Identification de vulnérabilités dans les applications Java

Raounak Benabidallah, Salah Sadou, and Isabelle Borne

Université de Bretagne Sud-IRISA, France
prénom.nom@irisa.fr

1 Introduction

Les cyberattaques, fléau mondial, continuent d'augmenter à la fois en termes de fréquence, de gravité et d'impact. Face à ce fléau, les entreprises se sentent de plus en plus vulnérables. En effet, les attaques de sécurité peuvent entraîner de graves dommages pour les organisations, pouvant aller de l'atteinte à leur image et leur réputation jusqu'à la paralysie de leur système d'information. Ainsi, il devient primordial pour ces organisations de se protéger et de sécuriser leur système d'information afin qu'elles puissent limiter les impacts liés aux cyberattaques. Il existe deux approches, complémentaires et nécessaires, qui permettent la sécurisation des systèmes d'information : La cyberdéfense et l'informatique de confiance. La première consiste à surveiller et à protéger le système lors de son fonctionnement, alors que la seconde vise à construire des systèmes les moins vulnérables possible. Le travail présenté dans cet article s'inscrit dans le domaine de l'informatique de confiance.

Par définition, une vulnérabilité est une faille dans le système, qui peut correspondre à un défaut de conception ou de mise en œuvre, exploitable par les attaquants pour nuire aux acteurs de l'application [10]. Malheureusement, la mise en évidence d'une vulnérabilité est généralement réalisée par les attaquants au stade opérationnel du système. Par conséquent, les défenseurs se retrouvent toujours en retard par rapport aux attaquants. Ainsi, les bonnes pratiques, de la conception à la mise en œuvre du système, deviennent une nécessité. De nos jours, il existe un certain nombre d'analyseurs permettant la détection des vulnérabilités les plus connues dans le code [7, 11]. Leur utilisation lors du cycle de développement est utile, mais pas suffisante. En effet, ils permettent de détecter seulement les vulnérabilités connues mais le système reste toujours sous la menace de vulnérabilités non encore découvertes. Ainsi, le problème à résoudre est : comment caractériser un code logiciel par rapport à son niveau de vulnérabilité.

Pour répondre à ce problème, plusieurs travaux se sont intéressés à l'utilisation des métriques logicielles pour estimer la vulnérabilité d'un code logiciel. Certains proposent des métriques dédiées à la vulnérabilité [9, 16]. Nous pensons qu'il peut être utile d'explorer la possibilité de construire des modèles de prédiction, en exploitant les métriques existantes avant de chercher à en construire de nouvelles. Des travaux ont exploré cette voie [13, 5], néanmoins, les auteurs se sont limités soit à un nombre restreint de métriques, soit à quelques approches de construction de modèles de prédiction. Dans le travail que nous proposons ici, nous ne faisons aucune restriction vis-à-vis des métriques et approches à utiliser. En effet, notre objectif est de pouvoir identifier les approches les plus pertinentes par l'expérimentation. Pour cela, nous disposons d'un jeu de données contenant plus de 20K applications Java.

Dans ce qui suit, nous présentons l'approche générale de notre processus de détection de vulnérabilités tout en détaillant le processus expérimental. Dans la section 3, nous discutons les premiers résultats d'expérimentation et les limites de notre approche. Avant de conclure en section 5, nous présentons quelques travaux connexes en section 4.

2 Processus de détection de vulnérabilités

Cette étude présente une évaluation empirique des méthodes d'apprentissage pour la détection de vulnérabilités. Le but n'est pas seulement d'identifier la méthode la plus adéquate mais aussi

d'analyser les faux positifs dans l'espoir de détecter des vulnérabilités non encore connues. En effet, notre objectif est de construire des modèles caractérisant un code vulnérable sans se limiter aux seules vulnérabilités déjà identifiées.

La figure 1 illustre le processus que nous suivons pour atteindre nos objectifs. Les étapes de ce processus sont détaillées ci-dessous :

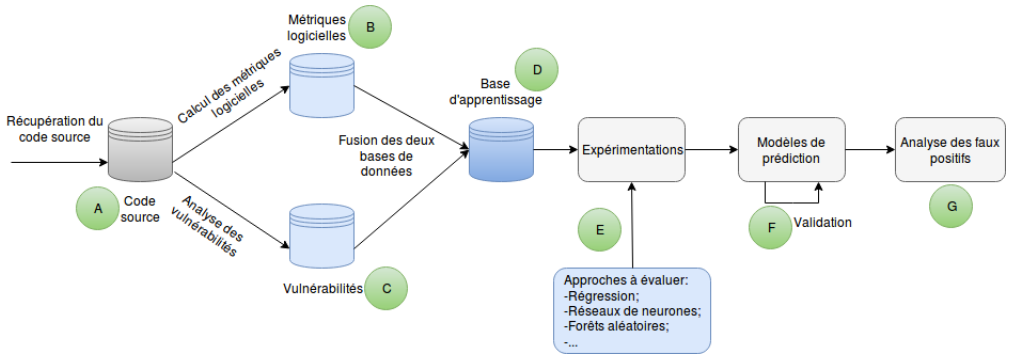

Figure 1: Processus expérimental

1. **Choix et sources de données (A) :** Afin de mener à bien nos expérimentations, nous avons besoin d'un nombre conséquent d'applications Java liées à des domaines les plus variés que possible. À cet effet, nous avons choisi d'utiliser le corpus Git [1] construit pour être utilisé dans le domaine de l'exploitation des données de manière générale, mais aussi dans l'étude des pratiques de codage en Java. Ainsi, les applications que nous exploitons varient en termes de taille et domaines d'utilisation. De plus, leurs propriétaires ont des niveaux de programmation différents, ce qui limite le risque d'avoir les mêmes pratiques de codage. Cela nous permet d'obtenir des résultats variés ainsi qu'un nombre important de vulnérabilités.

2. **Métriques logicielles (B) :** Dans cet article, nous nous concentrons sur la granularité classe pour le nombre de métriques qui la caractérisent. Ces dernières sont calculées avec l'outil d'analyse statique Sonargraph [8] qui fournit un ensemble de métriques incluant les métriques de couplage, taille, complexité, cohésion, etc.

3. **Détection de vulnérabilités (C) :** La deuxième phase d'analyse du code source consiste à attribuer à chaque classe Java une étiquette (sortie) de vulnérabilité. Pour cela, nous avons jugé nécessaire d'utiliser plusieurs analyseurs afin de s'assurer de l'absence de vulnérabilités. En effet, les outils existants tentent de maximiser le nombre de vulnérabilités découvertes tout en minimisant celui des faux positifs (i.e les fichiers étiquetés vulnérables alors qu'ils ne le sont pas) [4]. De plus, l'ensemble de vulnérabilités couvert par l'analyse diffère d'un outil à un autre. Par conséquent, l'utilisation de plusieurs analyseurs nous permettrait de couvrir le maximum de vulnérabilités. Enfin, l'étiquetage que nous avons choisi consiste à calculer l'union des résultats de tous les analyseurs utilisés. Ceci revient à considérer la classe comme étant vulnérable si et seulement si au moins un des analyseurs la classifie comme telle.

4. **Base d'apprentissage (D) :** Après avoir effectué les analyses expliquées précédemment, nous procédons à la fusion des résultats obtenus. L'opération consiste à associer à chaque classe la liste des métriques qui la caractérisent ainsi que l'étiquette de vulnérabilité. Cette dernière est d'ordre binaire : 1 pour une classe vulnérable, 0 sinon. Enfin, les données subissent un dernier traitement avant d'être exploitables. Cette étape consiste en la normalisation

des données dans l'intervalle [0,1]. Ce traitement a été nécessaire à cause de l'incompatibilité des unités de mesure entre les différentes métriques logicielles. Ainsi, nous évitons de privilégier les métriques ayant les plus grands domaines de variation telles que le nombre de lignes de code.

5. **Méthodes d'apprentissage (E) :** Lors de la phase de sélection des approches d'apprentissage, nous avons effectué une étude comparative qui nous a permis de distinguer celles pouvant être appliquées à notre problème. Certaines d'entre elles reviennent dans plusieurs travaux antérieurs telles que la *régression logistique* dans [13, 14, 6], les *arbres de décision* [14, 6], les *forêts aléatoires* [6, 2], etc. Nous avons donc repris ces approches et enrichi l'ensemble par de nouvelles méthodes incluant les *classifieurs bayésiens naïfs*, les *machines à vecteurs de support*, les *réseaux de neurones* et enfin les *K plus proches voisins*.

6. **Évaluation (F) :** Afin de pouvoir comparer les différents modèles de prédiction, il est important de définir des métriques permettant d'évaluer leurs performances. Dans la classification binaire, nous trouvons généralement les mesures d'exactitude, précision, rappel ainsi que la F-mesure.

7. **Analyse des faux positifs (F) :** La dernière étape de notre approche correspond à l'analyse des résultats obtenus avec les modèles de prédiction, et plus particulièrement les faux positifs. En effet, les modèles de prédiction que nous construisons prennent en entrée l'union des résultats des analyseurs. Les modèles vont donc tenter d'imiter le comportement des analyseurs, ce qui donnerait, dans le meilleur des cas, des performances légèrement supérieures à celles des analyseurs. Afin d'apporter des améliorations plus considérables, nous intégrerons une phase d'analyse des faux positifs dans l'espoir d'identifier des vulnérabilités non encore découvertes par les analyseurs. Ce traitement se fera manuellement, en collaboration avec des partenaires experts dans le domaine de la sécurité.

3 Discussion

Étant donné que l'analyse du code source est coûteuse en temps de calcul, nous avons commencé nos expérimentations sur une partie du corpus, en attendant la fin du traitement sur le reste des applications. Ainsi, nous avons appliqué le processus détaillé précédemment sur un benchmark contenant 30873 classes Java, caractérisées par 15 métriques logicielles. Les classes de vulnérabilités ont été déterminées avec l'outil d'analyse statique SonarQube qui a réussi à détecter des vulnérabilités dans 2361 instances, soit 7 % du nombre total des instances. Nous avons par la suite lancé des expérimentations sur cette base en utilisant quelques méthodes d'apprentissage. Les premiers résultats obtenus avec les réseaux de neurones montrent l'efficacité de cette approche avec des performances supérieures à 92 % d'exactitude. Néanmoins, l'approche proposée présente certaines limites que nous discutons dans ce qui suit.

Tout d'abord, nous proposons d'utiliser plusieurs analyseurs de vulnérabilités via une analyse statique plutôt que d'exploiter des vulnérabilités sauvegardées dans des bases de données. Ceci est justifié par l'insuffisance du nombre de vulnérabilités déjà détectées et répertoriées, ce qui pose un réel problème lors de la phase d'apprentissage. En contrepartie, les outils d'analyse ont tendance à produire beaucoup de faux positifs, ce qui peut induire en erreur le modèle de prédiction. À cet effet, il est important de passer par une phase manuelle pour la validation des résultats des analyseurs.

Par ailleurs, la classification que nous utilisons dans nos expérimentations est effectuée avec un seul outil d'analyse statique : SonarQube. Ce dernier est connu pour ses bonnes performances dans les métriques logicielles mais il s'avère moins performant dans la détection

de vulnérabilités. En effet, l'ensemble de vulnérabilités qu'il couvre est relativement petit comparé à d'autres outils commerciaux tels que Fortify. Pour remédier à ce problème, nous prévoyons d'intégrer d'autres outils qui raffineraient davantage les résultats d'étiquetage.

Enfin, il est à noter que le choix de nos applications s'est fait de manière aléatoire afin d'éviter de biaiser les résultats. Néanmoins, les résultats obtenus pourraient être spécifiques aux applications utilisées ou encore au langage Java de manière générale. Il serait donc intéressant d'entamer de nouvelles études basées sur d'autres ensembles d'applications (libres et/ou commerciales) afin de pouvoir généraliser l'approche proposée.

4 Travaux connexes

L'approche que nous proposons a comme objectif à terme de pouvoir définir les bonnes pratiques pour le développement d'un logiciel sécurisé. De ce fait, il est intéressant d'utiliser des modèles de prédiction capables d'apprendre puis généraliser ces caractéristiques. Les informations dont nous avons besoin dans ce cas, sont liées au code source en soi. Ainsi, dans cette étude de travaux connexes, nous ne discutons que les approches liées au code source.

Dans [13] et [14], Shin et al. se sont intéressés à l'utilisation des métriques de complexité pour l'identification de code vulnérable. Ils avaient comme hypothèse qu'un code plus complexe contiendrait plus de vulnérabilités découvertes. Pour confirmer cette hypothèse, ils ont fait une étude dont les résultats montrent une faible corrélation entre les métriques de complexité et le nombre de vulnérabilités découvertes. Les auteurs ont poursuivi leurs recherches dans [15] et [12] où ils analysent l'efficacité d'autres métriques de complexité telles que la complexité dynamique et la complexité du "code Churn". Les résultats des expérimentations montrent cette fois une bonne corrélation entre les différentes métriques et l'existence de vulnérabilités.

Par ailleurs, Chowdhury et al. [5] ont exploré la relation entre les vulnérabilités et les métriques CCC, à savoir la complexité, le couplage et la cohésion. Ils ont opté pour une étude statistique qui a montré qu'il existe une forte corrélation entre les métriques CCC et l'existence de vulnérabilités. En se basant sur ces résultats, les auteurs ont proposé une plate-forme dans [6] pour l'automatisation de la prédiction en utilisant des méthodes d'apprentissage. Les résultats enregistrés montrent l'efficacité de l'approche avec des performances supérieures à 70% pour certains modèles de prédiction.

Dans [3], les auteurs ont étudié la corrélation entre les métriques logicielles et l'existence de vulnérabilités. Pour ce faire, ils construisent une base de données importante contenant les informations relatives à 5 grands projets d'un point de vue sécuritaires. Les résultats de cette étude montrent une fois encore que la majorité des métriques analysées peuvent être utilisées pour distinguer les codes vulnérables. Dans une seconde étude [2], Alves et al. utilisent cette base comme jeu de données pour leurs expérimentations. Ces dernières consistent à reprendre les méthodes d'apprentissage déjà utilisées pour identifier les fichiers vulnérables, et ce dans le but de comparer leurs performances. Les auteurs ont ainsi montré l'efficacité des méthodes d'apprentissage en général et les forêts aléatoires en particulier.

5 Conclusion

Dans cet article, nous avons présenté une approche pour la détection de vulnérabilités en se basant sur les métriques logicielles. Contrairement aux autres travaux existants, nous avons pris en considération toutes les approches de construction de modèles de prédiction pouvant être appliquées à notre problème, ainsi que toutes les métriques logicielles pour une bonne caractérisation du code source. Bien que les expérimentations ne soient qu'à leur début, nous avons réussi à obtenir de premiers résultats prometteurs avec les réseaux de neurones. Par

conséquent, nous prévoyons de poursuivre nos expérimentations en enrichissant le jeu de données avec le reste des applications Java dont nous disposons.

References

[1] Miltiadis Allamanis and Charles Sutton. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE, 2013.

[2] H. Alves, B. Fonseca, and N. Antunes. Experimenting machine learning techniques to predict vulnerabilities. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 151–156, Oct 2016.

[3] H. Alves, B. Fonseca, and N. Antunes. Software metrics and security vulnerabilities: Dataset and exploratory study. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 37–44, Sept 2016.

[4] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106, Sept 2011.

[5] Istehad Chowdhury and Mohammad Zulkernine. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1963–1969, New York, NY, USA, 2010. ACM.

[6] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294 – 313, 2011. Special Issue on Security and Dependability Assurance of Software Architectures.

[7] Hewlett Packard Entreprise. Fortify Static Code Analyzer, 2017.

[8] Hello2morrow. Sonargraph Product Family, 2017.

[9] Leanid Krautsevich, Fabio Martinelli, and Artsiom Yautsiukhin. Formal approach to security metrics.: What does "more secure" mean for you? In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 162–169, New York, NY, USA, 2010. ACM.

[10] Owasp (Open Web Application Security Project). Category:Vulnerability, 2016.

[11] Yagaan Software Security. Yag Scanner, 2016.

[12] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, Nov 2011.

[13] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 315–317, New York, NY, USA, 2008. ACM.

[14] Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, pages 47–50, New York, NY, USA, 2008. ACM.

[15] Yonghee Shin and Laurie Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, SESS '11, pages 1–7, New York, NY, USA, 2011. ACM.

[16] Ju An Wang, Hao Wang, Minzhe Guo, and Min Xia. Security metrics for software systems. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 47:1–47:6, New York, NY, USA, 2009. ACM.