Université de Montréal

Détection des Utilisations à Risque d'API : Approche Basée sur le Système Immunitaire

par

Maxime Gallais-Jimenez

Département d'informatique et recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

29 juin 2018

RÉSUMÉ

Les APIs sont des ingrédients essentiels pour développer des systèmes logiciels complexes. Cependant, elles sont difficiles à apprendre et à utiliser. Par conséquent, les développeurs peuvent les utiliser à mauvais escient, ce qui entraîne différents types de problèmes.

La mauvaise utilisation d'une interface de programmation peut entraîner des erreurs très difficiles à détecter et qui peuvent se révéler graves. Détecter ces erreurs d'utilisation n'est pas simple.

C'est pour cela que nous avons développé une approche basée sur le système immunitaire, un mécanisme qui permet de détecter à la fois des types d'anomalies connus et inconnus. Ici, les cellules de l'organisme seront les bonnes utilisations de l'API et les mauvaises utilisations seront les cellules étrangères de l'organisme. Le système immunitaire a la particularité d'être un système décentralisé qui fonctionne grâce à des détecteurs, les lymphocytes T, qui ont pour rôle de détecter les cellules étrangères. Avec APIMMUNE, nous allons donc générer des détecteurs pour être capables de détecter des utilisations à risque des APIs.

Notre approche a été évaluée sur deux ensembles de données et plus particulièrement MUBench. Les résultats montrent que notre approche complète les travaux précédemment réalisés dans ce domaine de recherche. Les détecteurs peuvent être générés à partir de code source, en abstrayant des utilisations possibles des APIs et en générant des formes déviantes de ces utilisations. De plus, pour permettre la détection, le code qui a servi à générer les détecteurs n'a pas besoin d'être révélé. Par ailleurs, les détecteurs peuvent être produits pour différentes versions de l'interface de programmation, ce qui apporte une vraie modularité dans la détection.

Mots clés : Erreur d'utilisation d'API, API, Interface de programmation, Système immunitaire artificiel, Génie logiciel

ABSTRACT

APIs are essential ingredients for developing complex software systems. However, they are difficult to learn and use. As a result, developers can misuse them, resulting in different types of issues.

Misuse of a programming interface can lead to errors that are very difficult to detect and can have consequences. Detecting these misuses is not easy.

We have thus developed an approach based on the immune system, a mechanism that allows to detect known and unknown anomaly types. Here the cells of the organism will be the good uses of the API and the bad ones will be the foreign cells of the organism. The immune system has the particularity of being a decentralized system that functions thanks to detectors, T lymphocytes, whose role is to detect foreign cells. With APIMMUNE we will therefore generate detectors that have the ability to detect risky uses of APIs.

Our approach was evaluated on two sets of data and more specifically MUBench. The results show that our approach complements previous works in this area of research. In addition, the detectors can be generated from source code by abstracting the API usages and by generating artificial deviations from these usages. Moreover, for the detection purpose, only the artificial detectors are necessary, and the code used to generate them is not disclosed. Finally, the detectors can be produced for different versions of the programming interface, that brings modularity in the detection.

Keywords : API Misuses, API, Programming Interface, Artificial Immune-System, Software Engineering

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

Le bon déroulement de ma maîtrise a été rendu possible grâce à l'aide et au soutien de plusieurs personnes. Je profite de cette occasion pour leur adresser un sincère merci.

Je tiens, dans un premier temps, à remercier mon directeur de recherche le Professeur Houari Sahraoui qui m'a accompagné du premier au dernier jour de mon cursus Montréalais. Je souhaite louer sa patience, son dévouement et sa gentillesse à mon égard et celui de tous les autres élèves.

Je veux aussi exprimer ma gratitude au Dr Hoan Nguyen qui m'a accompagné tout au long du projet ainsi qu'au Dr Tien Nguyen. Leur collaboration a été un exemple du genre.

Je souhaite remercier les membres du laboratoire de génie logiciel de l'Université de Montréal pour leurs conseils et leur bienveillance, et en particulier le Dr Aymen Saied qui m'a guidé, aidé et permis d'avancer tout au long de mes recherches.

Je voudrais finir par remercier mes proches pour leur soutien quotidien et particulièrement ma famille qui m'a permis de réaliser ce grand projet.

Chapitre 1

INTRODUCTION

1.1. Contexte

Nous vivons dans un monde de plus en plus connecté qui tend à s'automatiser et à être de plus en plus instantané. Avec l'avènement du téléphone intelligent, le déploiement du cloud et l'accessibilité aux objets connectés (**I**nternet **O**f **T**hings), la connectivité entre services est primordiale. Cisco estime qu'entre 2008 et 2009 le nombre d'appareils connectés a surpassé le nombre de personnes sur Terre et que l'écart ne cesse de se creuser [**20**]. Ceci signifie que la population utilise de plus en plus de services connectés et que les exigences sur le plan de la qualité et de l'innovation croissent. Ces phénomènes contraignent les services informatiques à avoir des cycles de développement plus agiles et rapides, mais surtout continus. Le cloud computing permet de répondre à cette demande d'évolutivité et d'efficacité, ainsi les entreprises peuvent utiliser des ressources logicielles et des infrastructures en toute flexibilité selon leurs exigences. L'institut Gartner prévoit des dépenses au-dessus des 300 milliards de dollars (US) dans le secteur du cloud [1] pour les deux années à venir. Les utilisateurs deviennent très exigeants avec les applications qu'ils utilisent. Ils désirent une expérience numérique instantanée et fluide. Le domaine du génie logiciel doit donc s'adapter à ces contraintes d'agilité et de réactivité [**13**]. Toute cette évolution est en partie portée par les géants du Web, les G.A.F.A.M. (Google, Amazon, Facebook, Apple et Microsoft) qui adoptent une stratégie d'attribution de l'innovation en acquérant les entreprises et start-up pouvant s'intégrer dans leurs gammes de produits comme de nouvelles fonctionnalités. Ces pratiques combinées aux exigences utilisateurs ont causé le retrait des logiciels monolithiques [**6**] pour un passage à des systèmes plus distribués avec le fameux principe de *"diviser pour régner"*. La variété de plateformes obligeant à composer des infrastructures modulaires, où les modules sont interconnectés. Cette manière de procéder évite des coûts de maintenance exorbitants sur le logiciel lorsque de nouvelles plateformes apparaissent. Des conteneurs ont aussi vu le jour pour lancer des applications sur tous les environnements possibles et tester

1. https://gartner.com/newsroom/id/3616417 (accédé le 11 avril 2018)

les différents composants [13]. Les applications existantes subissent donc des migrations et les nouvelles architectures doivent s'adapter.

C'est pour cela que de nos jours l'utilisation d'API (Application Programming Interface) est omniprésente. Car une API permet l'interaction entre deux logiciels sans l'intervention humaine [2]. L'interface relie les modules entre eux et facilite les échanges entre services. Elle permet que le code soit réutilisé afin de gagner en productivité. Les développeurs ne sont pas obligés de réinventer la roue pour des problématiques qui ont été résolues précédemment et peuvent se consacrer à la valeur ajoutée qu'ils peuvent apporter à leur projet [13].

L'utilisation d'API nous entoure. Peu importe le service que l'on utilise, au moins une API est utilisée. Elles peuvent être de formes variées. Par exemple, sur un site Web lorsque l'on voit un affichage Twitter, il y a l'utilisation de l'interface proposée par Twitter [3] pour avoir des données en temps réel. De même, nous pouvons par exemple voir une carte fournie par l'API Google Maps [4] qui transmet des données cartographiques.

Les APIs se popularisant, leur utilisation est soumise à plusieurs problématiques. Tout d'abord, il y a la courbe d'apprentissage. Est-ce fastidieux de manipuler cette API ? Car à défaut de réinventer la roue il faut savoir comment s'en servir. Et chaque API possède sa propre utilisation. Par exemple lorsque j'utilise une interface permettant la manipulation de ressources, je ne dois pas oublier de fermer cette ressource après manipulation au risque de la corrompre. La corruption étant une erreur d'utilisation de l'API, mais qui n'entraîne pas de problème immédiatement visible. C'est un effet de bord qui est difficilement détectable. De plus, chaque développeur possède ses techniques d'apprentissage, soit grâce à la documentation ou par l'exemple. Cette diversité entraîne donc des erreurs d'utilisation et par conséquent des comportements à risque. C'est pour cela que nous proposons à travers ce mémoire une approche pouvant répondre aux problématiques que nous allons évoquer ci-dessous.

1.2. Problématique

La popularité des APIs étant établie, la question de leur création et de leur utilisation est un sujet de recherche en effervescence. Elles sont utilisées dans un but de productivité, car elles simplifient la production de code. Le problème est que produire une API est très accessible, mais la rendre utilisable est une tâche beaucoup plus complexe. Utiliser une API peut rapidement se révéler compliqué et entraîner des erreurs qui peuvent être très nuisibles à la production finale du développeur.

Différents types de travaux ont été réalisés sur les APIs et plus particulièrement sur la détection d'erreur d'utilisation [37, 18, 34]. Il y a d'une part les travaux pour améliorer

2. https://medium.com/@mercier_remi/c-est-quoi-une-api-f37ae350cb9 (accédé le 11 avril 2018)
3. https://developer.twitter.com/en.html (accédé le 12 avril 2018)
4. https://developers.google.com/maps/?hl=fr (accédé le 12 avril 2018)

le design de l'API. Plusieurs d'entre eux ont pour but d'améliorer la documentation en la rendant plus adaptative. Ces travaux visent à faire que la documentation évolue en même temps que l'API, qu'elle se mette à jour de manière synchrone ou en l'enrichissant grâce aux différents exemples présents en ligne. Ce sont des travaux qui visent à résoudre le problème des erreurs d'utilisation en amont. Une autre partie des travaux ne prend pas le problème à la racine, mais cherche plutôt à améliorer la situation actuelle, en particulier, en mettant en place des outils pour détecter les erreurs d'utilisation. Pour ce faire, une des approches très couramment utilisée consiste à relever de bons ou mauvais patrons d'utilisation et de chercher leur présence dans le code des clients. Certaines APIs sont considérées comme à risque, donc des recherches ont aussi été faites pour voir si des clients utilisaient ces APIs et quels risques sur le plan de la sécurité cela peut causer. La dernière catégorie de recherche souvent réalisée concerne JavaScript qui est un langage que l'on peut considérer comme permissif et dont l'utilisation des APIs est une coutume. Un pan de recherche se consacre donc à essayer de rendre ce langage plus restrictif en y instaurant du typage et des règles plus contraignantes.

Tous ces travaux réalisés proposent des solutions partielles au problème d'utilisation d'API. Ce problème étant complexe il y a donc divers angles pour le traiter. Cependant, augmenter la documentation peut se révéler très intéressant surtout en y ajoutant des exemples qui permettent de mieux comprendre comment utiliser l'API. Mais cela ne résout pas le problème des développeurs peu rigoureux qui ne se donnent pas la peine de consacrer du temps à un bon apprentissage de l'API qu'ils utilisent. De plus, cela n'aide pas à corriger les erreurs déjà présentes. Donc cette approche doit être combinée avec un travail de détection. Or, les outils de détection sont encore perfectibles. Cela s'explique par leur approche basée sur les patrons d'utilisation. Ce qui fait que les outils fonctionnent mieux avec des patrons très clairs qui s'obtiennent plus facilement avec des APIs aux utilisations peu variées. Par contre, quand l'API peut être utilisée de façons variées, abstraire des patrons est plus difficile. Ceci explique le faible taux de détection de ces outils sur des APIs complexes. De plus, la majorité des approches a tendance à extraire les patrons d'utilisation et voir s'ils sont violés au sein du même client. Or, un client a dans une grande partie des cas tendance à utiliser l'API de la même façon. Le dernier point qui peut être exploré est le fait que ces approches ne sont pas spécifiques à une API, ce qui peut poser des problèmes pour détecter différents types d'erreur d'utilisation. Des optimisations sont donc possibles, avec en particulier une combinaison de différentes techniques pour attaquer de plusieurs fronts ce domaine de recherche.

1.3. Contribution

Dans ce mémoire nous proposons une approche pour détecter les utilisations risquées d'API grâce à une approche basée sur le système immunitaire. La particularité de l'approche est le parallèle qui est fait entre les détecteurs générés pour détecter les utilisations à risque et les lymphocytes T qui circulent dans l'organisme pour traquer les pathogènes. L'outil

procède en deux étapes qui sont dans un premier temps la génération des détecteurs et ensuite la détection des utilisations à risque. L'avantage est que la génération de détecteurs n'est nécessaire qu'une seule fois. Les détecteurs ainsi générés peuvent être utilisés à volonté pour assigner un score de risque aux méthodes évaluées. Le fait de ne comparer les méthodes qu'à un ensemble restreint de détecteurs par rapport à tous les bons cas d'utilisation est par conséquent plus rapide.

Une autre singularité concerne l'extraction des données. L'apprentissage par la foule (crowd learning) pour générer des détecteurs nécessitant de nombreux cas d'utilisations nous a conduits à élaborer une nouvelle technique d'extraction de données. Cette technique consiste à extraire des méthodes utilisant l'API à un moment `t` et un moment `t+1`. Nous considérons ainsi la méthode au moment `t` comme un mauvais cas d'utilisation si l'arbre syntaxique a été modifié par rapport à la version du code au moment `t+1` [38]. La version la plus récente étant considérée comme une correction de la précédente. Nous avons ajouté à cela un raffinement qui consiste à ne considérer comme mauvaises que les méthodes dont l'utilisation de l'API diffère entre les deux versions. Cette comparaison est faite grâce à la représentation de la méthode sous forme d'*API groum* [42], un graphe d'utilisation de l'API, et la comparaison de la similarité des *groum*s des deux versions.

Un point fort de la contribution est que le code source de clients qui utilisent correctement l'API peut rester privé. Ce code est utilisé pour la génération des détecteurs, qui ne s'intéressent qu'aux méthodes de l'API et non aux autres méthodes qui ne révèlent aucunement comment utiliser l'API. La confidentialité du code source des clients peut ainsi être préservée. Une compagnie peut donc contribuer à l'enrichissement de l'outil sans pour autant révéler ses secrets de fabrication. Ainsi la propriété intellectuelle est protégée. De plus il est possible de générer des détecteurs en fonction de la version de l'API. L'approche étant guidée par l'apprentissage par la foule, tout dépend des clients donnés en entrée et de l'API donnée en entrée. La seule contrainte est de fournir le nom de l'API et les méthodes qui la composent. L'approche est donc spécifique à une API.

Notre approche a été testée sur un ensemble de données obtenu grâce à notre méthode d'extraction des commits de correction de code. Elle a permis d'étudier l'apprentissage et les différentes configurations que peut avoir notre outil sur 3 APIs Java différentes et populaires. Le second ensemble de données testé est extrait de l'ensemble de données MUBench [5]. Il permet de comparer notre approche aux autres outils de détection d'erreur d'utilisation et montrer en quoi APIMMUNE est complémentaire des autres recherches menées dans le domaine.

1.4. Structure du mémoire

La suite de ce mémoire est organisée de la manière suivante : le chapitre 2 définit ce qu'est le système immunitaire et une API avant de faire un état de l'art visant à présenter

les différents travaux réalisés en termes de détection des erreurs d'utilisation d'API. Ensuite dans le chapitre 3 nous expliquerons notre approche avant de la détailler dans les deux chapitres suivants. Le chapitre 4 présente le travail préliminaire effectué. Il aborde la structure de données utilisée, l'extraction des données pour tester notre approche et le pré-traitement de ces dernières pour optimiser nos résultats. Ensuite, le chapitre 5 constitue le noyau de ce mémoire. L'approche y est détaillée en trois parties, la description de l'algorithme génétique, la génération de détecteur et l'ordonnancement des méthodes à évaluer. Puis le chapitre 6 concerne la validation de l'approche exposée précédemment avec la présentation des paramètres utilisés pour l'expérimentation et les résultats de ces tests avec bien évidemment la discussion. Ce mémoire s'achève par une conclusion au 7e chapitre.

Chapitre 2

DÉFINITIONS ET ÉTAT DE L'ART

Dans ce chapitre, nous allons tout d'abord définir les termes de base nécessaires à la compréhension de ce mémoire, en commençant par définir le système immunitaire synthétiquement et plus précisément le système immunitaire artificiel. Ensuite, nous aborderons le terme API qui est le sujet central de notre recherche. Dans un second temps, nous allons passer en revue l'état de l'art concernant les problèmes d'utilisabilité des APIs. Cet état de l'art se compose de trois parties, d'abord ce qui concerne la gestion du problème d'utilisabilité des APIs en amont, ensuite, comment détecter les erreurs d'utilisation et enfin comment aider à les corriger. Nous finirons par une synthèse pour récapituler tout ce qui concerne ce domaine de recherche et pour montrer en quoi notre approche est complémentaire aux travaux déjà réalisés.

2.1. Le système immunitaire

Notre approche est basée sur la métaphore du système immunitaire biologique. Nous allons donc commencer par voir ce qu'est le système immunitaire biologique. Ensuite, nous verrons ce qu'est le système immunitaire artificiel et comment nous pouvons faire un parallèle avec notre problématique de détection.

2.1.1. Le système immunitaire biologique

Le système immunitaire est le mécanisme permettant de défendre l'organisme de toute attaque extérieure. C'est un système qualifié de complexe, adaptatif, intelligent et robuste. Sa particularité est qu'il ne possède pas d'organe central (à contrario du système nerveux par exemple). Son fonctionnement se base sur une armée de détecteurs, les lymphocytes T, qui ont pour rôle de parcourir l'organisme afin de détecter les cellules pathogènes, et les lymphocytes B qui servent de mémoire afin de pouvoir lancer la réaction immunitaire plus rapidement et efficacement. Les pathogènes sont les cellules n'appartenant pas à l'organisme (comme des virus ou des bactéries). Elles sont donc différentes des cellules du corps en question. L'action du système immunitaire se divise en trois étapes : (1) la découverte, (2) l'identification et

(3) l'élimination des intrus. Pour notre approche, nous nous sommes intéressés à la première étape, la découverte.

Pour procéder à cette étape de découverte, les lymphocytes T errent dans l'organisme à la recherche d'intrus pouvant causer des dommages à l'organisme hôte. Ces détecteurs (lymphocytes) possèdent des pores n'acceptant qu'un certain type de cellule. Ce mécanisme de reconnaissance fonctionne de manière à ce que si les protéines de la cellule ont une correspondance avec un pore du détecteur, cela signifie que cette cellule est un antigène nuisible. Dans ce cas, on dit que la détection a lieu et la réaction immunitaire est lancée, avec pour but d'éliminer toutes les cellules nocives du même type. Un antigène est une substance étrangère à l'organisme tel qu'un microbe, une toxine ou une substance chimique. Il existe deux types d'antigène, les antigènes dites *self* qui sont les substances étrangères, telles que les bactéries dans les yaourts, et qui sont tolérées par l'organisme. Le deuxième type concerne les antigènes *non-self* qui sont des substances étrangères à l'organisme et qui doivent être éliminées, comme la bactérie de la tuberculose. Les antigènes *self* aident de différentes manières le système immunitaire en permettant de mieux différencier les *self* et les *non-self* ou en empêchant les *non-self* de s'introduire dans certaines cellules par exemple.

Le système immunitaire génère aléatoirement un grand nombre de détecteurs. Une optimisation naturelle de ces détecteurs est effectuée après leur génération [1]. C'est le principe de sélection négative. Elle a comme rôle d'éliminer les détecteurs qui matchent des cellules présentes dans le thymus [2] et la moelle osseuse, ces cellules étant, par conséquent, des antigènes *self* gardés en mémoire car inoffensifs envers l'organisme. Les détecteurs échouant à ce test sont donc éliminés. En revanche, les détecteurs passant ce test en ne détectant aucune cellule sont dupliqués pour répondre aux prochaines attaques. Lors de la duplication, les détecteurs peuvent être altérés. On dit qu'ils sont mutés. La mutation est une notion très importante du système immunitaire. Celle-ci permet de diversifier les potentielles cellules pouvant être détectées.

Nous venons de faire une description synthétique et générale du système immunitaire. Le but étant de comprendre son principe de fonctionnement et comment un comportement naturel peut répondre à une problématique que l'on peut catégoriser d'artificielle. Cela nous montre à quel point le biologique peut aider l'informatique. Une plus ample présentation du système immunitaire peut être lue dans le livre [**25**].

2.1.2. Le système immunitaire artificiel

Nous ne sommes pas les premiers à utiliser une approche basée sur le système immunitaire en informatique. Le système immunitaire artificiel a vu le jour dans les années 90 [**15**]. Ce paradigme de l'intelligence artificielle réplique le fonctionnement du système immunitaire

1. https://youtu.be/u2qRUtg2k3Y (accédé le 18 avril)
2. http://www2.nau.edu/~fpm/immunology/Exams/Tcelldevelopment-401.html (accédé le 17 avril)

organique. Il a servi pour des applications diverses et variées, comme, la détection d'anomalies [16], la reconnaissance de patterns [12], le data mining [28] ou encore la détection de fautes [10]. Cette liste des utilisations est loin d'être exhaustive, mais les deux principaux domaines d'étude où le système immunitaire artificiel est utilisé sont la sécurité et la maintenance. Une part importante de l'utilisation du système immunitaire artificiel dans la recherche est donc la détection, car il est par construction fait pour cette tâche, ce qui explique l'intérêt qui lui est porté dans ces deux domaines de recherche.

Pour utiliser cette approche, il est nécessaire de faire le pont entre les différents artefacts qui composent le système immunitaire artificiel. Pour faire l'analogie, il est nécessaire de définir cinq concepts fondamentaux. Tout d'abord, il faut définir le *corps*. Le *corps* est le domaine d'application du problème, dans les problématiques informatiques ce sera la majorité du temps l'ensemble du code que l'on souhaite étudier. Ensuite, il faut définir les *détecteurs* qui sont les entités qui vont permettre de trouver ce qui pose problème au sein du *corps*. Ce qui pose problème ce sont les *antigènes non-self*, ces cellules sont éléments du code que l'on cherche à détecter pour pouvoir les éliminer et répondre à la problématique posée. Le pendant de ces cellules sont les *antigènes self* qui eux, sont les bons éléments qui permettent la génération des *détecteurs*. C'est grâce aux *antigènes self* que la sélection négative est possible et que les *détecteurs* peuvent être efficaces. Le dernier artefact à définir est l'*affinité*. L'*affinité* est la métrique permettant de savoir s'il y a match ou non. Sans elle, il n'est pas possible de générer les *détecteurs* car on ne peut pas procéder à la sélection négative, et on ne peut pas procéder à la détection car il est impossible de déterminer si un détecteur matche avec une cellule.

2.2. Interface de Programmation d'Application (Application Programming Interface)

Les APIs voient le jour en 1985 grâce au précurseur Microsoft qui propose la toute première API publique pour participer au développement d'applications Windows. C'est une des raisons pour lesquelles ce système d'exploitation vivra l'essor qu'on lui connaît tous [45]. Mais c'est avec l'arrivée de l'Internet que les APIs se sont démocratisées. Avec, en particulier, Salesforce et eBay qui en 2000 sont les premiers à mettre à disposition leur propre interface [3]. Ils seront suivis par toutes les grandes firmes ensuite. Aujourd'hui, il est inimaginable de ne pas proposer sa propre API rattachée à son service, dans un monde où l'interaction est omniprésente.

3. https://history.apievangelist.com/ (accédé le 12 avril 2018)

2.2.1. Définition d'une API

Une API est une interface de programmation (**A**pplication **P**rogramming **I**nterface). Décomposons le terme API afin de mieux comprendre en quoi consiste ce terme d'interface programmation.

Tout d'abord, **A** c'est pour application. L'API est fournie pour une application, utilisation en particulier. Une application est un terme vaste qui peut désigner une classe Java, comme la classe `java.util.Iterator`, qui est la classe qui implémente l'itérateur en Java. Ou bien un package de classes Java comme `javax.crypto` qui comporte plusieurs classes qui interagissent pour proposer des fonctionnalités cryptographiques. Mais cela peut aussi être Twitter qui fournit une **Re**presentational **s**tate **t**ransfer (Rest) API pour pouvoir par exemple récupérer des tweets, des utilisateurs ou des messages. La différence entre une API et une Rest API, est que cette dernière possède une sur-couche qui fait qu'au lieu d'utiliser des méthodes directement fournies pour interagir, on obtient des données sous format JSON [4]. La Rest API peut être définie comme une API de récupération de données. Elle est très populaire sur le Web. En somme, toute partie de logiciel pouvant être séparée de son environnement pour être appliqué dans un autre programme peut être considérée comme le **A** d'API.

Le **P** est pour programmation, car l'API est destinée à celui qui programme, même si certains services permettent l'encapsulation de l'utilisation de l'interface sans programmer (par exemple `Zapier` [5] et `IFTTT` [6]). Mais lorsque le développeur interagit avec l'API, il peut recevoir une réponse sous un format particulier qu'il va devoir traiter pour l'intégrer dans son projet, comme une réponse au format JSON dans le cas de la Rest API. L'API peut aussi donner accès à une fonctionnalité comme une structure de données, telle qu'une liste avec `java.util.ArryList` [7]. Les projets sont aussi variés que les développeurs. L'API utilisée par ce dernier lui permet de récolter des données ou des fonctionnalités avec lesquelles son programme doit interagir et ainsi gagner en productivité. Pas besoin de réinventer la roue ou devoir créer un outil d'extraction de données spécifique à une plateforme, l'API se charge de tout. Ainsi le développeur se retrouve face à un niveau d'abstraction plus élevé et n'est pas contraint de se préoccuper de ce qui se produit derrière l'API qu'il utilise.

Le **I** est pour interface. C'est de mon point de vue le côté le plus intéressant de l'API. Car en tant qu'interface cela veut dire qu'il y a une interaction. Et le but de l'API est de faciliter cette interaction entre l'application et le programme. Il faut d'une part faciliter l'accès à l'interface aux développeurs qui veulent utiliser l'API, avec par exemple une bonne documentation, des exemples et une architecture réfléchie, et d'autre part, faciliter l'accès

4. https://medium.com/digitalmind/what-is-rest-api-e126b35606f9 (accédé le 18 avril 2018)
5. https://zapier.com (accédé le 18 avril 2018)
6. https://ifttt.com (accédé le 18 avril 2018)
7. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html (accédé le 18 avril 2018)

aux programmes qui utilisent cette API, avec des performances optimisées. Toujours dans le but de gagner en productivité, rapidité et agilité.

2.2.2. Types d'API

Les APIs n'ont pas toutes le même rôle et peuvent être très différentes. C'est pour cela que nous pouvons les regrouper en quatre types distincts [8] :

1. Les APIs pour *systèmes d'exploitation* qui permettent au programme développé d'interagir avec la machine sur laquelle il est exécuté. Cela permet par exemple de récupérer des événements tels que l'appui sur une touche du clavier ou d'utiliser les fonctions du microphone. C'est le premier type d'API qui a vu le jour comme nous l'avons vu avec Microsoft en 1985 [45].
2. Les APIs pour *langage de programmation*. C'est ce type d'API que nous allons étudier dans ce mémoire. Cela permet d'utiliser des fonctionnalités déjà implémentées dans un langage pour ne pas "*réinventer la roue*". Cela peut être par exemple des structures de données ou des protocoles réseau, où il n'y aura qu'à faire appel aux méthodes publiques sans se soucier de l'implémentation. Ces APIs permettent d'augmenter le niveau d'abstraction lors de l'utilisation du langage, on dit que l'on programme à plus haut niveau (d'abstraction).
3. Les APIs d'*infrastructure* permettent quant à elles l'accès à un service à distance comme une machine virtuelle, un serveur... Amazon Web Service [9] en est un parfait exemple.
4. Les APIs de *service Web* permettent, comme leur nom l'indique, d'accéder à des services en ligne. C'est le type d'API le plus en vogue car elles ont permis à certaines entreprises d'acquérir leur renommée. Parmi les plus connues, il y a les réseaux sociaux Facebook et Twitter. Mais les APIs de Google Maps entrent aussi dans cette catégorie en offrant différents services de localisation.

2.3. État de l'art sur l'utilisabilité des APIs

Comme nous venons de le voir, les APIs sont variées de par leurs types et leurs caractéristiques d'utilisation. Leur nombre et leur popularité ne cessent de croître afin de répondre aux demandes de productivité imposée par l'essor de la connectivité au sein de la Société. Cependant, cette diversité fait que l'interface de programmation n'est pas toujours employée de la bonne façon. Cela est dû à plusieurs facteurs. Les facteurs peuvent être une mauvaise documentation [47], une mauvaise étude de l'utilisation ou encore une incompréhension causée

8. https://medium.com/@mercier_remi/02-il-%C3%A9tait-une-fois-les-apis-a8a723b2b96b (accédé le 11 avril 2018)
9. https://aws.amazon.com/fr/ (accédé le 18 avril 2018)

par l'architecture. Il peut y avoir des utilisations hors du commun conduisant à des risques. Ces risques pouvant être des effets de bords peu faciles à anticiper. Les chercheurs se sont donc penchés sur ces différents aspects et nous allons faire un tour d'horizon des différents travaux réalisés dans ce domaine de recherche. Dans un premier temps, nous allons explorer les études concernant le design de l'API pour voir les causes des erreurs d'utilisation. Ensuite, nous verrons les études concernant la détection des erreurs d'utilisation et nous finirons par l'analyse de ce qui pourrait aider à la correction de ces erreurs.

2.3.1. Problèmes d'utilisabilité liés au design de l'API

Bayer et Pinzger [9], après une étude menée sur le site *Stack Overflow* [10] de question et réponses (Q&A) à propos du développement d'application Android, ont découvert qu'une des principales problématiques rencontrées par les développeurs est l'utilisation d'API. Ce qui montre l'intérêt qu'il y a à aider les développeurs dans leur processus d'utilisation d'API, au vu du nombre d'applications mobiles développées. Linares-Vásquez et al. [31] ont réalisé une étude empirique sur le lien entre la popularité d'une application et les APIs qu'elle utilise. Leur conclusion est que les APIs employées par les applications à fort succès sont moins sujettes aux changements et ont moins de fautes répertoriées. L'utilisation d'une bonne API aide donc à produire un programme que l'on peut juger de meilleure qualité. Stevens et al. [53] confirment ces résultats en faisant le lien entre la popularité et les erreurs d'utilisation. Nadi et al. [36] ont également réalisé une étude empirique auprès des développeurs sur les obstacles rencontrés lors de l'utilisation des APIs de cryptographie Java. Les auteurs basent leurs recherches sur des données qui comportent l'analyse de 100 publications de *Stack Overflow*, de 100 projets *GitHub* et de 48 contributions développeurs. Ils découvrent que, bien que les développeurs aient du mal à utiliser correctement certains algorithmes cryptographiques, ils se montrent étonnamment confiants dans la sélection des concepts de cryptographie appropriés. Ils constatent également que les APIs sont généralement perçues comme étant trop compliquées à utiliser en raison d'un niveau d'abstraction trop bas et que les développeurs préfèrent des solutions plus basées sur les tâches, avec des exemples et des corrections suggérées pour les problèmes les plus récurrents. Comme le constatent ces différentes études, de multiples facteurs peuvent être étudiés pour améliorer l'utilisation des APIs et donc diminuer le nombre d'erreurs. À travers cette section nous allons voir tout ce qui peut être fait en amont pour prévenir les erreurs d'utilisation et contribuer à une meilleure utilisabilité des interfaces de programmation.

2.3.1.1. *Migration*

Comme nous l'avons vu précédemment, une API qui subit de nombreux changements entraîne des programmes plus susceptibles de comporter des fautes. Lamothe et Shang [29]

10. https://stackoverflow.com/

ont donc étudié la migration d'une API d'une version à une autre. Ils ont décidé de migrer des applications utilisant l'API Android en applications utilisant `FDroid`. Pour ce faire, ils ont utilisé le code source de l'API Android et sa documentation. Ils se sont rendu compte au cours de la migration que plusieurs méthodes étaient dépréciées et ils n'avaient aucune méthode de substitution. Ils en concluent que pour réaliser une bonne migration il est donc nécessaire d'avoir une bonne documentation, robuste aux différents changements pouvant survenir durant l'évolution de l'interface de programmation.

2.3.1.2. *Dépréciation de méthode*

Un des points négatif relevé lors de la migration faite par Lamothe et Shang [29] est la présence de méthodes dépréciées. Sawant et al.[50] ont décidé de mener une enquête sur les méthodes dépréciées. La dépréciation de méthode est une caractéristique des APIs qui permet de marquer une méthode comme obsolète. Dans cet article, l'investigation porte sur les effets au niveau de l'utilisateur. Leur objectif est de voir quand est-ce qu'une méthode est dépréciée et pourquoi ces méthodes sont catégorisées comme obsolètes. Cela a été réalisé avec une interview semi-structurée de 17 tierces parties productrices d'API Java et une enquête menée sur 170 développeurs Java. Pour en arriver à la conclusion qu'il n'y a pas de compréhension possible d'un bon mécanisme de dépréciation et que ce mécanisme est utilisé sans précepte derrière. Les développeurs quant à eux perçoivent ce recours comme l'ultime avertissement avant la suppression d'une méthode et qu'il faut pallier au plus vite à ce problème. Cependant comme Lamothe et Shang l'ont montré, certaines méthodes dépréciées perdurent avec le temps sans pour autant être remplacées au sein du code et peuvent créer de potentiels problèmes.

2.3.1.3. *Documentation*

Un sujet primordial lors de la création de l'API hormis la qualité du code, c'est la qualité de la documentation. La documentation a le premier rôle lorsque l'on veut comprendre comment utiliser une API. Or, il n'est pas facile de la créer et encore moins de la maintenir à jour, car comme nous l'avons vu précédemment une API évolue avec le temps. Lors de leur revue des travaux menés sur *Stack Overflow*, Ahmad et al. [3] relèvent que la documentation est d'une pauvre qualité et manque d'instructions détaillées et d'exemples. Cela étant dû au fait que la documentation est écrite par peu de personnes, principalement par les développeurs qui savent comment l'utiliser, car ils l'ont créée. En revanche, elle est utilisée par de nombreuses personnes, et tous les utilisateurs qui arrivent dans la documentation se posent pléthore de questions.

Pour combler le manque d'information, les développeurs se rendent sur des sites de Q&A (**Q**uestion & **A**nswer) comme *Stack Overflow* pour avoir des réponses à leurs questions.

Ainsi toute la communauté peut aider à éclaircir un point qui ne l'est pas dans la documentation. C'est de ce postulat que Treude et Robillard [57] sont partis pour produire leur outil `SISE`. Cette approche permet de renforcer la documentation avec des détails utiles grâce à l'apprentissage machine supervisé. Ces informations supplémentaires sont extraites de *Stack Overflow* et ajoutées à la documentation pour la compléter et aider le développeur lors de son apprentissage de l'utilisation de l'API.

Donc comme on vient de le voir, ce qui est recherché sur les sites de Q&A ce sont des exemples. Les exemples permettent une meilleure compréhension et pour certains, une meilleure technique d'apprentissage. Le problème est que les exemples et la documentation sont indépendantes. Subramanian et al. [54] ont donc travaillé sur `Baker` pour fournir un lien entre la documentation et des exemples pour des APIs Java et JavaScript. `Baker` peut tout aussi bien être utilisé pour incorporer des exemples au sein de la documentation de l'API que pour ajouter des liens vers la documentation au sein de fragments de code utilisant l'API. Cet outil apporte la complémentarité qu'il manque tant entre la documentation et les exemples pour fournir le complément d'information nécessaire à l'utilisation de l'API.

Un léger contre-pied est pris par Buse et Weimer [11] qui nous présentent une technique automatique pour extraire et synthétiser une documentation succincte et représentative des interfaces de programmation. Leur algorithme est basé sur une combinaison d'analyse de chemin de flux de données sensibles, de regroupement et d'abstraction de patterns. Leur approche produit des extraits de programme bien typés qui documentent l'initialisation, les appels de méthode, les affectations, les constructions en boucle et le traitement des exceptions. Leurs exemples générés se sont révélés être au moins aussi bons dans des instances écrites par l'homme d'après une étude impliquant plus de 150 participants. Les auteurs ont donc décidé de partir des exemples pour produire une documentation, mais de nombreuses recherches partent du principe que l'apprentissage de l'utilisation d'une interface de programmation se fait, à l'inverse, grâce aux exemples, c'est ce que nous allons voir.

2.3.1.4. *Exemples*

Subramanian et al. [54] ne sont pas les seuls à avoir travaillé sur les exemples d'utilisation d'API et avoir compris leur rôle primordial lors de l'apprentissage. Grâce à l'Internet, l'entraide pour le développement est très présente, mais cela engendre un problème. Lorsque le développeur recherche un exemple comment trouver le bon ? Il est difficile de comprendre à premier abord les nuances entre deux exemples, surtout lorsque l'on est en apprentissage. De plus, la masse d'exemples sur l'Internet est considérable, il n'est pas simple de faire un choix parmi toutes les plateformes disponibles et parmi tous les postes présents.

Pour aider à cet apprentissage par l'exemple, Wang et al. [58] ont décidé de fournir un outil de visualisation d'exemple d'utilisation d'API. Pour ce faire, ils extraient des exemples d'utilisation à travers l'Internet, à la recherche de code et de textes explicatifs accompagnant

ce code. Leur outil `APIExample` permet d'afficher les résultats de manière triée et regroupée par utilisation. De leur côté, Glassman et al. [24] répondent au problème d'apprentissage de l'utilisation d'API en fournissant un outil de visualisation d'exemples aussi. `Examplore` mine plus de 380K projets *GitHub* afin de récolter un maximum d'exemples, et ensuite génère un squelette synthétique de code qui résume différentes utilisations de l'API en une seule visualisation. Ils ont réussi à montrer que leur approche permet aux 16 développeurs interrogés de répondre à plus de questions à propos de l'utilisation d'API et avec plus de confiance.

Dans la continuité de ces travaux, Zhang et al. [63] ont développé `EXAMPLECHECK` qui extrait des patrons d'utilisation Java du même corpus de 380 125 projets *GitHub*, mais cette fois-ci pour vérifier les exemples sur *Stack Overflow*. Ils ont découvert que 31% des 217 818 postes analysés sur *GitHub* contiennent de potentielles violations d'utilisation d'API qui peuvent causer des comportements inattendus tels que des crashs ou la corruption de ressources. Les trois principales raisons de ces violations d'utilisation sont le manque de structure de contrôle, le manque ou l'inversion d'appels à l'API ou des conditions de vérification incorrectes. De plus, l'étude montre que les exemples les plus plébiscités par la communauté ne sont pas ceux avec le moins d'erreurs d'utilisation. Donc l'apprentissage par l'exemple peut se révéler dangereux et propager des erreurs d'utilisation. Cependant, ces résultats sont à nuancer, car les exemples proposés en réponse aux questions servent plus de point de départ à une investigation plus poussée, et se veulent précis que sur le point abordé par la question. L'utilisateur de l'API pourra éclaircir avec d'autres exemples les points sombres de l'exemple qu'il a sous les yeux et pourra peaufiner avec la documentation les détails manquants.

Donc le site *Stack Overflow* reste tout de même une référence en termes d'approvisionnement d'exemple car il permet de connaître les difficultés rencontrées lors de l'utilisation d'API et d'avoir à la fois un texte en langage naturel accompagné de code, soit les deux langages que maîtrise le développeur. Wang et al. [60] se sont attelés à fournir les meilleurs postes pour un problème concernant une API spécifiée pour des applications mobiles. Leur approche se base sur plusieurs techniques, dont l'analyse de réseaux sociaux et le minage sur des sujets spécifiques. Ainsi ils ont montré qu'ils ont des recommandations plus stables comparées à une recommandation aléatoire ou par réputation de l'auteur. Donc en fournissant des exemples plus pertinents accompagnés de texte en langage naturel cela peut être une grande une aide à la correction des erreurs.

Mais tous les exemples ne sont pas tirés de *Stack Overflow*. L'inconvénient de ce site de Q&A est que les exemples sont de petite taille. Aller explorer directement des répertoires Open Source peut donc se révéler utile dans cette quête du meilleur exemple d'utilisation. Zhong et al. [65] ont développé un framework d'exploration d'API et son outil de support appelé `MAPO` (Mining API usage Pattern from Open source repositories) pour explorer automatiquement les modèles d'utilisation de l'API. Un modèle extrait décrit un certain scénario

d'utilisation. Des méthodes de l'API sont fréquemment appelées ensemble et leurs utilisations peuvent suivre certaines règles séquentielles. `MAPO` recommande en outre les modèles d'utilisation de l'API étudiée et leurs fragments de code associés à la demande des programmeurs. Leurs résultats montrent qu'avec ces modèles, `MAPO` aide les programmeurs à localiser des extraits de code utiles plus efficacement que deux outils de recherche de code à la pointe de la technologie (*Strathcona* et *Google code search*). Les résultats montrent que les programmeurs produisent du code avec moins de bugs face à des usages d'API relativement complexes. Kim et al. [**27**], dans le même esprit, proposent un nouveau moteur de recherche de code, combinant la puissance des documents de navigation et la recherche d'exemples de code. En renvoyant des documents incorporés avec des exemples de résumé de code de haute qualité extraits du Web. Leurs résultats d'évaluation montrent que leur approche `eXoaDocs` fournit des exemples de code avec une grande précision et stimulent la productivité des programmeurs.

Malgré certaines approches extrayant des exemples depuis des répertoires Open Source, certains chercheurs trouvent que les exemples proposaient ne prennent pas assez en compte les utilisations importantes d'interface de programmation. Ils trouvent en quelques sortes que les exemples proposés sont trop petits. C'est pour cela que Moritz et al. [**35**] présente une technique permettant d'extraire et de visualiser automatiquement des exemples d'utilisation d'API. Contrairement aux approches précédentes, leur technique est capable de trouver des exemples d'utilisation de l'API qui se produisent dans plusieurs fonctions d'un programme. Cette distinction est importante en raison d'un décalage entre les outils d'apprentissage d'API actuels et les besoins des programmeurs : les outils actuels extraient des exemples relativement petits de fichiers / fonctions uniques, même si les programmeurs utilisent des APIs pour construire de gros logiciels. Les petits exemples sont utiles dans les premières étapes de l'apprentissage de l'API, mais omettent les détails qui sont utiles dans les étapes ultérieures. Leur approche est destinée à combler cette lacune. Elle fonctionne en représentant un logiciel en tant que modèle de sujet relationnel (Relational Topic Model), où les appels d'API et les fonctions qui les utilisent sont modélisés comme un réseau de documents. Étant donné l'API de départ, leur approche peut recommander des exemples d'utilisation d'API complexes extraits d'un référentiel de plus de 14 millions de méthodes Java.

Autre aspect pouvant aider l'utilisateur, et qui rejoint le domaine de l'exemple, c'est la complétion du code. L'utilisation d'IDE est désormais plus qu'ancrée dans les usages. Le développeur s'est habitué à regarder dans un premier temps les suggestions de complétion qui lui sont faites lorsqu'il utilise une méthode. Hou et Pletcher [**26**] via `Best Code Completion` propose d'améliorer la complétion du code des APIs grâce à plusieurs méthodes. Dans un premier temps, ils proposent de trier les méthodes par nombre d'appels et type hiérarchique pour diminuer le nombre de suggestions proposées à l'utilisateur. Ensuite, ils montrent que

le contexte permet de proposer de meilleures suggestions. Et enfin, ils décident de regrouper les propositions par utilisation afin d'avoir les suggestions les plus pertinentes possible.

Récapitulatif *:* Les travaux que nous venons de passer en revue montrent que la réalisation d'une API facilement utilisable n'est pas simple. Le créateur doit réaliser une documentation claire et explicite, mais cela ne suffit pas, car les utilisateurs ont aussi besoin d'exemples. Des solutions de minage voient le jour pour aider le développeur à réduire son processus d'apprentissage en enrichissant la documentation, documentation qui évolue en même temps que l'API en montrant les différents aspects de son utilisation. Le constat final est que les utilisations à risque sont dues au manque d'information du développeur et à sa capacité à déceler les bonnes ressources à utiliser pour produire un code de qualité. C'est pour cela que plusieurs approches essayent d'allier tous ces aspects en proposant une documentation évolutive qui s'enrichit d'exemples pertinents et d'explications annexes récoltées en ligne.

2.3.2. Méthodes de détection de code à risque

Dans le paragraphe précédent, le précepte adopté était "*prévenir pour guérir*". Le but était de fournir l'information la plus pertinente possible pour que l'utilisateur emploie l'interface de programmation de la meilleure façon possible. Maintenant, nous allons voir les études qui portent sur la détection des risques liés aux APIs. Dans un premier temps, nous passerons en revue les différents outils permettant comme APIMMUNE de détecter les potentielles erreurs d'utilisation d'API. Et dans un second temps, nous verrons qu'il existe des approches permettant de détecter les utilisations des APIs à risque.

2.3.2.1. *Outils de détection des utilisations à risque d'API*

La communauté scientifique a établi différentes techniques de détection des erreurs d'utilisation d'API. Pour commencer, Nayrolles et Hamou-Lhadj [**37**] ont décidé de procéder à une vérification du code lors de la soumission de ce dernier via un commit. Leur approche nommée `CLEVER` intercepte les commits avec la présence de bugs et donne des indices de correction. Développé en partenariat avec Ubisoft `CLEVER` peut détecter les commits à risque avec une précision de 79% et un rappel de 65%. Il est à noter que pour un processus de détection, la précision est le ratio des vrais positifs sur le total des cas jugés positifs, et le rappel est le ratio des vrais positifs sur le nombre total de cas réellement positifs (détectés ou non). En plus de détecter les commits à risque, cet outil donne des indices pour corriger les troubles relevés. La méthode de détection récupère toutes les modifications présentent entre deux commits et les compare à leur base de données de commits à risque. Ce récent article (2018) a une approche dans le même esprit que notre outil APIMMUNE en comparant le commit aux mauvais commits précédemment relevés mais leur approche reste tout de même différente, car seul notre apprentissage utilise deux versions de commits.

Certains auteurs ont pris le parti d'étudier des APIs spécifiques. C'est le cas d'Egele et al. [18] qui ont mené une étude empirique sur 11 748 applications Android. Avec `CRYPTOLINT`, ils extraient automatiquement des erreurs d'utilisation des APIs cryptographiques, dans ce corpus, en passant d'une représentation bas niveau, à un graphe intraprocédural de flux de contrôle. Et à partir de ce graphe, ils cherchent des violations de 6 règles définies. Les résultats montrent que dans 88% des cas il y a au moins une erreur liée à l'utilisation de ces APIs. Ce qui met en lumière un grand problème de sécurité dû à ces erreurs d'utilisation.

Mais d'autres chercheurs ont préféré étudier des erreurs d'utilisation plus spécifiques. Wasylkowski et al. [62] se sont penchés sur la question des appels de méthodes. Avec leur outil `Jadet` qui se concentre sur l'ordre des appels dans les utilisations. Ils identifient des patrons d'utilisations en appairant des appels aux méthodes de l'API. Ainsi `Jadet` est capable de détecter les oublis d'appel de méthodes, de boucles principalement. Wasylkowski et Zeller ont aussi mis au point `Tikanga` [61] qui est construit sur `Jadet`. Cette fois-ci en travaillant sur les préconditions d'appel de fonction. Une précondition décrit l'état qui doit être atteint avec l'appel de la fonction. C'est un type très répandu d'erreur d'utilisation d'API car la documentation ne précise pas toujours le besoin de vérifier une condition avant d'appeler une méthode. En combinant une analyse statique et de la vérification de modèle, leur approche génère ce qu'ils appellent le `Computation Tree Logic` $^{\text{Fair}}$. Ils utilisent ce principe d'arbre sur les utilisations d'objets et ils appliquent la vérification de patron sur ces arbres. `Tikanga` emploie l'analyse conceptuelle formelle [22] pour explorer les patrons et relever les erreurs d'utilisation en même temps. Ainsi ils ont pu détecter 169 violations de préconditions avant appel d'une méthode. Il aura fallu donc plusieurs travaux afin de mettre au point des outils pouvant être combinés afin de détecter différents types d'erreurs d'utilisation.

Une approche qui se veut plutôt spécifique mais qui révèle différents types de violations est celle de Monperrus et al. [33], [34] qui vise à détecter les erreurs d'utilisation dans les appels de méthodes de l'API avec l'oubli d'exactement un appel. Avec leur approche `Detect Missing Method Calls`, ils s'intéressent aux utilisations de type, c'est-à-dire aux ensembles de méthodes appelés sur un type donné, d'une méthode donnée. L'hypothèse qu'ils font est que les violations ne devraient avoir que peu d'usages exactement similaires, et considèrent que la majorité des appels similaires sont les bonnes utilisations. Ils assignent donc des scores d'étrangeté aux méthodes pour déterminer leur degré de risque de violation de l'utilisation. Leur étude montre que cette technique révèle en plus des appels manquants, des violations des bonnes pratiques.

`Alattin` [55] quant à lui mine des patterns alternatifs pour la vérification de conditions. Il applique l'exploration des éléments fréquents sur l'ensemble des règles de vérification avant et après la condition, sur le récepteur, les arguments, et la valeur de retour d'un appel de méthode. Il détecte les vérifications de précondition de valeur nulle ou manquante, ou les conditions d'état qui sont assurées par vérification.

`CAR-Miner` [56], quant à lui, a pour but de détecter les erreurs d'utilisation d'API avec la manipulation d'erreur. Pour trouver une erreur d'utilisation, l'outil extrait la séquence d'appel normale et la séquence d'appel d'exception pour un appel à une méthode. Il apprend les règles d'association et ensuite détermine quelle exception est censée être utilisée. `CAR-Miner` déclare une violation si la séquence évaluée ne comporte pas l'exception attendue. Cette approche est capable de détecter la mauvaise gestion des exceptions ainsi que les appels de méthode manquants parmi les fonctions qui manipulent des erreurs.

`AX09` [2] détecte lui aussi les erreurs de manipulation d'exception. L'approche d'Acharya et Xie utilise la vérification de modèle pour générer des chemins de gestion d'erreur en tant que séquences d'appels de méthode et applique une exploration de sous-séquence fréquente pour détecter des patrons. Il utilise ensuite la vérification push-down du modèle pour vérifier la cohérence de ces modèles et identifier les mauvais usages respectifs tels que la gestion des erreurs manquantes ainsi que les appels de méthodes manquants parmi les fonctions de gestion des erreurs.

Comme on peut le constater, de nombreux outils ont tendance à relever les patrons d'utilisation des usages de l'API. L'outil `PR-Miner` [30] utilise une technique d'exploration de données appelée extraction fréquente d'éléments pour extraire efficacement les règles de programmation implicites du code de logiciels imposants, écrits dans un langage de programmation industriel tel que le C. Grâce à l'extraction d'éléments fréquents, `PR-Miner` peut extraire des règles de programmation sous des formes générales (sans être contraint par des modèles de règles fixes) qui peuvent contenir plusieurs éléments de différents types tels que des fonctions, des variables et des types de données. En plus de la possibilité d'extraire ces règles de programmation, Li et Zhou proposent également un algorithme pour détecter automatiquement les violations de ces règles de programmation extraites, qui sont des indications intéressantes de bugs. Les erreurs d'utilisation sont définies comme le sous-ensemble survenu au minimum 10 fois moins fréquemment que le pattern étudié. `PR-Miner` se concentre principalement sur la détection des appels de méthode manquants.

Autre approche qui considère comme erreur d'utilisation un usage moins fréquent. Celle de Mover et al. [51] qui utilisent l'extraction des *groum*s, comme notre approche APImmune, pour apprendre les patrons d'utilisation pour des frameworks orientés objet à partir de corpus de programmes. Un *groum* est un graphe acyclique orienté où les nœuds représentent les appels aux constructeurs, méthodes et structures de contrôle et où les arêtes représentent l'ordre d'usage temporel et les dépendances de données entre les nœuds [42]. Le principal défi consiste à exploiter les *groum*s de manière horizontale à travers différents programmes, par opposition à de simples utilisations verticales dans le programme d'un seul développeur. Pour relever ce défi, les auteurs ont développé un nouvel algorithme d'exploration de *groum*

qui évolue sur un grand corpus de programmes. Leur approche `BIGROUM` utilise d'abord l'extraction fréquente d'éléments pour restreindre la recherche de *groum*s à de plus petits sous-ensembles de méthodes dans le corpus donné. Ensuite, leur algorithme pose l'isomorphisme du sous-graphe comme un problème SAT (de satisfaisabilité) et applique des algorithmes de pré-traitement efficaces pour exclure les comparaisons infructueuses à l'avance. Enfin, il identifie les relations de confinement entre les groupes de *groum*s pour caractériser les modes d'utilisation populaires dans le corpus et ainsi classer les modèles moins populaires comme des anomalies possibles.

Ramanathan et al. [46] visent à détecter des patrons dans les ordres d'appels les plus fréquents à l'intérieur des graphes de flux de contrôle interprocéduraux. S'il n'y a pas plus de 80% de l'ordre d'appel respecté, la méthode testée est considérée comme une violation. `Chronicler` détecte les appels de méthode manquants. Puisque les boucles sont déroulées une seule fois, elles ne peuvent pas détecter les itérations manquantes.

Nguyen et al. utilisent une base différente pour leur approche, car avec leur outil `DroidAssist` [41] détecte les erreurs d'utilisation à partir du byte code Java. Il apprend de ce byte code les séquences d'appels de méthodes pour construire un modèle de Markov caché. Si une probabilité d'une séquence d'appels donnée est trop petite alors l'approche de Nguyen et al. juge la séquence comme une erreur d'utilisation. Cependant, la tâche première de cet outil est de recommander des usages d'API lors du développement d'application Android. Donc cet outil est plus fait pour la complétion que la détection des usages suspects.

2.3.2.2. *Outils de détection d'utilisation d'une API à risque*

Nous venons de voir une douzaine d'approches différentes qui tentent de répondre à leur manière au problème de la détection d'erreur d'utilisation d'API. Maintenant, nous allons voir les travaux qui concernent l'utilisation d'API à risque.

Dai et al. [14] présentent une solution qui identifie et empêche l'utilisation abusive de l'API de contrôle *ActiveX* dans *Internet Explorer*. Ils construisent des modèles pour représenter les fonctionnalités normales des méthodes *ActiveX*, et identifient l'utilisation abusive de l'API *ActiveX* en identifiant les méthodes pouvant atteindre les APIs (système) dangereuses. Ils développent ensuite une technique pour *Internet Explorer* pour empêcher l'utilisation de méthodes *ActiveX* dangereuses. Leur approche est efficace pour détecter l'utilisation abusive de l'API *ActiveX* et a un surcoût négligeable pour la prévention des attaques.

Pour continuer dans la communication entre services, Georgiev et al. [23] démontrent que la validation du certificat SSL est complètement interrompue dans de nombreuses applications et bibliothèques critiques pour la sécurité. Toute connexion SSL de l'un de ces programmes n'est pas sécurisée contre une attaque de type "man-in-the-middle". Les causes profondes de ces vulnérabilités sont les APIs des implémentations SSL mal conçues (telles

que JSSE, OpenSSL et GnuTLS) et les bibliothèques de transport de données (telles que cURL) qui proposent aux développeurs un ensemble de paramètres et d'options confus.

Padekar et al. [43] introduisent un framework de sécurité appelé `AEGIS` pour empêcher les APIs de contrôleur d'être mal utilisées par des applications réseau malveillantes. Grâce à la vérification au moment de l'exécution des appels d'API, `AEGIS` effectue un contrôle d'accès précis pour les APIs de contrôleur importantes qui peuvent être mal utilisées par des applications malveillantes. L'utilisation des appels API est vérifiée en temps réel par des règles d'accès de sécurité sophistiquées qui sont définies en fonction des relations entre les applications et les données dans le contrôleur **S**oftware-**D**efined **N**etworks.

Sauvanauda et al. [49] dans leur article décrivent un système de détection d'anomalies (ADS) conçu pour détecter les erreurs liées au comportement erroné du service et les violations de SLA dans les services de cloud. Un objectif majeur est d'aider les fournisseurs à diagnostiquer les machines virtuelles anormales (VM) sur lesquelles un service est déployé ainsi que le type d'erreur associé à l'anomalie. Leur `ADS` comprend une entité de surveillance du système qui recueille les compteurs de logiciels caractérisant le service de cloud, ainsi qu'une entité de détection basée sur des modèles d'apprentissage automatique. De plus, une entité d'injection de faute est intégrée dans l'`ADS` pour la formation des modèles d'apprentissage automatique. Cette entité est également utilisée pour valider l'`ADS` et pour évaluer sa performance de détection et de diagnostic d'anomalie.

Ahmadi et al. [4] visent à montrer la faisabilité de concevoir un anti-malware basé sur l'apprentissage, sur les appareils mobiles Android. En outre, ils souhaitent démontrer l'importance d'un tel outil pour mettre fin à de nouveaux logiciels malveillants qui ne peuvent pas être facilement détectés par des outils de sécurité basés sur la signature. Pour ce faire, les auteurs proposent d'abord l'extraction d'un ensemble de fonctionnalités légères mais puissantes à partir d'applications Android. Ensuite, ils intègrent ces fonctionnalités dans un espace vectoriel pour construire un modèle efficace. Par conséquent, le modèle peut effectuer l'inférence sur l'appareil pour détecter des applications potentiellement dangereuses. Leur approche `IntelliAV`, fournie des performances plus satisfaisantes que les principaux produits anti-malware populaires.

Stevens et al. [53] ont pour objectif d'obtenir un aperçu de la mauvaise utilisation des autorisations et des discussions sur les autorisations dans les forums en ligne. Ils analysent environ 10 000 applications populaires gratuites sur la plateforme de distributions d'application Android et ont trouvé une relation sous-linéaire significative entre la popularité d'une autorisation et le nombre de fois où elle est mal utilisée. Ils étudient également la relation entre l'utilisation des autorisations et le nombre de questions sur l'autorisation de *Stack Overflow*.

Shuai et al. [52] effectuent une analyse systématique des mauvaises utilisations cryptographiques, construisent le modèle de vulnérabilité de ces erreurs d'utilisation cryptographiques

et implémentent un prototype nommé **C**rypto **M**isuse **A**nalyzer. CMA peut effectuer une analyse statique sur les applications Android et sélectionner les branches qui invoquent l'API cryptographique. Ensuite, il lance l'application en suivant la branche cible et enregistre les appels d'API cryptographiques. Enfin, CMA identifie les failles de sécurité de l'API cryptographique à partir des enregistrements basés sur le modèle prédéfini. Les auteurs ont constaté grâce à leur approche que plus de la moitié des applications sont touchées par des vulnérabilités concernant les APIs de cryptographie.

Récapitulatif *:* Nous venons de voir qu'il y a plusieurs approches qui emploient les patrons pour détecter des erreurs d'utilisation d'interface de programmation. Nous avons pu noter que ces patrons servent à comparer les utilisations évaluées aux méthodes des patrons, afin de qualifier d'anomalie toute utilisation qui n'est une instance d'un patron. En ce qui concerne la grande partie de ces travaux, la détection n'identifie qu'un aspect des anomalies d'usage. Or, vu la variété d'API existante, le nombre de types d'erreur d'utilisation d'API est grand. Tout ça montre la complexité du problème. L'utilisation des APIs à risque fait partie de la sécurité. Les techniques de détection sont donc majoritairement basées sur l'étude des communications. Même si les études montrent que le fait d'utiliser des APIs cryptographiques ne prémunit pas des risques de sécurité, et montre à quel point il est important d'utiliser une API correctement.

2.3.3. Aide à la correction des erreurs

Nguyen et al. [40] dans leur article présentent LibSync, un outil qui guide les développeurs dans l'adaptation du code d'utilisation de l'API en apprenant des modèles d'adaptation d'utilisation d'API complexes à partir d'autres clients qui ont déjà migré vers une nouvelle version de l'API. LibSync utilise plusieurs techniques basées sur les graphes pour (1) identifier les modifications apportées aux déclarations en comparant deux versions de l'API, pour (2) extraire les squelettes d'utilisation de l'API avant et après la migration de cette dernière et pour (3) comparer ces squelettes. En utilisant les modèles appris. LibSync recommande les emplacements et les opérations d'édition pour adapter les usages de l'API. Comme cette approche, d'autres travaux ont été menés dans le but d'aider à la correction des erreurs.

Wang et Godfrey [59] se sont eux mis à étudier les interfaces de programmation mobiles. Ils ont étudié les postes de *Stack Overflow* et ont plus particulièrement extrait les cas concernant les APIs d'IOS et Android. Ils ont découvert que les postes concernent majoritairement les obstacles d'utilisation d'API et se sont aperçus que certaines classes sont vraiment problématiques. Les auteurs ont donc extrait des postes identifiés avec l'usage des APIs en question des scénarios qui se répètent lorsque les obstacles sont présents. Leur contribution consiste donc à mieux comprendre les problèmes rencontrés par les développeurs en extrayant des modèles d'utilisation des postes pour suggérer des améliorations de l'utilisation de l'API. Et plus particulièrement ce qui concerne les patterns qui causent des problèmes aux

développeurs en les relevant pour pouvoir les étudier et les expliquer. Ici, l'identification des problèmes se fait via les questions posées par la communauté.

2.3.3.1. *L'ensemble de données d'erreurs d'utilisation*

Amann et al. [5] présentent dans leur article `MUBench` un ensemble de données de 89 erreurs d'utilisation d'API qui ont été collectées à partir de 33 projets réels et d'un sondage. Avec l'ensemble de données, les auteurs ont analysé empiriquement la prévalence des erreurs d'utilisation de l'API par rapport à d'autres types de bugs, en constatant qu'ils sont rares, mais causent presque toujours des incidents. Ils ont aussi défini 13 types différents d'erreurs d'utilisation pour pouvoir contribuer à cet ensemble de données. `MUBench` est open source ce qui permet à la communauté scientifique de continuer à contribuer à son enrichissement. De plus, cet ensemble de données est considéré comme la référence en son genre et peut permettre de se comparer à d'autres outils de détection d'erreur d'utilisation d'API.

2.3.3.2. *Le typage fort de JavaScript*

Liu et al. [32] proposent une approche pour renforcer l'utilisation correcte des APIs de Java gérant la concurrence. Leur technique se découpe en 2 parties. La première est d'annoter les méthodes grâce à la documentation. La seconde consiste à utiliser ces annotations pour détecter les erreurs d'utilisation grâce un vérificateur de type léger. Dans un premier temps, ils extraient des annotations de la documentation de l'API en utilisant des techniques de traitement du langage naturel. Ensuite, ils ont implémenté leurs vérificateurs de type dans le `Checker Framework` pour détecter les abus. Ils appliquent leur approche pour extraire des annotations pour toutes les classes de la bibliothèque standard Java et les utiliser pour détecter les abus d'API de concurrence dans des projets Open Source sur *GitHub*. Les auteurs confirment que la mauvaise utilisation d'API de concurrence Java est fréquente et entraîne souvent des bogues ou une inefficacité. Cette façon de procéder fait penser aux travaux menés sur le langage JavaScript. Ce langage étant plus permissif que Java les chercheurs ont décidé de plus le contraindre et y ajouter des annotations.

Santos et al. [21] présentent `JaVerT`, une chaîne d'outils de vérification JavaScript semi-automatique, basée sur la logique de séparation et destinée aux développeurs souhaitant disposer de spécifications riches et vérifier mécaniquement du code JavaScript critique. Pour spécifier les programmes JavaScript, ils ont conçu des abstractions qui capturent ses structures clés (par exemple, des chaînes prototypes et des fermetures de fonctions), permettant au développeur d'écrire des spécifications claires et succinctes avec une connaissance minimale des composants internes JavaScript. `JaVerT` est un pipeline de vérification de programmes JavaScript composé d'un compilateur, d'un langage intermédiaire capturant les caractéristiques dynamiques fondamentales de JavaScript, un outil de vérification semi-automatique basé sur une logique de séparation, et un vérificateur de spécifications axiomatiques des

fonctions internes JavaScript. À l'aide de `JaVerT`, les propriétés de correction fonctionnelle des bibliothèques de structure de données (carte de valeurs-clés, file d'attente prioritaire) écrites dans un style orienté objet sont vérifiées, ainsi que les opérations sur les structures de données telles que les arbres de recherche binaires (BST) et les listes.

Park [44] propose une nouvelle approche pour détecter les erreurs d'utilisation de bibliothèques JavaScript en utilisant deux outils distincts, `DefinitelyTyped` et `SAFE`. Il construit un outil pour analyser les déclarations *TypeScript* des bibliothèques JavaScript de `DefinitelyTyped` pour les stocker dans des fichiers de base de données. Park étend `SAFE` pour reconnaître les informations de type à partir des déclarations *TypeScript* préanalysées dans les fichiers de base de données et les ajouter dans les informations initiales pour l'analyse de programme JavaScript. Son outil parcourt les programmes JavaScript pour trouver les utilisations de l'API afin de vérifier s'ils l'utilisent correctement et signale les erreurs pour les mauvaises utilisations.

Bae et al. [7] ont mis au point `SAFEWapi`, un outil pour analyser les API Web et les applications Web JavaScript qui utilisent les API Web et pour détecter les erreurs d'utilisation possibles des APIs Web par les applications web. Même si la sémantique du langage JavaScript permet d'appeler une fonction définie avec certains paramètres sans aucun argument, les développeurs de plateforme peuvent demander aux développeurs d'application de fournir le nombre exact d'arguments. Étant donné que les fonctions de la bibliothèque dans les API Web exposent clairement la sémantique voulue aux développeurs d'applications Web, contrairement aux fonctions purement JavaScript, les auteurs peuvent détecter les utilisations incorrectes des API Web avec précision. Quant à Zhang [64], il propose d'étendre le langage *WebIDL* pour avoir des performances plus élevées que `SAFEWapi`. Son approche permet de vérifier en profondeur et de façon modulaire les erreurs d'utilisation d'API grâce à un système d'annotations.

2.4. Synthèse

Nous avons vu au travers de ce chapitre les définitions permettant de comprendre dans quel contexte l'approche développée dans ce mémoire a lieu. Ensuite, nous avons fait l'état de l'art, et vu que la communauté scientifique qui aborde le sujet des erreurs d'utilisation d'API a réalisé des travaux très variés. Nous avons vu dans un premier temps, les travaux qui concernent l'amont du problème et plus particulièrement ce qui concerne le binôme documentation et exemples. Comme le révèlent les différentes études [3, 9, 36, 47], il y a des lacunes en termes d'information, et cela fait barrage au bon apprentissage de l'utilisation d'une API. Donc différents travaux se sont attelés à enrichir d'une part la documentation et à permettre l'accès à de plus nombreux exemples, plus simples et surtout de qualité. Ces

travaux se basent les projets Open Source et les sites de Q&A comme *Stack Overflow* pour fournir des données optimales.

Ensuite, nous avons vu qu'un autre pan de la recherche a considéré ce problème des anomalies d'utilisation du point de vue de l'aval, en mettant au point différentes approches pour détecter ces erreurs d'utilisation. Le problème étant leur variété. Il est difficile de produire une approche qui est capable de détecter tous les types d'erreurs. On remarque aussi qu'une des approches couramment utilisées emploie les patrons d'utilisation pour comparer les méthodes étudiées aux patterns relevés et déclarer comme anomalie les usages qui s'éloignent de ces patrons. Nous avons le tableau 2. I qui est un récapitulatif de la douzaine d'approches relevées visant à détecter les erreurs d'utilisation d'API. On remarque que dans la majorité des cas les approches ne sont pas spécifiques à une API, de plus l'apprentissage est dépendant des clients étudiés et le type d'erreur détecté n'est pas varié. C'est de ce constat que nous sommes partis pour élaborer APIMMUNE.

Enfin, pour finir, nous avons vu que certains chercheurs proposent des solutions pour aider à la correction des erreurs. Celles qui nous intéressent le plus étant l'ensemble de données `MUBench` [5] qui est un ensemble de données Open Source faisant référence en termes d'ensemble de données pour les erreurs d'utilisation Java.

Outil	Méthode de détection	Mono API	Apprentissage	Type d'erreur détecté
Alattin	Mine des patterns alternatifs pour la vérification de conditions. Exploration des éléments fréquents sur l'ensemble des règles de vérification avant et après la condition, sur le récepteur, les arguments, et la valeur de retour d'un appel de méthode.	NON	À partir d'exemples extraits grâce aux CSE (code search engines).	Vérifications de précondition de valeur nulle ou manquante, conditions d'état qui sont assurées par vérification.
AX09	Vérification de modèle pour générer des chemins de gestion d'erreur en tant que séquences d'appels de méthode. Exploration de sous-séquence fréquente pour détecter des patrons.	NON	Sur le client lui-même.	Manipulation manquante des erreurs.
BIGROUM	Extraction de patterns fréquents présents dans le client et rassemblement des utilisations pour classement des groupes du plus populaire au plus anormal.	NON	Sur le client lui-même.	
CAR-Miner	Extraction de la séquence d'appel normale et la séquence d'appel d'exception pour un appel à une méthode. Apprentissage des règles d'association pour déterminer quelle exception est sensée être utilisée.	NON	Apprend sur l'application elle-même	Manipulation manquante des erreurs.
Chronicler	Détection des patrons dans les ordres d'appels les plus fréquents à l'intérieur des graphes de flux de contrôle interprocéduraux. Si moins de 80% de l'ordre d'appel respecté, la méthode testée est considérée comme une violation.	NON	Sur le client lui-même.	Appels manquants
CLEVER	Comparaison des commits aux mauvais commits déjà relevés.	NON	A partir des cas passés en revue	
CRYPTOLINT	Relève les violations de 6 règles statiques dans les graphes intra-procédural de flux de contrôle.	crypto	Aucun, règles statiques	Erreurs cryptographiques spécifiques.
DMMC	Considération d'une liste d'appels de méthode sur une variable d'un type donné. Attribution d'un score d'étrangeté par rapport à la majorité.	NON	A partir de byte code Java.	Appels manquants.
DroidAssist	Apprentissage des séquences d'appels de méthodes pour construire un modèle de Markov caché. Si une probabilité d'une séquence d'appels donnée est trop petite alors c'est erreur d'utilisation	NON	Ensemble d'entraînement de byte code	
Jadet	Comparaison des patrons d'utilisation en appairant des appels de méthodes de l'API.	NON	Sur le même client.	Appels de méthodes, boucles manquants.
PR-Miner	Extraction des ensembles d'éléments fréquemment appelés ensembles pour en déduire des règles de programmation. Leur violation est une erreur d'utilisation.	NON	Sur le client lui-même.	Appels manquants.
Tikanga	Extraction des CTL à partir des objets. Vérification des patterns pour détection.	NON	Sur le client lui-même.	Préconditions manquantes.

Tableau 2. I. Comparatif des différents outils de détection d'erreur d'utilisation d'API

Chapitre 3

FORMULATION ET APERÇU DE L'APPROCHE

Dans ce court chapitre, nous allons introduire la formulation utilisée tout au long de ce mémoire et voir comment la passerelle se fait entre la détection des utilisations à risque des interfaces de programmation et le système immunitaire biologique. Ensuite, nous donnerons un aperçu global de l'approche qui se décline sous forme de deux grandes étapes : (1) la génération des détecteurs et (2) la détection proprement dite. Les détails de ces étapes seront expliqués respectivement dans les chapitres pour mieux comprendre comment elle sera détaillée dans les prochains chapitres 4 et 5.

3.1. Formulation du lien entre la détection des erreurs d'utilisation et le système immunitaire

Comme nous l'avons vu dans l'état de l'art, le problème des approches actuelles c'est qu'elles ne sont pas spécifiques à une API, elles sont souvent dépendantes du client sur lequel elles sont utilisées et ne détectent principalement qu'un seul type d'erreur d'utilisation. C'est à partir de ce constat que nous avons eu l'idée de développer notre approche APImmune.

Nous avons vu que le système immunitaire est un moyen très puissant et efficace pour la détection. De plus, ce système est capable de gérer la détection de problèmes variés. C'est pour cela que l'on peut considérer que la détection d'utilisation à risque d'API est très semblable à la détection de pathogènes dans l'organisme. Lorsque l'on fait l'analogie entre une approche spécifique et le système immunitaire artificiel, nous avons vu dans la Section 2.1.2 qu'il y a 5 rôles à remplir. Pour chacun des rôles nous auront :

- **Le corps** : C'est l'ensemble des clients utilisant l'API étudiée. C'est donc du code.
- **Les détecteurs** : Ce sont des API *groum*s (graphe d'utilisation de l'API) dérivés des méthodes clients. Ce sont des mutations de bonnes utilisations de l'API. Ces mutations ayant pour but de transformer une bonne utilisation en mauvaise utilisation.
- **Les antigènes *self*** : Ce sont des API *groum*s qui servent d'exemple d'apprentissage en tant que bonne utilisation. C'est à la base du code jugé sain.

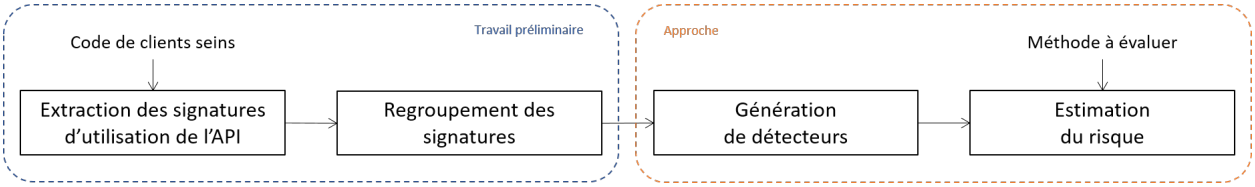

FIGURE 3.1. Approche globale

- **Les antigènes *non-self*** : Ce sont des API *groum*s d'utilisation à risque de l'API étudiée. C'est du code jugé à risque.
- **L'affinité** : La similarité entre un détecteur et un API *groum* extrait d'une méthode utilisant l'API.

Nous avons décidé d'adopter le système immunitaire artificiel dans notre approche car il est plus rapide de tester la similarité avec un petit ensemble de détecteurs que tous les bons cas de bonnes utilisations relevés. L'ensemble de détecteurs au sein de l'organisme est normalement conséquent mais pour des raisons computationnelles nous ne pouvons générer infiniment des détecteurs. D'une part, car ce processus de génération de détecteurs est complexe en termes de temps et de mémoire. D'autre part, lors de la phase de détection, plus l'ensemble de détecteurs est petit, plus la comparaison d'une méthode avec ces détecteurs se fait rapidement. Autre fait important, relever de bons cas d'utilisation s'avère difficile. Or, ces cas sont essentiels à l'approche, ils représentent les antigènes *self* et permettent la génération des détecteurs. Le système immunitaire artificiel est un processus d'apprentissage par la foule, donc plus il y a d'exemples fournis en entrée plus l'apprentissage est efficace.

3.2. APERÇU DE L'APPROCHE

Comme le montre la Figure 3.1 l'approche APIMMUNE se compose de quatre étapes principales. On peut regrouper ces étapes deux par deux, dans ce que l'on peut appeler le travail préliminaire et l'approche générale.

Le travail préliminaire consiste à extraire les signatures des utilisations de l'API au sein des clients relevés pour l'apprentissage. Cette extraction consiste à passer du code source de chaque méthode des clients jugés sains à un graphe d'utilisation de l'API (nommé *API groum*) pour chacune des méthodes des clients. Ce graphe permet d'avoir une représentation de l'utilisation de l'API en y distinguant l'ordre des appels et les dépendances de données. La seconde étape du travail préliminaire consiste à regrouper ces signatures extraites en fonction de leur utilisation de l'API. Ainsi nous obtenons l'ensemble des différents usages de l'API. Cet ensemble des différents bons usages de l'API est appelé l'*ensemble d'apprentissage*.

En ce qui concerne l'approche générale, il y a d'abord la génération des détecteurs, car comme nous l'avons vu le système immunitaire fonctionne grâce à un ensemble de détecteurs. Ces détecteurs sont générés grâce aux différents groupes d'utilisation de l'API relevés

en mutant une bonne utilisation de l'*ensemble d'apprentissage* pour qu'elle devienne potentiellement mauvaise et puisse matcher avec d'autres mauvaises utilisations. La dernière étape de l'approche consiste à estimer le score de risque de la méthode évaluée en la comparant à l'ensemble des détecteurs générés. Cet ensemble de détecteurs est appelé la solution finale.

Les deux prochains chapitres concerneront ces deux phases de l'approche. Nous détaillerons les différentes étapes réalisées lors du travail préliminaire afin d'optimiser l'approche générale qui est décrite dans le chapitre qui suit. Dans ce chapitre concernant l'approche, nous détaillerons les différents processus pour générer les détecteurs et comment le score de risque est assigné aux méthodes évaluées.

Chapitre 4

EXTRACTION ET TRAITEMENT DES SIGNATURES D'UTILISATION DES APIS

Nous venons de voir dans le chapitre précédent que APIMMUNE procède à quatre étapes (cf. Figure 3.1). Dans ce chapitre, nous allons détailler les deux premières étapes et ce que l'on peut considérer comme le travail préliminaire. Dans un premier temps, nous allons parler de la structure de données qui est utilisée pour représenter les utilisations de l'API. C'est grâce aux *groum*s que nous sommes capables d'extraire les signatures des usages d'une interface de programmation. Ensuite nous allons décrire les trois APIs considérées au sein de ce mémoire avant de spécifier l'étape décrire l'étape de regroupement des signatures des utilisations de l'API. Pour finir, nous expliquerons le travail d'analyse de données réalisé sur les signatures extraites pour mieux comprendre le problème de détection des erreurs d'utilisation des interfaces de programmation.

4.1. Génération des groums

Pour débuter nous allons expliquer ce qu'est un *groum*, d'où vient cette structure de données et en quoi elle s'avère très utile pour notre problématique de détection d'utilisation à risque des APIs. Nous détaillerons les améliorations apportées à cette structure de données afin qu'elle s'adapte mieux à notre problème. Et finalement nous verrons comment le passage d'un *groum* à un *API groum* se fait afin d'obtenir seulement une représentation de l'utilisation de l'API par une méthode et non plus la signature complète de la méthode.

4.1.1. Structure de données

Afin de faire l'analogie entre le code et les cellules de l'organisme, nous avons choisi une structure de données nommée *groum*. Un *groum* est un graphe acyclique orienté permettant de représenter l'ordre d'appel des fonctions et les dépendances de données. Cela signifie qu'il ne possède aucune boucle et que toutes les arêtes possèdent une direction. Le *groum* est tiré du travail de Nguyen et al [42]. L'outil `GROUMINER` permet de passer le code d'une

méthode sous forme de graphe, plus précisément de *groum*. Une lecture de l'arbre syntaxique de chaque méthode du client étudié est faite afin de créer chaque nœud et chaque arrête. Ce format de données se révèle intéressant car il nous permet à la fois de voir les dépendances de données et surtout, il permet d'extraire l'ordre des appels des méthodes, en particulier celles de l'API étudiée. Cependant, l'outil `GROUMINER` a été modifié afin d'obtenir des *groum*s plus spécifiques à notre problématique.

Chaque nœud d'un *groum* représente l'appel d'une méthode ou d'une structure de contrôle tel qu'un if, for, while... Ce nœud est ensuite relié aux autres nœuds du graphe par des arêtes orientées. Afin de mieux comprendre l'utilisation de l'API, nous avons décidé de typer les arêtes. Quatre différents types d'arêtes ont donc été créés.

— Les arêtes de type IN qui signifient que l'appel de la fonction ou la structure de contrôle est réalisé à l'intérieur d'une autre méthode ou structure de contrôle. On peut considérer que cette arête permet de représenter ce qu'il se situe entre les parenthèses d'un appel. Les arêtes IN sont représentées de couleur rouge.

— Les arêtes de type INSIDE marquent le lien entre une structure de contrôle et un autre nœud. Ce type d'arête permet de délimiter le début du périmètre d'une structure de contrôle. Cela correspond à relier la structure de contrôle avec le premier appel fait après l'ouverture de l'accolade. Les arêtes INSIDE sont représentées de couleur orange.

— Les arêtes de type FOLLOW permettent de suivre l'ordre d'exécution des appels et clore le périmètre d'une structure de contrôle. Un nœud ne possède qu'une arête FOLLOW sortante mais peut en posséder plusieurs entrantes. Les arêtes FOLLOW sont représentées de couleur verte.

— Les arêtes de type DATA montrent les différentes dépendances de données entre les nœuds. Ce sont les arêtes les plus présentes au sein d'un *groum*. Les arêtes DATA sont représentées de couleur bleue.

Le typage des arêtes ne fait pas partie de l'outil de base. Nous avons décidé de l'ajouter pour mieux répondre aux problèmes posés par notre approche. Les erreurs d'utilisations étant majoritairement dues à des erreurs dans les appels des méthodes, ou un manque de vérification de pré-condition. Typer les arêtes du *groum* permet de mieux voir l'ordre d'appel des méthodes, d'une part, et d'autre part, d'identifier le périmètre d'application d'une structure de contrôle. L'arête IN permet de savoir si une donnée est utilisée lors de l'appel de la structure de contrôle et les arêtes FOLLOW et INSIDE rendent possible la distinction du champ d'application de la structure de contrôle en question (on sait quels appels ont lieu entre les accolades). Un exemple de *groum* simplifié aux arêtes typées est illustré figure 4.2 et représente le code 4.1.

4.1.2. Simplification de *groum*

De base, l'outil `GROUMINER` extrait sous forme de *groum* toutes les composantes de la méthode donnée en entrée. C'est-à-dire, toutes les méthodes et toutes les structures de contrôle. Nous avons choisi cette structure de données car elle est simple. Cependant, nous voulons qu'elle nous permette d'exprimer seulement l'utilisation de l'API.

Pour ce faire, nous avons décidé de supprimer du *groum* originel, les informations qui nous paraissaient superflues. Dans un premier temps, nous générons le *groum* originel de la méthode donnée en entrée. Ensuite, on enlève tous les nœuds dont les méthodes ne font pas partie de l'API et toutes les structures de contrôle n'étant pas reliées aux méthodes de l'API. Ainsi nous obtenons seulement l'ordre et la structure des appels à l'API. De ce fait, nous avons un *groum* exprimant la façon dont est utilisée l'API sans informations jugées superflues. On appelle ce *groum* simplifié un *API groum*.

Lorsque l'on supprime un nœud, ses arêtes doivent être supprimées, mais il faut procéder au rattachement des nœuds qui avaient en source ou en destination cette entité qui va être ôtée du graphe. Étant donné que les arêtes sont désormais typées, cette opération de suppression peut entraîner des changements de types des arêtes, des nœuds adjacents au nœud en cours de suppression. Le tableau 4. I établit les règles utilisées pour modifier le type de chaque arête en fonction du type d'arête d'entrée et de sortie. En tout, il y a cinq possibilités d'arête d'entrée et de sortie mais seulement trois sont exprimées dans le tableau. Nous n'avons pas mis dans le tableau les cas où l'arête de sortie ou d'entrée est une arête de type DATA ou NUL car elle sera tout simplement supprimée. C'est pour cela qu'il y a trois cas d'entrée (FOLLOW, INSIDE et IN). Cela s'explique car il ne peut y avoir qu'une seule arête de qui peut provenir d'un nœud source. En revanche pour les sorties il y a plus de configurations. Le cas où il y a en sortie aucune arête IN, une seule ou plusieurs, car une méthode ou une variable peut être utilisée plusieurs fois. Ensuite, les cas des arêtes INSIDE et FOLLOW sont traités comme des booléens car on ne peut pas avoir deux arêtes FOLLOW ou deux arêtes INSIDE en entrée, mais il est possible d'en avoir une de chaque. Par exemple, une condition `if` qui fait appel à une méthode dans son périmètre, elle est reliée avec une arête INSIDE et lorsque les accolades du `if` se ferment elle est suivie par un appel à une seconde méthode reliée avec une arête FOLLOW le cas échéant, comme on peut le voir avec le nœud *CONTROL.IF* de la figure 4.2 qui possède une arête FOLLOW pointant vers le nœud *Iterator.hasNext* et une arête INSIDE pointant vers *Iterator.remove*.

Le tableau 4. I permet de savoir par quel type d'arête les nœuds liés par le nœud en cours de suppression vont être reliés. Il y a donc le nœud que l'on appelle *source*, celui qui est relié par une arête entrante au nœud en cours de suppression, et le nœud appelé *cible*, relié par une arête sortante à ce même nœud. En fonction de la combinaison entrée / sortie, une arête est créée entre le nœud *source* et le nœud *cible* pour remplacer les arêtes entre le nœud *source* et

le nœud à supprimer et celle entre le nœud à supprimer et le nœud *cible*. Pour chaque cellule du tableau, on considère le remplacement de l'arête *source* par la nouvelle arête. Par exemple si nous avons une sortie IN et INSIDE (IN == 0, I == 1 et F == 1) et une entrée IN. On a la règle suivante `I → F F → ∅` . Ce cas revient à supprimer le nœud *CONTROL.IF* de la figure 4.2 qui possède bel et bien en entrée une arête IN et en sortie une arête INSIDE et une arête FOLLOW. Supprimer ce nœud supprime la structure de contrôle et on peut imaginer que le code de *Iterator.next* précède désormais directement *Iterator.remove* et n'a plus de lien avec *Iterator.hasNext* car il possède déjà une arête FOLLOW en sortie et un nœud ne peut avoir plus d'une arête FOLLOW en sortie cela signifierait que l'appel est suivi directement par plusieurs appels, ce qui est impossible car le code est considéré comme étant une suite d'appel ligne par ligne.

Ces règles ont été établies pour conserver l'information extraite du *groum* au sein d'un *API groum*. Ce typage des arêtes a été ajouté pour mieux détecter les erreurs d'utilisation d'API. En particulier au niveau des structures de contrôle où l'arête IN permet de savoir ce qui se situe au sein de la structure (entre les parenthèses), les arêtes INSIDE ce qui est dans le périmètre de la structure (ce qui est entre les accolades), et les arêtes FOLLOW ce qui est appelé après la structure (après les accolades). Il est très important de conserver une arête INSIDE qui par exemple relie une structure de contrôle `if` avec une méthode ne faisant pas partie de l'API, puis, après cette méthode d'appeler une méthode de l'API toujours dans le périmètre du `if`. Donc, lors de la simplification, le nœud de la méthode qui ne fait pas partie de l'API est supprimé et une arête INSIDE est créée entre le `if` et la méthode de l'API pour remplacer l'arête INSIDE d'entrée et l'arête FOLLOW de sortie.

4.2. APIs étudiées

Comme précisé dans le chapitre 2, les APIs étudiées sont de tailles différentes. Nous avons donc étudié trois APIs Java distinctes et populaires.

1. `java.util.Iterator`[1]. Cette API fait partie des plus utilisées de la communauté Java. Par conséquent, il y a une multitude de projets comportant des fragments de codes pouvant servir d'exemple. Sa popularité permet de récupérer des exemples d'utilisateurs très variés. Mais on peut supposer qu'une forte popularité facilite l'apprentissage de celle-ci et que nous ferons moins face à des erreurs. L'API possède seulement quatre méthodes publiques. Ce facteur nous permet de penser qu'il y a très peu de types différents d'utilisation de cette API. Les utilisations à risque pourraient être rares. `Iterator` est utilisée, comme son nom l'indique, pour itérer sur les objets itérables.

1. https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html (accédé le 10 avril 2018)

Sortie \ Entrée			FOLLOW (F)	INSIDE (I)	IN (IN)
IN == 0	I == 0	F == 0	$\varnothing$	$\varnothing$	$\varnothing$
		F == 1	F $\rightarrow$ F	F $\rightarrow$ I	F $\rightarrow$ F
	I == 1	F == 0	I $\rightarrow$ F	I $\rightarrow$ I	I $\rightarrow$ F
		F == 1	I $\rightarrow$ F F $\rightarrow$ $\varnothing$	I $\rightarrow$ I F $\rightarrow$ $\varnothing$	I $\rightarrow$ F F $\rightarrow$ $\varnothing$
IN == 1	I == 0	F == 0	IN $\rightarrow$ F	IN $\rightarrow$ I	IN $\rightarrow$ IN
		F == 1	IN $\rightarrow$ $\varnothing$ F $\rightarrow$ F	IN $\rightarrow$ $\varnothing$ F $\rightarrow$ I	IN $\rightarrow$ IN F $\rightarrow$ F
	I == 1	F == 0	IN $\rightarrow$ $\varnothing$ I $\rightarrow$ F	IN $\rightarrow$ $\varnothing$ I $\rightarrow$ I	IN $\rightarrow$ IN I $\rightarrow$ F
		F == 1	IN $\rightarrow$ $\varnothing$ I $\rightarrow$ F F $\rightarrow$ $\varnothing$	IN $\rightarrow$ $\varnothing$ I $\rightarrow$ I F $\rightarrow$ $\varnothing$	IN $\rightarrow$ IN I $\rightarrow$ F F $\rightarrow$ $\varnothing$
IN > 1	I == 0	F == 0	IN $\rightarrow$ F IN $\rightarrow$ $\varnothing$	IN $\rightarrow$ $\varnothing$ IN $\rightarrow$ I	IN $\rightarrow$ IN
		F == 1	IN $\rightarrow$ $\varnothing$ F $\rightarrow$ F	IN $\rightarrow$ $\varnothing$ F $\rightarrow$ I	IN $\rightarrow$ IN F $\rightarrow$ IN
	I == 1	F == 0	IN $\rightarrow$ $\varnothing$ I $\rightarrow$ F	IN $\rightarrow$ $\varnothing$ I $\rightarrow$ I	IN $\rightarrow$ IN I $\rightarrow$ F
		F == 1	IN $\rightarrow$ $\varnothing$ I $\rightarrow$ F F $\rightarrow$ $\varnothing$	IN $\rightarrow$ $\varnothing$ I $\rightarrow$ I F $\rightarrow$ $\varnothing$	IN $\rightarrow$ IN I $\rightarrow$ F F $\rightarrow$ $\varnothing$

TABLEAU 4. I. Règle de remplacement des arêtes lors de la simplification

2. `javax.crypto`[2]. Cette API est un package Java comportant 10 classes. On peut donc en conclure qu'il existe de multiples cas d'utilisation de cette API. Ce package est très populaire chez les développeurs Java pour tout ce qui est chiffrement de données [**18**, **36**, **52**]. Il fait partie des références, surtout dans les différentes études d'erreurs d'utilisation. Le fait que ce soit un package, laisse à penser que de multiples utilisations à risque existent et qu'apprendre à utiliser cette API peut se révéler complexe. On peut supposer y trouver par conséquent plus d'utilisations à risque.

3. `javax.servlet.http`[3]. Cette API est elle aussi un package de 7 classes. Elle possède par conséquent plusieurs utilisations. Cette API permet la communication sous le protocole http qui est un des protocoles d'échange les plus utilisés.

2. https://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html (accédé le 10 avril 2018)

3. https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/package-summary.html (accédé le 10 avril 2018)

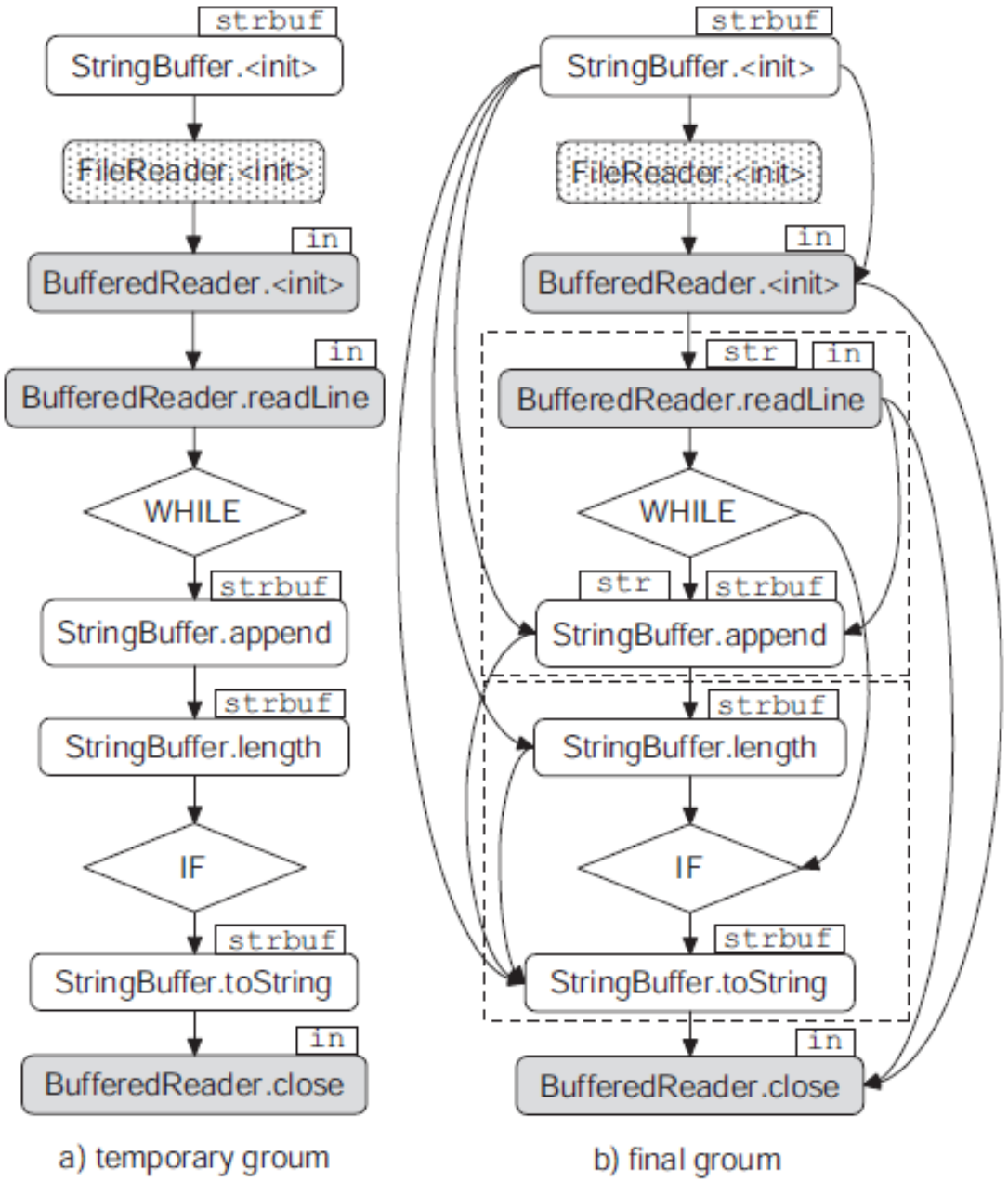

FIGURE 4.1. Exemple de *groum* avant typage des arêtes [42]

4.3. REGROUPEMENT (CLUSTERING)

Afin d'obtenir un meilleur ensemble de détecteurs, nous avons décidé de regrouper les cas d'utilisation de l'API comme cela a été fait par Saied et al. [48]. Pour procéder au clustering nous regroupons chaque méthode sous forme de vecteur. Les différentes colonnes du vecteur représentent l'utilisation d'une méthode de l'API ou non (1 si elle est présente et 0 sinon) par une méthode client. Ainsi nous avons une matrice de vecteurs (Tableau 4. II) permettant

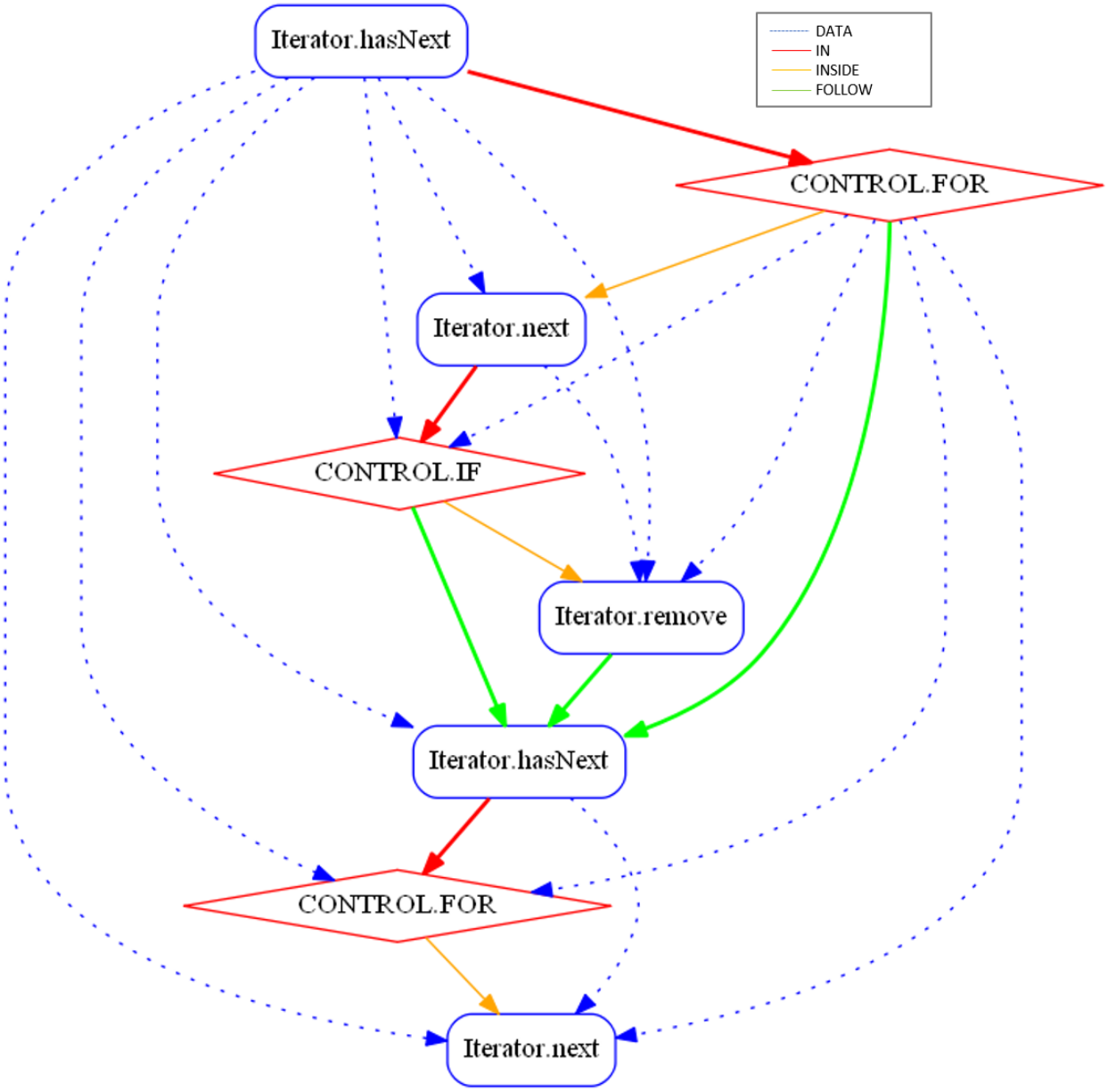

FIGURE 4.2. Exemple du *groum* de la méthode `execute` (Code 4.1) avec typage des arêtes.

de visionner quelle utilisation de l'API est faite. Nous mettons sous forme de vecteur les méthodes afin de pouvoir les donner en entrée à l'algorithme de clustering DBSCAN.

4.3.1. Dbscan

Le regroupement ou clustering de différentes utilisations de l'API est fait grâce à SPMF [4] – dbscan [19] qui nécessite de spécifier deux paramètres : le *minimum de points* par cluster et *Epsilon*, la distance minimale entre chaque paire de points dans le cluster. Comme entrée

4. http://www.philippe-fournier-viger.com/spmf/DBScan.php (accédé le 10 avril 2018)

nous devons fournir une matrice de méthodes de l'API utilisées par les méthodes des clients (tableau 4. II). Ces dernières sont représentées par des vecteurs où les 1 correspondent à une utilisation de la méthode de l'API et 0 à l'absence d'utilisation de cette méthode par une méthode client.

	m_{client_1}	m_{client_2}	...	$m_{client_{k-1}}$	m_{client_k}
m_{API_1}	1	0	...	0	1
m_{API2_2}	1	1	...	1	0
...	...	...	...	...	...
m_{API_n}	0	0	...	0	1

TABLEAU 4. II. Matrice de calcul de distance pour dbscan

L'algorithme utilise une fonction pour mesurer la distance entre deux points. La fonction utilisée par défaut est la distance Euclidienne entre vecteurs [17]. Cependant la distance euclidienne a le défaut de donner autant de poids à la présence qu'à l'absence d'une méthode. Or, nous voulons mettre l'emphase sur les méthodes qui utilisent l'API. Donc, dans notre cas nous avons défini notre propre fonction de distance qui prend en entrée deux vecteurs représentant l'utilisation des méthodes de l'API (tableau 4. II) et retourne une distance comprise entre 0 et 1. La distance est le nombre de méthodes client utilisées par la méthode de l'API pour les deux vecteurs, divisé par le nombre total de méthodes du client utilisées. Par exemple, la distance entre m_{API_1} et m_{API_2} est :

$$\frac{\|m_{client_1}\|}{\|\{m_{client_2}, m_{client_1}, \ldots, m_{client_{k-1}}, m_{client_k}\}\|}$$

Après nos expérimentations nous avons défini les paramètres suivants : un minimum de 2 points par cluster et un *Epsilon* de 0,8. Cela signifie que nous considérons deux méthodes de l'API ayant au moins 20% de leurs méthodes clients qui les appellent en commun sont considérée dans le même groupe (cluster).

De ce fait, on regroupe les méthodes de l'API en fonction de leur utilisation par les méthodes des clients. Ensuite, toutes les méthodes des clients sont regroupées en fonction des clusters des méthodes de l'API. Donc une méthode client se retrouve dans les mêmes groupes que les autres méthodes client qui utilisent les mêmes méthodes de l'API. Cela signifie qu'une méthode peut se trouver dans plusieurs groupes à la fois. Si l'on considère la figure 4.3, nous avons une interface de programmation avec 8 méthodes. En fonction des méthodes clients qui appellent ses méthodes de l'API nous obtenons 3 groupes de méthodes de l'API {m1, m8}, {m3, m4, m5 et m7} et {m2, m6}. Ensuite parmi les méthodes clients on regroupe toutes les méthodes clients par groupe en fonction de ces 3 ensembles pour également obtenir 3 ensembles. Ainsi nous avons regroupé les utilisations de l'API et nous pourrons nous en servir avec APIMMUNE.

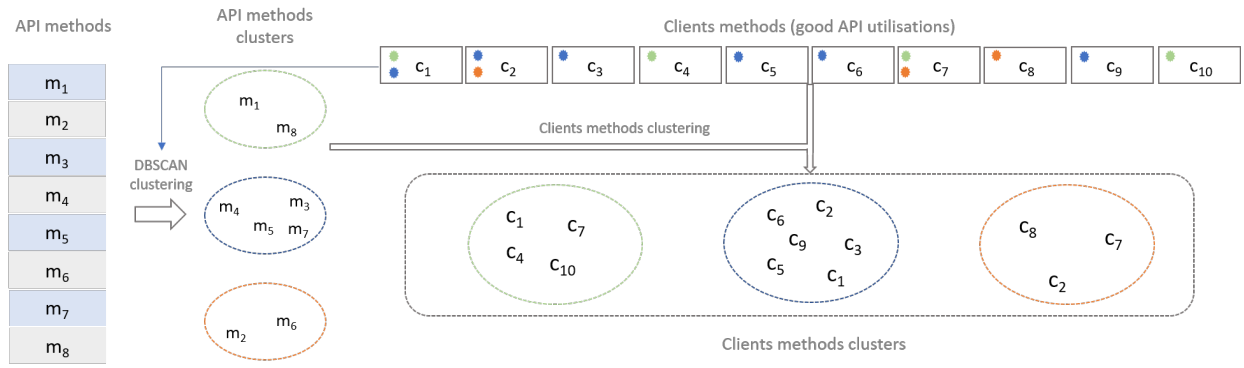

FIGURE 4.3. Regroupement des méthodes d'apprentissage

4.3.2. Formation des groupes

Nous venons de voir comment nous avons classé les méthodes de l'API par groupe en fonction des utilisations d'une part. Et d'autre part nous avons extrait les *API groums* à partir des signatures des méthodes client. Donc maintenant nous pouvons séparer notre *ensemble d'apprentissage*, qui est l'ensemble des *API groums* en plusieurs sous-ensembles en fonction de leur utilisation de l'API. Les *API groums* possédant au moins une méthode d'un groupe d'utilisation sont regroupées dans le même sous-ensemble. Cela veut dire qu'un même *API groum* d'une méthode client peut se retrouver dans plusieurs sous-ensembles différents en fonction des méthodes qui le composent.

Par exemple si l'on regarde la Figure 4.3, les *API groums* des méthodes c1, c4, c7 et c10 seront regroupées dans le même sous-ensemble d'utilisation de l'API. Les *API groums* de c1, c2, c3, c5, c6 et c9 seront dans un second sous-ensemble et ceux de c2, c7 et c8 dans un troisième. Ainsi notre *ensemble d'apprentissage* d'*API groum* qui va nous servir pour la génération des détecteurs regroupe chaque type d'utilisation de l'API et nous verrons lors de la génération ce que ce regroupement peut apporter.

4.4. ANALYSE DE DONNÉES

La prochaine étape de notre approche est la génération des détecteurs. L'objectif de notre approche est d'avoir un nombre limité de détecteurs mais très efficaces. Notre but est donc d'avoir des détecteurs différents les uns des autres mais à une distance "acceptable" des *groums* à partir desquels on apprend. En effet, une erreur d'utilisation peut être l'oubli d'une méthode où le changement d'ordre d'utilisation de deux méthodes, soit, une légère déviance par rapport à une bonne utilisation. Nous avons donc étudié une partie de nos données afin de pouvoir déterminer la distance "acceptable" à laquelle doit se situer un bon détecteur pour détecter une erreur d'utilisation. Ainsi en sachant à quelle distance se situe l'*API groum* d'une mauvaise utilisation par rapport à celui d'une bonne utilisation, nous allons pouvoir optimiser la génération des détecteurs.

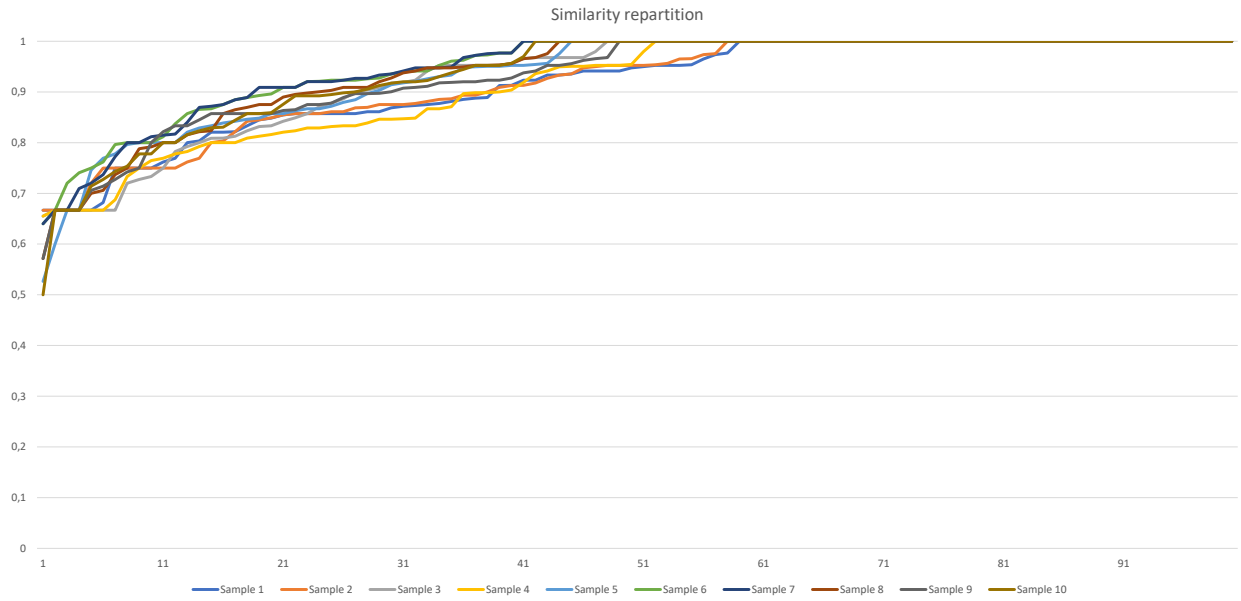

FIGURE 4.4. Similarités maximales entre les bonnes utilisations et 100 mauvaises utilisations

Pour ce faire, nous prenons toutes les méthodes considérées comme bonnes (méthode à un moment `t+1`, une correction) et 100 méthodes considérées comme de mauvaises utilisations (méthode à un moment `t`, une méthode corrigée). Ensuite, nous faisons une matrice de similarité entre chaque bonne et mauvaise utilisation. La similarité est calculée en divisant le nombre de nœuds, d'arêtes et d'exceptions communs entre deux *API groum*s par le nombre total de nœuds, d'arêtes et d'exceptions total. L'opération est répétée dix fois avec 100 mauvais cas différents, sélectionnés aléatoirement. Ainsi nous obtenons une représentation graphique des différences entre mauvaises et bonnes utilisations, afin de déterminer quel intervalle de similarité privilégier pour notre fonction objectif grâce à la figure 4.4. Sur cette figure, on peut observer que chacune des 100 mauvaises utilisations a au moins une similarité de 0,5 avec n'importe quelle bonne utilisation. Ce graphique met en lumière que les erreurs d'utilisation d'API sont proches des bonnes utilisations. Il faudra donc que les détecteurs ne soient pas trop éloignés des bonnes utilisations pour détecter correctement les utilisations à risque.

La figure 4.4 montre aussi qu'il y a une partie des mauvaises utilisations qui sont identiques à une bonne utilisation. Cela ne veut pas dire forcément que ces méthodes qui ont une similarité de 1 sont forcément une correction qui se révèle ne pas en être une. Donc, une autre analyse est faite pour retirer les méthodes dont la bonne et mauvaise utilisation sont identiques. Pour ce faire nous comparons toutes les méthodes à leur instant `t` (mauvaise utilisation) et leur instant `t+1` (bonne utilisation). Si la similarité entre les deux *API groum*s

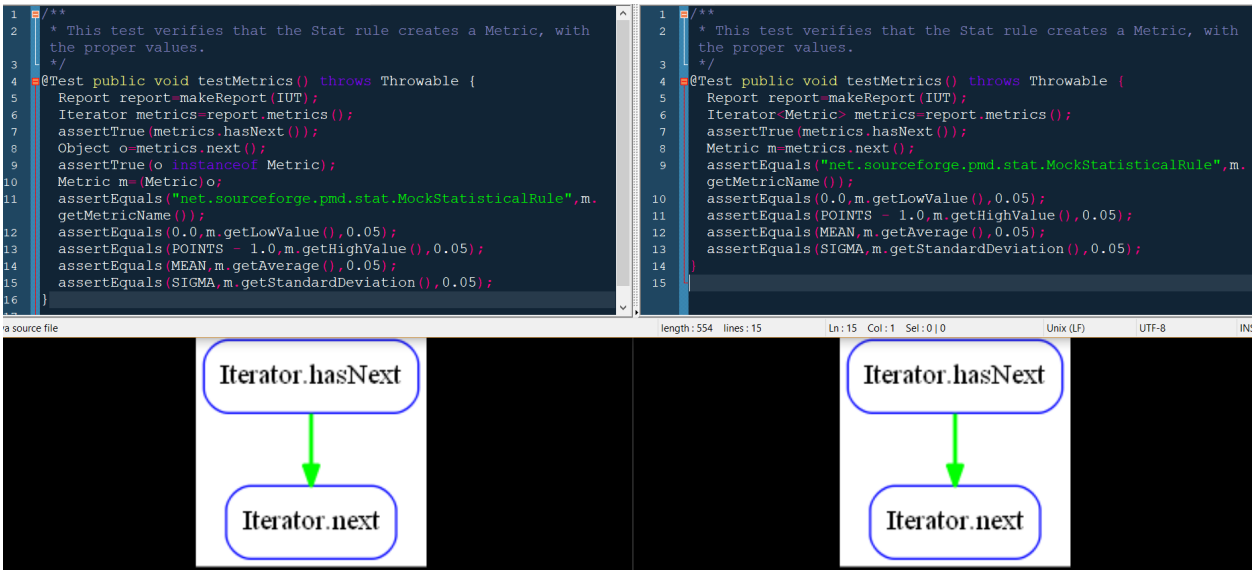

FIGURE 4.5. Méthode à moment `t` et `t+1` avec l'*API groum* correspondant

est de 1, cela signifie que l'utilisation de l'API est identique. Par conséquent, la méthode considérée erronée est retirée de l'ensemble des mauvaises utilisations. Nous obtenons de tels cas lorsque la modification sur la méthode était par exemple un renommage sur la ligne faisant appel à l'API. Ce nettoyage pouvait difficilement être fait pendant l'extraction des données, c'est pour cela que la structure de *groum* s'avère très utile. Cette fonctionnalité n'a pas été exploitée plus en profondeur mais elle pourrait l'être dans des travaux futurs, pour détecter le changement d'utilisation d'une API d'une version du code à l'autre par exemple. La figure 4.5 montre un exemple où une méthode a été modifiée sans pour autant que l'utilisation de l'API soit différente. On peut voir à gauche la méthode commit au moment `t` et à droite la correction, la même méthode modifiée, au moment `t+1`, avec en dessous du code, l'*API groum* représentant l'utilisation de l'API. On remarque que la seule modification d'utilisation de l'API se situe `ligne 8` où l'appel de `Iterator.next` renvoie une `Metric` plutôt qu'un `Object`. L'utilisation d'un *groum* représentant l'utilisation de l'API permet de bien visualiser la similarité des deux utilisations.

```
1  public MavenExecutionResult execute(MavenExecutionRequest request){
2    request.setStartTime(new Date());
3    MavenExecutionResult result=new DefaultMavenExecutionResult();
4    ReactorManager reactorManager=createReactorManager(request,result);
5    if (result.hasExceptions()) {
6      return result;
7    }
8    EventDispatcher dispatcher=new DefaultEventDispatcher(request.
       getEventMonitors());
9    String event=MavenEvents.REACTOR_EXECUTION;
10   dispatcher.dispatchStart(event,request.getBaseDirectory());
11   MavenSession session=createSession(request,reactorManager,dispatcher);
12   for (Iterator i=request.getGoals().iterator(); i.hasNext(); ) {
13     String goal=(String)i.next();
14     if (goal == null) {
15       i.remove();
16       continue;
17     }
18     TaskValidationResult tvr=lifecycleExecutor.isTaskValid(goal,session,
         reactorManager.getTopLevelProject());
19     if (!tvr.isTaskValid()) {
20       result.addBuildFailureException(tvr.generateInvalidTaskException());
21       return result;
22     }
23   }
24   getLogger().info("Scanning␣for␣projects...");
25   if (reactorManager.hasMultipleProjects()) {
26     getLogger().info("Reactor␣build␣order:␣");
27     for (Iterator i=reactorManager.getSortedProjects().iterator(); i.
         hasNext(); ) {
28       MavenProject project=(MavenProject)i.next();
29       getLogger().info("␣␣" + project.getName());
30     }
31   }
32   try {
33     lifecycleExecutor.execute(session,reactorManager,dispatcher);
34   }
35  catch ( LifecycleExecutionException e) {
36     result.addLifecycleExecutionException(e);
37     return result;
38   }
39 catch ( BuildFailureException e) {
40     result.addBuildFailureException(e);
41     return result;
42   }
43   result.setTopologicallySortedProjects(reactorManager.getSortedProjects())
       ;
44   result.setProject(reactorManager.getTopLevelProject());
45   return result;
46 }
```

LISTING 4.1. Méthode execute

Chapitre 5

GÉNÉRATION ET UTILISATION DES DÉTECTEURS DE MAUVAISES UTILISATIONS

Notre approche est basée sur le système immunitaire. Ce système permet de défendre un organisme contre les attaques qu'il subit. Il a pour rôle de reconnaître les cellules étrangères appelées pathogènes. Pour jouer sa partition dans le corps humain, il n'a pas d'organe central. Le système immunitaire agit grâce à un ensemble de détecteurs que sont les lymphocytes et plus particulièrement les lymphocytes T. Ce sont ces lymphocytes T qui vont errer dans l'organisme à la recherche de pathogènes compatibles avec leurs récepteurs. Dans ce chapitre nous allons voir les différents mécanismes qui permettent l'élaboration de détecteurs pour l'identification d'utilisation à risque d'une API.

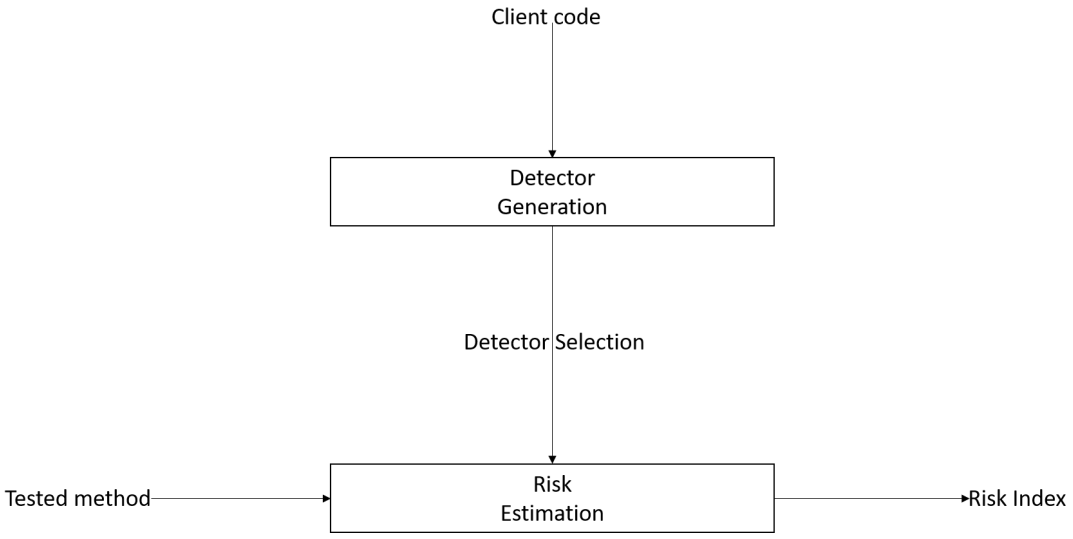

FIGURE 5.1. Approche

5.1. Algorithme génétique

Dans notre approche, les détecteurs sont des mutations des exemples de bonnes utilisations relevées. Notre but est de fournir une solution comportant les meilleurs détecteurs. C'est-à-dire, les détecteurs les plus différents les uns des autres et à une distance acceptable des bonnes méthodes à partir desquelles on apprend. Ceci est calculé par une fonction avec un objectif à deux critères appelée *fitness*. Afin d'obtenir la solution optimale, nous allons utiliser un algorithme génétique. Ce dernier va consister à produire un ensemble de solutions, une solution étant un ensemble de détecteurs. L'algorithme génétique va permettre de mélanger ces différentes solutions grâce à différents mécanismes. Ce brassage est effectué dans l'optique de faire progresser les solutions et générer la solution avec la meilleure diversité possible.

Pour ce faire, il y a un nombre de *groum*s de bonnes utilisations de l'API qui sont sélectionnés aléatoirement, parmi ce que l'on appelle notre *ensemble d'apprentissage*. Ce nombre est la taille de la solution (`SIZE_SOLUTION`). Ensuite, ces *groum*s subissent une mutation afin de devenir détecteur de génération initiale, la *génération 0*. La population initiale générée est un ensemble de solutions comportant toutes le même nombre de détecteurs (`SIZE_SOLUTION`). L'algorithme génétique va consister à appliquer différentes opérations sur ces solutions pour un nombre de générations donné (`GENERATION`).

5.1.1. Élitisme

La première de ces étapes qui compose l'algorithme génétique est l'élitisme. L'élitisme permet de garantir la conservation, en tout temps, de la meilleure des solutions à chaque génération. Une certaine proportion des solutions est ainsi gardée intacte de la génération `g` à la génération `g`+1. Cette proportion `PROP_ELIT` est un paramètre fixé lors de l'exécution de la génération de détecteurs. Pour sélectionner quelles solutions font partie de l'élite, les solutions sont ordonnées par *fitness*. Les solutions ayant la *fitness* la plus élevée sont ainsi gardées, et perdurent au fil des générations, car une modification, ne serait-ce que légère pourrait faire grandement varier leur efficacité. C'est pour cela qu'elles sont conservées intactes.

5.1.2. Cross-over

La seconde étape de cet algorithme génétique est le cross-over. Cette étape va permettre de mixer certaines solutions entre elles pour obtenir un plus grand brassage génétique. La probabilité qu'un cross-over se produise est aussi un paramètre fixé (`PROB_CROSS`).

On commence par sélectionner deux parents grâce à la méthode du **tournoi**. Celle-ci consiste à prendre deux solutions au hasard parmi l'ensemble complet des solutions, y compris les solutions qui ont été sélectionnées parmi l'élite. Même si l'élite est déjà présente dans la génération suivante, une solution peut être utilisée plusieurs fois lors du processus

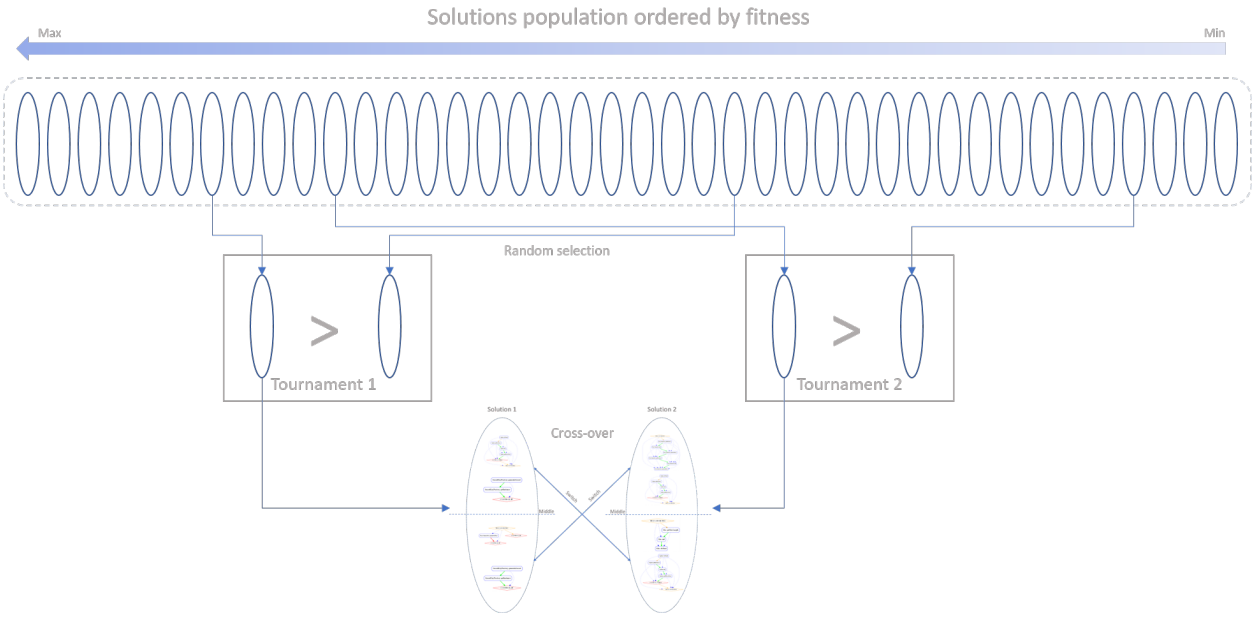

FIGURE 5.2. Sélection par tournoi

de cross-over. Parmi les deux solutions sélectionnées, celle ayant la *fitness* la plus élevée remporte le tournoi et devient par conséquent parent. On répète l'opération afin d'obtenir le deuxième parent. Tout ce processus est illustré dans la figure 5.2.

Les deux parents sélectionnés vont ensuite permuter une partie de leurs détecteurs. Cette permutation peut être à point fixe ou non. Si la permutation est à point fixe, les deux méthodes échangent le même nombre de détecteurs (`SIZE_SOLUTION` divisé par 2) et restent ainsi à une taille similaire. Dans l'autre cas, le nombre de détecteurs à échanger de chaque côté est tiré aléatoirement. On obtient par conséquent des solutions de taille différente. Dans notre approche, nous n'avons considéré que le cross-over par point fixe. La figure 5.3 illustre cet échange d'une moitié de leur détecteur entre les deux solutions désignée comme parent. Le rôle du cross-over est de permettre le brassage génétique. Il va favoriser le changement de détecteurs à l'intérieur des solutions et ainsi permettre la diversification de ces derniers au sein de la solution. Mais il peut aussi avoir l'effet inverse, car il peut apporter un lot de détecteurs déjà présents dans une solution. C'est pour cela que l'élitisme (vu ci-dessus) et la sélection grâce au tournoi sont primordiaux, car l'élitisme va permettre de garder les bons brassages et les solutions avec les meilleures *fitness*. Le tournoi va éviter que l'on sélectionne les méthodes avec de mauvaises *fitness*, tout en gardant une probabilité de les utiliser à nouveau car elles peuvent tout de même contenir un matériel génétique (des détecteurs) très utile pour diversifier encore plus les solutions.

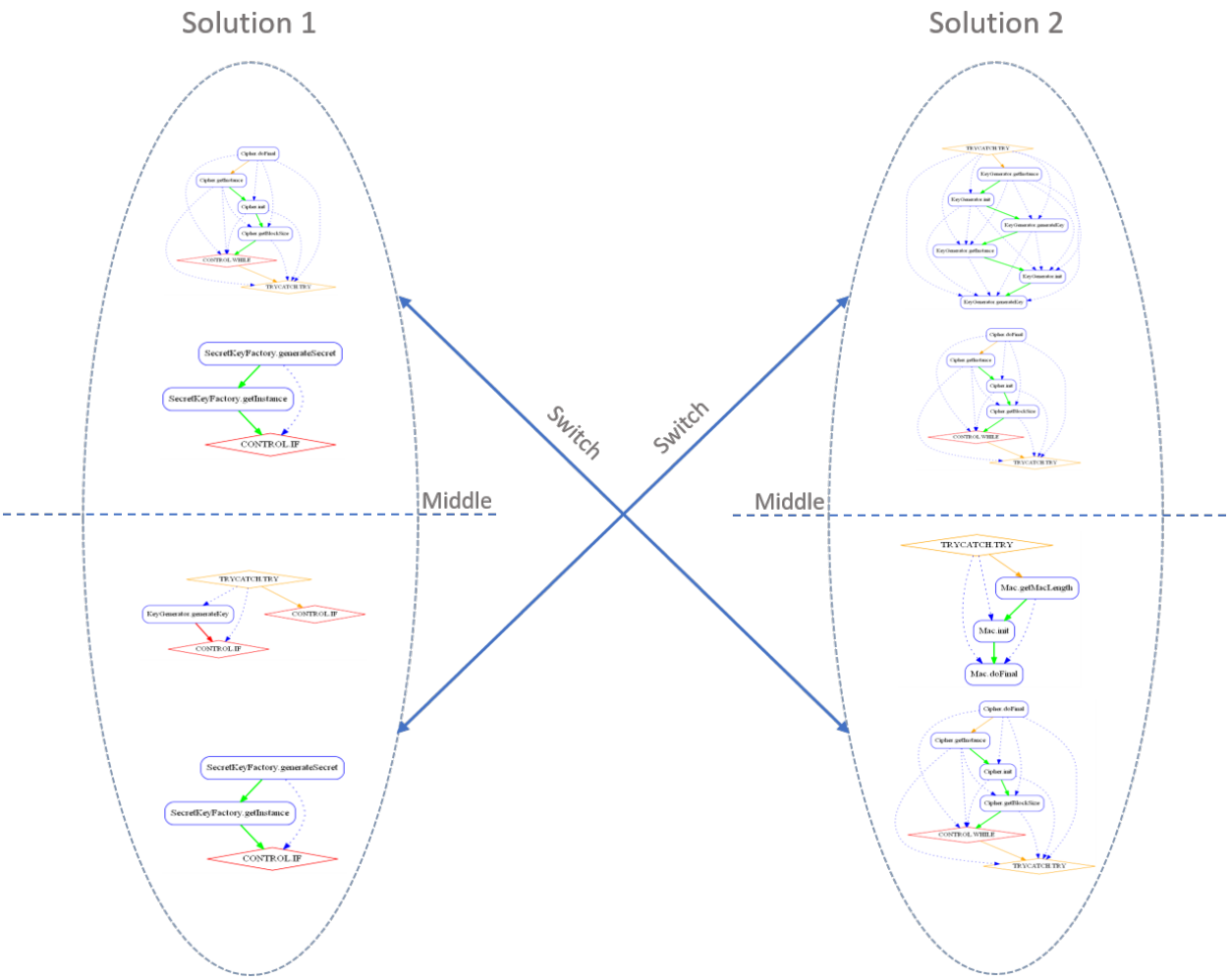

FIGURE 5.3. Cross-over

5.1.3. Mutation

La troisième et dernière étape de l'algorithme génétique est la mutation. Celle-ci a une probabilité (`PROB_MUTE`) d'être appliquée sur toutes les solutions ne faisant pas partie de l'élite. Il existe 4 types de mutation : le remplacement de détecteur, l'ajout de détecteur, la suppression de détecteur et l'altération de détecteur. La mutation a deux rôles. Dans un premier temps, celui d'apporter de la nouveauté dans le matériel génétique disponible dans toutes les solutions. Les détecteurs étant des mutations des *groum*s d'utilisation de l'API. En procédant à la mutation d'un détecteur cela peu générer un détecteur qui n'est présent dans aucune des solutions du processus de génération. Dans un second temps, la mutation peut apporter une légère modification sur la solution mais peut avoir un important impact sur sa *fitness*. En apportant par exemple un détecteur très différent de tous ceux déjà présents dans la solution. Voici donc les quatre types de mutation.

5.1.3.1. *Remplacement*

Le remplacement consiste à prendre dans la solution un détecteur au hasard et de le remplacer par un autre détecteur. Ce détecteur est généré à partir de l'ensemble de bonnes utilisations. Une bonne utilisation est sélectionnée au hasard dans l'*ensemble d'apprentissage* et est par la suite mutée pour donner un détecteur. La mutation en question est la même que l'altération d'un détecteur (présentée ci-dessous section 5.1.3.4).

5.1.3.2. *Suppression*

La suppression est comme son nom l'indique la suppression d'un détecteur, choisi une nouvelle fois aléatoirement dans la solution. Cette opération peut avoir pour conséquence qu'une solution puisse devenir légèrement plus petite que les autres. La suppression peut permettre de supprimer un détecteur déjà présent afin de diminuer le phénomène de similarité entre détecteurs d'une même solution. Cependant, cette opération peut avoir l'effet inverse en supprimant un détecteur apportant de la diversité.

5.1.3.3. *Ajout*

L'ajout quant à lui consiste à ajouter un détecteur généré aléatoirement à partir de l'*ensemble d'apprentissage*. Même processus que pour le remplacement, cependant le détecteur généré ne prend la place d'aucun autre il est ajouté à la solution. Cette mutation peut créer des solutions de taille supérieure mais peu aussi compenser la suppression de détecteurs. Cela peut permettre d'ajouter de la diversité au sein de la solution.

5.1.3.4. *Altération*

L'altération, au lieu de toucher la totalité d'un détecteur, ne s'applique que sur une partie de ce dernier. Un détecteur étant composé de nœuds, d'arêtes et d'exceptions, différentes techniques peuvent modifier ces entités. Il existe 10 types d'altérations de détecteur :

- Ajout d'arête : Le graphe du détecteur se voit ajouter une arête. Cette arête est ajoutée de manière à ce que le graphe reste bien un graphe orienté acyclique et que le type des arêtes reste conforme aux règles établies tableau 4. I.
- Retrait d'arête : Une arête est retirée aléatoirement du graphe.
- Changement du type d'arête : L'arête voit son type modifié.
- Ajout de nœud : Une méthode de l'API contenue dans les méthodes clients de l'*ensemble d'apprentissage* est sélectionnée. Elle est ensuite ajoutée au graphe afin que celui-ci garde toujours ses propriétés de *groum*. Par conséquent, quelques arêtes sont créées pour que le nœud soit dans l'*API groum*. Cet ajout est illustré avec la figure 5.4.

— Retrait de nœud : Un nœud du détecteur est supprimé aléatoirement, et par conséquent ses arêtes.
— Remplacement de nœud : Le remplacement consiste à sélectionner un nœud du détecteur et le remplacer avec une méthode de l'API sélectionnée aléatoirement dans l'*ensemble d'apprentissage*. C'est semblable à l'ajout sauf que l'on remplace un des nœuds de l'*API groum*.
— Échange de nœuds : L'échange, comme le remplacement va sélectionner un nœud aléatoirement, cependant le nœud du détecteur est interverti avec un autre nœud de ce même détecteur.
— Ajout d'exception : On ajoute au détecteur une exception prise dans l'ensemble des exceptions déclarée par toutes les bonnes utilisations de l'*ensemble d'apprentissage*.
— Retrait d'exception : On supprime du détecteur une des exceptions au hasard qu'il possède, s'il en possède une.
— Changement d'exception : On remplace une exception du détecteur par une autre sélectionnée aléatoirement parmi les exceptions déclarées par les bonnes utilisations. Le remplacement a lieu seulement si le détecteur possède déjà une exception.

L'altération est aussi utile lors de la génération des détecteurs. La première étape de génération d'un détecteur est l'altération d'un *API groum* de l'*ensemble d'apprentissage*.

Lorsque ces trois étapes, élitisme, cross-over et mutation ont été exécutées une autre génération prend place. Ces étapes sont réalisées jusqu'au nombre de générations `GENERATION` décidé. Grâce à l'élitisme, la meilleure solution est sûre d'être gardée à chaque génération. Le cross-over et les mutations servent, quant à elles à faire évoluer les solutions pour qu'elles puissent augmenter leur fitness. Ces deux processus permettent un brassage des détecteurs, ce qui permet l'évolution des *fitness*. Ainsi la solution avec la meilleure *fitness* sera celle utilisée. L'ensemble de détecteurs étant considéré le plus performant possible lorsque la *fitness* est plus élevée.

5.1.4. Fitness

La fonction permettant de calculer la *fitness* est primordiale lors de l'utilisation d'un algorithme génétique. Celle-ci permet de classifier les solutions. Sans classification il serait impossible de faire progresser les solutions et obtenir la solution la plus efficace possible. Cette fonction a un objectif à deux critères. Le premier est que les détecteurs au sein d'une solution soient le plus variés possible. C'est-à-dire que plus un détecteur est différent des autres détecteurs de sa solution, meilleur est son score. Le second critère est que les détecteurs doivent se trouver dans un certain intervalle de similarité avec les méthodes considérées comme bonne utilisation. Cet intervalle a été estimé à [0,66 ;0,99] après l'étude de la figure 4.4. Cela signifie que si un détecteur a entre 66% et 99% de ses nœuds, arêtes et exceptions

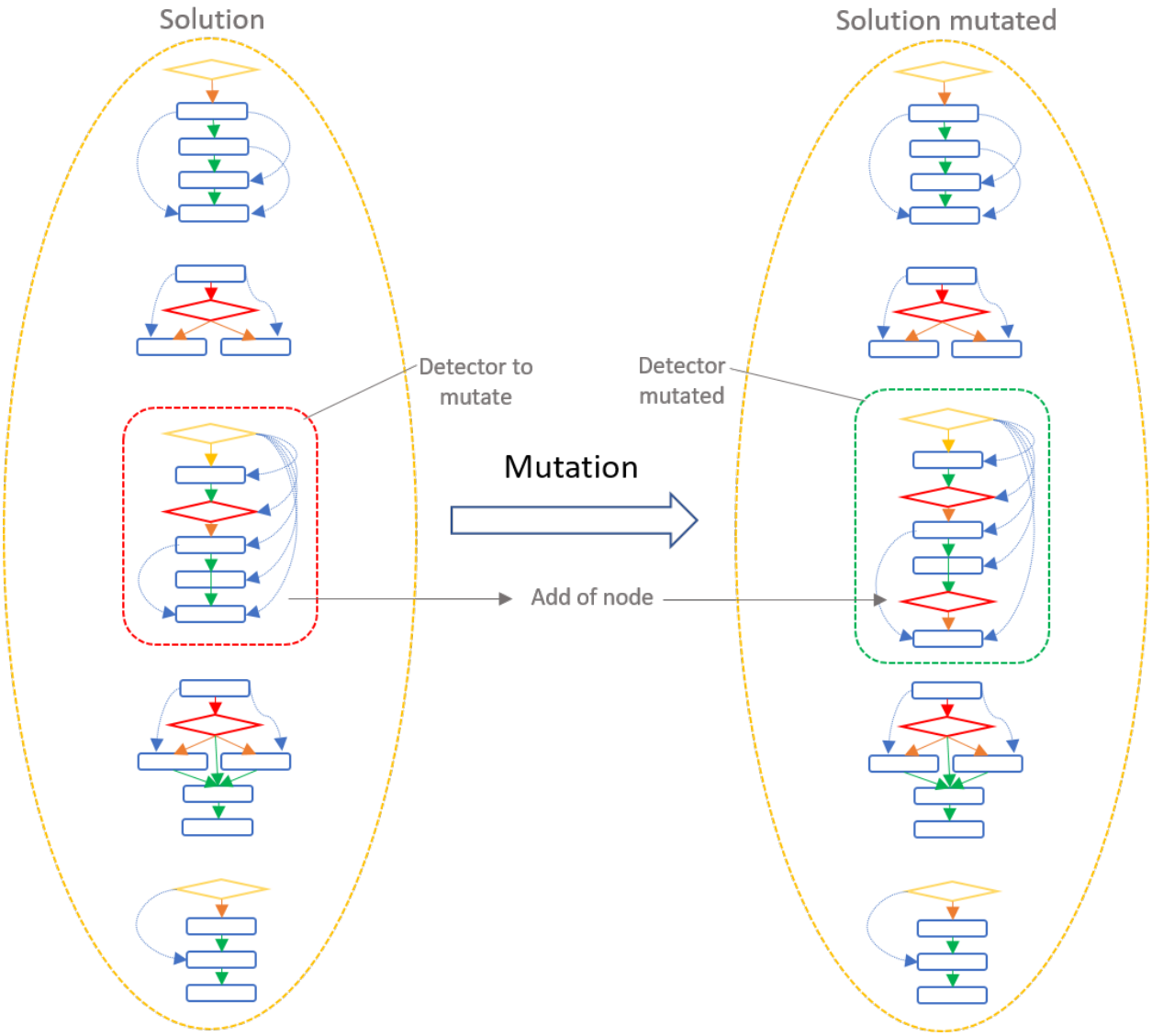

FIGURE 5.4. Exemple de mutation

identiques à un seul des *groums* de bonne utilisation, à partir desquels on apprend, il est favorisé.

5.1.4.1. *Similarité*

La similarité, qui peut être considérée comme l'opposée de la distance (distance = 1 - similarité), permet de définir une valeur comprise entre 0 et 1. Plus la valeur est proche de 1, plus les deux méthodes comparées sont identiques. La méthode de calcul de la similarité compare les *groums* de deux méthodes. Elle passe en revue tous les nœuds, arêtes et exceptions de chacun des *groums* pour voir lesquels sont communs aux deux *groums*. Un nœud est similaire à un autre nœud s'il représente le même appel à une fonction de l'API ou à la même structure de contrôle. Deux arêtes sont jugées identiques, si elles ont un nœud similaire

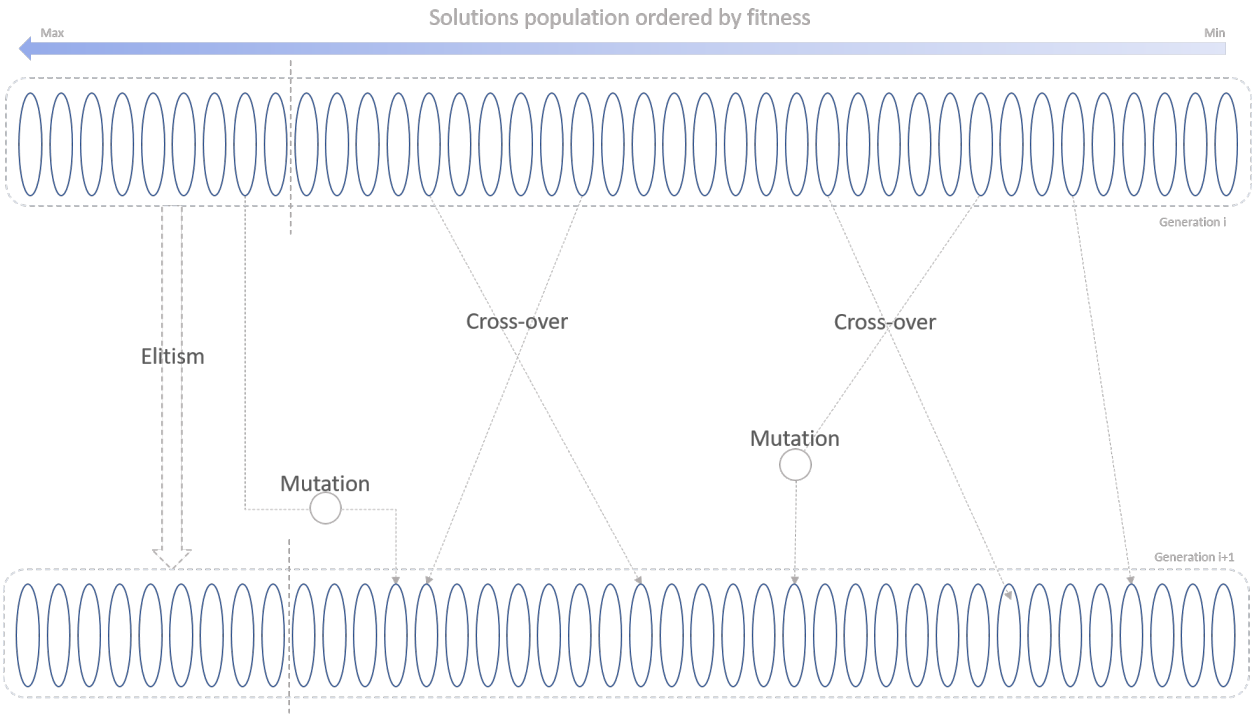

FIGURE 5.5. Algorithme génétique sur une génération

en guise de *source*, un nœud similaire en guise de *cible* et le même type. Et des exceptions similaires si les deux exceptions ont le même nom. La formule compte l'intersection de tous les nœuds, arêtes et exceptions présentent dans les deux *groum*s et divise cette valeur par le nombre total de nœuds, d'arêtes et d'exceptions des deux *groum*s, l'union des deux *groum*s.

$$
\begin{aligned}
intersection(g_1, g_2) = & \\
& |\cap\{g_1.\text{nœud}s, g_2.\text{nœud}s\} \\
& +\cap\{g_1.arêtes, g_2.arêtes\} \\
& +\cap\{g_1.exceptions, g_2.exceptions\}| \\
union(g_1, g_2) = & \\
& |\cup\{g_1.\text{nœud}s, g_2.\text{nœud}s\} \\
& +\cup\{g_1.arêtes, g_2.arêtes\} \\
& +\cup\{g_1.exceptions, g_2.exceptions\}| \\
& -intersection(g_1, g_2) \\
sim(g_1, g_2) = & \frac{intersection(g_1, g_2)}{union(g_1, g_2)}
\end{aligned}
$$

Nous avons donc décidé de considérer la similarité comme l'intersection des éléments en commun des deux *groum*s divisé par l'union des éléments en commun des deux *groum*s, car

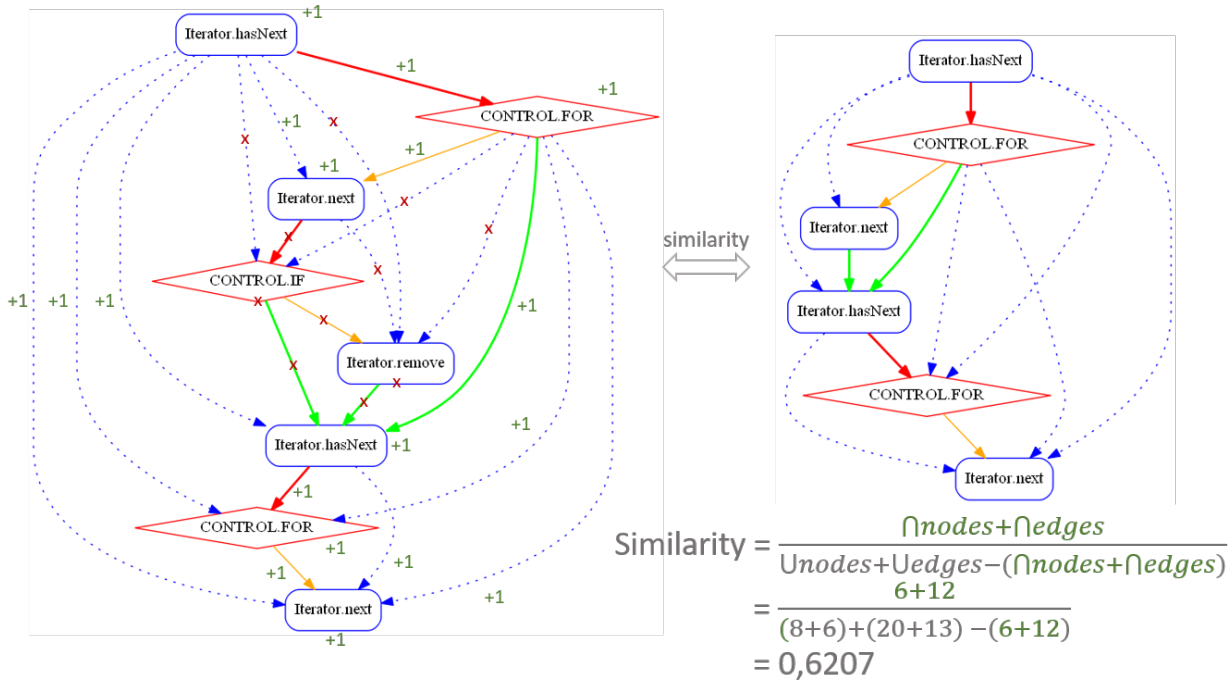

FIGURE 5.6. Calcul de similarité entre deux méthodes

cette façon de penser est intuitive. Nous avions aussi étudié la formule utilisée par Nguyen et al. [39], qui est légèrement différente, mais a tendance à augmenter la valeur des petites similarités. Leur formule additionne tous les éléments communs présents dans le premier *groum* qui sont également dans le second à tous les éléments du second *groum* également présents dans le premier, ce qui équivaut à multiplier par deux l'intersection des deux *groum*s. Le tout étant divisé par la somme du nombre d'éléments du premier et second *groum*. Ici, nous entendons par éléments les nœuds, arêtes et exceptions d'un *groum*.

Un exemple de calcul de similarité entre deux *groum*s est dépeint figure 5.6, où la similarité est calculée pour deux *groum*s n'ayant pas d'exception. On ajoute donc à l'intersection tous les nœuds et arêtes qui sont communs aux deux graphes. C'est ce qui est représenté sur la figure avec le symbole vert`+1`. Au contraire, tous les éléments ayant une croix rouge dans le *groum* de gauche ne font pas parti de l'intersection. Ensuite, la similarité est calculée entre ces deux graphes qui possèdent 18 éléments en commun pour 29 éléments différents présents au sein de ces deux entités. On obtient donc une similarité de 0,6207, soit 62,07% de similarité.

5.1.4.2. *Score*

Maintenant que l'on sait calculer la similarité entre deux *groum*s nous allons pouvoir calculer le score d'un détecteur qui est l'équivalent d'un *API groum* muté. Le score nous sert à calculer la fameuse *fitness*. La *fitness* d'une solution est la moyenne des scores de chacun

des détecteurs. Le score est donc calculé en deux parties pour répondre aux deux critères imposés par la *fitness* qui sont la diversité des détecteurs et une distance comprise entre 0,01 et 0,33 des méthodes de l'*ensemble d'apprentissage*.

Dans un premier temps, on calcule la distance moyenne (`distMoyDet`) entre le détecteur dont on évalue le score et tous les autres détecteurs de la solution dans laquelle ce détecteur est présent. La distance moyenne avec les détecteurs est égale à l'opposé de la somme des similarités entre chaque détecteur de la solution et le détecteur auquel on calcul le score, divisé par le nombre de détecteurs moins 1 (car on ne prend pas en compte le détecteur étudié). Si nous considérons D l'ensemble des détecteurs de la solution avec le détecteur d évalué exclus, la distance moyenne du détecteur d par rapport à D se calcule comme suit :

$$distMoyDet(d) = 1 - \sum_{i=1}^{\texttt{SIZE\_SOLUTION}} \frac{sim(d, D_i)}{\texttt{SIZE\_SOLUTION} - 1}$$

Dans un second temps, il faut calculer la distance aux usages de l'API qui sont dans l'*ensemble d'apprentissage*. La première approche calculait le minimum de dissimilarité avec l'ensemble des méthodes à partir desquelles on apprend (`minGroumDisimilarity`). Ce calcul consiste à trouver la plus petite dissimilarité, la plus petite distance entre le détecteur d et les *API groum*s d'utilisation de l'API de l'*ensemble d'apprentissage* que l'on nommera C. Ce minimum de dissimilarité représente la plus petite distance entre le détecteur et les usages desquels on apprend, et cela veut dire, par conséquent, que le détecteur se situe au moins à cette distance `minGroumDissimilarity` de tous les autres usages jugés comme corrects de l'*ensemble d'apprentissage*. Cette dissimilarité ce calcul ainsi :

$$minGroumDissimilarity(d_i) = 1 - max_{s_i \in C}(sim(d, s_i)$$

Mais comme nous l'avons vu, les erreurs d'utilisation sont de légères déviances des bonnes utilisations. Donc favoriser les détecteurs les plus éloignés des bonnes utilisations ne s'avère pas judicieux car les détecteurs produits seront trop éloignés de l'objectif fixé et ne rempliront pas leur rôle d'identification correctement. La distance entre bonne est mauvaise utilisation a été estimée dans la Section 4.4 où l'on observe sur la Figure 4.4 qu'une mauvaise utilisation est similaire entre 66% et 99% à une bonne utilisation. Il faut donc favoriser les détecteurs qui ont une similarité jugée "correcte". Pour ce faire, nous avons établi la formule suivante :

$$maxSim(d) = max_{s_i \in C}(sim(d, s_i)$$

$$optimalSimilarity(d) = \begin{cases} 1 & \text{si } maxSim(d) \in [0{,}66; 0{,}99] \\ 0 & \text{si } maxSim(d) = 1 \\ 1 \div (0{,}66 \times 0{,}75 \times maxSim(d)) & \text{si } maxSim(d) < 0{,}66 \\ 1 \div ((1 - 0{,}99) \times 0{,}75 \times (1 - maxSim(d))) & \text{si } maxSim(d) > 0{,}99 \end{cases}$$

Ainsi, cette formule se décompose en quatre parties. La première attribue un `optimalSimilarity` de 1 si le maximum de similarité `maxSim` est situé dans l'intervalle de similarité jugé "correct" [0,66 ;0,99]. Le cas qui est pénalisé au maximum est le cas où un détecteur d est strictement identique à un *API groum* présent dans l'*ensemble d'apprentissage* C. Dans ce cas, une valeur de 0 est attribuée. Sinon lorsque le maximum de similarité du détecteur avec C est en dehors de l'intervalle un seuil a été fixé à 0,75. Ce seuil est la valeur maximale que peut atteindre l'`optimalSimilarity` en étant en dehors des bornes [0,66 ;0,99]. De plus, plus la `maxSim` est éloignée de cet intervalle, plus l'`optimalSimilarity` diminue. Ainsi ces détecteurs sont pénalisés proportionnellement à la similarité obtenue.

Dans notre cas, nous avons donné autant de poids aux deux objectifs présentés ci-dessus. Comme `distMoy` et `optimalSimilarity` renvoient des valeurs entre 0 et 1, 0 lorsque l'on s'éloigne de l'objectif et 1 lorsqu'on l'atteint. Pour que le score du détecteur d soit lui aussi entre 0 et 1 et que chaque objectif ait la même importance, on multiplie ces deux objectifs par 0,5 comme le montre la formule suivante :

$$score(d) = 0{,}5 * distMoyDet(d) + 0{,}5 * optimalSimilarity(d)$$

5.2. Génération de détecteurs

Afin de générer les détecteurs, nous avons utilisé différentes méthodes. Mais pour chacune des alternatives, nous commençons de la même manière. Nous prenons l'ensemble des méthodes considérées comme de bonnes utilisations. Ces méthodes sont ensuite transformées en *groum*s, et forment ce qu'on appelle l'*ensemble d'apprentissage* d'*API groum*s.

5.2.1. Génération sans regroupement

Pour générer les détecteurs sans regroupement (clustering), `NB_SOLUTIONS` solutions sont générées avec `SIZE_SOLUTION` détecteurs par solution. Un détecteur est créé en utilisant la mutation d'altération, à une reprise, sur un *API groum* sélectionné aléatoirement de l'*ensemble d'apprentissage*. Ces solutions sont donc la génération 0 (initiale) des solutions. Ensuite, pour faire progresser ces solutions nous utilisons le brassage génétique pendant `GENERATION` générations. L'algorithme génétique consiste d'abord à sélectionner l'élite qui restera au vu de la génération suivante, ensuite de faire un cross-over entre des solutions sélectionnées grâce à la méthode du tournoi et enfin de faire muter quelques solutions au hasard. Les opérations sont répétées à chaque génération. Au

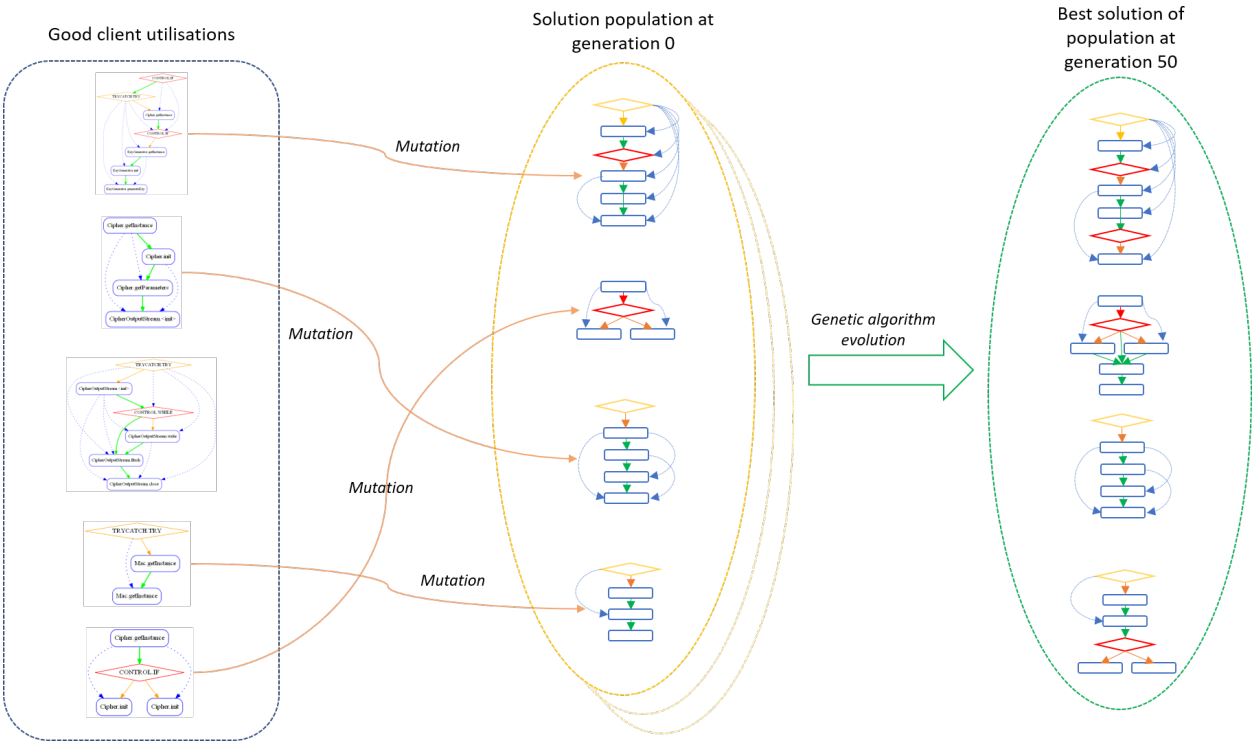

FIGURE 5.7. Processus de génération sans regroupement

bout des `GENERATION` générations on garde la solution avec la *fitness* la plus élevée. Les détecteurs obtenus au sein de cette meilleure solution sont nos détecteurs finaux. Ce processus de génération est illustré figure 5.7 et sera nommé processus de génération de détecteur sans regroupement ou clustering. Ce nom vient du fait que ce processus ne prend pas en compte le travail de regroupement des usages l'API réalisé dans le travail préliminaire.

5.2.2. Génération en créant les solutions en fonction des regroupements

5.2.2.1. *Dérivée n°1 : sélection équitable parmi les regroupements*

La nuance dans les processus utilisant les clusters est que l'ensemble d'apprentissage est divisé en sous-ensembles d'utilisation de l'API grâce au processus de clustering dbscan (cf. Section 4.3). Ainsi, l'*ensemble d'apprentissage* se retrouve réparti en plusieurs groupes de méthodes, d'*API groums* qui vont permettre de générer des détecteurs. Ensuite, avec ce processus, la population de solutions initiales est créée en générant autant de détecteurs pour chaque cluster, de manière strictement équitable. Par exemple, si nous avons 3 clusters pour une API, nous prenons aléatoirement 33% de détecteurs provenant du cluster1, 33% du cluster2, et 33% du cluster3. Par la suite, nous lançons l'évolution grâce au brassage génétique sur la population de détecteurs issue de ces trois clusters. A la fin de la maturation de la population, la meilleure solution, celle avec la meilleure *fitness*, est gardée comme solution finale. Ce processus ne mettant pas assez en valeur le travail réalisé par le regroupement des utilisations de l'API, nous avons décidé de ne pas le garder pour les expérimentations effectuées lors de la validation.

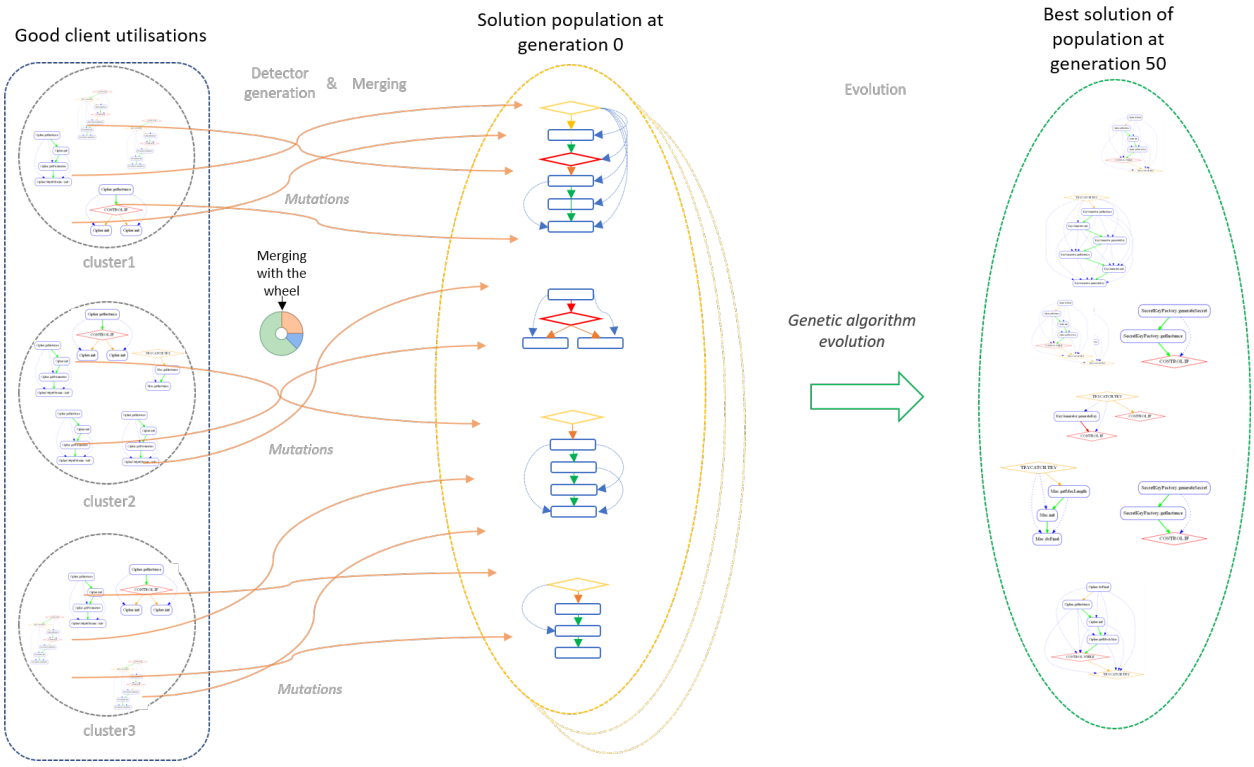

FIGURE 5.8. Processus de génération à partir de clusters avec regroupement avant brassage génétique

5.2.2.2. *Dérivée n°2 : sélection grâce à la technique de la roulette parmi les regroupements*

Pour l'expérimentation, nous avons préféré utiliser cette seconde dérivée. Nous partons comme pour la dérivée précédente avec les clusters. Mais dans ce cas-ci au lieu de sélectionner aléatoirement et équitablement parmi chaque cluster nos méthodes d'apprentissage nous les choisissons grâce au processus de la roulette. Cette méthode s'appelle la roulette car elle consiste à utiliser une roulette que l'on tourne pour faire un choix. La particularité de cette roulette c'est que chaque portion de celle-ci représente la taille d'un cluster. Par exemple si nous avons 3 clusters, le premier $c1$ de taille 65, le deuxième $c2$ de taille 20 et le troisième $c3$ de taille 15. La roulette sera un camembert où nous avons 65% de chance de tomber sur le c1, 20% de chance de tomber sur c2 et 15% de chance de tomber sur c3. La population initiale de détecteurs est donc générée de façon proportionnelle à la taille des groupes d'utilisation de l'API dans l'*ensemble d'apprentissage*, tout en restant aléatoire. Ensuite, la population évolue grâce au brassage génétique et l'on garde la meilleure solution au final, celle avec la meilleure *fitness*. Cette solution profite mieux des travaux préalables de regroupement des usages de l'API car la roue va piocher aléatoirement des *API groum*s au sein des clusters desquels on apprend. Mais la roulette permet que la démarche continue d'être complètement aléatoire. Donc en moyenne chacun des groupes d'utilisation de l'API est proportionnellement représenté dans la population de solutions. Tout ce processus est dépeint avec la figure 5.8

5.2.3. Génération de solutions à partir de clusters, regroupés au fur et à mesure

Pour cette deuxième alternative de regroupement, on crée une population de solutions par cluster, de taille `NB_SOLUTION`, solutions qui contiennent chacune `SIZE_SOLUTION` ÷ `#CLUSTERS` détecteurs. Chaque population évolue indépendamment pendant `GENERATION` générations. Ensuite on regroupe la première solution du premier cluster avec la première solution de tous les autres clusters, et ainsi de suite. Cela forme une population de solutions regroupées de taille `SIZE_SOLUTION`. La solution finale est la solution de cette population possédant la meilleure *fitness*. Ce processus de génération n'a pas été conservé pour diverses raisons. La première est que les solutions de taille `SIZE_SOLUTION` ÷ `#CLUSTERS` composées uniquement de détecteurs dérivés d'un seul cluster d'utilisation peuvent se révéler peu efficace lorsqu'il y a beaucoup de clusters et des clusters de taille très différente. Car les solutions vont être de petite taille à cause du grand nombre de groupes d'utilisations et que les groupes possédant un grand nombre d'utilisations ne seront peut-être pas assez représentés. De plus, le regroupement des solutions de population par ordre d'apparition dans chaque population peut entraîner la perte des meilleurs détecteurs générés et gâcher tout le travail effectué précédemment. Ce sont ces facteurs qui ont principalement motivé notre choix de ne pas continuer à explorer ce processus de génération de détecteurs.

5.2.4. Génération de solutions à partir de clusters, regroupement des détecteurs des meilleures solutions

5.2.4.1. *Dérivée n°1 : Regroupement par roulette*

Dans cette alternative, on génère aléatoirement des solutions de `SIZE_SOLUTION` détecteurs pour chaque cluster. Ici aussi, on crée une population de solutions pour chaque cluster séparément. La différence avec l'alternative précédente, c'est donc la taille des solutions qui font `SIZE_SOLUTION`. On a donc chaque population qui évolue individuellement pendant `GENERATION` générations. A la fin de l'algorithme génétique, nous ne gardons que la meilleure solution pour chaque population. Ensuite, nous utilisons la technique de la roulette décrite figure 5.9 pour sélectionner à l'intérieur des solutions restantes des détecteurs aléatoirement pour remplir notre solution finale. `SIZE_SOLUTION` détecteurs sont donc piochés dans la meilleure solution de chaque cluster grâce à la roulette qui donne une probabilité à chaque solution proportionnelle à la taille du cluster duquel les détecteurs sont issus. Cependant, ce processus n'a lui aussi pas été retenu pour les expérimentations car on a considéré pouvoir encore l'améliorer avec la dérivée suivante.

5.2.4.2. *Dérivée n°2 : Regroupement par brassage génétique*

Avec cette seconde dérivée, nous générons encore une fois pour chaque cluster une population de solutions de taille `NB_SOLUTION` comportant chacune `SIZE_SOLUTION` détecteurs. Les populations évoluent chacune indépendamment et nous gardons seulement la meilleure solution de chaque population. La différence se fait lors de la génération de la solution finale. Ici aussi nous utilisons la roulette (comme dans la section 5.2.2.2) mais pour générer une population entière de `NB_SOLUTION` solutions de taille `SIZE_SOLUTION`. Ce regroupement des différentes populations est inspiré du regroupement utilisé par Baki et Sahraoui [8]. Cette population évolue également grâce au brassage

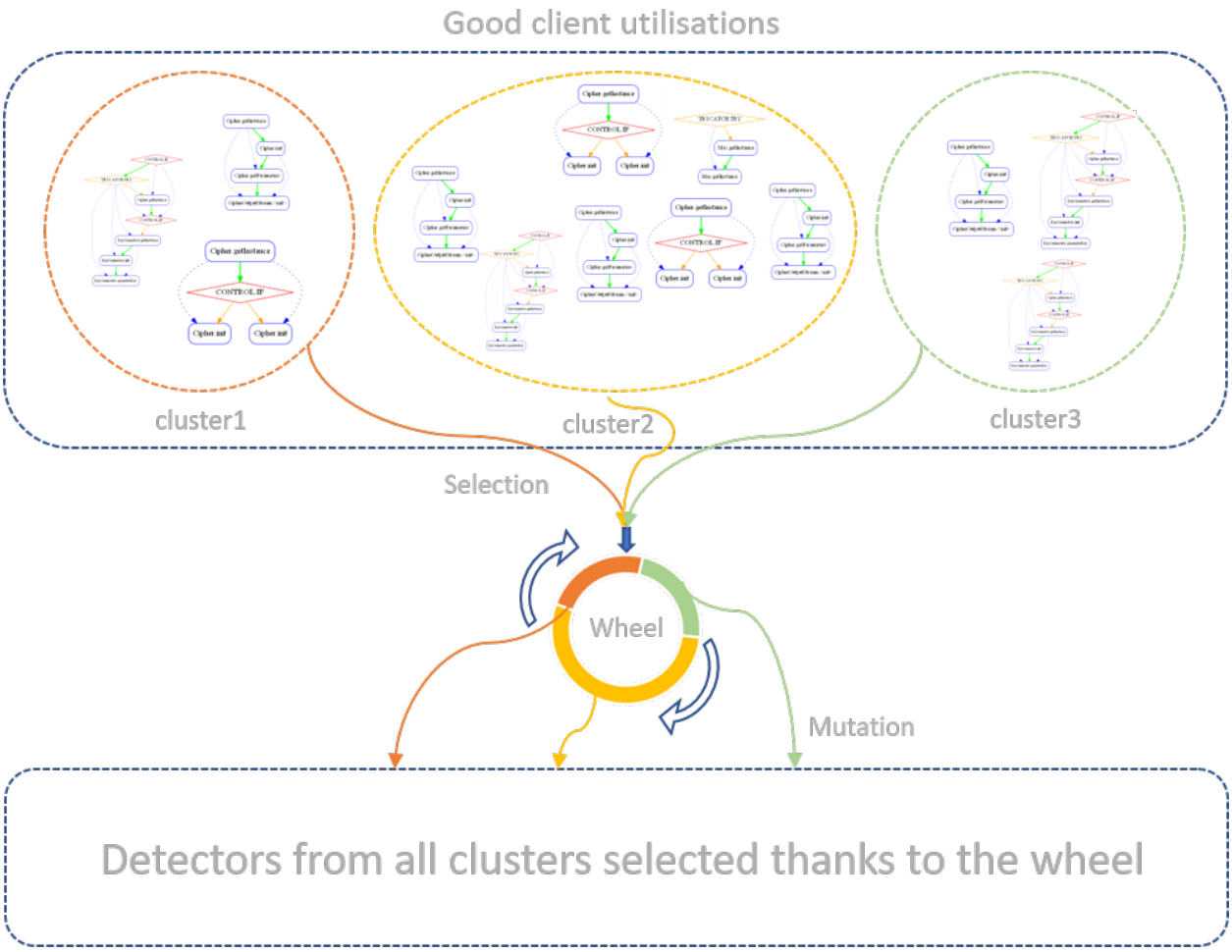

FIGURE 5.9. Sélection grâce à la roulette

génétique, comme une population normale, à la seule différence que dans la mutation il n'y a pas de modification de détecteurs. Ce brassage permet de recréer une solution où les détecteurs sont les plus différents possible. Cela corrige le défaut que pouvait avoir la première dérivée, car ne pas faire de brassage et seulement prendre des détecteurs des meilleures solutions ne garantit pas que la solution obtenue soit la plus efficace. Cela peut créer une solution avec des détecteurs proches les uns des autres par exemple. Le brassage permet de se conformer de nouveau à l'objectif de la fonction de *fitness*. Finalement, on garde la meilleure solution de cette population brassée obtenue par sélection avec roulette, notre solution finale est donc celle avec la meilleure *fitness*.

5.2.5. Récapitulatif

Parmi les différents processus de génération de détecteurs décrits au cours de cette section, nous avons décidé d'en évaluer seulement trois d'entre eux. Ces trois processus sont la génération sans regroupement (ou clustering), la génération avec regroupement avant l'évolution et celle avec le regroupement après l'évolution. Le premier processus sans regroupement permettra de valider ou non l'utilité du regroupement par rapport aux deux autres processus qui l'utilisent. Ensuite,

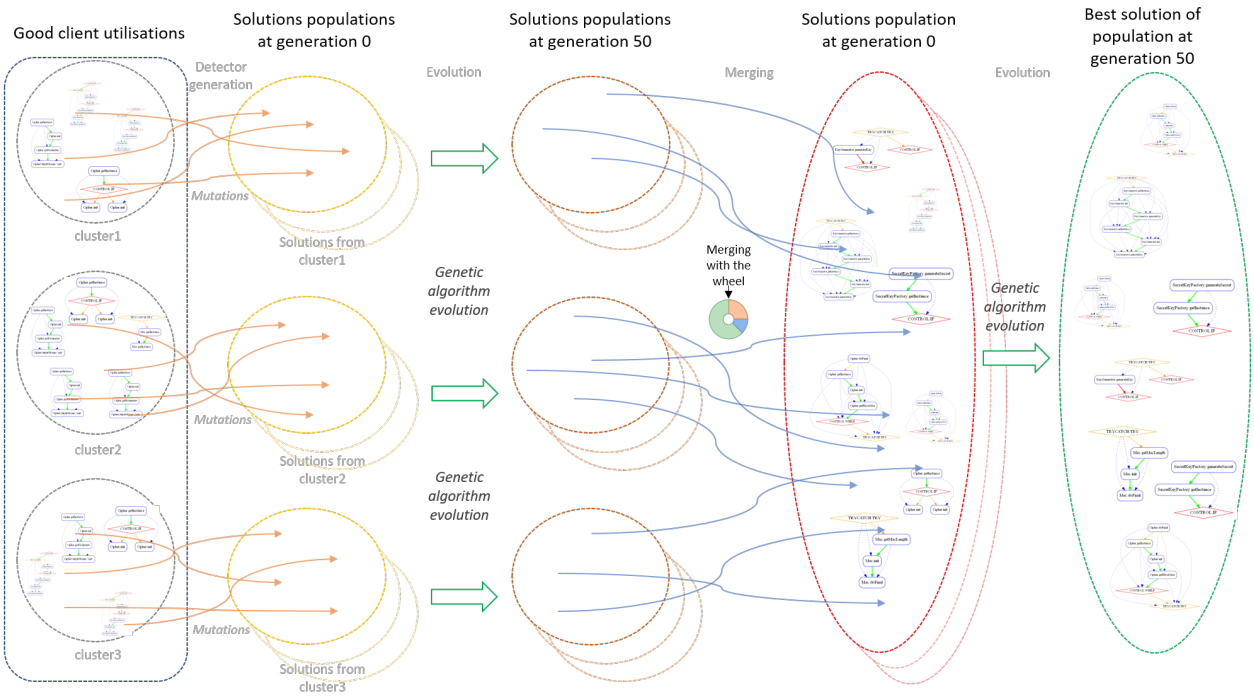

FIGURE 5.10. Processus de génération avec regroupement après l'évolution des populations de solutions

les deux processus utilisant le clustering nous tâcherons de voir s'il est plus efficace de générer des détecteurs qui évoluent de manière *parallèle*, c'est-à-dire que les populations de solutions sont générées pour chaque cluster et évoluent distinctement. Ou s'il vaut mieux une évolution *globale* en regroupant des détecteurs de chaque cluster grâce à la roulette et laisser l'évolution suivre son cours. Nous testerons ces différentes approches au sein de la validation.

5.3. RANKING DES MÉTHODES

Le but de notre approche est de détecter les méthodes qui utilisent une API d'une manière différente de projets qualifiés comme fiables. Nous avons donc appris grâce à la foule en procédant à un apprentissage par le nombre. Cette foule qui représente les bonnes utilisations relevées, et qui a permis de générer les détecteurs, que l'on peut qualifier d'utilisations déviantes artificielles. La dernière étape consiste à donner un score de risque aux méthodes que l'on veut étudier. Ce score sera une valeur entre 0 et 1. Plus le score est proche de 1 plus la méthode sera considérée comme une méthode utilisant l'API évaluée de manière risquée. Pour assigner un score, nous avons utilisé deux méthodes différentes : **max** et **noisy or**.

5.3.1. Max

Le **max** consiste à comparer l'ensemble des méthodes que l'on veut tester avec nos détecteurs. L'indice donné à chaque méthode sera le score de similarité maximal entre la méthode étudiée et un détecteur de la solution finale. Pour ce faire, nous devons tout d'abord passer la méthode à évaluée

sous forme d'*API groum*, sous forme de graphe représentant l'utilisation de l'API. Ensuite, nous utilisons la fonction de similarité utilisée dans le calcul de score (cf. Section 5.1.4.1). C'est-à-dire que le score de risque de la méthode m est égal au maximum de similarité entre m et les détecteurs de la solution D :

$$score(m) = max_{d_i \in D}(sim(m, d_i)$$

Ainsi, les scores des méthodes nous permettent de les ordonner. Chaque score est compris entre 0 et 1. Plus le score est élevé, plus la méthode est sensée être à risque et sera en haut du classement. Ce score peut être qualifié d'indice de risque ici. Utiliser le **max** permet donc de chercher les méthodes qui matchent le plus avec un des détecteurs de la solution. Par analogie avec ce qui se passe dans l'organisme, il y a détection de pathogène lorsqu'un lymphocyte T arrive à matcher avec une cellule. Donc la méthode qui se rapproche le plus d'un détecteur de la solution est donc la plus à risque. Le **max** permet donc d'évaluer la qualité de la détection pour assigner un score de risque.

5.3.2. Noisy or

Le **noisy or** [1] est une approche légèrement différente du **max**. Plutôt que de regarder le détecteur qui matche au mieux la méthode que l'on teste, nous allons assigner des scores plus élevés aux méthodes qui sont proches de plusieurs détecteurs. Plus la méthode est proche de différents détecteurs avec de fortes similarités, plus elle sera considérée à risque. Dans notre métaphore avec le système immunitaire cela signifierait qu'une cellule qui a tendance à matcher avec plusieurs lymphocytes T serait qualifiée comme pathogène. Ainsi on favorise le nombre de ressemblances sans pour autant pénaliser une forte ressemblance. Le risque serait ici évalué de manière plus quantitative par rapport au **max** qui l'évalue de manière strictement qualitative. Le score de risque de la méthode m est égale à 1 moins le produit des distances de m aux détecteurs de la solution D, lorsque la similarité est supérieure à 0,1. Ce seuil a été fixé mais c'est un paramètre qui peut être changé.

$$score(m) = 1 - (\prod_{i=1}^{\texttt{SIZE\_SOLUTION}} 1 - sim(m,D_i))$$

La différence entre ces deux approches de calcul du score de risque est résumée avec la figure 5.11. Cette figure montre que l'on compare la méthode étudiée à tous les détecteurs qui composent la solution et ensuite on assigne un score de risque en fonction de la formule choisie.

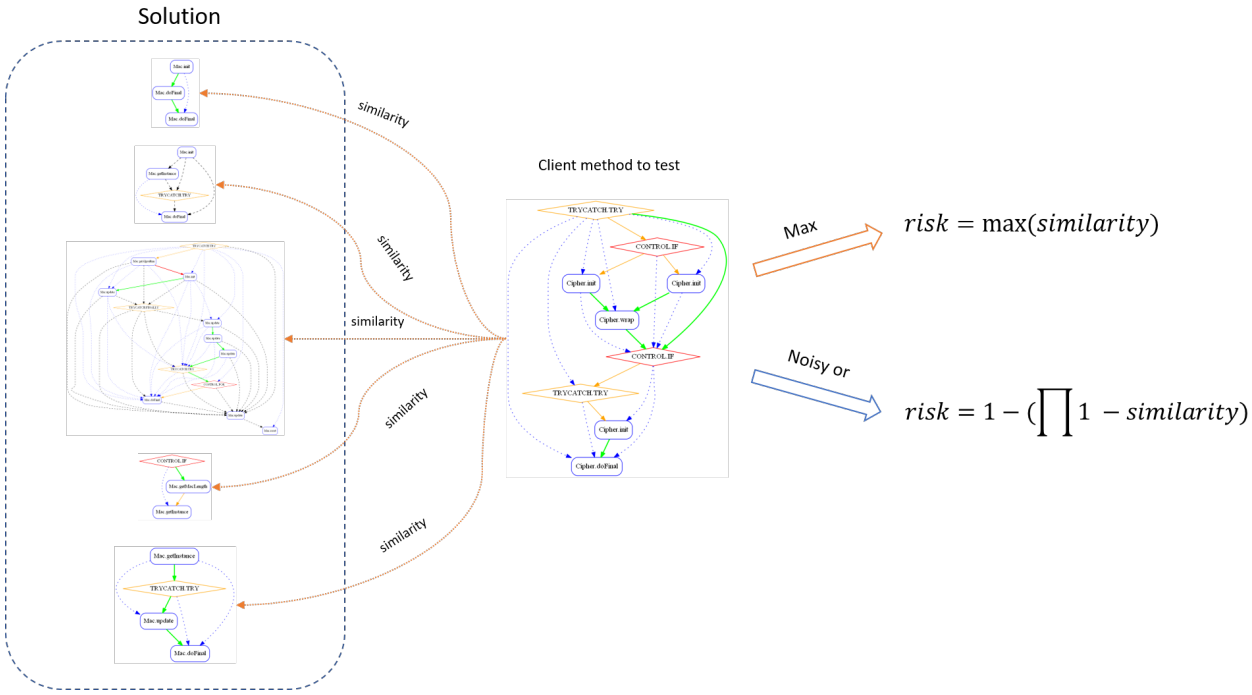

FIGURE 5.11. Processus d'attribution de score à une méthode évaluée

Chapitre 6

VALIDATION

La validation de notre approche a été faite de plusieurs manières, afin de répondre aux problématiques posées. Elle a été réalisée avec trois APIs Java populaires et différentes. Ces APIs sont `java.util.Iterator`, `javax.crypto` et `javax.servlet.http` (cf. Section 4.2). Le but étant de répondre aux questions de recherche suivantes :

RQ1 *: Quel est l'impact des différentes stratégies de génération de détecteurs sur la précision de la détection ?*

RQ2 *: Quel est le coût en termes de temps et de mémoire de notre approche ?*

RQ3 *: Où se situe notre approche par rapport aux approches déjà existantes ?*

Dans un premier temps, nous détaillerons le cadre expérimental en faisant une revue des différents paramètres de l'approche. Ensuite dans la Section 6.2 nous avons testé la validité des détections sur notre ensemble de données et sur un ensemble d'erreurs d'utilisation référence (MUBench). Nous avons ensuite comparé la précision de ces détections par rapport à d'autres outils de détection d'erreur d'utilisation d'API. Nous continuons dans la Section 6.3 l'évaluation de l'outil, mais ici du point de vue de la performance, que ce soit en terme de mémoire ou de temps d'exécution. Ce chapitre s'achève par la Section 6.4, où l'on expose les résultats et où l'on aborde les différentes limitations rencontrées.

6.1. Cadre expérimental

Notre approche comporte de nombreux paramètres, c'est pour cela que nous allons tester différentes configurations au sein de cette validation. Les trois principaux axes de variation seront le processus de génération de détecteurs, la méthode d'assignation du score de risque et l'utilisation des arêtes de dépendance de données des *groum*s. Mais il y a aussi un grand nombre de paramètres que nous avons fixés pour des raisons de temps d'exécution et d'efficacité.

6.1.1. Paramètres fixes

Dans un premier temps, nous avons donc décidé de fixer des paramètres afin de pouvoir réaliser les validations de ce projet en un temps raisonnable. L'approche possède en tout 19 paramètres pouvant être changés. Fixer une partie de ces variables était indispensable pour réaliser une validation dans des délais acceptables. Nous allons les passer en revue et expliquer nos choix pour chacun d'entre eux.

6.1.1.1. *Taille de la solution*

La taille de la solution (`SIZE_SOLUTION`), qui est un paramètre crucial de l'approche, a été fixée à 50. Nous avons choisi d'avoir des solutions comportant une cinquantaine de détecteurs pour des raisons de performance de l'algorithme. Nous disons une cinquantaine, car leur nombre peut légèrement varier en fonction des différentes mutations survenues durant la génération (cf. Section 5.1.3). La taille de la solution est fortement liée à la performance, car plus la solution est de taille conséquente plus le temps de génération de solutions sera important. Vu que lors de la génération de cette dernière il faut manipuler `SIZE_SOLUTION` détecteurs pour `NB_SOLUTION` solutions, la complexité algorithmique augmente significativement. De plus, ce processus est répété pendant `GENERATION` générations. Associé au phénomène de copie en profondeur (deep copy) des détecteurs qui consiste à recopier entièrement chaque détecteur après chaque génération pour ne pas subir d'effet de bord, la complexité ne cesse de croître. La question de performance se pose aussi lors de l'utilisation des détecteurs. Car plus la solution est grande plus il y a de détecteurs avec lesquels comparer. Cela est dû au fait que pour calculer le score de risque d'une méthode, il faut calculer la similarité entre la méthode évaluée et chaque détecteur de la solution, peu importe si on utilise **max** ou **noisy or**. Donc fixer ce paramètre à 50 permet d'avoir des performances jugées raisonnables. Ensuite, avoir 50 détecteurs signifie que l'on compare la méthode évaluée à 50 utilisations supposées aberrantes de l'API en cause. Se comparer à 50 détecteurs, les plus variés possible, qui est un des objectifs de notre approche, semble être suffisant au vu des expérimentations réalisées.

6.1.1.2. *Nombre de solutions*

Le nombre de solutions (`NB_SOLUTION`) a aussi été fixé à 50. Nous avons choisi cette valeur dans un souci de performance également. Le nombre de solutions impacte la production de la solution finale, peu importe le processus de génération utilisé. Lorsqu'il n'y a pas d'utilisation du regroupement (cf. Section 5.2.1), il y a donc une production de 50 solutions de taille 50 (comme nous l'avons vu Section 6.1.1.1), soit 2500 détecteurs. Pour le processus avec regroupement après l'évolution des détecteurs (cf. Section 5.2.4.2) il y aura cette fois-ci 2500 détecteurs pour chaque groupe (cluster). Ce qui consiste à faire une génération sans regroupement pour chaque groupe (cluster). De plus, le nombre de solutions influe lors du regroupement, car il se fait lui aussi par le biais de l'algorithme génétique (cf. Section 5.1). Pour le processus avec le regroupement avant l'évolution (cf. Section 5.2.2.2) il y aura aussi 2500 détecteurs présents dans la population. Le choix du nombre de solutions n'a d'impact que durant le processus de génération et aucunement durant

l'évaluation du score de risque. Il a été fixé à 50 pour obtenir des temps de génération considérés comme raisonnables de notre part, tout en gardant une diversité correcte avec 2500 détecteurs générés par population, les 50 détecteurs retenus ne devraient avoir aucun mal à être variés.

6.1.1.3. *Nombre de générations*

Le nombre de générations (`GENERATION`) a été défini à 50. Cela signifie que lors de l'utilisation de l'algorithme génétique les opérations sont exécutées cinquante fois. Ce paramètre a donc lui aussi un impact sur les performances lors de la génération. Car si l'on a 50 solutions, il va falloir procéder 50 fois aux étapes de l'algorithme génétique (cf. Section 5.1), qui sont, l'élitisme, le cross-over et la mutation. Et dans le cas du processus de génération de détecteurs avec regroupement après génération (cf. Section 5.2.4.2) les opérations génétiques sont à nouveau répétées lors du regroupement, donc il y a un double impact. Cependant, ce paramètre n'influe pas lors de l'évaluation du score de risque d'une méthode client. De plus, on peut considérer que 50 générations permettent de bien brasser les solutions de la population pour créer de la variété et des mutations, pour obtenir une solution efficace.

6.1.1.4. *Proportion de l'élitisme*

La proportion de l'élite (`PROP_ELIT`) a été établie à 0,2. Cela veut dire que les 20 meilleurs pour cent des solutions sont gardés intacts de la génération `g` à la génération `g`+1. Les meilleures solutions étant les solutions possédant la meilleure *fitness*. Ce paramètre, pour sa part, n'intervient que pendant le processus de génération des détecteurs, et le processus de regroupement après génération des détecteurs. Il n'influence pas le temps d'exécution de l'algorithme, que ce soit lors de la génération ou l'utilisation des détecteurs. Ce paramètre est cependant essentiel pour le brassage génétique, car il permet de conserver en tout temps la meilleure solution d'une génération à l'autre. Le fixer à 20% permet donc de garder les 10 meilleures solutions entre chaque évolution et permettre lors des cross-over de brasser des solutions avec une plus grande probabilité d'être convenable. Ainsi, malgré les altérations de l'algorithme génétique, nous pouvons conserver une proportion raisonnable de solutions avec les meilleures *fitness*, tout en laissant 80% des solutions être bien brassées pour qu'elles progressent.

6.1.1.5. *Probabilité de cross-over*

La probabilité de cross-over (`PROB_CROSS`) a été fixée à 0,9. Cela signifie qu'il y a 90% de chance que deux solutions subissent un cross-over, lors du passage d'une génération à la suivante. Ce paramètre a un impact négligeable sur le temps d'exécution et n'a aucun impact lors de la détection. En revanche, prendre une forte probabilité de cross-over permet un grand brassage génétique. Ce brassage permet de générer une grande variété de solutions. Pour rappel, l'un des objectifs de nos solutions est qu'elles possèdent des détecteurs les plus différents les uns des autres. Échanger une partie de deux solutions entre elles permet donc d'ajouter rapidement de la variété. Mais cela peut aussi avoir l'effet inverse, en mettant des détecteurs trop proches ensemble et donc pénaliser cette solution qui verra sa *fitness* descendre. C'est pour cela que l'élitisme est présent, car il permet de

toujours sauvegarder les meilleures solutions de la population. De plus, une solution avec une faible *fitness* devrait moins intervenir dans le brassage. Car la sélection pour le cross-over se fait grâce à la méthode du **tournoi** (cf. Section 5.1.2) qui sélectionne deux fois aléatoirement deux solutions et ne considère que celles avec les deux plus hautes *fitness* de chaque duel. Donc cette probabilité élevée de cross-over a pour but de favoriser la variété des détecteurs présents au sein de la solution finale.

6.1.1.6. *Probabilité de mutation*

La probabilité de mutation (`PROB_MUTE`) a été établie à 0,1. Il y a donc 10% de chance qu'une mutation soit réalisée sur chaque solution qui est sélectionnée pour passer de la génération `g` à la génération `g+1`. La mutation cause une légère altération, mais qui peut avoir des conséquences assez importantes sur la *fitness* d'une solution, c'est pour cela que sa probabilité a été fixée à une valeur plutôt faible. Car la forte probabilité de cross-over altère fortement les solutions, et qu'apporter trop de nouveau matériel grâce à la mutation pourrait entraîner une dissolution du matériel déjà présent. Ce paramètre influence le temps d'exécution de génération de façon négligeable et n'impacte aucunement celui de la détection. Il permet d'apporter de la variété au sein des détecteurs, mais avec parcimonie.

6.1.1.7. *Poids de la fonction objectif*

Lors du calcul de la *fitness* d'une solution, deux objectifs sont visés (cf. Section 5.1.4.2). Il y a d'une part, la volonté que les détecteurs soient les plus différents les uns des autres. Et d'autre part, l'objectif que les détecteurs se situent dans un intervalle de similarité spécifié. Le calcul de la valeur de la *fitness* comprend donc 2 variables. Nous avons décidé que les deux objectifs auraient le même poids. Nous avons donc défini la valeur de `SIMILARITY_APIGROUMS` et de `SIMILARITY_DETECTORS` à 0,5 chacun. Ces deux paramètres n'influent pas sur le temps d'exécution, ils permettent juste d'ajuster l'importance apportée à chaque objectif de la méthode de calcul de la *fitness*.

6.1.1.8. *Intervalle de similarité à privilégier*

Comme nous venons de le voir ci-dessus, le calcul de la *fitness* d'une solution a comme objectif d'attribuer une valeur plus élevée aux solutions avec des détecteurs ayant une similarité avec les *API groum*s de bonnes utilisations de l'API situées dans un intervalle déterminé (cf. Section 5.1.4). Cet intervalle a été défini à [0,66 ;0,99] au vu de la figure 4.4. Sur cette figure on peut observer que les erreurs d'utilisation ont toutes une similarité au-dessus de 0,5 et majoritairement au-dessus de 0,66. Cela implique que tous les détecteurs ayant une similarité située entre 0,66 et 0,99 avec une bonne utilisation seront favorisés. Ensuite, plus la similarité est à l'extérieur de l'intervalle, plus leur score est pénalisé. Cet intervalle peut être interprété comme le fait que les détecteurs doivent être au moins à 66% similaires aux bonnes utilisations relevées sans être identiques à aucune bonne utilisation existante. Ce paramètre n'interfère pas avec le temps d'exécution et n'est pas utilisé lors de la détection.

6.1.1.9. *Seuil minimal de similarité pour noisy or*

La méthode de calcul de score **noisy or** a besoin d'un seuil minimal afin de choisir quels détecteurs prendre en compte. Ce seuil a été fixé à 0,1. Cela veut dire que lors du calcul du score de risque (cf. Section 5.3.2) seules les similarités au-dessus de 0,1 sont prises en compte dans la formule. Donc plus la méthode évaluée est similaire à différents détecteurs, avec une similarité au-dessus des 10%, plus son score de risque augmente. Ce seuil est défini à 0,1, en raison du fait que les similarités sont très souvent aux alentours de 0,2. Cela permet de voir l'impact d'être similaire avec un grand nombre de détecteurs, grâce à **noisy or**, face à une similarité très élevée, le **max**. C'est en quelque sorte la comparaison de la quantité de similarité avec la qualité de la similarité.

6.1.1.10. *Probabilité de générer une nouvelle arête*

Le dernier paramètre fixé est la probabilité de générer une nouvelle arête (`PROB_EDGE`). Elle est configurée à 0,35. Cette probabilité est utilisée lors de l'ajout d'un nœud à un détecteur pendant la mutation ajout (cf. Section 5.1.3). Car lors de l'ajout d'un nœud il faut le rattacher au reste de l'*API groum* que forme le détecteur. Donc les nœuds du détecteur sont passés en revue et il y a 35% de chance qu'une arête soit générée entre les deux nœuds. Cependant, l'arête n'est générée qu'à la condition qu'elle ne viole pas le principe que le *groum* est un graphe acyclique orienté. Par conséquent, une arête ne respectant pas cette règle verra sa création avorter. Ce paramètre a été fixé à 35% pour ne pas surcharger d'arêtes les détecteurs générés. Enfin, cette variable a un impact négligeable sur le temps d'exécution de la génération de détecteurs et aucun lors de l'évaluation de risque d'une méthode.

6.1.2. Paramètres variables

Après avoir fixé tous les paramètres cités ci-dessus, nous avons décidé de tester des configurations autour de 3 paramètres. Nous avons testé les différentes combinaisons en faisant varier le processus de génération de détecteurs, la méthode de calcul, avec **noisy or** et **max** et la prise en compte des arêtes de dépendance de données des *API groum*s.

6.1.2.1. *Processus de générations de détecteurs*

Trois processus de génération de détecteurs ont été soumis à une validation. Le premier processus est la génération de détecteurs sans regroupement (cf. Section 5.2.1). Le deuxième processus génère une population de solutions pour chaque groupe (cluster) de bonne utilisation de l'API. Le regroupement se fait après l'évolution de chaque population, où les détecteurs de la meilleure solution de chaque population sont sélectionnés grâce à la méthode de la *roulette* pour créer une nouvelle population qui évolue jusqu'à obtenir la solution finale (cf. Section 5.2.4.2). Le troisième processus va quant à lui procéder au regroupement des bonnes utilisations, également grâce à la *roulette*, au moment de la création de la population de solutions, qui va ensuite évoluer afin de produire la solution finale (cf. Section 5.2.2.2).

Ces trois processus seront confrontés dès que possible. Car pour pouvoir lancer les processus qui utilisent le regroupement il faut que l'API étudiée possède plusieurs types d'utilisations. Parmi

les trois APIs considérées, `javax.crypto` et `javax.servlet.http` possèdent chacune plusieurs groupes d'utilisation. Alors que `java.util.Iterator` ne possède qu'un seul type d'utilisation. Cela s'explique par le fait que `java.util.Iterator` est une classe ne possédant que 4 méthodes. En revanche, `javax.crypto` est un package de 10 classes et `javax.servlet.http` un package de 7 classes. Par conséquent, il y a plus d'utilisations possibles, chez les deux dernières APIs, grâce à la possibilité de faire interagir les différentes classes d'un package entre elles.

Nous tenterons de montrer grâce aux différentes validations s'il existe des bénéfices à procéder à un regroupement des bonnes utilisations par rapport au fait de considérer toutes les utilisations recueillies comme une seule entité. Nous tenterons de montrer aussi s'il vaut mieux procéder à une évolution globale ou parallèle des détecteurs. Cela signifie-t-il qu'il vaut mieux regrouper les détecteurs avant ou après l'évolution avec l'algorithme génétique ? Nous répondrons à ces questions durant les expérimentations.

6.1.2.2. *Méthode de calcul du score de risque*

Deux méthodes de calcul du score de risque seront aussi évaluées. D'une part, il y aura **max** (cf. Section 5.3.1), et d'une autre part **noisy or** (cf. Section 5.3.2). **Max** est la méthode de calcul que nous pourrions qualifier d'intuitive. Elle consiste à comparer la méthode évaluée à tous les détecteurs de la solution et d'assigner comme score de risque la similarité maximale obtenue entre un détecteur et la méthode. On assigne un score par la qualité de sa similarité. Alors que pour **noisy or**, on calcule aussi la similarité entre la méthode évaluée et tous les détecteurs, mais le score de risque assigné est égal à l'opposé du produit des opposés des similarités au-dessus d'un certain seuil (cf. Section 6.1.1.9). Dans ce cas-ci, on qualifie de score de risque la mesure de la quantité de similarité obtenue entre la méthode évaluée et les détecteurs, sans pour autant ne pas considérer la qualité de cette similarité.

Ces deux méthodes sont tout de même liées, car elles prennent en compte la similarité et donneront toujours plus d'importance à une forte similarité. Donc il est possible que les résultats aient tendance à varier de la même manière. Nous vérifierons cela durant l'expérimentation. Nous allons tenter de comparer à travers différentes validations quel type de méthode de calcul vaut-il mieux choisir entre **max** et **noisy or**, entre ce qu'on pourrait appeler le qualitatif et le quantitatif de la détection.

6.1.2.3. *Considération des arêtes de dépendance de données*

Dernier paramètre que nous allons faire varier, c'est la prise en compte des arrêtes de dépendance de données au sein des *groum*s. Ce paramètre est à prendre en considération, car les arêtes interviennent lors du calcul de la similarité entre deux *groum*s. Lorsque les arêtes ont le même poids qu'un nœud, cela signifie que l'ordre et les dépendances entre les méthodes sont aussi importants que la présence d'une méthode. Ainsi, nous avons décidé de comparer la considération ou non des arêtes de dépendance de données, car ce sont les arêtes les plus présentes au sein des *groum*s. Le fait d'ignorer leur présence a pour but de mettre l'emphase sur les autres types d'arêtes qui représentent plus la structure de l'utilisation d'une API. Les arêtes de dépendance de données permettent de

révéler comme leur nom l'indique les dépendances de données, donc de voir si entre plusieurs appels de méthodes, des données, comme par exemple des variables sont partagées.

Avec nos expérimentations, nous allons tenter de voir si les dépendances de données permettent ou non une meilleure détection des utilisations à risque d'API.

6.2. Validation des détections

Nous avons mené deux expérimentations sur deux ensembles de données différents. La première expérience est réalisée sur l'ensemble de données extrait par Hoan Nguyen (cf. Section 4.2). L'ensemble de tests considéré comporte un grand nombre de clients, mais les utilisations à risque n'ont pas été vérifiées. Dans la deuxième expérimentation, l'ensemble de tests comprend des clients dont des erreurs d'utilisation ont été relevées dans le l'ensemble de données MUBench [5]. L'ensemble de tests est donc moins conséquent, mais nous pouvons comparer nos résultats avec une référence établie.

6.2.1. Validation sur l'ensemble de données extrait

6.2.1.1. *Description des concepts de la validation*

Nous allons diviser cette validation en fonction des processus de génération de détecteurs. Pour chacune des trois expérimentations, nous avons procédé à une validation croisée à 10 échantillons. Nous disposons pour chacune des trois APIs étudiées, de deux ensembles d'utilisations. Un étant composé de bonnes utilisations et l'autre de mauvaises. Cette qualification découle du fait que les bonnes utilisations sont des corrections apportées aux mauvaises utilisations (cf. Section 4.2). Pour la réalisation de cette validation croisée à 10 échantillons nous avons donc découpé les deux ensembles d'utilisation de chaque API en 10 sous-ensembles égaux (cf. figure 6.1). Ensuite nous avons exécuté la génération de détecteurs, avec les trois processus différents, en basant notre apprentissage sur 9 des 10 sous-ensembles d'utilisations correctes de l'API et en gardant le dixième sous-ensemble pour évaluer les méthodes qui le composent avec les détecteurs générés à partir des 9 autres sous-ensembles. Nous ajoutons au dixième sous-ensemble de bonnes utilisations, un des 10 sous-ensembles de mauvaises utilisations. Ainsi nous obtenons un ensemble de méthodes à évaluer comportant autant de bonnes et de mauvaises utilisations et comportant donc 10% du total des utilisations relevées.

Lors de cette validation, nous allons évaluer plusieurs critères. Le premier sera la précision. La précision se calcule pour un rang donné. Cette valeur représente le pourcentage de mauvaises utilisations détectées pour un rang donné. C'est le rapport entre le nombre de méthodes qui sont qualifiées comme de mauvaises utilisations et le nombre de méthodes considérées. Ensuite, nous comparerons le score de risque moyen des bonnes utilisations avec le score moyen des mauvaises utilisations pour vérifier que l'approche discrimine bien les bonnes et mauvaises méthodes.

6.2.1.2. *Validation du processus de génération sans regroupement*

Nous allons commencer par décrire et analyser le tableau 6. I. Nous générons donc dans un premier temps les détecteurs sans procéder à aucun regroupement d'utilisations, pour les APIs

FIGURE 6.1. Validation croisée

`java.util.Iterator`, `javax.crypto` et `javax.servlet.http`, avec le paramétrage détaillé dans la Section 6.1.1.

La première configuration testée calcule le score de risque grâce au **max** et prend en considération les arêtes de dépendance de données. Pour l'API `java.util.Iterator` la moyenne des scores des mauvaises utilisations est supérieure à la moyenne des bonnes utilisations de 2% (0,2865 contre 0,2808). Pour `javax.servlet.http`, la discrimination est meilleure, avec 30% de différence entre les moyennes de score de risque. En revanche pour l'API `javax.crypto` le score attribué en moyenne aux bonnes utilisations est supérieur à celui des mauvaises utilisations avec une moyenne de 0,2104 pour les bons usages et de 0,1947 pour les mauvais. Cela signifie que les détecteurs ont majoritairement dysfonctionné dans ce cas. Si l'on regarde la précision des détections, elle est cohérente avec les résultats de la discrimination. Pour l'API `Iterator`, la précision est légèrement au-dessus des 50% avec 53,78% de précision sur les 10% des méthodes avec les scores de risque les plus élevés. Cette précision diminue à 50,18% lorsque l'on regarde les 30% de méthodes les mieux classées. Pour `http`, la précision à 10% est bonne avec 70,0% de précision et elle décroît à 58,89% pour les 30% de méthodes les mieux classées. Et pour `crypto`, la précision est de 50,0% à 10% et croît à 57,14% lorsque l'on considère les 30% de méthodes les mieux classées.

Cette première configuration donne des résultats mitigés. Avec d'une part, les détecteurs qui n'ont pas été efficaces sur l'API `javax.crypto`, avec une précision à 50% pour les 10 premiers pour cent des méthodes les mieux classées. Cette précision est équivalente à un classificateur aléatoire, ce qui n'est pas suffisant. La précision de l'API `java.util.Iterator` est équivalente, avec 53,78%. En revanche, les détecteurs ont montré de meilleurs résultats avec l'API `javax.servlet.http`, grâce à une précision de 70,0% sur les 10% de méthodes évaluées les mieux notées.

La seconde configuration tri les méthodes grâce au `max`, mais cette fois sans prendre en compte les arêtes de dépendance de données. Pour les trois APIs la discrimination entre les bons usages de l'API et les mauvais est correcte, car pour les trois APIs étudiées la moyenne des scores des mauvaises utilisations est supérieure à la moyenne des scores des bonnes utilisations. En particulier pour `javax.servlet.http` où il y a une différence de plus de 28% alors que pour les deux autres cela tourne aux alentours des 5%. En ce qui concerne les précisions, elles sont plus homogènes. La précision pour `java.util.Iterator` a augmenté à 60,81% sur les 10% des méthodes avec le meilleur score et à 54,74% à 30%. Dans le cas de `javax.servlet.http`, la précision à 10% a diminué à 63,33%, mais elle à progressé a 60,0% lorsque l'on considère 30% des données. En revanche, aucun impact n'a été perçu sur la précision de la dernière API.

On peut considérer que le fait de ne pas prendre en considération les arêtes de dépendance de données a un impact positif, car il a fait augmenté la précision générale pour l'API `java.util.Iterator`. Mais le fait d'avoir fait baissé la précision de `javax.servlet.http` et que celle de `javax.crypto` soit restée constante peut laisser penser que la non prise en compte de ces arêtes n'est pas si significative.

Nous allons maintenant prendre en considération la partie droite du tableau 6. I. Cette partie représente les résultats avec les configurations utilisant la méthode d'attribution de score de risque **noisy or** (cf. Section 5.3.2). Dans un premier temps, considérons les résultats en prenant en compte les arêtes de dépendance de données. Les discriminations sont positives, avec comme on peut le constater des moyennes de scores qui ne sont plus du même ordre de grandeur. Cela s'explique par les variétés d'utilisation de chaque API. Du fait que l'API `Iterator` n'a qu'une seule utilisation possible, la majorité des détecteurs vont matcher la méthode évaluée. Ceci explique ces scores au-dessus des 0,9. Alors que pour `crypto` et `http` les utilisations sont plus variées et donc les scores se situent entre 0,3 et 0,55. Les moyennes de score entre bonnes et mauvaises utilisations sont quasi similaires sauf pour l'API `http`, encore une fois, avec une différence de 35%, avec des mauvais usages possédant un score de 0,41 en moyenne et les bons de 0,55. Cela se ressent au niveau des précisions avec 66,67% de précision pour cette API alors que `crypto` est en dessous des 50% de précision avec 45% lorsque l'on évalue les dix premiers pour cent des méthodes les mieux classées. La précision remonte à 54,29% à 30%. Pour `Iterator`, les précisions sont légèrement au-dessus des 50% avec 52,70% et 54,56%.

On peut donc observer que cette configuration utilisant **noisy or** et qui considère les dépendances de données ne donne des résultats intéressants que sur l'API `http`. Dans les deux autres, cas les précisions ne peuvent être considérées comme convaincantes.

La dernière configuration avec le processus de génération sans regroupement utilise aussi **noisy or**, mais ne prend pas en compte les arêtes de dépendance de données. Les discriminations obtenues se révèlent cohérentes avec des écarts de 5% à 15% entre les moyennes de scores de risque des bonnes utilisations et ceux des mauvaises. Ces faibles écarts se traduisent par des précisions situées entre 50% et 55,70% pour les trois APIs. Encore une fois, le fait de ne pas considérer les arêtes de dépendance de données ne permet pas d'améliorer les résultats.

Les conclusions que nous pouvons déduire de ces premières expérimentations avec le processus de génération de détecteurs sans regroupement sont que le fait de ne pas considérer les arêtes de

dépendance de données n'a pas permis d'obtenir de meilleurs résultats. Nous avons testé cette hypothèse, car ces arêtes sont très nombreuses dans les *groum*s et donc ont un poids qui aurait pu se révéler significatif lors du calcul de similarité entre deux *groum*s et plus particulièrement entre un détecteur et la méthode que l'on évalue. Ensuite, en ce qui concerne la méthode de calcul de score le **noisy or** permet d'obtenir de meilleures discriminations de score, donc il permet d'assigner en moyenne des scores de risque plus élevés aux mauvaises utilisations de l'API qu'aux bonnes. Cependant, en ce qui concerne la précision, la méthode de calcul avec **max** fournit des résultats plus convaincants. Avec une précision maximale de 70% pour l'API `javax.servlet.http` lorsque l'on considère les méthodes faisant partie des 10% de scores les plus élevés. On peut donc envisager qu'en utilisant le processus de génération sans regroupement, la méthode d'attribution de score avec le **max** se révèle plus performante. Il vaut mieux privilégier la qualité de la similarité à la quantité. Par la suite, nous allons évaluer les différents processus de génération.

6.2.1.3. *Validation du processus de génération avec regroupement après évolution*

Nous avons donc mené le même lot d'expériences avec le processus de génération de détecteurs avec regroupement après évolution (cf. Section 5.2.4.2) que pour le processus sans regroupement. Cependant, pour ce processus, seulement les APIs `javax.crypto` et `javax.servlet.http` sont retranscrites dans le tableau 6. II, car elles possèdent différents cas d'utilisations qui ont pu être regroupés. Ce qui n'est pas le cas de l'API `java.util.Iterator` qui ne possède qu'un seul cas d'utilisation. Cela signifie que les résultats pour `Iterator` sont identiques aux valeurs présentent dans le tableau 6. I. De nouveau, quatre configurations différentes ont été testées. Analysons les résultats du tableau 6. II.

Dans un premier temps, nous allons étudier les deux expériences qui utilisent le **max** pour assigner le score de risque. Dans le cas de prise en compte des arêtes de dépendance de données, `http` confère des résultats moyens, la moyenne des scores des mauvaises utilisations est supérieure à celle des bonnes et les précisions pour le top 10% et top 30% des méthodes sont respectivement de 53,33% et 57,78%. En revanche, pour `crypto` rien n'est bon. La discrimination n'est pas bonne et les précisions sont en dessous des 50%. Pour `Iterator`, les résultats sont identiques aux processus de génération précédents avec des résultats très moyens.

Donc, pour cette première configuration nous obtenons des résultats très moyens même moins bons que pour le processus de génération sans clustering avec utilisation du **max** et la considération des arêtes de dépendance de données. Cependant, on remarque que les résultats ne sont toujours pas satisfaisants pour l'API `javax.crypto`.

Lorsque l'on ne considère plus les arêtes de dépendance de données on ne change pas le sens des discriminations des deux APIs avec `http` qui attribue bien des scores plus élevés aux mauvaises utilisations alors que c'est l'inverse pour `crypto`. Par contre, les précisions sont remontées avec une précision de 60% obtenue sur les 10% des méthodes aux plus hauts scores pour `http`. Pour `crypto`, la précision a augmenté pour passer de 40% à 45% sur le top 10% et de 48,57% à 53,86% sur le top 30% des méthodes.

Le fait de ne pas prendre en compte les arêtes de dépendance de données bonifie les résultats des APIs pour le processus de génération avec évolution parallèle. Cependant, ils ne sont pas meilleurs

Processus de génération sans regroupement												
	Max						Noisy or					
	Avec DP			Sans DP			Avec DP			Sans DP		
	Iterator	http	crypto	Iterator	http	crypto	Iterator	http	crypto	Iterator	http	crypto
Score moyen bonnes utilisations	0,2808	0,1505	0,2104	0,2469	0,1504	0,1802	0,9246	0,4127	0,3763	0,8818	0,4555	0,3905
Score moyen mauvaises utilisations	0,2865	0,1984	0,1947	0,2648	0,1936	0,1878	0,9483	0,5583	0,3841	0,9207	0,5021	0,4461
Précision moyenne 10% données (%)	53,78	**70,0**	50,0	**60,81**	**63,33**	50,0	52,70	**66,67**	45,0	54,32	50,0	50,0
Précision moyenne 30% données (%)	50,18	58,89	57,14	54,74	**60,0**	57,14	54,56	**66,67**	54,29	55,70	53,33	55,71

TABLEAU 6. I. Résultats d'expérimentation pour le processus de génération de détecteurs sans regroupement. DP signifiant considération des arêtes de dépendance de données.

que les résultats obtenus précédemment sans utiliser le regroupement des utilisations de l'API, mais restent des résultats encourageants avec comme bémol `javax.crypto` encore une fois.

Maintenant, nous allons considérer les résultats du même processus de génération, mais en utilisant la méthode **noisy or** pour assigner les scores de risque aux méthodes. Et comme précédemment nous avons `Iterator` qui évolue très peu en gardant une discrimination correcte et des précisions au-dessus des 50%, `http` qui fournit les meilleurs résultats avec une précision de 60% sur le top 10% des méthodes et une discrimination très bonnes avec des scores de 0,63 en moyenne pour les mauvaises utilisations et 0,46 pour les bonnes. Par contre `crypto` encore une fois déçoit, avec une attribution de scores supérieurs aux bonnes utilisations plutôt qu'aux mauvaises et cela se ressent sur la précision qui chute à 40% pour le top 10%, mais passe tout de même la barre des 50% pour le top 30%.

Donc les résultats de la génération de détecteurs avec évolution parallèle, **noisy or** et dépendance de données donnent des scores légèrement inférieurs à la même configuration sans clustering, mais restent encourageants mis à part la précision sur le top 10% de `crypto`.

La dernière configuration de ce processus de génération fournit des résultats plus uniformes. Les trois APIs discriminent positivement les bonnes utilisations des mauvaises en assignant un score de risque plus élevé aux mauvaises utilisations de l'interface de programmation qu'aux utilisations correctes. De plus, toutes les précisions, que ce soit sur le top 10% ou le top 30% sont supérieures ou égales à 50% allant au-dessus des 55% pour `Iterator`, avec des précisions sur le top 30% légèrement supérieures à celles sur le top 10%. Donc le fait de ne pas prendre en compte les arêtes de dépendance a fait diminuer la précision de l'API `http`, mais a permis `crypto` d'avoir des performances égales à un classificateur aléatoire pour la première fois.

Malheureusement, les résultats du processus de génération avec regroupement après évolution se révèlent moins bons que ceux obtenus sans regroupement des utilisations de l'API. Cependant, parmi ces résultats on remarque une nouvelle fois que la bibliothèque `javax.crypto` produit les résultats les moins probants alors qu'à l'inverse `javax.servlet.http` a tendance à fournir des résultats plus satisfaisants. Enfin, les résultats de l'API `java.util.Iterator` n'évoluent pas, car le clustering ne s'applique pas sur cette interface de programmation à utilisation unique. Nous pouvons aussi remarquer que des quatre configurations testées avec ces deux processus de génération seule l'utilisation du **noisy or** sans prendre en considération les arêtes de dépendance de données a permis d'avoir des résultats toujours au-dessus du classificateur aléatoire sur les trois APIs.

6.2.1.4. *Validation du processus de génération avec regroupement avant évolution*

Notre troisième expérimentation nous amène à étudier le processus de génération avec l'évolution globale, où les détecteurs sont générés à partir de chaque cluster d'utilisation tiré aléatoirement grâce à la technique de la *roulette* pour former une seule population de détecteurs. Les résultats de cette expérience sont compilés dans le tableau 6. III.

Comme nous l'avons fait précédemment, nous allons commencer par considérer les résultats en utilisant l'attribution de score de risque avec **max**. Lorsque l'on considère les arêtes de dépendance de données, les discriminations entre bonnes et mauvaises utilisations sont correctes pour les 3 APIs. En revanche, en ce qui concerne la précision il n'y a pas de corrélation, car pour `http` et

Processus de génération avec regroupement après l'évolution								
	Max				Noisy or			
	Avec DP		Sans DP		Avec DP		Sans DP	
	http	crypto	http	crypto	http	crypto	http	crypto
Score moyen bonnes utilisations	0,1866	0,1841	0,1825	0,1591	0,4653	0,4227	0,5259	0,4077
Score moyen mauvaises utilisations	0,2286	0,1520	0,2118	0,1530	0,6312	0,3830	0,6727	0,4392
Précision moyenne 10% données (%)	53,33	40,0	**60,0**	45,0	**60,0**	40,0	53,33	50,0
Précision moyenne 30% données (%)	57,78	48,57	54,44	53,86	57,78	51,43	55,5	50,0

TABLEAU 6. II. Résultats d'expérimentation pour le processus de génération de détecteurs avec regroupement après l'évolution des populations de solutions. DP signifiant considération des arêtes de dépendance de données.

`crypto` la précision sur les 10% des méthodes avec les scores les plus élevés sont en dessous des 50%, aux alentours des 45%, mais dépassent les 50% lorsque l'on considère le top 30%.

De tels résultats se répètent et même s'amplifient lorsque l'on ne prend plus en compte les arêtes de dépendance de données avec une moyenne de scores de bonnes utilisations qui diminue pour `crypto`, passant de 0,20 à 0,18, mais cela entraîne étrangement une diminution de la précision à 10% avec pour `http` une précision de 43,33% et seulement 30% de précision `crypto`. Mais paradoxalement, la précision à 30% a augmenté pour dépasser les 57% pour les deux APIs et en ce qui concerne `Iterator`, la précision à 30% augmente aussi.

Donc l'utilisation du processus de génération de détecteurs à évolution globale avec le **max** se révèle infructueuse avec des précisions faibles, en dessous des 50% pour `crypto` et `http` lorsque l'on considère le top 10% des méthodes étudiées. Cependant, les précisions à 30% restent uniformes entre les 3 APIs en étant toujours au-dessus des 50% et en ce qui concerne la discrimination, l'approche attribut des scores plus élevés aux mauvaises utilisations de l'interface de programmation qu'aux bonnes utilisations.

La seconde partie de l'expérimentation utilise **noisy or** pour déterminer un score de risque pour chaque méthode étudiée. Lorsque l'on prend en compte les dépendances de données, les scores attribués en moyenne sont plus élevés pour les mauvaises utilisations que pour les usages corrects ce qui est une très bonne chose. Avec une différence de plus de 17% pour `http` et de 23% pour `crypto`. Ces bons résultats se répercutent aussi sur les précisions avec la meilleure précision obtenue par `http` et `crypto` depuis le début des expérimentations, avec respectivement 70% et 55% de précision sur le top 10% et des précisions légèrement en dessous des 60% au top 30%.

Ces bons résultats s'intensifient lorsque l'on ne considère plus les arêtes de dépendance de données, avec un écart encore plus important entre les scores des bonnes et mauvaises utilisations de l'interface d'utilisation. Et cela se traduit aussi sur les précisions qui stagnent pour `http` tout en restant très élevées avec 70% de précision lorsque l'on considère les 10% des méthodes les mieux

notées. Quant aux deux autres APIs leurs précisions ont augmenté et en particulier `crypto` qui atteint des précisions de 60% et 65,71% pour le top 10% et 30%.

En conclusion, si d'une part le processus de génération avec regroupement avant l'évolution fournit des résultats plutôt médiocres lorsque le score est attribué grâce au **max**, ils deviennent bons lorsque l'on utilise **noisy or**, et en particulier lorsque l'on ne considère pas les dépendances de données avec une précision record de 70%.

Processus de génération avec regroupement avant l'évolution								
	Max				Noisy or			
	Avec DP		Sans DP		Avec DP		Sans DP	
	http	crypto	http	crypto	http	crypto	http	crypto
Score moyen bonnes utilisations	0,1641	0,2064	0,1671	0,1880	0,4515	0,4654	0,4644	0,4241
Score moyen mauvaises utilisations	0,1947	0,2129	0,1984	0,2102	0,5303	0,5754	0,5679	0,6167
Précision moyenne 10% données (%)	46,67	45,0	43,33	30,0	**70,0**	55,0	**70,0**	**60,0**
Précision moyenne 30% données (%)	56,67	52,86	57,78	58,57	57,78	58,57	55,56	65,71

TABLEAU 6. III. Résultats d'expérimentation pour le processus de génération de détecteurs avec regroupement avant l'évolution de la population de solutions. DP signifiant considération des arêtes de dépendance de données.

6.2.1.5. *Conclusions*

Après ces trois expérimentations sur l'ensemble de données extrait par nos soins nous pouvons tirer plusieurs conclusions. Dans un premier temps, si l'on considère l'approche que je qualifierais de "naïve", celle qui consiste à n'utilise aucune optimisation, qui est la génération de détecteurs sans clustering en utilisant le **max** et en considérant les arêtes de dépendance de données, les résultats de cette approche naïve sont déjà meilleurs que le classificateur aléatoire. En effet, pour les trois APIs étudiées les précisions sont toutes supérieures ou égales à 50% que ce soit en considérant les 10% des méthodes ayant les plus hauts scores ou en considérant les 30% des meilleures. Cela valide un des premiers objectifs du développement de APIMMUNE.

Ensuite, à cette approche naïve nous avons apporté différentes améliorations. La première est le regroupement des usages de l'API. De ce regroupement découlent deux types d'évolution, l'évolution parallèle et l'évolution globale. La première consiste à générer une population de détecteurs pour chaque cluster d'utilisation de l'API, tandis que la seconde génère une seule population de détecteurs en piochant des utilisations grâce à la technique de *roulette* pour générer ces détecteurs. La seconde modification concerne la façon dont on calcule le score avec d'une part le **max** utilisé dans l'approche naïve et d'autre part le **noisy or**. Enfin, la troisième variation consiste à ne pas prendre en considération les arêtes de dépendance de données des *groum*s.

Si l'on regarde individuellement chacune de ces améliorations, aucune d'entre elles n'apporte à elle seule une amélioration de la précision de la détection. En revanche, en les combinant, les résultats

sont beaucoup plus intéressants. En particulier lorsque l'on utilise **noisy or** sans considérer les arêtes de dépendance de données. Avec cette configuration, peu importe le processus de génération de détecteurs utilisé, les précisions sur le top 10% des méthodes et le top 30% se sont toujours révélées être au-dessus ou égales à 50%, donc au-dessus des performances d'un classificateur aléatoire, le tout en garantissant une discrimination entre les bonnes et mauvaises utilisations de l'interface de programmation correctes. À chaque fois, un score de risque plus élevé est attribué aux mauvais usages plutôt qu'aux usages corrects.

En ce qui concerne l'utilisation du regroupement d'utilisation, on a pu constater que les résultats de l'évolution parallèle sont légèrement inférieurs à ceux produits par le processus de génération de détecteurs sans clustering. En revanche, le processus utilisant l'évolution globale obtient les meilleures performances lorsqu'il utilise **noisy or** et plus particulièrement sans prendre en considération les arêtes de dépendance de données. En combinant toutes les optimisations que nous avons établies on obtient les résultats les plus probants, avec une précision maximale de 70% pour l'API `javax.servlet.http` sur le top 10% des méthodes évaluées.

Ces résultats s'expliquent par le fait que ne pas prendre en compte les arêtes de dépendance de données permet d'accorder moins de poids à ces arêtes qui ne permettent pas de mieux comprendre comment l'interface de programmation est utilisée. Ceci combiné au **noisy or** qui permet de prendre en compte si une méthode matche plusieurs détecteurs tout en privilégiant la qualité des matchs à la quantité. Le troisième facteur étant le clustering des utilisations avec des détecteurs qui évoluent globalement permet que l'ensemble des utilisations puissent être représentées par des détecteurs et que ces détecteurs évoluent tous en même temps vers le même objectif fixé par la fonction de *fitness*. C'est pour ces raisons que les résultats les plus probants combinent ses trois optimisations et répondent à la question de recherche ***RQ1*** sur l'impact des différentes stratégies de génération de détecteurs sur la précision de la détection. De plus, on a en partie un début de suggestion de ce que peut être la réponse à la question de recherche ***RQ3***, car notre approche se montre plus performante que le classificateur aléatoire.

6.2.2. Validation sur des clients de MUBench

Cependant, pour répondre entièrement à la question de recherche ***RQ3*** il faut se comparer sur un ensemble de données commun. C'est pour cela que nous allons expérimenter notre approche sur l'ensemble de données MUBench [5].

Pour cela nous avons extrait de l'ensemble de données MUBench 20 erreurs d'utilisation pour l'API `java.util.Iterator`(mais seulement 13 peuvent être considérées par notre outil), 7 pour `javax.crypto`, mais aucune pour `javax.servlet.http`, car il n'y en avait toujours pas dans MUBench. Par conséquent, la validation peut être réalisée que sur les deux interfaces de programmation ayant des erreurs d'utilisation. En plus d'avoir relevé les erreurs d'utilisation, nous avons aussi récupéré les projets complets dans lesquels ces erreurs d'utilisation sont présentes. Soit 5 projets pour `java.util.Iterator`, dont un avec deux versions et 6 projets pour `javax.crypto`. Parmi ces projets, il y en a qui utilisent à la fois une API et l'autre.

Tous ces projets font que nous avons à disposition plus 21 000 méthodes pour les clients relevés dont 2 100 méthodes qui utilisent l'API `Iterator`. En ce qui concerne `crypto`, nous avons seulement

120 méthodes dont 20 qui utilisent l'API. Nous avons donc généré 50 détecteurs pour ces deux APIs et appliqué la détection d'utilisation de l'API de manière risquée sur l'agglomérat de tous les clients utilisant l'API.

Nous avons compilé tous les résultats dans le tableau 6. IV. Pour les deux APIs nous avons regardé combien de méthodes étaient présentent dans le top 13 du classement `java.util.Iterator` et dans le top 7 pour `javax.crypto`. Ainsi nous pouvons juger combien d'erreurs d'utilisation notre outil est capable de faire remonter en haut du classement des utilisations à risque. Nous avons encore une fois testé la détection avec les trois processus de génération de détecteur, sans regroupement, avec regroupement après évolution et avec regroupement avant évolution. À chaque fois en faisant varier la technique d'attribution du score (**max**, **noisy or**) et en considérant ou non les dépendances de données.

Pour l'API `java.util.Iterator` nous avons les mêmes résultats pour les trois processus, car le clustering d'utilisation ne s'applique pas. Le meilleur résultat obtenu est lorsque nous assignons un score grâce au **max** sans considérer les dépendances de données. En ce qui concerne les résultats pour `javax.crypto`, ils varient en fonction des processus. Pour le processus de génération sans clustering, les résultats sont faibles avec **max** et dépendance de données et pour **noisy or** sans dépendances données, avec seulement une erreur d'utilisation détectée dans le top 8 des méthodes classées. Les deux autres combinaisons ont de meilleurs résultats avec 3 erreurs d'utilisations détectées. Pour le processus de génération avec regroupement après évolution nous avons 2 erreurs d'utilisation dans le top 7 lorsque l'on utilise le **max** et 3 lorsque l'on utilise **noisy or**. Pour le dernier processus de génération avec regroupement avant évolution les résultats sont uniformes avec 3 erreurs d'utilisation détectées dans le top 7 des méthodes classées, peu importe l'utilisation de **max**, **noisy or**, avec ou sans la prise en compte des arêtes de dépendance de données.

Conclusion : Il n'y a pas de variation spectaculaire des résultats en fonction des configurations de APIMMUNE utilisées. Cependant, on peut ressortir de cette expérience deux configurations qui ont bien fonctionné. Ce sont les processus de génération sans clustering et avec un regroupement avant utilisation qui assignent un meilleur score lorsqu'ils utilisent le **max** et ne considèrent pas les dépendances de données. Comme avec la validation sur notre ensemble de données, le processus de génération avec une évolution globale a des résultats supérieurs aux autres. Ceci aide à conforter la première validation. Cependant, cette fois-ci, ce processus de génération est plus efficace en utilisant le **max** plutôt que le **noisy or**.

6.2.2.1. *Comparaison aux outils existants*

Maintenant que nous avons procédé à l'expérimentation et vu quels résultats APIMMUNE est capable de produire, il est temps de comparer nos résultats aux autres outils existants. Les autres outils ont donc été exécutés sur le même corpus de clients pour voir quel est le nombre d'erreurs d'utilisations détectées par chacun. Les outils auxquels APIMMUNE est comparé sont DMMC [**33, 34**], GROUMiner [**42**], Jadet [**62**] et Tikanga [**61**].

La particularité de ces quatre outils c'est qu'ils ne sont pas dépendants de l'API, cela veut dire qu'une seule exécution est nécessaire pour voir le nombre d'erreurs d'utilisations détectées pour l'API `java.util.Iterator` et `javax.crypto`, alors que le nôtre a besoin de deux exécutions, une

Méthodes dans le top #erreurs d'utilisation								
	Max				Noisy or			
	Avec DP		Sans DP		Avec DP		Sans DP	
	Iterator	crypto	Iterator	crypto	Iterator	crypto	Iterator	crypto
Processus de génération sans clustering	0	1	**1**	**3**	0	3	0	1
Processus de génération avec clustering après évolution	0	2	1	2	0	3	0	3
Processus de génération avec clustering avant évolution	0	3	**1**	**3**	0	3	0	3

TABLEAU 6. IV. Résultats de validation pour les erreurs d'utilisation de MU-Bench. DP signifiant considération des arêtes de dépendance de données.

pour chaque API pour utiliser le bon lot de détecteurs. L'autre différence réside dans le fait que ces quatre outils utilisent les patrons d'utilisation des APIs à contrario de notre approche qui est plus globale et apprend par le nombre.

Les résultats de l'expérimentation sont présents dans le tableau 6. V. En ce qui concerne l'API `java.util.Iterator`, APIMMUNE se classe en troisième position avec une précision de 8%, derrière Tikanga et ses 23% de précision et DMMC avec 14% de précision. Deux outils qui utilisent les patrons d'utilisation pour la détection. Les deux dernières places sont prises par GROUMiner et Jadet qui n'ont détecté aucune erreur d'utilisation. En ce qui concerne l'interface de programmation `javax.crypto` notre approche a de meilleurs résultats avec 3 erreurs d'utilisation trouvées, ce qui fait une précision de 43%. Les quatre autres outils en revanche n'ont pas été capables de détecter la moindre erreur. Ceci est en grande partie dû au fait qu'il n'y avait pas assez d'utilisations pour générer des patterns d'utilisation.

Donc on constate que, sur un échantillon relativement réduit, APIMMUNE permet de détecter certaines erreurs que les autres outils ne détectent pas (cas de `javax.crypto`). Ceci laisse a penser que notre approche reste pertinente. La faible performance dans le cas d'`Iterator` s'explique par le fait que cette API est petite avec seulement quatre méthodes, donc il est très simple d'extraire des patrons d'utilisation pour les autres outils. `crypto`, par contre, est une interface de programmation composée de 10 classes et moins populaire que `Iterator`, c'est pour cela que l'extraction de pattern est beaucoup plus difficile. Ainsi, nous montrons que notre approche peut être vraiment complémentaire des travaux déjà réalisés dans le domaine, qui comme le montre les résultats du tableau 6. V sont encore perfectibles à tous points de vue. Cela laisse penser qu'une utilisation hybride entre l'*API groum* et le patron d'utilisation serait intéressante à explorer, afin de cumuler les avantages des deux approches et améliorer la détection sur tous les types d'API.

6.2.3. Description des résultats obtenus

Après ces expérimentations sur ces deux ensembles de données, une tendance ressort à propos des trois processus de génération de détecteurs. On constate que le processus de génération sans

Outil	Iterator		crypto	
	Précision	Misuses	Précision	Misuses
APIMMUNE	8%	1	43%	3
DMMC	14%	2	0%	0
GROUMiner	0%	0	0%	0
Jadet	0%	0	0%	0
Tikanga	23%	3	0%	0

TABLEAU 6. V. Comparaison des précisions obtenues par les différentes approches de détection

clustering obtient des résultats qui sont satisfaisants, alors que le processus avec évolution parallèle quant à lui ne semble pas stable et ne fournit pas de bons résultats. Le dernier processus de génération avec évolution globale pour sa part a fourni les meilleurs résultats lorsqu'on ne considère pas les arêtes de dépendance de données.

Ces résultats constatés se sont ressentis lors de la comparaison avec les autres approches étudiées. Même si ces résultats ne sont pas aussi bons que ceux obtenus lors de la première phase expérimentale sur l'ensemble de données extrait de *GitHub*, ces résultats sont très honorables et encourageants sur MUBench.

6.3. PERFORMANCE

Maintenant que nous avons validé l'approche de notre outil APIMMUNE. Il est intéressant de regarder quelles sont ses performances en terme de temps et de mémoire.

6.3.1. Mémoire

Dans un premier temps, en ce qui concerne la mémoire, il y a deux types de mémoire à étudier la mémoire dure, qui est le volume de stockage nécessaire pour utiliser le programme et la mémoire vive qui est la quantité de mémoire nécessaire pour que le programme s'exécute de manière optimale. Ces performances ont été calculées sur un DELL XPS 15, avec processeur Intel Core i7 7ème génération cadencé à 2,8 GHz, avec 16 Go de RAM et 500 Go d'espace de stockage.

6.3.1.1. *Capacité de stockage*

APIMMUNE a besoin d'espace de stockage sur plusieurs aspects. L'espace de stockage est principalement utile pour sauvegarder les détecteurs. Toutes les capacités nécessaires pour l'utilisation de APIMMUNE sont récapitulées dans le tableau 6. VI. On y voit la taille des clients à partir desquels les détecteurs sont générés. On remarque qu'il y a plus de cas d'utilisation pour `Iterator` avec 1,1 Go de clients alors que pour `crypto` et `http` c'est seulement 183 Mo et 273 Mo. Il est aussi possible de stocker les *API groum*s pour différents cas d'études, mais ce n'est pas indispensable. On remarque donc que nous avons 1 820 *API groum*s pour `Iterator`, 135 pour `crypto` et 478 pour `http` (ceci est le nombre d'*API groum*s de bonnes utilisations, il y en a autant pour les mauvaises). Donc le nombre de clients de l'API fait varier le nombre de méthodes utilisant l'API de manière plutôt proportionnelle. Ensuite pour une cinquantaine de détecteurs il faut entre 14

Mo et 251 Mo de mémoire pour les stocker. Cette large variation peut s'expliquer par la taille des utilisations d'une API. On remarque que la taille moyenne d'une utilisation `Iterator` est de 0,6 Mo, pour `crypto` de 0,1 Mo et 0,6 Mo pour `http`. Ce qui se traduit avec les détecteurs où la taille moyenne par détecteur est de 2,22 Mo pour `Iterator`, 0,29 Mo pour `crypto` et 5,02 Mo pour `http`. Cette taille varie en fonction du besoin de combiner plus ou moins de méthodes et de structures de contrôle pour utiliser l'interface de programmation.

On peut conclure qu'il faut au moins 251 Mo de stockage pour pouvoir lancer une détection avec APIMMUNE. Toutes les autres capacités de stockage sont seulement nécessaires pour la génération des détecteurs.

	Iterator	crypto	http
Taille de l'entrée (Go)	1,1	0,0183	0,273
Taille de la sauvegarde des API *groums* (Go)	16,8	0,843	3,26
Nombre d'API *groums*	1 820	135	478
Taille de la sauvegarde des détecteurs (Mo)	111	14	251
Nombre de détecteurs	50	48	50

TABLEAU 6. VI. Espace de stockage utilisé

6.3.1.2. *Mémoire vive*

Durant la génération des détecteurs pour connaître la taille de l'espace de stockage nécessaire à nos détecteurs nous avons utilisé l'outil `JVM Monitor`[1] qui est un plugin de l'IDE `Eclipse`. Cet outil nous a permis de connaître la mémoire utilisée lors de la génération des détecteurs pour les trois APIs étudiées. Ces valeurs sont inscrites dans le tableau 6. VII.

On constate que pour les trois exécutions, le même espace mémoire a été alloué (3,695 Go) et que les trois processus ont, à peu de choses près, utilisé la même quantité de mémoire. À savoir pour `Iterator` 1254 Mo de mémoire vive, 1035 Mo pour `crypto`, et pour ce qui concerne `http` 1038 Mo. On peut remarquer que ces valeurs sont directement reliées à la taille des clients de chaque interface de programmation, car plus l'interface a de clients qui l'utilisent plus l'espace mémoire utilisé est important. Cependant, les variations de clients sont moins conséquentes que les écarts de ressources de stockage. Ainsi, par exemple, les clients d'`Iterator` prennent 4 fois plus d'espace disque, mais seulement 1,2 fois plus de mémoire vive est consommée à l'exécution.

D'après les relevés fournis par `JVM Monitor` au maximum 1,254 Go de mémoire vive aura été nécessaire pour générer 50 détecteurs.

6.3.2. Temps

En ce qui concerne le temps, il y a deux valeurs de temps à calculer. Il y a d'une part, le temps de génération des détecteurs et d'autre part, le temps de détection de ces détecteurs. La

1. `http://jvmmonitor.org/` (accédé le 7 mai)

	Iterator	crypto	http
Max heap memory (Mo)	3695	3695	3695
Used heap memory (Mo)	1254	1035	1038

TABLEAU 6. VII. Mémoire vive utilisée

génération de détecteurs n'est nécessaire qu'une seule fois. Même s'il est possible de générer par exemple plusieurs lots de détecteurs en fonction des versions de l'API par exemple. En tout cas c'est une étape ponctuelle qui est à faire quand on le désire, mais pas régulièrement.

Nous avons donc calculé le temps nécessaire pour générer des détecteurs pour les trois APIs que nous étudions depuis le début de ce mémoire, qui sont, `java.util.Iterator`, `javax.crypto` et `javax.servlet.http`. Pour chaque interface, nous avons décomposé le temps de génération des détecteurs en trois sous étapes qui sont la génération des *API groum*s, le clustering et la génération des détecteurs.

La première sous étape de génération des *API groum*s peut prendre entre 5 secondes et 9 minutes 30 en fonction du nombre de clients. Comme nous l'avons vu précédemment, l'API `Iterator` possède énormément d'utilisations desquelles apprendre, d'où le temps plus long. Alors que `crypto` possède nettement moins d'utilisations, combiné au fait que ces utilisations sont plus petites, cela explique le temps de génération plus court.

Ensuite, le temps d'exécution de SPMF - dbcsan [2] est en dessous des 15 ms avec 14 ms pour `Iterator` et 11 ms pour les deux autres interfaces qui sont plus petites.

Enfin, en ce qui concerne le temps de génération et de stockage des détecteurs il faut plus de 18 minutes pour `Iterator`, environ 14 minutes pour `crypto` et environ 53 minutes pour `http` qui est l'API avec la plus grande taille d'utilisation en terme de mémoire.

Si l'on cumule tous ces temps, on constate qu'il faut entre un quart d'heure et une heure pour générer les détecteurs de ces trois interfaces de programmation. Cela peut paraître long, mais les détecteurs n'ayant besoin d'être générés qu'une seule fois ou très rarement cela reste un temps que l'on peut juger acceptable s'il est prévenu.

	Iterator		crypto		http	
	Temps	Cumulé	Temps	Cumulé	Temps	Cumulé
Génération des *API groum* (min)	09 :30	09 :30	00 :05	00 :05	04 :46	04 :46
DBSCAN (min)	00 :00 :014	09 :30	00 :00 :011	00 :05	00 :00 :011	04 :46
Génération de détecteurs (min)	18 :43	28 :13	13 :54	13 :59	53 :15	58 :01

TABLEAU 6. VIII. Temps de génération des détecteurs

Le temps le plus important est le temps de détection. Alors ce temps se décompose en plusieurs étapes. En premier lieu, il faut générer les *API groum*s pour chaque méthode du client utilisant l'interface de programmation. Ces temps sont calculés uniquement pour les APIs `java.util.Iterator`

2. http://www.philippe-fournier-viger.com/spmf/ (accédé le 7 mai)

et `javax.crypto`, car nous avons réalisé ce chronométrage avec les clients extraits pour la validation avec MUBench [5]. L'extraction des *API groum*s est indispensable pour pouvoir les comparer aux détecteurs qui sont eux aussi des *groum*s. La durée de génération des *groum*s pour `Iterator` est de 22,681 secondes et de 9,39 secondes pour `crypto`. Et la durée de simplification des *groum*s pour obtenir les *API groum*s est de 1,963 secondes pour `Iterator` et 0,024 seconde pour `crypto`. Les résultats sont présents dans le tableau 6. IX et s'expliquent par le fait que pour `Iterator` environ 2 100 *API groum*s sont générés à partir de 21 000 méthodes clients et que pour `crypto` seulement 20 *API groum*s sont générés à partir de 120 méthodes clients. Donc le nombre de méthodes fait varier le temps d'extraction des *API groum*s. Ensuite, il y a le temps de dé-sérialisation des détecteurs, car nous avons besoin de stocker les détecteurs afin de diminuer le temps d'exécution de l'outil. Ainsi nous passons d'un temps de génération de plus de 28 minutes et de 14 minutes à un temps de dé-sérialisation de seulement d'une dizaine de secondes. Enfin, le temps d'attribution d'un score varie lui aussi en fonction du nombre de méthodes avec 8,813 secondes pour `Iterator` et 0,162 seconde pour `crypto`. Ce qui fait en moyenne 6 ms pour attribuer un score à une méthode.

Donc on constate que la durée d'exécution de l'outil APIMMUNE varie en fonction du nombre de méthodes dans le client et le nombre de méthodes utilisant l'API. Cependant, l'ensemble de tests pour `Iterator` de 21 000 méthodes est un agglomérat de clients. Ce qui fait qu'il y a très peu de chances d'avoir autant de méthodes à évaluer. Mais au final aucune exécution ne prend plus de 45 secondes, et celle de `crypto` ne prend qu'une dizaine de secondes.

	Iterator		crypto	
	Temps	Cumulé	Temps	Cumulé
Construction des *groum*s (s)	22,681	22,681	09,390	09,390
Construction des *API groum*s (s)	1,963	24,644	0,024	09,414
Dé-sérialisation des détecteurs (s)	10,341	34,985	0,738	10,152
Temps d'attribution de score (s)	8,813	43,798	0,162	10,314

TABLEAU 6. IX. Temps d'évaluation d'un client

6.4. DISCUSSION

Maintenant que nous avons tous les résultats obtenus lors des expérimentations, nous allons les discuter au sein de cette Section. Nous allons juger l'efficacité des détecteurs générés, puis les performances de APIMMUNE, avant de le comparer à d'autres outils de détection d'erreur d'utilisation. Et nous finirons par aborder les limitations.

6.4.1. Les processus de génération de détecteurs

Au sein de notre approche APImmune nous avons exploré différentes techniques de génération de détecteurs. En plus de ces différentes techniques, nous avons étudié les forces et les faiblesses de notre approche pour y apporter des améliorations potentielles pour combler les problèmes rencontrés. Et plus particulièrement l'utilisation du **noisy or** pour assigner un score de risque et l'absence de considération des arêtes de dépendance de données.

Nous avons donc testé notre approche dans un premier temps sur un ensemble de données extrait des meilleurs projets GitHub et considérant deux fragments de code à un moment `t` et à un moment `t+1`. Le plus récent fragment de code étant la correction du premier nous avons qualifié de mauvaise utilisation le fragment de code au moment `t` et de bonne utilisation celui au moment `t+1`.

Ainsi nous avons réalisé une validation croisée à 10 ensembles pour chaque combinaison de processus de génération, avec clustering, avec évolution parallèle et avec évolution globale. En variant à chaque fois la technique d'attribution de score et la prise en compte des arêtes de dépendance de données. Cette expérimentation nous a permis de voir que l'approche naïve, qui n'utilise pas le regroupement des utilisations de l'API, assigne un score grâce au **max** et considère les arêtes de dépendance de données des *groum*s, se révèle déjà meilleure que le classificateur aléatoire. Ce premier résultat est encourageant, car il montre que notre approche peut apporter quelque chose. Ensuite, nous avons montré que sur cet ensemble de données notre approche donnant les meilleurs résultats est celle qui utilise l'évolution globale, en assignant des scores de risque grâce à la formule du **noisy or** et ne considère pas les dépendances de données des *groum*s.

Cela nous montre que nos intuitions se sont montrées relativement justes. Le fait d'utiliser l'évolution globale confirme que regrouper les utilisations de l'API permet d'avoir des détecteurs plus efficaces, car ils sont plus représentatifs de l'API. Attribuer des scores de risque avec **noisy or** permet de ne pas seulement prendre en compte la qualité de la meilleure détection, mais de balancer cette qualité avec la quantité. Et finalement le fait de ne pas considérer les arêtes de dépendance de données permet de réattribuer un poids plus important aux autres éléments de l'*API groum*, qui sont plus nécessaires à la compréhension de l'utilisation de l'API.

Cette expérience nous a permis de répondre à notre première question de recherche (***RQ1*** *: Quel est l'impact des différentes stratégies de génération de détecteurs sur la précision de la détection ?* Nous avons démontré que notre stratégie la plus complète améliore les résultats en permettant d'obtenir une précision en moyenne supérieure de 10% à notre approche naïve.

6.4.2. Les performances

En ce qui concerne les performances, APImmune nécessite un bon nombre de ressources. En ce qui concerne la mémoire, notre approche a absolument besoin d'espace pour stocker les détecteurs. Cet espace de stockage varie en fonction du nombre de détecteurs et du volume de l'utilisation de l'API. On entend par volume, le fait que l'API lors de son utilisation nécessite un grand nombre d'appels à des méthodes et structures de contrôle ou non. D'après nos observations, une cinquantaine de détecteurs peut prendre jusqu'à 251 Mo d'espace disque sur une machine.

Pour ce qui est de la mémoire vive, l'approche a utilisé au plus 1,254 Go. Sachant que de nos jours il existe très peu de machines possédant moins de 2 Go de mémoire vive, APIMMUNE devrait pouvoir être exécutable sur une très grande partie des machines.

Le dernier aspect en rapport avec la performance est le temps d'exécution. Celui-ci se décompose en deux parties, les deux étapes de notre approche. Dans un premier temps, il faut générer les détecteurs. Ce processus peut être long en fonction de la taille l'ensemble d'apprentissage, car plus il est grand plus il a de méthodes à considérer et à regrouper en fonction des utilisations. Autre facteur qui peut jouer, c'est encore une fois le volume des usages de l'API, qui plus ils sont grands, plus les manipulations des *API groum*s est longue. Enfin, le dernier facteur est bien évidemment le nombre de détecteurs que nous voulons générer, car plus nous voulons générer de détecteurs plus cela prend du temps, pour des conditions de génération identiques, car les conditions de génération peuvent influencer le temps de génération. Un plus grand nombre de générations ou une population de solutions plus grande augmentent par conséquent le temps de génération. Lors de notre expérimentation avec les paramètres spécifiés nous avons obtenu un temps de génération variant de 14 à 58 minutes.

La seconde étape de l'approche est la détection, qui également, a un temps d'exécution qui dépend de deux facteurs. Le premier est le nombre de méthodes à évaluer, car plus il y en a et plus cela prend de temps de comparer chaque méthode à tous les détecteurs. Le second facteur est le nombre de détecteurs pour les mêmes raisons. Ce qui fait que lors de l'expérimentation nous avons obtenu des résultats variés encore une fois avec un temps de détection de 42,798 secondes et un de 10,314 secondes.

Ainsi nous avons pu répondre à la deuxième question de recherche (***RQ2)*** *: Quel est le coût en terme de temps et de mémoire de notre approche ?* En ce qui concerne la mémoire disque, il est préférable d'avoir au moins 251 Mo de libres pour sauvegarder les détecteurs. Pour ce qui est la mémoire vive, une machine avec au moins 2 Go de RAM est conseillée. Pour le temps d'exécution, les valeurs sont très dépendantes des entrées. En ce qui concerne la génération des détecteurs, il faut compter entre 15 et 60 minutes, ce qui ne pose pas forcément problème, car la génération ne se fait pas fréquemment. Pour la détection ; il faut envisager entre 10 et 45 secondes. Tout cela si l'on considère 50 détecteurs.

6.4.3. Comparaison aux outils existants

Enfin, nous avons comparé la précision entre notre approche APIMMUNE et quatre autres outils DMMC [**33, 34**], GROUMiner [**42**], Jadet [**62**] et Tikanga [**61**], sur un ensemble de 7 erreurs d'utilisation de l'API `javax.crypto` et 13 erreurs d'usage de l'interface `java.util.Iterator`. Au classement des précisions notre outil se classe à la troisième position pour `Iterator`, mais à la première position en ce qui concerne `crypto`, car aucun des outils n'a été capable de détecter la moindre erreur d'utilisation.

Cela nous permet de répondre à la dernière question de recherche (***RQ3)*** *: Où se situe notre approche par rapport aux approches déjà existantes ?* Donc notre approche ne surpasse pas les autres en tout point. Nous avons mis au point une approche permettant de compléter les travaux existants. Comme nous l'avons vu dans l'état de l'art (cf. Section 2), une grande partie des techniques de

détections se basent sur les patterns d'utilisation. Cette technique se révèle efficace et en particulier sur des interfaces de programmation comme `java.util.Iterator` où les utilisations sont faciles à, caractériser. Notre approche quant à elle vise plus à regrouper les utilisations en fonction des appels faits par les méthodes pour générer les détecteurs. Et donc au lieu de voir à quelle distance d'un comportement juste se situe la méthode on regarde à quel point une méthode est proche d'un mauvais comportement. Donc notre approche se situe en termes de résultats proches des autres en ce qui concerne la précision, mais plus éloignée en fonction du type de résultats. Ce qui explique ces différences de précision, c'est que APIMMUNE détecte des erreurs d'utilisation de différents types alors que les autres outils détectent grâce à leur patron d'utilisation toutes les erreurs ayant le même patron. Et comme les erreurs d'utilisation sur `java.util.Iterator` sont majoritairement du même type, appel manquant, leurs autres outils performent mieux.

Comme évoqué précédemment, une singularité de notre approche c'est que les erreurs d'utilisation détectées sont toutes de types différents. Ces types sont l'oubli d'appel de fonction, l'oubli d'appel d'exception et l'oubli de vérification d'une valeur. Alors que les outils sont spécialisés dans la détection d'un type d'erreur d'utilisation comme nous l'avons vu dans l'état de l'art, APIMMUNE est capable d'attaquer le problème par différents aspects ce qui renforce sont aspect complémentaire aux travaux déjà réalisés.

6.4.4. Menace à la validité

Les résultats obtenus lors de l'expérimentation sont tout de même à nuancer en présence de certaines menaces à la validité de l'étude. Tout d'abord, dans le cas de l'API `javax.crypto`, c'est nous même qui avons exécuté les autres outils et non pas leurs auteurs. Il est possible que certains calibrages aient été possibles. Le choix d'utiliser `javax.crypto`dans la comparaison est motivé par le fait que c'est une API populaire et souvent prise en exemple pour les travaux concernant les erreurs d'utilisation. Par ailleurs, des erreurs de son utilisation sont répertoriées dans MUBench.

Toujours en ce qui concerne la comparaison avec les autres outils, nous n'avons pas testé notre approche avec les autres APIs utilisées pour la validation de leurs outils. Ceci n'a pas été possible, car l'utilisation de APIMMUNE requière un volume élevé de données d'entraînement, données qui n'étaient pas disponibles.

Une autre menace possible à la validité concerne la qualité des utilisations sur lesquelles nous avons basé la génération des détecteurs. En effet, pour cette génération, nous considérons qu'une utilisation est correcte s'il existe un commit dans lequel le fragment de code a été corrigé. Cependant, rien ne prouve que le résultat de la correction est lui-même une bonne utilisation de l'API. Mais le fait d'apprendre par le nombre veut dire que nous considérons comme bonne utilisation l'utilisation la plus commune donc il n'y a pas tant de raison de penser que notre approche pourrait être biaisée. C'est plus lors de la détection où l'on utilise une partie de ces bons et mauvais usages que notre détection pourrait être faussée, mais la validation sur MUBench a montré que nos résultats sont cohérents.

6.4.5. Limitations

Malgré les résultats satisfaisants obtenus, notre approche présente quelques éléments qui méritent des améliorations. Une première limite concerne le nombre de cas pour chacune des APIs. Nous avons réussi à relever plus de cas pour `java.util.Iterator` que pour `javax.crypto` et `javax.servlet.http`. `Iterator` compte 1 820 cas de bonnes utilisations pour l'apprentissage des détecteurs, `crypto` en compte que 135 et `http` 478, ce qui fait de gros écarts et en particulier pour `Iterator` qui en possède beaucoup plus que les autres. Cela fait que lors de la première validation sur notre ensemble de données que pour les deux interfaces avec moins d'exemples il y avait plus de variation des résultats et surtout que le top 10% comportant moins de méthodes la précision peut être rapidement plus élevée. Cependant, la précision sur le top 10% était le meilleur moyen d'obtenir des résultats comparables. En ce qui concerne la seconde évaluation sur MUBench, la différence de résultats entre `java.util.Iterator` et `javax.crypto` peut s'expliquer par le fait que nous cherchions 13 erreurs d'utilisations parmi 2 100 méthodes, donc il est plus compliqué de faire remonter une erreur dans les 13 premières positions du classement des méthodes à risque que pour `crypto` où nous cherchions 7 erreurs parmi 20 méthodes, mais nous n'avions pas le choix.

Générer seulement une cinquantaine de détecteurs peut être vu comme peu et nous pourrions envisager d'en produire une plus grande quantité. Mais lors de l'étude en détail de nos détecteurs nous avons découvert qu'ils n'étaient pas aussi diversifiés qu'attendu. Dans le tableau 6. X nous pouvons voir qu'en moyenne seulement un tiers des détecteurs sont différents les uns des autres. C'est qu'au lieu d'avoir par pour `Iterator` 46 détecteurs nous n'en avons que 15 uniques. Ceci s'explique par le fait qu'un détecteur est généré en altérant (mutant) une seule fois une bonne utilisation ce qui apporte seulement une légère variation de la bonne utilisation, et que nous voulons que nos détecteurs restent dans un intervalle de similarité relativement proche des bonnes utilisations (entre 66% et 99% de similarité). Cet intervalle bloque donc tous les gros bouleversements d'utilisation et fait que seulement les mutations qui ne s'éloignent pas trop restent considérées. Voilà pourquoi nos détecteurs manquent de diversité. Cette problématique mérite un travail complémentaire qui pourrait permettre d'obtenir des résultats potentiellement supérieurs à ceux que nous avons.

Processus	Sans clustering		Évolution parallèle	Évolution globale
API	Iterator	crypto	crypto	crypto
Nombre de détecteurs uniques	15	13	12	16
Nombre de détecteurs	46	48	48	46

Tableau 6. X. Variations des détecteurs

La dernière limite de notre approche réside dans le fondement de celle-ci. C'est le fait que APImmune est dépendant de l'API que l'on étudie. Nous sommes obligés de générer les détecteurs pour une API particulière, il n'est pas possible d'en vérifier plusieurs à la fois. Et le fait que la génération des détecteurs prenne du temps n'aide pas. Cependant, les détecteurs ayant besoin d'être générés seulement une seule fois, la contrainte devient assez faible. De plus, notre détection se veut être une vérification de fin de projet en complément aux autres approches. Elle est là pour

venir vérifier à la fin d'un projet lors de la phase de test qu'il n'y a pas d'utilisation à risque de l'API.

Chapitre 7

CONCLUSION

Nous avons proposé à travers ce mémoire une méthode de détection d'utilisation à risque d'API. Pour une API donnée notre approche APIMMUNE génère à partir d'exemples de bonnes utilisations de l'API des détecteurs que l'on peut assimiler aux lymphocytes T du système immunitaire. Ces détecteurs vont ensuite matcher les utilisations à risque de l'API comme lorsque que le système immunitaire détecte un antigène *non self*.

Le mécanisme de génération des détecteurs apprend grâce aux bonnes utilisations faites de l'API dans les projets populaires de *GitHub*. Ces utilisations sont ensuite regroupées en fonction des méthodes employées et les détecteurs sont générés à partir de ces groupes en mutant les bonnes utilisations. Une population initiale de détecteurs étant maintenant créée, elle évolue grâce à un algorithme génétique qui permet le brassage des détecteurs afin d'obtenir une solution finale de détecteurs avec des individus les plus diverses les uns des autres et avec une similarité relativement proche des utilisations de l'API. Le mécanisme de détection consiste quant à lui à comparer une méthode évaluée, aux détecteurs. Et plus la méthode matche plus elle est considérée à risque.

Notre approche APIMMUNE a été validée sur deux ensembles de données. La première validation est une validation croisée à 10 échantillons sur un ensemble d'utilisations de l'API que l'on considère mauvaises car elles possèdent une correction. Cette correction est considérée comme la bonne utilisation. Nous avons ainsi testé notre approche avec 3 APIs Java populaires, `java.util.Iterator`, `javax.crypto` et `javax.servlet.http` et montré que notre approche est meilleure qu'un classificateur aléatoire et peut même atteindre une précision de 70% dans la détection des utilisations à risque. Le second ensemble de validation MUBench nous a permis de nous comparer aux autres travaux réalisés dans le domaine en procédant à une détection sur des projets utilisant les APIs `java.util.Iterator` et `javax.crypto`. APIMMUNE ne se classe qu'en troisième position pour la détection des erreurs d'utilisation sur l'interface `Iterator` mais c'est le seul outil capable de détecter les erreurs dans les clients de l'interface `crypto`. De plus les erreurs détectées sont de types différents et montrent que APIMMUNE peut vraiment être un complément aux approches existantes.

Malgré les résultats encourageants il y a encore des explorations à réaliser avec APIMMUNE. Nous l'avons vu, l'approche possède énormément de paramètres qu'il a fallu fixer. Il serait intéressant d'explorer les résultats avec des paramètres différents pour voir leur impact sur la détection.

Comme évoqué dans les limitations, nous pourrions augmenter le nombre de détecteurs mais il faut d'abord régler le problème de redondance des détecteurs avec la variation des paramètres.

Nous avons mis au point une approche basée sur le système immunitaire qui apprend des usages d'une API sans avoir besoin que le code source soit révélé. Une entreprise pourrait très bien aider à la génération de détecteurs sans pour autant révéler ses secrets industriels. Notre approche vise à compléter les travaux réalisés, un axe de progression serait de renforcer cette détection afin de fournir la solution la plus complète possible. Pour y parvenir il faudrait sûrement s'inspirer de plusieurs techniques existantes pour les associer afin de détecter les différents types d'erreur d'utilisation. De plus notre approche ne s'est intéressée qu'aux APIs Java, mais un grand nombre d'APIs existent dans différents langages de programmation donc une voie de recherche s'ouvre à nous de ce côté-ci du domaine aussi.

Le sujet de la détection d'erreur d'utilisation d'API est très complexe, car ces erreurs sont de légères déviances de l'usage correct, donc de nombreux travaux peuvent encore être réalisés dans ce domaine de recherche passionnant. Dans un premier temps, il pourrait être intéressant de continuer l'exploration des différents paramètres de notre approche. Cela permettrait de l'optimiser. Toujours en termes d'exploration, il serait intéressant de continuer le travail avec d'autres APIs ou types d'APIs. Ceci nous permettrait d'affiner le processus de génération de détecteurs. Une autre piste d'amélioration, évoquée lors de la validation, est l'exploration de la combinaison de notre approche avec une approche basée sur les patrons pour augmenter la couverture des types d'erreurs détectés.

Bibliographie

[1] Hani Abdeen, Khaled Bali, Houari Sahraoui et Bruno Dufour : Learning dependency-based change impact predictors using independent change histories. *Inf. Softw. Technol.*, 67(C):220–235, novembre 2015.

[2] Mithun Acharya et Tao Xie : Mining API error-handling specifications from source code. *In Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering : Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 370–384. Springer-Verlag GmbH, 2009.

[3] Arshad Ahmad, Chong Feng, Shi Ge et Abdallah Yousif : A survey on mining stack overflow : question and answering (q&a) community. *Data Technologies and Applications*, 52(2):190–247, 2018.

[4] Mansour Ahmadi, Angelo Sotgiu et Giorgio Giacinto : Intelliav : Building an effective on-device android malware detector. *CoRR*, abs/1802.01185, 2018.

[5] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen et M. Mezini : Mubench : A benchmark for api-misuse detectors. *In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 464–467, May 2016.

[6] Stephen J. Andriole : The death of big software. *Commun. ACM*, 60(12):29–32, novembre 2017.

[7] SungGyeong Bae, Hyunghun Cho, Inho Lim et Sukyoung Ryu : Safewapi : Web api misuse detector for web applications. *In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 507–517, New York, NY, USA, 2014. ACM.

[8] Islem Baki et Houari Sahraoui : Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Trans. Softw. Eng. Methodol.*, 25(3):20 :1–20 :37, juin 2016.

[9] S. Beyer et M. Pinzger : A manual categorization of android app development issues on stack overflow. *In 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 531–535, Sept 2014.

[10] A. M. Bradley, D. W.and Tyrrell : Immunotronics : Hardware fault tolerance inspired by the immune system. *In* Julian Miller, Adrian Thompson, Peter Thomson et Terence C. Fogarty, éditeurs : *Evolvable Systems : From Biology to Hardware*, pages 11–20, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[11] R. P. L. Buse et W. Weimer : Synthesizing api usage examples. pages 782–792, June 2012.

[12] Yuehua Cao et Dipankar Dasgupta : *An Immunogenetic Approach in Chemical Spectrum Recognition*, pages 897–914. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[13] Google Cloud : Guide sur le développement d'applications ouvertes à l'usage des responsables informatiques. 2017.

[14] Ting Dai, Sai Sathyanarayan, Roland H. C. Yap et Zhenkai Liang : Detecting and preventing activex api-misuse vulnerabilities in internet explorer. *In Proceedings of the 14th International Conference on Information and Communications Security*, ICICS'12, pages 373–380, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] D. Dasgupta, Z. Ji et F. Gonzalez : Artificial immune system (ais) research in the last five years. *In Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 1, pages 123–130 Vol.1, Dec 2003.

[16] Dipankar Dasgupta et Stephanie Forrest : *An Anomaly Entection Algorithm Inspired by the Immune Syste*, pages 262–277. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[17] Michel Marie Deza et Elena Deza : *Encyclopedia of Distances*. 2009.

[18] Manuel Egele, David Brumley, Yanick Fratantonio et Christopher Kruegel : An empirical study of cryptographic misuse in android applications. *In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

[19] Martin Ester, Hans-Peter Kriegel, Jörg Sander et Xiaowei Xu : A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. *In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[20] Dave Evans : The internet of things how the next evolution of the internet is changing everything. 2011.

[21] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood et Philippa Gardner : Javert : Javascript verification toolchain. *Proc. ACM Program. Lang.*, 2(POPL):50 :1–50 :33, décembre 2017.

[22] Bernhard Ganter et Rudolf Wille : *Formal Concept Analysis : Mathematical Foundations*. Springer-Verlag New York, Inc., 1st édition, 1997.

[23] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh et Vitaly Shmatikov : The most dangerous code in the world : Validating ssl certificates in non-browser software. *In Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[24] Elena L. Glassman, Tianyi Zhang, Björn Hartmann et Miryung Kim : Visualizing api usage examples at scale. pages 580 :1–580 :12, 2018.

[25] Richard A. Goldsby, Thomas J. Kindt, Janis Kuby et Barbara A. Osborne : *Immunology fifth edition*. 2002.

[26] Daqing Hou et David M. Pletcher : An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. pages 233–242, Sept 2011.

[27] Jinhan Kim, Sanghoon Lee, Seung won Hwang et Sunghun Kim : Towards an intelligent code search engine. *Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, 2010.

[28] Thomas Knight et Jon Timmis : A multi-layered immune inspired machine learning algorithm. *In* Ahamad Lotfi et Jonathan M. Garibaldi, éditeurs : *Applications and Science in Soft Computing*, pages 195–202, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[29] Maxime Lamothe et Weiyi Shang : Exploring the use of automated api migrating techniques in practice : An experience report on android. *MSR 18*, 2018.

[30] Zhenmin Li et Yuanyuan Zhou : Pr-miner : Automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Softw. Eng. Notes*, 30(5):306–315, septembre 2005.

[31] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto et Denys Poshyvanyk : Api change and fault proneness : A threat to the success of android apps. *In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, New York, NY, USA, 2013. ACM.

[32] S. Liu, G. Bai, J. Sun et J. S. Dong : Towards using concurrent java api correctly. *In 2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 219–222, Nov 2016.

[33] Martin Monperrus, Marcel Bruch et Mira Mezini : Detecting missing method calls in object-oriented software. *In Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP '10, pages 2–25. Springer-Verlag GmbH, 2010.

[34] Martin Monperrus et Mira Mezini : Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–25, 2013.

[35] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan et M. Gethers : Export : Detecting and visualizing api usages in large source code repositories. *In 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 646–651, Nov 2013.

[36] Sarah Nadi, Stefan Krüger, Mira Mezini et Eric Bodden : Jumping through hoops : Why do java developers struggle with cryptography apis ? *In Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946, New York, NY, USA, 2016. ACM.

[37] Mathieu Nayrolles et Abdelwahab Hamou-Lhadj : Clever : Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. *MSR 2018*, 2018.

[38] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi et T. N. Nguyen : Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, Sept 2012.

[39] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi et Tien N. Nguyen : Accurate and efficient structural characteristic feature extraction for clone detection. pages 440–455, 2009.

[40] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim et Tien N. Nguyen : A graph-based approach to api usage adaptation. *SIGPLAN Not.*, 45(10):302–321, octobre 2010.

[41] T. T. Nguyen, H. V. Pham, P. M. Vu et T. T. Nguyen : Recommending API usages for mobile apps with Hidden Markov Model. *In Proceedings of the 30th ACM/IEEE International Conference on Automated Software Engineering*, ASE '15, pages 795–800. IEEE Computer Society Press, 2015.

[42] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi et Tien N. Nguyen : Graph-based mining of multiple object usage patterns. *In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[43] Hitesh Padekar, Younghee Park, Hongxin Hu et Sang-Yoon Chang : Enabling dynamic access control for controller applications in software-defined networks. *In Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, SACMAT '16, pages 51–61, New York, NY, USA, 2016. ACM.

[44] Jihyeok Park : Javascript api misuse detection by using typescript. *In Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 11–12, New York, NY, USA, 2014. ACM.

[45] Charles Petzold : Operating systems 1990 : Which way will we go ? *PC Magazine*, 10(1):182, 1991.

[46] Murali Krishna Ramanathan, Ananth Grama et Suresh Jagannathan : Path-sensitive inference of function precedence protocols. *In Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 240–250. IEEE Computer Society Press, 2007.

[47] M. P. Robillard : What makes apis hard to learn ? answers from developers. *IEEE Software*, 26(6):27–34, Nov 2009.

[48] M. A. Saied, O. Benomar, H. Abdeen et H. Sahraoui : Mining multi-level api usage patterns. pages 23–32, March 2015.

[49] Carla Sauvanaud, Mohamed Kaâniche, Karama Kanoun, Kahina Lazri et Guthemberg Da Silva Silvestre : Anomaly detection and diagnosis for cloud services : Practical experiments and lessons learned. *Journal of Systems and Software*, 139:84 – 106, 2018.

[50] Anand Sawant, Maurício Aniche, Arie van Deursen et Alberto Bacchelli : Understanding developers' needs on deprecation as a language feature. *ICSE 18*, 2018.

[51] Sriram Sankaranarayanan Sergio Move and, Rhys Braginton Pettee Olsen et Bor-Yuh Evan Chang : Mining framework usage graphs from app corpora. *SANER 2018*, 2018.

[52] S. Shuai, D. Guowei, G. Tao, Y. Tianchang et S. Chenjie : Modelling analysis and auto-detection of cryptographic misuse in android applications. *In 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, Aug 2014.

[53] R. Stevens, J. Ganz, V. Filkov, P. Devanbu et H. Chen : Asking for (and about) permissions used by android apps. *In 2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 31–40, May 2013.

[54] Siddharth Subramanian, Laura Inozemtseva et Reid Holmes : Live api documentation. *In Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM.

[55] Suresh Thummalapenta et Tao Xie : Alattin : Mining alternative patterns for detecting neglected conditions. *In Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 283–294. IEEE Computer Society Press, 2009.

[56] Suresh Thummalapenta et Tao Xie : Mining exception-handling rules as sequence association rules. *In Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 496–506. IEEE Computer Society Press, 2009.

[57] Christoph Treude et Martin P. Robillard : Augmenting api documentation with insights from stack overflow. *In Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 392–403, New York, NY, USA, 2016. ACM.

[58] Lijie Wang, Lu Fang, Leye Wang, Ge Li, Bing Xie et Fuqing Yang : Apiexample : An effective web search based usage example recommendation system for java apis. pages 592–595, Nov 2011.

[59] W. Wang et M. W. Godfrey : Detecting api usage obstacles : A study of ios and android developer questions. *In 2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 61–64, May 2013.

[60] Wei Wang, Haroon Malik et Michael W. Godfrey : Recommending posts concerning api issues in developer q&a sites. *10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 61–64, 2015.

[61] Andrzej Wasylkowski et Andreas Zeller : Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[62] Andrzej Wasylkowski, Andreas Zeller et Christian Lindig : Detecting object usage anomalies. *In Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[63] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan et Miryung Kim : Are code example on online q&a forum reliable ? a study of api misuse on stack overflow. *In ICSE'18 : The 40th International Conference on Software Engineering*, May 27-June 3, 2018 2018.

[64] Zhen Zhang : xwidl : Modular and deep javascript api misuses checking based on extended webidl. *In Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications : Software for Humanity*, SPLASH Companion 2016, pages 63–64, New York, NY, USA, 2016. ACM.

[65] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei et Hong Mei : Mapo : Mining and recommending api usage patterns. *In Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 318–343, Berlin, Heidelberg, 2009. Springer-Verlag.