RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Herodotos, l'historien de vos défauts

Nicolas Palix

DIKU, Université de Copenhague, Danemark
npalix@diku.dk

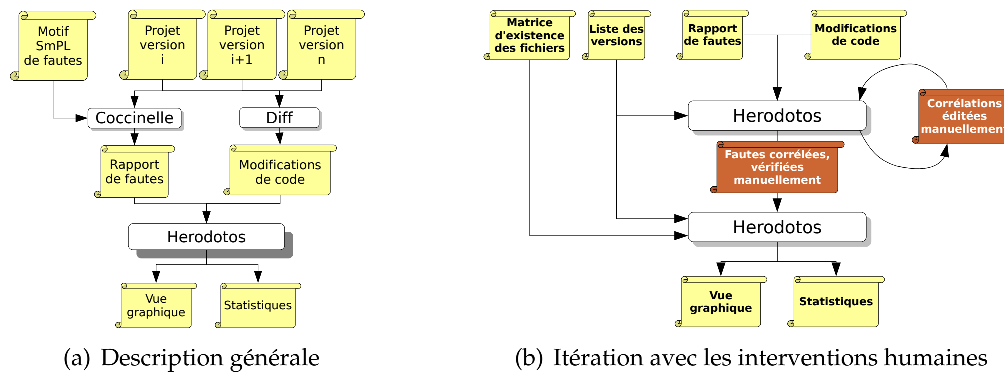
Résumé
Les logiciels évoluent continuellement afin d'améliorer les performances, corriger les erreurs, ou ajouter des fonctionnalités. Cependant, les modifications du code conduisent inévitablement à l'introduction de défauts logiciels. Pour prévenir l'introduction de fautes, il est nécessaire de comprendre pourquoi elles surviennent. Il est donc important de développer des outils et des pratiques aidant à trouver, suivre et prévenir ces défauts.
Dans cet article, nous proposons une méthodologie et son outil associé, Herodotos, permettant d'étudier les fautes dans le temps et d'ainsi déterminer le moment de leur apparition et de leur disparition. Herodotos suit, de manière semi-automatisée, les fautes sur plusieurs versions d'un même projet indépendamment des modifications annexes dans les fichiers sources. Il construit des historiques graphiques et calcule des statistiques sur la vie des fautes. Nous avons évalué cette approche sur l'historique de quatre projets open source sur les trois dernières années. Pour chaque projet, nous explorons plusieurs types de fautes qui ont été trouvées par une analyse statique du code. Nous analysons les résultats produits pour comparer les projets sélectionnés et les types de fautes étudiées.

Mots-clés : Historique de défauts logiciels, Qualité logicielle, Évolution logicielle, Analyse statique

---

1. Introduction

Les logiciels s'améliorent constamment, aussi bien en terme de fonctionnalités que de corrections des erreurs. Par exemple, durant les trois dernières années, le noyau Linux a augmenté de 45%, et ses pilotes de périphériques de plus de 50%. Des modifications du code de cette ampleur conduisent inévitablement à l'introduction de fautes logicielles. Pour prévenir leur introduction, il est nécessaire de comprendre pourquoi elles surviennent. Il est donc important de développer des outils et des pratiques aidant à trouver, suivre et prévenir ces fautes.
Une approche, qui peut aider à mieux comprendre et prévenir les fautes logicielles, consiste à les étudier dans le temps au sein d'un projet et de notamment déterminer les instants où un défaut est introduit et supprimé. Une telle étude permet de répondre à des questions telles que : le projet s'améliore-t-il ? Combien de temps une faute reste-t-elle dans le projet ? Est-ce que la durée de vie d'une faute dépend du type de faute ? De l'aspect critique du logiciel ? Les fautes d'un projet logiciel sont souvent référencées dans un gestionnaire d'erreurs *(bug tracker)*. Cependant, les rapports d'erreur dans un tel système sont souvent écrits en langage naturel, sans lien explicite au code. Il est donc très difficile de relier un rapport d'erreur à une ligne de code ou à un ensemble de versions.



(a) Description générale (b) Itération avec les interventions humaines

FIG. 1 – Description du processus

Pour obtenir un historique précis des fautes, il est nécessaire de travailler au niveau du code. Cependant, un défi lors de l'étude temporelle des fautes est que le code évolue autour de chaque faute. Par exemple, durant les trois dernières années, il y a eu une nouvelle version du noyau Linux tous les trois mois, et plus de 6% du noyau a été changé entre chaque version. Cela implique que certaines fautes sont ajoutées ou supprimées tandis que d'autres changent d'emplacement. Afin d'étudier les fautes dans le temps, il faut tout d'abord savoir où elles se trouvent, et donc avoir un ensemble cohérent et conséquent de fautes sur plusieurs versions d'un logiciel. Une fois collecté l'ensemble des fautes, chacune doit être corrélée d'une version à la suivante malgré les modifications du code.

Dans cet article, nous proposons une méthodologie et son outil associé, Herodotos [16], afin d'étudier l'historique des fautes. La méthodologie repose sur l'outil Coccinelle [3, 15] pour trouver les fautes, et `diff` [12] pour inférer les modifications de code. À partir de ces informations, Herodotos fait la corrélation des fautes détectées entre les versions, découvrant ainsi l'historique de chaque faute. Il calcule également des statistiques sur ces historiques et permet à l'utilisateur de spécifier les faux positifs afin d'obtenir des résultats précis. Le processus est illustré Figure 1(a).

Les contributions de cet article sont les suivantes :

– Nous proposons une méthodologie pour étudier et corréler dans le temps des fautes logicielles.
– Nous fournissons un outil, Herodotos, qui construit un historique des fautes en générant des représentations graphiques et calcule des statistiques sur les fautes. Pour cela, il utilise les rapports de fautes et la description des changements dans le code. Ces informations sont respectivement générées par un analyseur statique de code, Coccinelle dans notre cas, et par `diff`.
– Nous évaluons la méthodologie proposée et Herodotos sur un ensemble représentatif d'infrastructures logicielles. De fait, nous démontrons l'aspect pratique et utilisable de notre méthodologie et de son outil. Les logiciels sélectionnés s'étendent du système d'exploitation Linux au composant multimédia VLC, en passant par la librairie de sécurité OpenSSL.

La suite de cet article est organisée ainsi. Dans les Section 2 et 3, nous présentons pas à pas notre méthodologie pour corréler les fautes, puis l'outil Herodotos qui la met en œuvre. Dans la Section 4, nous décrivons nos expériences sur quatre projets logiciels et évaluons notre méthodologie à partir de ces expériences. Finalement, la Section 5 présente des travaux connexes et la Section 6 conclut.

2. Méthodologie

Notre méthodologie pour obtenir l'historique des fautes d'un projet est fondée sur les étapes suivantes. Tout d'abord, l'utilisateur sélectionne les catégories de fautes qui l'intéressent. Ensuite, il

génère les rapports de fautes pour ces catégories à l'aide d'un analyseur statique de code. Pour les fichiers contenant des fautes, l'outil `diff` permet d'identifier les changements qui ont eu lieu entre chaque version et sa suivante. Enfin, l'utilisateur exécute Herodotos pour construire l'historique des fautes à partir des informations précédemment calculées.

2.1. Classification des fautes

Afin d'obtenir une vue représentative de comment et pourquoi les fautes ont été introduites dans un projet, il est nécessaire d'avoir une vue représentative des fautes elles-mêmes. Les fautes ont déjà été étudiées et classifiées plusieurs fois [4, 10, 13]. Une référence largement utilisée est la *Common Weakness Enumeration (CWE)* [13] qui définit des classes pour les défauts typiques, *p.ex.* la gestion de ressources et la structure du code. Nous étudions certaines fautes de ces classes dans notre évaluation présentée à la Section 4.

2.2. Recherche des fautes

Avant de construire un historique des fautes, il est nécessaire de les trouver dans une multitude de versions d'un projet. Afin d'avoir une base cohérente de fautes, nous avons choisi de reposer sur un outil pour trouver les fautes plutôt qu'un gestionnaire d'erreurs qui est souvent rempli manuellement, et où la quantité et la qualité des rapports dépendent des habitudes des testeurs et des utilisateurs. Concrètement, pour réaliser l'analyse statique nous avons sélectionné l'outil Coccinelle pour sa disponibilité et sa flexibilité [9, 15]. Coccinelle permet de spécifier des règles pour la recherche de fautes en utilisant une notation proche du code C. De plus, ces règles peuvent comporter des éléments en Python, permettant de générer des rapports de fautes personnalisés.

Pour obtenir une vue d'ensemble des fautes d'un logiciel, il est nécessaire d'avoir des informations sur plusieurs classes de fautes. Pour chaque classe, l'utilisateur crée au moins une règle de recherche qui est ensuite appliquée à chaque version étudiée du projet avec Coccinelle pour produire un rapport de fautes. Notre approche s'appuie sur le fait que ce rapport est formaté pour le mode Emacs Org [14] qui permet de définir, pour chaque faute, un lien vers le code source. Chaque lien contient des informations relatives à une faute telles que le nom du fichier, la ligne et la colonne, et éventuellement une information définie par l'utilisateur. La navigation dans le rapport et la consultation des liens permettent de rapidement vérifier les fautes signalées.

2.3. Calcul des modifications d'une version à la suivante

Quand une faute n'est pas corrigée par un développeur, elle apparaît, et sera donc signalée, dans de multiples versions. Il est peu vraisemblable que les fautes restent aux mêmes emplacements, avec le même contexte, car des changements concernant d'autres préoccupations peuvent avoir lieu à n'importe quel endroit dans le fichier. Pour corréler les fautes sur plusieurs versions, nous avons donc besoin de connaître les changements qui ont eu lieu dans le même fichier. À partir des fautes signalées, la liste des fichiers affectés est déterminée. Pour chaque fichier, les changements d'une version à la suivante sont calculés par l'outil `diff`. Toutes les modifications sont finalement agrégées dans un unique fichier pour chaque projet afin de disposer d'un ensemble consolidé de modifications.

2.4. Herodotos

Pour chaque projet, nous avons à ce stade un rapport de fautes et un fichier décrivant des modifications. Grâce à ces informations, Herodotos corrèle les fautes sur plusieurs versions ordonnées en prenant en compte les modifications. Une fois que les fautes sont corrélées, Herodotos génère une représentation graphique de la durée de vie de chaque faute sur la période couverte par les versions étudiées. Des événements tels que la création ou la suppression d'un fichier, l'introduction ou la suppression d'une faute sont tous illustrés dans cette représentation. Finalement,

Herodotos calcule des statistiques sur les fautes dans le projet : (1) le nombre de fichiers affectés, (2) le nombre moyen de fautes qu'ils contiennent, (3) le nombre de fautes introduites par un nouveau fichier, (4) le nombre de fautes supprimées avec le fichier auquel elles appartiennent, (5) la durée de vie minimum, maximum, et moyenne d'une faute, et (6) le nombre de fautes par sous répertoire. D'autres statistiques peuvent être aisément ajoutées.

3. Herodotos en détail

La Figure 1(b) illustre en détail comment Herodotos utilise le rapport de fautes et les changements du code pour générer l'historique des fautes, et comment l'utilisateur peut interagir avec l'outil. Herodotos réalise trois grandes étapes : 1) définir un ensemble initial de corrélations des fautes, 2) vérifier et affiner ces corrélations grâce à celles manuellement définies par l'utilisateur, et 3) générer des graphiques et des statistiques qui décrivent l'historique des fautes. L'utilisateur peut intervenir dans ce processus de deux manières : i) après l'étape 1, en fournissant des corrélations particulières qui n'auraient pas été détectées automatiquement, et ii) après l'étape 2, en indiquant les faux positifs générés par l'analyseur statique. Le reste de cette section explique ce processus.

3.1. Corrélation des fautes

Herodotos doit corréler les fautes identiques entre les versions, y compris lorsque des changements dans le code environnant ont lieu. Pour résoudre ce problème, Herodotos repose sur l'outil `diff` pour inférer les modifications. La description des modifications lui permet de transposer la position d'une faute dans une version à la position supposée dans la version suivante. Pour une paire de fichiers donnés, `diff` produit une suite de blocs, nommés *hunks*. Chaque bloc décrit une séquence de lignes contiguës qui doivent être supprimées ou ajoutées afin de transformer le fichier d'origine et de produire le fichier de la version suivante.[1] De plus, `diff` annote chaque bloc avec sa position. Une position est définie par le numéro de ligne où commence le bloc dans le fichier d'origine, le nombre de lignes supprimées, le numéro de ligne dans le fichier produit, et le nombre de lignes ajoutées.

Pour corréler les fautes, Herodotos recherche le dernier bloc dont la ligne de début commence avant ou à la même ligne qu'une faute. Si la faute se situe dans les lignes supprimées, Herodotos considère que la faute est supprimée. Cependant, si la faute se situe après la fin du bloc, alors le reste du code est le même jusqu'au prochain bloc. Herodotos calcule alors une prédiction de la ligne comportant la faute dans la version suivante. Il ajoute pour cela la différence entre le nombre de lignes ajoutées par le bloc et le nombre de lignes qu'il supprime. Herodotos recherche alors une faute dans le rapport de la version suivante mettant en jeux le même fichier et la prédiction. Si une faute est trouvée, les deux fautes sont considérées comme corrélées. S'il n'y a pas de faute à l'emplacement de la prédiction, cela signifie généralement que la faute a été corrigée par d'autres modifications dans le fichier.

Chaque projet logiciel a sa propre convention pour nommer des versions, et celle-ci peut éventuellement changer. L'utilisateur doit donc fournir à Herodotos une liste ordonnée des noms de version. Cette liste est utilisée lors de la phase de corrélation pour définir la version suivante.

3.2. Proposition d'informations correctives

Des modifications affectant une ligne comportant une faute peuvent conduire à une mauvaise corrélation car cette ligne est considérée comme supprimée par un bloc, alors même que la faute reste présente. Dans ce cas, Herodotos infère que la faute est supprimée dans une version et qu'une autre est introduite dans la version suivante.

[1] Nous utilisons l'option `-U0` de `diff` afin d'omettre le contexte du code.

Lorsque cela se produit, l'utilisateur peut compléter l'historique d'une faute. Pour faciliter ce travail, Herodotos génère une liste de corrélations possibles constituée de toutes les paires de fautes non corrélées d'un même fichier entre deux versions consécutives. Afin de limiter les propositions, seules les fautes de longueur identique sont proposées. Cette liste est au format du mode Org de Emacs avec des hyperliens vers le code source. Chaque paire possible de fautes est annotée avec un état, `TODO`, `SAME`, ou `UNRELATED`, indiquant respectivement une paire à vérifier, une paire de fautes à associer, ou une paire de fautes sans relation. Dans la liste initiale générée par Herodotos, toutes les paires ont l'état `TODO`. Pour chaque paire `TODO`, l'utilisateur peut suivre les hyperliens et vérifier les fautes signalées dans les deux versions. Si les deux fautes de la paire sont identiques mais à des positions différentes, l'utilisateur change l'état de la paire de `TODO` à `SAME`. Si les fautes sont sans relation, l'utilisateur marque la paire avec l'état `UNRELATED`.

Comme Herodotos propose une corrélation pour chaque paire possible de fautes dans un fichier donné, le nombre de propositions peut être quadratique. Cependant, chaque faute ne peut être en relation qu'avec une seule autre. Ainsi, lorsque l'utilisateur identifie certaines corrélations proposées comme valides, Herodotos élimine automatiquement les propositions obsolètes. Cela conduit à un processus itératif dans lequel Herodotos prend en entrée non seulement les fautes et les changements de code, mais aussi un ensemble partiel de corrélations. Il produit alors un nouvel ensemble de corrélations que l'utilisateur peut vérifier. Un point fixe est atteint lorsque l'utilisateur a marqué toutes les corrélations comme étant soit dans l'état `SAME`, soit dans l'état `UNRELATED`.

3.3. Vérification du rapport de fautes

En appliquant de l'analyse statique à des projets logiciels exploitant toutes les fonctionnalités du langage C, des faux positifs sont inévitables à cause des limites de l'analyse. L'intervention humaine est alors requise pour vérifier les fautes signalées. Afin de ne pas vérifier chaque faute dans toutes les versions dans laquelle elle apparaît, les faux positifs ne sont pas recherchés en amont de l'utilisation d'Herodotos mais entre la phase de corrélation et la phase finale de génération de l'historique des fautes. Dans ce cas, l'utilisateur n'a plus qu'à vérifier une unique instance dans la vie de chaque faute.

Chaque historique d'une faute est représenté par un entête décrivant la faute suivi de l'ensemble de ses positions dans différentes versions. Chaque position est représentée par un hyperlien et un état est associé à l'entête. L'état d'une faute est soit `TODO`, `BUG`, `FP`, représentant respectivement une faute à vérifier, une faute réelle, ou un faux positif. Toutes les entrées ont initialement l'état `TODO`. Pour chaque entrée, l'utilisateur peut suivre l'hyperlien pour étudier le code environnant, et ainsi changer l'état en `BUG` ou `FP` selon ce qui est approprié. Seules les fautes marquées `BUG` sont considérées lors du rendu graphique et du calcul des statistiques.

3.4. Construction de représentations graphiques de l'historique des fautes

Une fois les fautes réelles identifiées, Herodotos génère les représentations graphiques et calcule les statistiques. Pour également étudier les causes possibles qui affectent les fautes, l'existence des fichiers est intégrée. Il est en effet communément admis que l'introduction de nouvelles fonctionnalités introduit également des fautes. Afin de quantifier cet impact, Herodotos utilise une matrice indiquant pour chaque version l'existence des fichiers contenant une faute. Nous construisons automatiquement cette matrice à partir des fautes vérifiées et des dépôts de code.

Pour la dernière étape, l'utilisateur fournit à Herodotos le rapport de fautes vérifiées et corrélées, la liste ordonnée des noms de version, et la matrice indiquant l'existence des fichiers affectés. L'existence d'un fichier est illustrée dans la représentation graphique et est utilisée pour calculer le nombre de fautes introduites ou supprimées au même moment que le fichier auquel elles ap-

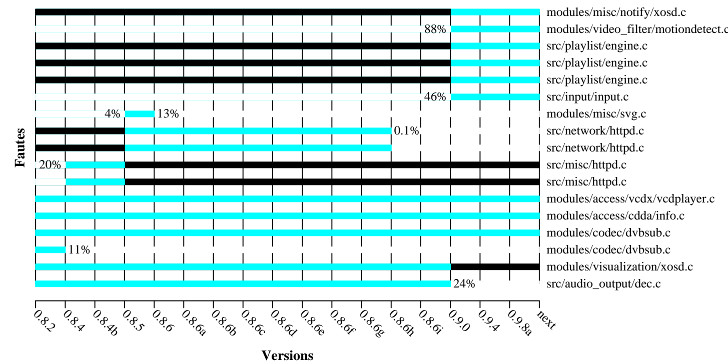


FIG. 2 – Graphique généré pour les fautes de type déréférence de NULL dans VLC

partiennent.

Herodotos crée un graphique à partir d'une liste de fautes. Comme le présente la Figure 2, pour chaque faute, une barre cyan/grise commence à la version où la faute est introduite, et se termine à la version où la faute disparaît. Le taux de renouvellement du fichier est alors indiqué en pourcentage. Une barre noire indique les versions où le fichier n'existe pas, soit parce qu'il n'a pas encore été ajouté, soit parce qu'il a été supprimé. Par exemple, dans la Figure 2, la première faute est introduite en même temps que son fichier, tandis que la seconde faute est introduite par des modifications sur du code existant. Dans ce cas, le fichier avait été amplement réécrit, plus de 80%. Pour cet exemple, les versions qui introduisent une faute ont, en général, également fait l'objet d'importants changements par rapport à la version précédente. À la ligne sept, la faute a été introduite lors de l'ajout d'une fonctionnalité dans du code existant de la version 0.8.5. Cette faute a été corrigée dans la version 0.8.6. Finalement, l'avant dernière faute disparaît en même tant que son fichier est supprimé.

4. Évaluation

Nous avons appliqué notre méthodologie à quatre projets logiciels open source, sélectionnés pour leur variété. Dans notre évaluation, nous considérons à la fois la facilité d'utilisation de Herodotos et l'information qu'il peut apporter à propos de l'historique des fautes dans les projets.

4.1. Logiciels sélectionnés

Pour notre évaluation, nous avons sélectionné des projets avec différents profils : Linux, Wine , VLC et OpenSSL. Ils couvrent des préoccupations fonctionnelles allant d'un système d'exploitation (SE) complet (Linux), à l'interface utilisateur d'un SE (Wine), jusqu'à un composant applicatif (VLC). OpenSSL a été sélectionné pour comparer ces préoccupations vis à vis de la sécurité.

Parmi ces projets, Linux a le développement le plus stable et mature, avec une version publiée environ tous les trois mois. Les versions v2.6.13 à v2.6.28 incluse nous servent de références. Pour les autres projets, nous avons sélectionné les versions publiées à peu près au même moment.[2]

Linux est le plus grand projet, avec 4 à 6 millions de lignes de code C dans la période considérée. Wine est le suivant, avec 1 à 1,5 millions, et OpenSSL et VLC sont les plus petits, avec 200 000 à 330 000 lignes de code. Si l'on considère l'augmentation du nombre de lignes dans chaque projet proportionnellement à la taille de la première version étudiée, Linux, Wine, et VLC ont augmenté d'environ 50%, tandis que OpenSSL a seulement augmenté d'environ 15%. Pour VLC, la taille du code est restée essentiellement la même durant une longue période, avant d'augmenter

[2] Deux versions ont été omises pour OpenSSL car les archives correspondantes sont corrompues.

brutalement mi-2008. L'augmentation des autres projets, est quant à elle beaucoup plus linéaire.

4.2. Types de fautes étudiées

Nous avons sélectionné les types de fautes suivants : 1) de la mémoire est allouée, mais n'est pas relâchée (malloc/kmalloc), 2) un pointeur NULL est déréférencé (isnull), 3) un pointeur est déréférencé avant d'être testé à NULL (null_ref), 4) un pointeur est testé à NULL alors qu'il est déjà connu pour ne pas l'être (notnull), 5) affectation d'une constante à une variable qui n'est jamais utilisée (unused), 6) test d'un pointeur avec la valeur 0 plutôt que NULL (badzero), 7) usage erroné d'une opération booléenne et d'une opération sur les bits (notand), et 8) test qu'une valeur non signée est inférieure à zéro (unsigned). Certaines fautes peuvent causer des défaillances ou des fuites mémoires (*p.ex.*, isnull et malloc). D'autres ne sont pas intrinsèquement la cause d'erreurs mais peuvent être le symptôme d'autres fautes. Par exemple, lors de nos expériences nous avons trouvé des endroits où le test redondant sur la valeur NULL (notnull) devait être réalisé sur une autre variable. D'autres types rendent simplement le code plus difficile à comprendre. Par exemple, comparer une expression de type pointeur avec zéro plutôt que NULL (badzero) suggère que l'expression renvoie un entier, ce qui peut induire en erreur le développeur.

4.3. Expérimentations

Une fois les résultats générés par Coccinelle, Herodotos réalise la corrélation des fautes. Nous avons ensuite identifié les faux positifs parmi les fautes signalées, et mis en œuvre Herodotos pour construire les représentations graphiques et calculer les statistiques correspondantes. Dans les sections suivantes, nous présentons une synthèse de ces statistiques depuis plusieurs points de vue : par projet, par type de fautes, et par évolution des fautes au sein des projets.

4.4. Résultats

Les Tableaux 1 et 2 et les Figures 3 et 4 résument les résultats de la recherche de fautes ainsi que les statistiques et les graphiques produits par Herodotos. Le Tableau 1 présente le nombre de fautes signalées dans les différentes catégories, et le nombre qui a été manuellement confirmé. Le Tableau 2 décrit le ratio de fautes confirmées par rapport aux emplacements possibles pour ce type de fautes. Par exemple, pour le premier type de fautes sur les ressources, nous comptons le nombre d'appels à (k)malloc. De manière similaire, pour la faute consistant à vérifier deux fois un pointeur avec la valeur NULL, nous comptons le nombre de tests pour NULL. Il n'y a pas d'entrée pour unused dans ce tableau car il n'est pas clair comment doit être défini un site potentiel qui soit approprié sans considérer l'ensemble du programme. La Figure 3 montre pour chaque projet et chaque type de fautes : le nombre d'occurrences, la durée de vie moyenne, et le nombre de fautes introduites et supprimées en même temps que le fichier auquel elles appartiennent. Finalement, la Figure 4 montre des graphiques générés illustrant l'évolution des fautes.

Par projet logiciel

Linux est le projet qui a le plus de fautes, et a au moins une faute de chaque type (Tableau 1). L'aspect critique du SE a également un impact sur la qualité. Linux a le plus faible taux maximum après OpenSSL. Mais OpenSSL est un projet environ 24 fois plus petit que Linux et dédié à la sécurité. Linux a cependant le taux maximum de fautes pour le tiers des types de fautes.

OpenSSL confirme sa position de logiciel de sécurité stable. En effet, il a le plus petit nombre de fautes confirmées (Tableau 1) et son plus haut taux de fautes est aussi un ordre de grandeur plus petit que celui des trois autres projets (Tableau 2). Les développeurs de ce projet sont également très conservateurs. La durée de vie des fautes en est rallongée comme l'illustre la Figure 3(b).

VLC ne se distingue pas des autres avec des valeurs extrêmes. Son nombre de fautes est relative-

ment bas mais trois fois plus élevé que celui de OpenSSL qui est de taille comparable. VLC a aussi un taux environ trois fois supérieurs à OpenSSL à la fois pour le taux moyen et le taux maximum. Finalement, Wine présente des similarités avec Linux. Dans plusieurs cas, ils ont des taux de fautes similaires et Wine a également le taux maximum dans le tiers des types de fautes. Enfin, Wine a aussi au moins une occurrence de chaque type, contrairement à OpenSSL et VLC.

Par catégorie de fautes

Bien que de nombreux outils d'analyses se focalisent sur les problèmes de gestion mémoire, la Figure 3(a) montre que de telles fautes sont encore nombreuses. C'est particulièrement le cas dans Linux où l'utilisation d'outils de l'espace utilisateur n'est pas possible. Pour Wine, ces fautes ont le plus haut taux des projets. Cependant, elles ont tendance à avoir une durée de vie plus courte (Figure 3(b)), comme pour Wine avec malloc ou dans les autres projets avec isnull.
Les fautes avec une longue durée de vie sont la déréférence de valeur NULL (isnull, dans Wine), les tests redondants de NULL (notnull, dans VLC), le test d'un pointeur avec zéro (badzero, dans Wine) et les fautes dans OpenSSL en général. Parmi ces fautes, seule la première conduit à un arrêt brutal du programme. L'aspect non critique des trois autres types de fautes peut expliquer leur plus longue durée de vie.
La comparaison d'un pointeur avec zéro est courante dans les nouveaux fichiers et a une longue durée de vie. Ce type de problème est souvent soit encore présent, soit disparaît avec le fichier. Finalement, le test de comparaison d'un entier non signé avec les valeurs inférieures à zéro est courant, dans le cas de Linux cela représente près d'1% des comparaisons avec zéro.

Types de fautes		Linux C. / S.	Wine C. / S.	VLC C. / S.	OpenSSL C. / S.
Ressources	malloc / kmalloc	42 / 44	2 / 2	2 / 2	0 / 0
	isnull	42 / 92	1 / 4	3 / 6	3 / 4
	null_ref	276 / 309	19 / 27	17 / 20	4 / 7
Code inutile	notnull	48 / 69	30 / 30	4 / 4	11 / 13
	unused	267 / 286	39 / 46	8 / 9	16 / 18
Code non sûr	badzero	851 / 862	248 / 255	115 / 115	7 / 7
Code erroné	notand	72 / 76	16 / 16	2 / 2	0 / 0
	find_unsigned	188 / 189	1 / 1	0 / 0	2 / 2
	Total	1 786 / 1 927	356 / 381	151 / 158	43 / 51

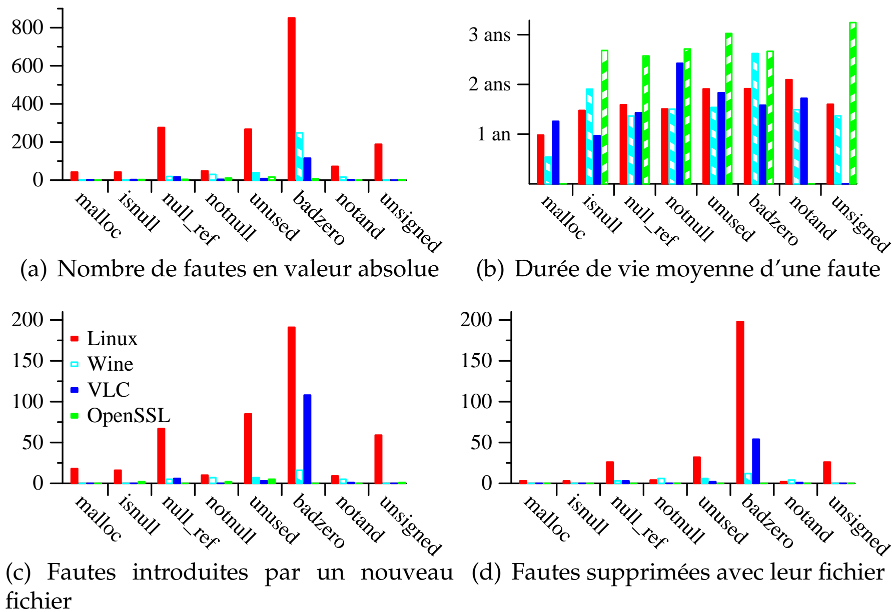
(Projets logiciels : Linux, Wine, VLC, OpenSSL)

TAB. 1 – Fautes par catégorie et par projet (C. pour confirmées, et S. pour signalées)

Types de fautes		Linux	Wine	VLC	OpenSSL	Min	Avg	Max
Ressources	malloc / kmalloc	0,53%	2,30%	0,16%	0,00%	0,00%	0,75%	2,30%
	isnull	0,01%	0,001%	0,01%	0,02%	0,001%	0,01%	0,02%
	null_ref	0,10%	0,01%	0,03%	0,02%	0,01%	0,04%	0,10%
Code inutile	notnull	0,05%	0,10%	0,05%	0,09%	0,05%	0,07%	0,10%
Code non sûr	badzero	0,82%	0,80%	1,49%	0,06%	0,06%	0,79%	1,49%
Code erroné	notand	0,08%	0,12%	0,10%	0,00%	0,00%	0,07%	0,12%
	find_unsigned	0,92%	0,09%	0,00%	0,42%	0,00%	0,36%	0,92%
	Minimum	0,02%	0,001%	0,00%	0,00%			
	Moyenne	0,36%	0,49%	0,26%	0,09%			
	Maximum	0,92%	2,30%	1,49%	0,42%			

(Projets logiciels : Linux, Wine, VLC, OpenSSL)

TAB. 2 – Ratio entre les fautes confirmées et les occurrences possibles de ce type de fautes



(a) Nombre de fautes en valeur absolue (b) Durée de vie moyenne d'une faute

(c) Fautes introduites par un nouveau fichier (d) Fautes supprimées avec leur fichier

FIG. 3 – Statistiques générées pour chaque type de fautes et pour chaque logiciel

Évolution des projets

Pour chaque logiciel et pour chaque type de fautes, une courbe de la Figure 4 indique l'évolution du nombre de fautes. Les courbes relatives aux fautes de type badzero ont été omises car leur nombre élevé rendait la comparaison difficile entre les autres types de fautes.

Le nombre de fautes signalées concernant la mauvaise utilisation des opérateurs bit/booléen (notand) et la comparaison de valeur non signé par rapport à zéro a chuté depuis respectivement les versions 2.6.24 et 2.6.25. L'utilisation de Coccinelle sur le noyau Linux depuis la version 2.6.24, pour trouver et corriger des fautes, peut en partie expliquer cette observation sur les dernières versions que nous avons étudiées. Nous n'observons pas de tendance similaire pour les autres types de fautes. Les fautes de type null_ref, notnull, et unused ont même tendance à augmenter.

Les fautes sur OpenSSL sont stables ce qui suggère que peu d'efforts a été déployé pour les corriger. Le nombre de variables non utilisées a même récemment augmenté. Cependant OpenSSL

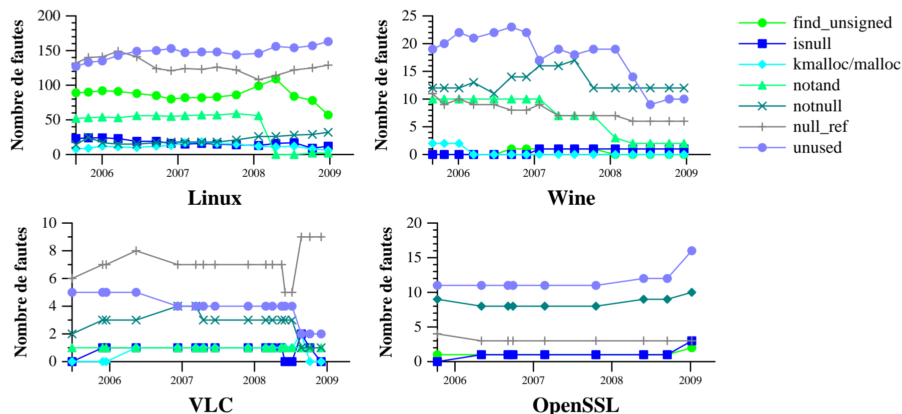


FIG. 4 – Graphiques générés montrant l'évolution par logiciel et type de fautes (excepté badzero)

est aussi le projet avec le plus faible taux d'augmentation du code, environ 15%, ce qui indique que son développement n'est pas très actif, même s'il s'agit d'un projet largement utilisé.
VLC, qui est le moins critique des projets, se distingue par une mauvaise qualité du code avec des taux relativement élevés de comparaisons de pointeur avec 0 et de variables non utilisées. Cependant, les variables non utilisées ont été récemment nettoyées. Nous notons également qu'après une diminution des déréférences de pointeurs NULL (isnull et null_ref) à la mi-2008, le nombre de fautes de ce type a de nouveau augmenté dans les trois dernières versions étudiées. Les autres types de fautes sont stables dans le temps.
Wine est le second plus grand projet de cette étude. Il a un grand nombre de variables non utilisées qui ont été nettoyées en 2008. Nous notons également que depuis 2007 l'utilisation erronée des opérateurs de type bit/booléen (notand) a diminué significativement.

4.5. Évaluation

Pour toutes les versions de Linux, OpenSSL, VLC et Wine et pour les règles de recherche des fautes considérées, Coccinelle signale plus de 22 000 fautes pour une période d'un peu plus de 3 ans. Comme le montre le Tableau 3, Herodotos infère automatiquement plus de 99% des corrélations entre les fautes. 19 207 corrélations sont automatiques, tandis que seulement 220 sont initialement proposées à l'utilisateur pour vérification. À partir de ces propositions, et grâce au processus itératif, l'utilisateur peut se limiter à la corrélation explicite de seulement 146 paires de fautes. 108 de ces paires sont marquées `SAME` tandis que les 38 autres sont marquées `UNRELATED`.
Au travers de ces expérimentations, Herodotos a montré son utilité et son efficacité dans la corrélation des fautes signalées sur de multiples versions. Le calcul de statistiques a été simple à développer et l'intégration de nouvelles statistiques ne devrait pas poser de difficulté. Enfin, les représentations graphiques peuvent aider les développeurs à voir rapidement ce qui se passe au niveau de la qualité de leur logiciel. De nouveaux graphiques peuvent aisément être envisagés.

	Fautes signalées	Fautes automatiquement corrélées	Corrélations proposées initialement	Corrélations proposées par l'utilisateur		Fautes corrélées
Linux	16 440	14 031	142	118	0,81%	1 927
Wine	4 140	3 746	75	25	0,67%	381
VLC	1 313	1 152	2	2	0,17%	158
OpenSSL	331	278	1	1	0,36%	51
Total	22 224	19 207	220	146	0,74%	2 517

TAB. 3 – Efficacité de la corrélation

5. Travaux relatifs

Il existe déjà des études que ce soit pour la recherche de fautes à partir d'analyses statiques de code [1, 4, 5, 6, 17], ou l'historique des fautes fondé sur des systèmes de gestion d'erreurs [8, 11] ; mais rien ne semble avoir été entrepris dans la mise à disposition d'outils pour suivre les fautes en s'appuyant sur les modifications du code et l'analyse statique du code. Chou et coll. [4] ont réalisé une étude détaillée de l'historique de 12 types de fautes dans Linux jusqu'en 2001. Leur étude signale la propagation des fautes sur des versions successives mais aucune explication n'a été donnée sur sa réalisation et aucun outil n'a été fourni. Plus récemment, Li et coll. [10] ont mené une étude empirique sur des projets open source. Cependant, aucun outil pour construire automatiquement ou semi-automatiquement des historiques des fautes n'est mentionné et les fautes, ou plutôt erreurs, considérées proviennent d'un système de gestion d'erreurs.

L'analyse statique de code a essentiellement été utilisée pour rechercher des fautes dans les versions en développement d'un logiciel, mais sans considération pour une longue période de temps afin de construire un historique. L'historique des fautes a cependant été étudié en couplant un système de gestion de code source *(SCM)*, généralement CVS, avec un système de gestion d'erreurs. DynaMine [11] met en œuvre des techniques d'exploration de données sur les données collectées par le SCM afin d'identifier des motifs de code récurrents et spécifiques à un projet donné. Ces motifs peuvent ensuite être exploités pour rechercher des déviations représentant des fautes. ROSE [18] utilise la même technique pour suggérer des sites de code à modifier lors d'évolutions logicielles. Enfin, iBUGS [7] explore l'information présente dans les SCM pour inférer des fautes et les tests associés afin de construire un jeu de test pour les outils de recherche de fautes. Cependant, aucune de ces approches n'a été étendue avec l'utilisation d'un analyseur statique de code. Enfin, le SCM n'est pas exploité pleinement d'un point de vue temporel car ces approches se concentrent sur les évolutions entre deux révisions plutôt que les évolutions plus macroscopiques.

Lors de nos expériences, nous aurions pu utilisé, au lieu de GNU `diff` [12], l'algorithme Patience [2] qui construit une base de modification plus aisé à lire. Il aurait peut-être dans notre cas permis une meilleure corrélation automatique. Malheureusement, l'intégration dans notre approche d'un outil implantant cet algorithme n'a pas pu être effectué par manque de temps.

6. Conclusion

Dans cet article, nous avons présenté l'outil Herodotos qui construit automatiquement l'historique des fautes d'un logiciel, génère des représentations graphiques et calcule des statistiques sur ces fautes. Ce processus repose sur des outils existants pour inférer les modifications du code et la position des fautes. Pour dépasser les problèmes d'imprécision des outils sur lesquels il s'appuie, Herodotos permet à l'utilisateur de donner des informations pour améliorer la corrélation et éliminer les faux positifs issus de l'analyseur statique. Herodotos assiste l'utilisateur dans ce processus en proposant des corrélations possibles sur la base d'heuristiques.

Nous prévoyons de plus exploiter le SCM afin d'améliorer la corrélation. Le SCM git traque le contenu, et non pas les fichiers, et est donc capable d'inférer des changements de nom. Cette information aidera Herodotos à inférer plus de corrélations lorsqu'un fichier est renommé ou déplacé dans un autre répertoire. Enfin, Coccinelle permet à l'utilisateur de définir des motifs de fautes spécifiques à un logiciel [9]. Nous prévoyons d'exploiter cette fonctionnalité pour étudier des fautes spécifiques aux logiciels ainsi que de nouvelles catégories de fautes.

Lors de l'évaluation d'Herodotos, nous avons pu observer sur les quatre logiciels étudiés que le nombre de fautes avait tendance à soit rester stable, soit augmenter. Dans le cas de Linux, l'utilisation de Coccinelle, depuis maintenant un certain temps, montre des effets positifs significatifs. Une utilisation plus systématique serait donc souhaitable.

Disponibilité

Herodotos est disponible à l'adresse : `http://www.diku.dk/~npalix/herodotos/`

Remerciements

Je tiens à remercier Julia Lawall et Gilles Muller pour leurs relectures et leurs remarques.

Bibliographie

1. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 73–85, Leuven, Belgium, Apr. 2006.

2. S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2) :7–11, 2000.
3. J. Brunel, D. Doligez, R. R. Hansen, J. Lawall, and G. Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, Jan. 2009.
4. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 73–88, Banff, Canada, Oct. 2001.
5. P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Varieties of static analyzers : A comparison with ASTRÉE. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 3–20, Shanghai, China, June 2007.
6. Static source code analysis, static analysis, software quality tools by Coverity Inc. http ://www.coverity.com/, 2008.
7. V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE '07 : Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436, Atlanta, GA, USA, Nov. 2007.
8. M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, 2003.
9. J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, G. Muller, and N. Palix. WYSIWIB : A declarative approach to finding protocols and bugs in Linux code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, (DSN 2009)*, pages 43–52, Estoril, Portugal, June 2009.
10. Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now ? : an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, San Jose, CA, 2006.
11. B. Livshits and T. Zimmermann. DynaMine : finding common error patterns by mining software revision histories. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296–305, Lisbon, Portugal, Sept. 2005.
12. D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003.
13. Mitre. Common Weakness Enumeration. `http://cwe.mitre.org/`.
14. Org-mode homepage. `http://orgmode.org/`.
15. Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.
16. N. Palix, J. L. Lawall, and G. Muller. Herodotos : A tool to expose bugs'lives. Research report RR-6984, INRIA, July 2009.
17. D. Wheeler. Flawfinder home page. Web page : `http://www.dwheeler.com/flawfinder/`, Oct. 2006.
18. T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6) :429–445, 2005.