. Ce domaine est d'une grande importance pour tous les acteurs des technologies de l'information, du simple utilisateur à domicile aux grandes agences gouvernementales. Comme le montre la Figure 2.1 [31], ce domaine est très vaste et le devient de plus en plus. La cybersécurité couvre des domaines techniques tels que les opérations de sécurité, l'architecture de sécurité ou l'évaluation des risques, des domaines réglementaires tels que la gouvernance ou la conformité aux normes et des domaines liés aux ressources humaines tels que le développement de carrière ou l'éducation des utilisateurs. Nous nous concentrons sur le domaine de l'évaluation des risques et plus particulièrement sur les sous-domaines de l'analyse des vulnérabilités et de l'analyse du code source.

L'analyse de vulnérabilité est généralement effectuée à l'aide d'outils automatisés qui signalent les vulnérabilités déjà connues. Ces vulnérabilités connues sont publiées et documentées dans différentes bases de données en ligne. Parmi celles-ci, la plus célèbre est la liste des identificateurs communs de vulnérabilité **CVE** (*Common Vulnerabilities and Exposures*) [1], qui est gérée par l'organisation **MITRE** [2] avec le financement de la division nationale de la sécurité informatique du gouvernement américain.

Le base CVE attribue à chaque vulnérabilité rendue publique un identifiant unique qui peut être utilisé pour récupérer des informations sur celle-ci. La base CVE est le dictionnaire qui associe à tous ces identificateurs une brève description de la vulnérabilité et toute référence pertinente à celle-ci. Bien que le système CVE soit assez connu, il s'avère que la plupart des personnes qui le mentionnent se réfèrent en réalité à la base de données nationale des vulnérabilités **NVD** (*National Vulnerability Database*) [32]. Cette dernière est une base de données établie par le **NIST** (*National Institute of standards and Technology*) [33] et le gouvernement américain afin d'encourager le développement de logiciels sécurisés, la divulgation publique et la gestion des vulnérabilités. Le NVD enrichit chaque entrée du CVE avec des informations telles que la gravité (appelée CVSS) et le type (ap-

FIGURE 2.1 – Cartographie des domaines de la cybersécurité [31]

pelé CWE) d'une vulnérabilité. En outre, les données sont continuellement mises à jour par le personnel du NVD.

Parmi les informations trouvées dans le NVD, deux sont particulièrement intéressantes : le *Common Vulnerability Scoring System* (**CVSS**) et le *Common Weakness Enumeration* (**CWE**). Le système CVSS saisit les principales caractéristiques d'une vulnérabilité et produit une note numérique allant de 0 à 10 reflétant sa gravité. Cette note peut être traduite en une représentation qualitative, telle que *faible* [0 - 4[, *moyenne* [4,0 - 7[, *élevée* [7,0 - 9[et *critique* [9,0 - 10,0] pour aider à hiérarchiser la vulnérabilité. La version actuelle du CVSS est la 3.0 qui a été publiée en juin 2015 [34]. Le CWE est un système de catégorisation des faiblesses et des vulnérabilités des logiciels. Il est piloté par un projet communautaire dont l'objectif est de comprendre les failles logicielles. Il est utilisé par le NVD pour catégoriser les vulnérabilités et est actuellement en version 3.1 [35].

L'analyse des vulnérabilités doit être effectuée en continu pour s'assurer que le système d'information est au moins immunisé contre les vulnérabilités connues. D'autre part, l'analyse des codes logiciels se fait soit dans une boîte blanche avec un accès au code source,

soit dans une boîte noire avec un accès au code binaire des logiciels qui peut être utilisé pour découvrir des vulnérabilités généralement nouvelles (appelées "zero days" par la communauté de la sécurité). Le résultat de l'analyse est un ensemble d'alertes ou d'avertissements expliquant où se trouve la vulnérabilité. L'utilisation de la taxonomie CWE (Common Weakness Enumeration) [21] permet aux auditeurs de logiciels d'utiliser facilement différents outils ou techniques et de conserver le même vocabulaire pour décrire les mêmes choses.

Dans la suite de ce manuscrit, nous nous appuyons sur le CWE comme source d'information et l'utilisons pour récupérer la catégorie d'une vulnérabilité.

2.2 Vulnérabilités logicielles

Les vulnérabilités informatiques peuvent être classées dans différentes classes, les plus importantes sont les vulnérabilités matérielles, logicielles ou de réseau. Il existe certes d'autres types de vulnérabilités telles que les vulnérabilités liées au personnel ou au site physique où sont stockées les machines. Dans ce manuscrit, nous nous intéressons aux vulnérabilités logicielles car elles sont responsables de la majorités des exploits. De plus, une vulnérabilité logicielle peut entraîner plusieurs autres vulnérabilités ce qui est appelé un effet cascade.

2.3 Exemples de Vulnérabilités

Pour avoir une idée précise sur les vulnérabilités logicielles, nous présentons quelques exemples de vulnérabilités accompagnées de leurs descriptions et d'exemples d'exploits possibles.

2.3.1 Débordement de tampon

Le débordement de tampon, aussi appelé dépassement de tampon (en anglais, buffer overflow) est une vulnérabilité qui se produit lorsque l'écriture dans un tampon dépasse la taille autorisée. En conséquence, des informations se retrouvant après le tampon sont écrasées. Cela peut entraîner un mauvais fonctionnement du programme, voire de tout le système, puisque les nouvelles données peuvent corrompre les données d'autres tampons ou processus. Le débordement de la mémoire tampon peut également être utilisé inten-

tionnellement pour injecter un code malveillant afin de modifier l'exécution normale du programme et de prendre le contrôle du système. Le langage de programmation C est particulièrement affecté par cette vulnérabilité en raison de sa gestion dynamique de la mémoire. En effet, certaines applications critiques comme l'aéronautique interdisent l'utilisation de pointeurs ou d'allocations dynamiques de mémoire pour éviter ces problèmes. Par exemple, dans le programme C suivant :

```
int main(int argc, char **argv) {
char buffer[1024];
strcpy (buffer, argv[1]);
}
```

Le paramètre *argv[1]* peut contenir plus de 1024 caractères. La fonction *strcpy* copiera le contenu de *argv[1]* dans la variable *buffer* sans aucune vérification : elle copie le contenu de la chaîne de caractères jusqu'à ce qu'elle rencontre un caractère de fin de chaîne (caractère nul).

Cette vulnérabilité peut être exploitée de différentes manières. Lorsque la fonction *strcpy* est appelée dans une méthode, elle peut écraser l'adresse de retour au programme initial. Ceci permet de bloquer le processus ou de contrôler l'exécution. Cet exploit est illustré par la Figure 2.2 qui présente la pile d'exécution d'un programme lors de la copie d'une chaîne de caractères d'une taille inférieure à la taille du buffer (vulnérabilité non exploitée) et une autre pile d'exécution lorsque la taille de la chaîne est supérieure à celle du buffer (vulnérabilité exploitée).

Un autre exploit possible serait de copier une chaîne longue qui finit par le chiffre "1" avant un booléen important. Il en résulte que la valeur "1" sera affectée à ce booléen ce qui pourrait représenter un réel danger selon l'importance du booléen.

2.3.2 Injection SQL

Toute application qui utilise une base de données SQL doit être protégée contre l'injection SQL (SQLi). Un attaquant peut obtenir des informations sensibles de la base de données en injectant des entrées élaborées non prévues par le système et pouvant en compromettre la sécurité. Si les entrées ne sont pas bien filtrées, elles peuvent être exécutées par l'interpréteur SQL et exposer le contenu de la base de données. Par exemple, si une

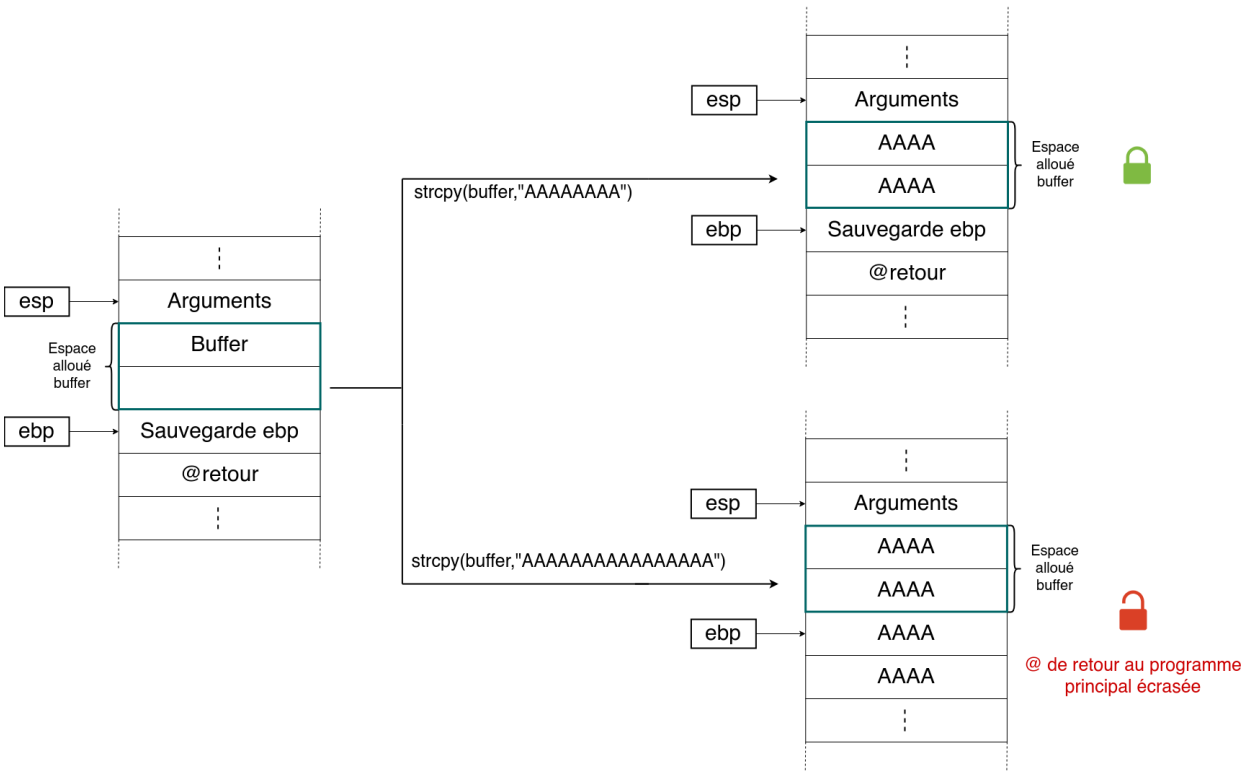

FIGURE 2.2 – Exemple d'un débordement de tampon suite à l'utilisation de la fonction *strcpy*

application demande à l'utilisateur de saisir son nom pour se connecter et que l'attaquant entre le texte suivant : ' OR '1' = '1, alors l'application peut exécuter la commande SQL suivante :

*SELECT * FROM users WHERE name = '' OR '1'='1';*

Ainsi, l'attaquant obtiendra un nom d'utilisateur valide puisque l'évaluation de l'énoncé '1'='1' est toujours vraie. De la même manière, l'attaquant peut obtenir des informations confidentielles, modifier le contenu ou même supprimer les enregistrements de la base de données ayant un impact sur le service et/ou l'activité. Un autre exemple très simple d'une injection SQL serait d'introduire, dans le champ alloué au nom, une chaîne de caractères suivie de *';–* comme dans ce qui suit :

SELECT uid FROM Users WHERE name = 'Dupont';–' AND password = 'XXXX';

Les caractères – marquent le début d'un commentaire en SQL. La requête est donc équivalente à :

SELECT uid FROM Users WHERE name = 'Dupont';

L'attaquant peut alors se connecter sous l'utilisateur Dupont avec n'importe quel mot de passe. Ceci représente une injection SQL réussie.

2.3.3 Cross-Site Scripting

Le cross-site scripting (abrégé XSS) est une vulnérabilité associée aux applications web permettant d'injecter du code dans les pages web qui sont accessibles à d'autres utilisateurs. L'attaquant peut exploiter cette faille pour contourner les contrôles d'accès, pratiquer l'hameçonnage, l'usurpation d'identité ou mettre à jour les connexions. Cette vulnérabilité est très répandue et se produit partout où une application web utilise les données d'un utilisateur sans les valider. Un attaquant peut exploiter la faille XSS pour envoyer un script malveillant à un utilisateur sans méfiance. Le navigateur de l'utilisateur exécutera le script ce qui permettra à l'attaquant d'accéder à n'importe quelles informations sensibles enregistrées par le navigateur et utilisées avec ce site.

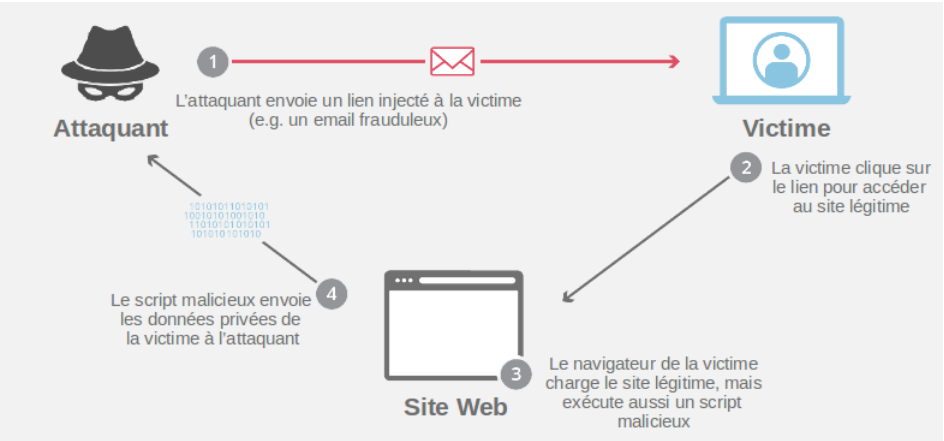

FIGURE 2.3 – Exemple d'une vulnérabilité XSS exploitée par un attaquant

2.4 Causes de Vulnérabilités

Il n'existe malheureusement pas de source unique de vulnérabilités en matière de sécurité. Les vulnérabilités se produisent à chaque étape du cycle de vie du développement logiciel (Software Development Life Cycle : SDLC). Au stade du développement, ces vulnérabilités peuvent se localiser dans n'importe quelle portion de code. Une vulnérabilité peut rester endormie pendant des décennies jusqu'à ce qu'elle soit découverte et exploitée, ce qui peut entraîner des dommages importants. Par exemple, la vulnérabilité Heartbleed [1] [36] qui a affecté la bibliothèque cryptographique, OpenSSL, largement utilisée, est restée dans le code pendant au moins une décennie avant d'être reconnue publiquement et corrigée.

Les principales sources et causes des vulnérabilités de sécurité sont les suivantes :

— **Conception non sécurisée :**
Beaucoup de concepteurs de logiciels se focalisent sur le bon fonctionnement d'un logiciel mais négligent sa sécurité. De plus, la sécurité est souvent mise en contraste avec la facilité d'utilisation, c'est-à-dire que plus le logiciel est sûr, moins il est facile à utiliser et vice versa. Les concepteurs qui acceptent ce compromis sacrifient généralement la sécurité au détriment de la facilité d'utilisation. Par exemple, les principaux protocoles Internet ont été conçus avec très peu de sécurité mais visaient la facilité d'utilisation.

— **Langages intrinsèquement peu sécurisés :**
L'utilisation de certains langages dans le développement logiciel rend plus probable la possibilité de commettre une erreur conduisant à une vulnérabilité. Cela est dû à la complexité et/ou à l'ambiguïté des spécifications du langage. Le langage C est un exemple de langage dans lequel il est facile de commettre une erreur conduisant à une vulnérabilité de sécurité [37]. Le choix du langage de programmation est parfois un bon point de départ pour éliminer certains types de vulnérabilités.

1. Heartbleed est une vulnérabilité logicielle présente dans la bibliothèque de cryptographie open source OpenSSL à partir de mars 2012, qui permet à un « attaquant » de lire la mémoire d'un serveur ou d'un client pour récupérer, par exemple, les clés privées utilisées lors d'une communication avec le protocole Transport Layer Security (TLS). Découverte en mars 2014 et rendue publique le 7 avril 2014, elle concerne de nombreux services Internet. Environ un demi-million de serveurs auraient été touchés par la faille au moment de la découverte du bogue.

— **Erreurs et défaillances :**
Une erreur de copier-coller [38], une erreur syntaxique dans les noms de variables et toutes les autres erreurs qu'un humain peut éventuellement commettre peuvent être à l'origine d'une faille de sécurité. Bien entendu, toutes les erreurs ne conduisent pas à des vulnérabilités. Ces erreurs peuvent conduire à des bogues qui sont la classe mère des vulnérabilités de sécurité.

— **Des tests médiocres :**
Chaque logiciel doit être testé et validé avant d'être déployé [39]. Une mauvaise méthodologie de test peut permettre de passer à côté de failles de sécurité triviales. L'exploitation des failles de sécurité sur des logiciels mal testés peut être très facile et pratiquement sans coût pour un adversaire. D'autre part, un logiciel bien testé peut rendre le coût d'une attaque réussie si élevé qu'il pousse l'attaquant à abandonner et à rechercher d'autres points d'attaque.

— **Délibéré :**
Dans ce cas, la vulnérabilité de sécurité est considérée comme une caractéristique logicielle cachée et insérée par une personne malveillante [40]. Ce type de vulnérabilités peut être très difficile à éradiquer, surtout si le responsable est très compétent et déterminé.

2.5 Remédiation

Pour éradiquer la vulnérabilité logicielle, différentes actions et décisions doivent être prises dans chaque étape du cycle de vie du logiciel. Ces décisions comprennent notamment le choix de bonnes pratiques de codage, un langage sécurisé tel que Java, des règles de codage telles que CERT Coding Standards [41] et plus particulièrement le test et l'analyse du logiciel. Toutes ces bonnes pratiques permettent de réduire le nombre de vulnérabilités. Toutefois, elles ne garantissent pas que le logiciel en sortie sera immunisé de toute vulnérabilité. À cet effet, il est important de développer des méthodes d'analyse et de test efficaces afin de détecter le maximum de vulnérabilités. Dans la section suivante, nous détaillons les techniques utilisées pour analyser un code logiciel dans le but d'identifier les vulnérabilités qu'il contient.

2.6 Résumé et discussion

Avant d'entamer une étude sur les vulnérabilités logicielles, il est important de comprendre les notions de vulnérabilité et d'exploit ainsi que leurs conséquences sur la confidentialité, l'intégrité et la disponibilités des systèmes logiciels. Le présent chapitre a présenté plusieurs exemples de vulnérabilités logicielles et de potentiels exploits. Nous avons également cité les causes des vulnérabilités et les bonnes pratiques qui peuvent les corriger. Bien que ces pratiques soient utiles pour éviter quelques vulnérabilités, leur utilisation n'est nullement suffisante pour assurer la correction du code logiciel. Il est donc important d'analyser continuellement le code, que ce soit avant ou après son déploiement, afin de découvrir les vulnérabilités le plus tôt possible. Le chapitre suivant détaille les approches d'analyse de code source existantes en présentant leurs points forts et faibles.

Chapitre 3

Identification de vulnérabilités

Les travaux présentés dans ce manuscrit sont liés à deux domaines de recherche : (i) l'identification de vulnérabilités et (ii) la prédiction du code vulnérable. Ces deux domaines comprennent diverses techniques et approches qui ont été proposées au fil du temps. Dans ce chapitre, nous commençons par le premier domaine de recherche, à savoir l'analyse et l'identification de vulnérabilités. Nous présentons les différentes approches utilisées avec les travaux associés.

Sommaire

3.1 Méthodes d'analyse logicielle

Trouver des vulnérabilités manuellement est une tâche fastidieuse qui demande du temps et des compétences spécifiques que la plupart des fournisseurs de logiciels ne peuvent pas se permettre. En raison de la criticité de ce problème, beaucoup d'efforts ont été investis au fil des ans pour suggérer et développer des techniques de découverte automatique des vulnérabilités. Pour l'instant, trouver une approche parfaite qui permet de détecter les vulnérabilités de manière solide et complète, c'est-à-dire détecter toutes les vulnérabilités sans fausses alertes, est impossible. Ainsi, la plupart des recherches actuelles se concentrent sur la suggestion d'approches qui permettent une meilleure identification des vulnérabilités que celles qui les précèdent. Ces méthodes peuvent généralement être classées en trois catégories, l'analyse statique, l'analyse dynamique et l'analyse hybride.

3.1.1 Analyse statique

Cette catégorie de méthodes analyse directement le code source du programme sans l'exécuter afin de déterminer la qualité du logiciel. L'analyse peut être effectuée à plusieurs niveaux du programme qui comprennent, entre autre, le niveau "*unité*" qui se concentre sur une portion de code sans prendre en compte le contexte général du programme et le niveau "*système*" où la globalité du système est analysée avec les différentes relations existantes entre les unités. Ce type d'analyse est susceptible de trouver un grand nombre de vulnérabilités, mais a souvent pour défaut de retourner beaucoup de fausses alertes.

L'analyse statique de code peut être effectuée manuellement, communément appelée la revue de code ou inspection de code [42, 43], ou bien en utilisant des outils automatisés (*Static Application Security Testing (SAST)*) [44]. Les outils d'analyse statique analysent le code source ou une version compilée et s'appuient généralement sur un ensemble de règles existantes ou sur un ensemble de patterns qui permettent de mettre en évidence les vulnérabilités. Parmi ces outils, nous pouvons citer FindSecBugs [11] et Fortify Static Analyzer [12].

Un autre type d'approche bien connu qui entre dans cette catégorie est l'analyse des flux de données corrompues [45-48] où les variables "contaminées" par des entrées contrôlables par l'utilisateur sont identifiées. Les fonctions qui sont reliées à ces variables, appelées "puits", sont considérées comme étant potentiellement vulnérables. Si une variable

altérée est transmise à un "puits" sans avoir été préalablement nettoyée, elle est signalée comme une vulnérabilité.

Parmi les travaux qui ont introduit de nouveaux outils d'analyse de vulnérabilités, nous retrouvons l'étude pertinente de Livshits et al. dans [28] où les auteurs proposent une technique d'analyse statique appliquée aux applications Java. Leur approche vise à traduire automatiquement les spécifications données par l'utilisateur en un analyseur statique. Les résultats de l'analyse sont intégrés dans Eclipse rendant les potentielles vulnérabilités faciles à examiner et à corriger dans le cadre du processus de développement. Cependant, cette approche est fortement dépendante des spécifications fournies par l'utilisateur, qui sont sujettes aux erreurs et parfois incomplètes.

L'étude de DaCosta et al. [29] est l'une des premières sur la recherche automatisée des vulnérabilités. Les auteurs suggèrent que les fonctions proches d'une source d'entrée, telles que les entrées/sorties de fichiers, sont les plus susceptibles de contenir une vulnérabilité. La proximité est mesurée par le nombre d'invocations de fonctions qui se produisent entre l'entrée et la cible (fonction vulnérable). Les auteurs ont donc construit un outil nommé FLF (Front Line Functions) détectant de telles fonctions. L'outil est évalué sur 31 vulnérabilités appartenant à 30 projets open source. L'évaluation est différente des études précédentes car les auteurs proposent une nouvelle mesure basée sur la densité FLF. Cette métrique représente le pourcentage de fonctions potentiellement vulnérables dans un système logiciel (i.e, fonctions qui transmettent des données à partir d'une entrée). Les expérimentations sont effectuées en deux étapes : (i) une première évaluation de l'outil sur 3 projets open-source présentant des vulnérabilités connues et documentées, et (ii) une évaluation sur le projet du serveur OpenSSH qui ne présente aucune vulnérabilité connue mais qui avait subi une reconstruction très médiatisée, appelée séparation des privilèges. Cette séparation vise à minimiser la quantité de code qui s'exécute avec des privilèges élevés. En minimisant la quantité de code privilégié, cela réduit le risque qu'une vulnérabilité de sécurité se produise dans ce code. Les résultats montrent que la densité moyenne des FLF de l'échantillon était de 2,87% avec un écart-type de 1,83%.

Yamaguchi et al. [30] présentent une nouvelle approche de découverte des vulnérabilités dans le code source. Leur approche consiste à extraire les appels API d'un logiciel pour créer un espace vectoriel, qui ensuite, en calculant la similarité, peut être utilisé

pour analyser les appels ressemblant à une vulnérabilité signalée. Les auteurs évaluent l'approche sur un ensemble de données extraites du noyau Linux et ffmpeg et trouvent 2 nouvelles vulnérabilités et un exploit. Les auteurs étendent cette approche dans [49] à l'analyse de l'arbre syntaxique abstrait (AST). Ils l'évaluent sur un nouvel ensemble de données et trouvent plusieurs vulnérabilités 0-day.

L'un des principaux avantages des outils d'analyse statique est leur extensibilité et leur résultat compréhensible. Ils sont particulièrement bien adaptés pour trouver des vulnérabilités comme le débordement de tampon et l'injection SQL. Cependant, ces outils sont peu fiables car il est difficile de déterminer si les alertes retournées sont réelles d'autant plus que le nombre de fausses alertes est très important. Certaines vulnérabilités telles que les problèmes d'authentification ou de contrôle d'accès ne sont généralement pas découverts avec des outils d'analyse statique.

3.1.2 Analyse dynamique

L'analyse dynamique surveille le comportement d'un programme lorsqu'il est exécuté avec un ensemble spécifique d'entrées. Pour que l'analyse dynamique de programme soit efficace, le programme cible doit être exécuté avec suffisamment d'entrées de test pour couvrir presque toutes les sorties possibles. Néanmoins, comme le nombre d'entrées possibles est infini, des mesures de test du logiciel, telles que la couverture du code, sont utilisées pour s'assurer qu'une portion adéquate de l'ensemble des comportements possibles du programmes a été couverte. À l'inverse de l'analyse statique, les vulnérabilités que produit l'analyse dynamique sont moins nombreuses mais plus réelles.

Le type d'approche le plus connu dans cette catégorie est le *fuzzing*, ou le test à données aléatoires [13, 14]. Il s'agit d'une technique automatisée de test logiciel où des entrées aléatoires sont injectées au programme pour le tester. Le programme est ensuite contrôlé pour détecter des exceptions telles que le plantage ou la génération d'erreur. En général, cette méthode est utilisée pour tester les programmes qui prennent des entrées structurées. Cette structure est spécifiée, par exemple, dans un format de fichier ou un protocole et distingue les entrées valides des entrées non valides. Un *fuzzer* efficace génère des entrées semi-valides qui sont "suffisamment valides" pour qu'elles ne soient pas directement rejetées par l'analyseur, mais "suffisamment invalides" pour créer des comportements inattendus dans le programme et ainsi mettre en avant les défauts à corriger.

Dans cette catégorie, nous retrouvons le travail de Huang et al. [50] qui proposent WAVES, une technique dynamique pour tester les vulnérabilités des applications web afin de détecter les injections SQL. La technique est basée sur un processus de rétro-ingénierie qui identifie les points d'entrée des données de l'application et les attaques en utilisant des modèles malveillants. Un algorithme est proposé pour éliminer les faux négatifs. Pendant la phase d'attaque, les réponses de l'application aux attaques sont surveillées et des techniques d'apprentissage machine sont utilisées pour améliorer la méthodologie d'attaque.

Par ailleurs, Kals et. al. [51] introduisent un outil d'analyse de vulnérabilités web nommé SecuBat. L'outil utilise une approche dynamique pour explorer et analyser les sites web afin de détecter la présence de vulnérabilités exploitables par injection SQL et XSS. SecuBat ne s'appuie pas sur une base de données de défauts connus. Après le lancement de l'attaque, la réponse est analysée et préparée afin de trouver des critères de réponse et des mots clés spécifiques à l'attaque. Les faux positifs sont possibles, ce qui nécessite un réglage minutieux du seuil de la valeur de confiance. Le problème des deux dernières études est que WAVES et SecuBat ne peuvent être appliqués qu'à des applications web.

3.1.3 Analyse hybride

Cette dernière combine l'analyse statique et dynamique pour obtenir le meilleur des deux méthodes. Il est possible de croire qu'une telle combinaison pourrait retourner des vulnérabilités correctes et complètes. La combinaison peut se faire dans les deux sens, soit en utilisant l'analyse dynamique comme moyen d'élaguer les résultats d'une analyse statique, soit en utilisant l'analyse statique pour guider l'analyse dynamique. Malheureusement, si les approches d'analyse hybride peuvent bénéficier des avantages de l'analyse statique et dynamique, elles souffrent également des limites des deux approches.

Parmi les travaux de recherche combinant l'analyse statique et dynamique, nous retrouvons le travail de Balzarotti et al. [15] qui introduit une approche hybride pour analyser le processus de sanitisation dans les applications web. Tout d'abord, une technique basée sur l'analyse statique modélise les modifications que subissent les entrées le long du code. Cette approche utilise un modèle d'opérations sur les chaînes de caractères, ce qui

peut conduire à des faux positifs. Ensuite, une technique basée sur l'analyse dynamique permet, en partant des puits, de reconstruire le code utilisé afin de modifier les entrées à injecter.

D'autres techniques de détection des attaques par injection SQL nécessitent que les développeurs de l'application modifient le code afin d'y insérer les mécanismes de détection. SQLCheck, présenté par Su et al. [16], vérifie les requêtes SQL au moment de l'exécution pour voir si elles sont conformes à un modèle de requêtes SQL attendues. Le modèle est exprimé sous la forme d'une grammaire qui n'accepte que les requêtes légales. Les parties fournies par l'utilisateur sont marquées dans les requêtes avec une clé secrète. Un analyseur syntaxique est généré sur la base de la grammaire. Au moment de l'exécution, l'analyseur vérifie la requête générée et lorsqu'une attaque par injection SQL est détectée, la requête est rejetée. La sécurité dépend des clés générées, et l'approche exige que le développeur réécrive le code pour insérer manuellement les clés secrètes dans les requêtes SQL générées dynamiquement. Un autre problème de cette technique est qu'elle introduit une complexité supplémentaire dans la phase de développement, ce qui limite son applicabilité.

Les outils de détection des anomalies d'exécution peuvent être utilisés à la fois pour la détection des attaques et des vulnérabilités. L'un de ces outils est AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks) [17] qui combine l'analyse statique et la surveillance de l'exécution pour détecter et éviter les attaques par injection SQL. L'analyse statique est utilisée pour analyser le code source d'une application web donnée en construisant un modèle des requêtes légitimes que cette application peut générer. Au moment de l'exécution, AMNESIA surveille toutes les requêtes générées dynamiquement et vérifie leur conformité avec le modèle généré statiquement. Lorsqu'une requête qui viole le modèle est détectée, elle est classée comme une attaque et ne peut pas accéder à la base de données. Le problème est que le modèle construit lors de l'analyse du code statique peut être incomplet car il manque une vue dynamique du comportement d'exécution.

Par ailleurs, Antunes et al. [52] présentent une nouvelle approche générique pour la conception d'outils de test de vulnérabilités pour les services web. L'approche est basée sur le développement modulaire et définit les composants et la procédure de test qu'un outil doit mettre en œuvre. Les composants comprennent un émulateur de charge de

travail (chargé de générer et d'exécuter un ensemble de requêtes pour exercer le service web), un émulateur d'attaque (chargé de générer et d'injecter des requêtes qui simulent des attaques), un moniteur de service (chargé d'instrumenter le service testé, si nécessaire, et de collecter des informations pertinentes pour l'identification des vulnérabilités), et un détecteur de vulnérabilité (chargé d'analyser les informations collectées et d'identifier les vulnérabilités, et d'exécuter la procédure de test).

Sur la base de l'approche proposée, trois outils de test innovants sont présentés. Ces derniers mettent en œuvre trois techniques complémentaires : tests de pénétration améliorés, signatures d'attaques et surveillance des interfaces, et détection des anomalies d'exécution, offrant ainsi un support étendu pour différents scénarios. Une étude de cas a été conçue pour démontrer les outils pour le cas particulier des vulnérabilités d'injection SQL. L'évaluation expérimentale démontre que les outils peuvent être utilisés efficacement dans différents scénarios et qu'ils surpassent les outils commerciaux connus en obtenant une couverture de détection plus élevée et un taux de faux positifs plus faible.

L'évaluation de l'approche comprend une analyse de la modularité des outils mis en œuvre et une étude de cas destinée à démontrer que les outils mis en œuvre sont capables de fonctionner aussi bien que les outils existants. L'analyse de la modularité devrait montrer la grande cohésion des modules définis et le couplage lâche entre eux. L'étude de cas utilise un ensemble de référence de 80 opérations provenant de 21 services (adopté à partir du benchmark des outils de détection de la vulnérabilité présenté dans [8]). Bien que l'étude de cas se concentre sur l'injection SQL, la plupart des concepts et techniques présentés peuvent être facilement adaptés à d'autres types de vulnérabilités d'injection. Les résultats montrent clairement que les trois techniques de détection mises en œuvre ont des performances différentes selon les conditions d'accès aux services web. Ces résultats ont été comparés à ceux obtenus par trois outils d'analyse de vulnérabilité commerciaux, et ont pu montrer que les outils conçus dans cette étude sont plus performants que les outils commerciaux.

3.2 Évaluation et comparaison d'outils

Dès l'apparition des premiers outils d'analyse de vulnérabilités, les chercheurs ont commencé à étudier leurs performances, leurs zones de couverture ou encore les comparer afin d'identifier les plus efficaces pour chaque type d'analyse.

Par exemple, Autunes et al [53] introduisent une approche pour évaluer l'efficacité des outils d'analyse statique et des tests de pénétration automatiques. L'étude s'est principalement concentrée sur les injections SQL dans les services web. Les auteurs ont montré que les approches d'analyse statique sont plus efficaces que les tests de pénétration automatiques. Ils ont également constaté que les deux méthodes renvoyaient un grand nombre de faux positifs. Les auteurs étendent leur étude en proposant une approche standard d'évaluer et comparer les performances des outils. L'approche s'appuie principalement sur l'utilisation d'un corpus de vulnérabilités fiable et des mesures de performances qui comprennent la précision, le rappel et la f-mesure. Ils utilisent cette approche pour comparer des outils d'analyse statique, des outils de tests de pénétrations et des outils de détection d'anomalies.

Fonseca et al. [54] présentent une nouvelle approche d'évaluation des outils d'analyse de vulnérabilités dans les applications web. Tout comme l'étude précédente, les auteurs se concentrent principalement sur les injections SQL et les vulnérabilités XSS car ces deux vulnérabilités sont responsables de la plupart des exploits dans les applications web. L'approche proposée est basée sur l'injection de fautes logicielles réalistes dans les applications afin d'évaluer l'efficacité des outils dans la détection des vulnérabilités causées par les fautes injectées. Les auteurs se basent sur l'étude de Duraes et al. [55] où les 12 fautes les plus fréquentes sont présentées. Sur les 12 types de fautes, seuls 6 ont produit de réelles vulnérabilités. L'évaluation de cette approche a été réalisée en utilisant les 3 principaux outils d'analyse de vulnérabilités des applications web. Les résultats montrent que les outils produisent des résultats très différents et que tous les outils laissent un pourcentage considérable de vulnérabilités non détectées. Le pourcentage de faux positifs est très élevé, allant de 20% à 77% dans les expériences réalisées. Les résultats obtenus montrent également que l'approche proposée permet de comparer facilement la couverture et les faux positifs des outils d'analyse des vulnérabilités web.

Austin et al. [56] étudient ces techniques et leur complémentarité sur Tolven, CCHR et Open EMR, c'est-à-dire le test de pénétration manuel exploratoire, l'analyse statique, le test de pénétration automatisé et le test de pénétration manuel systématique, ce dernier trouvant le plus de vulnérabilités tout en observant que les vulnérabilités trouvées par les outils d'analyse statique étaient de différentes sortes.

Ranganath et al. [57] présentent une étude sur l'évaluation des outils de détection de vulnérabilités pour les applications Android. Les auteurs commencent leur étude par examiner 64 outils puis évaluent de façon empirique 14 outils de détection de vulnérabilités et 5 outils de détection de comportements malveillants. L'évaluation s'appuie sur le benchmark Ghera comprenant 42 vulnérabilités connues et qui est composé à la fois d'applications vulnérables et sécurisées pour chaque vulnérabilité. Parmi les 24 observations d'évaluation mises en place par les auteurs, la principale montre que les outils de détection de vulnérabilité existants pour les applications Android sont très limités dans leur capacité à détecter les vulnérabilités connues, malgré les efforts considérables pour développer de nouveaux outils, d'autant plus que la majorité des vulnérabilités considérées ont été découvertes et signalées avant que les outils ne soient développés. Contrairement aux attentes des auteurs, l'évaluation suggère que la plupart des outils et techniques ne peuvent détecter indépendamment qu'un petit nombre de vulnérabilités considérées. Même en regroupant tous les outils, plusieurs vulnérabilités sont restées non détectées. En effet, tous les outils ensemble ne pouvaient détecter que 30 des 42 vulnérabilités couvertes par l'étude.

3.3 Combinaison d'outils

Il existe de nombreux outils d'analyse de vulnérabilités disponibles et chacun d'entre eux a ses propres points forts et points faibles. Plutôt que d'utiliser un seul outil, plusieurs outils peuvent être utilisés simultanément pour découvrir des vulnérabilités et ce dans le but de réduire la probabilité que des vulnérabilités restent non détectées. Toutefois, pour que la diversité soit efficace, il est intéressant que les outils soient conçus avec différentes approches de conception. De cette façon, une vulnérabilité non détectée par un outil devrait être détectée par un autre. Ce processus garantit une meilleure précision des résultats car la combinaison des résultats des outils permet d'augmenter le nombre de bonnes alertes de sécurité. Cependant, cela pourrait entraîner une augmentation considérable du nombre de fausses alertes. Ce problème a suscité l'intérêt de plusieurs chercheurs qui proposent des heuristiques de plus en plus efficaces pour la combinaison des outils d'analyse de vulnérabilités.

Rutar, et al. [58] mènent une étude de cas en utilisant cinq outils de recherche de bogues pour analyser une variété de programmes Java. Leurs expérimentations montrent

que les outils découvrent des bogues qui ne se chevauchent pas et qu'aucun outil ne peut être considéré comme le meilleur outil. Pour exploiter le nombre d'alertes renvoyées pour chaque individu, les auteurs proposent un méta-outil qui combine les résultats des outils en fonction de la fréquence d'alerte pour un même individu. Les alertes ne sont pas examinées manuellement. Nous ne disposons donc d'aucune information sur la pertinence des résultats renvoyés par les outils.

Meng et al. [59] proposent également une approche permettant de fusionner les résultats de différents outils d'analyse statique. L'utilisateur doit spécifier le programme à analyser et les classes de bogues à détecter. Après avoir déterminé les outils qui peuvent détecter les classes de bogues spécifiées, les auteurs mettent en place les configurations d'outils, les utilisent pour analyser le code puis combinent leurs résultats dans un seul rapport de sécurité. Les auteurs appliquent deux politiques pour classer les résultats afin que les alertes critiques passent avant les rapports inutiles et faux.

Wang et al. [60] proposent une approche basée sur un service web qui encapsule plusieurs outils d'analyse statique. L'utilisateur a la possibilité d'analyser un code source sans télécharger aucun outil d'analyse. Pour ce faire, l'utilisateur rend disponible le code source et des informations supplémentaires telles que le langage de programmation et les classes de bogues à détecter. Après la phase d'analyse du code source, les résultats sont fusionnés et affinés en supprimant les redondances de défauts. Cette approche est validée sur un cas de test Java en terme de durée d'exécution. Aucune mesure d'évaluation des performances n'est présentée dans cette étude.

Nunes et al. [61] se concentrent sur le problème de savoir comment combiner plusieurs outils d'analyse statique pour un objectif donné. Dans ce travail, les auteurs ont considéré quatre niveaux de criticité des scénarios de développement de logiciels. Pour chaque scénario, ils ont utilisé la métrique la plus adéquate pour classer toutes les combinaisons d'outils possibles. Ainsi, la meilleure solution peut être un seul outil ou une combinaison de certains outils. Contrairement aux travaux précédents, les auteurs font une distinction entre les vrais positifs (VP) et les faux positifs (FP). Pour cette étude, les auteurs n'ont pas fait annoter les applications avec leurs vulnérabilités. Par conséquent, pour évaluer une combinaison donnée, les auteurs ont analysé manuellement les résultats renvoyés par les outils inclus dans la combinaison. Suite à cette analyse, tous les VP retournés sont ajoutés à la classe positive, et tous les FP sont correctement ajoutés aux instances négatives. Il

est évident que cette méthode ne permet pas de détecter les FN. Comme le soulignent les auteurs, cette combinaison des résultats augmente considérablement le nombre de fausses alertes. Dans ce travail, la variabilité réside dans le sous-ensemble d'outils sélectionné. L'agrégation des résultats renvoyés par tous les outils reste la même.

Algaith et. al. [62] combinent les résultats de différents outils d'analyse statique pour diminuer le nombre de faux négatifs. L'étude s'est basée sur les résultats de 5 outils différents où plusieurs combinaisons d'outils sont évaluées : 10 combinaisons de 2 outils, 10 combinaisons de 3 outils, 5 combinaisons de 4 outils et enfin une combinaison de 5 outils. Trois approches sont utilisées pour identifier les vulnérabilités SQLi et XSS :(i) 1-out-of-N : déclenche l'alerte lorsqu'un des outils signale l'alerte ; (ii) N-out-of-N : déclenche l'alerte lorsque tous les outils signalent l'alerte ; et (iii) majorité simple : déclenche l'alerte lorsque la majorité simple des outils signale l'alerte. L'étude est menée sur une base de données publiée par Nunes et al. [61] où 5 outils sont utilisés afin de détecter les vulnérabilités SQLi et XSS dans 134 plugins WordPress. Les résultats de l'évaluation sont présentés en utilisant la mesure de sensibilité (mesure la performance de trouver les vulnérabilités) et la mesure de spécificité (mesure la performance de l'outil à ne pas produire de fausses alarmes). Les résultats montrent que l'approche N-out-of-N a une meilleure spécificité que 1-out-of-N ou majorité simple, au prix d'un faible rappel. L'heuristique la plus performante en ce qui concerne les vulnérabilités détectées est 1-out-of-N, ce qui se traduit par un taux de rappel très élevé. D'autre part, le nombre de faux positifs augmente, ce qui réduit la précision. La principale limite de cette approche est l'impossibilité d'explorer différents compromis de performance, car les règles de combinaison suivies sont rigides.

Enfin, Pereira et al. [63] étudient la possibilité d'intégrer les techniques d'apprentissage automatique dans la combinaison des résultats des outils existants. Cette étude s'appuie sur l'hypothèse que les techniques d'apprentissage automatique permettrait d'explorer les différents compromis de performances entre la précision des alertes retournées et le taux de bonnes alertes non détectées. Les auteurs présentent ainsi une étude exploratoire sur l'ensemble de données utilisé dans les deux études précédentes, focalisé sur les vulnérabilités SQLi et XSS. Plusieurs techniques d'apprentissage automatique sont utilisées comprenant les arbres de décision, les réseaux de neurones, les forêts aléatoires et les SVM combinées à diverses méthodes de sélection de caractéristiques (appelée ingénierie des caractéristiques qui sera détaillée dans le chapitre suivant). En comparant les résultats de

cette approche avec ceux de l'heuristique traditionnelle 1-out-of-N, une légère amélioration peut être notée dans la détection des vulnérabilités SQLi, à savoir une réduction des fausses alarmes. En revanche, pour les vulnérabilités XSS, les résultats utilisant l'apprentissage automatique sont égaux à ceux obtenus avec l'heuristique 1-out-of-N. De plus, les résultats montrent qu'il est possible de créer un classement des fichiers de code source en considérant une estimation du nombre potentiel de vulnérabilités dans chaque fichier, tel que rapporté par les multiples outils. En pratique, ces informations peuvent être utilisées par les développeurs pour hiérarchiser leur travail et se concentrer sur les fichiers présentant le plus de vulnérabilités.

3.4 Résumé et discussion

Ce chapitre passe en revue les travaux existants concernant l'identification des vulnérabilités logicielles. La première partie sert à introduire les différentes approches d'analyse utilisées dans le développement logiciel. Ces approches sont également utilisées pour l'identification des vulnérabilités à travers des outils d'analyse statique et dynamique. Nous avons présenté plusieurs travaux qui se sont focalisés sur la création d'outils spécifiques à des classes de vulnérabilités particulières ou plus génériques en englobant toutes les vulnérabilités qui peuvent être couvertes.

L'évaluation des performances des outils diffèrent d'une étude à l'autre. La majorité des études utilise les métriques de précision et rappel pour mesurer les performances des outils et les comparer entre eux. Une des méthodes les plus efficaces est d'analyser un benchmark avec des vulnérabilités connues et d'analyser les résultats produits par l'outil. À partir de là, il est possible de dresser des tables de confusion et calculer des métriques de performances.

Dans la dernière partie de ce chapitre, nous présentons les travaux qui se sont intéressés à la combinaison d'outils existants. Il est intéressant de noter, d'après les résultats de l'évaluation des outils, que les performances des outils varient considérablement et qu'un taux de détection élevé s'accompagne généralement d'un taux important de fausses alertes. Les études présentées ont envisagé de combiner les outils pour améliorer la capacité de détection globale en utilisant des heuristiques différentes : 1-out-of-N, N-out-of-N, un vote majoritaire ou simplement en combinant tous les résultats dans un seul rapport de sécurité.

Ces approches sont prometteuses mais aucune d'entre elles ne prend en considération les performances individuelles des outils d'analyse. Il est connu que les performances des outils dépendent des classes de vulnérabilités à détecter. Les combiner simplement en utilisant une des heuristiques présentées ne permet pas de tirer profit des points forts de ces outils. Par conséquent, dans le chapitre 6, nous proposons une heuristique qui combine les résultats des outils en se basant sur les performances individuelles des outils pour chaque catégorie de vulnérabilités.

Chapitre 4

Modèles de Prédiction de Vulnérabilités

Outre les approches mentionnées dans le chapitre précédent, il existe une autre catégorie de travaux qui utilisent des techniques issues des domaines de la science des données et de l'intelligence artificielle pour aborder le problème de l'analyse et la découverte des vulnérabilités logicielles. Dans ce chapitre, nous commençons par exposer les bases fondamentales des techniques d'apprentissage automatique dans le contexte de la prédiction des vulnérabilités. Par la suite, nous présentons les études qui portent sur les modèles de prédiction de vulnérabilités, en précisant leurs similarités et leurs différences ainsi que les résultats expérimentaux de chaque étude.

Sommaire

4.1 Utilisation de techniques d'apprentissage automatique et d'exploration de données

Les techniques d'apprentissage automatique se sont avérées efficaces dans la pratique pour de nombreux domaines d'applications, notamment dans le domaine de la sécurité informatique et de la protection de la vie privée. De nombreux problèmes de sécurité ont été traitées à l'aide de ces techniques telles que le filtrage du spam [64, 65] et les systèmes de détection des intrusions [66, 67].
Comme défini par Arthur Samuel dans son travail fondateur [68], l'apprentissage automatique est le domaine d'étude qui permet de développer des techniques et des algorithmes de calcul qui permettent aux systèmes informatiques d'acquérir de nouvelles capacités sans être explicitement programmés. L'exploration de données, connu sous le nom de *data mining*, est le processus informatique d'extraction de connaissances à partir de grandes quantités de données. D'après Han et al. [69], ce processus comprend plusieurs étapes : l'extraction et la collecte de données, le nettoyage et l'intégration de données, la sélection et la transformation de données, l'exploration de connaissances, et enfin la visualisation et la communication.

Les algorithmes et les techniques d'apprentissage automatique sont souvent utilisés dans le processus d'exploration des données pour le pré-traitement, la reconnaissance de patrons et la génération de modèles de prédiction.

Les techniques d'apprentissage automatique peuvent être classées en trois grandes catégories :

Apprentissage supervisé : le système d'apprentissage déduit une fonction/modèle souhaité sur la base d'un ensemble d'exemples étiquetés, où chaque exemple est constitué de données d'entrée (généralement un vecteur) et de la valeur de sortie correspondante souhaitée (l'étiquette).

Apprentissage non supervisé : dans les situations où les données de formation étiquetées ne sont pas disponibles, l'objectif du système d'apprentissage est d'identifier des modèles et des structures dans l'ensemble de données donné.

Apprentissage de renforcement : le système d'apprentissage est formé pour atteindre un certain objectif, en recevant des récompenses et des pénalités par l'in-

teraction avec un environnement dynamique.

Dans le domaine de l'analyse et de la découverte des vulnérabilités logicielles, de nombreuses études ont été publiées au cours des années précédentes sur l'utilisation des techniques d'apprentissage automatique et d'exploration des données. Ces travaux ont été classés par Ghaffarian et al. [70] dans trois catégories principales qui sont définies comme suit :

1. **Approches de détection des anomalies :**

   La détection des anomalies fait référence au problème de détection de patrons dans les données qui ne sont pas conformes au comportement normal et attendu, souvent appelé "anomalies" ou "valeurs aberrantes" (Chandola et al. [71]). Ce problème a été largement étudié dans de nombreux domaines de recherche et d'application, notamment dans le domaine de la découverte des défauts et des vulnérabilités logicielles. Dans le domaine de l'intelligence artificielle (IA), cette catégorie de travaux utilise une approche d'apprentissage non supervisée pour extraire automatiquement un modèle de normalité ou bien des règles à partir du code source du logiciel. Ceci permet de détecter les vulnérabilités en tant que comportement déviant de la majorité des règles normales.

2. **Reconnaissance de patrons de code vulnérable :**

   Cette catégorie de travaux utilise une approche d'apprentissage automatique (principalement supervisée) pour extraire des modèles de segments de code vulnérables à partir de nombreux échantillons de vulnérabilités. Ainsi, il devient possible d'utiliser des techniques de correspondance de modèles pour détecter et localiser des vulnérabilités dans le code source des logiciels.

3. **Modèles de prédiction de la vulnérabilité :**

   Un grand nombre d'études proposent une approche d'apprentissage automatique (principalement supervisée) pour construire un modèle de prédiction utilisant les métriques logicielles comme ensemble de caractéristiques. Le modèle produit est utilisé pour prédire le statut d'un code vis-à-vis des vulnérabilités, et ce en fonction des métriques logicielles mesurées.

Dans cette thèse, nous nous intéressons particulièrement à la dernière catégorie de travaux, à savoir les travaux liés aux modèles de prédiction de vulnérabilités. Le processus de création et d'évaluation de tels modèles est décrit dans ce qui suit.

4.2 Processus de construction de modèles de prédiction de vulnérabilités

Les modèles de prédiction de vulnérabilités, en anglais *Vulnerability Prediction Models* (VPM), visent à classer les composants logiciels comme potentiellement vulnérables dans le but ultime de soutenir les tests et l'examen du code. Les VPM se sont inspirés de la modélisation prédictive des défauts (Defect Prediction Modelling : DPM) dont l'objectif est de classer les composants logiciels comme probablement bogués plutôt que vulnérables. Les modèles de prédiction de vulnérabilités, contrairement aux catégories d'approches précédemment introduites, ne visent pas à trouver directement les vulnérabilités. Leur but initial est plutôt d'aider à établir les priorités des tests de sécurité. Cela rend les VPM plus proches des approches de modélisation des prédictions (en particulier celles concernant les DPM) que des approches de découverte de vulnérabilités.

Depuis le travail fondateur de Neuhaus et al. en 2007 [18], la recherche dans le domaine de la modélisation prédictive des vulnérabilités n'a cessé de se diversifier. La différence réside principalement dans les bases de données utilisées, les métriques qui caractérisent le code logiciel ou encore les méthodes d'apprentissage automatique. Cependant, tous les travaux suivent un processus similaire défini dans la Figure 4.1 et dont les étapes sont détaillées ci-dessous :

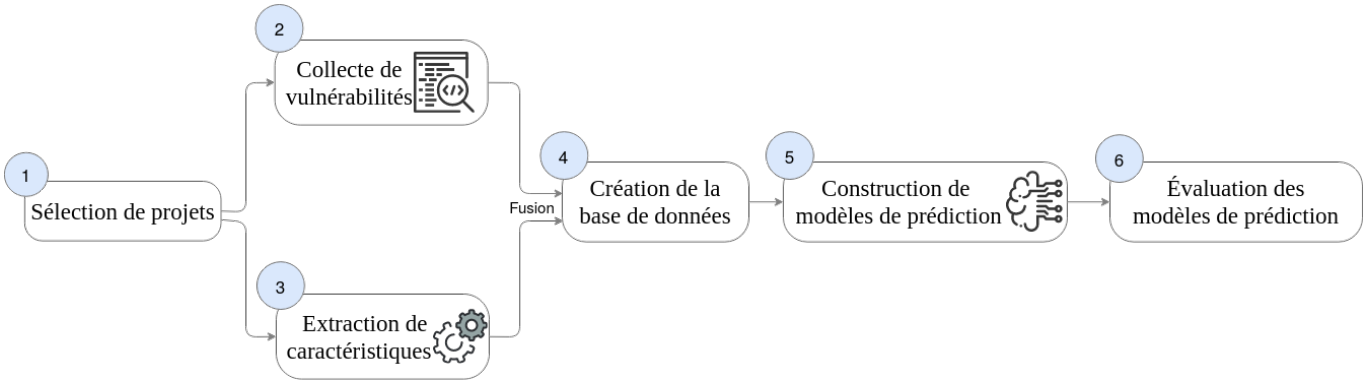

FIGURE 4.1 – Processus de construction des modèles de prédiction de vulnérabilités

1. **Sélection des projets :**
   Le choix des projets est la première étape de toute étude sur les VPM. Les projets ont des propriétés différentes qui sont susceptibles d'affecter le résultat d'une

étude. En outre, la faible quantité de vulnérabilités à analyser peut également avoir un impact sur les performances des modèles. Ainsi, un projet candidat doit avoir un long historique de vulnérabilités bien documenté. Cela explique que des projets tels que Mozilla Firefox et le noyau Linux ont souvent été utilisés pour réaliser des études VPM. En effet, ces derniers sont (i) sensibles à la sécurité, (ii) open source et (iii) ont un long historique de vulnérabilités signalées. Néanmoins, ces choix sont principalement réalisés lorsque les vulnérabilités sont collectées à partir de l'historique de développement.

2. **Collecte de vulnérabilités :**
   La collecte des vulnérabilités est la partie la plus critique dans les études empiriques sur la vulnérabilités. Il s'agit d'identifier le plus grand nombre de vulnérabilités réelles dans ces projets. Dans certains travaux de recherche [18, 72], la méthode utilisée consiste à exploiter les correctifs de vulnérabilités à travers différents *commits* d'un même projet, puis considérer comme vulnérable le code qui précède la correction. D'autres chercheurs utilisent des outils d'analyse statique axés sur la sécurité dans le but de collecter les vulnérabilités [27, 73].

3. **Extraction des caractéristiques**
   Avant d'entamer les expérimentations d'apprentissage, il est important de choisir les caractéristiques à utiliser pour décrire les individus de l'étude. En suggérant une approche, les chercheurs proposent en réalité un ensemble de propriétés, appelées attributs ou variables significatives, à utiliser comme caractéristiques pour les algorithmes d'apprentissage. Ces attributs sont calculés pour chaque individu de la base de données. Cette phase peut être réalisée avant ou après la division de la base en ensemble d'apprentissage et de test. Cependant, il est impératif de choisir la granularité de l'étude avant l'étape d'extraction des caractéristiques.
   Le niveau de granularité est l'entité (partie de code) qui sera présentée à l'algorithme d'apprentissage et considérée comme un individu. Cette dernière est caractérisée par différentes métriques (variables indépendantes) et la ou les vulnérabilités que l'entité contient (variables dépendantes). Cette entité est présentée dans la phase d'apprentissage et de test. Elle est également retournée aux développeurs en cas d'alerte de sécurité détectée par le modèle. Évidemment, différents niveaux de granularité offrent différents avantages [74]. Par exemple, la granularité au niveau

de la ligne de code peut être la plus précise pour localiser une vulnérabilité. Cependant, une ligne de code ne permet pas aux développeurs d'identifier le problème, d'autant plus que le nombre de fausses alertes est important pour ce niveau de granularité. La plupart des études adopte le niveau de granularité fichier à la suite des conclusions de Morrison et al. [74], qui ont constaté que le niveau des fichiers (composants) était suffisant pour les développeurs de Microsoft. Cependant, selon le paradigme de programmation, d'autres granularités peuvent être considérées. Par exemple, en programmation orientée objet, il est possible d'utiliser la classe ou la méthode comme granularité.

4. **Création de la base de données :**
   Dans cette étape, les ensembles des vulnérabilités collectées et des caractéristiques sont fusionnés afin de créer un ensemble d'individus caractérisés et étiquetés. La base doit contenir des individus vulnérables et non vulnérables simultanément afin de permettre à l'algorithme d'apprendre à distinguer les individus des deux catégories.
   Dans un problème de prédiction, le modèle reçoit un ensemble de données avec des labels (vulnérabilités) connus sur lesquels l'apprentissage est effectué (ensemble de données d'apprentissage), et un ensemble de données inconnues sur lesquelles le modèle est testé (appelé ensemble de données de validation ou ensemble de test). Le but de la validation est de tester la capacité du modèle à prédire de nouvelles données qui n'ont pas été utilisées dans la phase d'apprentissage, afin de signaler des problèmes comme le sur-apprentissage ou le biais de sélection et de donner un aperçu des performances du modèle sur des données inconnues.

   Il existe plusieurs méthodes pour diviser la base de données en un ensemble d'apprentissage et un ensemble de test. La méthode la plus basique est Train/Test Split qui divise les données de façon aléatoire en respectant un pourcentage donné (généralement mis à 66%, 70% ou 75%). Comme le montre la Figure 4.2, si le pourcentage de découpage est mis à 75%, ceci revient à sélectionner 75% d'individus de la base initiale pour construire l'ensemble d'apprentissage et le reste d'individus constitue l'ensemble de test. Le pourcentage de chaque étiquette est maintenu dans les deux ensembles au même niveau que dans l'ensemble de données initial. Le fractionnement peut être répété (généralement 50 à 100 fois) pour assurer la généralisation des résultats.

Une autre méthodologie courante est la validation croisée à k-blocs, qui peut être considérée comme une application spécifique de l'échantillonnage aléatoire. Cette méthode consiste à diviser de façon aléatoire l'ensemble de données en k sous-ensembles avec la même proportion entre les composantes vulnérables et non vulnérables que l'ensemble de données (stratifié). Chaque sous-ensemble est ensuite utilisé comme ensemble de test tandis que les autres sont utilisés pour l'apprentissage. Cela conduit à la création et à l'évaluation de k modèles, où généralement k est égal à 5 ou 10. Pour un découpage ayant 5 blocs, la base est divisée comme dans la Figure 4.3. Ce type d'expérience peut également être répété plusieurs fois (généralement 10) pour valider les résultats.

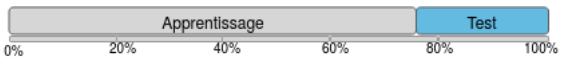

FIGURE 4.2 – Découpage de la base de données avec la méthode Train-Test Split avec un pourcentage de 75%

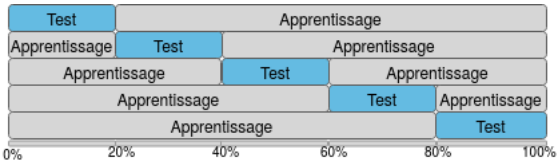

FIGURE 4.3 – Découpage de la base de données avec la méthode de validation croisée avec 5 blocs

5. **Construction de modèles de prédiction :**
   Une fois que les données sont collectées, préparées et caractérisées, l'étape suivante est de construire les modèles de prédiction. Cette opération est effectuée en utilisant différents algorithmes d'apprentissage tels que les arbres de décision, les forêts aléatoires, les réseaux de neurones, les machines à vecteurs de support (SVM), etc. Le type de classificateur le plus courant dans les études de VPM est de loin l'arbre de décision. Les arbres de décision ont l'avantage d'être lisibles par l'homme et assez faciles à interpréter.

6. **Évaluation des modèles de prédiction :**
   Dans cette phase, les modèles de prédiction construits en utilisant l'ensemble d'apprentissage sont évalués sur l'ensemble de test. Traditionnellement, dans les études sur les VPM, une classe est soit vulnérable, soit non vulnérable, ce qui en fait un problème de classification binaire. Cependant, les classes peuvent prendre d'autres formes telles que le nombre de vulnérabilités dans un individu (0 pour un individu

non vulnérable), ou encore la ou les catégories de vulnérabilités de cet individu. Dans ce dernier cas, il s'agit d'une classification multi-classes si la catégorie en sortie est unique parmi plusieurs catégories possibles, ou multi-labels si plusieurs vulnérabilités sont retournées pour un même individu.
Lors de la phase d'évaluation, les étiquettes des individus sont connues, ce qui permet d'évaluer les performances des modèles de prédiction. Un modèle peut soit prédire qu'un composant est vulnérable alors qu'il ne l'est pas (Faux Positif - FP), non vulnérable alors qu'il est vulnérable (Faux Négatif - FN), vulnérable et il l'est réellement (Vrais Positif - VP) et non vulnérable alors qu'il l'est (Vrais Négatif - VN). Une fois toutes les prévisions évaluées, la performance globale du modèle est généralement indiquée dans une matrice de confusion (Tableau 4.4). Cette matrice est utilisée pour mesurer les performances des modèles telles que la précision et le rappel.

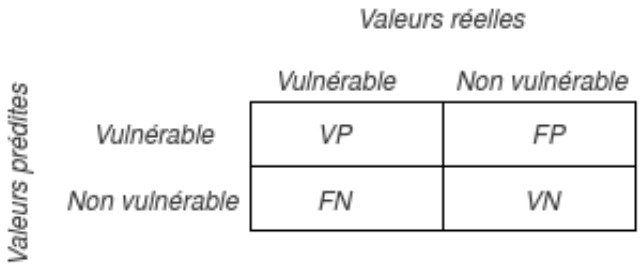

FIGURE 4.4 – Matrice de confusion des modèles de prédiction

Les étapes de ce processus sont davantage détaillées dans le chapitre 8 où nous présentons notre approche de construction et d'évaluation des modèles de prédiction du code vulnérable. Ce processus a été respecté par la communauté des chercheurs en sécurité logicielle ce qui a permis d'avoir une multitude d'études sur les VPM. Cependant, il est à noter que la plupart des études se distinguent les unes des autres soit par les caractéristiques utilisées pour construire les modèles, soit par le niveau de granularité utilisé pour représenter un individu dans la base d'apprentissage, ou encore par les méthodes d'évaluation des modèles.

4.3 Travaux sur les modèles de prédiction de la vulnérabilité

Dans cette section, les travaux sont présentés selon les métriques utilisées pour décrire le code logiciel. Nous commençons par une première partie qui comprend les études sur l'analyse de dépendances. Ensuite, nous présentons les études utilisant des métriques de code. Cette partie est suivie par une section sur les travaux utilisant des techniques d'analyse de texte. Nous présentons également les travaux sur les métriques liées au facteur humain qui ont également été étudiées dans le cadre de la prédiction de vulnérabilités. Enfin, nous présentons les études qui combinent différentes approches avant de détailler quelques études qui prédisent d'autres informations sur la vulnérabilité.

4.3.1 Analyse de dépendances

Neuhaus et al. [18] ont présenté l'une des premières études empiriques sur les modèles de prédiction de vulnérabilités. Cette preuve empirique montre que des caractéristiques telles que les appels d'importation et les appels de fonction sont en corrélation avec les vulnérabilités. Les auteurs ont utilisé ces mesures pour concevoir un nouvel outil capable de prédire les composantes vulnérables. Cette approche est basée sur l'hypothèse que les fichiers vulnérables sont susceptibles de partager des ensembles similaires d'importations et d'appels de fonction qui pourraient être utilisés pour les identifier. Pour valider cette hypothèse, les auteurs commencent leur étude avec une analyse de corrélation entre les appels de fonction/importation et l'existence de vulnérabilités. Les auteurs utilisent un ensemble de vulnérabilités collectées à partir du projet Mozilla Firefox et construisent leur modèle avec les machines à vecteurs de support (SVM). Les résultats de cette étude montrent que cette approche a atteint un rappel de 45% et une précision de 70%. Ces résultats ont encouragé les auteurs à faire davantage d'expérimentations sur le sujet.

Neuhaus et al. [75] étendent leur recherche à l'analyse des dépendances entre les paquets de la distribution Linux RedHat. En effet, la plupart des paquets de RedHat nécessitent l'installation d'autres paquets avant de commencer leur installation. Ce processus peut conduire à de longues chaînes de dépendances qui peuvent être conflictuelles et propager des vulnérabilités. Grâce à une analyse conceptuelle formelle et à des tests d'hypothèses statistiques, les auteurs identifient les dépendances qui augmentent les chances

qu'un paquet soit vulnérable et celles qui les réduisent. Ensuite, ils utilisent un ensemble de données de plus de 3000 paquets pour construire et évaluer des VPM en utilisant les SVM comme algorithme d'apprentissage. Les auteurs montrent que leur étude atteint des performances de 83% de précision et de 65% de rappel. Cette étude sur l'analyse de dépendances entre les paquets permet, d'une part, de sensibiliser les développeurs au risque que présente l'utilisation de paquets et, d'autre part, de les aider à choisir judicieusement leurs nouvelles dépendances. Une autre observation faite par les auteurs est que l'hypothèse selon laquelle les paquets vulnérables auront tendance à développer encore plus de vulnérabilité ne s'applique pas aux paquets de RedHat : le nombre de paquets vulnérables nécessitant deux corrections ou moins (584) est plus élevé que le nombre de paquets nécessitant plus de deux corrections (549).

Nguyen et al. [19] introduisent une nouvelle approche de construction de modèles de prédiction en utilisant des graphes de dépendances. Ces graphes sont basés sur la relation entre des éléments logiciels de différentes grannularité telles que les composants, les classes, les fonctions ou les variables. À partir des graphes, les auteurs utilisent 22 métriques qui les caractérisent et qui sont utilisées pour construire des modèles de prédiction. Pour valider leur approche, les auteurs utilisent deux versions du moteur JavaScript de Firefox avec 5 algorithmes d'apprentissage automatique. Les résultats montrent qu'en terme de rappel, les réseaux bayésiens sont les plus performants avec des valeurs de score atteignant les 75%, tandis que les SVM et les réseaux de neurones dépassent tous les autres algorithmes en terme de précision en atteignant des valeurs de score allant jusqu'à 85%.

4.3.2 Métriques logicielles

L'étude fondamentale de Shin et al. [76] est le point de départ de toutes les études qui utilisent des mesures de complexité pour construire des VPM. Leur hypothèse est que plus un code est complexe, plus il susceptible de contenir des vulnérabilités. Afin de valider cette hypothèse, les auteurs tentent de déterminer si un code complexe est en corrélation avec les vulnérabilités et s'il existe des différences significatives de complexité entre le code vulnérable et le code défectueux. Pour vérifier cela, les auteurs ont calculé un ensemble de 9 mesures de complexité pour chaque fonction présente dans 4 versions de Mozilla Firefox. Parmi ces mesures, nous pouvons citer la complexité cyclomatique de MacCabe, les imbrications et les lignes de code. Dans l'ensemble, ils ont constaté une faible corrélation entre le code vulnérable et la complexité et que le code vulnérable a

tendance à être plus complexe que le code défectueux.

Les résultats de l'étude précédente ont été ensuite utilisés par les mêmes auteurs pour construire des VPM [20]. L'idée est directement inspirée des travaux de Nagappan et al. [77] sur la prédiction des défauts. Dans ce travail, les auteurs explorent 3 scénarios différents : (i) prédire les fonctions défectueuses à partir de toutes les fonctions, ce qui est un cas typique des DPM (modèles de prédiction des défauts) ; (ii) prédire les fonctions vulnérables à partir de toutes les fonctions ; et (iii) prédire les fonctions vulnérables à partir des fonctions défectueuses. Pour chaque scénario, ils construisent un modèle utilisant les 9 métriques et la régression logistique binaire comme algorithme d'apprentissage. L'évaluation est effectuée sur un ensemble de données composé de 6 versions du moteur JavaScript de Mozilla Firefox. Les auteurs ont constaté que dans les deux premiers scénarios, un faible taux de faux positifs (haute précision) était réalisable mais au détriment d'un taux élevé de faux négatifs (faible rappel), ce qui peut être problématique pour la fiabilité du modèle. En ce qui concerne le troisième scénario, les résultats se sont avérés peu concluants, car aucune tendance commune entre les différentes expériences n'a pu être observée.

Shin et al. [78] étendent leurs travaux à l'utilisation de mesures de la complexité d'exécution, par opposition aux précédentes complexités qui étaient statiques. L'intuition est que les métriques collectées pendant l'exécution d'un logiciel peuvent être efficaces pour identifier les emplacements du code vulnérable. L'idée est empruntée des travaux de Khoshgoftaar et al. [79] appliqués aux DPM. Au total, 23 métriques sont étudiées. Pour tenir compte des informations redondantes dans les nombreuses mesures, les auteurs ont procédé à une réduction de l'espace des caractéristiques en fonction du classement des gains d'information. Un autre problème est le déséquilibre important entre la classe majoritaire (fichiers neutres) et la classe minoritaire (fichiers vulnérables), que les auteurs ont résolu par un sous-échantillonnage aléatoire de la classe majoritaire. Les auteurs évaluent d'abord le pouvoir discriminant des métriques sur Firefox et Wireshark. Les résultats suggèrent que presque toutes les mesures montrent un pouvoir discriminant significatif sur l'ensemble de données de Firefox mais pas sur Wireshark. En ce qui concerne les métriques d'exécution, 2 d'entre elles montrent un pouvoir discriminant sur Firefox mais aucune sur Wireshark. Ensuite, les auteurs construisent 4 modèles VPMs en utilisant la régression logistique, un par type de métrique et un autre en les combinant et découvrent

que les modèles utilisant des métriques de dépendance sont les plus performants en terme de précision tandis que ceux utilisant des métriques d'exécution sont intéressants en terme de rappel.

Dans [80], Shin et al. réalisent une étude encore plus approfondie afin de déterminer si les mesures de complexité, de niveau d'activité de code (code churn) et d'activité du développeur, peuvent être utilisées pour la prédiction de vulnérabilités. Si l'utilisation des mesures de complexité est une extension logique des travaux précédents, l'utilisation des mesures de code churn et d'activité du développeur peut être considérée comme nouvelle. Comme pour les travaux précédents, l'idée d'utiliser ces mesures remonte aux travaux de Nagappan et al. [79] et Meneely et al. [81] dans le contexte des DPM. Au total, 28 métriques logicielles ont été analysées dans cette étude, dont 14 métriques de complexité, 3 métriques d'activité de code et 11 paramètres d'activité du développeur. Les auteurs ont évalué le pouvoir discriminant et prédictif de ces métriques par rapport à deux projets open source : Mozilla Firefox et RedHat Linux. Parmi les 28 métriques, 24 ont été jugées discriminantes sur les deux projets. En ce qui concerne l'évaluation du pouvoir prédictif des mesures, les auteurs considèrent à la fois un modèle univarié et multivarié construit à l'aide de plusieurs classifieurs, mais ne présentent que le résultat de la régression linéaire car les résultats des autres classifieurs sont similaires. En ce qui concerne le modèle univarié, c'est-à-dire le modèle utilisant une seule mesure, seul le nombre de modifications d'un fichier et son nombre de développeurs ont fourni un résultat acceptable par le seuil choisi par les auteurs (plus de 70% de rappel et moins de 25% de fausses alertes) pour les deux projets. Quant aux modèles multivariés, 4 types ont été considérés, un par type de métrique et un combinant tous les types. Le modèle utilisant les métriques d'activité de code a donné les meilleurs résultats sur les données de Firefox, tandis que celui utilisant l'activité des développeurs a donné les meilleurs résultats sur RedHat. Enfin, le modèle utilisant toutes les métriques étant le plus stable entre les deux projets. Les auteurs concluent que les modèles basés sur l'historique du développement sont plus solides que ceux basés sur la complexité.

Dans leur dernière étude, Shin et al. [82] étudient la possibilité d'utiliser des modèles de prédiction des défauts (DPM) basés sur des mesures de complexité du code et de l'historique des défauts pour la prédiction des vulnérabilités. Les auteurs envisagent trois scénarios : (i) un modèle est construit sur des données défectueuses et évalue si un fichier

est défectueux ; (ii) un modèle est construit sur des données défectueuses et évalue si un fichier est vulnérable ; et (iii) un modèle est construit sur des données vulnérables et évalue si un fichier est vulnérable. Les auteurs ont réalisé une étude empirique sur 2 versions de Mozilla Firefox, en utilisant 18 mesures de complexité, 5 mesures d'activité de code et une mesure de l'historique des fautes. Plusieurs techniques de classification ont été testées pour prédire les fichiers défectueux et vulnérables ; les auteurs déclarent que les résultats de toutes les techniques sont similaires. Alors que les fichiers de code source défectueux étaient sept fois plus nombreux que les fichiers de code source vulnérables, le modèle de prédiction des fautes et le modèle de prédiction de la vulnérabilité ont donné des résultats similaires en matière de prédiction de la vulnérabilité, avec un rappel d'environ 83% et une précision d'environ 11%. Sur la base de ces résultats, les auteurs concluent que les modèles de prédiction des fautes basés sur des mesures traditionnelles peuvent également être utilisés pour la prédiction de la vulnérabilité. Cependant, des recherches futures sont nécessaires pour améliorer la précision (réduire les faux positifs) tout en conservant un taux de rappel élevé.

Dans l'article [21], Zimmermann et al. étudient la possibilité de prédire l'existence de vulnérabilités dans les modules binaires d'un produit commercial propriétaire (Microsoft Windows Vista). Les auteurs se basent sur des métriques classiques qui ont été utilisées dans des recherches antérieures pour la prédiction des défauts. Ces métriques regroupent le niveau d'activité du code (code churn), la complexité, la couverture, les dépendances ainsi que des mesures organisationnelles. L'étude commence par une analyse des corrélations entre ces métriques et le nombre de vulnérabilités avant d'entamer la phase de construction de modèles de prédiction en utilisant la régression logistique. L'évaluation des modèles a montré que chaque ensemble de métriques avait des forces et des faiblesses mais qu'aucun ensemble universel n'était efficace pour la précision et le rappel en même temps.

En outre, Chowdhury et al. [83] explorent la relation entre les vulnérabilités et les mesures de CCC, à savoir la complexité, le couplage et la cohésion, en particulier pour les programmes orientés objet. Pour chaque catégorie de mesures, les auteurs suggèrent des mesures qui peuvent être calculées au niveau du code ou de la conception. Toutes les mesures au niveau du code sont similaires à celles de Shin et al. [80], tandis que les mesures au niveau de la conception sont nouvelles dans ce domaine. Dans une première étude [83],

les auteurs cherchent à savoir si ces mesures sont en corrélation avec la vulnérabilité en utilisant Firefox comme cas d'étude. Ils trouvent que la complexité et le couplage sont en effet fortement corrélées avec la vulnérabilité tandis que la cohésion a une corrélation moyenne.

Dans le prolongement de cette étude [84], Chowdhury et al. utilisent les mêmes mesures pour construire des modèles issus de 4 classificateurs différents en utilisant 52 versions de Firefox. Ils ont découvert que l'algorithme basé sur un arbre de décision surpasse les autres algorithmes et pourrait atteindre 75% de rappel et 28% de taux de faux positifs.

Morrison et al. [74] affirment que si les modèles de prédiction des défauts sont adoptés par les équipes de Microsoft, ce n'est pas le cas des modèles de prédiction des vulnérabilités. Pour expliquer cette divergence, les auteurs tentent de reproduire un VPM proposé par Zimmermann et al. [21] pour les versions 7 et 8 du système d'exploitation Microsoft Windows à deux niveaux de granularité : binaire et fichier. Selon les auteurs, les résultats au niveau binaire ont tendance à être bons, mais ils sont moins exploitables, tandis que les résultats au niveau du fichier sont insuffisants mais exploitables. Les auteurs évaluent différents modèles basés sur 29 métriques de 6 catégories différentes : l'activité de code, la complexité, la dépendance, l'héritage, la taille et l'historique de la vulnérabilité et un large ensemble de classifieurs. Après avoir analysé les résultats, les auteurs concluent que dans l'état actuel, les résultats ne sont pas assez bons pour être réellement utiles. En effet, au mieux, une précision de 76% et un rappel de 42% sont atteints au niveau binaire, et respectivement 47% et 14% au niveau du fichier.

Moshtari et al. [85] reproduisent l'étude de Shin et al. [80], en utilisant les métriques de complexité et de couplage. Les auteurs proposent un cadre d'analyse semi-automatique pour détecter les vulnérabilités logicielles et utilisent les résultats retournés comme informations supplémentaires sur la vulnérabilités, au lieu de les signaler uniquement. Selon les auteurs, cette méthode fournit des informations plus complètes sur les vulnérabilités d'un logiciel. Contrairement aux études précédentes qui n'étudiaient que la prédiction de la vulnérabilité à l'intérieur d'un projet, cette étude a examiné la prédiction de la vulnérabilité à la fois à l'intérieur d'un projet et entre projets, en se basant sur des données recueillies dans cinq projets open-source. Différentes techniques de classification ont été utilisées pour les expériences. Un ensemble de 11 métriques de complexité unitaire et 4 métriques de couplage ont été mesurées au niveau de la granularité des fichiers. Les résultats rapportés

pour la prédiction intra-projet effectuée sur Mozilla Firefox sont impressionnants pour les différentes techniques de classification (rappel supérieur à 90% et taux de faux positifs inférieur à 10%). Les auteurs affirment que les informations rajoutées sur la vulnérabilité contribuent à cette amélioration, et justifient leur affirmation en comparant l'approche proposée avec le travail de Shin et al. [80]. En ce qui concerne l'évaluation croisée entre les projets, les résultats sont moins prometteurs avec une précision de 20% et un rappel de 31%.

Moshtari et al [86] étendent leur étude à l'analyse du pouvoir de la complexité et des métriques de couplage sur la prédiction de projets croisés avec des langages de programmation différents. Les auteurs utilisent les mêmes caractéristiques que Shin et al. [76] en y ajoutant également des métriques de couplages spécifiques qui prennent en considération la communication externe à l'application. Cette dernière catégorie de métriques correspond au nombre d'en-têtes liées aux appels sensibles comme les E/S, le réseau, les bases de données, la mémoire et les systèmes et est en fait inspirée des travaux de Neuhaus et al. [18]. Pour évaluer les modèles construits à partir de ces caractéristiques pour la prédiction croisée, les auteurs construisent un ensemble de données basé sur 5 logiciels libres (Apache Tomcat, Eclipse, OpnScada, Mozilla Firefox et le noyau Linux) dont les vulnérabilités sont identifiées avec avec un outil d'analyse statique. Les auteurs ont constaté que la combinaison des mesures de complexité et de de couplage améliorait considérablement les résultats, le taux de rappel atteignant jusqu'à 87%.

Alves et al. [87] reproduisent 18 expérimentations issues d'études utilisant des mesures logicielles sur un ensemble de données de 2 875 correctifs de vulnérabilité provenant de 5 logiciels : Firefox, noyau Linux, httpd, glibc, XenHV [25]. Dans l'ensemble, les auteurs estiment que la mesure de la précision n'est pas pertinente et que d'autres mesures de performances sont plus intéressantes. Ils observent également que les modèles utilisant les forêts aléatoires sont plus performants dans tous les cas.

Enfin, dans [88], Du et al. proposent et mettent en place un framework générique pour identifier les fonctions potentiellement vulnérables. Ce framework, appelé LEOPARD, s'appuie sur les métriques de code pour faire la prédiction comprenant des métriques de complexité et des métriques de vulnérabilités. LEOPARD comporte deux étapes de traitement : (i) une première étape où les métriques de complexité sont utilisées pour répartir

les fonctions d'une application cible dans un ensemble de catégories, et (ii) une deuxième étape où les métriques de vulnérabilités sont exploitées pour identifier les fonctions les plus importantes, dans chaque catégorie, en terme de sécurité. Les auteurs justifient leur approche de classement par la relation proportionnelle entre les métriques de complexité et celles liées à la vulnérabilité. Suite à la première catégorisation faite dans cette étude, chaque catégorie a un niveau de complexité différent. Ceci permet à la deuxième catégorisation d'identifier les vulnérabilités à tous les niveaux de complexité, sans manquer celles de très faible complexité. Les auteurs ont évalué leur approche sur un ensemble de 11 projets C/C++. Les résultats expérimentaux montrent que LEOPARD peut couvrir 74% des fonctions vulnérables en identifiant 20% des fonctions comme potentiellement vulnérables. En appliquant LEOPARD sur PHP, MJS, XED, FFmpeg et Radare2 et en procédant à un audit manuel supplémentaire ou à un fuzzing automatique, les auteurs ont découvert 22 nouveaux bogues, dont huit sont de nouvelles vulnérabilités.

4.3.3 Analyse de texte

Les chercheurs se sont également intéressés à appliquer les techniques d'analyse de texte dans le domaine de la sécurité informatique et plus particulièrement sur le code source des logiciels. L'hypothèse de base, commune à toutes les études de cette section, est que le code source peut être considéré comme un texte ordinaire sur lequel les techniques d'analyse de texte peuvent être pertinentes. Cette idée est inspirée des travaux de Hata et al. [89] sur la prédiction des défauts.

Hovsepyan et al. [90] présentent une étude préliminaire sur la détection des composants logiciels vulnérables à l'aide de techniques d'analyse de texte. Comme précédemment expliqué, l'idée est de considérer le code source logiciel comme du texte et d'y appliquer des techniques d'analyse de texte au lieu d'utiliser d'autres caractéristiques telles que les métriques de code. Ils suggèrent donc de considérer le code comme n'importe quel texte anglais ordinaire qui peut être utilisé comme entrée aux algorithmes d'apprentissage. Les auteurs ont évalué leur approche sur un ensemble de données composé de 19 versions de l'application Android K9 mail. Les vulnérabilités de la première version sont obtenues en utilisant un outil d'analyse de vulnérabilités et sont utilisées pour former un modèle de prédiction. Ce dernier est ensuite appliqué sur toutes les versions suivantes pour prédire leurs vulnérabilités. Les résultats obtenus ont été considérés comme encourageants avec une moyenne de rappel de 88% et une précision de 85%.

Une version plus élaborée de ce travail est présentée par Scandariato et al. [27]. Tout comme l'étude précédente, celle-ci est principalement basée sur la technique du *sac de mots* (bag-of-words en anglais), où une représentation vectorielle de chaque composant logiciel (c'est-à-dire le fichier de code source) est construite en fonction de la fréquence des différents jetons textuels (tokens) dans le code source. Cinq techniques de classification sont utilisées sur un ensemble de données de 20 applications open source étiquetées avec un outil d'analyse de vulnérabilité (Fortify [12]). Le choix de s'appuyer sur ce type de vérité de terrain plutôt que sur des rapports de vulnérabilité est justifié par les auteurs sur la base des résultats obtenus par Walden et al. [91], qui montrent qu'il existe une forte corrélation entre les alertes provenant de l'outil d'analyse statique de sécurité et les vulnérabilités. Dans cette étude, les auteurs évaluent leurs modèles en utilisant 3 moyens différents : (i) prédiction au sein du même projet en analysant la première version uniquement (ii) prédiction sur des versions ultérieures et (iii) prédiction croisée entre tous les projets. Ils ont trouvé des résultats intéressants, atteignant en moyenne un rappel de 77% et une précision de 90%. Selon les résultats rapportés, les modèles de classification les plus performants sont obtenus avec les forêts aléatoires. Les résultats finaux sont que la prédiction de la vulnérabilité au sein du projet et des versions ultérieures sont acceptables, alors que les résultats de la prédiction de la vulnérabilité entre projets ne le sont pas.

Pang et al. [92] étudient la possibilité de prédire les composants logiciels vulnérables en utilisant les N-grammes. Dans les études antérieures, telles que Scandariato et al. [27], les auteurs ont testé la technique du sac de mots qui est la forme la plus simple d'analyse N-gram. Une préoccupation majeure lors de l'analyse N-gram est de traiter la haute dimensionnalité, qui pourrait considérablement entraver la performance des modèles de classification. Pour surmonter ce problème, les auteurs utilisent une approche hybride, combinant l'analyse N-gram et la sélection de caractéristiques statistiques pour prédire les composants logiciels vulnérables. Pour évaluer cette idée, ils utilisent les données de 4 des applications Android de l'étude de Scandariato et al. [27] et les SVM comme algorithme d'apprentissage. Les auteurs extraient toutes les caractéristiques N-gram pour ($1 \leq N \leq 5$) de tous les fichiers sources Java. En moyenne, le modèle a atteint près de 96% de précision et 87% de rappel pour la configuration au sein du projet. Les résultats des expériences croisées ont été en moyenne de 67% pour la précision et de 63% pour le rappel. Les mêmes auteurs étendent leur étude, dans [93], à l'utilisation d'un réseau de neurones

profond pour remplacer les SVM et ont trouvé un résultat légèrement meilleur avec une précision et un rappel proches de 95%. Cependant, dans les deux documents, les auteurs soulignent la nécessité de réaliser davantage d'expériences pour valider et généraliser ces résultats.

Suite à leurs travaux sur l'exploration de textes [27], Walden et al. [94] comparent le résultat obtenu par des modèles construits à partir de sacs de mots avec celui obtenu par des mesures de code de l'étude Shin et al. [80]. Pour effectuer cette comparaison, ils construisent un ensemble de données de 3 applications PHP contenant 223 vulnérabilités et ont procédé à une validation croisée des résultats. Au final, les auteurs ont constaté que les modèles d'exploration de texte étaient toujours plus performants que leurs homologues de métriques de code.

4.3.4 Métriques liées au facteur humain

La sécurité informatique est largement impactée par le facteur humain et les vulnérabilités logicielles ne font pas exception. Il est évident que le développement logiciel est sujet à de nombreuses erreurs causées par les développeurs. Une simple erreur syntaxique peut engendrer un plantage du programme, ce qui est considéré comme une vulnérabilité vu que la propriété de disponibilité est impactée. Bien que les développeurs ne soient pas tous formés en sécurité, leur expérience peut leur permettre d'éviter certaines vulnérabilités. Ceci est l'hypothèse de base de la communauté des chercheurs qui a essayé d'étudier l'éventuelle relation entre le développement logiciel et l'existence de vulnérabilités. Une première étude que nous pouvons citer est celle de Shin et al. [80] où les auteurs ont introduit les métriques liées à l'activité de développement avec les métriques de code.

Meneely et al. [95], ont aussi étudié la relation entre les mesures d'activité des développeurs et les vulnérabilités logicielles. Les mesures d'activité mesurées comprennent : le nombre de développeurs distincts qui ont modifié un fichier source, le nombre de commits effectués pour un fichier et le nombre de chemins géodésiques [1] contenant le fichier dans le réseau de contribution. Les auteurs ont réalisé l'étude sur trois projets où l'ensemble des données recueillies comprenait une étiquette indiquant si un fichier de code source

1. Un chemin géodésique est le chemin le plus court entre deux noeuds. Ce chemin n'est pas nécessairement unique. Dans le cas où les arcs sont pondérés, le chemin géodésique correspond au chemin ayant le plus petit poids.

avait été corrigé ou non, ainsi que les mesures de l'activité des développeurs à partir des journaux de contrôle de version. En utilisant une analyse de corrélation statistique, les auteurs indiquent qu'une corrélation statistiquement significative a été trouvée pour chaque mesure avec le nombre de vulnérabilités ; mais les corrélations varient et ne sont pas très fortes. Les auteurs ont utilisé le réseau bayésien comme modèle prédictif et ont obtenu des résultats qui montrent que l'activité des développeurs peut être utilisée pour prédire les fichiers vulnérables. Néanmoins, les valeurs de précision et de rappel sont décevantes avec une précision entre 12% et 29%, et un rappel entre 32% et 56%).

Bosu et al. [96] réalisent une étude empirique similaire dont le but est de vérifier si les révisions de code menées par des développeurs peuvent identifier les vulnérabilités. Pour ce faire, les auteurs analysent plus de 260.000 révisions de code provenant de 10 grands projets open-source. Ils ont pu identifier plus de 400 changements de code vulnérables. Ces derniers correspondent aux modifications de code qui contiennent une vulnérabilité exploitable. Leur objectif était d'identifier les caractéristiques des modifications de code vulnérable, et d'identifier les caractéristiques des développeurs susceptibles d'introduire des vulnérabilités. Les principales conclusions de cette étude sont que (i) les modifications apportées par des développeurs moins expérimentés étaient sensiblement plus susceptibles d'introduire des vulnérabilités ; (ii) la probabilité de vulnérabilité augmente avec la taille de la modification (plus de lignes modifiées) ; et (iii) les nouveaux fichiers sont moins susceptibles de contenir des vulnérabilités que les fichiers modifiés.

4.3.5 Combinaison d'approches

Perl et al. [97] étudient l'effet de l'utilisation des méta-données contenues dans les dépôts de code source en plus des métriques de code pour identifier les commits contribuant à la vulnérabilité. Les auteurs affirment que les logiciels se développent progressivement et que la plupart des projets open-source utilisent des systèmes de contrôle de versions tels que GitHub, ce qui fait que les commits sont des unités naturelles pour vérifier les vulnérabilités. Pour cette raison, les auteurs rassemblent un ensemble de données de plus de 170.000 commits provenant de 66 projets open-source dont 640 commits contribuant à la vulnérabilité. Les auteurs sélectionnent un ensemble de métriques de l'activité du développeur et de l'activité du code (code churn), ainsi que des méta-données GitHub (projet, auteur, commit et fichier) et extraient ces caractéristiques pour l'ensemble de données rassemblées. Sur la base de cet ensemble de données, les auteurs évaluent le système qu'ils

proposent, appelé VCCFinder, qui utilise un classifieur SVM pour identifier les commits qui introduisent la vulnérabilité à partir des commit neutres. Les auteurs comparent les résultats du système qu'ils proposent avec les résultats de l'outil d'analyse statique FlawFinder. Au même niveau de rappel, FlawFinder atteint une précision de seulement 1%, tandis que VCCFinder atteint une précision de 60%, produisant beaucoup moins de faux positifs. L'ensemble de données susmentionné est rendu publique par les auteurs à titre de contribution à la communauté des chercheurs.

Zhang et al. [98] proposent, d'autre part, de combiner les métriques de code et les techniques d'exploration de texte. Pour ce faire, les auteurs présentent une approche combinant le résultat de 6 classificateurs différents, trois s'entraînent sur les métriques de code, et trois sur les métriques de texte. Pour combiner les différentes prédictions en une seule, les auteurs suggèrent d'utiliser un autre classificateur en plus qui utilise le score de confiance de chaque classificateur. Les auteurs évaluent cette approche sur le même ensemble de données que Walden et al. [94], composé de trois applications PHP. Dans l'ensemble, les auteurs ont réussi à améliorer la précision mais n'ont pas amélioré le rappel.

4.3.6 Autre modèles de prédiction

Toutes les études que nous avons présentées dans cette section portent sur la prédiction du code vulnérable. Cependant, les modèles de prédiction peuvent être utilisés pour prédire d'autres caractéristiques du code vulnérable. Par exemple, Alhazmi et al. [22] présentent une approche permettant d'identifier le nombre de vulnérabilités d'un système. Pour ce faire, les auteurs s'appuient sur deux mesures pour construire un modèle : (i) la densité de la vulnérabilité et (ii) le taux de découverte des vulnérabilités. Ils effectuent une évaluation sur 5 versions de Windows et 2 de Red Hat Linux et observent que la plupart des vulnérabilités découvertes récemment dans la vie d'un système sont liées à la sortie d'une nouvelle version.

Pokhrel et al. [99] suggèrent de s'appuyer sur l'utilisation d'un modèle prédictif de séries chronologiques pour prédire le nombre de vulnérabilités présent dans un système. Ils ont trouvé un résultat intéressant avec un modèle non linéaire. Zhang et al. [100] tentent de déterminer si les métadonnées du NVD peuvent être utilisés pour prédire la date la prochaine découverte de vulnérabilité, mais ils obtiennent un résultat négatif. En utili-

sant la même source d'information, Bozorgi et al. [23] essaient de créer des modèles pour déterminer si une vulnérabilité est effectivement exploitable en se basant sur les données CVE.

Younis et al. [101] tentent d'identifier les caractéristiques du code qui contient des vulnérabilités susceptibles d'être exploitables. À cette fin, les auteurs rassemblent 183 vulnérabilités provenant du noyau Linux et des projets du serveur web Apache HTTPD, incluant 82 vulnérabilités exploitables. Les auteurs sélectionnent huit mesures logicielles de quatre catégories différentes pour caractériser ces vulnérabilités : une métrique de taille, 3 métriques décrivant la structure du code, une métrique correspondant à la facilité d'accès au code et enfin 3 métriques liées à la communication entre entités. Les résultats du pouvoir discriminant des métriques sont mitigés ; certaines métriques ont un pouvoir discriminant statistiquement significatif alors que d'autres n'en ont pas. Les auteurs cherchent également à savoir s'il existe une combinaison de mesures pouvant être utilisées comme prédicteurs des vulnérabilités exploitables, en testant trois méthodes différentes de sélection de caractéristiques : une basée sur une étude de corrélation, deux méthodes wrapper et l'analyse par composantes principales (ACP). Les expérimentations ont été effectuées en utilisant quatre algorithmes de classification différents. La combinaison qui a donné les meilleurs résultats est l'approche de sélection Wrapper Subset et les forêts aléatoires avec des résultats qui atteignent les 84% de f-mesure. Ces résultats montrent que la différence entre une vulnérabilité qui n'a pas d'exploit et celle qui a un exploit peut potentiellement être caractérisée en utilisant les métriques logicielles bien sélectionnées. Cependant, les auteurs affirment que la prévision de l'exploitation des vulnérabilités est plus complexe que la simple prévision de la présence de vulnérabilités, ce qui nécessite des recherches supplémentaires.

4.4 Résumé et discussion

Dans ce chapitre, nous avons présenté une description des techniques d'apprentissage et d'exploration de données appliquées dans les travaux examinés. La description suit les étapes nécessaires à la construction de modèles de prédiction.

La première étape correspond au processus de sélection et d'extraction des caractéristiques connu sous le nom d'*ingénierie des caractéristiques*, qui est considéré comme la clé

du succès des systèmes d'apprentissage automatique (Domingos [102]). Les performances des modèles d'apprentissage automatique dépendent largement de la qualité des caractéristiques extraites des données, c'est-à-dire du pouvoir discriminant des caractéristiques. Cependant, l'ingénierie des caractéristiques est spécifique à un domaine (Domingos [102]) et chaque domaine d'application exige des fonctionnalités spécifiques, d'où la difficulté de trouver les bonnes caractéristiques pour chaque domaine applicatif. Dans le Tableau 4.1, nous pouvons constater que plusieurs modèles de prédiction de la vulnérabilité ont été construits sur la base de métriques logicielles. Cela est principalement dû au fait que ces métriques sont facilement disponibles et peuvent facilement être intégrées dans des représentations vectorielles. Seules quelques études ont appliqué des techniques d'analyse statistique pour la sélection ou la réduction des caractéristiques (Shin et al. [80] et Younis et al. [101]). Comme indiqué précédemment, la plupart des études de cette catégorie font état de résultats moyens, qui pourraient être liés à la mauvaise ingénierie des caractéristiques réalisée dans ce domaine. En se basant sur les études présentées, les techniques basées sur le text mining semblent être plus performantes. Pourtant, la forte demande de calcul est clairement un inconvénient, que la réduction de dimensionnalité ne peut que légèrement tempérer.

La deuxième partie présente les techniques de construction de modèles utilisées dans certains travaux. Toutes les études présentées utilisent des algorithmes de classification supervisée pour construire le modèle cible. Ceci est en effet trivial, puisque par définition le problème de prédiction de la vulnérabilité est un problème de classification. L'utilisation exclusive d'algorithmes d'apprentissage supervisé peut être liée à la facilité d'accès aux mesures logicielles en grande quantité, qui permet d'obtenir des données d'apprentissage étiquetées. Plusieurs algorithmes de classification ont été expérimentés. Parmi eux, nous retrouvons notamment la régression logistique, les réseaux bayésiens, le perceptron multicouches, les machines à vecteurs de support et les forêts aléatoires. Les forêts aléatoires, présentes dans plus de 50% des études, sont les plus utilisées, et semblent être les plus performantes. D'après Ghaffarian et al. [70], la plupart des études font état de performances similaires pour tous les algorithmes de classification. les auteurs arrivent à la conclusion que l'algorithme de classification n'est pas un facteur majeur dans les performances des modèles de prédiction de la vulnérabilité basés sur des métriques logicielles.

L'étape finale consiste à évaluer les performances du modèle construit. La méthode

d'évaluation dominante observée dans les travaux est la technique de validation croisée qui permet de calculer une matrice de confusion, et de rapporter des valeurs de précision et de rappel. Il s'agit d'une méthodologie d'évaluation bien connue et bien établie, adoptée par la majorité des études précédentes. Quelques études ont également utilisé la f-mesure pour qualifier les performances des modèles. Pour le calcul de ces métriques, les études se différencient sur les données à considérer. Quelques études se focalisent sur un projet à la fois alors que d'autres mélangent les données de tous les projets dans la phase d'apprentissage et de test. La première évaluation semble donner de meilleurs résultats. Néanmoins, en l'absence de bases de données de référence, il est très difficile de comparer toutes les études citées dans ce chapitre.

	Analyse de dépendances			Métriques logicielles					Analyse de texte		Facteur humain	
	Dep	Fonc	Dep.met	Comp	Coupl	Coh	Exe	Churn	Sac	N-gram	Dev	Commits
Neuhaus et al. [18]	X	X										
Neuhaus et al. [75]	X											
Nguyen et al. [19]			X	X	X							
Shin et al. [20, 76]				X	X							
Shin et al. [78]				X	X		X					
Shin et al. [80]				X	X						X	
Shin et al. [82]				X	X			X				
Zimmermann et al. [21]	X			X	X			X			X	
Chowdhury et al. [83]				X	X	X						
Chowdhury et al. [84]				X	X	X						
Morrison et al. [74]	X			X	X			X			X	
Moshtari et al. [85, 86]				X	X							
Alves et al. [87]				X	X	X		X			X	
Hovsepyan et al. [90]				X	X	X			X			
Scandariato et al. [27]									X			
Pang et al. [92]										X		
Walden et al. [94]				X	X				X			
Meneely et al. [95]											X	X
Bosu et al. [96]											X	X
Perl et al. [97]								X			X	X
Zhang et al. [98]				X	X				X			

TABLE 4.1: Métriques utilisées pour construire des modèles de prédiction de vulnérabilités

Chapitre 5

Résumé de la partie 2

Cette thèse se situe dans le domaine de la sécurité informatique et se focalise particulièrement sur l'identification et la prédiction des vulnérabilités logicielles. Dans ce sens, nous cherchons à développer des techniques d'assistance aux développeurs, leur permettant d'orienter leurs efforts vers les zones à risque en terme de sécurité logicielle. Pour ce faire, nous nous sommes intéressés à la modélisation prédictive des vulnérabilités qui s'appuie sur la construction de modèles à partir de données labélisées avec la/les vulnérabilité(s). Ainsi, une contrainte importante apparaît : les données doivent être annotées avec précision afin que les modèles puissent fournir des résultats quantitatifs et qualitatifs acceptables dans le domaine de la cybersécurité.

Cette partie a posé les bases des travaux existants dans le domaine de l'analyse des vulnérabilités dans le code source afin de mettre en évidence les principaux concepts utilisés. Tout d'abord, nous avons présenté quelques exemples de vulnérabilités logicielles, les manières dont elles peuvent être exploitées, et les conséquences de certains exploits et ce dans le but de montrer l'intérêt de détecter ces vulnérabilités et de les corriger le plus rapidement possible.

Dans cette partie, nous avons également fait un état des lieux des méthodes utilisées traditionnellement pour détecter les vulnérabilités logicielles et les méthodes plus récentes qui s'appuient sur des techniques d'apprentissage automatique et d'exploration de données. Dans la suite de ce manuscrit, nous tirons profit des deux catégories de méthodes pour répondre à nos objectifs de thèse. Nous utilisons l'analyse statique de code source afin de collecter les vulnérabilités logicielles. L'approche exploite plusieurs outils d'analyse statique et combine leurs résultats en appliquant une heuristique qui maximise les performances tout en réduisant le nombre d'erreurs. Cependant, l'utilisation des outils peut s'avérer coûteux d'autant plus que notre approche repose sur plusieurs outils conjointement pour une meilleure annotation des données. Les modèles de prédiction sont une

solution à ce problème car il est possible d'apprendre des modèles à partir des données en entrée, et prédire les vulnérabilités dans de nouveaux individus (code source).

En se basant sur le schéma représentant les différentes étapes de la modélisation prédictive de vulnérabilités(Fig. 4.1), nous suivons les même processus fondamentaux mais nous positionnons nos contributions majeures dans les étapes de collecte de vulnérabilités et d'évaluation des modèles de prédiction.

Nous commençons d'abord par le problème de collecte de vulnérabilités. En analysant les études précédentes, nous avons noté que le projet Mozilla Firefox a été largement utilisé. Ceci s'explique par le nombre élevé de vulnérabilités signalées et la facilité de collecte d'informations à leur sujet. L'utilisation d'un projet reste néanmoins étroitement liée aux chercheurs. Dans la plupart des cas, un nouveau groupe travaillant sur les VPM conduit à l'introduction d'un nouvel ensemble de données, ce qui empêche la réplication et la comparaison entre les études. L'explication de cette multiplication d'ensembles de données, qui parfois ne partagent même pas la même vérité de base, c'est-à-dire basée sur des rapports de vulnérabilité, ou sur les résultats d'un outil, réside dans le fait que la plupart ne sont pas accessibles au public.

Dans cette thèse, nous présentons, non seulement un corpus de sécurité, mais aussi une approche permettant d'en construire de nouveaux qui sont fiables et à jour en terme de vulnérabilités découvertes. Bien que l'utilisation d'un corpus standard permette la réplication et la comparaison des études, se baser sur un corpus qui date de plusieurs années n'est pas la bonne solution car le nombre de vulnérabilités croît de façon exponentielle depuis des années, que ce soit en nombre d'instances ou en nombre de familles de vulnérabilités. D'où l'importance de proposer une approche qui s'adapte à l'existant dans le domaine des vulnérabilités logicielles.

La partie suivante introduit nos contributions de thèse. La première contribution étant la conception d'un méta-scanner de vulnérabilités logicielles.

Troisième partie

Contributions

Chapitre 6

Conception d'un méta-scanner de vulnérabilités logicielles

Le frein majeur de toutes les études sur la vulnérabilité est l'absence de bases de données standards où les données sont annotées avec précision. Ce chapitre introduit notre première contribution de thèse qui est la conception d'un méta-scanner de vulnérabilités logicielles. Nous commençons d'abord par détailler les limites des approches existantes sur l'annotation automatique de code source. Ensuite, nous décrivons le processus que nous suivons pour relever ces limites. Enfin, la validation de cette approche est présentée suivie d'une analyse des résultats.

Ce chapitre est basé sur le travail publié dans les actes de la conférence International Conference on Information Security Practice and Experience (ISPEC2019) [73].

Sommaire

6.1 Introduction

L'un des nombreux facteurs expliquant le manque d'études empiriques dans ce domaine de recherche est l'absence d'ensembles de données standards sur la vulnérabilité, qui pourraient être utilisés pour construire puis évaluer des approches de détection de vulnérabilités. En effet, la plupart des chercheurs commencent par créer leurs propres ensembles de données ce qui complique la recherche dans ce domaine. En effet, les informations requises sont généralement rares et dispersées dans différents endroits. Par conséquent, plusieurs projets devraient être utilisés pour créer un tel ensemble de données afin d'assurer la généralisation. Idéalement, ces projets devraient comporter un grand nombre de vulnérabilités signalées, être sensibles à la sécurité et être open source, ce qui réduit le nombre de candidats.

La création d'un ensemble de données est intéressante pour le chercheur car elle garantit un contrôle et une compréhension totale des données. Pourtant, cela est contre-productif à grande échelle car de nouvelles vulnérabilités sont découvertes quotidiennement. Un ensemble de données peut rapidement être dépassé et fausser les résultats.

Une des méthodes les plus utilisées pour construire des corpus de vulnérabilités est d'utiliser des outils d'analyse statique axés sur la sécurité logicielle. Ces outils analysent le code source ou binaire et produisent des rapports avec les vulnérabilités détectées accompagnées généralement de leur localisation. L'efficacité de cette méthode a été évaluée par Walden et al. [91] où les auteurs ont montré une forte corrélation entre les alertes des outils et les réelles vulnérabilités. Plusieurs autres chercheurs ont étudié les performances des outils d'analyse de vulnérabilités (Section 3.2). Les conclusions les plus fréquentes sont que les outils ne couvrent pas les mêmes vulnérabilités et que leurs performances diffèrent d'une catégorie de vulnérabilités à une autre. Un autre problème récurrent lors de l'utilisation de ces outils est le nombre de faux positifs renvoyés ce qui rend les résultats peu fiables pour les développeurs. De ce fait, utiliser un outil pour construire un ensemble de données fiable reste discutable, particulièrement dans les études basées sur les modèles de prédiction. Les techniques d'apprentissage automatique tentent d'extraire des modèles pour se rapprocher au mieux des données utilisées dans la phase d'apprentissage. Par conséquent, lors de l'utilisation de données annotées avec un outil d'analyse de vulnérabilités, les modèles résultant se rapprocheront au mieux des résultats de l'outil

ce qui serait complètement inutile étant donné que les modèles n'amélioreraient pas les résultats de l'outil utilisé.

Pour remédier aux limites que présentent les outils d'analyse de code, quelques travaux se sont intéressés à leur combinaison en utilisant différentes heuristiques 3.3 :

— ***1-out-of-N*** : cette heuristique considère le code comme étant vulnérable dès qu'un des outils utilisés détecte une vulnérabilité. Cette méthode permet d'augmenter le nombre de VP mais également le nombre de FP, ce qui réduit les performances globales de la fusion.

— ***N-out-of-N*** : combine les résultats de plusieurs outils d'analyse statique et considère un code comme étant vulnérable si et seulement si tous les outils détectent une vulnérabilité dedans. Cette heuristique permet d'avoir des alertes plus précises que l'heuristique précédente et un nombre plus réduit de faux positifs. Néanmoins, le nombre de vulnérabilités non découvertes est plus important car il suffit qu'il y ait un seul outil qui ne détecte pas une alerte pour l'ignorer et ne pas la remonter aux développeurs.

— ***Vote majoritaire*** : cette approche récupère les alertes renvoyées par les outils pour une même portion de code. Lorsque le nombre d'outils qui détectent une vulnérabilité est supérieur au nombre d'outils qui ne la détectent pas, le code est considéré comme étant vulnérable. Autrement, il est considéré comme non vulnérable. Cette approche a le bénéfice de prendre en compte le nombre d'outils qui affirment une vulnérabilité ce qui donnerait plus de garantie sur la précision des alertes qu'avec la méthode 1-out-of-N mais moins de précision qu'avec la méthode N-out-of-N. Cependant, les performances des outils ne sont pas prises en compte dans un tel vote. Par conséquent, il est possible que le vote suive les outils qui ont de faibles performances au lieu de suivre les outils qui détectent la vulnérabilité efficacement.

Ces trois heuristiques mettent en place des règles rigides à appliquer pour l'annotation de tout individu et avec n'importe quel outil donné. Pour remédier à cette limite, nous proposons dans le présent chapitre une approche pour construire un méta-outil, que nous appelons CVMS (*Code Vulnerability Meta-Scanner*), qui combine des outils existants en

se basant sur leurs performances individuelles. Un outil construit avec notre approche permet d'identifier les vulnérabilités avec une meilleure précision que les outils existants et avec moins de faux positifs.

6.2 Approche générale

L'approche que nous proposons est inspirée de la méthode de vote majoritaire entre différents outils : récupérer les alertes renvoyées par tous les outils et les intégrer dans un vote pour prendre une décision cohérente. Nous ajoutons une étape importante dans ce processus qui est l'évaluation des performances des outils. Cette étape permet d'attribuer des poids à chaque outil dans la prise de décision. Ceci reviendrait à effectuer un vote pondéré entre les outils existants. Le processus décrivant notre approche est défini par la figure 6.1.

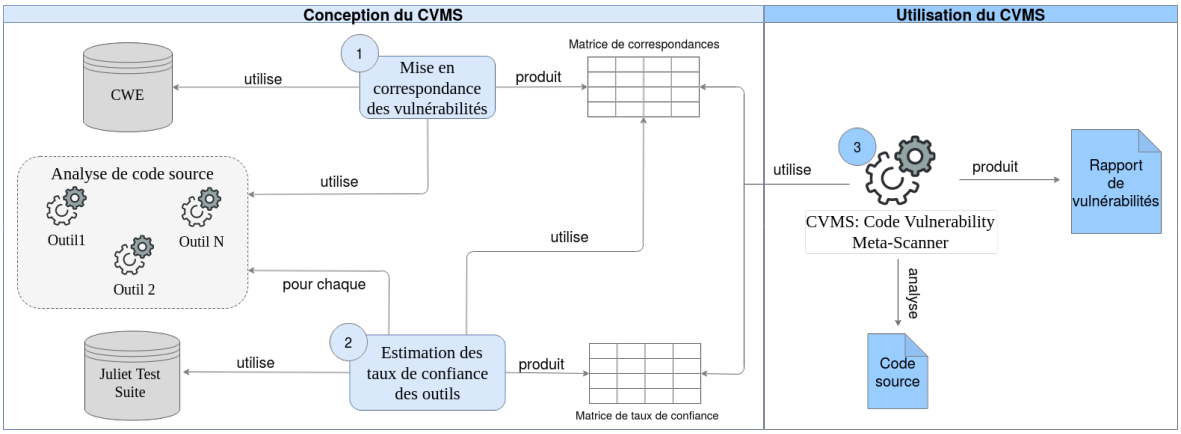

FIGURE 6.1 – Aperçu global du processus de construction d'un méta-outil d'analyse de vulnérabilités

L'existence d'une vulnérabilité est suspectée dès qu'un des outils la signale. Cependant, avant de prendre une décision, il est intéressant de consulter le résultat des autres outils concernant cette vulnérabilité dans la même portion de code. Dans d'autres termes, nous organisons un vote entre les différents outils avec la question suivante : y a-t-il une vulnérabilité x dans la portion de code y ? La réponse ne peut être que binaire : oui ou non. Cependant, l'organisation d'un tel vote, de manière automatique, soulève plusieurs problèmes :

1. La même vulnérabilité n'est pas identifiée de la même manière par les différents outils d'analyse de vulnérabilités.

2. La précision des outils d'analyse de vulnérabilités diffère d'une vulnérabilité à l'autre.
3. Comment agréger les informations ci-dessus pour produire le résultat du vote ?

La première étape du processus (Figure 6.1) consiste à mettre en place une correspondance entre les résultats des outils et les réelles vulnérabilités. Cette opération vise à fournir un référencement commun des vulnérabilités pour différents outils. Elle est basée sur la catégorisation CWE et fournit une mise en correspondance pragmatique entre les informations renvoyées par un outil et les vulnérabilités correspondantes et la catégorisation standard CWE. Cette étape répond au problème (1).

La deuxième étape de notre approche consiste à estimer le taux de confiance des outils d'analyse de vulnérabilité. Pour ce faire, nous évaluons chaque outil en utilisant la suite de tests Juliet [103] afin de déterminer ses performances pour chaque vulnérabilité. Nous considérons l'exactitude de détection comme un taux de confiance (solution au problème 2).

La dernière étape de notre approche consiste à utiliser les éléments produits dans les étapes précédentes lors de l'analyse du même code avec différents outils afin de déterminer l'existence de certaines vulnérabilités. Pour chaque partie de code (dans notre cas une méthode), nous recueillons les résultats de chaque outil pour les agréger sous forme de vote pondéré par le taux de confiance des outils (solution au problème 3).

Dans ce qui suit, nous décrivons tous les éléments nécessaires à la réalisation de notre approche.

6.3 Outils d'analyse de vulnérabilités logicielles

Les outils d'analyse statique sont sélectionnés sur la base des deux critères suivants :

— Le premier critère de sélection concerne la qualité de couverture de la vulnérabilité de l'outil. L'idée est de choisir des outils d'analyse statique qui ne couvrent pas exactement les mêmes vulnérabilités.

— Le second correspond à la méthode de détection des vulnérabilités utilisée par les outils. Bien que nous soyons limités aux méthodes basées sur l'analyse statique du code source, il existe plusieurs techniques dans ce domaine qui sont largement

exploitées dans le contexte de l'identification des vulnérabilités (modélisation de la vulnérabilité, IA, etc.). Il est donc intéressant de varier les techniques utilisées afin d'atténuer les faiblesses individuelles des outils sélectionnés.

Dans notre étude, plusieurs outils ont été éliminés à cause de leur indisponibilité (en termes de prix, de documentation, etc.) et d'autres pour le manque d'informations supplémentaires qu'ils fournissent. Nous avons sélectionné les trois scanners suivants : Find Security Bugs (également appelé SpotBugs) [11], Fortify Static Code Analyzer [12] et Yag suite [104]. Voici une description synthétique de chacun de ces outils :

— ***Find Security Bugs*** ou ***SpotBugs*** est un outil open source pour l'analyse statique de code source. SpotBugs s'appuie sur la modélisation des vulnérabilités et gère une base de connaissances contenant 128 modèles de vulnérabilité [105].

— ***Fortify Static Code Analyzer*** (SCA) utilise de multiples algorithmes et une vaste base de connaissances de règles de codage sûres pour analyser le code source. Pour traiter un code, Fortify SCA convertit le code source en une structure intermédiaire améliorée pour l'analyse de la sécurité. Le moteur d'analyse, qui se compose de plusieurs analyseurs spécialisés, utilise des règles de codage sécurisées pour analyser le code source afin de trouver les violations des pratiques de codage. Fortify SCA fournit également un générateur de règles permettant d'étendre et de développer les capacités d'analyse statique avec certaines règles spécifiques [106].

— ***Yag Suite*** est une suite logicielle basée sur des techniques d'apprentissage automatique pour la prédiction de vulnérabilités. L'une des forces de cet outil est qu'il évalue la pertinence de chaque vulnérabilité identifiée et estime sa criticité. De plus, la base de connaissances est affinée en fonction du domaine d'application en intégrant les réponses de certains utilisateurs [104].

Les premières expérimentations effectuées sur les cas de test Juliet ont montré que les outils sélectionnés couvrent différents ensembles de vulnérabilités. Le Tableau 6.1 résume les vulnérabilités de la suite de tests Juliet qui sont couvertes par au moins un des trois scanners sélectionnés. Les vulnérabilités sont triées par leur identifiants CWE.

Outils / Vulnérabilités	23	36	78	80	81	83	88	89	90
Fortify	X	X	X	X	X	X	X	X	-
Yag Suite	X	X	X	X	X	X	-	X	X
SpotBugs	-	-	X	-	X	X	-	X	X

Outils / Vulnérabilités	113	134	256	259	315	319	321	325	327
Fortify	-	-	X	X	-	X	X	X	X
Yag Suite	X	-	-	-	X	-	-	-	X
SpotBugs	X	X	-	X	-	-	X	-	X

Outils / Vulnérabilités	328	329	330	338	470	506	510	534	535
Fortify	-	-	-	-	-	X	X	-	-
Yag Suite	-	-	X	X	X	-	-	X	X
SpotBugs	X	X	X	X	-	X	-	-	X

Outils / Vulnérabilités	539	566	601	606	614	643	1004
Fortify	-	X	X	-	-	-	-
Yag Suite	X	-	X	X	X	-	-
SpotBugs	X	-	X	-	-	X	X

TABLE 6.1: Les vulnérabilités de Juliet couvertes par les outils sélectionnés

Sur les 113 failles présentes dans les cas de test Juliet, seules 34 vulnérabilités sont détectées par au moins un des outils d'analyse statique. Bien que ce pourcentage puisse être considéré comme faible, il est à noter que certaines failles présentes dans les cas de test sont relatives à de simples défauts et ne sont liées à aucune vulnérabilité. Les autres failles de sécurité ne sont détectées par aucun des trois outils. Ceci confirme les conclusions de Fonseca et al. [54] sur le faible pourcentage des vulnérabilités détectées par les outils automatisés. Une autre information que nous pouvons tirer du Tableau 6.1 est le chevauchement entre les ensembles de couverture qui est illustré par la Figure 6.2. Ce chevauchement est particulièrement recherché dans cette étude car il permet d'une part de renforcer les décisions des outils lorsqu'ils sont plusieurs à détecter une vulnérabilité et d'autre part, à soulever les cas où les outils ne retournent pas les mêmes résultats pour un cas de test. Ce dernier cas de figure est davantage détaillé dans les sections suivantes.

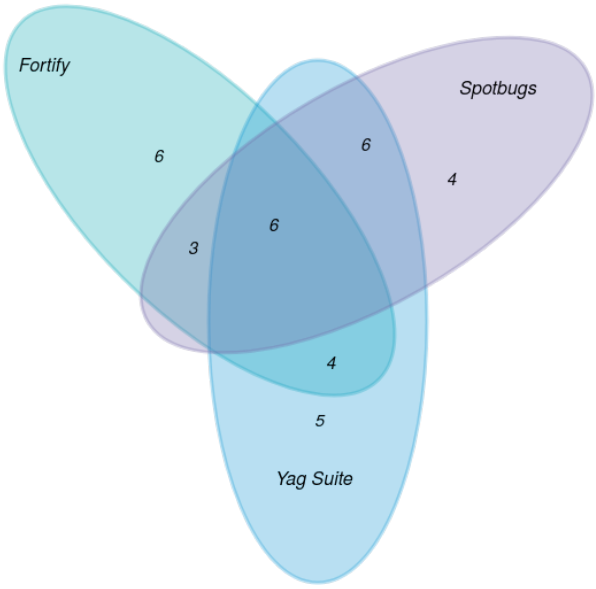

FIGURE 6.2 – Chevauchement entre les ensembles des vulnérabilités découvertes par les outils sélectionnés

6.4 *Benchmark* de vulnérabilités logicielles

L'approche proposée repose essentiellement sur une utilisation pragmatique des outils d'analyse de vulnérabilités existants. Pour ce faire, nous évaluons leurs performances afin

de les combiner efficacement. L'un des moyens les plus efficaces d'évaluer les outils d'analyse statique consiste à les utiliser pour analyser une partie de code dont les vulnérabilités sont connues, puis comparer leurs résultats avec les vulnérabilités réelles. Le choix des données à analyser est important et doit répondre aux besoins suivants :

- L'analyse d'un code doit retourner les mêmes résultats lorsqu'elle est effectuée avec le même outil ;
- Les données doivent contenir du code vulnérable et du code non vulnérable afin d'évaluer le pouvoir discriminant des outils utilisés ;
- Le code vulnérable doit être annoté avec précision et les vulnérabilités doivent être clairement identifiées ;
- Le code non vulnérable doit également être connu afin de déterminer la couverture d'un outil et son efficacité à ne pas retourner de fausses alertes (faux positifs).

Afin d'étudier les outils d'analyse statique, les chercheurs ont envisagé d'utiliser des logiciels "naturels" et "artificiels". Un logiciel naturel est un logiciel qui n'a pas été créé pour tester des outils d'analyse statique. Les applications logicielles libres, telles que le serveur web Apache (httpd.apache.org) et la suite OpenSSH (www.openssh.com), sont des exemples de logiciels naturels. Les logiciels artificiels, quant à eux, sont des logiciels qui contiennent des failles intentionnelles et sont créés spécifiquement pour tester des outils d'analyse statique. Les cas de test sont un exemple de logiciel artificiel.

Les expérimentations faites sur le code naturel et artificiel ont montré que l'utilisation du code naturel présente souvent des défis, tels que :

- **Évaluer les résultats des outils pour déterminer leur exactitude :**
  Lorsqu'un outil d'analyse statique est utilisé sur le code naturel, chaque résultat doit être examiné pour déterminer si le code présente effectivement le type de défaut spécifié à l'endroit indiqué (c'est-à-dire si le résultat est correct ou s'il s'agit d'un faux positif). Cet examen n'est pas trivial pour la plupart des résultats sur le code naturel et souvent, l'exactitude d'un résultat donné ne peut être déterminée avec un degré élevé de certitude dans un délai raisonnable.

- **Identifier les défauts du code qu'aucun outil ne trouve :**
  Lors de l'évaluation des outils d'analyse statique, une liste standard de tous les

défauts du code est nécessaire afin d'identifier les failles que chaque outil n'a pas réussi à signaler. En ce qui concerne le code naturel, la création de cette "norme" est difficile, notamment pour identifier les défauts qui ne sont signalés par aucun des outils automatisés et qui ne peuvent donc être trouvés qu'avec un examen manuel du code.

— **Évaluation de la performance des outils sur des failles qui n'apparaissent pas dans le code :**
Lors de l'utilisation de code naturel, il est très probable, même en combinant différent projets, que le code ne contiendra pas tous les cas défectueux et non défectueux d'une faille. En effet, même lorsque la faille est détectée par un outil, celle-ci peut se présenter de différentes manières. Il est donc intéressant d'évaluer l'outil sur tous les cas possibles.

À partir de ces défis, le *Center for Assured Software* (CAS) de la *National Security Agency* (NSA) a décidé de développer des cas de tests artificiels pour tester les outils d'analyse statique. L'utilisation de code artificiel simplifie l'étude des outils en permettant de contrôler, d'identifier et de localiser les failles et les défauts inclus dans le code. Nous avons opté pour ces cas de test, appelés Juliet Test Cases, car cette suite de tests répond aux besoins que nous avons préalablement spécifiés.

6.4.1 Cas de test Juliet

La suite de test Juliet a été créée par le CAS et développée spécifiquement pour évaluer l'efficacité des outils d'analyse statique [107]. Il s'agit d'une collection de programmes C/C++ et Java présentant des défauts connus. Les cas de test utilisent la liste CWE comme base de dénomination et d'organisation. Ils couvrent 113 entrées CWE, mais seulement 11 des 25 erreurs logicielles les plus dangereuses du Top 25 de la CWE/SANS 2011. Les 14 erreurs restantes ne sont pas détectables par analyse statique du code.

La suite de test Juliet contient des fichiers de code source qui sont structurés dans différents dossiers (Figure 6.3). Chaque dossier couvre une entrée CWE et comporte des fichiers de code source qui contiennent plusieurs variantes possibles de la faille ciblée par le répertoire. Par exemple, le répertoire "CWE23" dans la figure6.3 comporte 444 cas de test avec différentes variantes possibles de cette vulnérabilité. Un cas de test nommé

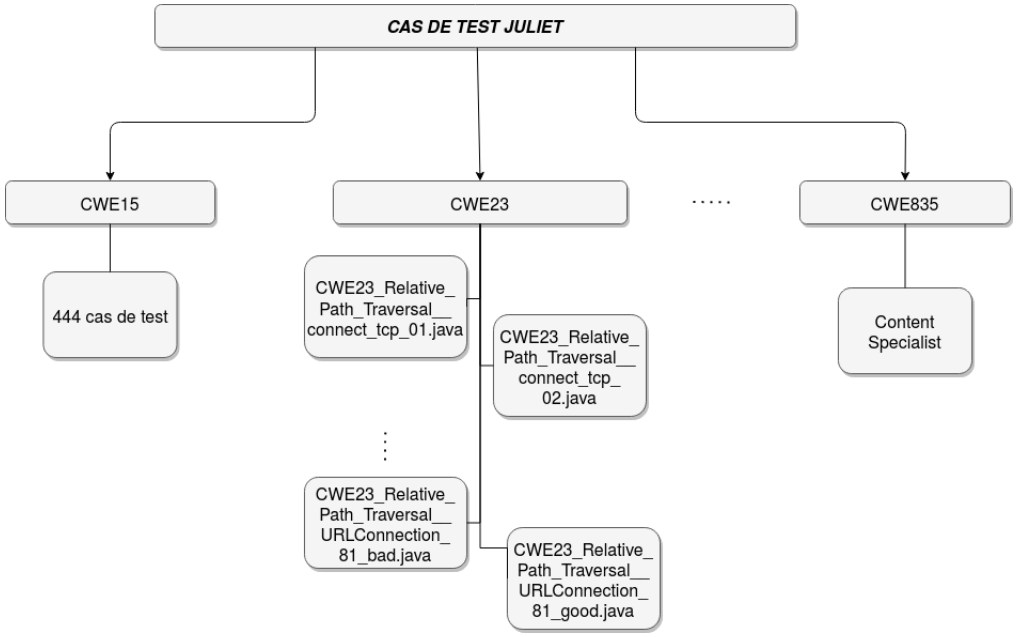

FIGURE 6.3 – Organisation des cas de test Juliet

"*CWE23_Relative_Path_Traversal___connect_tcp_01.java*" signifie que ce fichier cible la faille "*CWE23*" qui correspond dans la liste CWE à la faille "*Relative_Path_Traversal*" ; "*connect_tcp*" correspond à la variante de cette faille. Cette information est plus spécifique que l'identifiant CWE qui lui correspond ; "01" correspond à la forme la plus simple de cette variante ; et enfin l'extension ".java" spécifie le langage de programmation utilisé.

Outre les fonctions vulnérables (mauvaises), il existe aussi des fonctions non vulnérables (bonnes) qui contiennent presque la même logique que les mauvaises fonctions mais sans les erreurs de sécurité. Les bonnes fonctions peuvent être utilisées pour prouver la qualité des outils d'analyse statique axés sur la sécurité. Afin de pouvoir distinguer les méthodes vulnérables des méthodes non vulnérables, les noms des méthodes comprennent le terme "bad" pour signaler une faille dans la méthode, ou "good" lorsque la faille est corrigée. Il est également possible d'avoir un fichier dont toutes les méthodes sont vulnérables ou encore un fichier ne contenant que des méthodes non vulnérables (par exemple "*CWE23_Relative_Path_Traversal___URLConnection_81_**bad**.java*" et "*CWE23_Relative_Path_Traversal___URLConnection_81_**good**.java*" respectivement Figure 6.3).

La suite de test Juliet est également accompagnée d'un manifeste qui regroupe toutes les failles en spécifiant leur type et leur localisation exacte dans le code (Figure 6.4). Ces informations sont représentées par des balises XML *testcase*, *file* et *flaw* qui facilitent la récupération des données relatives aux failles.

```
-<testcase>
  -<file path="CWE89_SQL_Injection__Environment_executeBatch_04.java">
      <flaw line="43" name="CWE-089: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')"/>
      <flaw line="67" name="CWE-089: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')"/>
    </file>
  </testcase>
```

FIGURE 6.4 – Un exemple de cas de test dans le manifeste de Juliet

6.4.2 Utilisation des cas de test Juliet

Les cas de test ont été conçus de manière à ce que les résultats des outils d'analyse statique puissent être facilement évalués. Cette section décrit les résultats souhaités lors de l'exécution d'un outil d'analyse statique sur les cas de test.

6.4.2.1 Vrais positifs et faux négatifs

Lorsqu'un outil d'analyse statique est exécuté sur un cas de test, l'un des résultats souhaités est que l'outil signale une faille ciblée par le cas de test. Cette faille signalée doit se trouver dans une méthode dont le nom contient le mot "*bad*" (comme *bad*(), *badSource*(), ou *badSink*()) ou dans une classe dont le nom contient le mot "*bad*" (comme *CWE23_Relative_Path_Traversal__URLConnection_81_bad*). Un rapport correct de ce type est considéré comme un vrai positif. Si l'outil ne signale pas une faille du type de la cible dans une mauvaise méthode ou classe dans un cas test, alors le résultat de l'outil sur le cas test est considéré comme un faux négatif.

6.4.2.2 Faux positifs et vrais négatifs

Le deuxième résultat souhaité lors de l'analyse d'un cas de test est que l'outil ne signale pas de défauts de la catégorie ciblée dans une méthode ou une classe ayant le mot "*good*" dans son nom. Un retour incorrect dans une bonne méthode est considéré comme un faux positif. En contre partie, si l'outil ne retourne pas la vulnérabilité cible dans une bonne méthode, alors le résultat est considéré comme un vrai négatif.

6.4.2.3 Rapports de défauts non liés au défaut ciblé

Un outil peut également signaler des défauts avec des types qui ne sont pas liés au type de défaut cible dans un cas de test. Cela peut se produire dans deux cas de figure :

— Ces rapports de défauts peuvent signaler correctement des défauts du type non ciblé qui sont présents dans le cas de test. Les défauts de ce type sont connus sous le nom de défauts "accidentels". Les développeurs des cas de test ont tenté de minimiser les défauts accidentels et ont marqué les défauts accidentels inévitables avec un commentaire contenant la chaîne "*INCIDENTAL*". Cependant, de nombreuses failles accidentelles non commentées subsistent dans les cas de test. De ce fait, les utilisateurs ne doivent pas tirer de conclusions sur les rapports d'outils de failles non ciblées sans avoir étudié de manière approfondie le résultat rapporté.

— Les rapports de défauts peuvent indiquer des défauts qui n'existent pas dans le cas de test. Les rapports de défauts de ce type sont des faux positifs car il s'agit de rapports de défauts incorrects et non liés au type de défaut ciblé par le cas de test.

Les rapports de défauts de types non ciblés ne peuvent généralement pas être caractérisés comme corrects ou incorrects de manière automatisée ou triviale. Ils peuvent être déclenchés par des constructions de code communes qui sont répétées dans un grand nombre de cas de test (en raison du processus de génération automatisé utilisé pour créer les cas de test). Pour ces raisons, ces rapports de défauts sont ignorés dans notre évaluation des outils d'analyse de vulnérabilité.

6.4.3 Localisation des failles : Choix de la granularité

Les outils d'analyse de vulnérabilités retournent les résultats avec différents niveaux de granularité. La granularité représente l'entité qui est retournée par l'outil et peut correspondre à une ligne de code, une méthode ou encore une classe. Le choix de la granularité est important car il assure une comparaison équitable entre les outils utilisés mais également pour la suite de nos travaux sur les modèles de prédiction. Différents niveaux de granularité offrent différents avantages [74]. Par exemple, la ligne de code peut être considérée comme une alerte précise mais cette granularité est trop fine pour que les développeurs puissent identifier le problème [72]. La plupart des études adoptent

le niveau de granularité du fichier de composant suivant les conclusions de Morrison et al. [74]. Cependant, ce niveau peut ne pas être précis dans le cas d'études de vulnérabilité, surtout si les fichiers sont très volumineux. En effet, dans ce dernier cas, l'analyse devient coûteuse en temps d'exécution ou sujette à des erreurs si l'on utilise une analyse humaine. Pour cette raison, nous proposons d'utiliser la méthode de la classe comme la plus petite granularité. Cela représente un bon compromis entre le nombre de lignes de code à analyser et les informations que nous pouvons en déduire.

Les trois outils sélectionnés précédemment utilisent cette granularité et renvoient les vulnérabilités détectées dans chaque méthode. Cependant, les cas de test Juliet utilisent une granularité plus fine en spécifiant, pour toute vulnérabilité détectée, la ligne qui lui correspond. Par conséquent, afin de pouvoir utiliser les cas de test Juliet efficacement, il a été nécessaire de développer un programme qui prend en entrée un numéro de ligne de code et retourne la méthode concernée par la vulnérabilité.

Dans la suite du manuscrit, nous utiliserons le terme *individu* pour représenter une méthode de classe.

6.5 Mise en correspondance d'identifiants CWE

Les outils d'analyse de vulnérabilité sont largement utilisés pour la revue automatique du code. Afin d'augmenter leur taux de couverture et améliorer leur efficacité, nous proposons de combiner leurs résultats. Cependant, la combinaison d'outils soulève le problème suivant : chaque outil représente, identifie et classifie les vulnérabilités en utilisant sa propre dénomination [108]. Par exemple, la vulnérabilité Cross-Site Scripting (XSS) est appelée "***Insecure Interaction - CWE ID 079***" par Fortify, "***XSS-Servlet***" par Spotbugs, et "***xss.stored***" par Yag Suite. En raison de cette diversité de réponses pour une même vulnérabilité, il est très difficile d'utiliser plusieurs outils pour un objectif commun.

De plus, pour comparer l'efficacité des différents outils d'analyse, nous pouvons nous appuyer sur la classification CWE [35] pour classer les vulnérabilités et sur Juliet comme suite de tests. Malheureusement, s'il n'est évident de trouver l'identifiant CWE le plus abstrait qui correspond aux résultats retournés par les outils d'analyse, il est encore moins évident de trouver l'identifiant correspondant dans Juliet. En effet, les cas de test Juliet sont créés pour tous les types de défauts et chacun est nommé en utilisant l'entrée CWE

la plus pertinente. Par exemple, la vulnérabilité XSS (CWE-79) peut correspondre à au moins un des identifiants suivants dans Juliet :

— **CWE-80** : Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS) ;
— **CWE-81** : Improper Neutralization of Script in an Error Message Web Page ;
— **CWE-83** : Improper Neutralization of Script in Attributes in a Web Page.

Il n'est pas toujours utile de disposer de tels détails pour distinguer deux vulnérabilités. En effet, l'évaluation d'un outil se fait généralement en comparant ses résultats avec les vulnérabilités réelles. Cependant, évaluer des outils d'analyse de vulnérabilité en les confrontant à Juliet et en comparant les identifiants CWE n'est pas toujours adéquate. Comme indiqué précédemment, une alerte de sécurité renvoyée par un outil pourrait couvrir plusieurs vulnérabilités de Juliet lorsque : (i) Les cas de test de Juliet correspondent à des vulnérabilités moins abstraites que celles renvoyées par l'outil ; (ii) Les vulnérabilités sont fortement liées les unes aux autres.

Pour assurer une évaluation efficace et comparer les performances des outils d'analyse statique, nous procédons à une mise en correspondance entre les vulnérabilités couvertes par les outils et les identifiants CWE présents dans la suite Juliet. Ce processus consiste à créer une référence unique et commune à tous les outils sélectionnés, en rassemblant toutes les correspondances correctes entre les résultats des outils et les cas de test de Juliet. Ce processus comporte trois étapes :

1. **Analyser les cas de test Juliet** : l'objectif est de collecter toutes les vulnérabilités définies dans Juliet et couvertes par les outils utilisés.
2. **Attribuer un identifiant CWE** : pour chaque définition de vulnérabilité donnée par chaque outil d'analyse, trouver l'identifiant CWE lui correspondant.
3. **Proposer une mise en correspondance commune** : regrouper les définitions de Juliet qui sont liées à la même vulnérabilité dans une définition commune et significative.

Concrètement, dans la dernière étape de ce processus, nous partons de chaque étiquette retournée par chaque outil et nous analysons tous les individus où cette vulnérabilité a été détectée. Pour faciliter cette tâche, nous nous appuyons sur la catégorisation de CWE pour

vérifier la correspondance entre les résultats des outils et les identificateurs de vulnérabilité sur Juliet. La catégorisation CWE a été mise en place par l'organisation MITRE [35] et consiste en une organisation hiérarchique entre les vulnérabilités. Les rôles attribués aux identifiants représentent plusieurs niveaux d'abstraction, du plus abstrait au plus concret, comme suit :

- **Catégorie** : une entrée CWE qui contient un ensemble d'entrées qui partagent une caractéristique commune ;
- **Classe** : une faiblesse qui est décrite de manière très abstraite, généralement indépendante de tout langage ou technologie spécifique.
- **Base** : une faiblesse qui est décrite de manière abstraite, mais avec suffisamment de détails pour déduire des méthodes spécifiques de détection et de prévention.
- **Variante** : une faiblesse qui est décrite à un très faible niveau de détail, généralement limitée à un langage ou une technologie spécifique.

De plus, les liens entre les différents niveaux sont représentés par de multiples relations telles que "***MemberOf***", "***ParentOf***", "***ChildOf***", "***CanAlsoBe***", etc. Nous avons utilisé toutes ces informations pour tracer les chemins possibles entre les identifiants de vulnérabilité. Le tableau 6.2 résume la catégorisation que nous proposons pour certaines définitions de vulnérabilités présentes dans la suite Juliet. Nous ne prenons en compte que les vulnérabilités qui ont été détectées par au moins un des outils d'analyse sélectionnés, à savoir Fortify, SpotBugs ou Yag Suite. D'après notre connaissance des autres outils d'analyse de vulnérabilités (pour la plupart en accès libre), les vulnérabilités couvertes par les trois scanners utilisés englobent presque toutes les vulnérabilités que les outils d'analyse de code existants peuvent détecter. Ainsi, le tableau de correspondance que nous proposons reste suffisamment réutilisable dans le cas d'un autre groupe d'outils de d'analyse de vulnérabilités logicielles.

Dans le cas où un nouvel outil d'analyse de code est ajouté avec de vulnérabilités non encore couvertes, ce tableau de correspondance devra être complété. Pour faciliter l'ajout d'une vulnérabilité à notre tableau de correspondance, nous donnons plus de détails sur la façon dont nous procédons sur un exemple de vulnérabilité, à savoir le Traversée de chemin (catégorie 1). En analysant les résultats des outils d'analyse de vulnérabilités sur les cas de test Juliet mettant en œuvre les failles CWE-23 et CWE-36, nous avons remarqué que Fortify renvoie le message "***Risk Management Resource - CWE ID 022***",

la suite Yag renvoie le message "***Injection.path***" tandis que Spotbugs ne détecte aucune vulnérabilité pouvant correspondre à cette catégorie. D'après les définitions de CWE, la vulnérabilité CWE-22 couvre deux sous-catégories, à savoir CWF-23 et CWE-36 qui correspondent, respectivement, à la traversée de chemins relatifs et la traversée de chemins absolus. En effet, les liens entre la vulnérabilité CWE-22 et les vulnérabilités CWE-36 et CWE-23 sont représentés dans la catégorisation du CWE par la relation "*parentOf*" ("*childOf*" inversement). Par conséquent, cette relation est confirmée par la catégorisation proposée par le CWE.

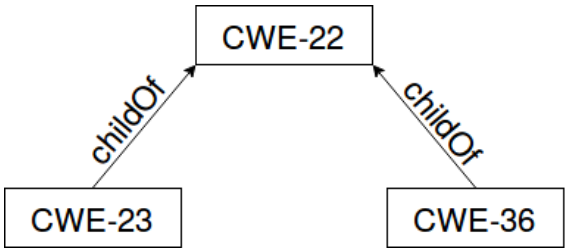

FIGURE 6.5 – Relations CWE entre les vulnérabilités CWE22, CWE23 et CWE36

Identifiant	Vulérabilités renvoyées par les outils	Vulnérabilités correspondantes dans la suite Juliet
1	Path traversal (CWE-22)	- CWE-23 : Relative path traversal - CWE-36 : Absolute path traversal
2	Os command injection (CWE-78)	- CWE-78 : OS Command Injection - CWE-506 : Embedded malicious code - CWE-88 : Argument injection or modification
3	XSS : Improper neutralization Of input during Web Page Generation (CWE-79)	- CWE-80 : XSS - CWE-81 : XSS Error Message Servlet File - CWE-83 : XSS Attribute Servlet connect tcp - CWE-535 : Info exposure shell error servlet
4	SQL injection (CWE-89)	CWE-89 : SQL injection
5	Injection LDAP (CWE-90)	CWE-90 : Injection LDAP
6	Uncontrolled format string (CWE-134)	CWE-134 : Uncontrolled format string
7	Use of broken or risky Cryptographic Algorithm (CWE-327)	- CWE-256 : Plaintext storage pwd - CWE-319 : Cleatext Tx sensitive info - CWE-321 : Hard coded cryptographic key - CWE-325 : Missing required cryptographic step - CWE-327 : Use of broken or risky cryptographic algorithm - CWE-328 : Reversible One-Way Hash - CWE-329 : Not Using a Random IV with CBC Mode
8	Use of insufficiante random values (CWE-330)	- CWE-330 : Use of insufficiante random values - CWE-338 : Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

9	Use if externally controlled Input to select classes or code (CWE-470)	CWE-470 : unsafe reflection
10	Open redirect to untrusted site (CWE-601)	CWE-601
11	Unchecked loop (CWE-606)	CWE-606 : Unchecked Input for Loop Condition
12	XPATh injection (CWE-643)	CWE-643 : XPATh injection
13	Use of hard coded credentials (CWE-798)	- CWE-256 : Plaintext storage pwd - CWE-259 : Hard codded password - CWE-319 : Cleatext Tx sensitive info - CWE-321 : Hard coded cryptographic key
14	Reliance on untrusted Inputs in a security decision (CWE-807)	CWE-510 : Trapdoor
15	Incorrect authorization (CWE-863)	- CWE-566 : Authorization Bypass Through User-Controlled SQL Primary Key
16	Sensitive cookies in HTTPs session (CWE-1004)	- CWE-614 : Sensitive cookies in HTTPs session without 'secure' attribute - CWE-539 : Information exposure through persistent cookies - CWE-315 : Cleatext storage of sensitive information in a cookie - CWE-113 : HTTP response splitting - CWE-1004 : Sensitive Cookie Without 'HttpOnly' Flag

TABLE 6.2: Mise en correspondance des vulnérabilités retournées par les outils et celles spécifiées dans la suite de test Juliet

6.5.1 Estimation des performances des outils d'analyse de vulnérabilités

Afin de détecter les vulnérabilités d'un individu, nous devons l'analyser avec tous les outils d'analyse de vulnérabilités. Lors de nos premières expérimentations, nous avons remarqué que les performances des outils dépendent de la vulnérabilité ciblée et que aucun des outils n'est meilleur que tous les autres pour toutes les vulnérabilités. Ainsi, lorsque la confrontation d'un individu avec les outils donne des résultats divergents, nous avons besoin d'informations complémentaires afin de renvoyer une réponse commune et consistante. Nous pensons que l'exactitude de la détection d'une vulnérabilité donnée par un outil donné peut être très utile pour cette opération. Cette métrique peut servir comme un taux de confiance attribué à un outil pour une vulnérabilité donnée.

Pour déterminer le taux de confiance associé à chaque vulnérabilité, nous avons utilisé les cas de test de Juliet. Ainsi, pour chaque outil d'analyse de vulnérabilité et pour chaque catégorie de vulnérabilité détectée, nous calculons le taux de confiance comme suit :

$$C_{o,v} = Exactitude_{o,v} = \frac{VP_{o,v} + VN_{o,v}}{VP_{o,v} + FP_{o,v} + VN_{o,v} + FN_{o,v}} \tag{6.1}$$

Où :

— $C_{o,v}$ est le taux de confiance C de l'outil o pour la vulnérabilité v
— $VP_{(o,v)}$ (Vrais Positifs) représente le nombre d'individus bien détectés par l'outil o pour la vulnérabilité v.
— $VN_{o,v}$ (Vrais Negatifs) représente le nombre d'individus considérés comme ne contenant pas la vulnérabilité v, et qui sont réellement non vulnérables.
— $FP_{o,v}$ (Faux Positifs) représente le nombre d'individus que l'outil o a considéré comme contenant la vulnérabilité v alors qu'ils ne la contiennent pas.
— $FN_{o,v}$ (Faux négatifs) est le nombre d'individus où l'outil o n'a pas détecté la vulnérabilité v alors qu'ils la contiennent.

Le Tableau 6.3 résume les taux de confiance obtenus avec les trois outils d'analyse de vulnérabilités pour les vulnérabilité proposées. D'après ce tableau, nous pouvons noter que l'exactitude de détection varie entre 0,26 et 0,82 pour Fortify, entre 0.57 et 1 pour Yag Suite et enfin entre 0,55 et 1 pour SpotBugs. Ces fourchettes de variation confirment la nécessité d'attribuer des poids par catégorie de vulnérabilités.

En outre, les outils qui ne détectent pas une catégorie peuvent avoir un poids positif (>= 0). Cela est dû au nombre d'individus qui ne contiennent pas la vulnérabilité ciblée et pour lesquels l'analyseur ne détecte rien (VN). En effet, parmi les cas de test, certains ont été générés pour la faille cible mais sans la vulnérabilité afin d'évaluer la capacité de l'outil à distinguer entre les cas de test vulnérables et non-vulnérables. Dans ce cas, un outil qui ne détecte pas la vulnérabilité ne retournera aucune alerte pour ces cas de test et cela augmentera son nombre de vrais négatifs. Pour résoudre ce problème, nous ignorons simplement le résultat de cet outil pour cette vulnérabilité en fixant son taux de confiance à ∅.

Outil Identifiant	Fortify	Yag Suite	SpotBugs
1	0.82	0.64	0
2	0.42	0.61	0.56
3	0.76	0.74	0.67
4	0.8	0.75	0.69
5	-	0.64	0.76
6	-	-	0.83
7	0.33	0.57	0.71
8	-	1.0	1.0
9	-	0.64	-
10	0.82	-	0.8
11	-	0.70	-
12	-	-	0.84
13	0.54	-	0.66
14	0.26	-	-
15	0.82	-	-
16	-	0.71	0.71

TABLE 6.3 – Taux de confiance des outils sélectionnés pour chaque catégorie de vulnérabilités

6.6 Combinaison des outils

L'objectif de notre approche de construction de méta-outil n'est pas seulement de classer un code comme vulnérable ou non vulnérable, mais plutôt de trouver toutes les vulnérabilités qu'il contient. En effet, une même méthode peut contenir différentes catégories de vulnérabilités. Il est donc important de guider le développeur vers les catégories

de vulnérabilités qui doivent être analysées et/ou corrigées. Nous ne prenons pas en considération le nombre d'occurrence d'une vulnérabilité dans un code. À cet effet, dès qu'une occurrence de vulnérabilité est détectée, la méthode est annotée comme vulnérable et contenant la vulnérabilité en question. L'objectif est d'indiquer la présence des catégories de vulnérabilités mais pas leur nombre d'occurrences.

La présence ou l'absence d'une catégorie de vulnérabilité dans un individu donné (une méthode d'une classe) est calculée en utilisant le taux de confiance (noté C ci-dessous) précédemment déterminés pour tous les outils. Pour chaque individu, la présence ou l'absence d'une vulnérabilité donnée est calculée comme suit :

$$V(v) = \begin{cases} 1 & \text{if } \sum_{i=0}^{K} C(v)_{i+} > \sum_{j=0}^{M} C(v)_{j-} \quad i,j \in \{Outils\} \\ 0 & \text{if } \sum_{i=0}^{K} C(v)_{i+} < \sum_{j=0}^{M} C(v)_{j-} \end{cases}$$

Où :

— v représente la catégorie de vulnérabilités ;
— $C_{i+}(v)$ est le poids de l'$i^{ème}$ outil qui a détecté la vulnérabilité v dans l'individu ;
— $C_{j-}(v)$ est le poids du $j^{ème}$ outil qui n'a pas détecté la vulnérabilité v dans l'individu ;
— Les éléments K et M représentent respectivement le nombre d'outils qui ont ou n'ont pas détecté la vulnérabilité v.

En d'autres termes, pour chaque individu et pour chaque vulnérabilité, nous attribuons la valeur "1" lorsque les poids des outils qui ont détecté cette vulnérabilité sont supérieurs aux poids des outils qui ne l'ont pas détectée. Ceci revient à un vote pondéré entre les différents outils.

6.7 Validation du méta-outil proposé

L'objectif principal de cette étape est de valider l'approche consistant à construire un méta-scanner. Cette étape est réalisée en effectuant, dans un premier temps, une comparaison entre les résultats du CVMS construit avec ceux des outils d'analyse de vulnérabilité. Dans un second temps, notre approche est comparée avec les heuristiques qui ont été proposées dans la littérature afin d'en déduire les points forts et faibles.

6.7.1 CVMS vs. les outils d'analyse de vulnérabilités

L'objectif de cette première étape est de vérifier si notre approche donne de meilleurs résultats que les outils utilisés pris individuellement. Étant donné que les outils sélectionnés dans cette études sont évalués sur la base de leurs résultats sur les cas de test Juliet, il est intéressant d'utiliser les mêmes cas de test pour évaluer l'approche proposée et ainsi comparer ses résultats avec ceux obtenus avec les outils d'analyse de vulnérabilités. La comparaison étant effectuée sur des familles de vulnérabilité, la précision de l'approche est calculée pour chaque catégorie de vulnérabilité **(voir colonne 5 du Tableau 6.4)**. Comme le montre le Tableau 6.4, nous pouvons constater que les résultats de notre CVMS convergent souvent vers les résultats du (des) outil (s) ayant la meilleure précision pour les catégories de vulnérabilité considérées.

Outil Identifiant	Fortify	Yag Suite	SpotBugs	**CVMS**
1	**0.82**	0.64	-	**0.82**
2	0.42	0.61	0.56	**0.63**
3	**0.76**	0.74	0.67	0.75
4	**0.8**	0.75	0.69	0.77
5	-	0.64	**0.76**	**0.76**
6	-	-	**0.83**	**0.83**
7	0.33	0.57	**0.71**	0.57
8	-	**1.0**	**1.0**	**1.0**
9	-	**0.64**	-	**0.64**
10	**0.82**	-	0.8	**0.82**
11	-	**0.70**	-	**0.7**
12	-	-	**0.84**	**0.84**
13	0.54	-	**0.66**	**0.66**
14	0.26	-	-	0.26
15	**0.82**	-	-	**0.82**
16	-	**0.71**	**0.71**	**0.71**

TABLE 6.4 – Comparaison des résultat du CVMS avec ceux des outils d'analyse de vulnérabilités

Néanmoins, certaines exceptions doivent être soulignées, en particulier pour la catégorie 7 où notre approche a une précision de 0,57. Ceci est dû au fait que la somme des poids des deux outils les plus faibles est plus forte que le poids du troisième outil. Pour les autres catégories, nous enregistrons une performance similaire à celle de (des) outil (s) le (s) plus efficace (s).

Dans un deuxième temps, nous comparons les macro-moyennes des outils d'analyse de vulnérabilité et du CVMS indépendamment de la catégorie de vulnérabilité. La métrique utilisée pour la comparaison est l'exactitude de détection des outils. Concrètement, dans cette étape, nous évaluons deux moyennes différentes que nous présentons dans le tableau 6.5 :

— La première, nommée "*macro-moyenne*", est la moyenne de l'outil uniquement sur l'ensemble des vulnérabilités qu'il couvre. Nous la calculons comme suit :
  — Soit V l'ensemble de vulnérabilités détectées par l'outil o

$$macro - moyenne_o = \frac{\sum_{v \in V} Exactitude_{o,v}}{|V|} \tag{6.2}$$

— La seconde, que nous appelons la "macro-moyenne-globale", représente la moyenne de l'outil o sur l'ensemble des vulnérabilités couvertes dans cette étude, c'est-à-dire les 16 catégories :

$$macro - moyenne - globale_o = \frac{\sum_{v \in V} Exactitude_{o,v}}{16} \tag{6.3}$$

Performances	Fortify	Yag Suite	SpotBugs	CVMS
Macro-moyenne	0.61	0.7	0.74	0.72
Macro-moyenne-globale	0.35	0.44	0.51	0.72

TABLE 6.5 – Macro-moyennes de l'exactitude des outils d'analyse de vulnérabilité et du CVMS résultant

Bien que les résultats de notre approche soient similaires à ceux des autres outils pour la première moyenne, ils sont beaucoup plus élevés lorsque l'ensemble des vulnérabilités des vulnérabilités est examiné (moyenne globale). Ceci répond à un objectif important de notre étude : augmenter l'ensemble des vulnérabilités détectées et améliorer les performances globales.

Enfin, le Tableau 6.6 présente les résultats globaux des outils en termes de précision, rappel et F-mesure :

— La précision se concentre sur l'équilibre entre les vrais positifs (VP) et les faux positifs (FP) et reflète le taux de bonnes alertes parmi toutes les alertes retournées

par l'outil o :

$$Précision_o = \frac{VP_o}{VP_o + FP_o} \tag{6.4}$$

— Le rappel reflète reflète le taux de vraies alertes parmi toutes les alertes qui devraient être retournées :

$$Rappel_o = \frac{VP_o}{VP_o + FN_o} \tag{6.5}$$

— La F-mesure est moyenne entre la précision et le rappel :

$$F - mesure_o = \frac{2 * Précision_o * Rappel_o}{Précision_o + Rappel_o} \tag{6.6}$$

Outils	Précision	Rappel	F-mesure
Fortify	0.67	**0.31**	0.43
Yag Suite	**0.91**	0.14	0.25
Spotbugs	0.74	0.18	0.29
CVMS	0.87	**0.31**	**0.46**

TABLE 6.6 – Performances moyennes des outils d'analyse de vulnérabilité et du CVMS résultant

D'après les performances présentées dans le tableau, il est facile de noter que le CVMS permet d'améliorer les performances globales des outils existants. En effet, le CVMS dépasse les autres outils que ce soit pour la métrique de rappel ou de F-mesure. Ces deux métriques restent relativement faibles avec tous les outils étudiés et nécessitent plus de recherche pour que ces performances soient améliorées. Nous notons également que la suite Yag a le meilleur taux de précision. Cet outil d'analyse a l'avantage d'évaluer la pertinence des vulnérabilités. En effet, cet outil ne vise pas la simple détection de la vulnérabilité mais plutôt la réduction du nombre de faux positifs. Pour réduire ce taux, nous suivons les préconisations des développeurs de l'outil et nous fixons le seuil de pertinence à 30%, ce qui conduit à ignorer les alertes de sécurité dont le pourcentage de pertinence est inférieur à cette limite. Par conséquent, la valeur de précision est très élevée mais la valeur du rappel est très faible.

Ainsi, sur la base de la suite de tests Juliet, nous pouvons conclure que notre approche donne globalement de meilleurs résultats que les outils d'analyse pris individuellement en termes de nombre de vulnérabilités et de performances.

6.7.2 CVMS vs les approches existantes

Plusieurs travaux se sont intéressés à la combinaison d'outils d'analyse statique. Parmi eux, certaines études combinent seulement les résultats des outils en supprimant les redondances alors que d'autres proposent des heuristiques de combinaison telles que 1-out-of-N, N-out-of-N ou encore le vote majoritaire. Nous avons jugé intéressant de tester les 3 heuristiques et de comparer leurs résultats avec ceux obtenus avec notre approche. Pour ce faire, nous utilisons également les cas de test Juliet et nous présentons les résultats dans le Tableau 6.7.

Approche Identifiant	1-out-of-N	N-out-of-N	Vote majoritaire	CVMS
1	0.81	0.55	0.64	**0.82**
2	0.42	0.54	**0.63**	**0.63**
3	**0.79**	0.64	0.75	0.75
4	**0.77**	0.71	**0.77**	**0.77**
5	**0.76**	0.55	0.64	**0.76**
6	**0.83**	0.71	0.71	**0.83**
7	0.4	0.6	**0.63**	**0.63**
8	**1**	0.63	**1**	**1**
9	**0.64**	0.55	0.55	**0.64**
10	0.8	0.55	**0.82**	**0.82**
11	0.7	**0.71**	**0.71**	0.7
12	**0.84**	0.71	0.71	**0.84**
13	0.54	**0.66**	**0.66**	**0.66**
14	0.26	**0.47**	**0.47**	0.26
15	**0.82**	0.55	0.55	**0.82**
16	**0.71**	**0.71**	**0.71**	**0.71**
Moyenne	0.73	0.66	0.71	**0.75**

TABLE 6.7 – Comparaison des résultats du CVMS avec les heuristiques existantes

Les approches existantes ne prennent en considération que le nombre d'outils qui détectent une vulnérabilité sans tenir compte de leurs performances. D'après ce tableau, nous pouvons constater que notre approche donne des résultats globaux encourageants en atteignant une exactitude moyenne de 75%. Cependant, les résultats sont parfois similaires à d'autres approches pour certaines catégories de vulnérabilités, notamment lorsqu'il n'y a qu'un outil qui détecte cette catégorie de vulnérabilités, ou encore lorsque deux outils détectent une vulnérabilité avec des poids égaux. Ceci revient principalement au nombre

d'outils qui ont été utilisé pour construire le CVMS, d'où la nécessité d'étendre les expérimentations en ajoutant de nouveaux outils.

6.7.3 Discussion

L'approche que nous avons proposée est basée sur l'utilisation d'outils existants pour améliorer les performances de détection des vulnérabilités. Comme la performance de notre approche dépend fortement de l'efficacité des outils utilisés, il est très important de sélectionner un ensemble d'outils efficaces qui s'appuient sur des approches variées (analyse statique, analyse dynamique, test de pénétration, etc.). Avec un plus grand nombre d'outils d'analyse de vulnérabilités couvrant une plus grande variété d'approches d'analyse, nous espérons augmenter la qualité des résultats obtenus.

La comparaison de la somme des poids positifs (scanners ayant répondu positivement) et négatifs (scanners ayant répondu négativement) nous permet d'utiliser la technique de vote pour décider si un individu est vulnérable ou non. Cependant, d'autres solutions peuvent être utilisées pour prendre cette décision, comme la logique floue [109] ou les réseaux bayésiens [110].

6.7.4 Menaces à la validité

Dans cette section, nous présentons les menaces à la validité de nos expérimentations de deux points de vue : les menaces internes et les menaces externes.

— **Menaces internes :** l'utilisation de l'exactitude comme mesure de performance pourrait altérer les résultats. En effet, lorsque le nombre d'individus sans vulnérabilité est important, cela augmente considérablement le nombre de vrais négatifs. Comme expliqué précédemment, si ce résultat est pris en considération pour les outils d'analyse statique qui ne détectent même pas la vulnérabilité, le résultat final sera biaisé. Pour surmonter ce problème, nous ignorons les réponses des outils qui ne sont pas censés détecter la vulnérabilité. En outre, l'utilisation des cas de test Juliet permet d'atténuer ce problème car il existe, pour chaque vulnérabilité donnée, des cas de test contenant la vulnérabilités et d'autres qui ne la contiennent pas.

— **Menaces externes :** le corpus que nous avons choisi concerne les applications écrites en Java. En conséquence, les résultats que nous avons obtenus peuvent être

différents dans le cas d'applications écrites avec un autre langage de programmation. De plus, le choix des outils d'analyse a imposé un sous-ensemble de vulnérabilités couvertes par la suite de test Juliet. Le choix d'autres outils donnera un autre sous-ensemble, il sera donc difficile de comparer les résultats obtenus avec les nôtres. L'approche proposée dans ce chapitre décrit le processus conçu pour construire un méta-scanner à partir de nombreux outils indépendants, nous avons choisi la suite de test Juliet comme référence pour la mesure des performances. Ainsi, l'utilisation d'un autre corpus de test peut donner d'autres résultats de performance.

6.8 Résumé et discussion

Dans ce chapitre, nous avons présenté une approche pour construire des méta-scanners de vulnérabilité de code en utilisant plusieurs outils existants. Nous nous sommes principalement concentrés sur la manière d'agréger les outils d'analyse de vulnérabilités afin d'améliorer leur efficacité. Ainsi, nous avons proposé une heuristique qui combine les outils en fonction de leur précision dans l'identification de chaque catégorie de vulnérabilités.

Nous avons expérimenté notre approche en utilisant trois outils d'analyse de vulnérabilités, à savoir Fortify, SpotBugs et Yag Suite. Nous avons comparé l'efficacité du CVMS construit avec les performances des outils séparément. Les résultats montrent que la combinaison de plusieurs outils permet, d'une part, de couvrir un ensemble de vulnérabilités plus large que les outils d'analyse pris individuellement. D'autre part, les performances de notre CVMS convergent vers les résultats du meilleur outil pour chaque catégorie de vulnérabilités.

Les premiers résultats obtenus avec notre approche prouvent son efficacité, mais de nombreuses améliorations peuvent être mises en œuvre. Nous en mentionnons plusieurs dans ce chapitre, notamment l'utilisation d'un plus grand nombre d'outils pour une meilleure précision. Il convient de noter que l'approche que nous présentons peut être appliquée à différents types d'analyse, comme l'analyse dynamique à l'aide d'outils basés sur des méthodes dynamiques ou une analyse hybride en intégrant des outils statiques et dynamiques.

Chapitre 7

Création d'un corpus de vulnérabilités logicielles

Les études sur les vulnérabilités en matière de sécurité exigent l'analyse, l'étude et la compréhension de réelles vulnérabilités. Cependant, comme nous l'avons constaté dans les chapitres précédents, il n'existe pas d'ensemble de données standard pour les études sur les vulnérabilités. La collecte manuelle d'un nombre suffisant est difficile et l'utilisation de méthodes d'annotation n'est pas toujours évidente. Pour faire face à ce problème et relever un des défis de la thèse, nous introduisons dans ce chapitre un large corpus présentant plusieurs vulnérabilités réparties en plusieurs catégories différentes.

Ce chapitre a fait l'objet de la publication [111] dans la conférence International Conference on Cyber Security for Emerging Technologies (CSET2019)

Sommaire

7.1 Corpus existants

Afin de mener des expériences et d'accroître la capacité de soutenir les pratiques de développement logiciel, une grande quantité de données est nécessaire. Les études empiriques de code sont coûteuses et leurs résultats sont difficiles à comparer. L'utilisation d'un corpus peut réduire le coût des études de code et permettre de comparer les mesures des mêmes artefacts.

La suite de test Juliet est très utile pour l'évaluation des outils d'analyse de vulnérabilités. Les cas de test couvrent un ensemble important de vulnérabilités et sont composés uniquement de code artificiel afin de simplifier le processus d'évaluation et de comparaison des outils. Pour déterminer si une erreur signalée par un outil de sécurité est correcte, il est nécessaire de savoir exactement où se trouvent les éventuelles failles de sécurité dans le code source. Cette tâche est difficile à réaliser sur du code naturel, car le code source complet devrait faire l'objet d'un examen minutieux. Pour le code artificiel, c'est une tâche beaucoup plus facile, car le code a été généré et documenté en tenant compte d'erreurs spécifiques.

Bien que l'utilisation des cas de test simplifie l'étude des outils d'analyse statique, les cas de test ne correspondent pas à des situations concrètes de la vie réelle et présentent quelques limites telles que :

— Les cas de test sont plus simples que le code naturel - Certains cas de test sont intentionnellement créés avec la forme la plus simple de la faille testée tant en terme de nombre de lignes de code (LOC) qu'en termes de nombre et de types de branches, de boucles et d'appels de méthodes. Cette simplicité peut fausser les résultats dans la mesure où les outils peuvent signaler des défauts dans les cas de test qu'ils signaleraient rarement dans un code naturel et non trivial.

— Les fréquences des défauts et des constructions non défectueuses dans les cas de test peuvent ne pas refléter leurs fréquences dans le code naturel - Chaque type de défaut est testé une fois dans les cas de test, indépendamment de la fréquence ou de la rareté de ce type de défaut dans le code naturel. Pour cette raison, deux outils qui ont des résultats similaires dans les cas de test peuvent donner des résul-

tats très différents dans le code naturel, par exemple si un outil trouve des défauts communs et l'autre des défauts rares.

Ces limites rendent difficile la généralisation de l'utilisation de la suite Juliet dans les études empiriques, particulièrement dans des applications d'apprentissage automatique. Lorsque des méthodes d'apprentissage automatique sont utilisées, il est important que les données (appelés individus) aient des caractéristiques variées dans le but d'éviter un sur-apprentissage entraînant la non généralisation des résultats. Comme montré dans le chapitre 4, les caractéristiques peuvent correspondre à des dépendances et importations, des métriques logicielles ou textuelles, des informations liées au facteur humain, etc. Toutes ces caractéristiques sont utilisées pour permettre au modèle d'apprentissage de distinguer un code vulnérable d'un code non vulnérable. Cependant, dans la suite de test Juliet, les cas de test sont trop simplifiés pour être utilisés dans des méthodes d'apprentissage automatique. Les cas de test ne contiennent que quelques lignes de code, avec une complexité minime et sans dépendance entre les méthodes. Par conséquent, il est judicieux d'utiliser d'autres corpus orientés vulnérabilités qui sont plus appropriés dans les études empiriques.

Actuellement, il existe plusieurs bases de données qui fournissent des informations sur les vulnérabilités. Par exemple, la base de données nationale sur la vulnérabilité (NVD) [32] offrent un grand nombre de vulnérabilités avec des détails importants. Néanmoins, les données ne sont pas structurées de manière à soutenir des études empiriques. D'autres référentiels tels que Stanford SecuriBench [112] ou WebGoat [113] contiennent des codes vulnérables et sont principalement utilisés à des fins éducatives. Les vulnérabilités sont peu nombreuses et ne sont pas structurées pour être exploitées dans des études empiriques.

En analysant les études sur les vulnérabilités, il en ressort que les auteurs ne publient pas leur ensemble de données en même temps que leur étude pour permettre la reproduction, ce qui obligent les nouveaux chercheurs à construire leur propre ensemble de données. Pourtant, certains ensembles de données accessibles au public existent, mais chaque ensemble concerne un problème spécifique et ne répond pas aux besoins des autres études. Perl et al [97] ont rendu leur ensemble de données disponible mais celui-ci ne contient que des commits suspectés de contribuer à une vulnérabilité et non des corrections de vulnérabilité réelles, ce qui réduit le champ de son utilisation. En outre, le lien vers cet ensemble

de données présent dans le document n'est plus disponible. Walden et al. [94] ont publié leur ensemble de données sur l'application PHP, qui a été réutilisée par Zhang et al. [98], mais cet ensemble de données s'avère relativement petit. Alves et al. [26] et Gkortzis et al. [114] ont tous deux publié des ensembles de données de vulnérabilité, mais il s'agit d'ensembles de données traitées, c'est-à-dire qu'ils ne contiennent que les métriques des fichiers vulnérables, et non les fichiers vulnérables eux même, ce qui empêche l'utilisation de nouvelles approches. De plus, comme la plupart des ensembles de données, ces ensembles de données, une fois publiés, ne sont plus jamais mis à jour. Il est donc nécessaire de disposer d'un vaste ensemble de données à jour sur les vulnérabilités et accessible au public.

Dans ce chapitre, nous relevons ce défi en proposant un ensemble de données de vulnérabilités qui peut être exploité dans de nombreuses études sur la vulnérabilité. Notre objectif est de fournir un corpus issu de projets du monde réel, couvrant différents domaines d'application et caractérisé par des vulnérabilités. Nous pensons qu'un tel corpus sera très utile aux chercheurs dans le domaine des vulnérabilités logicielles. Pour répondre à ce besoin, nous exploitons le méta-scanner que nous avons précédemment créé en utilisant les 3 scanners : SpotBugs, Yag Suite et Fortify Analyzer.

7.2 Critères du corpus

Un corpus utile dans les études empiriques sur la vulnérabilités doit répondre à des besoins spécifiques qui comprennent :

— **Informations sur la vulnérabilité :**
Contrairement aux autres études sur les vulnérabilités, nous voulons mettre à disposition des chercheurs un corpus standard qui peut être exploité dans plusieurs études sur les vulnérabilités. À cet effet, nous créons une première version du corpus qui contient les vulnérabilités et leurs localisations. Nous créons une deuxième version du corpus dans le chapitre suivant afin qu'il réponde au mieux aux besoins de notre étude.

— **Précision des informations :**
Les informations du corpus doivent être fiables et précises. Dans ce sens, il existe

plusieurs méthodes de création de corpus qui ont été détaillées dans la littérature. La technique la plus utilisée est de localiser les correctifs de sécurité à partir de projets GitHub importants et considérer le code avant le correctif comme vulnérable et après le correctif comme non vulnérable. La deuxième méthode largement utilisée est d'utiliser un outil d'analyse statique pour annoter des applications existantes. Nous nous sommes inspirés de cette dernière pour créer notre corpus, à savoir rassembler des applications Java et les annoter. Pour maximiser la fiabilité de l'annotation, nous utilisons le CVMS que nous avons construit dans le chapitre précédent car il permet d'obtenir des résultats plus fiables que les outils d'analyse de vulnérabilités pris individuellement.

— **Ensemble de données conséquent :**
Pour être utiles, les ensembles de données nécessitent un grand nombre de vulnérabilités. Malheureusement (et heureusement), les vulnérabilités sont très rares dans les systèmes logiciels et sont généralement dispersées dans le code de façon aléatoire. Par conséquent, il est impératif d'analyser un grand nombre de projets pour pouvoir rassembler un nombre important de vulnérabilités. Ce problème a conduit un grand nombre de chercheurs à choisir des projets sensibles en termes de sécurité et dont les vulnérabilités sont connues. Parmi ces projet, Firefox Mozilla et le noyau Linux sont les plus utilisés. Étant donné que nous ne récupérons pas les vulnérabilités à partir des projets, nous n'avons aucune contrainte sur le choix des projets à analyser.

— **Variabilité des caractéristiques :**
Bien que nous ne présentons aucune caractéristique sur les individus de notre corpus, nous pensons qu'il est intéressant que les individus du corpus soient différents les uns des autres en termes de taille (lignes de code), de compétences de développement, de domaine d'applications. Ces critères permettent d'assurer la validité externe des résultats obtenus avec notre corpus.

— **Reproductibilité et comparaison des études empiriques :**
Un autre critère important d'un corpus est de permettre la reproductibilité des études empiriques et de soutenir la comparaison des approches. En effet, évaluer les performances d'une approche ou d'un outil d'analyse statique en utilisant seule-

ment des cas de test artificiels pourrait fausser les résultats. Ce corpus peut être utilisé comme un complément d'évaluation car il contient des vulnérabilités réelles et connues.

— **Disponibilité des données :**
Bien évidemment, le corpus final est rendu publique afin qu'il puisse être utilisé dans d'autres études sur la vulnérabilité.

7.3 Choix des données

Comme pour tout problème d'annotation, le choix des données est d'autant plus important lorsqu'il s'agit de vulnérabilités logicielles. Comme notre objectif est de fournir un corpus Java annoté de vulnérabilités, la première étape de notre processus consiste à collecter un nombre important d'applications Java réelles. Certes, il existe un nombre important d'applications Java pouvant être analysées dans cette étude. Cependant, pour répondre à tous les critères qu'un corpus doit satisfaire, nous avons choisi d'utiliser un corpus déjà validé par la communauté du génie logiciel et largement utilisé par les chercheurs, à savoir le corpus Qualitas [115].

Le Corpus Qualitas est une collection de systèmes open-source écrits en langage Java. Le corpus a été développé par Tampero et al. [115] afin de réduire le coût des grandes études empiriques de code. Le corpus Qualitas apporte une contribution énorme à l'expérimentation dans le domaine du génie logiciel. Cependant, il existe plusieurs études, telles que les expériences qui s'appuient sur l'arbre syntaxique abstrait (AST) ou le bytecode, pour lesquelles un corpus compilé est nécessaire. Ainsi, Ricardo et al. [116] fournissent une version compilée de Qualitas appelée Qualitas.class Corpus. Le corpus contient une collection de systèmes, chacun d'entre eux comprenant un ou plusieurs projets. Qualitas compte un total de 111 systèmes et 802 projets internes et est considéré comme le plus grand corpus pour les études d'analyse de code.

7.4 Choix de la granularité

Le choix de la granularité est important car il permet d'assurer une bonne représentation des données. Nous avons opté pour la représentation la plus fine possible afin de

Caractéristiques	Valeurs
Systèmes	111
Lignes de code (LOC)	18 548 026
Projets internes	802
Packages	16 509
Classes	202 052
Méthodes	1 464 893

TABLE 7.1 – Caractéristiques du corpus Qualitas.Class.

permettre aux chercheurs de choisir la granularité qui leur correspond le mieux. Étant donné que les outils d'analyse de vulnérabilité sur lesquels s'appuie le CVMS renvoient les alertes par méthode, nous utilisons la même granularité pour présenter les résultats. Une méthode donnée peut contenir plusieurs vulnérabilités. De ce fait, nous attribuons à chaque méthode de notre corpus un vecteur de vulnérabilités contenant 16 cases correspondant aux 16 catégories de vulnérabilités étudiées. Ceci donne des garanties, d'une part sur la présence d'une ou plusieurs vulnérabilité, et d'autre part sur l'absence des autres vulnérabilités. Le CVMS permet d'étudier chaque type de vulnérabilité séparément, nous exploitons ces informations pour améliorer la précision de nos données.

7.5 Corpus `SecureQualitas`

En raison de problèmes de compatibilité entre les outils d'analyse de vulnérabilité et certaines applications de Qualitas, nous avons effectué nos expériences sur un sous-ensemble des applications de Qualitas qui comprend 41 systèmes Java contenant plus de 170k méthodes. Le corpus résultant, nommé `SecureQualitas`, est présenté sous un format CSV où chaque méthode est représentée par son chemin, son nom et un vecteur de taille égale au nombre de catégories couvertes par le CVMS (voir Figure 7.5. Une case i contient la valeur "0" lorsque la méthode ne contient pas la vulnérabilité i, sinon elle contient la valeur "1". Nous avons choisi ce format car une méthode peut contenir différentes catégories de vulnérabilités simultanément. Nous pensons que ce format simplifié serait très utile à plusieurs études dans le domaine des vulnérabilités du logiciel.

Chemin	Méthode	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
cobertura-1.9.4.1/ examples/basic/src/ com/example/ simple/Simple.java	square	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1

TABLE 7.2 – Exemple de représentation des données dans le corpus SecureQualitas.

Les résultats obtenus montrent que ce corpus contient 957 vulnérabilités réparties en 12 catégories différentes. Le tableau 7.3 présente toutes les catégories avec le nombre d'occurrence correspondant.

Identifiants	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Occurrences	671	5	0	2	2	0	0	64	59	1	0	3	15	108	25	2

TABLE 7.3 – Vulnérabilités découvertes dans le corpus `SecureQualitas`.

D'après ce tableau, il est évident que la répartition des vulnérabilités n'est pas homogène vis-à-vis des catégories de vulnérabilités. Les catégories les plus fréquentes dans le corpus `SecureQualitas` sont résumées dans le Tableau 7.4 et comprennent notamment :

1. Les traversées de chemins (CWE-22) où le logiciel utilise une entrée externe pour construire un nom de chemin destiné à identifier un fichier ou un répertoire situé sous un répertoire parent restreint, mais le logiciel ne neutralise pas correctement les éléments spéciaux à l'intérieur du nom de chemin qui peuvent entraîner la résolution du nom de chemin vers un emplacement qui se trouve en dehors du répertoire restreint.

2. La dépendance à l'égard de données d'entrées non fiables dans une décision de sécurité (CWE-807) : cette faille résulte de l'utilisation d'un mécanisme de protection qui repose sur l'existence ou les valeurs d'une entrée. Lorsque les décisions de sécurité telles que l'authentification et l'autorisation sont prises sur la base des valeurs de ces entrées, les attaquants peuvent contourner le mécanisme de protection. Sans cryptage suffisant, sans vérification de l'intégrité ou sans autre mécanisme, toute entrée provenant d'un tiers ne peut être fiable.

3. La faille CWE-330 est présente dans les logiciels qui utilisent des nombres ou des valeurs insuffisamment aléatoires dans un contexte de sécurité qui dépend de

nombres imprévisibles. Plus précisément, lorsqu'un logiciel génère des valeurs prévisibles dans un contexte nécessitant de l'imprévisibilité, il peut être possible pour un attaquant de deviner la prochaine valeur qui sera générée, et d'utiliser cette supposition pour se faire passer pour un autre utilisateur ou accéder à des informations sensibles.

4. La vulnérabilité CWE-470 correspond à une utilisation d'une entrée externe pour sélectionner les classes ou le code à utiliser sans vérifier s'il s'agit de classes ou codes inappropriés. Si l'application utilise des entrées externes pour déterminer la classe à instancier ou la méthode à invoquer, alors un attaquant pourrait fournir des valeurs pour sélectionner des classes ou des méthodes inattendues. Si cela se produit, l'attaquant pourrait alors créer des chemins de flux de contrôle qui n'ont pas été prévus par le développeur. Ces chemins pourraient contourner les contrôles d'authentification ou de contrôle d'accès, ou faire en sorte que l'application se comporte de manière inattendue.

5. Autorisation incorrecte (CWE-863) : cette faille correspond à un mauvais contrôle d'autorisation effectué lorsque qu'un utilisateur tente d'accéder à une ressource ou d'effectuer une action. Cela permet aux attaquants de contourner les restrictions d'accès prévues. En supposant qu'un utilisateur ait une identité donnée, l'autorisation est le processus qui consiste à déterminer si cet utilisateur peut accéder à une ressource donnée, sur la base des privilèges de l'utilisateur et de toute autorisation ou autre spécification de contrôle d'accès qui s'applique à la ressource. Lorsque les contrôles d'accès sont mal appliqués, les utilisateurs peuvent accéder à des données ou effectuer des actions qu'ils ne devraient pas être autorisés à effectuer. Cela peut entraîner un large éventail de problèmes, notamment l'exposition d'informations, le déni de service et l'exécution de code arbitraire.

Ces vulnérabilités peuvent être très dangereuses lorsqu'elles sont exploitées par des utilisateurs malveillants. Il existe d'autres catégories de vulnérabilités qui sont rares dans le corpus `SecureQualitas`. Ceci peut être dû à la rareté de ces vulnérabilités dans la réalité du terrain, aux performances des outils d'analyse de vulnérabilités dans la détection de ces catégories ou encore à la nature des projets analysés. Ces derniers correspondent principalement à des outils qui ne nécessitent pas d'être déployés sur le web, d'où la rareté

Catégorie de vulnérabilités	Nombre d'occurrences
- CWE-22 : Path traversal	671
- CWE-807 : Reliance on untrusted inputs in a security decision	108
- CWE-330 : Use of insufficiante random values	64
- CWE-470 : Use of externally controlled input to select classes or code	59
- CWE-863 : Incorrect authorization	25

TABLE 7.4 – Les vulnérabilités les plus récurrentes dans le corpus `SecureQualitas`.

des vulnérabilités orientées web. Une liste des projets qui contiennent le plus de vulnérabilités est présentée dans le Tableau 7.5.

Afin de valider les résultats de Qualitas, nous avons effectué une vérification manuelle sur une partie du corpus. Dans ce but, nous avons sélectionné au hasard 10% des individus (méthodes) sur la base d'un échantillonnage stratifié respectant les facteurs suivants (i) avoir à la fois des individus vulnérables et non vulnérables ; (ii) la partie vulnérable doit contenir des individus de chaque catégorie. Cette étape permet d'assurer l'équilibre de la répartition de ces facteurs au sein de la population sélectionnée. Les résultats de la vérification manuelle du corpus sont donnés dans le Tableau 7.6.

Les résultats obtenus montrent des performances très encourageantes. Pour les travaux de recherche visant à construire des modèles de prédiction, ces performances sont généralement acceptables. Ainsi, nous pouvons dire que notre approche de conception de méta-scanner fournit des résultats acceptables et assez précis sur du code source Java réel.

Nous avons appelé le corpus résultant de cette étude `SecureQualitas` et l'avons rendu disponible par le biais du lien suivant :
"`https://github.com/Brendan-LT/qualitas-vulnerabilities`"

7.6 Discussion

Le corpus que nous proposons peut être considéré comme une première contribution dans le domaine de la sécurité logicielle, et plus particulièrement pour le langage Java qui, à notre connaissance, n'a aucun corpus annoté avec les vulnérabilités. Il a été construit dans le but de faciliter les études empiriques sur les vulnérabilités.

Projets	Nombre de vulnérabilités
jasperreports-3.7.4	378
jgroups-2.10.0	105
jmeter-2.5.1	78
springframework-3.0.5	52
findbugs-1.3.9	47
poi-3.6	43
quartz-1.8.3	25
Cobertura-1.9.4.1	20
joggplayer-1.1.4	17
pmd-4.2.5	17
freecs-1.3.20100406	16
jag-6.1	15
marauroa-3.8.1	15
rssowl-2.0.5	14
ganttproject-2.1.1	10
webmail-0.7.10	10

TABLE 7.5 – Les projets qui contiennent le plus de vulnérabilités dans le corpus `SecureQualitas`.

Notons que, le corpus a été construit de manière automatique en utilisant le méta-scanner CVMS, décrit dans le précédent chapitre. Bien que le méta-scanner ait donné des résultats encourageants sur la suite de test Juliet, il est possible que les résultats soient moins bons sur des applications Java réelles. Les applications étant plus complexes, les outils peuvent être moins performants que sur la suite de test Juliet, ce qui pourrait entraîner une diminution des performances du CVMS. Pour garantir la qualité des résultats obtenus avec le CVMS, nous avons procédé à une vérification manuelle de 10% d'individus du corpus `SecureQualitas` tirés aléatoirement. Les résultats de la vérification ont été concluants. Ceci permet de donner une certaine garantie dans l'utilisation de `SecureQualitas`.

Par ailleurs, le nombre de vulnérabilités détectées peut paraître insignifiant par rapport au nombre de méthodes analysées. Cela reflète la réalité du terrain qu'il est intéressant de préserver dans le corpus. Il est évident que ce déséquilibre peut être un frein pour certains travaux (ex. apprentissage automatique). Dans ce cas, un pré-traitement des données du

Métriques	Valeurs
Exactitude	0.99
Précision	0.85
Rappel	0.89
F-mesure	0.87

TABLE 7.6 – Performances mesurées sur le corpus `SecureQualitas`.

corpus est nécessaire. Dans le chapitre qui suit, nous proposons quelques méthodes de près-traitement permettant d'atténuer le déséquilibre des données.

Enfin, il est intéressant de maintenir les données du corpus à jour afin qu'elles puissent être utilisées au fil du temps. La mise à jour peut être réalisée soit en corrigeant les individus du corpus ou soit en ajoutant de nouveaux individus. Pour la correction des individus, il est possible d'utiliser des outils d'analyse de vulnérabilité (ou le CVMS) pour ré-analyser les projets existants. Ces outils d'analyse de vulnérabilité évoluent en intégrant de nouvelles règles ou des patterns de vulnérabilités. De ce fait, il est possible de découvrir de nouvelles vulnérabilités dans les individus déjà annotés dans le corpus. Par ailleurs, le corpus est rendu publique à la communauté des chercheurs. Les résultats des autres études peuvent également servir pour corriger et/ou enrichir la version actuelle du corpus.

7.7 Résumé

Dans ce chapitre, le CVMS, construit dans le chapitre précédent, a été exploité pour annoter des projets Java. Cette expérience a permis de construire un corpus de codes Java (extrait de Qualitas) étiqueté avec les vulnérabilités qu'ils contiennent. Ce résultat est très utile pour les travaux de recherche dans le domaine des vulnérabilités du logiciel. Ce corpus est rendu disponible à la communauté via le lien suivant :
"`https://github.com/Brendan-LT/qualitas-vulnerabilities`"
Ce même corpus est utilisé dans le chapitre qui suit pour la construction de prédicteurs de codes vulnérables.

Chapitre 8

Modèles de prédiction du code vulnérable

Dans le chapitre 6, une approche de construction de méta-scanners a été proposée afin d'améliorer les performances des outils existants. Un prototype de l'approche est réalisé sur la base de trois outils d'analyse de vulnérabilité et évalué sur la suite de test Juliet. L'évaluation du méta-scanner résultant est encourageante car l'ensemble des vulnérabilités est plus important que celui des outils utilisés, de plus les performances de détection sont améliorées. Cependant, un méta-scanner construit avec cette approche reste fortement dépendant des outils utilisés dans sa création et son utilisation nécessite les résultats de tous les outils, ce qui entraîne un coût et un temps d'exécution importants.

Dans le présent chapitre, nous proposons de construire un modèle de prédiction du code vulnérable. Un tel modèle est autonome et ne nécessite l'utilisation d'aucun outil d'analyse de vulnérabilité. Dans la suite, nous donnons plus de détails sur les objectifs de cette étude et toutes les étapes permettant de les atteindre.

Ce chapitre est en cours de soumission à un journal.

Sommaire

8.1 Objectifs

La plupart des outils d'analyse de vulnérabilité de code sont basés sur la construction de modèles de vulnérabilités et sur une stratégie visant à analyser et à détecter toute occurrence de ces modèles dans le code. Si un modèle existe, cela implique la suspicion de l'existence de la vulnérabilité. Malheureusement, ce type d'outil peut générer un grand nombre de faux positifs. Cela est dû à deux problèmes :

1. Le modèle ne peut pas représenter parfaitement toutes les formes de la vulnérabilité ;
2. La mesure de similarité entre le modèle et le code est une approximation.

Pour remédier à ce problème, nous avons fait une première contribution qui vise à réduire le taux de faux positifs en combinant différents outils d'analyse de vulnérabilité. Dans ce sens, nous avons proposé de combiner les outils en se basant sur leurs performances vis-à-vis de chaque catégorie de vulnérabilités détectée. Un premier CVMS est construit sur la base de 3 outils d'analyse de vulnérabilité et évalué tant sur l'ensemble des vulnérabilités que sur les performances de détection. Il est évident que ce CVMS peut être utilisé comme outil d'analyse de vulnérabilité mais son utilisation nécessite de manipuler plusieurs outils conjointement, ce qui peut être très coûteux pour les développeurs.

Afin d'éviter ce problème, nous proposons de concevoir un modèle de prédiction basé sur le corpus construit dans le chapitre précédent (`SecureQualitas`). Cependant, en raison du nombre limité d'échantillons de certaines catégories de vulnérabilités (voir Tableau 7.3), il est très délicat de viser un système capable de rivaliser avec les outils existants pour chaque catégorie de vulnérabilités. Une stratégie plus viable consiste à concevoir un système qui se concentre sur une condition de vulnérabilité plutôt que sur l'identification des vulnérabilités.

Par ailleurs, la méthode de modélisation utilisée par les outils d'analyse de vulnérabilité nécessite une bonne compréhension de la faille et les raisons qui ont conduit à cette erreur afin de pouvoir déduire les différents motifs à rechercher dans le code. De ce fait, cette méthode ne peut être utilisée que sur un ensemble de vulnérabilités connu et facile à détecter. Ceci met le concepteur en retard vis-à-vis de l'attaquant.

Nous pensons qu'un modèle de prédiction binaire (vulnérable / non vulnérable) pour-

rait atténuer ce problème. Comme dans tout problème de classification, ce modèle de prédiction apprend à généraliser des caractéristiques d'individus donnés en entrée pour toute étiquette en sortie. Lorsque le modèle doit prédire le code vulnérable, il doit apprendre à distinguer les caractéristiques du code vulnérable de celles relatives au code non vulnérable, et ce indépendamment des catégories de vulnérabilités. De ce fait, nous construisons dans un premier temps un modèle de prédiction pour une classification binaire : le code est-il vulnérable ou non ? Un tel modèle pourrait être très utile en tant qu'assistant de surveillance de code dans un environnement de développement. Une alerte déclenchée par le modèle peut avertir le développeur pour des investigations supplémentaires à faire.

L'objectif de cette approche est double :

1. Construire des modèles de prédiction fiables et indépendants des outils existants
2. Détecter des vulnérabilités non encore connues.

Ce dernier point vise à étudier la capacité de généralisation des modèles de prédiction. En d'autres termes, nous cherchons à savoir s'il est possible d'apprendre à détecter un code vulnérable, à partir d'un ensemble de catégories de vulnérabilités connues, et ce même si le code contient une nouvelle catégorie de vulnérabilités. Ces modèles permettraient de protéger les systèmes informatiques des vulnérabilités connues et inconnues.

Notre hypothèse de travail que les mesures de qualité du code peuvent être utilisées pour distinguer le code vulnérable du code non vulnérable. En d'autres termes, nous considérons qu'une vulnérabilité est, en fait, un défaut de qualité dans le code. Ainsi, les questions de recherche soulevées dans cette étude sont les suivantes :

QR1 : Les modèles de prédiction de la vulnérabilité basés sur des métriques logicielles sont-ils capables de distinguer les codes vulnérables des codes non vulnérables ? Cette question ne concerne que les vulnérabilités utilisées lors de l'étape d'apprentissage.

QR2 : Les modèles de prédiction de vulnérabilités basés sur des métriques logicielles sont-ils capables d'identifier un code contenant une vulnérabilité non encore connue ? Cette question concerne les vulnérabilités non utilisées pendant l'étape de formation.

Dans ce qui suit, le processus de construction des modèles de prédiction du code vulnérable sera détaillé. Nous présenterons l'ensemble des données utilisées pour l'apprentissage et discuterons du choix des métriques logicielles. Étant donné que l'objectif est de fournir à l'utilisateur un modèle aussi précis que possible, nous décrirons une sélection de candidats utilisant une stratégie basée sur un ensemble optimal de Pareto [117]. Les modèles que nous voulons mettre en avant sont ceux qui ont de bonnes performances à la fois sur la population vulnérable et non vulnérable. Enfin, nous évaluerons l'analyseur à la fois pour les vulnérabilités incluses dans l'ensemble d'apprentissage et pour les vulnérabilités n'ayant pas participé pas à l'apprentissage.

8.2 Ensembles de données pour la construction de modèles de prédiction

La modélisation prédictive de la vulnérabilité (*Vulnerability Prediction Modelling (VPM)*) est un domaine d'étude relativement récent qui vise à classer automatiquement les entités logicielles comme vulnérables ou non vulnérables. Les études sur les modèles de prédiction nécessitent une sélection minutieuse d'un ensemble de données important et fiable. Les données doivent être caractérisées avec différentes métriques (*features*) et étiquetées avec la classe à prédire (*label*). Dans cette étude, nous exploitons les vulnérabilités détectées dans le corpus `SecureQualitas` pour construire un ensemble de données d'apprentissage. Le corpus `SecureQualitas` n'ayant que des étiquettes de vulnérabilités, nous devons attribuer à chacun de ses individus des caractéristiques le décrivant. Pour cela, nous utilisons des métriques logicielles qui ont montré leur efficacité sur d'autres ensemble de données et/ou langages de programmation. Le processus de création de l'ensemble de données est détaillé dans les sous-sections suivantes.

8.2.1 Choix de la granularité

Le choix de la granularité se pose dans cette étude également. Tout comme pour les deux premières contributions, nous avons choisi la granularité *méthode* qui représente un compromis entre la taille de l'individu et le nombre de métriques qui le caractérisent. En effet, choisir une granularité plus fine ne permet pas d'avoir assez d'informations sur la vulnérabilité ni assez de données pour caractériser correctement et efficacement un individu. Inversement, une granularité qui correspond à une classe ou fichier peut

contenir beaucoup de lignes de code, ce qui compliquerait l'analyse et la correction des vulnérabilités.

8.2.2 Métriques logicielles

Plusieurs chercheurs travaillant sur les VPM s'intéressent aux métriques logicielles pour caractériser le code des logiciels. Certains d'entre eux proposent de nouvelles métriques dédiées aux vulnérabilités [118, 119] alors que d'autres utilisent des métriques logicielles traditionnelles. Cette étude a pour but d'explorer la capacité des métriques logicielles pour caractériser un code Java. Contrairement aux travaux existants, nous ne faisons aucune restriction sur les métriques à utiliser. Bien au contraire, nous cherchons à regrouper toutes les métriques relatives à la qualité du code afin de pouvoir apprendre à caractériser le code vulnérable efficacement. Pour ce faire, nous utilisons l'outil JHawk [120] qui est un des outils qui fournissent le plus de métriques au niveau méthode. Les caractéristiques englobent 27 métriques classées en 4 grandes catégories :

1. Les métriques liées à la taille de la méthode : telles que le nombre de lignes de code (LOC), le nombre d'instructions Java, le nombre d'arguments, le nombre de commentaires, le nombre de variables déclarées et référencées, etc.

2. La complexité cyclomatique de McCabe qui mesure le nombre de chemins alternatifs possibles à travers un morceau de code. Une faible complexité cyclomatique peut être observée dans les méthodes comportant des énoncés simples, tandis que des valeurs de complexité plus élevées seront trouvées dans les méthodes comportant de nombreux énoncés *if*, boucles *for* et *while*, etc [121].

3. Les métriques de Halstead : ces métriques sont principalement basées sur le nombre d'opérateurs (noms de méthodes, opérateurs arithmétiques) et le nombre d'opérandes (variables, constantes numériques et chaînes de caractères). Les métriques de Halstead reflètent la complexité des lignes de code (ou instructions) individuelles.
   — La métrique la plus élémentaire est celle relative à la taille qui additionne simplement les nombres d'opérateurs et d'opérandes. Si un code comporte un petit nombre d'instructions avec une valeur Halstead élevée, cela suggère que

les instructions individuelles sont assez complexes.
— Le vocabulaire de Halstead donne une idée de la complexité des énoncés. Par exemple, l'utilisation répétée d'un petit nombre de variables est considérée comme moins complexe que l'utilisation d'un grand nombre de variables différentes.
— Le volume de Halstead utilise la longueur et le vocabulaire pour donner une mesure de la quantité de code écrit.
— La difficulté de Halstead utilise une formule pour évaluer la complexité basée sur le nombre d'opérateurs et d'opérandes uniques. Elle indique la difficulté d'écriture et de maintenance du code.
— L'effort de Halstead tente d'estimer la quantité de travail qu'il faudrait pour réécrire une méthode particulière.
— Enfin, le nombre de défauts tente d'estimer le nombre de défauts susceptibles de se trouver dans un morceau de code particulier [121].

4. L'index de maintenabilité est un ensemble de mesures polynomiales développé à l'Université de l'Idaho, en utilisant l'effort de Halstead et la complexité cyclomatique de McCabe, plus quelques autres facteurs relatifs au nombre de lignes de code (ou d'énoncés dans le cas de JHawk) et au pourcentage de commentaires. Cet index est utilisé principalement pour déterminer si un code a un degré de difficulté élevé, moyen ou faible à maintenir [121].

L'ensemble des métriques fourni par l'outil JHawk au niveau méthode est présenté dans le Tableau B.1. Pour chaque individu du corpus `SecureQualitas`, nous calculons toutes les métriques présentées avant de lui attribuer son étiquette de vulnérabilité

8.2.3 Utilisation du corpus SecureQualitas

Afin de mener à bien nos expérimentations, nous utilisons l'ensemble de données SecureQualitas, qui consiste en un ensemble de méthodes Java annotées avec le CVMS construit précédemment. Sur la base des résultats de validation de la sous-section 7.5, nous supposons que les données sont bien annotées et que cette base de données correspond à un ensemble de données réaliste. En effet, toutes les mesures des performances du CVMS ont été estimées à plus de 85%, ce qui devrait garantir une bonne précision des

résultats et donc réduire le nombre d'erreurs.

Comme indiqué précédemment, SecureQualitas est un corpus qui peut être utilisé pour plusieurs études empiriques de vulnérabilité. Cependant, il n'est pas évident d'utiliser un tel corpus dans la construction de modèles de prédiction de la vulnérabilité car ce corpus est très déséquilibré. Dans le domaine de l'identification des vulnérabilités des logiciels, beaucoup de chercheurs sont confrontés à ce défi important en raison du manque de données sur les vulnérabilités [74, 122]. Ce déséquilibre reflète une certaine réalité du terrain et le corpus SecureQualitas ne fait pas exception à cette règle. En effet, comme nous l'avons montré dans le tableau 7.3, le corpus comprend plus de 130k méthodes non redondantes (137996 précisément). Parmi ces individus, seules 952 méthodes ont été marquées comme vulnérables contre 137k méthodes non vulnérables. Cela signifie que la proportion de la population vulnérable est très faible par rapport à celle de la population non vulnérable.Les 952 vulnérabilités sont réparties en 12 catégories différentes. Ces catégories n'ont pas non plus un nombre équilibré de vulnérabilités. Par exemple, la catégorie de vulnérabilités 1 apparaît 671 fois dans le corpus, alors que certaines autres catégories n'apparaissent qu'une ou deux fois (catégories 10 et 4 respectivement). Les catégories de vulnérabilités 3, 6, 7 et 11, connues par le méta-scanner, n'apparaissent pas du tout dans le corpus.

Pour répondre à nos questions de recherche, à savoir la capacité des modèles à distinguer le code vulnérable du code non vulnérable et leur capacité à généraliser la prédiction à des vulnérabilités inconnues, nous avons trouvé intéressant de construire deux ensembles de données :

— Un ensemble de données d'apprentissage et de validation (**ED1**) : cet ensemble de données contient à la fois des individus vulnérables connus et des individus non vulnérables connus afin de garantir la pertinence de l'apprentissage.
— Un ensemble de données de test (**ED2**) : cet ensemble de données contient des individus présentant des vulnérabilités qui n'ont pas été utilisées dans la construction de ED1. Il s'agit ici de vulnérabilités faisant partie de catégories non apprises par le modèle et qui peuvent être considérées comme inconnues.

Avec un ensemble d'échantillons aussi limité, il est très difficile de faire des suppositions sur la topologie des classes de vulnérabilité. Dans le monde réel, l'ensemble des

échantillons de vulnérabilités est de toute façon très limité la plupart du temps. Comme un partitionnement est obligatoire pour l'apprentissage et l'évaluation, nous avons pris les décisions suivantes :

Population vulnérable :

La population vulnérable est divisée en deux sous-populations. La première population, dédiée à l'apprentissage et la validation (ED1), contient les 5 catégories de vulnérabilités les plus fréquentes. L'objectif est que le modèle puisse apprendre les caractéristiques intrinsèques du code vulnérable à partir de différentes catégories de vulnérabilités. Par conséquent, ED1 comprend les individus vulnérables des catégories 1, 8, 9, 14 et 15. En considérant la distribution des individus sur les catégories de vulnérabilité, nous réduisons le nombre de la catégorie 1 pour homogénéiser la population vulnérable. L'idée sous-jacente est d'éviter de rendre le classifieur plus sensible à cette catégorie et moins sensible aux autres catégories. Bien qu'il soit préférable d'avoir un certain équilibre entre les différentes classes, nous n'avons pas réduit le nombre d'individus des catégories des vulnérabilités et ce pour deux raisons :

— La première étant que le nombre d'individus est déjà assez faible lorsqu'il s'agit de vulnérabilités logicielles. Réduire ce nombre davantage diminuerait la probabilité que le modèle ait un nombre suffisant d'individus à apprendre.

— Étant donné que le modèle fait une classification binaire, toutes les classes de vulnérabilités sont réduites à un seul label : vulnérable (1). Tous les individus n'ayant aucune vulnérabilité détectée sont annotés non vulnérables (0). De ce fait, il n'est pas nécessaire de présenter le même nombre d'individus pour chacune des catégories.

Par ailleurs, les autres catégories de vulnérabilités présentent un nombre relativement limité dans SecureQualitas. Il s'agit de catégories de vulnérabilités rares dans le corpus et qui peuvent faire l'objet d'une étude plus approfondie. Dans cette étude, nous les utilisons précisément pour tester les performances des modèles à étendre leurs zones de couverture. Les catégories 2, 4, 5, 10, 12, 13 et 16 seront donc regroupées dans ED2.

En résumé, la population vulnérable du corpus SecureQualitas est répartie sur les deux ensembles d'apprentissage et de test comme indiqué respectivement dans les

Tableaux 8.1 et 8.2.

Population non vulnérable :

La répartition de la population non vulnérable doit tenir compte de la méthode de classification sélectionnée. Par exemple, les réseaux de neurones ou les SVM reposent sur une délimitation des classes en calculant des frontières qui séparent les différentes classes. Les classes étant binaires dans cette étude (vulnérable, non vulnérable), cela signifie que la phase d'apprentissage est conçue pour affiner les paramètres du classifieur afin de séparer la population de code vulnérable de l'ensemble de code non vulnérable. Le programme d'apprentissage implique la recherche du séparateur qui minimise une fonction de coût. Plus les individus non vulnérables sont utilisés dans l'apprentissage, mieux le séparateur sera ajusté pour la population non vulnérable. Toutefois, cela dépend fortement de l'ensemble, ainsi que de l'ordre des individus utilisés. Nous avons donc décidé d'éviter de faire de fortes suppositions sur le nombre d'individus non vulnérables. Ainsi, nous faisons varier ce nombre entre 100 et 1000 avec un pas de 100.

Les individus non vulnérables sont sélectionnés de manière aléatoire. Afin d'éviter le risque de faire une mauvaise sélection pour une taille donnée, nous avons fait 10 sélections aléatoires pour chaque taille. En résumé, nous disposons de plusieurs sous-ensembles d'apprentissage (ED1-1, ED1-2, ..., ED1-100) contenant le même nombre d'individus vulnérables mais un nombre différent d'individus non vulnérables.

Identifiant de vulnérabilités	Nombre d'individus
1	150
8	64
9	59
14	108
15	25
Total	406

TABLE 8.1 – Population vulnérable dans l'ensemble d'apprentissage et de validation (ED1)

Identifiant de vulnérabilités	Nombre d'individus
2	4
4	1
5	2
10	1
12	2
13	14
16	1
Total	25

TABLE 8.2 – Population vulnérable dans l'ensemble de test (ED2)

8.2.4 Construction des ensembles de données

La construction de l'ensemble de données final consiste à associer à chaque méthode, à partir des ensembles de données précédents, la liste des métriques qui la caractérisent ainsi que son étiquette de vulnérabilité. Cette dernière a une valeur binaire : 1 pour une méthode qui contient au moins une vulnérabilité et 0 sinon. Enfin, les données subissent un dernier traitement avant d'être exploitables. Cette étape consiste à normaliser les valeurs des métriques entre 0 et 1. Ce traitement a été nécessaire en raison de l'incompatibilité des unités de mesure entre les différentes métriques du logiciel. Ainsi, nous évitons de privilégier les métriques ayant les plus grands intervalles de variation comme le nombre de lignes de code.

8.3 Expérimentations

Dans cette étude, nous avons mené les expérimentations en utilisant les réseaux de neurones comme classifieur et en particulier des perceptrons à deux couches cachées. Nous divisons la phase d'évaluation en deux parties : la première vise à étudier l'efficacité des modèles de prédiction pour distinguer les codes vulnérables des codes non vulnérables pour des vulnérabilités connues (ciblant QR1), tandis que la seconde étudie la performance des meilleurs modèles sélectionnés pour identifier les codes contenant des vulnérabilités inconnues (ciblant QR2). Les vulnérabilités inconnues correspondent à des catégories de vulnérabilités que le modèle n'a jamais rencontrées au cours de sa phase d'apprentissage. Pour réaliser cette vérification, nous n'utilisons aucune instance des catégories de vulnérabilités de ED2 durant la phase d'apprentissage. Ainsi, nous pouvons considérer les vulnérabilités de l'ensemble de données ED2 comme complètement inconnues aux modèles.

8.3.1 Prédiction du code vulnérable : QR1

Pour répondre à cette question de recherche, nous avons évalué les VPM que nous avons construits selon deux critères importants :

— Le premier est la précision des alertes renvoyées aux développeurs. En effet, l'objectif des VPM est d'alerter les développeurs lorsque le code est vulnérable afin qu'ils puissent l'analyser et le corriger. Si les alertes ne sont pas pertinentes, en particulier lorsque les modèles renvoient de nombreux faux positifs, elles peuvent rendre le travail des développeurs plus difficile et les conduire à prendre de mauvaises

décisions.

$$Précision = \frac{TP}{TP + FP} \tag{8.1}$$

— Le deuxième critère concerne le rappel, qui est le taux de signalements pertinents renvoyés. En effet, si le modèle renvoie certaines alertes pertinentes mais en rate d'autres, les développeurs peuvent croire que leur code est non vulnérable alors qu'il contient encore des vulnérabilités.

$$Rappel = \frac{TP}{TP + FN} \tag{8.2}$$

Il est intéressant de trouver un équilibre entre ces deux métriques. Ceci se traduit par une évaluation multi-objectifs utilisant la précision et le rappel comme critères. Un des moyens les plus efficaces pour déterminer les modèles dominants par rapport à ces deux critères de sélection est d'utiliser l'optimum de Pareto [117] qui est déterminé à l'aide de la frontière de Pareto.

La frontière de Pareto est utilisée lorsqu'il n'est pas évident de trouver un modèle, ou un système de manière générale. Elle serait la meilleure solution pour tous les objectifs de l'étude. Un exemple de l'optimum de Pareto est montré dans la Figure 8.1 où les deux fonctions objectives f1 et f2 sont à minimiser. Si les situations préférables sont celles où f1 et f2 sont les plus faibles, le point C n'est pas sur la frontière de Pareto parce qu'il est dominé par les points A et B. Les points A et B sont tous les deux sur la frontière de Pareto, ce qui signifie qu'il ne sont dominés par aucune autre solution plus efficace.

Dans notre étude, nous cherchons à maximiser la précision et le rappel obtenus avec les modèles de prédiction. Pour ce faire, les performances des modèles sont récupérées à l'aide des rapports de classification et ce pour les deux populations étudiées (vulnérable et non vulnérable) ainsi que la performance globale des modèles. Un exemple de rapport de classification est présenté dans le Tableau 8.3 où la classe 0 représente la classe non-vulnérable et la classe 1 correspond à la classe vulnérable. Le support est le nombre d'occurrences d'une classe dans l'ensemble de données de test. Il est utilisé dans le calcul des performances moyennes comme un poids attribué à chaque classe.

La Figure 8.2 montre l'ensemble des modèles construits représentés par leurs coordon-

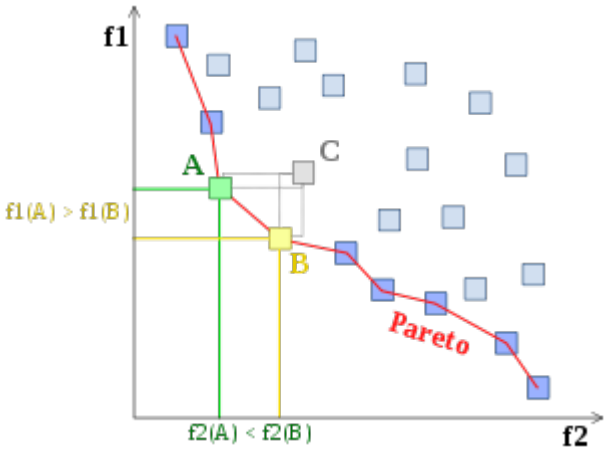

FIGURE 8.1 – Exemple d'un optimum de Pareto minimisant deux objectifs.

	Precision	Recall	F1-Score	Support
0	0.96	0.71	0.82	35
1	0.89	0.99	0.93	78
Avg / total	0.91	0.90	0.90	113

TABLE 8.3 – Exemple de rapport de classification.

nées (moyenne_rappel, moyenne_précision) correspondant au rappel et à la précision des modèles. Les individus (points dans la figure) ont une couleur qui représente la taille de leur population non vulnérable. On peut remarquer que plusieurs points sont superposés car ils ont les mêmes valeurs pour le couple (moyenne_rappel, moyenne_précision).

Cette figure montre que le meilleur modèle est clairement identifiable car il n'y a qu'un seul modèle qui domine tous les autres modèles pour les deux métriques étudiées. Cependant, lors de nos premières expérimentations, nous avons remarqué que les modèles ayant de "bonnes" performances globales, étaient souvent plus efficaces sur l'une des deux populations au détriment de la seconde. Ces cas sont dus à diverses raisons : mauvaise distribution des données, mauvaise sélection aléatoire de la population non vulnérable, mauvais apprentissage, etc.

Pour surmonter ce problème, nous avons décidé d'aller plus loin dans nos investigations en menant deux études séparément : l'une sur les individus vulnérables et l'autre sur la population non vulnérable. Ainsi, les modèles recherchés sont ceux qui maximisent la précision et le rappel sur les populations vulnérables et non vulnérables. Nous aboutissons

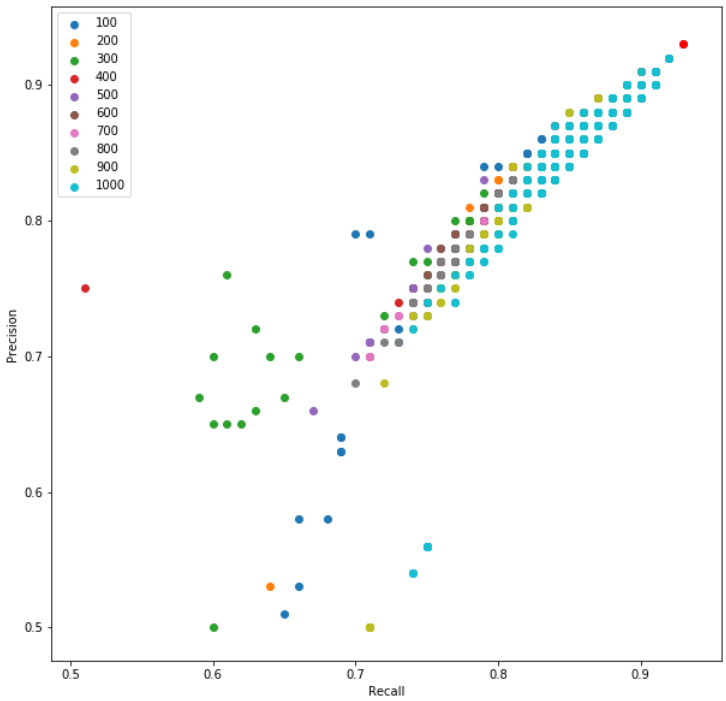

FIGURE 8.2 – Frontière de Pareto pour les performances moyennes des modèles (précision et rappel).

à une étude multi-objectifs avec 4 critères différents (deux pour chaque population). Pour résoudre ce problème et sélectionner les meilleurs individus, nous procédons comme suit :

1. Pour chaque individu (modèle), déterminer le nombre des individus qui le dominent (dominant au sens de l'optimisation de Pareto) selon ses performances sur la population vulnérable. Il s'agit de comparer cet individu avec tous les autres en utilisant comme critères le rappel et la précision sur la population vulnérable
2. Pour chaque individu, déterminer le nombre de ses dominants en fonction de sa performance sur la population non vulnérable. Comme pour la population vulnérable, ce traitement consiste à comparer l'individu avec tous les autres en utilisant le rappel et la précision sur la population non vulnérable
3. Utiliser les deux nombres de dominance dans un problème d'optimisation en utilisant la stratégie optimale de Pareto.

L'objectif de ces démarches est de trouver des modèles appartenant à la première frontière de Pareto. Cette frontière comprend tous les modèles de prédiction qui minimisent les nombres de dominants sur la population vulnérable et non vulnérable. Ainsi, ces modèles peuvent être considérés comme les plus efficaces parmi l'ensemble des modèles construits. Dans ce qui suit, nous détaillerons les étapes décrites ci-dessus et nous présenterons les résultats expérimentaux.

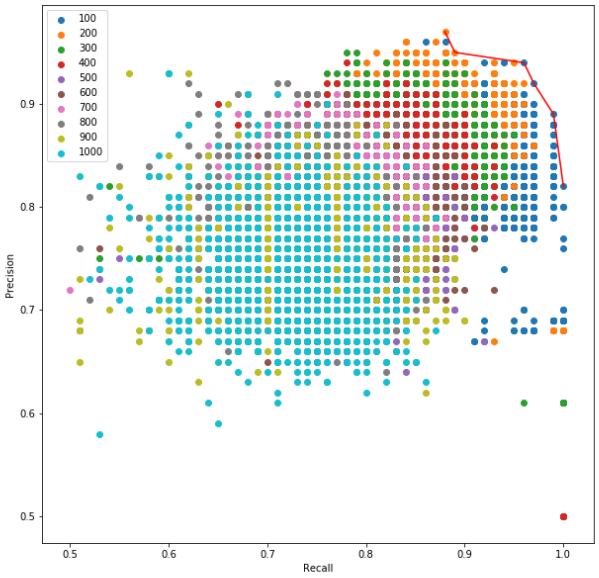

FIGURE 8.3 – Frontière de Pareto pour la population vulnérable.

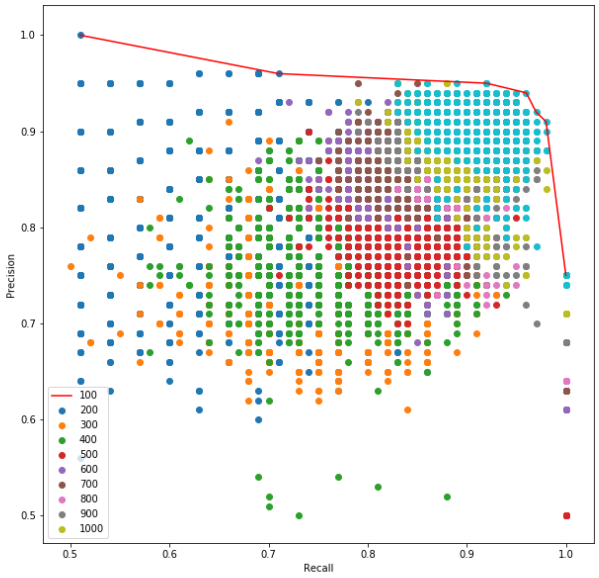

FIGURE 8.4 – Frontière de Pareto pour la population non vulnérable.

D'abord, nous représentons tous les modèles d'apprentissage dans les figures 8.3 et Figure 8.4. Nous utilisons comme coordonnées le couple (précision, rappel) sur les populations vulnérables et non-vulnérables respectivement.

Ensuite, nous exploitons les résultats présentés dans la figure 8.3 pour associer à chaque individu le nombre de modèles qui le dominent par leur performance (rappel, précision) sur la population vulnérable (vulnérable_dominants). Nous procédons de même sur la population non vulnérable afin de classer les individus selon leur performance (non-

vulnérable_dominants).

Enfin, nous représentons tous les individus sur un troisième graphique (voir Figure 8.5) en utilisant comme coordonnées le couple (vulnérable_dominants, non-vulnérable_dominants) puis nous identifions le premier front de Pareto qui minimise la dominance.

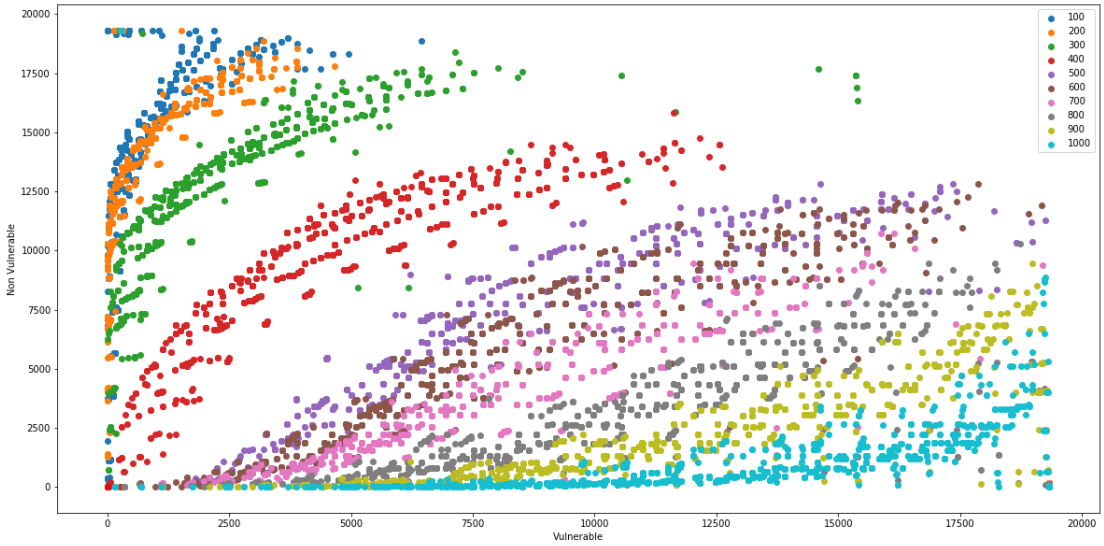

FIGURE 8.5 – La frontière de Pareto qui minimise le nombre de dominants.

La première frontière de Pareto n'apparaît pas clairement sur la Figure 8.5 car cette frontière contient deux modèles ayant les coordonnées (0,0). En fait, ces deux modèles n'avaient aucune dominance dans les deux populations et se trouvaient sur les deux frontières de Pareto identifiées dans les figures 8.3 et Figure 8.4. Nous présentons leurs performances dans le Tableau 8.4.

Modèles sélectionnés	Taille de la population non vulnérable	Vulnérable		Non Vulnérable		Moyenne	
		Précision	Rappel	Précision	Rappel	Précision	Rappel
Modèle 1	100	0.89	0.99	0.96	0.71	0.91	0.9
Modèle 2	100	0.82	1.0	1.0	0.51	0.88	0.85

TABLE 8.4 – Performances des modèles sélectionnés

Sur la base des résultats de précision et de rappel, nous constatons que le premier modèle a de meilleures performances que le second. Ceci est particulièrement évident dans

les résultats sur la population non vulnérable. La précision du modèle 2 est excellente (100 %) mais le rappel est très faible. Les performances globales des deux modèles appuient ce constat car le premier modèle enregistre des taux de précision et rappel plus élevés. Nous pouvons dès maintenant choisir le premier modèle de prédiction. Néanmoins, nous continuons notre étude sur les vulnérabilités inconnues avec les 2 modèles et nous ferons notre sélection en tenant compte de toutes les évaluations.

Avec les résultats ci-dessus, nous pouvons conclure que les modèles de prédiction basés sur des mesures logicielles peuvent distinguer un code vulnérable d'un code non vulnérable avec une très bonne efficacité.

8.3.2 Prédiction de vulnérabilités non connues : QR2

Dans la deuxième partie de l'évaluation, nous étudions l'efficacité des deux modèles sélectionnés pour identifier un code qui contient des vulnérabilités inconnues. Nous entendons par "vulnérabilités inconnues" toute vulnérabilité qui n'a pas été incluse dans l'ensemble de données d'apprentissage. Pour cela, nous évaluons nos modèles sur l'ensemble de données de test ED2 (Tableau 8.2) qui contient 7 catégories rares dans le corpus SecureQualitas. L'objectif principal est de vérifier si les caractéristiques apprises lors de la phase d'apprentissage permettent de détecter de nouvelles vulnérabilités.

Les performances de ces modèles sont détaillées dans le Tableau 8.5. Bien que nous n'ayons pas beaucoup d'exemples de vulnérabilités inconnues, nous pouvons tout de même constater que les modèles ont de bonnes performances de classification. Ces modèles ont réussi à atteindre de très bons taux de rappel (92 % pour le modèle 1) avec une précision de 100 %.

Modèles sélectionnés	Précision	Rappel	F-mesure
Modèle 1	1	0.92	0.96
Modèle 2	1	0.8	0.89

Table 8.5 – Les performances des modèles sélectionnés sur la base de test ED2

Une fois de plus, nous constatons que le modèle 1 est plus performant que le modèle 2. Nous pouvons ainsi choisir cet individu avec certitude car il répond à toutes nos exigences.

Le modèle que nous avons sélectionné est capable de reconnaître un code vulnérable uniquement sur la base de ses mesures de qualité, même si la vulnérabilité est totalement

inconnue au modèle. Nous validons donc la deuxième question de recherche et concluons que les métriques de qualité sont capables de prédire un code vulnérable comportant des vulnérabilités inconnues.

8.3.3 Discussion

Les figures 8.3 et Figure 8.4 montrent clairement que la performance des modèles dépend de la taille de la population non vulnérable. Dans la figure 8.3 (performance sur la population vulnérable), nous constatons que les deux mesures de qualité diminuent lorsque le nombre de la population non vulnérable augmente. Inversement, dans la figure 8.4 (performance sur la population non vulnérable), les deux mesures de qualité s'améliorent avec l'augmentation de la population non vulnérable.

En effet, lorsque l'ensemble des données d'apprentissage comporte un grand nombre d'individus non vulnérables, le modèle devient plus apte à reconnaître le code non vulnérable, ce qui améliore ses performances pour cette population d'individus. Cependant, lorsque la proportion de la population non vulnérable est beaucoup plus importante que celle de la population vulnérable, le modèle commence à mal classer les codes vulnérables. Ceci est également illustré dans la figure 8.5 qui montre que le nombre de dominants sur la population non vulnérable est plus important pour les modèles qui ont appris sur un petit échantillon d'individus non vulnérables (100 individus). Plus on ajoute d'individus non-vulnérables pendant la formation et moins on a de dominants dans cette catégorie jusqu'à atteindre la taille 1000 où le nombre de dominants diminue considérablement. D'autre part, on peut noter que la variation du nombre de dominants sur la population vulnérable est proportionnelle à la taille de la population non vulnérable.

Il est possible de croire que la façon la plus simple de résoudre ce problème serait de sélectionner simultanément des individus appartenant aux deux fronts de Pareto. Cependant, il n'est pas toujours facile de trouver des individus qui sont efficaces sur les deux populations. L'approche que nous proposons est générique, reproductible et applicable même si des modèles parfaitement dominants sur les deux populations n'existent pas.

La méthode Pareto a été largement utilisée dans divers domaines, notamment en finance et en économie. Dans cette étude, nous l'utilisons pour trouver un compromis entre la performance des modèles sur la population vulnérable et non vulnérable. Ayant 4 ob-

jectifs à maximiser, nous avons proposé une solution qui consiste à minimiser les individus dominants. En d'autres termes, nous recherchons les modèles qui sont les mieux classés selon nos critères. Néanmoins, si la frontière de Pareto contient plusieurs modèles, le développeur prendra en compte toutes ces informations pour spécifier le modèle optimal préféré. Une stratégie consisterait à utiliser la distance euclidienne pour choisir le modèle le plus proche du centre (coordonnées (0,0)).

L'approche de construction et d'évaluation des modèles de prédiction n'a été comparée à aucune autre approche existante. Ceci est principalement dû à l'utilisation d'ensembles de données différents et propres à chaque étude. En effet, les modèles de prédiction que nous avons créés donnent de bons résultats sur notre ensemble de données. Comparer leurs résultats à ceux obtenus dans les autres travaux de recherche est strictement impossible à cause de la différence importante des données et la non disponibilité de ces dernières.

8.4 Menaces à la validité

Dans cette section, nous discutons de la menace qui pèse sur la validité de l'expérimentation présentée ci-dessus. Ainsi, nous discuterons ci-dessous des menaces internes et externes.

1. **Menace interne**
   - Choix de la granularité du code : le choix de la granularité est très important dans les études de modèles prédictifs. Un code peut être divisé en différents aspects (paquet, classe, méthode, etc.). Nous avons choisi d'utiliser la granularité de la méthode car nous pensons qu'elle représente un compromis entre la taille de l'individu (LOC) et le nombre de métriques qui peuvent la caractériser. L'objectif principal de l'approche que nous proposons est de réduire la charge de travail du développeur en lui indiquant précisément où se situe la vulnérabilité. Dans cette perspective, l'idéal serait d'utiliser la ligne de code, sauf que pour cette granularité, il n'y a pas beaucoup de métriques qui lui sont associées. En revanche, l'utilisation de la granularité de la classe ou du paquet peut induire une charge de travail plus importante pour le développeur afin de localiser la vulnérabilité recherchée.

   - Choix de l'ensemble de données : les ensembles de données d'apprentissage et

de test sont construits à partir du corpus SecureQualitas. L'utilisation de ce corpus peut représenter un biais car les données peuvent contenir un certain bruit dû à l'utilisation du CVMS. Pour garantir l'exactitude des données, nous avons analysé manuellement une partie du corpus (10 %) sélectionnée aléatoirement et avons corrigé les erreurs détectées. Les performances obtenues sont supérieures à 87 % pour toutes les métriques étudiées ce qui garantit la qualité des données d'annotation. Un autre problème auquel nous avons été confrontés était le manque d'exemples pour l'apprentissage et le test. Ce problème a été amplifié par la disparité des proportions des vulnérabilités. Pour atténuer ce problème, nous avons réduit la taille de la catégorie de vulnérabilités qui était sur-représentée afin d'éviter que le modèle de prédiction soit particulièrement sensible à cette catégorie. Comme indiqué précédemment, toutes les catégories de vulnérabilités sont regroupées dans la classe vulnérable. De ce fait, le déséquilibre entre les différentes catégories n'influence pas beaucoup l'apprentissage.

— Choix du classifieur : nous avons choisi d'utiliser les réseaux de neurones car ces classificateurs sont des approximateurs universels capables de traiter des problèmes non linéaires. Il est possible que d'autres méthodes d'apprentissage ne donnent pas les mêmes résultats que ceux obtenus avec les réseaux de neurones. Cependant, l'objectif de cette étude n'est pas de comparer les méthodes d'apprentissage mais plutôt d'étudier les modèles de prédiction basés sur les métriques logicielles. L'approche d'apprentissage n'a pas une grande importance tant que les phases d'apprentissage et d'évaluation sont réalisées correctement.

2. **Menace extérieure**

— Java langage : en raison de l'utilisation du corpus Qualitas, nous avons expérimenté notre approche uniquement avec du code Java. Ainsi, nous ne pouvons pas généraliser nos résultats à d'autres langages de programmation, surtout s'ils ne sont pas basés sur une approche orientée objet. Cependant, comme nous avons utilisé des mesures orientées objet (OO), qui ne sont pas exclusives au langage Java, nous pensons qu'avec un autre langage OO similaire à Java (par exemple C#) les résultats peuvent être similaires.

— Enfin, nous avons utilisé un seul corpus pour faire nos expérimentations, à savoir le corpus Qualitas, ce qui peut générer un biais. Cependant, ce corpus est spécialement conçu pour les études empiriques concernant la qualité des logiciels. Il a été largement utilisé par la communauté de recherche en génie logiciel.

8.5 Résumé

Dans ce chapitre, nous avons présenté une approche pour construire des modèles de prédiction de vulnérabilités basés sur des métriques de qualité. L'objectif de ces modèles n'est pas seulement de prédire un code vulnérable, mais plutôt d'apprendre des vulnérabilités connues et d'étendre leur prédiction à des vulnérabilités inconnues. Par inconnu, nous entendons toute catégorie de vulnérabilités non incluse dans l'ensemble d'apprentissage.

Les résultats obtenus au cours de nos expérimentations permettent de valider l'hypothèse de recherche selon laquelle la sécurité peut être considérée comme un attribut de qualité pour le code logiciel. Par conséquent, il est tout à fait possible de prédire le code vulnérable en utilisant des métriques de qualité du code.

Les modèles de prédiction nous permettent de dire si un code est vulnérable ou non, mais nous n'avons aucune information sur la méthode de déduction suivie par le modèle de prédiction. Le fait de savoir qu'il est possible de déduire qu'un code est vulnérable ou non sur la base de ses mesures de qualité ouvre la voie à des travaux de recherche sur les mesures axées sur les bonnes pratiques pour éviter les vulnérabilités dans le code.

Quatrième partie

Conclusion

Chapitre 9

Conclusion

9.1 Contributions

La menace posée par les vulnérabilités logicielles croît de manière exponentielle. Ce phénomène est dû, d'une part, à l'omniprésence des logiciels, et d'autre part, au nombre important de failles existantes. Pour faire face à ce problème, plusieurs stratégies ont été élaborées au fil du temps. Néanmoins, le meilleur remède reste la prévention en sécurisant les systèmes logiciels. À cet effet, de nombreux éditeurs de logiciels décident d'intégrer la politique de sécurité et d'inspection dès le processus de développement.

Couvrir de manière continue l'ensemble du code est peu pratique et trop coûteux, en particulier pour des projets qui contiennent des millions de lignes de code. Afin de guider les efforts d'inspection de sécurité, les chercheurs ont proposé et évalué différentes méthodes. Parmi elles, nous retrouvons les méthodes de prédiction de vulnérabilités.

Le sujet de thèse, présenté dans le présent document, portait essentiellement sur les modèles de prédiction de vulnérabilités. Néanmoins, nos premières études nous ont montré d'importantes limites que nous devions relever. La plus importante étant le manque de données annotées qui peuvent être utilisées pour construire des modèles de prédiction. À cet effet, nous avons mis en place une chaîne de traitement complète allant de la création et de l'annotation automatique d'un corpus de sécurité jusqu'à la construction et l'évaluation des modèles de prédiction de vulnérabilités.

Créer soit même un ensemble de données annoté peut être intéressant, car il garantit un contrôle et une compréhension totale des données. Cependant, un corpus doit évoluer dans le temps car de nouvelles vulnérabilités apparaissent régulièrement. Un corpus qui n'est ni corrigé ni enrichi peut rapidement être dépassé et son utilisation risque de fausser les résultats. Ce travail continu dans le temps n'est pas un objectif pour une thèse, qui est limitée dans le temps.

Pour remédier à ce problème, la première contribution est plus axée sur l'approche de construction de corpus que sur le corpus lui-même. Dans le chapitre 6, nous proposons une approche de conception de méta-scanners permettant d'identifier des vulnérabilités de code efficacement. L'approche consiste à combiner plusieurs outils d'analyse statique en se basant sur leurs performances individuelles pour chaque catégorie de vulnérabilités. L'objectif de notre approche est de tirer profit des différents outils et d'agréger leurs résultats efficacement pour réduire le nombre de fausses alertes. Afin de pouvoir construire un corpus d'applications Java, nous construisons un premier méta-scanner en utilisant trois outils d'analyse de vulnérabilités, à savoir Fortify, SpotBugs et Yag Suite. Nous avons comparé l'efficacité du CVMS construit avec les performances des outils séparément. Les résultats montrent que la combinaison de plusieurs outils permet, d'une part, de couvrir un ensemble de vulnérabilités plus large que les outils d'analyse pris individuellement. D'autre part, les performances de notre CVMS convergent vers les résultats du meilleur outil pour chaque catégorie de vulnérabilités. Il convient de noter que le CVMS construit peut être utilisé comme outil de détection de vulnérabilités. Il nécessiterait d'utiliser plusieurs outils et permettrait de réduire le taux d'erreurs engendré par ces outils.

Notre deuxième contribution de thèse correspond au corpus SecureQualitas qui consiste en un corpus d'applications Java annotées avec les vulnérabilités qu'elles contiennent. Nous construisons ce corpus dans le Chapitre 7 en utilisant le CVMS construit précédemment. L'annotation consiste à attribuer à chaque méthode du corpus l'ensemble des vulnérabilités qu'elle détient. Ce corpus est rendu disponible à la communauté via le lien suivant :
"`https://github.com/Brendan-LT/qualitas-vulnerabilities`."

L'évaluation du méta-scanner construit dans le Chapitre 6 est encourageante car l'ensemble des vulnérabilités est plus important que celui des outils utilisés et les performances de détection sont améliorées. Cependant, un méta-scanner construit avec cette approche reste fortement dépendant des outils utilisés lors de sa création et son utilisation nécessite les résultats de tous les outils, ce qui entraîne un coût et un temps d'exécution importants. En réalité, la contribution réside dans l'approche proposée pour tirer profit d'un ensemble de scanners, bien que dans tous les cas nous restons toujours liés aux scanners utilisés pour la construction du méta-scanner.

Notre troisième contribution (Chapitre 8) a été de construire un modèle de prédiction du code vulnérable. Un tel modèle est autonome et ne nécessite l'utilisation d'aucun outil d'analyse de vulnérabilité. Nous avons opté pour l'utilisation de métriques de qualité pour caractériser le code et nous avons étudié les performances des modèles à la fois sur des catégories de vulnérabilités apprises par les modèles et sur des catégories non encore connues par le modèle. Nous avons étudié ce dernier cas pour voir à quel point nous pouvons anticiper le futur. Les résultats de nos expérimentations ont montré l'efficacité des modèles sur les deux populations de vulnérabilités : connues et non connues. Cependant, le choix des meilleurs modèles n'est pas toujours évident. En effet, un bon modèle de prédiction doit permettre une identification efficace des individus vulnérables, mais aussi des individus non vulnérables. Nous avons donc conduit une étude basée sur l'optimum de Pareto pour déterminer les modèles qui répondent à ces choix multi-critères.

De tels modèles de prédiction sont destinés à être intégrés dans un environnement de développement afin de servir comme système d'aide au développement. Ils peuvent aussi être utilisés comme des scanners pour une analyse a posteriori du code.

9.2 Perspectives

Lors de la réalisation des travaux de cette thèse, nous avons rencontré des besoins dont la réalisation était nécessaire. Nous les avons résolu pour atteindre les objectifs de cette thèse, néanmoins, nous pensons qu'ils peuvent être traités de manière plus générique et approfondie pour devenir plus réutilisables. Ceux-ci correspondent à la première catégorie des perspectives présentées ci-dessous. La deuxième catégorie correspondent aux idées que nous avions eu, durant ce travail de thèse, mais dont le travail nécessaire pour les développer demanderait une autre thèse à part entière.

9.2.1 Cartographie des vulnérabilités

De nombreuses vulnérabilités sont découvertes tous les jours, ce qui entraîne l'émergence de nouvelles catégories de vulnérabilités. La classification des vulnérabilités permet de mieux comprendre la nature des vulnérabilités et aide à proposer des solutions efficaces. Cependant, catégoriser les vulnérabilités manuellement est impraticable car le processus est subjectif et coûteux en temps de traitement.

Bien que la catégorisation CWE présente de nombreux avantages, cette catégorisation

a plusieurs failles qui sont principalement liées au facteur humain. En effet, la catégorisation CWE a été réalisée manuellement par des développeurs et maintenue ainsi au fil du temps. Les personnes qui catégorisent les vulnérabilités n'ont pas toujours une vision complète de toute la cartographie déjà présente pour pouvoir intégrer de nouvelles vulnérabilités. Ce processus est subjectif et sujet à des erreurs car les développeurs se réfèrent à leur propre expérience professionnelle [123].

Pour palier à ce problème, l'idée est de construire une nouvelle cartographie de vulnérabilités dont les relations reflètent la nature de la vulnérabilité. Pour ce faire, nous pouvons utiliser des méthodes d'apprentissage automatique en nous basant sur des caractéristiques textuelles. Celles-ci peuvent correspondre aux descriptions des vulnérabilités proposées par la CWE et celle listées par CVE. Une telle cartographie peut évoluer en intégrant les nouvelles vulnérabilités découvertes automatiquement.

9.2.2 Classification multi-classes

L'approche de construction de modèles de prédiction présentée dans le Chapitre 8 permet de prédire le code vulnérable efficacement mais sans préciser la catégorie de vulnérabilités. Nous avons entamé une première étude pour une classification multi-classes afin de pouvoir donner plus de précision sur la vulnérabilité à analyser et/ou corriger. Dans ce sens, nous voulons, encore une fois, mettre l'accent sur les catégories de vulnérabilités connues et non connues. Pour ce faire, nous maintenons le modèle de prédiction binaire et nous le complétons avec l'intégration de nouveaux modèles de prédiction qui précisent le type de vulnérabilités. L'objectif d'une telle étude est de retrouver des individus classifiés comme étant vulnérables par le premier modèle de prédiction, mais ne contenant aucune catégorie de vulnérabilités dans la classification multi-classes. Ces individus peuvent être mal classés par un des deux modèles (ou les deux). Cependant, si les classifications sont correctes, nous pouvons confirmer que le modèle qui prédit le code vulnérable a réussi à étendre sa zone de couverture en détectant de nouvelles catégories de vulnérabilités. Bien évidemment, l'étude est réalisée sur un ensemble réduit de vulnérabilités mais l'idée est d'apprendre à partir d'un ensemble de n vulnérabilités et de l'étendre à un ensemble de taille supérieure à n. L'objectif à terme est non seulement de corriger des failles inconnues mais aussi de mettre en place un mécanisme d'augmentation de la connaissance collective, une des clés de réussite des défenseurs sur les attaquants.

Une première implémentation a été réalisée avec un modèle multi-classes qui prédit les 5 catégories de vulnérabilités les plus récurrentes dans SecureQualitas. Un tel modèle a

été facile à mettre en place et ses performances globales ont atteint 67 % de f-mesure pour certains modèles. Cependant, d'un classifieur à un autre, il est difficile de faire une comparaison globale des performances. En effet, aucun modèle ne dépasse tous les autres pour toutes les classes à détecter. La différence est d'autant plus marquée par le déséquilibre important entre le nombre d'individus par catégorie de vulnérabilités. Pour atténuer cette différence, il aurait été nécessaire d'équilibrer la base en réduisant le nombre d'individus de certaines vulnérabilités, ce qui est problématique vu le nombre déjà limité d'individus vulnérables. De plus, lors de l'utilisation d'un modèle de prédiction multi-classes, un individu est systématiquement classifié dans une des catégories de vulnérabilités alors que le code peut contenir de nouvelles catégories. Pour palier à ce problème, l'idée est de construire un modèle de prédiction par catégorie de vulnérabilités, chaque modèle faisant une classification binaire pour détecter l'existence d'une catégorie de vulnérabilités. Nos premières expérimentations donnent de bonnes performances qui dépassent les 83 % pour la précision et le rappel. Cependant, ces expérimentations doivent être étendues en enrichissant la base et en testant de nouvelles méthodes de classification.

9.2.3 Développer des règles de bonnes pratiques

La détection des vulnérabilités est un domaine qui suscite l'intérêt de beaucoup de chercheurs en raison des conséquences que certaines failles peuvent avoir sur la confidentialité, l'intégrité et la disponibilité des logiciels. Détecter les vulnérabilités en utilisant des méthodes d'apprentissage automatique ne permet pas toujours d'avoir une transparence totale sur la méthode utilisée pour déduire la classe en sortie. Par exemple, les classifieurs construits avec des réseaux de neurones ajustent les poids du réseau afin de minimiser une certaine fonction de coût entre la sortie donnée par le réseau et la classe attendue. Cependant, il est très difficile de comprendre comment et encore moins pourquoi le réseau renvoie une telle sortie.

Un des axes de recherche que nous aimerions explorer consisterait à exploiter la masse importante de vulnérabilités et d'instances de chaque vulnérabilité pour apprendre à déduire des règles de bonnes pratiques. L'utilisation des métriques logicielles de qualité et des méthodes de classification telles que les arbres de décision ou les forêts aléatoires représente un bon point de départ à une telle approche. Ainsi, nous pourrions enrichir les bases de règles de codage telles que CERT Coding Standards [41] avec de nouvelles règles qui émergent de l'ensemble des vulnérabilités identifiées.

Un tel système pourrait être utilisé en complément à un système de détection de vulnérabilités. Le premier viserait à éviter les problème à la source, alors que le second servirait comme outil de vérification.

9.2.4 Correction automatique de vulnérabilités

Les vulnérabilités les plus dangereuses sont celles qui sont facilement exploitables pour atteindre une ressource critique. Les identifier, rapidement et efficacement, est un des moyens les plus sûrs de réduire le risque d'attaques. Cependant, il faut trouver la bonne correction de la vulnérabilité, en particulier lorsque les développeurs ont des connaissances limitées en sécurité. Il serait intéressant de pouvoir proposer à ces développeurs des correctifs générés automatiquement afin de les orienter sur les modifications potentielles à adopter. Ceci motive le recours à des méthodes de réparation automatique de programmes vulnérables.

La réparation automatique des programmes est une nouvelle approche de rectification particulièrement utilisée pour corriger des défauts logiciels ou des erreurs de programmation. Parmi les solutions proposées dans ce domaine, nous retrouvons celles basées sur la génération automatique de correctifs. L'objectif est de trouver une modification (minimale) d'un programme défectueux P pour produire un programme P' débarrassé du défaut. L'idée de base est d'abstraire, à partir de plusieurs exemples de changements de code, un modèle qui peut être appliqué à un code défectueux pour corriger la faille. Cette méthode produit des résultats prometteurs mais nécessite un nombre important de correctifs pour pouvoir générer des modèles de correction. Cependant, la collecte d'un ensemble fiable de corrections de défauts peut être difficile.

Les vulnérabilités sont parmi les défauts logiciels les plus critiques qui existent. Nous pouvons donc nous inspirer de cette méthode de correction de défauts pour corriger des vulnérabilités logicielles. Nous avons fait quelques expérimentations pour collecter des correctifs de vulnérabilités sur la base de projets GitHub. L'idée est de constituer une base de connaissance, conséquente et consistante, afin d'être capable d'utiliser des méthodes d'apprentissage pour générer des correctifs automatiquement.

BIBLIOGRAPHIE

[1] The Mitre CORPORATION. *Common Vulnerabilities and Exposures*. https://cve.mitre.org/. Accessed : 2020-08-22. 2020.

[2] MITRE. *Mitre*. https://www.mitre.org/. Accessed : 2020-08-26. 2019.

[3] Peter MELL, Karen SCARFONE et Sasha ROMANOSKY. « Common vulnerability scoring system ». In : *IEEE Security & Privacy* 4.6 (2006), p. 85-89.

[4] Gary MCGRAW. « Automated code review tools for security ». In : *Computer* 41.12 (2008), p. 108-111.

[5] Richard R. LINDE. « Operating system penetration ». In : *American Federation of Information Processing Societies National*. AFIPS Press, 1975, p. 361-368.

[6] *Explostion du vaisseau spatial Ariane*. http://www.around.com/ariane.html. Accessed : 2020-08-30. 2020.

[7] Ralph LANGNER. « Stuxnet : Dissecting a cyberwarfare weapon ». In : *IEEE Security & Privacy* 9.3 (2011), p. 49-51.

[8] *Toyota recall costs : $2 billion*. http://money.cnn.com/2010/02/04/news/companies/toyota_earnings.cnnw/index.htm. Accessed : 2020-08-12. 2020.

[9] *Software Security Initiatives*. https://www.owasp.org/images/f/f2/Education_Module_Embed_within_SDLC.ppt. Accessed : 2020-08-10. 2019.

[10] Chris F KEMERER. « Software complexity and software maintenance : A survey of empirical research ». In : *Annals of Software Engineering* 1.1 (1995), p. 1-22.

[11] Philippe ARTEAU. *Find Security Bugs*. https://find-sec-bugs.github.io. Accessed : 2020-08-30. 2019.

[12] Micro FOCUS. *Fortify Static Code Analyzer*. https://www.microfocus.com/fr-fr/products/static-code-analysis-sast/overview. Accessed : 2020-08-30. 2019.

[13] Patrice GODEFROID. « Random testing for security : blackbox vs. whitebox fuzzing ». In : *International workshop on Random testing*. ACM, 2007, p. 1-11.

[14] Patrice Godefroid, Michael Y. Levin et David A. Molnar. « Whitebox fuzzing for security testing ». In : *Network and Distributed System Security Symposium*. The Internet Society, 2008, p. 20.

[15] Davide Balzarotti et al. « Saner : Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications ». In : *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2008, p. 387-401.

[16] Zhendong Su et Gary Wassermann. « The essence of command injection attacks in web applications ». In : *ACM Sigplan Notices* 41.1 (2006), p. 372-382.

[17] William G. J. Halfond et Alessandro Orso. « AMNESIA : analysis and monitoring for NEutralizing SQL-injection attacks ». In : *International Conference on Automated Software Engineering*. ACM, 2005, p. 174-183.

[18] Stephan Neuhaus et al. « Predicting vulnerable software components ». In : *Conference on Computer and Communications Security*. ACM, 2007, p. 529-540.

[19] Viet Hung Nguyen et Le Minh Sang Tran. « Predicting vulnerable software components with dependency graphs ». In : *International Workshop on Security Measurements and Metrics*. ACM, 2010, p. 1-8.

[20] Yonghee Shin et Laurie Williams. « An empirical model to predict security vulnerabilities using code complexity metrics ». In : *International symposium on Empirical Software Engineering and Measurement*. ACM, 2008, p. 315-317.

[21] Thomas Zimmermann, Nachiappan Nagappan et Laurie Williams. « Searching for a needle in a haystack : Predicting security vulnerabilities for windows vista ». In : *Third International Conference on Software Testing, Verification and Validation*. IEEE. 2010, p. 421-428.

[22] Omar H Alhazmi, Yashwant K Malaiya et Indrajit Ray. « Measuring, analyzing and predicting security vulnerabilities in software systems ». In : *computers & security* 26.3 (2007), p. 219-228.

[23] Mehran Bozorgi et al. « Beyond heuristics : learning to classify vulnerabilities and predict exploits ». In : *International Conference on Knowledge Discovery and Data Mining*. ACM, 2010, p. 105-114.

[24] Michael Howard et David LeBlanc. *Writing Secure Code for Windows Vista™*. Microsoft Press, 2007.

[25] José Fonseca et Marco Vieira. « Mapping software faults with web security vulnerabilities ». In : *International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2008, p. 257-266.

[26] Henrique Alves, Baldoino Fonseca et Nuno Antunes. « Software Metrics and Security Vulnerabilities : Dataset and Exploratory Study ». In : *European Dependable Computing Conference*. IEEE Computer Society, 2016, p. 37-44.

[27] Riccardo Scandariato et al. « Predicting vulnerable software components via text mining ». In : *IEEE Transactions on Software Engineering* 40.10 (2014), p. 993-1006.

[28] V. Benjamin Livshits et Monica S. Lam. « Finding Security Vulnerabilities in Java Applications with Static Analysis ». In : *USENIX Security Symposium*. USENIX Association, 2005, p. 18.

[29] Dan DaCosta et al. « Characterizing the 'Security Vulnerability Likelihood' of Software Functions ». In : *International Conference on Software Maintenance*. IEEE Computer Society, 2003, p. 266.

[30] Fabian Yamaguchi, Felix Lindner et Konrad Rieck. « Vulnerability extrapolation : Assisted discovery of vulnerabilities using machine learning ». In : *Conference on Offensive technologies*. USENIX Association, 2011, p. 13-13.

[31] Henry Jiang (CISSP). *The world of cybersecurity map version 2.0*. `https://www.linkedin.com/pulse/map-cybersecurity-domains-version-20-henry-jiang-ciso-cissp/`. Accessed : 2020-09-01. 2020.

[32] NIST. *National Vulnerability Database*. `https://nvd.nist.gov/`. Accessed : 2020-08-10.

[33] NIST. *National Institute of Standards and Technology*. `https://www.nist.gov/`. Accessed : 2020-09-20. 2020.

[34] NIST. *The Common Vulnerability Scoring System*. `https://nvd.nist.gov/vuln-metrics/cvss`. Accessed : 2020-08-29. 2020.

[35] Mitre. *Common Weakness Enumeration*. `https://cwe.mitre.org/`. Accessed : 2020-08-20. 2019.

[36] Cybersecurity et Infrastructure Security Agency. *OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160)*. `https://us-cert.cisa.gov/ncas/alerts/TA14-098A`. Accessed : 2020-08-11. 2020.

[37] John Viega, M Howard et D LeBlanc. *Deadly Sins of Software Security : Programming Flaws and How to Fix Them*. Accessed : 2020-08-10. 2009.

[38] C Arthur. « Apple's SSL iPhone vulnerability : how did it happen, and what next ». In : *The Guardian* (2014).

[39] Gary McGraw. « Software security ». In : *IEEE Security & Privacy* 2.2 (2004), p. 80-83.

[40] Marisa R Randazzo et al. *Insider threat study : Illicit cyber activity in the banking and finance sector*. Rapp. tech. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2005.

[41] Carnegie Mellon University. *CERT Coding Standards*. `https://www.securecoding.cert.org`. Accessed : 2020-08-14. 2020.

[42] Brad Arkin, Scott Stender et Gary McGraw. « Software penetration testing ». In : *IEEE Security & Privacy* 3.1 (2005), p. 84-87.

[43] Matt Bishop. « About penetration testing ». In : *IEEE Security & Privacy* 5.6 (2007), p. 84-87.

[44] OWASP. *Source Code Analysis Tools*. `https://owasp.org/www-community/Source_Code_Analysis_Tools`. Accessed : 2020-08-21. 2020.

[45] David Evans et David Larochelle. « Improving security using extensible lightweight static analysis ». In : *IEEE software* 19.1 (2002), p. 42-51.

[46] James R Larus et al. « Righting software ». In : *IEEE software* 21.3 (2004), p. 92-100.

[47] Nathaniel Ayewah et al. « Using static analysis to find bugs ». In : *IEEE software* 25.5 (2008), p. 22-29.

[48] Al Bessey et al. « A few billion lines of code later : using static analysis to find bugs in the real world ». In : *Communications of the ACM* 53.2 (2010), p. 66-75.

[49] Fabian Yamaguchi, Markus Lottmann et Konrad Rieck. « Generalized vulnerability extrapolation using abstract syntax trees ». In : *Computer Security Applications Conference*. ACM, 2012, p. 359-368.

[50] Yao-Wen Huang et al. « Web application security assessment by fault injection and behavior monitoring ». In : *International World Wide Web Conference*. ACM, 2003, p. 148-159.

[51] Stefan Kals et al. « SecuBat : a web vulnerability scanner ». In : *International conference on World Wide Web*. ACM, 2006, p. 247-256.

[52] Nuno Antunes et Marco Vieira. « Designing vulnerability testing tools for web services : approach, components, and tools ». In : *International Journal of Information Security* 16.4 (2017), p. 435-457.

[53] Nuno Antunes et Marco Vieira. « Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services ». In : *Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society, 2009, p. 301-306.

[54] José Fonseca, Marco Vieira et Henrique Madeira. « Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks ». In : *Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society, 2007, p. 365-372.

[55] Joao A Duraes et Henrique S Madeira. « Emulation of software faults : A field data study and a practical approach ». In : *IEEE transactions on software engineering* 32.11 (2006), p. 849-867.

[56] Andrew Austin et Laurie A. Williams. « One Technique is Not Enough : A Comparison of Vulnerability Discovery Techniques ». In : *International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2011, p. 97-106.

[57] Venkatesh-Prasad Ranganath et Joydeep Mitra. « Are free Android app security analysis tools effective in detecting known vulnerabilities ? » In : *Empirical Software Engineering* 25.1 (2020), p. 178-219.

[58] Nick Rutar, Christian B. Almazan et Jeffrey S. Foster. « A Comparison of Bug Finding Tools for Java ». In : *International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2004, p. 245-256.

[59] Na Meng et al. « An Approach to Merge Results of Multiple Static Analysis Tools (Short Paper) ». In : *International Conference on Quality Software*. IEEE Computer Society, 2008, p. 169-174.

[60] Qianxiang Wang et al. « Towards SOA-Based Code Defect Analysis ». In : *International Symposium on Service-Oriented System Engineering*. IEEE Computer Society, 2008, p. 269-274.

[61] Paulo Jorge Costa Nunes et al. « On Combining Diverse Static Analysis Tools for Web Security : An Empirical Study ». In : *European Dependable Computing Conference*. IEEE Computer Society, 2017, p. 121-128.

[62] Areej Algaith et al. « Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools ». In : *European Dependable Computing Conference*. IEEE Computer Society, 2018, p. 57-64.

[63] José D'Abruzzo Pereira, João R. Campos et Marco Vieira. « An Exploratory Study on Machine Learning to Combine Security Vulnerability Alerts from Static Analysis Tools ». In : *Latin-American Symposium on Dependable Computing*. IEEE, 2019, p. 1-10.

[64] Thiago S Guzella et Walmir M Caminhas. « A review of machine learning approaches to spam filtering ». In : *Expert Systems with Applications* 36.7 (2009), p. 10206-10222.

[65] Godwin Caruana et Maozhen Li. « A survey of emerging approaches to spam filtering ». In : *ACM Computing Surveys* 44.2 (2008), p. 1-27.

[66] Pedro Garcia-Teodoro et al. « Anomaly-based network intrusion detection : Techniques, systems and challenges ». In : *computers & security* 28.1-2 (2009), p. 18-28.

[67] Chenfeng Vincent Zhou, Christopher Leckie et Shanika Karunasekera. « A survey of coordinated attacks and collaborative intrusion detection ». In : *Computers & Security* 29.1 (2010), p. 124-140.

[68] Arthur L Samuel. « Some studies in machine learning using the game of checkers ». In : *IBM Journal of research and development* 3.3 (1959), p. 210-229.

[69] Jiawei Han, Jian Pei et Micheline Kamber. *Data mining : concepts and techniques*. Elsevier, 2011.

[70] Seyed Mohammad Ghaffarian et Hamid Reza Shahriari. « Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques : A Survey ». In : *ACM Computing Surveys* 50.4 (2017).

[71] Varun Chandola, Arindam Banerjee et Vipin Kumar. « Anomaly detection : A survey ». In : *ACM computing surveys* 41.3 (2009), p. 1-58.

[72] Matthieu JIMENEZ. « Evaluating Vulnerability Prediction Models ». Thèse de doct. UNIVERSITÉ DU LUXEMBOURG, 2018.

[73] Raounak Benabidallah et al. « Designing a Code Vulnerability Meta-scanner ». In : *International Conference on Information Security Practice and Experience*. Springer. 2019, p. 194-210.

[74] Patrick Morrison et al. « Challenges with applying vulnerability prediction models ». In : *Symposium and Bootcamp on the Science of Security - HotSoS*. ACM, 2015, p. 1-9.

[75] Stephan Neuhaus et Thomas Zimmermann. « The Beauty and the Beast : Vulnerabilities in Red Hat's Packages ». In : *USENIX Annual Technical Conference*. USENIX Association, 2009, 527—538.

[76] Yonghee Shin et Laurie Williams. « Is complexity really the enemy of software security ? » In : *ACM workshop on Quality of protection*. ACM, 2008, p. 47-50.

[77] Nachiappan Nagappan, Thomas Ball et Andreas Zeller. « Mining metrics to predict component failures ». In : *International Conference on Software Engineering*. ACM, 2006, p. 452-461.

[78] Yonghee Shin et Laurie Williams. « An initial study on the use of execution complexity metrics as indicators of software vulnerabilities ». In : *International Workshop on Software Engineering for Secure Systems*. ACM, 2011, p. 1-7.

[79] Taghi M Khoshgoftaar, Ruqun Shan et Edward B Allen. « Using product, process, and execution metrics to predict fault-prone software modules with classification trees ». In : *International Symposium on High Assurance Systems Engineering (HASE 2000)*. IEEE Computer Society, 2000, p. 301-310.

[80] Yonghee Shin et al. « Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities ». In : *IEEE transactions on software engineering* 37.6 (2010), p. 772-787.

[81] Andrew Meneely et al. « Predicting failures with developer networks and social network analysis ». In : *International Symposium on Foundations of software engineering*. ACM, 2008, p. 13-23.

[82] Yonghee Shin et Laurie Williams. « Can traditional fault prediction models be used for vulnerability prediction ? » In : *Empirical Software Engineering* 18.1 (2013), p. 25-59.

[83] Istehad Chowdhury et Mohammad Zulkernine. « Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities ? » In : *ACM Symposium on Applied Computing*. ACM, 2010, p. 1963-1969.

[84] Istehad Chowdhury et Mohammad Zulkernine. « Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities ». In : *Journal of Systems Architecture* 57.3 (2011), p. 294-313.

[85] Sara Moshtari, Ashkan Sami et Mahdi Azimi. « Using complexity metrics to improve software security ». In : *Computer Fraud & Security* 2013.5 (2013), p. 8-17.

[86] Sara Moshtari et Ashkan Sami. « Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction ». In : *ACM Symposium on Applied Computing*. ACM, 2016, p. 1415-1421.

[87] Henrique Alves, Baldoino Fonseca et Nuno Antunes. « Experimenting Machine Learning Techniques to Predict Vulnerabilities ». In : *Latin-American Symposium on Dependable Computing*. IEEE Computer Society, 2016, p. 151-156.

[88] Xiaoning Du et al. « Leopard : identifying vulnerable code for vulnerability assessment through program metrics ». In : *International Conference on Software Engineering*. IEEE / ACM, 2019, p. 60-71.

[89] Hideaki Hata, Osamu Mizuno et Tohru Kikuno. « Fault-prone module detection using large-scale text features based on spam filtering ». In : *Empirical Software Engineering* 15.2 (2010), p. 147-165.

[90] Aram Hovsepyan et al. « Software vulnerability prediction using text analysis techniques ». In : *International workshop on Security measurements and metrics*. ACM, 2012, p. 7-10.

[91] James Walden et Maureen Doyle. « SAVI : Static-analysis vulnerability indicator ». In : *IEEE Security & Privacy* 10.3 (2012), p. 32-39.

[92] Yulei Pang, Xiaozhen Xue et Akbar Siami Namin. « Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection ». In : *International Conference on Machine Learning and Applications*. IEEE, 2015, p. 543-548.

[93] Yulei Pang, Xiaozhen Xue et Huaying Wang. « Predicting vulnerable software components through deep neural network ». In : *International Conference on Deep Learning Technologies*. ACM, 2017, p. 6-10.

[94] James Walden, Jeff Stuckman et Riccardo Scandariato. « Predicting vulnerable components : Software metrics vs text mining ». In : *International symposium on software reliability engineering*. IEEE Computer Society, 2014, p. 23-33.

[95] Andrew Meneely et Laurie Williams. « Strengthening the empirical analysis of the relationship between Linus' Law and software security ». In : *International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010, p. 1-10.

[96] Amiangshu Bosu et al. « Identifying the characteristics of vulnerable code changes : An empirical study ». In : *International Symposium on Foundations of Software Engineering*. ACM, 2014, p. 257-268.

[97] Henning Perl et al. « Vccfinder : Finding potential vulnerabilities in open-source projects to assist code audits ». In : *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, p. 426-437.

[98] Yun Zhang et al. « Combining software metrics and text features for vulnerable file prediction ». In : *International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 2015, p. 40-49.

[99] Nawa Raj Pokhrel, Hansapani Rodrigo, Chris P Tsokos et al. « Cybersecurity : Time series predictive modeling of vulnerabilities of desktop operating system using linear and non-linear approach ». In : *Journal of Information Security* 8.04 (2017), p. 362.

[100] Su Zhang, Doina Caragea et Xinming Ou. « An Empirical Study on Using the National Vulnerability Database to Predict Software Vulnerabilities ». In : *Database and Expert Systems Applications*.

[101] Awad A. Younis et al. « To Fear or Not to Fear That is the Question : Code Characteristics of a Vulnerable Functionwith an Existing Exploit ». In : *Conference on Data and Application Security and Privacy*. ACM, 2016, p. 97-104.

[102] Pedro Domingos. « A few useful things to know about machine learning ». In : *Communications of the ACM* 55.10 (2012), p. 78-87.

[103] Tim Boland et Paul E Black. « Juliet 1.1 C/C++ and Java test suite ». In : *Computer* 45.10 (2012), p. 88-90.

[104] YAGAAN Software Security. *Yag Suite*. https://www.yagaan.com/products.html. Accessed : 2020-09-01. 2017.

[105] Philippe Arteau. *Bugs Patterns*. https://find-sec-bugs.github.io/bugs.htm. Accessed : 2020-08-30. 2020.

[106] Micro Focus. *Fortify Static Code Analyzer (SCA) Static Application Security Testing*. https://www.microfocus.com/media/data-sheet/fortify_static_code_analyzer_static_application_security_testing_ds.pdf. Accessed : 2020-08-23. 2020.

[107] NSA. *Juliet Test Suite v1.2 for Java*. https://samate.nist.gov. Accessed : 2020-08-27. 2012.

[108] Hein S Venter, Jan HP Eloff et YL Li. « Standardising vulnerability categories ». In : *Computers & Security* 27.3-4 (2008), p. 71-83.

[109] Sebastian Blank, Tobias Föhst et Karsten Berns. « A fuzzy approach to low level sensor fusion with limited system knowledge ». In : *International Conference on Information Fusion*. IEEE, 2010, p. 1-7.

[110] Nir Friedman, Dan Geiger et Moises Goldszmidt. « Bayesian network classifiers ». In : *Machine learning* 29.2-3 (1997), p. 131-163.

[111] R. Benabidallah, S. Sadou et I. Borne. « SecureQualitas : A Security Corpus of Real Java Applications ». In : *International Conference on Cyber Security for Emerging Technologies*. IEEE, 2019, p. 1-6.

[112] Benjamin Livshits. *Stanford SecuriBench*. https://suif.stanford.edu/~livshits/securibench/. Accessed : 2020-08-19. 2005.

[113] OWASP. *OWASP WebGoat Project*. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project. Accessed : 2020-09-02. 2018.

[114] Antonios Gkortzis, Dimitris Mitropoulos et Diomidis Spinellis. « VulinOSS : a dataset of security vulnerabilities in open-source systems ». In : *International Conference on Mining Software Repositories*. ACM, 2018, p. 18-21.

[115] Ewan D. Tempero et al. « The Qualitas Corpus : A Curated Collection of Java Code for Empirical Studies ». In : *Asia Pacific Software Engineering Conference*. IEEE Computer Society, 2010, p. 336-345.

[116] Ricardo Terra et al. « Qualitas.class Corpus : A Compiled Version of the Qualitas Corpus ». In : *Software Engineering Notes* 38.5 (2013), p. 1-4.

[117] David W Corne, Joshua D Knowles et Martin J Oates. « The Pareto envelope-based selection algorithm for multiobjective optimization ». In : *International conference on parallel problem solving from nature*. Springer Berlin Heidelberg, 2000, p. 839-848.

[118] Leanid Krautsevich, Fabio Martinelli et Artsiom Yautsiukhin. « Formal approach to security metrics : what does "more secure" mean for you ? » In : *European Conference on Software Architecture*. ACM, 2010, p. 162-169.

[119] Ju An Wang et al. « Security metrics for software systems ». In : *ACM Southeast Regional Conference*. ACM, 2009, p. 1-2.

[120] Virtual Machinery. *JHawk Product*. `http://www.virtualmachinery.com/jhawkprod.htm`. Accessed : 2020-09-20. 2020.

[121] Virtual Machinery. *JHawk metrics at method level*. `http://www.virtualmachinery.com/jhawkmetricsmethod.htm`. Accessed : 2020-08-29. 2020.

[122] Matthieu Jimenez, Mike Papadakis et Yves Le Traon. « Vulnerability prediction models : A case study on the linux kernel ». In : *International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2016, p. 1-10.

[123] Tao Wen et al. « ASVC : an automatic security vulnerability categorization framework based on novel features of vulnerability data ». In : *Journal of Communications* 10.2 (2015), p. 107-116.

Appendices

Annexe A

Caractéristiques du corpus *SecureQualitas*

Projet	Nb méthodes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Total
Cobertura-1.9.4.1	1462	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	20
colt-1.2.0	2755	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	4
displaytag-1.2	1404	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
exoportal-v1.0.2	9773	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	2
findbugs-1.3.9	8210	47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	47
freecs-1.3.20100406	1318	0	0	0	0	0	0	0	5	0	0	0	0	0	11	0	0	16
galleon-2.3.0	3226	0	0	0	0	0	0	0	2	0	0	0	0	2	0	0	0	4
ganttproject-2.1.1	4157	9	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	10
htmlunit-2.8	7832	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	4
informa-0.7.0-alpha2	1221	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2
ivatagroupware-0.11.3	1315	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
jag-6.1	1144	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15
jasml-0.10	222	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
jasperreports-3.7.4	13569	377	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	378
javacc-5.0	745	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
jgraph-5.13.0.0	2333	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	3
jgraphpad-5.10.0.2	1484	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	8
jgrapht-0.8.1	855	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
jgroups-2.10.0	7066	24	1	0	0	0	0	0	7	3	0	0	0	0	69	1	0	105
jmeter-2.5.1	7648	45	0	0	0	2	0	0	4	18	0	0	0	3	6	0	0	78
joggplayer-1.1.4	1723	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	17
log4j-2.0-beta	2296	4	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	8
marauroa-3.8.1	1379	4	0	0	0	0	0	0	1	0	0	0	0	0	10	0	0	15

maven-3.0.5	2684	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
myfaces_core-2.1.10	6568	0	0	0	0	0	0	0	2	0	0	0	1	0	0	0	1	4
nekohtml-1.9.14	413	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
oscache-2.3	598	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
pmd-4.2.5	3943	16	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	17
poi-3.6	15900	36	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0	43
proguard-4.9	3998	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
quartz-1.8.3	1906	1	1	0	0	0	0	0	1	0	0	0	0	0	3	19	0	25
quilt-0.6-a-5	701	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rssowl-2.0.5	5923	7	3	0	0	0	0	0	2	0	0	0	0	1	1	0	0	14
sablecc-3.2	1206	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sandmark-3.4	5412	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0	7
springframework-3.0.5	23222	4	0	0	2	0	0	0	2	34	1	0	1	7	0	0	1	52
sunflow-0.07.2	1155	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
trove-2.1.0	398	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	2
wct-1.5.2	5127	3	0	0	0	0	0	0	0	0	0	0	0	1	1	4	0	9
webmail-0.7.10	870	7	0	0	0	0	0	0	2	0	0	0	0	0	1	0	0	10
xerces-2.10.0	7373	4	0	0	0	0	0	0	1	3	0	0	0	0	0	0	0	8
Total	170534	672	7	3	6	7	6	7	71	68	11	11	15	28	122	40	18	956

TABLE A.1: Type et nombre de vulnérabilités par projet

Annexe B

Métriques fournies par l'outil JHawk au niveau méthode

Métrique	Description
CAST	Le nombre de casts de classes dans la méthode
COMP	Complexité cyclomatique : elle est calculée à partir du nombre de points d'embranchements logiques dans la méthode. La méthode elle-même est comptée comme 1 point de branchement logique. Les opérateurs ?, if, switch, for,while et catch comptent comme des points de branchement logique. Une complexité cyclomatique supérieure à 10 est considérée comme un indicateur de mauvais code. Dans le fichier jhawk.properties initial, le niveau "Avertissement" est fixé à 10 et le niveau "Danger" à 15.
CREF	Nombre de classes différentes référencées dans la méthode. Cela inclura les classes référencées dans les déclarations de variables, les types d'arguments, les casts, exceptions levées et prises, des instanciations de variables et des références directes aux méthodes de classe et aux variables. La valeur CREF comprend à la fois des références de classe et d'interface. Une méthode avec un nombre élevé de références de classe peut suggérer qu'une méthode en fait trop.
EXCR	Le nombre d'exceptions référencées par cette méthode. Cette figure n'inclue pas les exceptions référencées
EXCT	Nombre d'exceptions levées par cette méthode.
HBUG	Estimation des bugs de Halstead dans la méthode. On le calcule en divisant le volume de Halstead (HVOL) par 3000.
HDIF	La difficulté Halstead d'une méthode est un indicateur de la complexité de la méthode. Elle est calculée à partir du nombre d'opérateurs uniques (UOR), du nombre d'opérandes (NAND) et du nombre d'opérandes uniques (UAND) à l'aide de la formule : (UOR/2)* (NAND/UAND)

HEFF	L'effort de Halstead pour la méthode est un indicateur du temps qu'il faudra à un programmeur pour implémenter la méthode. Il est calculé à partir du volume de Halstead (HVOL) et de la difficulté de Halstead (HDIF) à l'aide de la formule HVOL*HDIF
HLTH	La longueur Halstead de la méthode est la somme du nombre d'opérateurs plus le nombre d'opérandes. C'est un indicateur de la taille de la méthode.
HVOC	Le Vocabulaire Halstead de la méthode. Il s'agit de la somme du nombre d'opérateurs uniques et du nombre d'opérandes uniques. C'est un indicateur de la complexité de la méthode.
HVOL	Le volume Halstead d'une méthode est un indicateur de la taille de la méthode. Il est calculé à partir du vocabulaire de Halstead (HVOC) et de la longueur de Halstead (HLTH) à l'aide de la formule HLTH * log2(HVOC)
LMET	Nombre d'appels à des méthodes locales, c'est-à-dire des méthodes qui sont définies dans la classe de la méthode. Un nombre élevé d'appels à des méthodes peut être un indicateur que la méthode en fait trop.
LOOP	Le nombre de cycles dans la méthode. Les cycles sont indiqués par les opérateurs "while" et "for".
MDN	Profondeur maximale d'imbrication. Il s'agit de la profondeur de la boucle la plus profonde d'une méthode. Une profondeur maximale d'imbrication de 4 ou plus est considérée comme un indicateur d'une méthode trop complexe, difficile à tester et qui doit être divisée en sous-méthodes. Le niveau "Avertissement" est fixé à 4 et le niveau "Danger" à 6.
MOD	Nombre de modificateurs (public, static, protected) dans la déclaration de la méthode.
NAME	Nom de la méthode
NAND	Nombre d'opérandes dans la méthode. Un opérande est un token Java sur lequel des opérations peuvent être effectuées par d'autres tokens Java. Les opérandes comprennent des variables, des littéraux numériques et des chaînes de caractères, des variables spéciales telles que true, false, null, void, super, et ce, des classes et des types primitifs et des méthodes.
NEXP	Le nombre d'expressions Java dans la méthode. Une expression Java est un fragment de code qui évalue une valeur seule e.g. var1 * var2 or " FIRST "+" STRING ".
NLOC	Nombre de lignes de code dans la méthode. Une ligne de code est toute ligne non blanche dans un fichier de code qui n'est pas un commentaire..
NOA	Nombre d'arguments dans la signature de la méthode. Un très grand nombre d'arguments peut suggérer qu'une méthode en fait trop.

NOC	Nombre de commentaires. Il s'agit du nombre de commentaires discrets dans le code. C'est-à-dire qu'un commentaire sur plusieurs lignes sera compté comme un seul commentaire.
NOCL	Nombre de lignes de commentaires. C'est le nombre de lignes de commentaires dans le code, y compris les tokens de début et de fin des commentaires si ceux-ci sont sur des lignes séparées. Une fin de ligne de commentaire sur une ligne qui comprend également du code sera comptée comme une ligne de commentaire.
NOPR	Nombre d'opérateurs dans la méthode. Un opérateur est un token Java qui est utilisé pour effectuer une opération sur un autre token Java. Les exemples d'opérateurs sont les opérateurs arithmétiques (+, -, *, /, etc) et les opérateurs logiques (et, ou , non, etc). Les mots-clés Java tels que for et while sont également des opérateurs.
NOS	Nombre d'instructions Java dans la méthode. Une instruction Java est définie comme une série de tokens Java terminés par un point-virgule.
TDN	La profondeur totale de l'imbrication. C'est un total de la profondeur de l'imbrication de tous les cycles dans cette méthode.
VDEC	Nombre de variables déclarées dans la méthode. Un très grand nombre de variables déclarées peut suggérer qu'une méthode en fait trop.
VREF	Nombre de références variables dans la méthode. Il s'agit du nombre total de références, c'est-à-dire que si var1 est mentionné 3 fois et var2 4 fois, la valeur de VREF est de 7. Un très grand nombre de références variables peut suggérer qu'une méthode en fait trop.
XMET	Nombre d'appels à des méthodes qui ne sont pas définies dans la classe de la méthode. Un nombre élevé d'appels externes augmentera la dépendance de la méthode (et donc de sa classe) par rapport aux autres classes, ce qui rendra plus difficile le maintien de la classe. Cela peut également être un indicateur que la méthode en fait trop.

TABLE B.1: Définition des métriques logicielles fournies par l'outil JHawk et utilisées comme caractéristiques d'apprentissage

Titre : Identification automatique des vulnérabilités de sécurité dans les systèmes logiciels

Mot clés : Analyse statique, identification de vulnérabilités, modèles de prédiction

Résumé : La menace posée par les vulnérabilités logicielles croît de manière exponentielle. Ce phénomène est dû, d'une part, à l'omniprésence des logiciels, et d'autre part, au nombre important de failles existantes. Pour faire face à ce problème, plusieurs stratégies ont été élaborées au fil du temps. Certaines visent à mettre en place de bonnes pratiques de développement et les intégrer dès la phase de conception tandis que d'autres consistent à effectuer des inspections de sécurité en indiquant les zones vulnérables. Cette thèse s'inscrit dans la deuxième catégorie de travaux et porte essentiellement sur la construction de modèles de prédiction de vulnérabilités. La création de ces derniers soulève différents problèmes. Le plus important étant le manque de données sur les vulnérabilités logicielles. À cet effet, nous mettons en place une chaîne de traitement complète allant de la création et l'annotation automatique d'un corpus de sécurité jusqu'à la construction et l'évaluation des modèles de prédiction de vulnérabilités. La première contribution de cette thèse est plus axée sur l'approche de construction de corpus que sur le corpus lui-même. L'approche est basée sur la conception de méta-scanners de vulnérabilités permettant d'identifier des vulnérabilités de code efficacement. Cela consiste à combiner plusieurs outils d'analyse statique en se basant sur leurs performances individuelles pour chaque catégorie de vulnérabilités. Notre deuxième contribution correspond au corpus SecureQualitas qui consiste en un corpus d'applications Java annotées avec les vulnérabilités qu'elles contiennent. Nous construisons ce corpus en utilisant un méta-scanner construit à l'aide de trois outils d'analyse de vulnérabilités. Enfin, notre troisième contribution est de construire un modèle de prédiction du code vulnérable. Nous avons opté pour l'utilisation de métriques de qualité pour caractériser le code et nous avons étudié les performances des modèles à la fois sur des catégories de vulnérabilités apprises par les modèles et sur des catégories non encore connues par le modèle. Les résultats de nos expérimentations ont montré l'efficacité des modèles sur les deux populations de vulnérabilités : connues et non connues.

Title: Automatic identification of security vulnerabilities in software systems

Keywords: Static analysis, vulnerability identification, prediction models

Abstract: The threat caused by software vulnerabilities is growing exponentially. This phenomenon is due, on the one hand, to the omnipresence of software, and on the other hand, to the large number of existing vulnerabilities. To deal with this problem, several strategies have been developed over time. Some aim to establish good development practices and integrate them right from the design phase, while others consist of carrying out security inspections by identifying vulnerable areas. This thesis is related to the second category of work and focuses on the construction of vulnerability prediction models.The creation of the latter raises various problems. The most important one is the lack of data on software vulnerabilities.For this purpose, we are setting up a complete processing chain from the creation and annotation of a security corpus to the construction and evaluation of vulnerability prediction models. The first contribution of this thesis focuses more on the corpus construction approach than on the corpus itself. The approach is based on the design of vulnerability meta-scanners allowing to identify code vulnerabilities efficiently.This consists in combining several static analysis tools based on their individual performance for each category of vulnerabilities. Our second contribution corresponds to the SecureQualitas corpus which consists of a corpus of Java applications annotated with the vulnerabilities they contain. We build this corpus using a meta-scanner built with three vulnerability analysis tools. Finally, our third contribution is to build a prediction model of vulnerable code. We opted and studied the use of quality metrics to characterize code and we have studied the performance of the models both on categories of vulnerabilities learned by the models and on categories not yet known by the model. The results of our experiments showed the efficiency of the models on both populations of vulnerabilities: known and unknown.