Titre: Title:	Analyse de similarité de kits de phishing en PHP, HTML et JavaScript
Auteur: Author:	Mathieu Margier
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Margier, M. (2021). Analyse de similarité de kits de phishing en PHP, HTML et JavaScript [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/9161/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Analyse de similarité de kits de phishing en PHP, HTML et JavaScript

MATHIEU MARGIER
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2021

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Analyse de similarité de kits de phishing en PHP, HTML et JavaScript

présenté par **Mathieu MARGIER**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Marios-Eleftherios FOKAEFS, président
Ettore MERLO, membre et directeur de recherche
Guy-Vincent JOURDAN, membre et codirecteur de recherche
Mohammad HAMDAQA, membre

DÉDICACE

À la mémoire de mon cousin Nolan...

REMERCIEMENTS

Ce travail n'aurait pas pu être possible sans le soutient, direct ou indirect, de plusieurs personnes ou organismes.

Dans un premier temps, je souhaite remercier grandement mon directeur de recherche, le professeur Ettore Merlo, pour ses conseils avisés et son implication tout au long de ma maîtrise. Également, je tiens à remercier le professeur Guy-Vincent Jourdan de l'Université d'Ottawa pour avoir co-supervisé mon projet de recherche.

Je remercie les membres du jury, les professeurs Marios-Eleftherios Fokaefs et Mohammad Hamdaqa, de m'avoir fait l'honneur d'évaluer mon mémoire et d'avoir apporté leur regard critique éclairant mon travail sous une nouvelle perspective.

Je remercie également Stéphane Heudron pour son travail préliminaire qui a constitué la base sur laquelle s'est construit ce travail.

Je tiens à remercier IBM *Center for Advanced Studies* (CAS) et le Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG) pour leur aide financière qui m'a permis de me concentrer sur la recherche pendant mon séjour au Québec.

Mes gratitudes vont également au professeur Gregor v. Bochmann de l'Université d'Ottawa et au docteur Iosif-Viorel Onut d'IBM CAS pour leurs remarques constructives lors de nos réunions hebdomadaires.

Je remercie les équipes de Polytechnique Montréal et de l'École Centrale de Lyon, qui m'ont permis de réaliser mon double diplôme grâce à leur programme d'échange.

J'adresse un merci particulier à mes collègues Julien Cassagne et Emad Badawi avec qui j'ai eu le plaisir de travailler au sein de notre groupe de collaboration entre Polytechnique Montréal et l'Université d'Ottawa.

Enfin, je remercie chaleureusement ma famille et mes amis pour leur soutien indéfectible durant cette période hors norme.

RÉSUMÉ

Le *phishing* est une menace qui demeure active et en pleine croissance. Avec la pandémie de la COVID-19, l'année 2020 a vu un pic avec plus de 200 000 attaques signalées. Les attaquants déploient typiquement du code source sur un serveur de site web pour usurper une marque, ou imiter une situation dans laquelle il est attendu de l'utilisateur de renseigner des informations personnelles que les attaquants convoitent (comme des identifiants, numéros de carte de crédit, etc.). Les kits de *phishing* sont un ensemble de fichiers prêts à être déployés, qui peuvent simplement être copiés sur un serveur web et être quasiment utilisés tels quels.

Les attaques de *phishing* sont un sujet qui a été largement exploré sous l'angle de la contre-mesure, avec la détection de *spam* dans les courriels, ou l'identification de sites web de *phishing*. Cependant, peu d'études se focalisent sur le code source côté serveur des kits de *phishing*, en partie à cause de sa difficulté d'accès.

Dans ce travail, on considère l'analyse statique de la similarité du code source PHP, JS, HTML de 20 871 kits de *phishing*, totalisant plus de 180 millions de lignes de code. Ces kits ont été collectés pendant des attaques de *phishing* par des équipes de sécurité informatique. L'approche proposée peut aider à classer des kits collectés comme étant des « proches copies » ou « sauts intellectuels » de kits connus et déjà rencontrés. Cela pourrait faciliter l'identification et classification des nouveaux kits comme des variants de kits plus anciens et connus, et ainsi aider à rationaliser les efforts de contre-mesure.

Pour ce faire, on applique une approche basée sur la fréquence des types de jetons au code source des kits de *phishing* à notre disposition, en considérant chaque fonction ou méthode comme un fragment. Les clones identiques (type 1) et paramétriques (type 2) des fragments et des kits sont identifiés. Pour chaque langage étudié, une généalogie mono-langage est présentée et analysée en connectant ensemble les kits possédant les degrés de similarité les plus élevés. Une généalogie multi-langage est également étudiée, en combinant les trois généalogies mono-langages. La validité de la construction des généalogies est validée en utilisation 235 versions de WordPress, un logiciel *open-source* populaire.

Les résultats expérimentaux montrent que pas moins de 56 % des kits analysés ont des clones paramétriques, c'est-à-dire des petits changements tels que le renommage d'identificateurs ou la modification de valeurs de constantes. De plus, ces clones ne sont pas équitablement répartis : 70 % de tous les kits font partie de seulement 30 % des groupes de kits paramétriques. Seulement un tiers des kits n'ont pas de clone, que ce soit identique ou paramétrique. Pourtant, ces kits restent très similaires : les différences sont faibles, avec moins de 1 000 je-

tons (tels que les identificateurs, constantes, chaînes de caractères et autres valeurs) pour presque la majorité des cas. Par ailleurs, PHP a été identifié comme le principal langage utilisé dans le jeu de données. La répartition observée de la similarité des kits est cohérente avec l'hypothèse supposée sur le modèle de propagation des kits. Souvent, la propagation des kits est basée sur des copies identiques ou quasi identiques à bas coût pour les attaquants.

ABSTRACT

Phishing is still very much an active and growing problem. With the COVID-19 pandemic, the year 2020 saw a peak with more than 200,000 phishing attacks reported. Attackers typically deploy source code in some host website to impersonate a brand, or in general a situation in which a user is expected to provide some personal information of interest to phishers (e.g. credentials, credit card number, etc.). Phishing kits are ready-to-deploy sets of files that can be simply copied to a web server and used almost as they are.

Phishing attacks is a topic that has been widely explored from the perspective of countermeasure, such as spam detection in emails or phishing website identification. However, few studies have investigated the server-side source code of phishing kits, partly due to its difficulty of access.

In this work, we consider the static similarity analysis of the PHP, JS, HTML source code of 20,871 phishing kits totaling over 180 million lines of code, that have been collected during phishing attacks and recovered by forensics teams. The proposed approach may help classifying new incoming phishing kits as "near-copy" or "intellectual leaps" from known and already encountered kits. This could facilitate the identification and classification of new kits as derived from older known kits, and help rationalize countermeasure efforts.

We apply an approach based on token type frequency to the source code of the phishing kits at our disposal, considering each function or method as a fragment. Identical (type 1) and parametric (type 2) clones of fragments and kits are identified. For each considered language, a plausible single-language genealogy of phishing kits is presented and analyzed by connecting together kits with the highest similarity. A multi-language genealogy is also studied, combining all the three single-language genealogies. The validity of genealogy construction has been validated using 235 versions of WordPress, a popular open-source PHP software.

Reported experimental results show that as much as 56% of the analyzed kits have parametric clones, with small changes such as identifiers renaming or constant values modification. In addition, those clones are not equally distributed : 70% of the kits belong to 30% of the parametric groups. Only a third of the kits have no identical nor parametric clone. Yet, those kits are still highly similar : differences are small, and less than about 1,000 programming words, such as identifiers, constants, strings, or other values, in almost the majority of cases. Furthermore, PHP has been identified as the main language used in this dataset. Observed kits similarity distribution is consistent with the assumed hypothesis about kits propagation model. Often kit propagation is based on identical or near-identical copies at low cost changes.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

APWG	*Anti-Phishing Working Group* (association industrielle à but non lucratif luttant contre les attaques de *phishing*)
AST	Arbre de syntaxe abstrait, ou *Abstract Syntax Tree*
CSS	*Cascading Style Sheets* (langage informatique utilisé pour le web)
HTML	*Hypertext Markup Language* (langage informatique utilisé pour le web)
JC	Coefficient de Jaccard, ou *Jaccard Coefficient*
JS	*JavaScript* (langage de programmation utilisé pour le web)
LCS	Plus longue sous-séquence commune, ou *Longest Common Subsequence*
LOC	Lignes de code, ou *Lines Of Code*
MDS	Positionnement multidimensionnel, ou *Multidimensional Scaling*
MST	Arbre couvrant de poids minimal, ou *Minimum Spanning Tree*
PHP	*PHP Hypertext Preprocessor* (langage de programmation utilisé pour le web)
PPV	Plus Proche Voisin, ou *Nearest Neighbor (NN)*
RD	Différence relative, ou *Relative Difference*
URL	*Uniform Resource Locator* (adresse web)

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Les attaques de *phishing*, bien que connues et étudiées depuis des décennies, demeurent aujourd'hui encore un problème majeur en cybersécurité. Dans son dernier rapport trimestriel [1] paru en février 2021, l'Anti-Phishing Working Group (APWG) recense une augmentation commensurable du nombre d'attaques de *phishing*. Ainsi, on apprend que le nombre d'attaques a doublé entre le début et la fin de l'année 2020. Les attaquants ont profité de la période de crise avec la pandémie de la COVID-19 pour diversifier leurs thèmes [2]. Cela a alors contribué à inverser la légère tendance à la baisse que l'on observait auparavant. Afin de réduire le nombre d'attaques, plusieurs pistes sont envisageables : accélérer la détection des sites de *phishing* pour les bloquer par les navigateurs internet le plus tôt possible, améliorer la précision des filtres de pourriels, mieux éduquer les utilisateurs à reconnaître ces attaques, etc. Dès lors, il semble crucial de connaître correctement le fonctionnement derrière les sites de *phishing*, dans le but d'organiser une défense efficace. L'analyse de leur similarité est ainsi une piste permettant de rationaliser les efforts de contre-mesures.

1.1 Définitions et concepts de base

1.1.1 Analyse statique

Le code source, généralement composé de plusieurs fichiers, peut être découpé en plusieurs morceaux, que l'on nomme **fragments**. La définition de ces fragments est arbitraire, et dépendante des buts fixés. Dans le cadre de ce mémoire, la définition retenue est la suivante : un fragment correspond à une **fonction** ou à une **méthode**. Également, on compte le code source d'un fichier en retirant toutes les fonctions et méthodes comme un fragment.

Lors de l'analyse lexicale, le code source est transformée en une série de **jetons**, unités élémentaires d'un code source et propre à chaque grammaire de langage de programmation. Chaque jeton possède une **image** (la chaîne de caractères telle qu'elle apparaît dans le code source) et un **type** (un numéro attribué par l'analyse lexicale pour catégoriser les jetons, par exemple les identifiants, opérateurs, etc.) Par la suite, l'étape d'analyse syntaxique produit un arbre, l'arbre de syntaxe abstrait (AST). Grâce à celui-ci, on peut alors identifier les fragments dans le code source, et en extraire les informations voulues. Les étapes d'analyses lexicale et syntaxique sont généralement regroupées dans un même outil, le **parseur**.

Analyser le code source à l'aide d'un parseur permet de réaliser de l'**analyse statique**. Il s'agit d'étudier le code source sans l'exécuter, en approximant le comportement attendu

des programmes. Au contraire, l'**analyse dynamique** consiste à exécuter les programmes, et étudier leurs comportements. Il est possible de combiner les deux approches avec de l'**analyse hybride**.

1.1.2 Clones

Quand les codes source de deux fragments sont similaires, on parle d'une paire de **clones**. Un clone peut être à l'origine un fragment qui a été copié, et qui par la suite a été plus ou moins modifié.

Bien qu'il n'y ait pas de définition formelle consensuelle des types de clones, on observe principalement dans la littérature [3] la définition des quatre types suivants :

- **Type 1** : fragments de code syntaxiquement identiques, sauf pour des changements dans les espaces, retours à la ligne, et commentaires (changements qui n'affectent pas les informations syntaxiques, qui sont ignorés lors de l'analyse lexicale). On y réfère par la suite en tant que clones « **identiques** ».
- **Type 2** : fragments de code syntaxiquement identiques, sauf pour des changements dans les noms des identificateurs, des types, et des valeurs des littéraux (constantes). Ils peuvent aussi avoir les caractéristiques que les clones de type 1, avec des changements dans les espaces, retours à la ligne, commentaires. On y réfère par la suite en tant que clones « **paramétriques** », puisqu'ils peuvent être obtenus en changeant les valeurs de certains paramètres (constantes, noms d'identifiants, etc.). Il faut cependant noter que le remplacement paramétrique n'est pas nécessairement consistant, par exemple un même nom d'identifiant peut être renommé alternativement avec plusieurs noms différents.
- **Type 3** : fragments de code avec des modifications structurelles telles que des substitutions, ajouts, suppression ou permutation des instructions. On peut également avoir des modifications ayant les caractéristiques des clones de type 1 et 2 (changements syntaxiques). On y réfère par la suite en tant que clones « **similaires** », mais on trouve également les termes de « *gapped* » *clones* ou « *near-miss* » *clones* dans la littérature.
- **Type 4** : fragments de code effectuant le même calcul, mais implémentés avec des syntaxes différentes. On y réfère en tant que clones « **sémantiques** » dans la littérature.

Les définitions des clones de type 3 et 4 sont plus vagues et soumises à l'interprétation, puisqu'il ne s'agit pas de définitions formelles. D'autres définitions existent, adaptés pour des buts différents, comme pour les systèmes orienté-objet [4] ou pour la détection de mutation [5].

Un exemple est donné pour des fragments identiques (type 1), paramétriques (type 2), similaires (type 3). La figure 1.1 présente un fragment de référence en PHP. La figure 1.2 donne un exemple de fragment identique : le code est identique, excepté pour l'ajout d'un commentaire, et les espaces. La figure 1.3 donne un exemple de fragment paramétrique : seules quelques valeurs sont modifiées, qui sont le nom de la fonction et la chaîne de caractère. Les types de jetons restent identiques. Enfin, la figure 1.4 représente un fragment similaire : une instruction a été ajoutée, et une autre modifiée. Dans ce cas, la fonction a le même comportement, mais cela n'est pas garanti de manière générale.

```
1 function f() {
2     echo "Message";
3 }
```

FIGURE 1.1 Exemple – Code PHP du fragment de référence

```
1 function f() {
2 // Fragment identique
3 echo "Message";
4 }
```

FIGURE 1.2 Exemple – Code PHP d'un fragment identique

```
1 function g() {
2     echo "Message different";
3 }
```

FIGURE 1.3 Exemple – Code PHP d'un fragment paramétrique

Dans le cadre de ce mémoire, nous nous intéressons aux clones de type 1, 2 et 3, que l'on nomme respectivement clones identiques, paramétriques, et similaires.

1.1.3 Kit de phishing

Le *phishing* est défini par l'APWG comme un crime employant à la fois de l'ingénierie sociale et des subterfuges technologiques pour voler les données personnelles et les identifiants financiers des consommateurs [1] (traduction littérale).

Lors d'une attaque de *phishing*, les attaquants essaient de piéger des internautes afin de récupérer ces informations confidentielles. Pour cela, ils doivent disposer du code source d'un

```
function f() {
    $msg = "Message";
    echo $msg;
}
```

FIGURE 1.4 Exemple – Code PHP d'un fragment similaire

site web fonctionnel, qui est écrit dans le but d'imiter une marque connue pour gagner la confiance de l'utilisateur. Le site va alors demander à l'utilisateur de renseigner les informations désirées par l'attaquant, telles que le numéro de carte de crédit, le mot de passe d'un autre site, etc.

L'élaboration d'une attaque de *phishing* se fait généralement en quatre phases :

1. choix de la cible : l'attaquant recherche quelle marque, pays, ou organisation imiter, et définit également le public cible pour son attaque ;
2. création du site : l'attaquant développe, que ce soit lui-même ou par un tiers, le code source d'un site web, ayant l'air légitime, qui demandent aux utilisateurs de renseigner leurs informations confidentielles, qui sont alors transmises à l'attaquant ;
3. déploiement du site : l'attaquant peut louer un serveur web chez un hébergeur peu regardant, ou bien compromettre un site pré-existant en le piratant, et y déployer le code source de son site de *phishing*, tout en installant et configurant le serveur web ;
4. campagne de communication : l'attaquant communique l'adresse URL de son site malicieux à un grand nombre d'utilisateurs, le principal moyen utilisé étant les courriels.

Par souci de temps et de coût, il est possible de se procurer des kits de *phishing* [6, 7] au lieu d'écrire le code source de zéro. Ces kits se présentent sous la forme d'un ensemble de fichiers, code source et ressources (graphiques, polices, etc.), qui forment un site fonctionnel permettant de soutirer des informations aux utilisateurs et de les transmettre à l'attaquant. Parmi les fichiers sources, on retrouve souvent le PHP comme langage de programmation en *backend*, et HTML, CSS et JS en *frontend*. L'attaquant doit ainsi seulement renseigner quelques paramètres, dont le moyen de communication des informations récupérées à l'insu de l'utilisateur. Dans la quasi-totalité des cas, ces informations sont envoyées à une adresse e-mail [7–13]. Une fois le kit configuré, il suffit à l'attaquant de déployer le code source sur un serveur web, et de communiquer l'adresse du site à une grande quantité de personnes avec des campagnes de *phishing*.

Les sites de *phishing* faisant l'objet de contre-attaques de la part de l'industrie de la sécurité informatique, certains kits de *phishing* possèdent des fonctionnalités supplémentaires, des-

tinées à compliquer la détection de leur site. Ces fonctionnalités incluent la randomisation des URLs [14], le masquage côté client [15] ou côté serveur [10]. L'obfuscation du code est également utilisé dans certains cas pour envoyer les informations à un autre destinataire, et à l'insu de l'attaquant utilisant le kit [7, 14, 16].

1.2 Éléments de la problématique

La recherche sur la lutte contre les attaques de *phishing* se concentre majoritairement sur deux aspects : les moyens de détection des courriels de *phishing* que reçoivent les victimes [17–22], et les moyens de détection des pages web de *phishing* [23–31]. Ces différentes techniques s'intéressent aux attaques du point de vue du client. Au contraire, les études s'intéressant au point de vue du serveur (et donc de l'attaquant) sont plus rares dans la littérature. La difficulté d'accès à ces informations en est probablement une cause, puisque par définition ils font avoir accès aux serveurs utilisés pour les sites de *phishing*, tandis que l'accès aux courriels et aux sites web côté client est bien plus simple. Un des moyens de se procurer des kits de *phishing* à des fins d'analyse est l'utilisation de pot de miel (*honey pot*), comme décrit dans l'article de Han *et al.* [14]. De manière générale, l'analyse de kits est étudiée dans le milieu académique grâce à des bases de données fournies par des entreprises de sécurité informatique [10, 12].

Dans le cadre de ce travail, IBM nous a fourni une base de données contenant le code source de 20 871 kits de phishing. De cette manière, il est possible de faire de l'analyse statique sur ces kits afin d'évaluer la similarité entre les codes sources. La présence de code dupliqué, avec une certaine quantité de modifications, entre deux kits fournit alors un indice sur un potentiel lien entre ses auteurs, et révèle également plus d'informations sur le partage de certaines fonctionnalités des kits au sein de l'écosystème du *phishing*. L'analyse du code source se fait selon les principaux langages utilisés, à savoir PHP, HTML et JS.

En analysant la similarité entre des logiciels, il est possible d'inférer une généalogie, qui représenterait l'évolution possible de ces logiciels, au fur et à mesure des différentes versions. Toutefois, appliquer cette analyse à des kits de *phishing* présente des difficultés supplémentaire. Généralement, l'analyse de l'origine des clones reposent sur les informations contenues dans un système de versionnage [32–35]. Or, ces informations ne sont pas présentes dans les kits. Il est même possible que des kits aient été développés sans système de versionnage, ce qui rend cette information d'autant plus inaccessible. La démarche entreprise ici est donc de partir des mesures de similarité, afin d'inférer une potentielle généalogie, non exacte, qui rend compte de l'évolution probable des kits. Cette généalogie peut être vue comme une combinaison de « *forks* », où un logiciel est copié et modifié en parallèle de l'original. D'autres

difficultés s'ajoutent à celles précédemment mentionnées. Tout d'abord, les kits en notre possession ne représentent pas nécessairement une image fidèle des sites de *phishing* que l'on peut observer. En effet, seule une partie des kits a été collectée : certaines versions des kits peuvent être manquantes, créant ainsi des trous dans l'arbre évolutif des kits. D'une manière similaire, des versions intermédiaires des kits peuvent exister, sans avoir été déployées sur des sites, rendant leur collecte impossible. Ces versions manquantes, susceptibles de contenir des modifications mineures, créent alors des sauts évolutifs entre kits avec des modifications plus importantes. Le fait que certains auteurs de kit puissent partager des morceaux de code crée une autre difficulté. Cela peut alors augmenter la similarité entre différents kit, sans qu'il n'y ait de lien évolutif entre les kits pour autant. L'absence de date, outre celle de la collecte du kit, est un indice de moins pour prédire la généalogie, puisqu'on ne peut pas l'utiliser pour savoir qui est le kit originel entre deux kits. Également, l'absence d'oracle rend compliqué l'évaluation de la qualité de la généalogie prédite. Enfin, l'évolution des logiciels malveillants diffère de celle du logiciel dit « conventionnel ». Alors que le logiciel « conventionnel » suit généralement les lois de Lehman [36], en augmentant son nombre de lignes de code (LOC) à chaque version par exemple, le logiciel malveillant peut adopter des changements différents, dans le but de camoufler son fonctionnement. Par exemple, il est possible, à bas coût, de rajouter une grande quantité de code mort, jamais exécuté, dans l'unique but de noyer le code effectif du kit dans du code non pertinent. Il est aussi facilement possible de permuter les fonctions au sein d'un fichier, rendant un fichier différent d'une version à l'autre, sans pour autant modifier son fonctionnement. Il est également possible que certains fichiers soient remis à des versions antérieures, pour diverses raisons.

Ainsi, plutôt que de chercher une généalogie en adéquation avec l'évolution temporelle des kits, le but recherché de la généalogie prédite est plutôt de souligner les liens de similarité observés entre les kits. Les kits avec très peu de différences dans leur code source seront alors liés, avec occasionnellement des plus grandes différences pour certains kits.

La démarche entreprise dans ce mémoire peut servir de fondation pour un outil d'analyse *forensic*. À partir d'une base de données de kits de *phishing* connus, et d'un kit nouvellement identifié, on peut procéder à plusieurs analyses. Un mode interactif pourrait donner le plus d'informations possible au sujet de ce kit : a-t-il des clones identiques ou paramétriques (voir section 3.3.3), quels sont les plus proches voisins du kit dans la généalogie (section 3.3.8) et visualisation des changements apportés (section 3.3.9). On pourrait également voir la chaîne de propagation des modifications en partant d'un kit donné pour arriver jusqu'à ce nouveau kit.

Voici un scénario d'application d'une telle analyse : un site web suspect est découvert, on a

alors accès à son code source HTML et JS. On suppose que ce code source côté client est identique (ou très proche) au code source côté serveur. On peut alors vérifier si le code HTML, JS, ou les deux est déjà utilisé dans l'un des kits connus. Si c'est le cas, cela peut permettre d'identifier potentiellement le kit qui est utilisé par le site. Sinon, on peut placer le site dans les généalogies portant uniquement sur le code HTML et JS, trouver des kits similaires, et visualiser les différences. Cela peut également conduire à identifier le kit utilisé, et déduire le cas échéant qu'une nouvelle version a été développée, dont on ne connaît uniquement les changements côté client.

Un autre exemple d'analyse possible est le suivant : supposons que deux kits soient identifiés comme similaires en HTML et JS, mais très différents en PHP. Cela signifierait que les deux kits partagent le même *front-end*, mais avec des *back-end* différents. Le fait que le code PHP soit différent suggère que ces deux kits ont été écrits par des auteurs différents. De plus, le fait de partager un *front-end* similaire suggère que les auteurs se sont mutuellement copiés (ou qu'ils aient copié un tiers), cette partie étant accessible du point de vue du client contrairement au PHP. Ces informations pourraient servir à mieux comprendre le fonctionnement derrière l'industrie du *phishing*.

1.3 Objectifs de recherche

L'hypothèse suivante est adoptée pour ce travail de recherche : les auteurs de kits effectuent leurs changements avec le moins d'effort possible.

Les objectifs de la recherche sont la reconstruction optimale des liens de voisinages qui minimisent l'effort global de modification entre les kits à notre disposition, ainsi que son analyse pour identifier la similarité des kits. Cette reconstruction aboutit alors à une pseudo-généalogie plausible des kits, où des kits similaires traduisent un lien évolutif probable entre ces deux kits. Cependant, la pseudo-généalogie reste approximative et n'a pas pour but de déterminer exactement l'historique des kits, mais plutôt de souligner les liens de similarité observés pour étudier à quel point les kits partagent du code source. Le terme de généalogie est alors employé tout au long du mémoire pour désigner cette pseudo-généalogie.

1.4 Plan du mémoire

Ce mémoire est organisé en six chapitres, incluant la présente introduction. Dans le second chapitre, une revue de littérature sur les domaines en lien avec ce travail est présentée. Ensuite, le troisième chapitre décrit l'approche utilisée pour répondre aux objectifs de recherche en partant du code source à notre disposition. Par la suite, le quatrième chapitre propose de

présenter les jeux de données étudiés, et d'établir des expériences pour analyser les généalogies produites de la manière décrite dans l'approche. Le cinquième chapitre rapporte les résultats des expériences définies précédemment, et apporte une discussion de ces résultats. Finalement, le sixième et dernier chapitre synthétise les travaux et résultats obtenus, tout en mentionnant les limitations et améliorations futures.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Phishing

L'histoire du *phishing* remonte principalement vers le milieu des années 1990 [37]. Le terme, qui est traduisible par « hameçonnage » en français, est un dérivé du terme anglais *fishing*, dans le sens où les attaquants partent à la « pêche » aux identifiants et mots de passes des utilisateurs. La démocratisation de l'informatique personnel dans cette décennie a permis le ciblage massif des informations personnelles. À ce moment, le fournisseur d'accès internet *America Online* (AOL) dominait alors le marché aux États-Unis. Cela en a fait la cible des premiers systèmes de *phishing* automatisés rendus publiques comme AOHell [37]. Ce logiciel permettait à n'importe qui l'utilisant d'automatiser la collecte de mots de passes et de numéros de cartes de crédit, en générant des messages pour se faire passer pour des employés d'AOL. Le *phishing* existait déjà auparavant, mais les attaques étaient faites de manière manuelle, ou de manière automatique sans que le logiciel soit rendu publique.

À partir des années 2000, les attaques de *phishing* se sont par la suite améliorées et professionnalisées, engendrant alors toute une industrie du crime informatique. En 2001, une attaque de grande envergure cible le *E-gold* [38], un système de paiement électronique, après le versement d'indemnités suite aux attaques terroristes du 11 septembre contre le *World Trade Center*. La première attaque ciblant une banque est rapportée en 2003 [39]. À partir de 2004, le *phishing* est reconnu comme une menace majeure [40], avec l'existence de groupe organisé mettant au point des attaques de plus en plus complexes, avec par exemple l'ajout de virus en pièce jointe des polluriels.

Depuis 2004, l'APWG établit des rapports [1] (d'abord mensuels, puis trimestriels) pour suivre l'évolution de la tendance des attaques de *phishing*. Ils observent alors que le nombre d'attaques uniques passent de moins de 200 000 en 2005 à plus de 1,4 millions en 2015, témoignant de la très forte dynamique du *phishing* en une décennie. Par la suite, le nombre d'attaques uniques a diminué de manière consistante jusqu'à 500 000 en 2019, grâce à l'amélioration des outils de filtrage de pourriels et à l'éducation des utilisateurs. De plus, les navigateurs internet bloquent les sites de *phishing* lorsqu'ils sont détectés, en se reposant sur des listes de blocage comme le service SafeBrowsing [2] de Google. Cependant, l'année 2020 a vu de nouveau une hausse du nombre d'attaques, principalement dû au contexte provoqué par la pandémie de la COVID-19 [2]. Un changement notable des dernières années concernant les attaques

1. `https://apwg.org/trendsreports/`
2. `https://safebrowsing.google.com/`

de *phishing* est le recours de plus en plus massif au chiffrement en utilisant le protocole HTTPS [1] : seul 10 % des sites de *phishing* en avaient recours au premier trimestre de 2017, contre plus de 80 % au dernier trimestre de 2020. L'utilisation de ce protocole permet de tromper les utilisateurs, en détournant les fonctionnalités de sécurité des navigateurs internet leurs intentions premières. Ainsi, les utilisateurs qui visent ces sites de *phishing* peuvent alors penser qu'ils sont sur des sites de confiance.

Depuis la dernière décennie, on observe une nouvelle tendance dans l'industrie du crime informatique : des modèles *Crimeware as a Service* (CaaS) [41] se développent de plus en plus. Cela consiste à louer des services pour réaliser des attaques informatiques à des clients, ce qui induit une séparation entre les vendeurs, qui se contentent de fournir leurs services contre paiement, et les clients, qui choisissent leur cible sans avoir à se soucier de la réalisation technique. Ce phénomène concerne également le *phishing* [41], ce qui va en quelque sorte dans la même direction que l'introduction des kits de *phishing*. En plus de ne pas avoir besoin de développer un site web fonctionnel, les attaquants peuvent ainsi louer les services d'un organisateur pour héberger, personnaliser leur site de *phishing*, et déléguer la campagne de communication de l'adresse web à un publicitaire en précisant simplement le profil du public cible. Pour l'instant, il semblerait que les auteurs de kits n'aient pas recours à des architectures favorisant la ré-utilisabilité, comme le *Service-oriented architecture* (SOA).

Plusieurs types d'attaques de *phishing* sont répertoriés dans la littérature [42] :

1. ingénierie sociale : exploitation des failles des utilisateurs humains pour obtenir des informations confidentielles ;
2. subterfuge technique : utilisation de *malwares* pour infecter des utilisateurs ;
3. mobile : attaques exploitant les particularités des terminaux mobiles.

Parmi les techniques d'ingénierie sociale, on retrouve l'usurpation de site web, où les attaquants répliquent l'apparence d'un site de confiance. Il s'agit du type d'attaques étudié dans ce mémoire, qui ont été collectées dans le jeu de données à notre disposition, sans prendre en compte le médium utilisé pour la campagne de communication. Également, on y trouve l'usurpation de courriels, où l'attaquant se fait passer pour une personne ou institution de confiance en envoyant des courriels à un grand nombre de destinataires. Ces courriels peuvent contenir un lien vers un site usurpé, ou contenir des logiciels malicieux en pièce jointe. Un cas particulier est le *spear phishing*, où l'attaque est conçue pour des cibles précises. Généralement, l'activité en ligne de certaines personnes influentes est scrutée par ces attaquants dans le but de personnaliser le plus possible l'attaque, de manière à ne pas éveiller les soupçons de la victime.

Les subterfuges techniques comprennent, entre autres, l'utilisation de rançongiciel, chiffrant les données de l'utilisateur et lui demandant une rançon pour les déchiffrer ; les chevaux de Troie, ayant une apparence bénigne ; les *keyloggers*, capables d'enregistrer tout ce qu'écrit l'utilisateur.

Avec la démocratisation des terminaux mobiles, de nouvelles techniques de *phishing* ont vu le jour. Les attaquants peuvent effectuer du *smishing*, en envoyant des SMS aux utilisateurs ; déployer des fausses applications ayant l'air légitimes mais qui collecte les données de l'utilisateur et les transmettent à l'attaquant ; se passer pour une institution bancaire ou une agence du gouvernement par des appels téléphoniques avec le *vishing* ; usurper des accès Wi-Fi et demander aux utilisateurs de se connecter pour récupérer leurs identifiants.

De manière générale, les attaquants ont recours aux moyens de communications suivants pour contacter leurs cibles [42] : courriels, réseaux sociaux en ligne, SMS, messageries instantanées, commentaires sur les blogs ou forums.

Historiquement, la campagne de communication du lien vers un site de *phishing* passe par l'envoi massif de courriels à de nombreux utilisateurs, afin de toucher le plus grand public possible. Cela permet d'obtenir un grand nombre de visites sur leur site web, même si le taux de personnes visionnant le courriel et qui décident de cliquer sur le lien est faible. Ces courriels doivent tromper la vigilance des lecteurs, en se faisant passer par une personne familière ou une marque connue. Ainsi, plus le courriel est convaincant, plus le taux d'utilisateurs cliquant sur le lien augmente. La figure 2.1 présente une capture d'écran d'un courriel fictif pouvant être distribué lors de telles campagnes de *phishing*. Écrit en anglais, l'auteur du courriel se fait passer pour une institution bancaire reconnue (bien que fictive dans cet exemple). On constate dans l'exemple quelques fautes d'orthographe (*recieved*, *discrepency*), que l'on retrouve fréquemment dans les polluriels [43]. Le destinataire peut ne pas avoir de compte chez la banque usurpée, ce qui réduit alors le nombre de cibles effectives, mais n'est généralement pas un problème trop important étant donnée la grande quantité de personnes ciblées par ces campagnes de *phishing*. L'auteur du courriel prétexte une situation inhabituelle qui provoque un sentiment d'urgence chez le destinataire, et qui peut l'inciter à visiter le site immédiatement, sans prendre le temps d'exercer son esprit critique sur le contenu du courriel. Une fois le lien visité, le site peut alors demander à l'utilisateur de renseigner ces informations de connexion (identifiant et mot de passe, numéro de carte de crédit, etc.). L'attaquant peut alors récupérer ces informations et décider de les utiliser, les revendre sur le marché noir, etc.

3. Andrew Levine, Domaine publique, via Wikimedia Commons : `https://commons.wikimedia.org/wiki/File:PhishingTrustedBank.png`

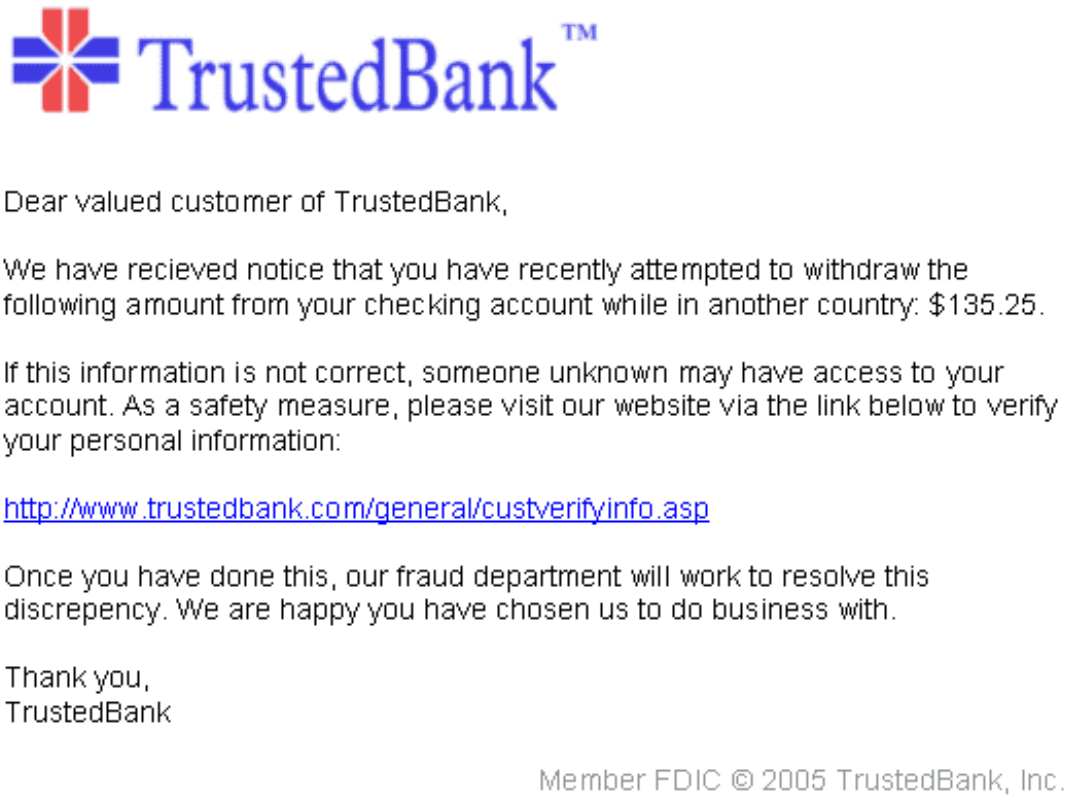


FIGURE 2.1 Exemple de courriel d'une campagne de *phishing* – image libre de droits [3]

Plusieurs approches sont possibles pour lutter contre le *phishing* [42]. Les attaques de *phishing* profitent de la confiance de certains utilisateurs pour réussir. Ainsi, un des principaux moyens de lutte contre ces attaques est l'éducation des utilisateurs. Des campagnes de sensibilisation permettent d'informer les usagers sur le fonctionnement et les caractéristiques de telles attaques, afin de ne pas tomber dans le piège des attaquants. Ces entraînements peuvent se faire sous la forme de jeux sérieux dans une entreprise, de sorte que les employés soient immergés dans une situation vraisemblable et apprennent de leurs erreurs sans conséquences néfastes. Également, des outils technologiques peuvent aider les utilisateurs à juger de la dangerosité potentielle d'un lien ou d'un courriel. Des extensions de navigateurs peuvent afficher un avertissement sur des pages web ayant l'air suspicieuses, ou bien un avertissement peut être émis lorsqu'un courriel provient d'une source externe à l'organisation, indiquant ainsi à l'usager d'être vigilant.

Les courriels étant l'un des moyens de diffusions privilégiés lors des attaques de *phishing*, l'élaboration de filtre bloquant les pourriels est une piste majeure de contre-mesure. La classification d'un courriel entre *ham* (contenu légitime) et *spam* (contenu indésirable) est un

problème bien adapté à l'apprentissage machine. De nombreux modèles ont été proposés, qui utilisent du SVM (*Support-Vector Machines*) [17], ou diverses techniques classiques d'apprentissages machines sur le contenu et l'en-tête des courriels [18–20]. Des travaux se sont également concentrés sur des approches plus spécifiques, par exemple avec le traitement naturel de la langue [21], ou en se focalisant sur le comportement des auteurs de courriels pour déjouer le *spearphishing* [22], qui habituellement échappe aux filtres traditionnels de par leur ingéniosité. Ces techniques ont l'avantage d'être en permanence améliorées puisqu'elles sont régulièrement entraînées sur les nouvelles attaques connues. Cela permet ainsi de créer des filtres avec une précision très élevée, et un faible taux de faux positifs, permettant ainsi de réduire l'exposition des utilisateurs aux *spams* sans bloquer les courriels légitimes.

L'autre axe majeur technologique pour contrer les attaques de *phishing* est la détection des sites de *phishing*. Tout comme les courriels, les approches d'apprentissage machines peuvent être appliqués aux sites [31], en extrayant d'une page web les informations jugées pertinentes (URL, mots clés, etc.). Également, un certain nombre de techniques s'inspire de ce qui est fait pour les moteurs de recherches. Les logos [28] ou *favicons* [29] présents sur les pages web peuvent être comparés aux images des sites officiels, mieux référencés par les moteurs de recherche, pour identifier les cas d'usurpation de marques. Également, le nom de domaine [30] peut être étudié, tout comme le fait que les pages associées à des pages de *phishing* déjà connues peuvent être considérées suspectes [27] en adoptant un algorithme similaire au *PageRank*. La similarité des sites de *phishing*, qui imitent d'autres sites légitimes, peut servir d'indice pour leur détection. On trouve des approches qui se basent sur le contenu des pages [23], sur la structure en exploitant l'arbre DOM des pages HTML [24], ou sur l'apparence visuelle des pages [25, 26]. Les sites ainsi détectés peuvent être alors signalés à des listes de blocage, comme SafeBrowsing de Google, qui sont alors déréférencés des moteurs de recherches, et sont marqués comme dangereux par les principaux navigateurs internet. Toutefois, procéder par de telles listes requière de détecter rapidement les nouveaux sites avant d'être visités par de potentielles victimes, ce qui est une tâche difficile au vu de la grande quantité de sites créés. Une autre difficulté vient du fait qu'une fois bloqués, ces sites peuvent très facilement changer de nom de domaine, et ainsi ne plus être bloqués, jusqu'à être détecté de nouveau.

Les précédents points décrivent ce qui est étudié en termes de contre-mesures du *phishing*, que ce soit la prévention chez les utilisateurs, les filtres anti-pourriels, ou la détection de sites de *phishing*. D'autres recherches, moins nombreuses, se sont focalisées sur l'analyse des kits de *phishing*. Comprendre leur fonctionnement permet de mieux appréhender l'écosystème autour du *phishing* et peut servir à renforcer les contre-mesures. Des kits gratuits, disponibles sur internet, ont été analysés par Cova *et al.* [6]. Les auteurs ont exécuté les kits dans des

environnements sécurisés, afin d'observer les courriels envoyés qui transmettent les résultats à l'attaquant. Dans *PhishEye* [14], les auteurs ont mis en place un *honeypot* pour que des attaquants viennent infecter leur serveur web et y placer leurs kits. Ils ont ainsi pu observer le comportement de ces kits lorsqu'ils sont déployés sur des serveurs compromis. Outre l'exécution des kits dans des environnements contrôlés, d'autres auteurs ont décidé d'analyser les fichiers contenus dans les kits. Imperva [9] a réalisé une analyse statistique sur plus de 1 000 kits, en extrayant des informations pertinentes du code source, comme les signatures des auteurs, les sujets des courriels envoyés, etc. Les kits d'un même groupe de pirates ont été étudiés par Mccalley *et al.* [7], révélant ainsi l'existence de portes dérobées dans leur code source qui transmettent le résultat des attaques à des tiers. Thomas *et al.* [12] ont analysé les informations de connexion dérobées (identifiant, mot de passe, etc.), en partie issues du *phishing*, et les risques encourus pour les utilisateurs Les fichiers `.htaccess` des kits, permettant notamment de filtrer certaines adresses IP, ont été l'objet d'une étude d'Oest *et al.* [10], qui décrivent les moyens auxquels les auteurs de kits ont recours pour échapper à la détection de leurs sites.

Aucun article concernant l'analyse de similarité du code source des kits de *phishing*, côté serveur, n'est présent dans la littérature, au meilleur de notre connaissance. Coté client, CrawlPhish [15] repose en partie sur de l'analyse de similarité. Les auteurs catégorisent les différentes techniques de *cloaking* observées dans les fichiers JS. Pour cela, ils comparent la similarité du code source des fichiers JS à ceux d'autres sites connus, mais ils ont également recours à la similarité visuelle des sites, tout comme certaines techniques de détection de site de *phishing* présentées plus haut.

2.2 Détection de clones

Plusieurs approches existent pour la détection de clones. L'approche textuelle considère le code source comme étant un document texte. Il est alors possible de comparer ligne par ligne deux code sources, d'effectuer du *string matching* [44], sans avoir besoin d'identifier le langage du code source. Ces méthodes présentent l'avantage d'être simples et rapides, au détriment d'une précision et d'un rappel moindre.

L'approche basée sur les jetons a été proposée en premier par Kamiya *et al.* [45] avec CCFinder. D'autres auteurs, comme Basit, Roy *et al.* [46, 47], ont continué de développer cette approche, qui est désormais une famille d'outils de détection de clones. Le principe est de séparer le code source en jetons via une analyse lexicale. On peut alors vérifier rapidement si deux séquences de jetons sont semblables en utilisant par exemple des arbres des suffixes [48]. Ce niveau de granularité permet d'être plus précis qu'une simple analyse textuelle,

notamment pour détecter les clones paramétriques : par exemple, si la valeur d'une constante change, le type de jeton demeure lui identique, ce qui n'est pas identifiable sans une analyse lexicale.

L'approche basée sur les métriques de code a été introduite par Mayrand *et al.* [49], et améliorée dans [50, 51] par Merlo *et al.* Dans cette approche, des métriques sont extraites du code source et représentées sous forme de vecteur. Ces métriques peuvent utiliser plusieurs informations syntaxiques extraites de l'AST : nombre d'instructions, d'instructions conditionnelles, de boucles, d'appels à des fonctions, etc. Ces vecteurs sont alors utilisés avec des critères de similarité, comme la distance euclidienne, la distance de Manhattan (norme 1), la similarité cosinus, etc.

Pour les clones de types 1 et 2, une méthode basée sur l'AST, a été proposée par Baxter *et al.* [52]. La représentation du code source par arbre est plus riche que celle de l'analyse lexicale, mais les outils basés sur l'AST ont comme inconvénient majeur des temps d'exécutions plus importants que les méthodes basées sur les jetons ou les métriques. Cela est d'autant plus impactant lorsqu'on travaille sur de large volume de codes source.

NiCAD [47], introduit par Cordy et Roy, est un outil de détection de clones basé sur de nombreux algorithmes de préfiltrage, avec la dernière étape de filtrage qui utilise la longueur de la plus longue sous-chaîne commune (LCS). La LCS est le dual de la distance de Levenshtein et les deux peuvent être vus comme grossièrement équivalent puisque la distance de Levenshtein permet de minorer la longueur de la LCS. L'outil CLAN [49] suggère également que l'utilisation de LCS permet d'affiner les résultats pour obtenir une meilleure précision. La différence entre NiCAD et CLAN est le nombre et la complexité de chaque étape avant le filtrage par LCS : NiCAD utilise un certain nombre d'étapes de prétraitement, lexical et syntaxique, tandis que CLAN utilise une seule étape de calcul de métriques.

La distance de Levenshtein a également été suggérée comme référence pour la détection de clones par d'autres auteurs, notamment par Tiarks *et al.* [53] qui ont proposé un *benchmark* de clones, de taille limitée. Un ensemble exhaustif de toutes les paires satisfaisant une distance de Levenshtein inférieure à un seuil fixé a été proposé par Lavoie et al. [54]. Sans pour autant être parfaite, il est raisonnable de considérer la distance de Levenshtein comme un choix de référence pour produire des bons résultats dans la détection de clones. Ainsi, l'utilisation d'une approche produisant des résultats similaires à ceux de cette distance est justifié dans ce contexte.

Lavoie *et al.* [55] ont montré que la distance de Manhattan, plus rapide à calculer que celle de Levenshtein, fournit une bonne approximation de cette dernière. Dans cet article, les auteurs ont recours à des arbres métriques [56] afin d'accélérer le parcours des fragments

similaires. Également, ils ont observé que l'utilisation de n-grammes de jetons n'améliorent pas de manière significative la détection de clones, par rapport aux unigrammes.

La détection de clone, associée au graphe d'évolution d'une application quand il est connu, permet d'analyser le problème de l'origine des clones et de leur évolution [32], connu sous le nom de *clone genealogy*. Dans cet article, les auteurs comparent comment évoluent les clones au fur et à mesure des différentes versions d'un logiciel. Ils mettent alors en évidence l'existence de clones à faible durée de vie et à longue durée de vie. D'autres recherches ont été poursuivies dans ce domaine. Par exemple, les propagations tardives où des clones évoluent de manière indépendante pendant une certaine durée avant d'être resynchronisés, ont été étudiées dans [34]. Saha *et al.* ont développé gCad [35], un outil d'extraction de généalogie de clone. Leur outil permet de suivre l'évolution d'un ensemble de clones de type 3 au fil des versions d'un logiciel, de manière incrémentale afin de s'adapter au volume potentiellement important des nombreuses versions.

Si le lecteur désire avoir des informations détaillées sur les différentes techniques de détection de clone présentes dans la littérature, il est invité à aller consulter les revues de Rattan *et al.* [57] et de Roy *et al.* [58].

Bien que la détection de clones se fait généralement sur le code source, l'approche a également été étendue aux codes binaires. Sæbjørnsen *et al.* [59] font partie des premiers à avoir développé cette approche, en analysant notamment les binaires systèmes de Windows. Ce genre de techniques s'avère nécessaire lorsque le code source n'est pas disponible, comme cela est souvent le cas avec les logiciels malicieux.

2.3 Analyse de logiciel malicieux

L'analyse des logiciels malicieux (*malwares*) a été principalement fait sur des binaires, le code source n'étant généralement pas accessible. Par conséquence, l'étude de similarité de propagation des clones a été fait en majorité sur les binaires.

Khoo *et al.* [60] ont proposé Rendezvous, un moteur de recherche pour la similarité de codes binaires. Leur outil permet de chercher efficacement si du code binaire provient d'un des logiciels connus dans la base de données, indépendamment du compilateur ou des options de compilation utilisés. Les auteurs l'ont évalué sur des logiciels *open-source*, mais la même approche peut être utilisée sur du logiciel malveillant. Pour identifier la similarité de certaines fonctions dans le code binaire, d'autres auteurs comme Egele *et al.* [61] ont décidé d'utiliser uniquement l'approche dynamique, en exécutant les logiciels et mesurant les effets des appels aux fonctions. Cela a pour avantager d'ignorer la variabilité introduite par les différentes

options de compilations, qui est un obstacle majeur lors de l'analyse statique. Appliqué au logiciel malveillant, cela nécessite néanmoins une vigilance accrue pour isoler l'exécution du reste du système.

Ming *et al.* proposent un moyen de calculer la différence sémantique entre binaires de *malwares* [62, 63]. En identifiant les blocs basiques dans le code binaire qui effectuent des opérations similaires, ils sont en mesure de rapidement identifier les logiciels malicieux qui auront des comportements semblables lors de l'exécution, ce qui peut alors aider à l'inférence de généalogies.

Concernant l'analyse de binaires des *malwares*, Lindorfer *et al.* [64] donnent des informations sur l'industrie derrière le logiciel malveillant, en autorisant les *malwares* à se mettre à jour. Ils comparent ainsi les différences entre versions du point de vue du code binaire, mais également en exécutant les logiciels dans des environnements sécurisés. Également, le code binaire relatif au réseau est modélisé sous forme de graphe par Wang *et al.* [65], dans le contexte de l'analyse de similarité entre *malwares*. L'analyse hybride, qui combine analyse statique et dynamique, a été employée avec des techniques d'apprentissage machine pour détecter les logiciels malveillants par Venkatraman *et al.* [66]. Dans cet article, les auteurs utilisent des techniques de similarité basées sur les images pour identifier des comportements inhabituels, propres aux *malwares*.

Pour ce qui est de la généalogie, Jang *et al.* [67] présentent une approche pour inférer automatique la généalogie d'un logiciel à partir des binaires. Cette approche a été testée sur des logiciels *open-source* et sur un certain nombre de logiciels malicieux dont l'évolution est connue.

La provenance de logiciels malveillants a également été investigué par Dumitras *et al.* [68], qui récapitulent les difficultés inhérentes à ce genre de problèmes, et proposent l'utilisation d'arbre phylogénétique se basant sur des caractéristiques identifiées par des analyses statique et dynamique. D'autres approches phylogénétiques pour la rétro-ingénierie ont été expérimentées par Khoo *et al.* [69] pour identifier les familles de *malwares*.

L'analyse de l'évolution de logiciel malveillant a été étudié par Gupta *et al.* [70] de manière empirique. Les auteurs ont recours à l'analyse la diversité des familles de *malwares* à partir de métadonnées fournies par McAffee. Du point de vue structurel, Darmetko *et al.* [71] ont étudié la construction de généalogie de familles de logiciel malveillants en observant quelles sont les modifications apportées à la structure du code binaire. Pour une même famille, ils utilisent la similarité des chaînes de caractères présentes dans le binaire pour calculer des distances entre les différentes versions, et ainsi afficher une possible généalogie sur une ligne droite avec le *Multidimensional Scaling* (MDS).

L'étude du *malware* s'intéresse également aux binaires qui ciblent des plateformes mobiles populaires. Canfora, Cimitile *et al.* [72, 73] se sont concentrés sur l'étude de logiciels malicieux sur Android. Sans pour autant construire une généalogie qui rend compte de l'évolution temporelle de ces logiciels, ils ont conçu un modèle phylogénétique permettant ainsi de les regrouper en familles de *malwares*. Également, Soto-Valero *et al.* [74] ont étudié la dissimilarité des malwares présents sur la plateforme Android, en utilisant les données d'étiquetage provenant d'outils identifiant les familles de logiciels malveillants. Ils ont mis en place des algorithmes de *clustering* soulignant la diversité de plus en plus importante de ces logiciels dans l'écosystème d'Android.

CHAPITRE 3 APPROCHE

L'approche globale utilisée dans ce travail est résumée dans la figure 3.1. Une généalogie est générée pour chaque langage, à savoir PHP, JS, et HTML. Les étapes suivies sont les suivantes : le pré-traitement (section 3.1), le parsage et fragmentation des codes sources (section 3.2), et l'analyse des fragments (section 3.3) pour aboutir à une généalogie mono-langage, dont les kits voisins sont comparés un à un. Une fois les trois langages analysés, leurs résultats sont combinés (section 3.4) pour obtenir une généalogie multi-langage.

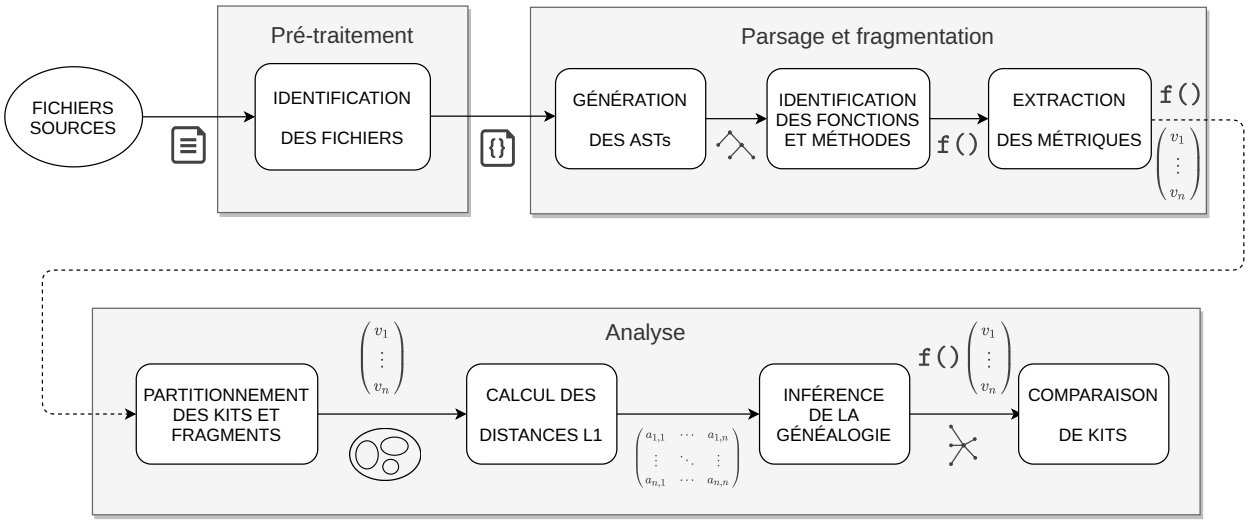


FIGURE 3.1 Schéma récapitulant les étapes de l'approche mono-langage

3.1 Pré-traitement

La première étape, le prétraitement, consiste à normaliser le code source des kits : conversion de l'encodage en UTF-8, utilisation des conventions Unix de retour à la ligne. Une fois ceci fait, les fichiers de chaque langage sont identifiés. On a alors une liste de fichiers sources pour PHP, HTML et JS. Pour HTML et JS, on récupère tous les fichiers ayant pour extension de nom de fichier respectivement `.html` ou `.htm`, et `.js`. Pour PHP, on recherche les fichiers textes ayant une balise ouvrante de PHP (par exemple `<?php`). Le code source en PHP est généralement écrit dans des fichiers avec une extension en `.php`, mais on peut également en trouver dans des fichiers avec d'autres extensions. Si un attaquant souhaite cacher ses fichiers PHP, il peut modifier la configuration du serveur web pour accepter d'autres extensions. Il peut également écrire son code PHP dans un fichier quelconque, et exécuter le contenu du

fichier avec la fonction `eval` de PHP, depuis un fichier ayant l'extension canonique de PHP. Pour ces raisons, on ne peut pas se reposer uniquement sur l'extension de nom de fichier pour PHP.

3.2 Parsage et fragmentation

La seconde étape correspond au parsage et à la fragmentation des fichiers sources. On commence par la sous-étape du parsage des fichiers. Chaque fichier source est passé en entrée du parseur adéquat, qui produit l'AST en sortie, dans un format tabulaire, propre au laboratoire. Pour les fichiers PHP, les morceaux de code statique HTML sont exportés, et ajoutés à la liste des fichiers HTML à parser. De manière similaire, les scripts JS contenus à l'intérieur de balises `<script>` en HTML sont exportés et ajoutés à la liste des fichiers JS à parser.

Pour PHP, un parseur utilisant JavaCC [1] est utilisé. Il a été conçu à partir d'une grammaire open-source [2], et extensivement modifiée par différents élèves du laboratoire d'Ettore Merlo, pour pouvoir supporter des versions ultérieures de PHP (jusqu'à 7.x), et des analyses de *data-flow* et *control-flow*.

Pour JS, le parser `acorn` [3] (écrit en JS) est utilisé. Puisque le code JS peut être entrecoupé de morceau de PHP, il peut devenir syntaxiquement invalide. Pour cela, il est utile d'avoir un parseur qui soit robuste aux erreurs de syntaxe, comme c'est le cas de la version tolérante aux erreurs de `acorn` : `acorn-loose`. Si un fichier JS n'arrive pas à être parsé par `acorn`, il est passé à `acorn-loose`, qui peut être moins précis, mais plus flexible. Dans ce travail, ce sont les jetons qui sont importants, les ASTs étant uniquement utilisés pour détecter les fonctions et méthodes. Ainsi, un AST moins précis n'est pas pénalisant pour la méthode employée, tandis que les jetons, issus de l'analyse lexicale, ont moins de chance d'être erronés.

Pour HTML, un parseur tolérant aux erreurs a également été utilisé : `htmlparser2` [4]. Pour les parseurs JS et HTML, un programme a été écrit pour traduire les ASTs dans le format tabulaire utilisé dans le laboratoire.

Ensuite, lorsque les ASTs des fichiers ont été générés, vient la sous-étape de la fragmentation. Les fonctions et méthodes sont identifiées en parcourant les ASTs. À chaque fois qu'un tel nœud est trouvé, un fragment est créé, qui contient le sous AST et les jetons associés au fragment. De plus, un fragment est crée pour chaque fichier au niveau de la racine de l'AST, en enlevant le contenu des fragments créés auparavant, pour prendre en compte le code des

1. `https://javacc.github.io/javacc/`
2. `https://raw.githubusercontent.com/javacc/javacc/master/grammars/PHP.jj`
3. `https://github.com/acornjs/acorn/`
4. `https://github.com/fb55/htmlparser2/`

scripts contenus en dehors des fonctions ou méthodes. HTML fait figure de cas particulier, puisqu'il n'y a pas de notions de fonctions ou méthodes. Par conséquent, un fichier HTML correspond à un unique fragment. Enfin, les métriques de chaque fragment sont calculées : chaque fragment se voit associer un vecteur de fréquences des types de jetons. Les tailles de ces vecteurs sont de 162 pour PHP, 76 pour JS, et 10 pour HTML. Au lieu de calculer la fréquence sur les jetons (unigrammes), il est théoriquement possible de calculer la fréquence sur des n-grammes. Cependant, la littérature [55] suggère que l'amélioration apportée par les n-grammes n'est pas suffisante au regard du coût de calcul supplémentaire associé.

Les statistiques sur l'étape de parsage sont données dans la section 4.1

3.3 Analyse mono-langage

Finalement, la troisième étape consiste en l'analyse des fragments et des kits, pour un langage donné. Pour chaque kit, on a une collection de fragments, qui contiennent chacun leurs jetons (à la fois leurs types et leurs images), et leur vecteur de métriques (fréquence des types de jetons). Seuls les kits contenant au moins un fragment du langage considéré sont pris en compte. À partir de ces informations, on désire inférer une généalogie plausible pour les kits.

3.3.1 Partitionnement des fragments

Pour partitionner les fragments, il faut d'abord définir les relations d'équivalence possible entre deux fragments f_1 et f_2 :

- f_1 est **identique** à f_2 si et seulement si f_1 et f_2 sont des clones de type 1 (ils ont la même séquence d'images de jetons) ;
- f_1 est **paramétrique** à f_2 si et seulement si f_1 et f_2 sont des clones de type 2 (ils ont la même séquence de types de jetons).

Pour rappel, les définitions et exemples des types de clone sont donnés dans la section 1.1.2. La relation d'égalité entre deux séquences étant une relation d'équivalence, les relations présentées ici sont bien des relations d'équivalences (réflexives, symétriques et transitives). Les définitions des types de clones sont données dans la section 1.1.2. Il est à noter que deux fragments identiques sont également paramétriques, puisque l'ensemble des clones de type 1 est un sous-ensemble de l'ensemble des clones de type 2. Par la suite, on emploie le terme paramétrique au sens strict, en excluant identique, sauf si explicitement mentionné.

L'algorithme pour le calcul de ces classes est donné à la figure 3.2. Il attend comme paramètre un ensemble d'objets, et une fonction de *hash*. Pour calculer ces classes d'équivalences, on ne

```
1: fonction calculClassesEq(objets, hash)
2:     partition ← {} // dictionnaire qui associe une valeur aux objets ayant ce hash
3:     pour tout x ∈ objets faire
4:         si hash(x) ∉ clés(partition) alors
5:             // pas encore de classe d'équivalence avec ce hash comme valeur
6:             partition[hash(x)] ← [] // liste vide
7:         fin si
8:         ajouter x à partition[hash(x)]
9:     fin pour
10:    renvoie partition
11: fin fonction
```

FIGURE 3.2 Algorithme du calcul des classes d'équivalence

compare pas directement les valeurs des objets (listes de jetons par exemple), ce qui serait coûteux en calculs vue la quantité de données à traiter. À la place, on utilise le *hash* des objets, pour vérifier si des objets ont des valeurs différentes ou non.

On commence ainsi par partitionner les fragments, en classes de fragments identiques, en utilisant la relation d'équivalence définie plus haut. L'algorithme (figure 3.2) est donc appelé sur les fragments, en calculant le *hash* de la liste des images de jetons.

Une fois les classes de fragments identiques calculées, c'est au tour des classes de fragments paramétriques. Le même principe est appliqué que précédemment, en remplaçant les images de jetons par les types de jetons. Afin d'accélérer le traitement, on n'applique pas le processus à tous les fragments, mais seulement à un représentant par classe de fragments identiques, puisque des fragments identiques sont nécessairement paramétriques (au sens large).

3.3.2 Composition des kits

Désormais, on considère les kits en entier, et non les fragments individuellement. Un kit est constitué d'un ensemble de fragments, chaque fragment appartenant à une classe d'équivalence. Pour les relations d'équivalence, on ne considère pas directement les fragments, mais plutôt les classes d'équivalences auxquelles ils appartiennent. Deux choix sont possibles pour ces classes d'équivalences.

Premièrement, on peut considérer un kit comme un ensemble de classes d'équivalence, et donc ignorer les fragments dupliqués au sein d'un kit. Les fragments appartenant à une même classe d'équivalence ne sont donc comptés qu'une seule fois.

Deuxièmement, on peut considérer un kit comme un multi-ensemble (*bag*) de classes d'équivalence, et donc prendre en compte la multiplicité des fragments dupliqués au sein d'un kit.

Les fragments appartenant à une même classe d'équivalence sont donc tous comptés. Cette dernière option permet de prendre en compte le nombre de clones internes au sein d'un kit, mais également de prendre en compte les relations $n \times m$ de clones externes entre deux kits.

Pour illustrer la deuxième définition, si un même fragment est dupliqué identiquement un nombre n de fois dans un kit (sans aucun autre fragment), mais un nombre m de fois dans un autre kit (sans aucun autre fragment), avec $n \neq m$, les deux kits seront considérés différents.

Au contraire, avec la première définition, les deux kits seront considérés identiques puisque la répétition du fragment est ignorée, seul le fait que les deux kits soient composés de la même classe de fragments importe.

Dans notre cas, la définition de kit retenue pour les relations d'équivalence est la première. Pour l'étude de similarité, l'information jugée pertinente est le fait que deux kits partagent certains fragments identiques ou paramétriques, peu importe le nombre de fois qu'ils ont été dupliqués. Un kit est donc considéré comme un ensemble de classes d'équivalence, identique ou paramétrique selon la relation d'équivalence choisie. Par la suite (section 3.3.6 et les suivantes), la définition retenue pour les kits est celles d'ensembles de classes d'équivalence de fragments paramétriques.

Dans le cas où un kit est « inclus » dans un autre, dans le sens où les fragments d'un kit k_1 forment un sous-ensemble (strict) de k_2, les deux kits ne sont considérés ni identiques, ni paramétriques. L'inclusion étant une relation d'ordre, cette relation ne peut pas être utilisée pour partitionner les kits. En revanche, cela peut être découvert lors de la comparaison de kits (section 3.3.9), en étudiant le taux de fragments identiques (ou paramétriques) de k_1 par rapport à k_2 (voir section 4.5).

3.3.3 Partitionnement des kits

À partir des considérations de la section 3.3.2, on définit les relations d'équivalences pour deux kits k_1 et k_2 ci-dessous :

— k_1 est **identique** à k_2 si et seulement si les ensembles des classes d'équivalence de fragments identiques de k_1 et de k_2 sont égaux (chaque fragment de k_1 est identique à au moins un fragment de k_2, et vice versa) ;
— k_1 est **paramétrique** à k_2 si et seulement si les ensembles des classes d'équivalence de fragments paramétriques de k_1 et de k_2 sont égaux (chaque fragment de k_1 est paramétrique à au moins un fragment de k_2, et vice versa).

La relation d'égalité entre deux ensembles étant une relation d'équivalence, ces relations sont donc bien des relations d'équivalences.

Le procédé du calcul des classes d'équivalence des kits est le même que celui des fragments, mais en remplaçant l'ensemble des images (respectivement types) des jetons d'un fragment par l'ensemble des classes d'équivalences de fragments identiques (respectivement paramétriques) d'un kit. De même que pour les fragments, le calcul des kits paramétriques ne se fait pas sur tous les kits, mais seulement sur les représentants de chaque classe d'équivalence de kits identiques.

À partir de cette sous-étape, on ne considère qu'un seul kit par classe d'équivalence de kits paramétriques. Cela signifie donc qu'on ne traite que des kits qui sont paramétriquement distincts. Cela permet alors d'enlever des doublons pour les sous-étapes suivantes.

3.3.4 Calcul de distance entre fragments

Ensuite, il s'agit de l'étape du calcul des distances. Dans un premier temps, on considère uniquement la distance entre fragments. Conceptuellement, on désire définir une distance entre deux fragments, qui mesure leur différence. Ainsi, une distance faible entre deux fragments montre que les deux fragments sont similaires, tandis qu'une distance élevée indique que, au contraire, ils sont plus éloignés d'un point de vue syntaxique.

On décide d'utiliser la distance de Manhattan (ou norme 1), définie dans l'équation 3.1, entre les vecteurs de fréquence des types de jetons des fragments. Elle constitue une bonne approximation de la distance de Levenshtein pour la détection de clones [55].

La distance de Manhattan, l_1, entre deux vecteurs u et v de taille k est définie comme :

$$l_1(u, v) = \|u - v\|_1 = \sum_{i=1}^{k} |u_i - v_i| \tag{3.1}$$

Si on considère que le vecteur de métriques d'un fragment représente le multi-ensemble des types de jetons, la distance entre les deux vecteurs de métriques correspond alors à la différence symétrique des multi-ensembles des types de jeton des fragments. Ainsi, la distance s'interprète comme le nombre de types de jetons différents entre deux fragments.

3.3.5 Exemple de calcul de distance entre fragments

Illustrons la manière présentée dans la section 3.3.4 de calculer les distances, mais en considérant des fragments au lieu de kits, pour des raisons de simplicité. Supposons que nous avons parsé des fichiers sources PHP, et identifié trois versions de la même fonction. Les codes source de ces trois versions sont donnés dans les figures 3.3, 3.4 et 3.5. Les fonctions `user_pass_ok` (suivi d'un numéro) vérifient si l'identifiant et le mot de passe d'un utilisateur correspondent.

La première version (figure 3.3) récupère simplement les données de l'utilisateur, et vérifie que le *hash* du mot de passe correspond à celui enregistré. La seconde version (figure 3.4) introduit une variable globale, qui sert de cache. Elle ajoute aussi une instruction `if-else` pour vérifier si les données de l'utilisateur sont dans le cache, et ensuite les récupère depuis la base de donnée si ce n'est pas le cas. Finalement, la troisième version (figure 3.5) ajoute un nouveau paramètre pour passer outre le cache si ce paramètre vaut `true`.

```
1 function user_pass_ok_1($user_login,$user_pass) {
2  $userdata = get_userdatabylogin($user_login);
3
4  return (md5($user_pass) == $userdata->user_pass);
5 }
```

FIGURE 3.3 Exemple – Code PHP du fragment 1

```
1  function user_pass_ok_2($user_login,$user_pass) {
2   global $cache_userdata;
3
4   if ( empty($cache_userdata[$user_login]) ) {
5    $userdata = get_userdatabylogin($user_login);
6   } else {
7    $userdata = $cache_userdata[$user_login];
8   }
9   return (md5($user_pass) == $userdata->user_pass);
10 }
```

FIGURE 3.4 Exemple – Code PHP du fragment 2

```
1  function user_pass_ok_3($user_login,$user_pass,$no_cache) {
2   global $cache_userdata;
3
4   if ( empty($cache_userdata[$user_login]) || $no_cache) {
5    $userdata = get_userdatabylogin($user_login);
6   } else {
7    $userdata = $cache_userdata[$user_login];
8   }
9   return (md5($user_pass) == $userdata->user_pass);
10 }
```

FIGURE 3.5 Exemple – Code PHP du fragment 2

Une fois que les fichiers ont été parsés, on réalise le calcul des métriques pour chaque fragment. Le parseur PHP utilisé reconnaît 162 types de jetons (par exemple les mot clés, opérateurs,

littéraux, identifiants, etc.). Même si dans l'analyse de PHP les 162 types de jetons sont traités individuellement, dans le cadre de cet exemple, on se limite à seulement quelques types de jetons, en fusionnant plusieurs types dans une même catégorie. Ici, les métriques utilisées sont les mots clés `function`, `if`, `else`, `global`, `return`, les identifiants (`id`), les opérateurs (accolades, crochets, virgules, points, point-virgules, symbole dollar, signe égal, etc.), et les opérateurs logiques & relationnels (par exemple `==`, `||`).

Dans la figure 3.3, on peut compter chaque mot clé `function` et `return` une seule fois, alors qu'il y a 10 identifiants (1 pour le nom de la fonction, 2 `user_login`, 3 `user_pass`, 2 `userdata` et 2 noms de fonctions appelées). Il y a également 21 opérateurs (8 parenthèses, 2 accolades, 6 dollars, 1 virgule, 2 points-virgules, 1 égal et 1 flèche), et 1 opérateur relationnel (le double égal). Dans la figure 3.4, l'ajout de l'instruction `if-else` incrémente l'occurrence des mots clés `if`, `else` et `global` de 1. Cela ajoute aussi 7 identifiants, et 21 opérateurs. Dans la figure 3.5, l'ajout de la condition dans la clause du `if` augmente le nombre d'identifiant par 2 (la nouvelle variable est utilisée 2 fois), le nombre d'opérateurs par 3 (1 virgule et 2 dollars), et le nombre d'opérateurs logiques et relationnels par 1 (l'opérateur `||`). Les métriques pour chaque fonction sont reportées dans la figure 3.6, dans leur ordre respectif.

$$\begin{pmatrix} \texttt{function} \\ \texttt{if} \\ \texttt{else} \\ \texttt{return} \\ \texttt{global} \\ \texttt{id} \\ \texttt{operator} \\ \texttt{log\&rel op} \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 10 \\ 21 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 17 \\ 42 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 19 \\ 45 \\ 2 \end{pmatrix}$$

FIGURE 3.6 Métriques des 3 versions de la fonction `user_pass_ok`

Pour déterminer la distance entre, par exemple, la première version et la seconde version, on calcule la distance de Manhattan entre leurs vecteurs de métriques :

$$\begin{aligned} l_1(u,v) = \|u-v\|_1 &= \sum_{i=1}^{k} |u_i - v_i| \\ &= 0+1+1+0+1+7+21+0 \\ &= 31 \end{aligned}$$

Cela signifie qu'il y a une différence de 31 types de jetons entre la première et la seconde

version de la fonction. On peut également faire le même calcul pour chaque paire de version, ce qui nous donne la matrice de distances décrite dans la table 3.1. Cette matrice est donnée à but indicatif, en pratique on ne calcule pas les distances entre chaque paire de fragments, qui sont bien trop nombreux.

La distance entre la seconde version et la troisième est plus petite que la distance entre la première version et n'importe quelle autre version. En effet, les deux dernières versions semblent très similaires, puisque leur différence est l'existence ou non du paramètre `no_cache` et son utilisation dans la clause du `if`. Cependant, la première version n'a ni une instruction `if-else`, ni une variable globale, ce qui la rend plus différente des autres. La distance entre la première et la troisième version est aussi plus grande que la distance entre la première et la seconde.

Exemple n°	1	2	3
1	0	31	37
2	31	0	6
3	37	6	0

TABLEAU 3.1 Matrice de distance pour les fonctions de l'exemple

3.3.6 Calcul de la matrice de distances des kits

Une fois la distance entre fragments définie, on considère la distance entre kits, pour toutes les paires de kits. Au préalable, un vecteur de métriques est associé à chaque kit. La définition du vecteur de métrique v_k d'un kit k est donnée dans l'équation 3.2, où $\text{fpd}(k)$ est l'ensemble des fragments paramétriquement distincts du kit k, et $\text{vecteurMétrique}(f)$ est le vecteur de fréquences de types de jetons du fragment f. Il s'agit de la somme des vecteurs de fréquences de types de jetons des fragments du kit, en ne prenant que les fragments paramétriquement distincts (on ignore donc les clones paramétriques, pour les raisons discutées dans la section 3.3.2). Cela permet de calculer rapidement la distance entre deux kits, ce qui est nécessaire au vu du nombre de paires de kits à traiter.

$$v_k = \sum_{f \in \text{fpd}(k)} \text{vecteurMétrique}(f) \tag{3.2}$$

Une fois ces vecteurs calculés pour les kits, on calcule la matrice de distances sur toutes les paires de kits. Tout comme les fragments (section 3.3.4), la distance choisie est la distance de Manhattan (voir équation 3.1) entre leurs vecteurs de kits (au lieu des vecteurs de métriques

des fragments). Une distance étant une fonction symétrique, la matrice obtenue est également symétrique. Par conséquent, on a besoin de calculer uniquement la moitié de la matrice. L'algorithme pour le calcul de la matrice, prenant en entrée les vecteurs de métriques des kits, est donné figure 3.7, avec `kitVecteurs` la liste des vecteurs de métriques des kits, calculés comme indiqué par l'équation 3.2.

```
1: fonction CALCULMATRICEDISTANCE(kitVecteurs)
2:     n ← #kitVecteurs
3:     M ← matrice nulle de taille n × n
4:     // distance symétrique, seule la moitié de la matrice doit être calculée
5:     pour tout i ∈ [0, n − 1] faire
6:         v_i ← kitVecteurs[i]
7:         M[i, i] ← 0
8:         pour tout j ∈ [i + 1, n − 1] faire
9:             v_j ← kitVecteurs[j]
10:            M[i, j] ← ‖v_i − v_j‖_1
11:            M[j, i] ← M[i, j] // symétrie de la matrice
12:        fin pour
13:    fin pour
14:    renvoie M
15: fin fonction
```

FIGURE 3.7 Algorithme du calcul de la matrice de distance des kits

Tout comme pour les fragments (voir section 3.3.6), la distance de Manhattan entre les vecteurs de deux kits peut s'interpréter comme étant la différence symétrique des multi-ensembles des types de jetons des deux kits.

Bien que cette étape ignore la granularité introduite par les fragments, cette granularité est prise en compte pour la comparaison des kits (voir section 3.3.9). On cherche ici à déterminer la similarité de toutes les paires de kits, ce qui nécessite un algorithme rapide. La comparaison de kits, plus lente, est uniquement effectuée entre les kits des arêtes de la généalogie.

Ne considérer que des kits paramétriquement distincts, comme indiqué à la fin de la section 3.3.3, permet de réduire la taille de la matrice de distances en enlevant des valeurs récurrentes. En effet, avec la définition de vecteurs de métriques de kit proposée ci-dessus, deux kits paramétriques ont les mêmes vecteurs de métriques, puisqu'on compte la fréquence des types de jeton des fragments paramétriquement distincts. Par conséquence, la distance entre deux kits reste inchangée même si on remplace un des kits par un autre kit paramétrique.

Calculer la distance de cette manière présente plusieurs avantages. Tout d'abord, faire la

somme des vecteurs est rapide, ce qui est nécessaire pour faire le calcul de distance entre toutes les paires considérées. Également, le fait de fusionner des fragments (par exemple concaténer le contenu de deux fonctions en une seule) ou de séparer un fragment en plusieurs n'impacte pas la distance, outre les jetons dus à l'en-tête des fonctions.

Cependant, cette distance présente des limitations. Elle ignore la structure du code en fragments, ce qui l'amène à sous-estimer le nombre de jetons différents entre deux kits. Par exemple, deux mot-clés `if` présents dans des fonctions non similaires seront quand même considérés identiques et ne contribueront pas à augmenter la distance. Dans le cadre d'une attaque antagoniste (*adversarial*), cela peut-être exploité pour faire croire que deux kits différents sont similaires : il suffit d'ajouter un fragment qui contient le bon nombre de jetons pour chaque type, pour arriver à une distance faible. Également, le fait de permuter deux instructions entre deux fragments ne change pas le vecteur de métriques de kit peut être exploité.

3.3.7 Exemple de calcul de distance entre kits

On illustre le calcul des distances entre kits avec un exemple. On considère deux kits composés chacun d'un fichier PHP, dont le code source est donné dans les figures 3.9 pour le premier kit et 3.10 pour le second. Le premier kit est composé de trois fragments (numérotés de 1 à 3), et le second de quatre (de 4 à 7).

On procède de la même manière que dans l'exemple sur les fragments de la section 3.3.5 : chaque fragment se voit attribuer un vecteur de métriques, simplifié dans le cadre de cet exemple. On peut remarque que le fragment 1, du premier kit (figure 3.9), possède 1 mot-clé `function`, 1 `for`, 1 `return`, 11 identifiants (1 `factorielle`, 2 `n`, 4 `res`, et 4 `i`), 28 opérateurs (10 dollars, 5 point-virgules, 4 parenthèses, 4 accolades, 3 signes égal, 1 symbole `++` et 1 `*`), et un opérateur relationnel (`<=`). De même, on calcule les vecteurs des autres fragments.

Au sein d'un même kit, tous les fragments sont paramétriquement distincts, on considère donc tous les fragments du kit. On passe ensuite à l'étape du calcul des vecteurs de métriques des kits, où l'on fait la somme de tous les fragments du même kit. Le résultat est donné figure 3.8.

Si on calcule la distance de Manhattan entre les deux vecteurs de kits obtenus, on a :

$$l_1(u, v) = 1 + 0 + 1 + 0 + 0 + 1 + 3 + 6 + 1 + 1 + 1 = 15$$

La distance entre les deux kits est donc de 15 jetons, ce qui correspond à la valeur d'une case de la matrice. En pratique, ce calcul est effectué pour toutes les paires de kits du jeu de

$$\begin{pmatrix}\texttt{function}\\ \texttt{if}\\ \texttt{elseif}\\ \texttt{else}\\ \texttt{for}\\ \texttt{return}\\ \texttt{id}\\ \texttt{operator}\\ \texttt{rel op}\\ \texttt{string}\\ \texttt{echo}\end{pmatrix} \rightarrow \begin{pmatrix}1\\0\\0\\0\\1\\1\\11\\28\\1\\0\\0\end{pmatrix} + \begin{pmatrix}1\\0\\0\\0\\0\\0\\1\\5\\0\\1\\1\end{pmatrix} + \begin{pmatrix}1\\1\\1\\1\\0\\0\\7\\18\\2\\3\\3\end{pmatrix} = \begin{pmatrix}3\\1\\1\\1\\1\\1\\19\\51\\3\\4\\4\end{pmatrix} ; \begin{pmatrix}1\\0\\0\\0\\1\\1\\11\\28\\1\\0\\0\end{pmatrix} + \begin{pmatrix}1\\0\\0\\0\\0\\0\\1\\5\\0\\1\\1\end{pmatrix} + \begin{pmatrix}1\\1\\0\\1\\0\\0\\5\\13\\1\\2\\2\end{pmatrix} + \begin{pmatrix}1\\0\\0\\0\\0\\1\\5\\11\\0\\0\\0\end{pmatrix} = \begin{pmatrix}4\\1\\0\\1\\1\\2\\22\\57\\2\\3\\3\end{pmatrix}$$

FIGURE 3.8 Métriques des deux kits de l'exemple, avec les fragments 1, 2, 3 pour le kit 1 à gauche, et 4, 5, 6, 7 pour le kit 2 à droite

données.

3.3.8 Génération de la généalogie

À partir de cette matrice de distances, on peut alors passer à la sous-étape suivante, l'inférence de la généalogie. Étant donné l'hypothèse de propagation avec le moins de changement possible, on cherche à minimiser la somme des distances des arêtes de la généalogie. La généalogie choisie est donc un *Minimum Spanning Tree* (MST) de la matrice de distance.

Chaque point de la généalogie représente un kit dans notre jeu de données, c'est-à-dire une déclinaison d'un site de phishing utilisé en pratique, et capturé. Des versions intermédiaires, non présentes dans le jeu de données, peuvent exister et être absentes de cette généalogie. Enfin, les arêtes peuvent être interprétées comme la dérivation la plus probable entre deux kits par similarité dans notre jeu de données. Ainsi, les chemins de la généalogie peuvent être interprétés comme des potentiels chemins de propagation de modification des kits. Comme indiqué lors de la présentation des objectifs de recherche (section 1.3), la généalogie ne reflète pas nécessairement l'évolution réelle des kits, mais elle garantie que l'effort de modification global est minimal du point de vue de la similarité.

3.3.9 Comparaison de kits

La distance de Manhattan entre les vecteurs de kits fournit une estimation, rapide à calculer, de la similarité entre les kits, permettant alors d'inférer une généalogie. Cependant, elle peut introduire des distorsions : on peut avoir des kits avec les mêmes vecteurs de fréquence

```php
function factorielle($n) {
  // Fragment 1
  $res = 1;
  for ($i = 2; $i <= $n; $i++) {
     $res = $res * $i;
  }
  return $res;
}
function afficherMessage() {
  // Fragment 2
  echo "Ceci est un message";
}
function comparer($a, $b) {
  // Fragment 3
  if ($a == $b)
    echo "Egal";
  elseif ($a > $b)
    echo "Superieur";
  else
    echo "Inferieur";
}
```

FIGURE 3.9 Exemple – Code PHP du kit 1

```php
function factorielle($n) {
  // Fragment 4
  $res = 1;
  for ($i = 2; $i <= $n; $i++) {
     $res = $res * $i;
  }
  return $res;
}
function afficherMessage() {
  // Fragment 5
  echo "Autre message";
}
function comparer($a, $b) {
  // Fragment 6
  if ($a == $b)
    echo "Egal";
  else
    echo "Different";
}
function somme($a, $b) {
  // Fragment 7
  return $a + $b;
}
```

FIGURE 3.10 Exemple – Code PHP du kit 2

avec pourtant des fragments différents, en permutant des instructions entre fragments par exemple. Pour remédier à ce problème, on cherche à faire une comparaison précise entre deux kits donnés, en considérant les fragments comme unités. Pour avoir une distance faible, les relations entre fragments sont importantes, il ne suffit plus d'avoir les mêmes fréquences de jetons à l'échelle du kit. Pour des raisons de performance, on ne peut pas appliquer ce calcul à toutes les paires de kits, mais on peut se restreindre à faire le calcul pour toutes les arêtes de la généalogie. Cela nous fournit alors des distances plus précises, mais seulement pour les arêtes de la généalogie.

Le but de la comparaison est d'associer chacun des fragments d'un premier kit à son fragment le plus proche (s'il existe) de l'autre kit. Pour cela, on définit d'abord les quatre types de relations possible entre un fragment f_1 d'un kit k_1 et un fragment f_2 d'un kit k_2 :

- f_1 et f_2 sont **identiques** si et seulement si ils sont des clones de type 1 (même définition que dans la section 3.3.1) ;
- f_1 et f_2 sont **paramétriques** si et seulement si ils sont des clones de type 2 (même définition que dans la section 3.3.1) ;
- f_1 et f_2 sont **similaires** si et seulement si $\|v_1 - v_2\|_1 \leq \alpha \min(\|v_1\|_1, \|v_2\|_1)$, avec v_1 (respectivement v_2) le vecteur de métriques de f_1 (respectivement de f_2) ;
- f_1 et f_2 sont **originaux** si et seulement si ils ne sont ni identiques, ni paramétriques, ni similaires.

La définition de similaire correspond à une interprétation algébrique des clones de type 3 (voir section 1.1.2). On considère que deux fragments sont des clones avec quelques modifications structurelles lorsqu'ils partagent un nombre significatif de types jetons, c'est-à-dire qu'ils ont une distance entre leurs vecteurs de métriques faible. Pour cela, on utilise donc un seuil proportionnel à la taille des fragments, plutôt qu'un seuil absolu. Dans ce travail, la valeur du seuil proportionnel retenue est $\alpha = 0.3$, valeur recommandée dans la littérature [55] lorsque la distance de Manhattan est utilisée. Il est à noter que la relation de similarité entre deux fragments n'est pas une relation d'équivalence, puisque non transitive, et donc les fragments ne peuvent pas être regroupés dans des classes d'équivalence.

Enfin, deux fragments avec une distance trop importante sont considérés originaux, car trop différents pour être des clones. Cela ne forme pas non plus une relation d'équivalence. Pour la comparaison entre deux kits, on considère qu'un fragment d'un kit est **original** quand aucun des fragments de l'autre kit n'est un clone de ce fragment (ni identique, ni paramétrique, ni similaire). Il n'est donc associé à aucun autre fragment, contrairement aux relations précédentes qui sont définies par paire.

Pour comparer deux kits, on cherche à calculer une couverture des fragments : chaque frag-

ment d'un kit est associé à un fragment de l'autre kit (si possible). Quand plusieurs fragments d'un autre kit peuvent être associés à un fragment, on l'associe en respectant l'ordre de priorité suivant :

1. fragment identique
2. fragment paramétrique
3. fragment similaire, par distance croissante

Pour chercher à quel fragment associer un fragment donné d'un kit, un algorithme naïf serait de calculer la distance à tous les autres fragments de l'autre kit, puis de les trier par ordre croissant. Des fragments identiques ou paramétriques ayant les mêmes types de jetons, ils sont donc à une distance de 0 jeton, et se retrouveraient parmi les premiers éléments triés. Cependant, calculer toutes les distances, en plus d'être coûteux, n'est pas nécessaire : au-delà d'une distance seuil, les fragments ne sont plus considérés similaires et peuvent donc être ignorés. Pour tirer profit de ce constat, l'utilisation des arbres métriques s'avère avantageux pour le calcul des voisins d'un fragment à un rayon fixé par le seuil de similarité α.

Comparer toutes les paires des n vecteurs avec la distance de Manhattan a une complexité asymptomatique de $\mathcal{O}(n^2)$ dans le meilleur et le pire des cas. Cela pose problème étant donné le grand nombre de vecteurs à traiter, et s'adapte mal à l'ajout de nouveaux vecteurs. Pour réduire le temps d'exécution et la complexité spatiale, on a recours aux arbres métriques [55] pour sauvegarder et accéder aux vecteurs. L'indexation des arbres métriques permet le calcul efficace des distances entre vecteurs, et également celui du plus proche voisin (PPV) d'un vecteur, sans perte d'information. Un arbre métrique possède, entre autres, une méthode `insert` qui permet d'indexer un nouveau vecteur, et une méthode `rangeQuery` qui permet de récupérer tous les vecteurs à distance d'un vecteur donné inférieure à un certain rayon, le tout avec une complexité moyenne de $\mathcal{O}(log(n))$. Cela réduit la complexité moyenne du calcul des voisins à un rayon de distance fixé d'un fragment à $\mathcal{O}(n \cdot log(n))$, qui est un ordre de grandeur inférieur au calcul de toutes les paires possibles.

La distance de Manhattan (équation 3.1), dans l'espace $\mathbf{N}^k$ avec k un naturel non nul, respecte bien les axiomes d'une distance (positive, symétrique, inégalité triangulaire), elle peut donc être utilisée comme distance d'un arbre métrique [56].

On peut désormais procéder à la comparaison de kits, en identifiant les relations entre paires de fragments, et en priorisant les fragments avec un haut degré de similarité, à l'aide d'arbres métriques pour accélérer le calcul.

Deux choix de couvertures des fragments entre un kit k_1 et k_2 sont envisagés :

— la couverture stricte, où un fragment de k_2 ne peut être associé qu'à au plus un fragment de k_1, et vice versa (algorithme détaillé donné en annexe A) ;
— la couverture relâchée, où un même fragment de k_2 peut être associé à plusieurs fragments de k_1 (algorithmes simplifié et détaillé donnés en annexe B).

```
1: fonction CouvertureStricte(kit1, kit2)
2:     distance ← 0
3:     pour tout f1 ∈ kit1 faire
4:         f2 ← le fragment non couvert du kit2 le plus proche de f1
5:         si f2 existe alors
6:             si f1 et f2 sont identiques ou paramétriques alors
7:                 marquer f1 et f2 comme tels, marquer f2 comme couvert
8:                 // distance inchangée, puisque d(f1, f2) = 0
9:             sinon si d(f1, f2) < α|f1| et d(f1, f2) < α|f2| alors
10:                marquer f1 et f2 comme similaires, marquer f2 comme couvert
11:                distance ← distance + d(f1, f2)
12:            sinon
13:                marquer f1 comme original, distance ← distance + |f1|
14:            fin si
15:        sinon
16:            marquer f1 comme original, distance ← distance + |f1|
17:        fin si
18:    fin pour
19:    pour tout f2 ∈ kit2 non couverts faire
20:        marquer f2 comme original, distance ← distance + |f2|
21:    fin pour
22:    renvoie distance, relations des fragments
23: fin fonction
```

Figure 3.11 Algorithme simplifié de la comparaison de kits par couverture stricte

Pour comparer deux kits, on choisit la version avec couverture stricte, dont l'algorithme simplifié est donné figure 3.11. La version détaillée est également disponible en annexe A. L'algorithme permet de déterminer quel fragment d'un kit correspond à quel fragment de l'autre kit, ainsi que leur relation de similarité. Il renvoie également une distance entre les deux kits, plus précise que la distance de Manhattan entre les vecteurs de kits, mais également plus lente à calculer. C'est pour cette raison que la comparaison de kits se fait uniquement sur les arêtes de la généalogie, et non pas sur toutes les paires de kits.

L'algorithme de couverture relâchée, imposant moins de contraintes, a tendance à trouver plus de fragments identiques, là où l'algorithme de couverture stricte peut en trouver moins, puisque la meilleure correspondance pour un fragment est potentiellement déjà couvert par

un autre fragment. En pratique, la différence observée de résultats entre les deux algorithmes de couverture est faible. C'est pourquoi, il a été choisi d'utiliser l'algorithme de comparaison de kit par couverture stricte. Cet algorithme est plus rapide à exécuter que le second, car il est symétrique, et ne nécessite donc pas d'être exécuté deux fois par paire de kits, contrairement au second.

Par rapport aux algorithmes (simplifiés et détaillés), une autre priorité est prise en compte : si plusieurs fragments candidats ont la même priorité (même type de relation, et même distance si similaires) pour être associés à un fragment donné, on choisit en priorité un fragment qui provient du même nom de fichier, si possible. En cas de fichiers dupliqués au sein d'un kit, cela permet de conserver les fragments d'un même fichier ensemble, notamment pour la visualisation de changements paramétriques. La visualisation de différences entre kits est décrite dans le chapitre sur les expériences, section 4.5.

La distance fournie par la comparaison de kits présente de multiples avantages. Elle prend en compte la granularité au niveau des fragments, ce qui permet d'obtenir une distance plus précise. Aussi, elle est plus robuste aux attaques antagonistes (*adversarial*) : si un attaquant essaye de créer un nouveau fragment qui contient des jetons pour être similaire à un autre kit, ce fragment sera classé comme original, et les autres fragments resteront associés correctement. Cela aura pour conséquence d'augmenter la distance par la taille du nouveau fragment, et non de la réduire.

Cependant, cette distance présente aussi quelques limitations. Premièrement, elle est plus lente à calculer que la distance de Manhattan entre deux vecteurs, ce qui la rend inapplicable pour toutes les paires de kits. Aussi, elle est sensible à la fusion de fragments et à la séparation d'un fragment en plusieurs autres. Par exemple, dans le cas de la séparation, si un fragment d'un kit k_1 est divisé en deux dans le kit k_2, ce fragment ne pourra être associé qu'à un seul des autres fragments de k_2, ce qui augmentera la distance de manière artificielle. Cela pourrait être exploité pour des attaques antagonistes.

3.3.10 Exemple de comparaison de kits

Reprenons les kits donnés en exemple dans la section 3.3.7, en appliquant cette fois l'algorithme de comparaison de kits par couverture stricte. Pour cet exemple, on choisit $\alpha = 0.5$ comme seuil pour les fragments similaires. On commence avec une distance totale nulle, puis on choisit le fragment 1 du premier kit (figure 3.9). Le fragment le plus proche, en respectant les priorités données dans la section 3.3.9, est donc le fragment 4 (figure 3.10), puisqu'ils ont le même code source en ignorant les commentaires. On marque alors les fragments 1 et 4 comme identiques, et on laisse la distance totale à 0. Ensuite, le fragment 2 est associé au

fragment 5, et ils sont marqués comme paramétriques, sans modifier la distance totale. Enfin, le fragment 3 est associé au fragment 6, ils sont marqués comme similaires, et on ajoute à la distance totale la distance entre leurs vecteurs de métriques, c'est-à-dire 0 + 0 + 1 + 0 + 0 + 0 + 2 + 5 + 1 + 1 + 1, soit 11 jetons. Si on note v_3 et v_6 les vecteurs des fragments 3 et 6, on a bien $l_1(v_3, v_6) < \alpha \min(\|v_3\|_1, \|v_6\|_1) = 13$. Finalement, il n'y a plus de fragments dans le premier kit, donc le fragment 7 est marqué comme original. Il ajoute alors tous ses jetons à la distance totale, ce qui l'augmente de 18 jetons.

On obtient une distance totale entre les deux kits de 29 jetons, ce qui est plus élevé que les 15 jetons obtenus en sommant les vecteurs de métriques des fragments dans la section 3.3.7. Les relations obtenues sont résumées ci-dessous :

— fragments 1 et 4 identiques
— fragments 2 et 5 paramétriques
— fragments 3 et 6 similaires
— fragment 7 original

3.4 Analyse multi-langage

Une fois que chaque langage a été analysé de manière indépendante, il est possible de combiner les résultats de PHP, JS et HTML pour obtenir une vision plus globale des kits. Certains langages comme PHP sont utilisés pour écrire la collecte des informations et leur acheminement jusqu'à l'attaquant, tandis que d'autres langages comme HTML et JS sont plus utilisés côté client pour jouer sur la forme et imiter des marques connues. Ainsi, d'un kit à l'autre, le code d'un langage peut changer (par exemple le visuel du site) tandis que le code des autres langages demeure inchangé (par exemple les informations demandées restent les mêmes). Combiner les trois langages analysés ici nous permet de voir les kits comme un agrégat de ces langages. Le schéma illustrant l'approche de l'analyse multi-langage est donné figure 3.12. Les étapes illustrées dans le schéma sont décrites dans les sections suivantes, à savoir la combinaison des partitions (section 3.4.1), la combinaison des distances (section 3.4.2), l'inférence de la généalogie multi-langage et la combinaison des comparaisons de kits (section 3.4.3).

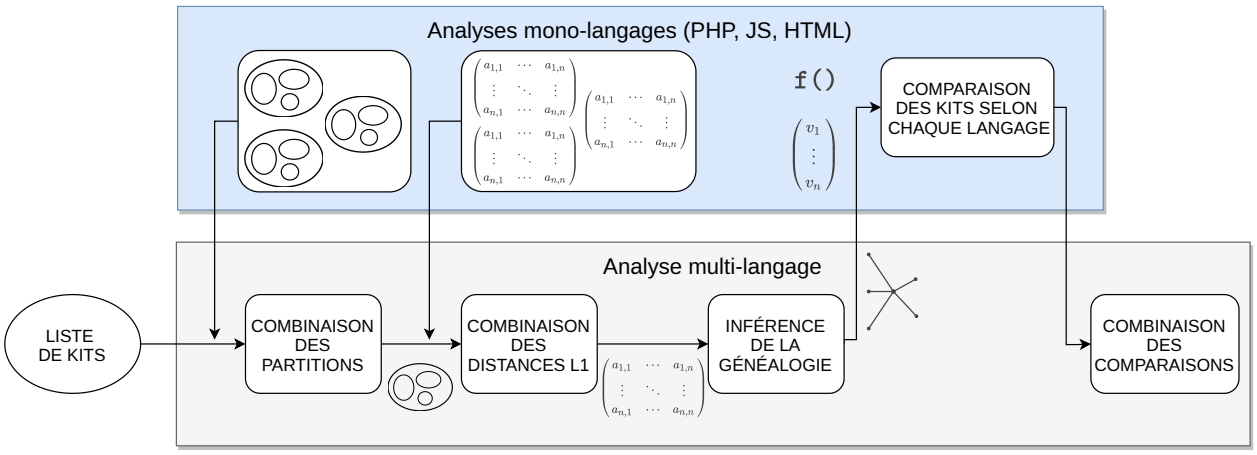


FIGURE 3.12 Schéma résumant l'approche de l'analyse multi-langage

3.4.1 Combinaison des partitions de kits

Un fragment ne pouvant être que d'un seul langage par définition, il n'y a pas calcul à faire concernant les classes d'équivalence de fragments.

Les kits ne sont désormais plus des ensembles de fragment pour un seul langage, ils contiennent des fragments de langages différents. Il n'est pas nécessaire de calculer les classes d'équivalence multi-langage en reprenant la même approche que pour celles mono-langages (section 3.3.3, en attribuant un ensemble de fragments à chaque kit). À la place, on peut réutiliser les résultats des classes d'équivalence mono-langages.

Considérons dans un premier temps les relations d'équivalence des kits identiques mono-langages. On cherche à étendre cette définition pour les kits n'ayant pas de fragments dans un langage. On considère deux kits k_1 et k_2, et on note $k_1 \sim_{PHP} k_2$ pour indiquer que les deux kits sont identiques en PHP. On a trois cas de figures :

— Les deux kits possèdent des fragments en PHP : ils appartiennent alors tous les deux à une classe d'équivalence de kits identiques de PHP. Dans ce cas, on a $k_1 \sim_{PHP} k_2$ si et seulement si k_1 et k_2 appartiennent à la même classe d'équivalence de kits identiques de PHP.
— Aucun des deux kits n'a de fragment en PHP. Dans ce cas, on a $k_1 \sim_{PHP} k_2$, puisque les ensembles de classes de fragments PHP des deux kits sont l'ensemble vide.
— Un seul des deux kits n'a pas de fragment en PHP. Alors, $k_1 \sim_{PHP} k_2$ est faux.

De la même manière, on définit les relations $\sim_{JS}$ et $\sim_{HTML}$ pour respectivement JS et HTML. On peut désormais vérifier si deux kits sont identiques de manière mono-langage pour tous les langages, peu importe les langages qui composent les kits.

On définit alors la relation d'équivalence multi-langage $\sim$ dans l'équation 3.3. Deux kits sont identiques selon cette définition si et seulement si ils sont identiques dans tous les langages.

$$k_1 \sim k_2 \iff \begin{cases} k_1 & \sim_{PHP} & k_2 \\ k_1 & \sim_{JS} & k_2 \\ k_1 & \sim_{HTML} & k_2 \end{cases} \tag{3.3}$$

On construit alors les classes d'équivalence multi-langages de kits identiques en vérifiant pour chaque kit ses classes d'équivalences mono-langages dans les langages de ses fragments.

On procède de la même manière pour les classes d'équivalence multi-langage de kits paramétriques, en prenant soin de remplacer la relation d'équivalence des kits identiques par celle des kits paramétriques.

Finalement, on ne considère pour le reste de l'analyse que les kits paramétriquement distincts selon les classes d'équivalences multi-langages, de manière similaire à ce qui est fait dans l'analyse mono-langage. Ces kits sont plus nombreux que ceux obtenus dans chacune des analyses mono-langages, puisque par définition, deux kits paramétriques de manière multi-langages sont nécessairement paramétriques de manière mono-langage dans chacun des langages.

3.4.2 Combinaisons des matrices de distance

Similairement à l'étape de calcul des classes d'équivalence (section 3.4.1), on combine les matrices de distances des différents langages, plutôt que de redéfinir les vecteurs et calculer les distances l_1 entre tous les kits.

Dans un premier temps, on étend la matrice de distances d'un langage à tous les kits, incluant ceux qui n'ont pas de fragments dans le langage considéré. Pour cela, on note k_1 et k_2 deux kits, respectivement v_1 et v_2 leurs vecteurs de métriques. On définit alors la fonction d_{PHP} dans l'équation 3.4, qui permet de calculer la distance en jetons PHP entre deux kits quelconques, ayant des fragments en PHP ou non. L'idée est de considérer qu'un kit n'ayant pas de fragment en PHP a un vecteur de métriques nul. Il est à noter que les valeurs du type $\|v_1 - v_2\|_1$ n'ont pas à être calculées, puisque le résultat est déjà stocké dans la matrice de distances. Cependant, il faut faire attention au fait que la matrice de distance n'est calculée que sur les kits paramétriquement distincts, et non sur tous les kits. Il faut donc considérer les représentants des classes de kits paramétriques, et non les kits eux-mêmes pour récupérer la distance dans la matrice.

$$d_{PHP}(k_1, k_2) = \begin{cases} \|v_1 - v_2\|_1 & \text{si } k_1 \text{ et } k_2 \text{ ont des fragments en PHP} \\ \|v_1\|_1 & \text{si } k_1 \text{ a des fragments en PHP, mais pas } k_2 \\ \|v_2\|_1 & \text{si } k_2 \text{ a des fragments en PHP, mais pas } k_1 \\ 0 & \text{si ni } k_1 \text{ ni } k_2 \text{ n'ont de fragment en PHP} \end{cases} \tag{3.4}$$

De manière similaire, on définit les fonctions d_{JS} et d_{HTML} pour respectivement le JS et le HTML. Avec ces trois fonctions, il est donc possible de calculer la distance entre deux kits quelconques, indépendamment des langages dont ils sont composés.

Enfin, on définit la fonction de distance multi-langage d dans l'équation 3.5, en notant toujours k_1 et k_2 deux kits quelconques. Il s'agit simplement de la somme de ces trois fonctions. Bien que les types des jetons n'aient pas la même définition pour les trois langages, on s'attend à ce que ces unités soient du même ordre de grandeur, puisque ils jouent tous le rôle d'unités élémentaires dans leur langage de programmation.

$$d(k_1, k_2) = d_{PHP}(k_1, k_2) + d_{JS}(k_1, k_2) + d_{HTML}(k_1, k_2) \tag{3.5}$$

Il suffit désormais de calculer la valeur de la fonction d pour toutes les paires de kits paramétriquement distincts selon l'analyse multi-langage, et de stocker le résultat sous forme d'une matrice de distance. Tout comme pour les matrices de distances mono-langage, seulement la moitié de la matrice multi-langage a besoin d'être calculée.

3.4.3 Génération de la généalogie et comparaison de kits

Pour générer la généalogie multi-langage, le même procédé qu'une généalogie mono-langage est appliqué, voir section 3.3.8. On utilise simplement la matrice de distance multi-langage à la place d'une matrice mono-langage.

La comparaison de kits, associant entre eux les fragments d'une paire de kits, est faite pour toutes les paires de kits dans chacune des généalogies mono-lanagages, qui sont différentes de la généalogie multi-langage. Ainsi, si on veut comparer les kits voisins de la généalogie multi-langage selon les trois langages à la fois, on commence par comparer les paires de kits des arêtes de la généalogie multi-langage, pour chacun des langages de manière indépendante, comme indiqué dans la section 3.3.9.

Ensuite, il suffit de combiner, pour chaque paire de kits de la généalogie multi-langage, les résultats des trois langages, par exemple en sommant le nombre de fragments identiques en PHP, en JS, et en HTML pour une même paire de kits.

CHAPITRE 4 EXPÉRIENCES

4.1 Description des jeux de données

Toutes les expériences ont été effectuées sur un ordinateur disposant d'un processeur Intel(R) Core(TM) i7 950, 8 cœurs cadencés à 3.07 GHz, avec 16 Gio de mémoire vive, sous le système d'exploitation Fedora 31.

Les détails sur les différents parseurs utilisés sont donnés dans la section 3.2.

4.1.1 Kits de phishing

Les données analysées ici sont composées de 20 871 kits de *phishing* uniques. Le tableau 4.1 montre la répartition des fichiers selon leur extension, sauf pour PHP. Comme indiqué dans l'étape de pré-traitement (section 3.1), du code PHP peut se trouver dans un fichier avec une autre extension que le `.php`, en étant exécuté avec la fonction `eval` par exemple. On ne se fie donc pas à l'extension de fichier pour identifier les fichiers PHP.

Les kits de phishing analysés ici sont principalement écrits dans des langages web tels que PHP, HTML et JS. PHP semble être le principal langage, si on se fie au nombre absolu de fichiers. En revanche, si on compte les lignes de code (en prenant garde que des blocs d'HTML statiques peuvent se trouver dans du PHP, idem pour JS dans du HTML), JavaScript semble être le langage prédominant. Cela peut être dû au fait que les kits hébergent les fichiers sources des principales libraires JS utilisées, comme `JQuery`, `Bootstrap`, etc. Également, il est raisonnable de penser que Javascript est un langage plus verbeux que PHP en termes de LOC. Quoi qu'il en soit, le nombre de LOC n'est pas pris en compte par l'analyse, et le langage principal est déterminé lors de la corrélation des distances de langages (voir résultats dans la section 5.5). Dans tous les cas, PHP, HTML et JS restent les principaux langages présents dans le jeu de données, ce qui justifie le fait d'orienter l'analyse vers ces trois langages. D'autres fichiers, bien que non analysés dans ce travail peuvent également prendre part à la logique des attaques de `phishing`. Les kits incluent aussi des fichiers images, qui sont principalement des fichiers PNG (54,4 % des fichiers images), GIF (23,8 %), JPEG (11,6 %) et SVG (10,2 %). Les types de fichiers les plus fréquents parmi les autres fichiers sont les suivants : les fichiers CSS (35,2 % des autres fichiers), `.download` (8,6 %), TXT (7,9 %) et `.htaccess` (6,9 %).

Type de fichier	# fichiers	# fichiers (%)	LOC	LOC (%)
PHP	338 985	17,95%	55 765 042	30,63%
JS	201 219	10,66%	82 169 018	45,13%
HTML	119 970	6,35%	44 152 496	24,25%
Images	669 146	35,44%	-	-
Autre	558 930	29,60%	-	-
Total	1 888 250	100%	182 086 556	100%

Tableau 4.1 Répartition des types de fichiers – kits de phishing

Les statistiques de parsage sont consignées dans les tableaux 4.2, 4.3, et 4.4, pour respectivement PHP, JS, et HTML. Elles rapportent le nombre de fragments trouvés dans le langage considéré, et également le nombre de kits ayant au moins un fragment dans ce langage. PHP est le langage présent dans le plus de kits, en étant absent de seulement de 4 kits. HTML est le second langage prédominant, seule une faible part des kits n'ont pas de fichiers HTML ou uniquement des fichiers PHP générant du code HTML de manière dynamique, sans bloc de code statique. Il est également possible que d'autres extensions de fichiers soient utilisées, mais ignorés par l'analyse. Enfin, du code JS est présent dans environ deux kits sur trois.

Puisque que le parseur PHP utilisé n'est pas tolérant aux erreurs, le taux d'erreurs est aussi rapporté. Le pourcentage de fichiers PHP syntaxiquement invalides a été mesuré à l'aide de l'option de vérification syntaxique de l'interpréteur PHP officiel. Il a été utilisé avec la ligne de commande `php -l`, avec la version `PHP 7.3.13 (December 2019)`. Le taux d'erreur du parseur développé dans le laboratoire est calculé sur le nombre de fichiers syntaxiquement corrects (tel qu'indiqué par l'interpréteur officiel) qui ne peuvent pas être parsés par le parseur.

À cause du très grand nombre de fragments JS générés, une limite a été établie pour ne pas ralentir l'analyse avec de trop nombreux fragments peu pertinents. Ainsi, seuls les fragments ayant une taille supérieure à 50 jetons (environ une dizaine de LOC) sont conservés.

# kits	20 867
# fragments	1 089 639
# fichiers PHP invalides	0,72%
# erreurs du parseur PHP	0,30%

Tableau 4.2 Statistiques de parsage des kits de phishing – PHP

# kits	14 413
# fragments	18 250 741

Tableau 4.3 Statistiques de parsage des kits de phishing – JS

# kits	19 341
# fragments	223 726

Tableau 4.4 Statistiques de parsage des kits de phishing – HTML

4.1.2 WordPress

Afin de s'assurer de la validité de notre approche, une autre expérience est effectuée sur WordPress. Cette application a été choisie parce qu'elle est écrite principalement en PHP, HTML et JS, est utilisée par de nombreuses personnes, et est publiquement disponible. Puisque l'historique d'évolution de WordPress est publique [1], les résultats de nos algorithmes peuvent être comparés à l'évolution réelle du logiciel. Dans ce but, les analyses effectuées sur les kits ont également été appliquées à chaque version de WordPress [2], depuis la version 2.0 jusqu'à la version 4.8.1. Le jeu de données contient alors 235 versions de WordPress, ce qui en fait un plus petit jeu de données que celui des kits. Le tableau 4.5 montre la répartition des fichiers selon leur extension, pour toutes les versions de WordPress cumulées. Les fichiers images sont principalement des fichiers PNG (50,8 %) et GIF (43,27 %), le reste étant quelques fichiers SVG (3,78 %) et JPEG (2,15 %). Les autres types de fichier les plus fréquents sont les fichiers CSS (69,63 %), TXT (7,90 %) et SCSS (3,55 %). Les statistiques sur les fichiers parsés sont données dans les tableaux 4.6, 4.7, 4.8 pour respectivement PHP, JS, HTML. On remarque bien que, quel que soit le langage, on trouve des fragments dans toutes les versions de WordPress.

1. https ://make.WordPress.org/core/
2. https ://WordPress.org/download/releases/

Type de fichier	# fichiers	# fichiers (%)	LOC	LOC (%)
PHP	102 320	41,82%	50 700 301	81,44%
JS	57 374	23,45%	11 448 656	18,39%
HTML	453	0,19%	105 182	0,17%
Graphics	44 457	18,17%	-	-
Other	40 073	16,38%	-	-
Total	244 677	100%	62 499 947	100%

Tableau 4.5 Répartition des types de fichiers – WordPress

# versions	235
# fragments	1 358 882
# fichiers PHP invalides	0%
# erreurs du parseur PHP	0%

Tableau 4.6 Statistiques de parsage de WordPress – PHP

# versions	235
# fragments	2 538 323

Tableau 4.7 Statistiques de parsage de WordPress – JS

# versions	235
# fragments	2 719

Tableau 4.8 Statistiques de parsage de WordPress – HTML

4.1.3 Comparaison des deux jeux de données

Le parseur utilisé dans le laboratoire affiche un très faible taux d'erreur (0.30% pour les kits, 0% pour WordPress), ce qui suggère sa fiabilité pour son utilisation dans des expériences. Cela a été mesuré sur les fichiers PHP syntaxiquement corrects, comme décrit dans la section 4.1.1.

Parmi les kits de *phishing*, une partie des fichiers PHP syntaxiquement incorrects ont été inspectés manuellement. La plupart des erreurs observées sont triviales, par exemple des chaînes de caractères dont le dernier guillemet est manquant, ou encore l'oubli de point-virgule en fin d'instruction. Il est possible que ces fichiers proviennent de kits non fonctionnels, mais il est également probable que ces fichiers soient des versions intermédiaires non vouées à être exécutés, qui n'ont pas été supprimées, tandis que les versions corrigées se situent dans d'autres fichiers.

Les fichiers images, ainsi que les autres types de fichiers, sont plus diversifiés dans le jeu de données des kits que dans celui de WordPress. Comme le jeu de données des kits est une collection hétérogène de logiciels, il contient de nombreux types de fichiers, tandis que celui de WordPress est une collection homogène du même logiciel, mais en plusieurs versions, qui utilise donc un nombre limité de type de fichiers.

Dans les deux jeux de données, les fichiers PHP représentent la majorité du code source, même si cela reste plus mitigé pour les kits. Puisque PHP est encore plus prédominant dans les versions de WordPress, on s'attend à ce que l'évolution de sa partie PHP soit très proche de l'évolution globale du logiciel.

Pour les kits, 0,90 % des fichiers PHP détectés n'ont pas une extension `.php`. Des fichiers avec pour extension `.html`, `.txt`, `.gif`, ou `.asp`, par exemple, peuvent également contenir du code PHP. Ces fichiers représentent 2,5 % des lignes de code totales en PHP. Cela montre que les kits de `phishing` utilisent d'autres types de fichier pour stocker du code PHP, présumablement dans le but de cacher ce code. À l'inverse, seul 0,06 % des fichiers PHP détectés dans WordPress représentent ce cas de figure. Ceci est attendu, puisque les développeurs de WordPress n'ont pas de raison particulière de cacher du code PHP, ils utilisent l'extension de nom de fichier correct pour PHP. Les quelques fichiers contenant du code PHP sans avoir cette extension sont par exemple des fichiers `README` qui contiennent du code PHP comme exemple.

Que ce soit l'interpréteur officiel PHP ou le parseur du laboratoire, aucun fichier erroné n'a été détecté lors du parsage du code source de WordPress. Comme il s'agit d'un logiciel standard, il ne contient pas de syntaxe inhabituelle dans son code source. Au contraire, pour les kits de *phishing*, certains fichiers, bien que syntaxiquement corrects, n'ont pas pu être parsés par le

parseur utilisé. Bien que le nombre d'erreurs de parsage est faible, certaines fonctionnalités n'ont pas été implémentées dans le parseur. Également, des syntaxes inhabituelles dans le code source peuvent provoquer des erreurs inattendues.

4.2 Classes d'équivalence

L'analyse des classes d'équivalence permet de rendre compte de la duplication des fragments et des kits, selon chaque langage, et en combinant les trois langages analysés. On peut ainsi savoir quelle est la fréquence de clones, qu'ils soient identiques (de type 1), sans modification fonctionnelle apportée au code source, ou paramétriques (de type 2), avec de légères modifications consistant à changer quelques paramètres. On utilise alors les résultats des sections 3.3.1, 3.3.3 et 3.4.1.

Dans un premier temps, on s'intéresse au taux de duplication : combien de fragments (ou kits) ont au moins un clone identique, un clone paramétrique, ou aucun clone. Cela est un indicateur de la réutilisation des kits de *phishing*.

Dans un second temps, on s'intéresse aux tailles des classes d'équivalence de fragments (ou kits) dupliqués, en réalisant des statistiques sur les tailles des classes d'équivalences non triviales (qui contiennent plus d'un élément). Ceci nous permet de savoir à quel point certains fragments (ou kits) sont réutilisés.

En complément du point précédent, le nombre cumulatif de kits en fonction des classes d'équivalence nous renseigne sur la part des kits de *phishing* ayant été utilisés de très nombreuses fois.

Les résultats associés à ces expériences sur les classes d'équivalence sont disponibles dans la section 5.1

4.3 Visualisation des généalogies

On utilise les généalogies calculées de la manière expliquée dans les sections 3.3.8 et 3.4.3. Les distances calculées entre chaque paire de kits s'inscrivent dans un espace de grande dimension, de la taille du vecteur de métriques utilisé (pouvant facilement dépasser la centaine). Il est alors très difficile de visualiser les kits dans cet espace, afin de pouvoir faire des raisonnements sur leurs similitudes.

Pour remédier à ce problème, on utilise le MDS [75] implémenté par `scikit` [76]. Il s'agit d'un ensemble de techniques statistiques permettant de réduire un espace de haute dimension en un espace de faible dimension, dans le but de visualiser des similarités. Ainsi, chacun des

kits (paramétriquement distincts) vont être positionnés dans un espace 2D, en essayant de respecter le plus possible les distances renseignées dans la matrice de distance. Une fois les kits positionnés, on peut également les relier grâces aux arêtes de la généalogie. On obtient alors une représentation en 2D de la généalogie mono-langage prédite pour les kits, pour chaque langage. De même, on peut visualiser la généalogie multi-langage.

Il faut cependant préciser que le MDS n'est utilisé qu'à des fins de visualisation, et non d'analyse. En effet, avec un grand nombre de kits à placer, la technique introduit des distorsions de distances, où deux kits très similaires peuvent être placés de manière trop éloignée par exemple. Ainsi, la généalogie (le MST) est recalculée sur les distances euclidiennes de l'espace 2D, pour corriger les distorsions des distances. Cependant, lors des analyses ce sont bien les généalogies calculées à partir des vecteurs de métriques, plus précises, qui sont utilisées.

Pour l'analyse multi-langage, plutôt que de visualiser l'ensemble de la généalogie, il est aussi possible de se concentrer sur un seul kit. En choisissant ce kit, on peut visualiser une partie de la généalogie multi-langage, avec uniquement les kits voisins de ce kit dans les autres généalogies mono-langages, ainsi que les kits intermédiaires qui se situent entre le kit considéré et ses voisins de généalogies. Cela a l'avantage de n'impliquer qu'un nombre restreint de kits, et donc d'obtenir des résultats plus précis avec le MDS. Ainsi, on peut directement afficher les arêtes de la véritable généalogie, sans en recalculer une sur les distances euclidiennes du graphe 2D. Également, le fait de montrer les voisins selon les généalogies mono-langages nous renseigne sur les interactions entre les différents langages lors des modifications de kits. Les visualisations des généalogies, qu'elles soient entières ou partielles, sont visibles dans la section 5.2.

4.4 Similarité le long des généalogies

Une fois les clones paramétriques étudiés, il est possible d'utiliser les généalogies produites pour se concentrer sur les kits paramétriquement distincts (voir sections 3.3.8 et 3.4.3). Parmi ces kits, on cherche à savoir à quel point ils sont proches ou éloignés. Pour ce faire, on dispose de plusieurs indicateurs.

Le premier indicateur est la distance absolue entre les kits, le long d'une généalogie. Cela nous indique le nombre de jetons séparant deux kits, ce qui nous donne une idée du nombre de LOC modifiées entre deux versions potentielles de kits, et donc de l'effort investi. La distance absolue provient des résultats de l'algorithme de comparaison de kits (section 3.3.9) sur la généalogie, qui fournit une distance plus précise que la distance de Manhattan.

Le second indicateur, après la distance absolue, est la similarité, relative aux tailles des kits.

Au lieu de s'intéresser à la quantité absolue de changement, on regarde quelle portion reste inchangée entre deux kits. Pour cela, on utilise les vecteurs de métriques des kits et la distance de Manhattan décrits dans la section 3.3.6. On utilise deux mesures de la similarité pour cela :

— le coefficient de Jaccard (en utilisant les opérateurs intersection/union des multi-ensembles) ;
— la différence relative.

D'autres mesures, bien que non testées ici, sont possibles comme la similarité du cosinus.

Pour définir ces deux mesures, on considère deux kits k_1 et k_2. On note respectivement A et B leurs représentations multi-ensemblistes (voir section 3.3.6) : ainsi si le vecteur de métriques de k_1 indique qu'il est composé de n jetons d'un certain type, on retrouve alors ce type de jeton répété n fois dans A.

Le coefficient de Jaccard est définie dans l'équation 4.1, où $A \triangle B$ est la différence symétrique entre A et B, qui correspond donc à la distance de Manhattan entre les vecteurs de métriques de k_1 et k_2.

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A| + |B| - |A \triangle B|}{|A| + |B| + |A \triangle B|} \tag{4.1}$$

Le coefficient de Jaccard est symétrique, bien que la taille des kits puisse différer ($JC(A, B) = JC(B, A)$). Il est à noter que le coefficient de Jaccard est souvent inférieur au pourcentage intuitif qu'on pourrait attendre. Par exemple, deux ensembles de même taille partageant uniquement la moitié de leurs éléments entre eux ont un coefficient de Jaccard de $\frac{1}{3}$, et non de $\frac{1}{2}$.

La différence relative entre deux kits est définie dans l'équation 4.2.

$$RD(A, B) = \frac{|A - B|}{|A|} \tag{4.2}$$

La différence relative n'est pas symétrique. En effet, RD(A, B) peut être différent de RD(B, A) à cause des tailles potentiellement différentes de A et B. Contrairement au coefficient de Jaccard, il faut donc calculer la différence relative deux fois par arête de la généalogie.

Une autre façon d'évaluer la similarité entre les kits paramétriquement distincts est d'identifier à quel point les kits sont proches les uns des autres. Ainsi, on fixe un seuil de similarité, et on compte le nombre de kits qui ont un degré de similarité supérieur au seuil avec au moins un autre kit. Cela donne alors une estimation du nombre de kits singuliers, et qui ont donc de grandes chances de ne pas avoir été influencé par d'autres kits.

Les résultats sur les distances et similarités des kits des différentes généalogies sont donnés dans la section 5.3. On y retrouve également les tables portant sur la quantité de kit proches d'autres kits à seuil de similarité fixé.

4.5 Comparaison de kits

Jusqu'à présent, pour des raisons de temps de calcul, l'analyse de similarité s'est concentrée sur les kits en entier, sans considérer les fragments individuellement. Néanmoins, on peut être plus précis en raisonnant au niveau de la granularité des fragments le long d'une généalogie.

On souhaite déterminer quelles relations entretiennent les kits voisins des généalogies. On peut alors utiliser les répartitions des fragments en identiques, paramétriques, similaires ou originaux entre des kits voisins dans une généalogie. Ces données proviennent de l'algorithme de comparaison de kits, décrit dans la section 3.3.9.

Afficher le taux de fragments identiques le long d'une généalogie, par exemple, permet de savoir quelle quantité de fragments peuvent être ré-utilisés entre des kits qui présentent pourtant des différences structurelles. Également, les taux de fragments similaires ou originaux indiquent à quel point des fragments ont pu être modifiés entre des kits. Il faut noter que pour calculer les taux de fragments, on les pondère par leurs tailles en jetons, et on normalise par la taille des deux kits en jetons. Ainsi, pour chaque arête, on a deux valeurs de taux pour chaque type de fragment.

Dans l'optique de fournir des pistes d'exploration des kits, un outil de visualisation des différences jeton par jeton entre deux kits a été développé. Il permet de résumer les différences entre deux kits dans un fichier HTML, en montrant la quantité de fragments identiques, quels sont les fragments qui n'apparaissent que dans l'un des deux kits, etc. Les fragments paramétriques ou similaires sont mis côte à côte, en mettant en valeur les jetons qui diffèrent entre les deux versions, afin de voir quelles sont les modifications qui ont été faites. Cette différence repose sur la différence jeton par jeton des fragments, qui donnent des résultats plus précis qu'un `diff` (ligne par ligne) entre deux fichiers. De cette manière il est facile de repérer, par exemple, les différences d'adresses *email* utilisées pour communiquer les résultats d'une attaque entre deux configurations de kits de phishing. Cette visualisation reposant sur les résultats de la comparaison de kits, elle est disponible pour les paires de kit des arêtes des généalogies. Il est également possible de l'effectuer entre n'importe quelle paire de kits, en calculant la couverture des fragments à la volée.

Les résultats associés aux comparaisons de kits sont disponibles dans la section 5.4.

4.6 Corrélation entre langages

On dispose des distances multi-langages (section 3.4.2), et également des distances mono-langages pour chacun des langages (section 3.3.6). On peut alors se demander quels sont les langages qui ont le plus d'influence, c'est-à-dire quel langage parmi PHP, JS et HTML rend le plus compte de l'évolution multi-langage des kits.

Pour répondre à cette question, on calcule la corrélation entre les distances mono-langages des trois langages, ainsi que la distance multi-langage. On ne calcule pas la corrélation des distances des différents langages entre deux paires quelconques de kits, mais seulement entre les paires de kits des arêtes de la généalogie multi-langage, puisque l'on souhaite connaître l'influence des langages dans le cadre de l'évolution des kits. Contrairement aux autres analyses basées sur les généalogies, on n'ignore pas la multiplicité des kits paramétriques : si une arête relie deux classes d'équivalence de kits paramétriques de taille n et m, on compte la distance associée à l'arête $n \times m$ fois, et non pas une seule fois. Cela permet de rendre compte du fait que des kits populaires sont dupliqués de nombreuses fois, et évite alors de les sous-représenter.

Les variables aléatoires associées aux distances entre des paires de kits ne suivant pas une loi normale, on n'utilise pas la méthode de *Pearson*. Ainsi, la corrélation a été calculée à l'aide de la méthode de *Spearman*, qui permet de décrire à quel point deux variables peuvent être reliés par une relation monotone.

La matrice de corrélation des distances entre les différents langages est donnée dans la section 5.5.

4.7 WordPress

Pour tester l'approche quant à la génération de généalogies plausibles, WordPress a été choisi comme pseudo-oracle. La même approche a été appliquée qu'aux kits, en parsant et fragmentant les codes sources PHP, JS et HTML des 235 versions présentées dans la section 4.1.2. La généalogie multi-langage est ainsi générée, et on la compare à l'évolution du logiciel, disponible publiquement puisque le logiciel est *open-source*.

Le but de cette expérience est de valider que notre approche est capable de relier correctement les versions d'un logiciel par similarité, pour pallier au manque d'oracle des kits. Cependant, des bons résultats pour WordPress ne garantit pas que l'approche est correcte pour les kits, pour plusieurs raisons. D'abord, les jeux de données n'ont pas la même nature : un contient une collection hétérogène de logiciels, tandis que l'autre contient plusieurs versions d'un

même logiciel. Ensuite, l'évolution du logiciel conventionnel (qui suit par exemple les lois de Lehman [36]) n'est pas la même que celle du maliciel, qui peut contenir des portes dérobées [7], avoir recours à de l'obfuscation [9], etc. Cependant, il nous semble nécessaire d'obtenir de bons résultats sur une telle expérience pour justifier l'approche.

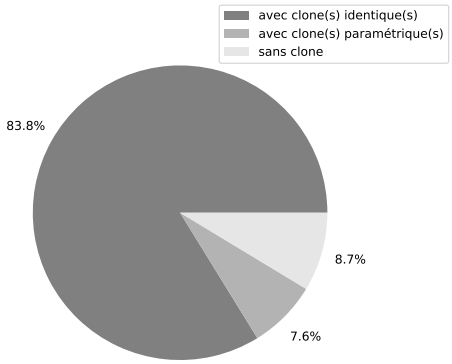
Pour comparer les généalogies, on s'assure que les versions mineures sont bien ordonnées, et on compte également le nombre d'arêtes présentes à la fois dans la généalogie prédite et celle réelle.

CHAPITRE 5 RÉSULTATS ET DISCUSSIONS

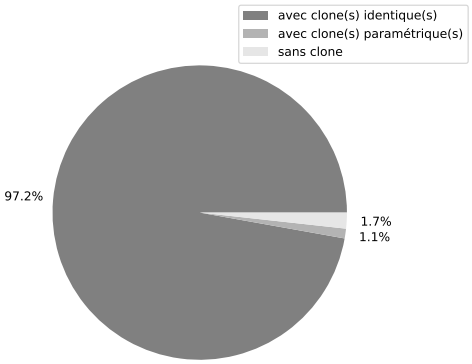
Les résultats des expériences décrites dans le chapitre 4 sont présentées ici, accompagnées de leurs discussions.

5.1 Classes d'équivalence

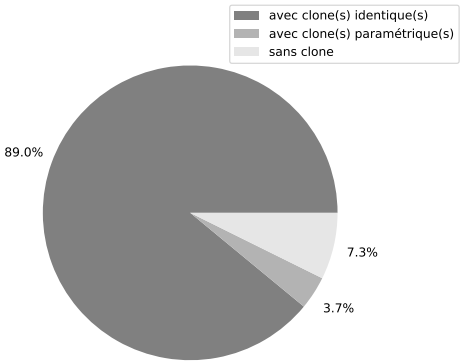
Les résultats présentés dans cette section proviennent des expériences décrites dans la section 4.2.



(a) PHP



(b) JS



(c) HTML

FIGURE 5.1 Distribution des taux de duplication des fragments

La figure 5.1 montre le taux de duplication des fragments selon chaque langage (PHP, JS et

HTML). Les fragments ne pouvant appartenir qu'à un langage, il n'y a pas de figure pour les trois langages combinés. Il s'agit des taux de fragments ayant au moins un clone identique, au moins un clone paramétrique (mais non-identique), ou aucun clone.

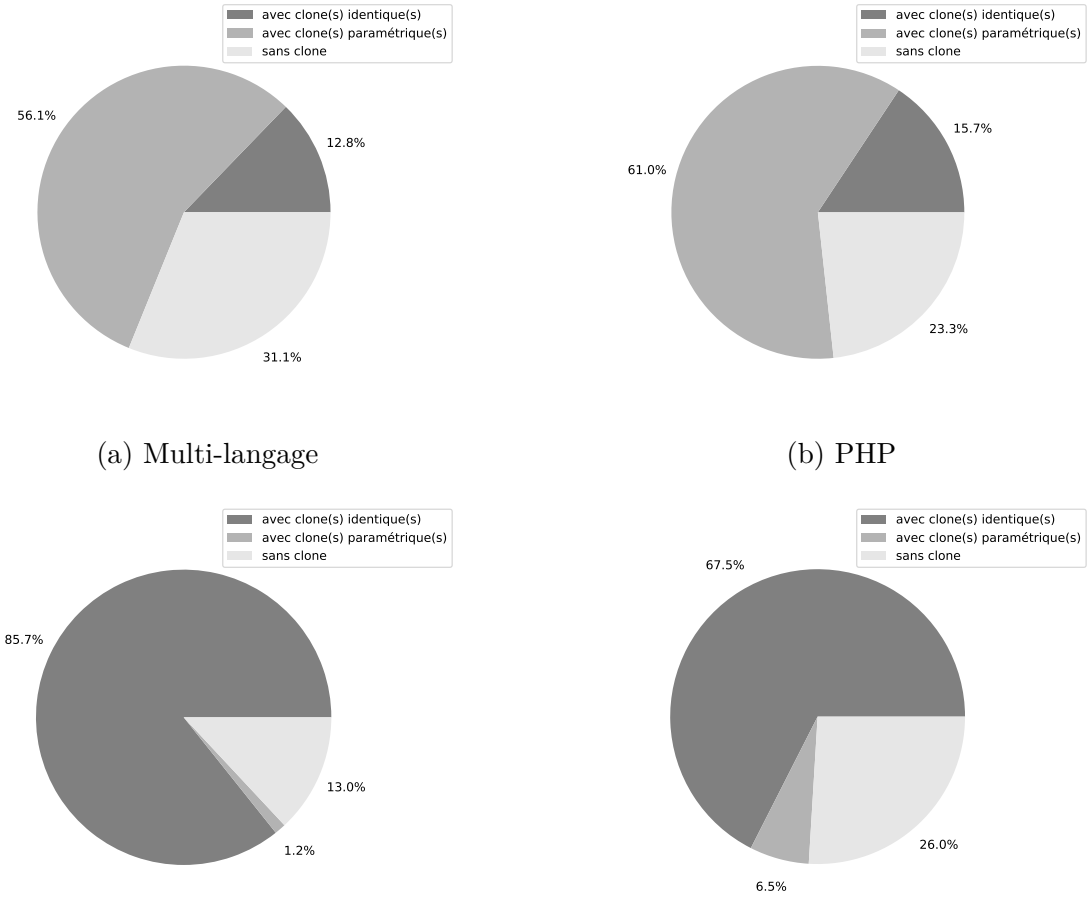
À travers les trois langages, on constate que les fragments sont très majoritairement réutilisés : les fragments avec au moins un clone identique sont majoritaires. Pour rappel, les fragments sont des méthodes ou fonctions en PHP et JS, et des pages entières en HTML. Ainsi, beaucoup de pages HTML sont communes à plusieurs kits, sans même avoir été éditée.

Pour PHP (figure 5.1a) et HTML (figure 5.1c), une part non négligeable des fragments ont au moins un clone paramétrique. Cela correspond à des fonctions ou pages qui sont réutilisés, mais en modifiants quelques paramètres : cela peut par exemple être des adresses *email* dans du PHP, ou du texte en HTML. Une autre part non négligeable des fragments n'ont pas de clone paramétrique (ou identique), il peut s'agir de fragments originaux, ou bien de fragments uniques mais qui sont (potentiellement) similaires à d'autres, avec par exemple en PHP quelques instructions ajoutées.

JS (figure 5.1b) semble se démarquer des deux autres langages : il ne présente que très peu de fragments sans clone identique. Il a été observé que les kits possèdent souvent des fichiers de bibliothèques Javascript telles que *JQuery*, *Bootstrap*, etc. Ces bibliothèques étant communes, il est très probable qu'au moins deux kits partagent leurs fichiers, et donc que leurs fragments aient des clones identiques. Le faible taux de fragments sans clones identiques (environ 3%) suggère que les modifications sur la quantité de code JS écrit par les auteurs de kits est minime, ou du moins négligeable face au volume des bibliothèques utilisées. Exclure les bibliothèques de l'analyse pourrait donner des résultats intéressants, mais cela représente un problème complexe puisque ces fichiers en apparence bénins peuvent comporter des instructions malveillantes dissimulées par les auteurs de kits. Également, JS a recours à plus de fonctions que les autres langages, ce qui peut artificiellement augmenter les taux de duplication identique.

La figure 5.2 montre le taux de duplication des kits selon les trois langages combinés et selon chaque langage (PHP, JS et HTML).

On observe que la majorité des kits ont des clones identiques du point de vue de JS (figure 5.2c) ou de HTML (figure 5.2d). Ainsi, on retrouve que la grande majorité des kits de phishing ont des rendus qui sont clonés identiquement plusieurs fois, sans que ce soit le même site pour tous ces kits. Pour HTML, une part importante des kits n'ont pas de clones, ce qui correspond à des kits ayant des contenus originaux ou avec des modifications plus ou moins importantes d'autres kits. Également, une certaine quantité de kits ont des clones paramétriques, ce qui peut correspondre à des modifications de textes, mais en conservant la même



(a) Multi-langage

(b) PHP

(c) JS

(d) HTML

FIGURE 5.2 Distribution des taux de duplication des kits

structure de page web. Dans les deux cas, cela suggère que les kits peuvent changer de contenu HTML, de manière supposée à imiter d'autres sites, tout en conservant potentiellement le même code PHP. Le raisonnement inverse est également valable : la grande quantité de kits avec clones identiques suggère aussi que de nombreux kits utilisent le même *template*, mais avec potentiellement des modifications dans d'autres langages (comme PHP) pour communiquer les résultats aux différents attaquants. Pour JS en revanche, les kits avec uniquement des clones paramétriques sont négligeables. Une petite quantité de kits sans clones identiques ou paramétriques existe pour Javascript, que cela corresponde à des combinaisons uniques de bibliothèques ou à du code JS unique écrit par les auteurs de kits.

On remarque notamment que la distribution des taux de duplication de kits sont très proches entre PHP (figure 5.2b) et les trois langages combinés (figure 5.2a). Cela laisse penser que PHP est le langage principal des kits de *phishing* étudiés ici. D'autres considérations décrites

dans la section 5.5 vont également dans ce sens.

Puisque que les deux distributions sont similaires, les remarques suivantes sur les résultats multi-langages s'appliquent également aux résultats de PHP.

Une faible portion des kits étudiés ont des clones identiques. Puisque tous les kits du jeu de données sont uniques, cela signifie que des modifications ne sont pas capturées par l'analyse des langages considérés : ces modifications peuvent être faites dans d'autres langages ignorés par l'analyse. L'analyse se focalisant uniquement sur le contenu du code source, et non sur les noms de fichiers ou dossiers, les modifications sur l'arborescence des fichiers sont également ignorées. Cela est également valable pour les permutations des fonctions ou méthodes dans un fichier PHP ou JS, puisque les kits sont considérés comme des ensembles de fragments, ignorant ainsi l'ordre.

Plus de la moitié des kits ont des clones paramétriques. Cela montre la haute similarité des kits de *phishing*, qui sont souvent dotés du même code source, avec un certain nombre de paramètres modifiés. On peut se retrouver par exemple avec deux kits imitant le même site web, mais avec des attaquants différents qui ont chacun modifié les moyens de récupérer les informations volées (adresse *email*, contenu de l'*email*, etc.). Le cas dual peut aussi se présenter, où deux kits imitent deux sites différents, mais envoie le résultat de l'attaque à la même personne

Enfin, quasiment un tiers des kits n'ont pas de clone identique ou paramétrique. Ils peuvent très bien être similaires à d'autres kits (par exemple en ajoutant simplement une nouvelle fonctionnalité à un kit), ou être radicalement différents (par exemple deux kits écrits de manière indépendante). Les expériences sur les généalogies dans les sections suivantes permettent d'en savoir plus sur la similarité de ces kits originaux, puisqu'on ignore les kits paramétriquement dupliqués pour construire ces généalogies.

Langage	Nombre de kits
Multi-langage	9 043
PHP	7 210
JS	3 023
HTML	6 931

Tableau 5.1 Nombre de kits paramétriquement uniques selon le langage

Le tableau 5.1 rapport le nombre de kits paramétriquement distincts selon les analyses mono-langages et selon l'analyse multi-langage. Comme observé avec les taux de duplications, PHP est le langage le moins dupliqué, suivi de HTML puis de JS. Le nombre de kits uniques est

le plus élevé pour l'analyse multi-langage, ce qui est assuré par la définition de la relation d'équivalence multi-langage pour les kits paramétriques.

Type	n	moyenne	é-t.	min	max	méd.
classes de kits identiques	1 110	2,4	1,2	2	17	2
classes de kits paramétriques	2 476	5,2	8,5	2	150	3

Tableau 5.2 Statistiques sur les tailles des classes d'équivalence – Multi-langage

Type	n	moyenne	é-t.	min	max	méd.
classes de kits identiques	1 342	2,4	1,2	2	17	2
classes de kits paramétriques	2 281	6,1	12,6	2	246	3
classes de fragments identiques	112 316	8,1	70,6	2	8 780	3
classes de fragments paramétriques	22 843	6,0	46,0	2	5 023	2

Tableau 5.3 Statistiques sur les tailles des classes d'équivalence – PHP

Type	n	moyenne	é-t.	min	max	méd.
classes de kits identiques	1 188	10,4	44,0	2	1049	3
classes de kits paramétriques	118	2,9	2,1	2	19	2
classes de fragments identiques	660 412	26,9	177,1	2	86 328	5
classes de fragments paramétriques	42 794	4,1	6,9	2	756	2

Tableau 5.4 Statistiques sur les tailles des classes d'équivalence – JS

Les tableaux 5.2, 5.3, 5.4, et 5.5 présentent les statistiques sur les tailles des classes d'équivalences non triviales (avec plus d'un élément) pour respectivement les trois langages combinés, le PHP, le JS, et le HTML. On observe bien que le minimum est toujours 2, puisque les classes de taille 1 ont été retirées. Pour les résultats multi-langages, seules les classes de kits sont calculées, ce qui explique l'absence des lignes de classes de fragments. Les distributions de fréquences associées pour les kits se trouvent en annexe C.

La taille médiane des classes d'équivalence se situent entre 2 et 3, ce qui signifie qu'une grande quantité de fragments (ou kits) sont réutilisés un faible nombre de fois.

Comme constaté précédemment avec les taux de duplications, JS (tableau 5.4) comporte un grand nombre de clones identiques, que ce soit les fragments ou les kits. En effet, les fragments les plus dupliqués le sont de l'ordre de 80 000, tandis qu'on observe le même code JS pour

Type	n	moyenne	é-t.	min	max	méd.
classes de kits identiques	2 365	5,6	11,0	2	223	3
classes de kits paramétriques	830	3,1	2,6	2	27	2
classes de fragments identiques	14 006	14,2	94,2	2	8 783	3
classes de fragments paramétriques	3 793	4,4	23,4	2	1127	2

Tableau 5.5 Statistiques sur les tailles des classes d'équivalence – HTML

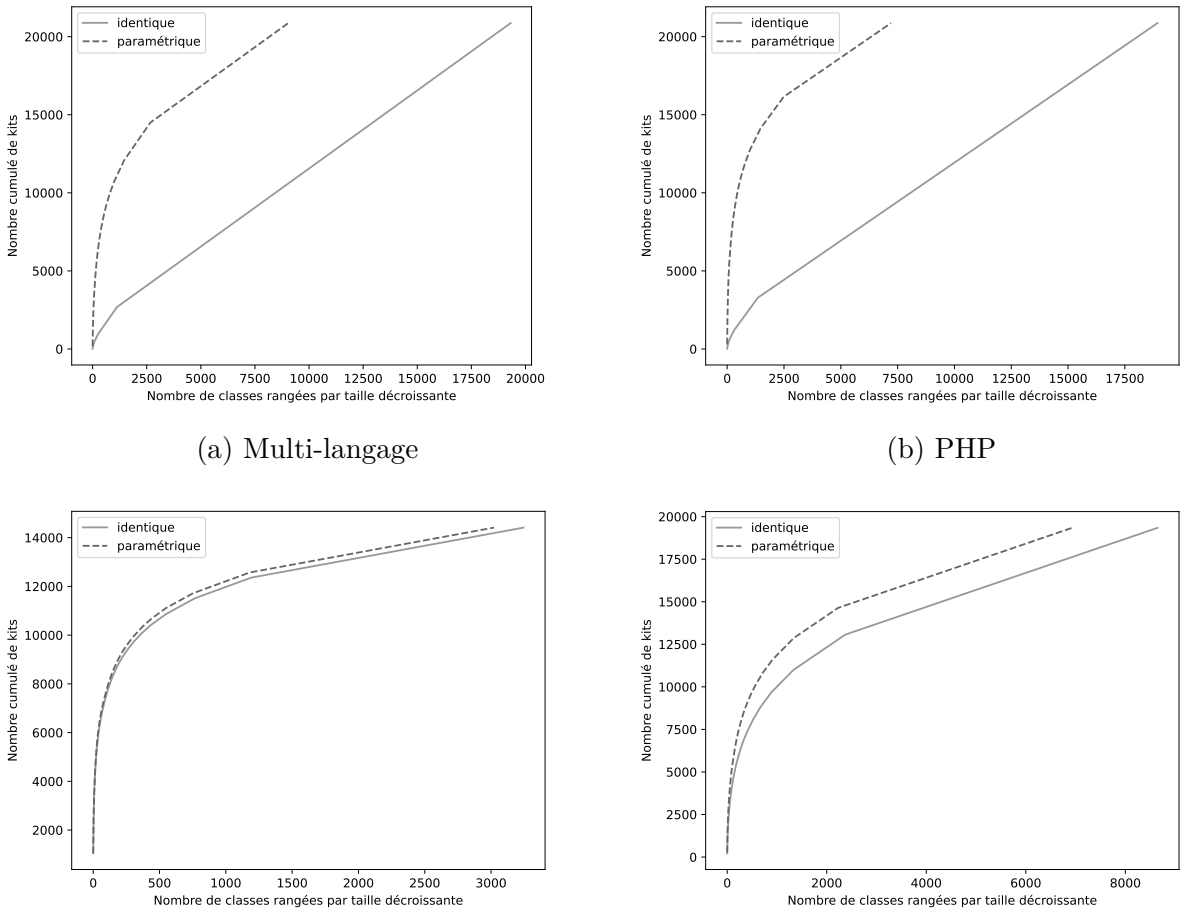
plus de 1 000 kits. Comparativement, les plus grandes classes d'équivalences de fragments ou kits paramétriques sont plus petites d'un facteur d'environ 100.

PHP (tableau 5.3) et HTML (tableau 5.5) présentent des tailles moyennes plus faibles que celle de JS. Le premier présente des classes de kits paramétriques plus grandes que ses classes de kits identique, tandis qu'on observe le phénomène inverse pour HTML. Pour les fragments, ce sont les classes de fragments identique qui sont plus volumineuses que les paramétriques, pour les deux langages.

Finalement, on remarque que les statistiques sur les classes de kit multi-langages sont assez proches de celles de PHP. En partant de PHP, deux kits identiques peuvent rester identiques, devenir paramétriques ou originaux, en ajoutant les fragments d'autres langages. En revanche, deux kits paramétriques ne peuvent que rester paramétriques ou devenir originaux, tandis que des kits originaux en PHP le resteront dans tous les cas. Ainsi, il est logique de constater que les tailles moyennes (et maximales) des classes de kits multi-langages sont inférieures aux classes de PHP, puisqu'on retrouve du PHP dans quasiment tous les kits. Ce constat n'est pas applicable à HTML ou à JS, qui ne se retrouvent que dans un nombre restreint de kits, et forment donc un ensemble de kits non représentatif de tous les kits. Un kit donné n'est dupliqué identiquement qu'un nombre de fois inférieur à 20, alors qu'il peut être dupliqué paramétriquement jusqu'à 150 fois. Cela conforte l'hypothèse que des kits nécessitant seulement de modifier des paramètres sont répandus.

La figure 5.3 montre l'évolution du nombre de kits de manière cumulative en fonction des classes d'équivalences de kits, identiques et paramétriques (au sens large), pour les différents langages. On observe que les courbes de classes de kits identiques sont en dessous de celles des kits paramétrique, puisque les kits identiques forment un sous-ensemble des kits paramétriques (au sens large). Chaque courbe finie par une évolution linéaire de coefficient directeur 1, ce qui correspond aux classes de taille 1 (kits sans clones).

Pour JS (figure 5.3c) et HTML (figure 5.3d), les courbes de kits identiques et de kits paramétriques sont assez proches, ce qui révèle la faible quantité de kits strictement paramétriques



(a) Multi-langage

(b) PHP

(c) JS

(d) HTML

FIGURE 5.3 Taille cumulative des classes d'équivalence de kits, triées par ordre décroissant

pour les deux langages.

Comme observé avec les figures et tableaux précédents, on remarque que le multi-langage (figure 5.3a) et PHP (figure 5.3b) présentent des profils similaires. Pour plus de clarté, on se concentre sur les résultats multi-langages pour la suite de l'analyse. Peu de kits sont identiques, ce qui explique que la courbe des kits identiques est proche d'une ligne droite. En revanche, la courbe des kits paramétriques croît très rapidement avant d'arriver à une croissance linéaire. Cela illustre le principe de Pareto : une majorité des kits (14 000, soit 70%) appartiennent à une minorité des classes de kits paramétriques (2 700, soit 30%), en considérant le point de la courbe qui précède son dernier segment. Si on considère une classe de kits paramétriques comme pouvant identifier un groupe d'auteurs de kits, il semble donc que la plupart des kits étudiés proviennent d'un nombre limité de sources.

5.2 Visualisation des généalogies

Les résultats présentés dans cette section proviennent des expériences décrites dans la section 4.3. Les généalogies excluant les doublons paramétriques (au sens large), ses études permettent d'analyser notamment la similarité des kits sans clone, entre eux et les représentants des classes de kits paramétriques. Les résultats présentés dans les sections suivantes sont extraits de chacune des généalogies : multi-langage, PHP, JS, et HTML.

Les figures 5.4, 5.5, 5.6 et 5.7 représentent respectivement les généalogies, multi-langage, de PHP, de JS, et de HTML. Pour rendre compte de la densité, plusieurs niveaux de zooms sont représentés. Pour rappel, plus deux points sont proches spatialement dans l'un des graphes, plus leurs kits correspondants sont similaires en principe. Également, comme indiqué dans la section 1.3, les généalogies permettent d'étudier la similarité des kits, sans pour autant refléter l'évolution réelle des kits.

On remarque un motif récurrent quelle que soit la généalogie : mis à part quelques kits extrêmement éloignés, la quasi-totalité des kits se situent au centre, très proches les uns des autres. Intuitivement, on peut alors estimer que les kits (non paramétriquement dupliqués) sont pour la plupart très similaires aux autres. Des analyses plus quantitatives dans les sections 5.3 et 5.4 étayent cette position.

Également, on remarque que les généalogies de PHP et du multi-langage sont plus denses, avec un plus grand nombre de nœuds, étant donné que ces deux analyses possèdent le plus grand nombre de kits paramétriquement distincts (voir table 5.1).

Les kits très éloignés correspondent à des cas particuliers, par exemple des fichiers PHP comportant une très grande quantité de jetons dues à des listes initialisées avec des millions d'éléments.

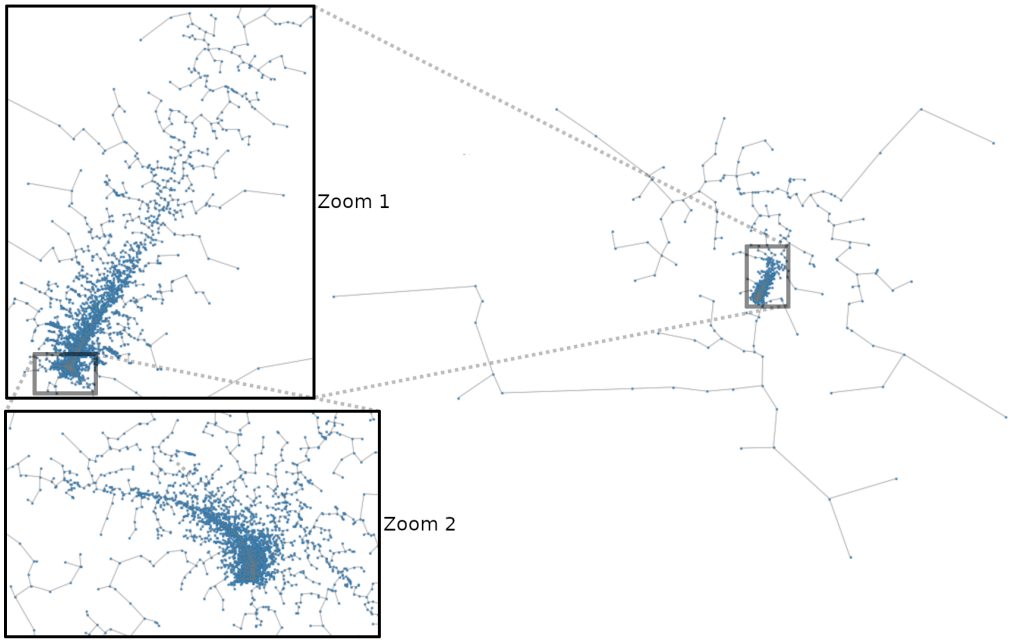


FIGURE 5.4 Représentation graphique de la généalogie multi-langage

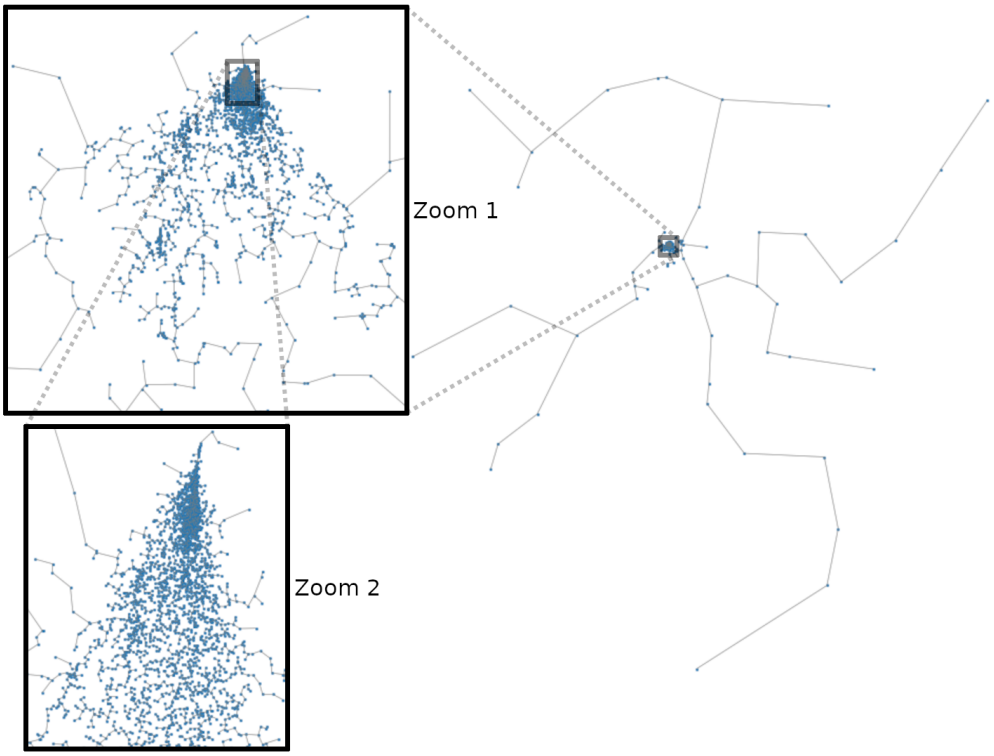


FIGURE 5.5 Représentation graphique de la généalogie PHP

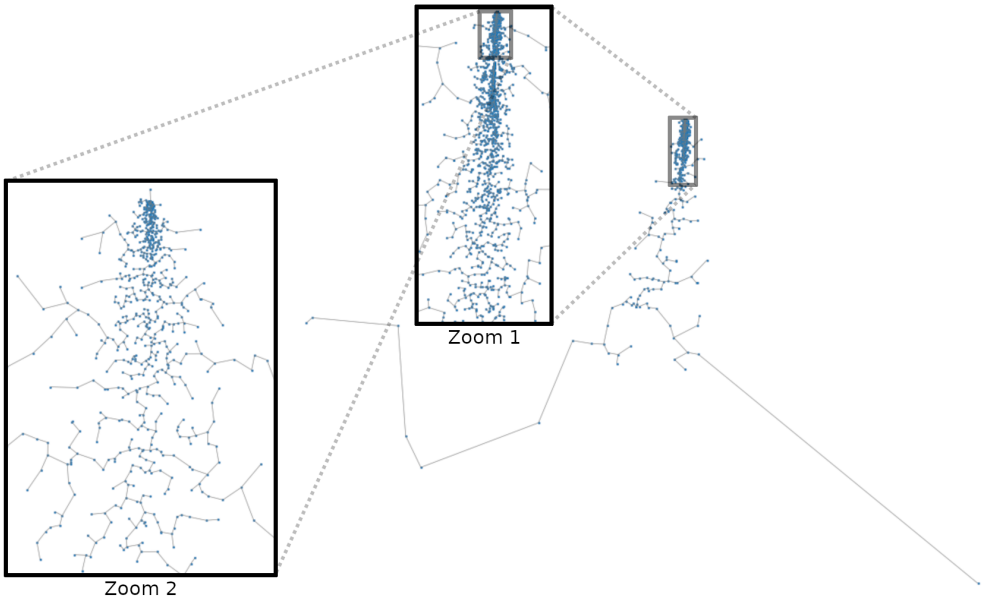


FIGURE 5.6 Représentation graphique de la généalogie JS

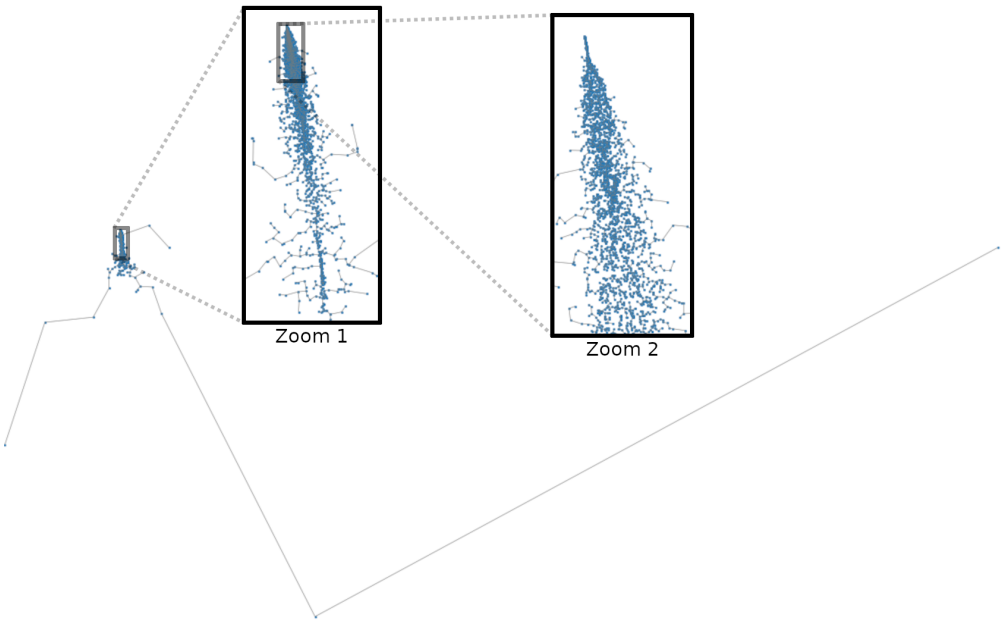


FIGURE 5.7 Représentation graphique de la généalogie HTML

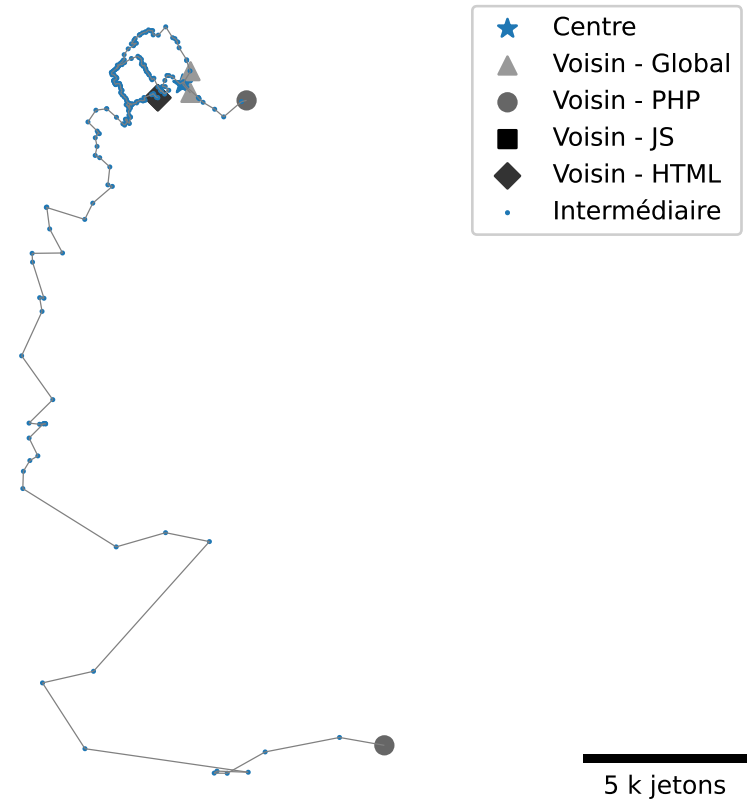


FIGURE 5.8 Visualisation partielle de la généalogie multi-langage

La figure 5.8 représente une partie de la généalogie multi-langage, centrée sur le kit en forme d'étoile. Pour rappel, seule la partie de la généalogie multi-langage reliant le kit central à ses voisins dans les autres généalogies est affichée. Cela a pour avantage d'obtenir une visualisation plus précise, avec le véritable MST. Pour cette raison, une échelle est indiquée en bas à droite de la figure. Ici, elle indique une distance de 5 000 jetons, tous langages confondus. Le kit « centre » se trouve au haut de la figure, à côté des triangles.

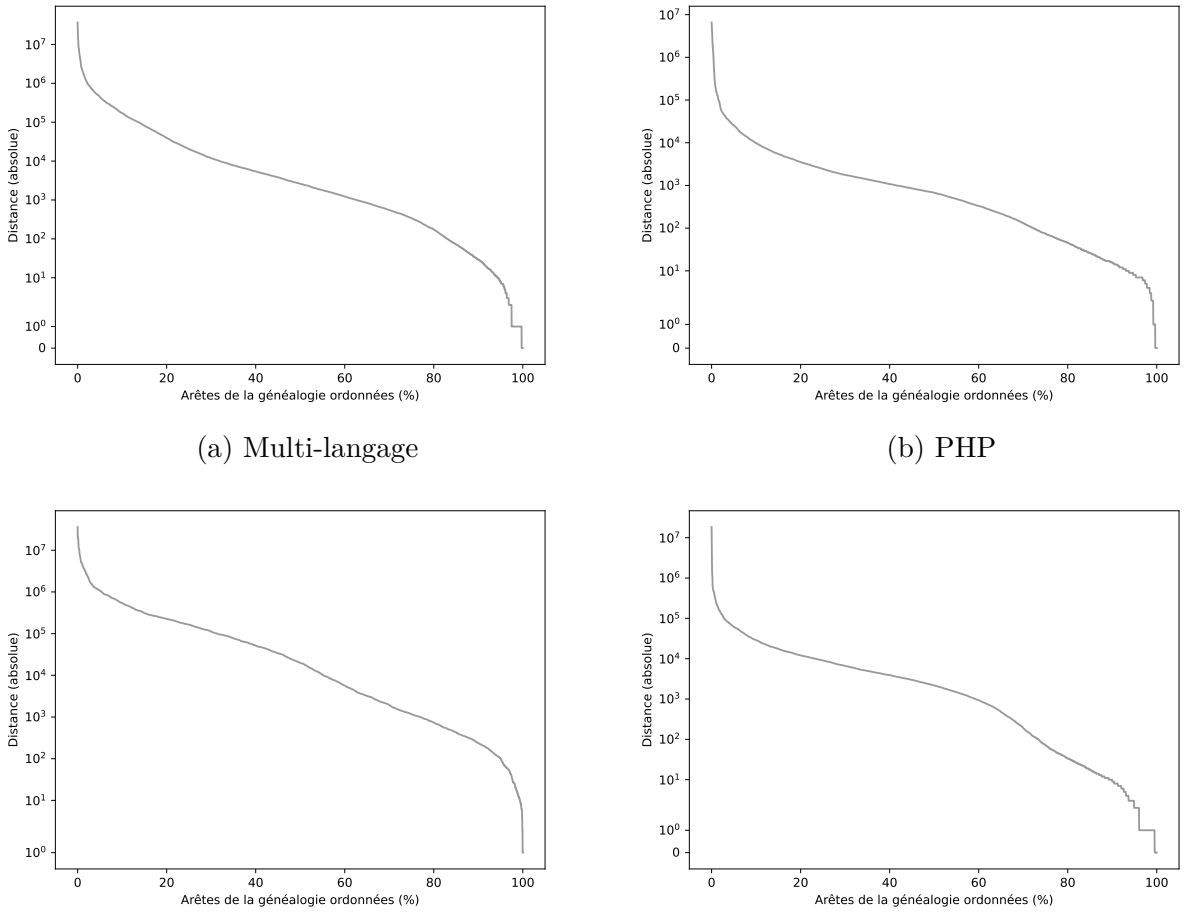
Ici, le kit centré n'a pas de code JS, d'où l'absence de voisins dans ce langage. Le voisin HTML du kit est proche, à un peu moins de 1000 jetons multi-langages. Si tous ces jetons étaient du code PHP consécutif, cela correspondrait à un peu moins de 100 LOC, ce qui est une modification assez faible à l'échelle d'un kit. Pour PHP, on retrouve deux voisins de généalogie : l'un est proche à environ 2500 jetons, mais l'autre est bien plus éloigné, à plus de 20 000 jetons. Ainsi, même si deux kits peuvent être proches dans la généalogie PHP, ils peuvent être proches ou éloignés dans la généalogie multi-langage.

Dans cet exemple, il s'agit de PHP qui semble avoir une évolution indépendante du multi-

langage, mais le même constat peut être observé pour d'autres langages avec d'autres kits centraux. Cette approche peut être appliquée à n'importe quel kit, permettant de faire une analyse des kits similaires d'un kit voulu, du point de vue multi-langage, mais également pour n'importe quel langage analysé dans ce travail.

5.3 Similarité le long des généalogies

Les résultats présentés dans cette section proviennent des expériences décrites dans la section 4.4. Les généalogies étudiées excluent les doublons paramétriques (au sens large), ainsi l'analyse de la similarité de leurs kits nous renseigne sur les changements structurels effectués.



(a) Multi-langage

(b) PHP

(c) JS

(d) HTML

FIGURE 5.9 Distances absolues des arêtes des généalogies triées par ordre décroissant

La figure 5.9 montre en échelle logarithmique les distances absolues des généalogies multi-langages, PHP, JS, et HTML, par ordre décroissant.

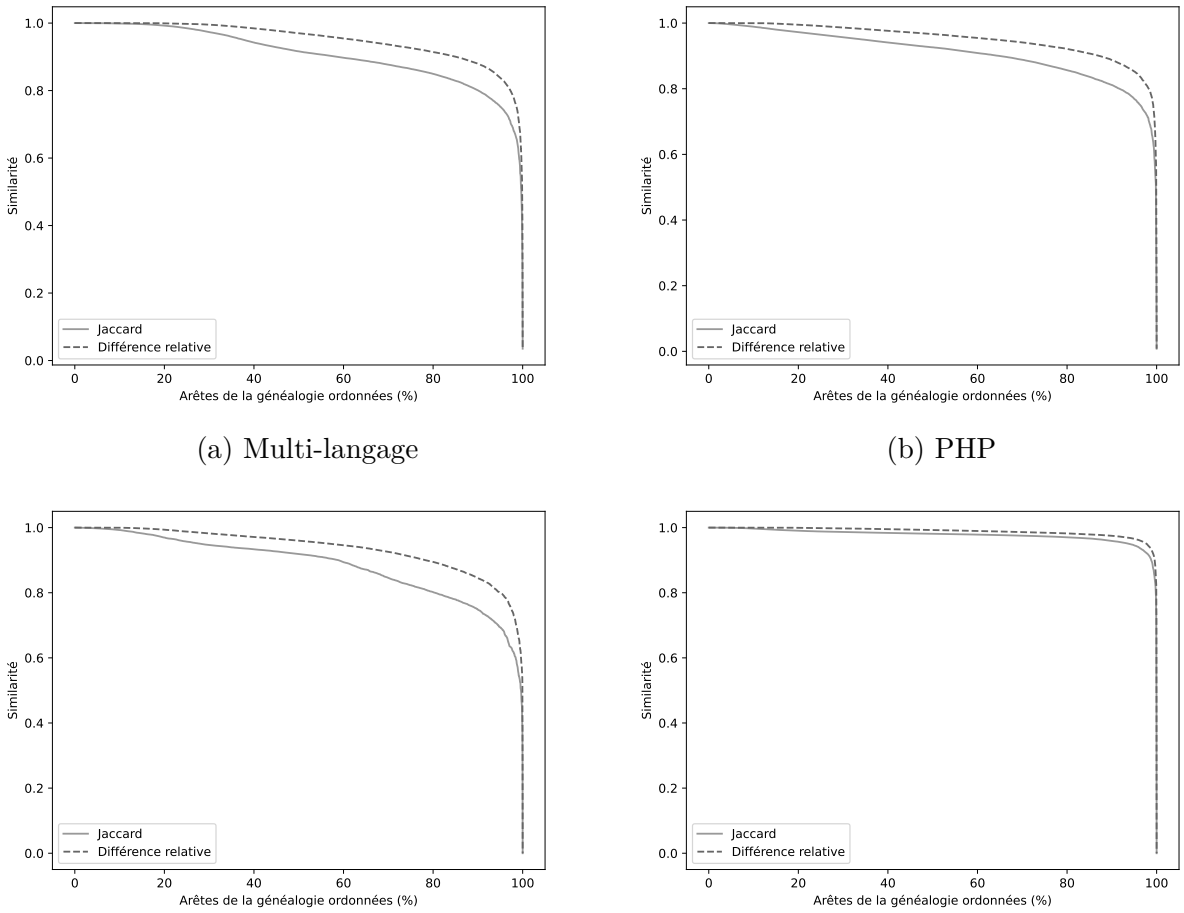
On remarque, quelle que soit la généalogie étudiée, que la courbe de distance présente trois cas de figures : un très faible nombre d'arêtes correspond à des changements très volumineux (à gauche), alors qu'une majorité d'arêtes correspond à des changements modérés (au centre), et enfin les quelques arêtes restantes montrent des distances très petites (à droite). On peut même observer des arêtes de distance nulle dans des généalogies, ce qui correspond à des kits ayant des fragments uniquement identiques, paramétriques, ou similaires mais avec les mêmes vecteurs de métriques. Ce cas de figure est possible sans que des kits soient paramétriques (les doublons étant ignorés dans les généalogies). Par exemple, deux kits avec des fragments ayant les mêmes instructions, mais permutées au sein des fragments, ne seront pas paramétriques mais auront les mêmes vecteurs de métriques. Enfin, pour les distances très élevées peuvent s'expliquer par la présence de kits avec un nombre anormalement grand de jetons. Relié à un kit de taille plus modeste dans la généalogie, on obtient alors une distance très élevée au vu de la différence des tailles des kits. De tels kits ont été observés, avec notamment des fichiers PHP contenant des listes initialisées avec des millions d'éléments.

Dans la généalogie multi-langage, plus de 40 % des arêtes sont associées à une distance inférieure à 1 000 jetons. Si ces jetons étaient du code PHP ou JS consécutif, cela représenterait quelques centaines de LOC. À l'échelle d'un kit, il s'agit de modifications de faible volume, ce qui met en évidence une certaine proximité des kits dans leur code source, même en ignorant les doublons paramétriques.

Pour PHP, la similarité est encore plus prononcée, puisque un peu moins de 60 % des arêtes correspondent à moins de 1 000 jetons, et près de 30 % à moins de 100 jetons, soit seulement quelques dizaines de LOC.

Au contraire, dans le cas de JS la similarité est moins marquée, avec 50 % des kits qui diffèrent de moins de 10 000 jetons. Cela peut être dû à l'utilisation des différentes versions de bibliothèques JS (comme `JQuery`, `Bootstrap`, etc.) enregistrées dans les kits. Elles peuvent induire beaucoup de changements, qui ne sont pourtant pas écrits par les auteurs des kits.

Les distances pour HTML sont semblables à celles du multi-langage. 40 % des arêtes traduisent des modifications de moins de 1 000 jetons. Puisque les jetons HTML peuvent être à la fois des symboles d'un seul caractère ou des morceaux de textes très volumineux, il est plus compliqué d'interpréter cette distance.



(a) Multi-langage

(b) PHP

(c) JS

(d) HTML

FIGURE 5.10 Similarités des kits le long des généalogies triées par ordre décroissant

La figure 5.10 montre les similarités relatives (JC et RD, définis dans la section 4.4) des généalogies multi-langages, PHP, JS, HTML par ordre décroissant. Le coefficient de Jaccard (JC) est toujours inférieur à la différence relative (RD), puisqu'il sous-estime la similarité, tandis que la RD ne normalise la distance que par la taille d'un des deux kits.

Comme pour les distances absolues, on remarque pour toutes les généalogies trois cas : une similarité très proche de 1 (à gauche) pour un faible nombre de kits, puis qui diminue lentement (au centre) avec une majorité de kits, avant d'attendre très rapidement un minimum (à droite) pour les kits restants. La similarité proche de 1 correspond aux kits de tailles similaires à faible distance. Elle descend jusqu'à une valeur très proche de 0 quand une arête relie un kit de petite taille à un kit de très grande taille, comme discuté pour les distances absolues.

Avec la généalogie multi-langage, la RD indique que près de 90% des arêtes relient des kits similaires à plus de 0,9. Pour Jaccard, la valeur de similarité descend à 0,8 pour ces 90% d'arêtes, ce qui reste très élevé. Pour les trois langages considérés, en plus d'avoir un grand nombre de clones paramétriques, les kits présentent également une similarité très élevée le long de la généalogie.

On constate le même ordre de grandeur de similarité pour PHP et JS. JS présente une similarité légèrement plus faible que PHP, avec par exemple 80% des arêtes propagent des kits avec un JC d'environ 0,8, contre approximativement 0,85 pour PHP.

Pour la similarité, HTML fait office de cas particulier, avec des similarités encore plus élevées que celles des autres langages. Par exemple, la RD indique qu'environ 98% des kits partagent 95% de leur code avec leur voisin. La syntaxe de HTML diffère largement de celle de PHP ou de JS, il y a beaucoup moins de types de jetons possibles en HTML. Par conséquence, il est plus simple pour deux codes HTML d'être similaire. De plus, les modifications de textes (significatives en HTML) n'entraînant pas de changement de types de jetons, elles sont ignorées par le calcul de similarité.

RD	Kits similaires	Pourcentage
1	97	1,1 %
0,9	6180	68,3 %
0,8	8270	91,5 %
0,7	8825	97,6 %
0,6	8968	99,2 %

Tableau 5.6 Pourcentage de kits similaires en fonction du seuil – Multi-langage

RD	Kits similaires	Pourcentage
1	113	1,6%
0,9	5 130	71,2%
0,8	6 653	92,3%
0,7	7 071	98,1%
0,6	7 171	99,5%

Tableau 5.7 Pourcentage de kits similaires en fonction du seuil – PHP

RD	Kits similaires	Pourcentage
1	0	0,0 %
0,9	1 925	63,7 %
0,8	2 457	81,3 %
0,7	2 824	93,4 %
0,6	2 947	97,5 %

Tableau 5.8 Pourcentage de kits similaires en fonction du seuil – JS

RD	Kits similaires	Pourcentage
1	127	1,8 %
0,9	6861	99,0 %
0,8	6923	99,9 %
0,7	6926	99,9 %
0,6	6926	99,9 %

Tableau 5.9 Pourcentage de kits similaires en fonction du seuil – HTML

Les tableaux 5.6, 5.7, 5.8, 5.9, rapportent le pourcentage de kits dont le degré de similarité avec au moins un autre kit est supérieur au seuil spécifié, respectivement pour les généalogies des langages combinés, de PHP, de JS, et de HTML.

Par exemple, pour PHP (tableau 5.7), 113 kits (1,6%) ont un voisin au niveau de similarité maximal. Puisqu'on considère uniquement les kits paramétriquement différents, cela indique qu'un faible pourcentage des kits ont le même vecteur de métriques sans être paramétriques. Ce cas a déjà été expliqué lors de la description de la figure 5.9, au début de la section. Seulement quelques kits (1.9%) semblent être hautement différents de tous les autres kits (une RD de moins de 0,7). Cela tend à confirmer l'hypothèse que les auteurs de kits n'écrivent pas leur code PHP de zéro.

Pour JS (tableau 5.8), on observe une tendance similaire à PHP. Peu de kits sont similaires pour un haut seuil, et de plus en plus le deviennent avec des seuils plus faibles. Cependant, la croissance du nombre de kits similaires n'est pas aussi importante que celle de PHP.

Pour HTML (tableau 5.9), très peu de kits de la généalogie ont exactement le même vecteur de fréquence de types de jetons. Cependant, contrairement à PHP ou à HTML, on observe qu'avec le second seuil de RD, presque tous les kits sont similaires à un autre. Puisque les vecteurs de métriques de HTML est plus petit que celui de PHP ou JS, l'espace des modifications de HTML et plus restreint. Ainsi, en HTML deux kits ont plus de chances d'être similaires avec un seuil élevé.

Les similarités multi-langages (tableau 5.6) présentent un profil très proche à celui de PHP, ce qui confirme également que PHP est le langage principal des kits. Pour un seuil de similarité donné, le pourcentage de kits similaires multi-langages est légèrement inférieur à celui des kits similaires en PHP.

La répartition de similarité observée des kits est cohérente avec l'hypothèse sur le modèle de propagation des kits. Souvent, la propagation des kits est basée sur des copies identiques, ou quasi identiques, avec des modifications à faible coût. Les changements à haut coût sont occasionnels, abruptes, et, pour les kits modifiés fructueux, donnent naissance à une prolifération de kits légèrement différents

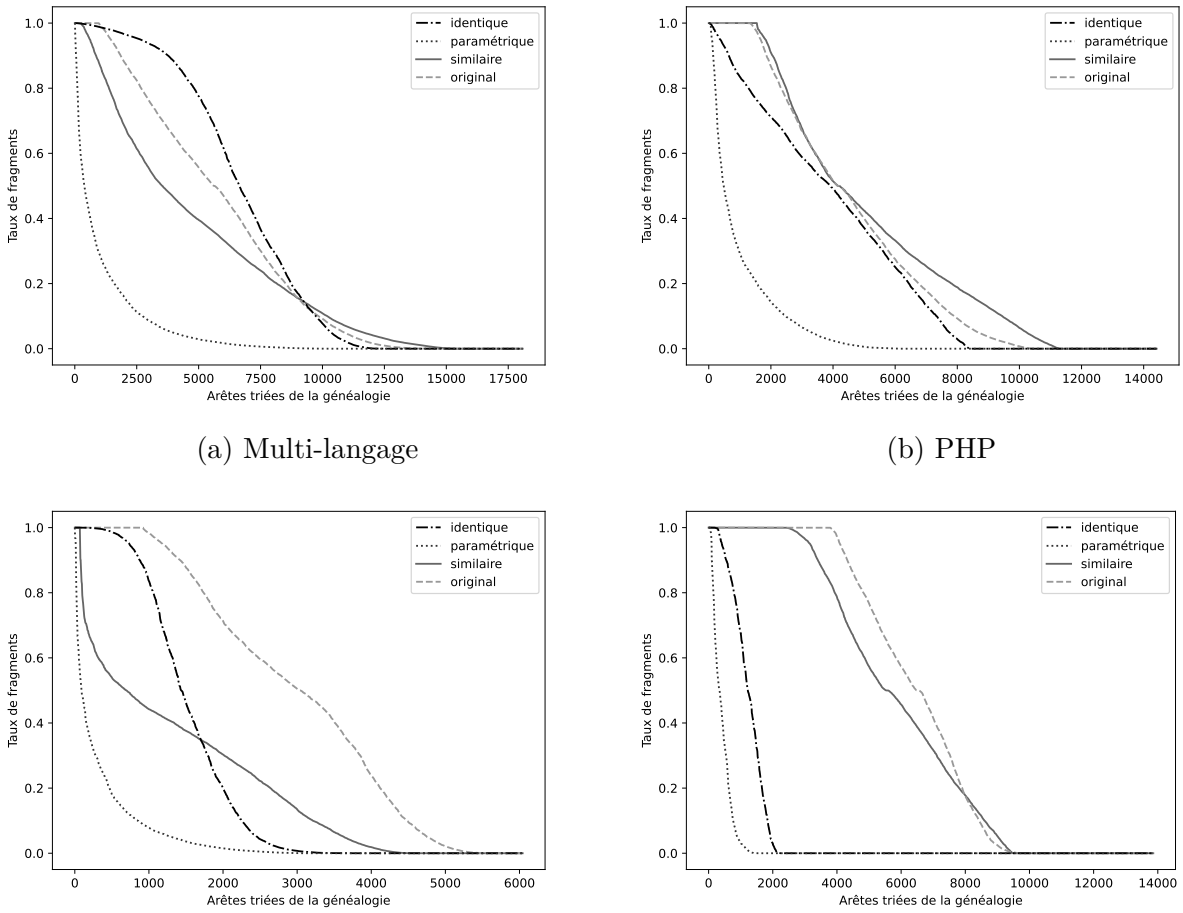
Les changements à haut coût sont divers et peuvent avoir potentiellement comme but le camouflage, la lutte contre la détection, ou l'introduction de changements techniques. Ils peuvent alors servir de base pour d'autres propagations : les changements superficiels identiques ou quasi identiques exploitent la facilité de ré-utilisation des kits fructueux.

5.4 Comparaison de kits

Les résultats présentés dans cette section proviennent des expériences décrites dans la section 4.5.

La figure 5.11 présente les taux de fragments identiques, paramétriques, similaires ou originaux par arêtes, triés par ordre décroissant, pour les généalogies multi-langage, PHP, JS, et HTML. Étant donné que les taux sont normalisés par les tailles des deux kits, l'axe des abscisses comporte le double du nombre réel d'arêtes des généalogies. Quel que soit le type de fragment, on observe pour toutes les généalogies que le taux commence à 1, et fini par atteindre 0. Pour illustre cela, on considère le taux de fragments identiques entre un kit k_1 et k_2, normalisé par la taille de k_1. Un taux de 1 signifie que tous les fragments de k_1 sont, chacun, identiques à au moins un fragment de k_2. Cependant, k_2 présente nécessairement d'autres fragments qui ne sont identiques à aucun des fragments de k_1, puisque les doublons paramétriques (et donc *a fortiori* identiques) ont été éliminés. En revanche, un taux de fragments identiques de 0 signifie qu'aucun fragment de k_1 n'est identique à un fragment de k_2. Dans ce cas, les fragments de k_1 peuvent être paramétriques, similaires, originaux, ou une combinaison de ces trois catégories de fragments.

Pour la généalogie multi-langage, on observe que les fragments identiques ont les taux les plus élevés. Bien que les kits étudiés de la généalogie soient tous distincts paramétriquement (et donc *a fortiori* identiquement), ils partagent quand même une grande quantité de fragments identiques. Les taux de fragments paramétriques est lui bien plus faible. Cela semble logique



(a) Multi-langage

(b) PHP

(c) JS

(d) HTML

FIGURE 5.11 Taille cumulative des classes d'équivalence de kits, triées par ordre décroissant

étant donné que les kits paramétriquement dupliqués sont ignorés : des changements paramétriques peuvent toujours être présents, mais il est plus probable qu'un fragment modifié devienne similaire au fragment original, puisque les changements possibles pour cette dernière catégorie sont plus nombreux. En effet, on constate que les taux de fragments similaires sont supérieurs aux taux de fragments paramétriques. Enfin, les fragments originaux représentent tout de même une part non négligeable des kits de la généalogie, signe que des modifications non triviales sont apportées entre kits.

Assez étonnamment, on constate que pour les autres généalogies, les taux de fragments identiques sont plus faibles que les autres catégories de fragments, contrairement à la généalogie multi-langage. Cela peut s'expliquer par le fait que les généalogies ne sont pas les mêmes, en ayant éliminé les doublons selon le langage. Par exemple, la généalogie de JS ne porte que sur environ 3 000 kits, alors que la généalogie multi-langage en comprend environ 9 000. On

va donc se retrouver avec des doublons de JS sur la généalogie multi-langage, qui n'apparaissaient pas dans la généalogie JS.

Pour PHP, on observe que les fragments identiques, similaires et originaux présentent des taux semblables. Les fragments identiques ont cependant des taux légèrement inférieurs. Les kits semblent pour ce langage également partager un certain nombre de fragments en communs. Similairement à la généalogie multi-langage, les fragments paramétriques ont des taux très faibles. Même si deux kits de la généalogie pouvaient être paramétriques, on s'attendrait quand même à un faible taux de fragments paramétriques. En effet, il suffit qu'un seul fragment paramétrique existe dans chaque kit, et que les autres fragments soient identiques, pour que les kits soient paramétriques. En pratique, les auteurs de kits n'ont qu'à modifier les valeurs d'un nombre restreint de fragments, en laissant la majorité des fragments telle quelle.

JS présente des taux semblables à ceux qu'on observe avec PHP. La principale différence est que les fragments identiques et similaires sont moins nombreux, au contraire des fragments originaux plus nombreux.

En revanche, HTML présente un profil différent des autres langages. Très peu de fragments sont identiques ou paramétriques, la plupart étant originaux ou similaires. Cela suggère que quand le code HTML est modifié, les auteurs ont plus tendance à faire des modifications importantes, par exemple en ré-écrivant entièrement les pages pour imiter d'autres sites. HTML étant un langage avec très peu de types de jeton, prendre en compte uniquement la distance en type de jetons ne permettait pas de conclure à de telles modifications. Le fait d'analyser également les images des jetons pour identifier la quantité de fragments identiques permet de dépasser cette limitation.

Les tableaux 5.10, 5.11, 5.12 et 5.13 rapportent les statistiques sur les tailles cumulées en jetons des fragments, pour chaque arête des généalogies multi-langage, PHP, JS, et HTML, respectivement. Par exemple, si on lit la première ligne du tableau 5.10 (multi-langage), on peut interpréter que, en moyenne, les fragments identiques entre deux kits d'une arête de la généalogie totalisent environ 115 000 jetons.

Les valeurs minimales, quelle que soit la ligne ou le table, valent toutes 0. Cela signifie qu'on trouve toujours une arête où les kits n'ont aucun fragment d'une (ou plusieurs à la fois) catégorie donnée. Les valeurs maximales sont très élevées, ce qui aboutit à des écart-types élevés. Cela a également pour conséquence d'augmenter les moyennes, qui sont très éloignées de leurs médianes. Par la suite, on se concentra sur les valeurs médianes, qui permettent d'ignorer les valeurs trop extrêmes dues à certains kits.

type	moyenne	é-t.	min	med.	max
identique	115 343	698 700	0	857	15 305 024
paramétrique	4 027	22 789	0	67	791 794
similaire	30 926	203 000	0	1 246	7 015 328
original	126 670	896 830	0	2 481	35 641 887

TABLEAU 5.10 Statistiques sur les tailles cumulées des fragments selon leur catégorie par arête – Multi-langage

	moyenne	é-t.	min	med.	max
identique	6 192	60 023	0	128	1 738 541
paramétrique	418	2 679	0	0	96 024
similaire	4 117	53 838	0	290	1 938 982
original	17 583	204 910	0	621	6 314 071

TABLEAU 5.11 Statistiques sur les tailles cumulées des fragments selon leur catégorie par arête – PHP

type	moyenne	é-t.	min	med.	max
identique	142 334	823 069	0	524	14 405 088
paramétrique	9 437	39 847	0	53	796 680
similaire	76 268	302 208	0	2 597	6 946 398
original	271 002	1 242 802	0	19 085	35 118 470

TABLEAU 5.12 Statistiques sur les tailles cumulées des fragments selon leur catégorie par arête – JS

type	moyenne	é-t.	min	med.	max
identique	3 090	17 175	0	0	339 950
paramétrique	370	2 164	0	0	42 498
similaire	4 691	20 380	0	837	753 049
original	22 289	311 522	0	2 010	18 398 132

TABLEAU 5.13 Statistiques sur les tailles cumulées des fragments selon leur catégorie par arête – HTML

Dans la généalogie globale, bien que la moyenne indique que les jetons des fragments identiques sont les plus nombreux, la médiane montre plus de jetons de fragments originaux qu'identiques. L'ajout de fragments originaux se fait donc en de plus grande proportion que les fragments identiques conservés. Les fragments similaires présentent eux une quantité de jetons légèrement supérieur à celles des identiques. Enfin, les fragments paramétriques restent une quantité négligeable de jetons, d'un facteur 10 devant les identiques.

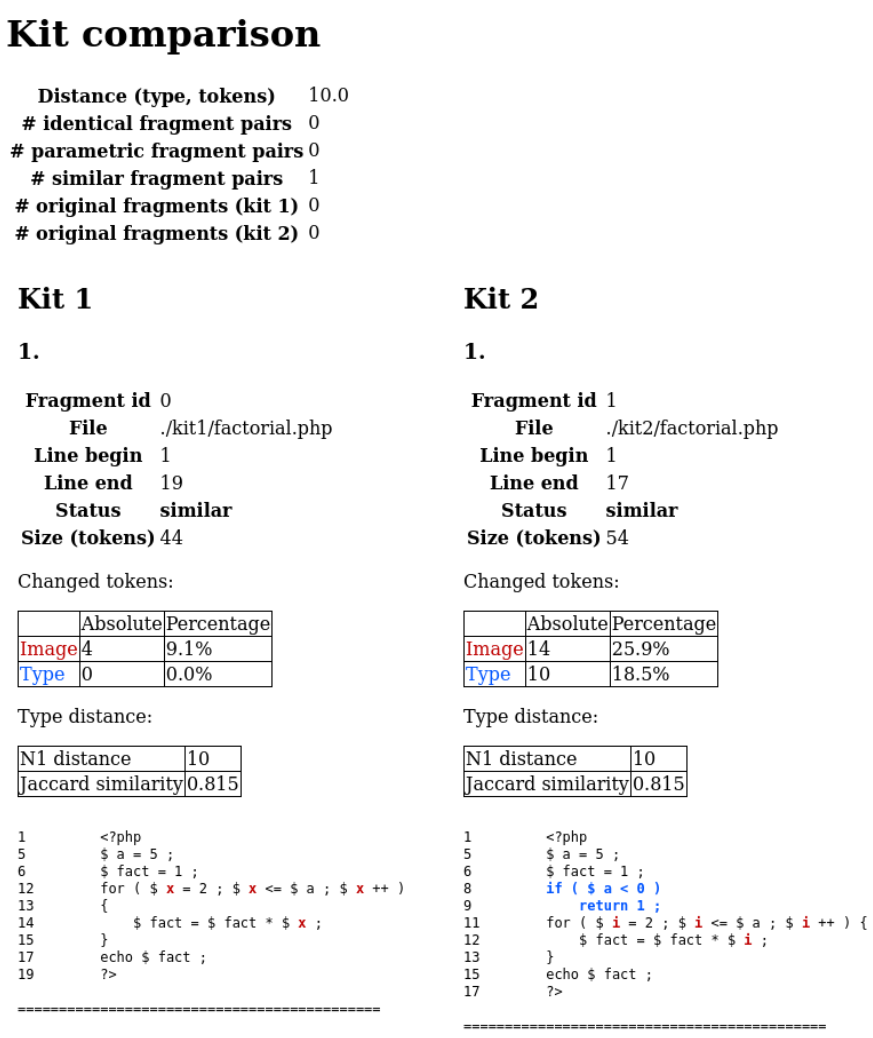
Pour PHP, on observe une tendance similaire, bien que les ordres de grandeur de jetons soient inférieurs. Les fragments paramétriques sont si peu nombreux que l'on constate que la médiane du nombre total de jetons de ces fragments entre deux kits est de 0.

JS a en revanche le même ordre de grandeur de jetons que le multi-langage. Étant donné que les jetons du multi-langage sont la somme des jetons des trois langages, JS est ainsi le langage qui influe les résultats de la comparaison de kits, de par son grand nombre de jetons. Ces grandes valeurs semblent être expliquées avec la présence des libraires JS dans les kits, très volumineuses.

Comme attendu, HTML présente le moins de jetons quelle que soit la catégorie de fragment, étant donné la nature de ses jetons. La médiane vaut 0 jeton pour les fragments identiques et paramétrique : plus de la moitié des kits ne partagent aucun code HTML identique ou paramétrique. Là encore, quand le code HTML est différent entre deux kits, très peu de code HTML est réutilisé. Les modifications sont d'autant plus importantes que la taille totale médiane des fragments originaux est plus de deux fois supérieure à celles des fragments similaires.

La figure 5.12 présente un exemple de l'outil de visualisation de la comparaison de deux kits. Deux kits fictifs en PHP ont été choisis, composés d'un fragment chacun. On peut voir en haut le résumé de la comparaison : la distance en type de jetons, le nombre de fragments selon la catégorie. En dessous, on voit la liste des fragments paramétriques et similaires des deux kits, mis côte-à-côte. Les informations de chaque fragment sont résumées au début du fragment. Le nombre de jetons changés sont indiqués pour les deux catégories : images en rouge (changements paramétriques), et types en bleu (changements structuraux). Enfin, les codes source des deux fragments sont affichés, en colorant les jetons qui changent d'image ou de type. En nuances de gris, le bleu apparaît gris clair, et le rouge gris foncé.

On voit dans cet exemple que la distance est de 10 ici, car le kit 2 ajoute une branche conditionnelle contenant 10 jetons. Les deux fragments de deux kits forment une paire de fragments similaires, ils sont affichés côte-à-côte. Deux modifications ont été faites pour passer du kit 1 au kit 2 : l'ajout d'une condition avec un *if*, en bleu, et le renommage de la variable local `x` en `i`, en rouge.



Kit comparison

Distance (type, tokens)	10.0
# identical fragment pairs	0
# parametric fragment pairs	0
# similar fragment pairs	1
# original fragments (kit 1)	0
# original fragments (kit 2)	0

Kit 1

1.

Fragment id	0
File	./kit1/factorial.php
Line begin	1
Line end	19
Status	**similar**
Size (tokens)	44

Changed tokens:

	Absolute	Percentage
Image	4	9.1%
Type	0	0.0%

Type distance:

N1 distance	10
Jaccard similarity	0.815

```
1       <?php
5       $ a = 5 ;
6       $ fact = 1 ;
12      for ( $ x = 2 ; $ x <= $ a ; $ x ++ )
13      {
14          $ fact = $ fact * $ x ;
15      }
17      echo $ fact ;
19      ?>
```

==========

Kit 2

1.

Fragment id	1
File	./kit2/factorial.php
Line begin	1
Line end	17
Status	**similar**
Size (tokens)	54

Changed tokens:

	Absolute	Percentage
Image	14	25.9%
Type	10	18.5%

Type distance:

N1 distance	10
Jaccard similarity	0.815

```
1       <?php
5       $ a = 5 ;
6       $ fact = 1 ;
8       if ( $ a < 0 )
9           return 1 ;
11      for ( $ i = 2 ; $ i <= $ a ; $ i ++ ) {
12          $ fact = $ fact * $ i ;
13      }
15      echo $ fact ;
17      ?>
```

==========

FIGURE 5.12 Exemple de visualisation de différences entre kits

Ainsi, il est possible de visualiser de cette manière la différence entre n'importe quels kits, pour le langage désiré parmi PHP, JS et HTML.

5.5 Corrélation entre langages

Les résultats présentés dans cette section proviennent des expériences décrites dans la section 4.6.

Le tableau 5.14 rapporte les résultats de la corrélation des distances multi-langages, PHP, JS, HTML sur la généalogie multi-langage. Pour chaque case du tableau, la p-valeur associée est très faible ($p < 2,2e-16$). Ainsi, on peut dire que les résultats de la matrice sont significatifs ($p < 0,05$), sans avoir besoin d'ajuster la p-valeur pour les comparaisons multiples.

	multi	php	js	html
multi	1	0.8437092	0.4980513	0.3926228
php	0.8437092	1	0.3525423	0.1363826
js	0.4980513	0.3525423	1	0.3951109
html	0.3926228	0.1363826	0.3951109	1

Tableau 5.14 Corrélation des distances sur la généalogie multi-langage

Comme constaté dans les sections 5.1 et 5.3, PHP est le langage principal des kits. En effet, la corrélation entre les distances PHP et multi-langage est très élevée : les chances de prédire la similarité entre deux kits en regardant uniquement la similarité du code PHP de ces kits sont grandes. En revanche, la corrélation entre le multi-langage et JS ou HTML est plus modérée.

Pour la corrélation entre langage, il est intéressant de noter que la corrélation entre PHP et HTML est assez faible. Les distances de PHP ne semblent pas beaucoup corréler avec celles de HTML : cela suggère que le code PHP est souvent modifié indépendamment du code HTML. On peut par exemple imaginer que le contenu des pages web est modifié (HTML), tandis que le code pour envoyer les informations récupérées est conservé (PHP).

5.6 WordPress

L'expérience pour valider la manière de générer les généalogies à l'aide de WordPress est présentée dans la section 4.7. Aucune version n'a de clone identique ou paramétrique, puisque chaque version apporte un grand nombre de modifications, qui n'ont quasiment aucune chance d'être uniquement paramétriques.

La généalogie multi-langage de WordPress est donnée dans la figure 5.13, en utilisant la technique de visualisation décrite dans la section 4.3. Il faut toutefois noter que les arêtes visibles sur la figure sont les véritables arêtes de la généalogie, et non une généalogie reconstruite pour la visualisation comme c'est le cas pour les kits. Cela est possible puisque le nombre de versions de WordPress est bien plus faible que le nombre de kits, ce qui donne des résultats plus précis avec MDS.

On observe que la généalogie est cohérente avec l'évolution de WordPress : elle forme quasiment une ligne droite, puisque on étudie un jeu de données homogène, composé d'uniquement un logiciel avec différentes versions. On remarque notamment que toutes les versions mineures (du style `x.y.0`) sont correctement ordonnées, partant de la version `2.0.0` en haut en descendant jusqu'à la version `4.8.0`. Notons l'absence de la version `2.4.0`, qui n'a jamais été

publiée.

Également, on remarque que les versions ne sont pas équitablement espacées dans l'espace, mais agglutinées en certains points. Lorsqu'une version mineure (`x.y.0`) de WordPress sort, on a par la suite plusieurs versions de correctif (`x.y.z`) qui sortent, très similaires à la version mineure. Puisque des versions similaires sont représentées proches les unes des autres avec cette représentation, cela a pour conséquence de regrouper les versions de correctif d'une même version mineure sur le graphe.

Enfin, la comparaison avec l'historique des versions WordPress tel que décrit par le système de gestion de versions, nous indique que 73% des arêtes de la généalogie prédite apparaissent également dans la véritable généalogie, sachant que les deux généalogies ont le même nombre d'arêtes. Bien que le pourcentage est élevé, il reste éloigné de 100%. Dans les faits, certaines versions de correctif sont mal ordonnées, tandis que les versions mineures et majeures sont toutes correctement ordonnées. Cela a pour conséquence de réduire le pourcentage d'arêtes correctes. Même si les versions de correctif ne sont pas correctement ordonnées du point de vue chronologique, l'impact en termes de similarité reste faible pour la généalogie prédite. En effet, le but n'est pas de trouver une généalogie exacte, mais plutôt une généalogie plausible du point de vue de la similarité des versions.

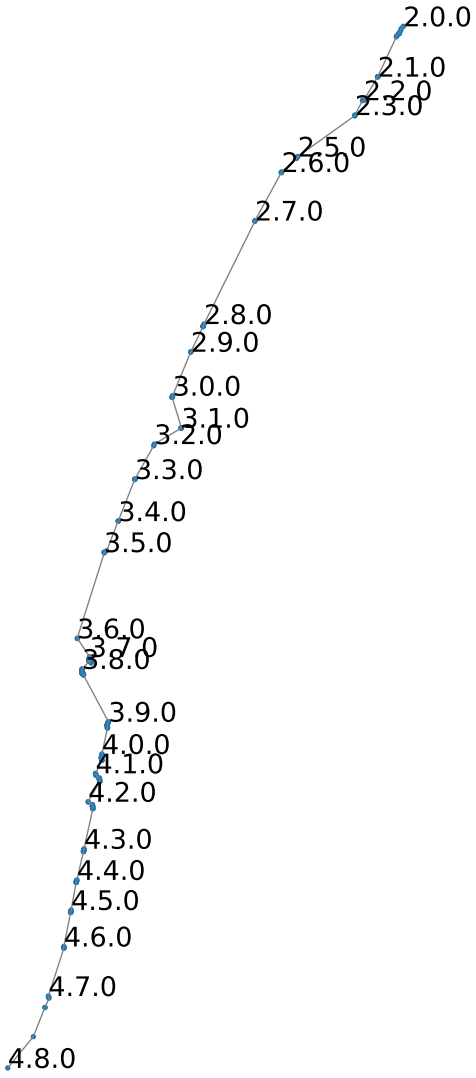


FIGURE 5.13 Visualisation de la généalogie multi-langage de WordPress

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Les plus de 20 000 kits de *phishing* fournis par IBM, totalisant plus de 180 millions de lignes de code, ont été analysés par une analyse statique basée sur la fréquence des types de jetons en PHP, JS, HTML afin d'étudier la similarité de leurs codes source.

Tout d'abord, l'étude de leurs taux de duplication identique et paramétrique a montré que plus de la moitié des kits ont des clones paramétriques, c'est-à-dire des modifications n'impliquant que la substitution de constantes, d'identificateurs, etc. Par ailleurs, la grande majorité des kits appartiennent à une minorité de groupes de kits paramétriques. Ensuite, pour chaque langage une généalogie a été générée rendant compte de la similarité des kits paramétriquement distincts. Ces généalogies ont été combinées pour construire une généalogie multi-langage, qui traduit une évolution plausible des kits en considérant les trois langages simultanément, répondant ainsi à la question de recherche. Pour confirmer la vraisemblance des généalogies, le même procédé a été appliqué à WordPress : la généalogie multi-langage obtenue est jugée satisfaisante. L'analyse des généalogies des kits a permis de constater que les kits paramétriquement distincts partagent néanmoins une grande similarité, avec près de 90% des kits partageant 90% de leurs jetons pour PHP notamment. Ces résultats concordent avec l'hypothèse de recherche de minimisation des efforts de la part des auteurs de kits. Enfin, la corrélation des distances indique que PHP est le langage qui rend le plus compte de l'évolution des kits.

Ce travail permet de déterminer quelles sont les classes de kits de *phishing* les plus utilisées, afin de rationaliser les efforts de contre-mesures. Également, l'étude de similarité facilite l'identification de nouveaux kits, en les comparant à des kits déjà connus, qu'ils soient identiques, paramétriques, ou à une certaine distance de kits déjà connus.

6.2 Limitations de la solution proposée

Plusieurs limitations s'appliquent aux résultats obtenus :

— biais de sélection : la similarité observée peut être en partie due à la manière dont les kits sont collectés, qui ne représentent sûrement pas un échantillon représentatif de tous les kits de *phishing* ;
— dépendance du choix de la distance : les résultats obtenus seront différents avec une

distance autre que celle de Manhattan, bien qu'elle reste une bonne approximation de la distance de Levenshtein ;
— dépendance du choix des parseurs : la définition des jetons pouvant varier d'un parser à l'autre pour un même langage, les résultats obtenus peuvent légèrement différer si on change de parseur ;
— absence d'oracle : bien que les résultats soient cohérents avec WordPress, la véritable évolution des kits est inconnue, on ne peut pas la comparer avec la généalogie multi-langage prédite ;
— obfuscation du code : certaines parties du code étant obfusquées, l'analyse statique ne permet pas de déterminer le code sous-jacent, ce qui peut fausser la similarité.

6.3 Améliorations futures

Plusieurs pistes d'améliorations futures sont envisageables. D'autres langages peuvent être analysés, notamment le CSS qui est présent dans de nombreux kits mais ignoré actuellement. Afin de générer une généalogie, plusieurs manières d'agréger les métriques de fragments d'un kit peuvent être explorées, outre le choix de la somme retenu dans ce travail. D'autres approches peuvent être utilisées pour étudier les kits, comme les analyses de *product families*. L'analyse s'est focalisée un nombre fixé de kits de *phishing*, mais il est possible d'adopter une approche incrémentale permettant l'ajout de nouveaux kits au fur et à mesure de leur collecte. Également, l'ajout de kits provenant de diverses sources permettrait de potentiellement réduire le biais de sélection.

RÉFÉRENCES

[1] Anti-Phishing Working Group. (2021) Phishing activity trends report - 4th quarter 2020. [En ligne]. Disponible : https://docs.apwg.org/reports/apwg_trends_report_q4_2020.pdf

[2] ——. (2020) Phishing activity trends report - 1st quarter 2020. [En ligne]. Disponible : https://docs.apwg.org/reports/apwg_trends_report_q1_2020.pdf

[3] C. K. Roy, J. R. Cordy et R. Koschke, "Comparison and evaluation of code clone detection techniques and tools : a qualitative approach," *Science of Computer Programming*, vol. 74, n°. 7, p. 470–495, mai 2009.

[4] M. Balazinska *et al.*, "Advanced Clone-analysis as a Basis for Object-oriented System Refactoring," dans *Proc. Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 2000, p. 98–107.

[5] C. K. Roy et J. R. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools," dans *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICST '09. IEEE Computer Society, 2009, p. 157–166.

[6] M. Cova, C. Kruegel et G. Vigna, "There Is No Free Phish : An Analysis of" Free" and Live Phishing Kits," dans *2nd Conference on USENIX Workshop on Offensive Technologies (WOOT)*, vol. 8, San Jose, CA, 2008, p. 1–8.

[7] H. Mccalley, B. Wardman et G. Warner, "Analysis of back-doored phishing kits," dans *IFIP International Conference on Digital Forensics*. Springer, 2011, p. 155–168.

[8] EC-Council, *How Strong is your Anti-Phishing Strategy ?*, 2018. [En ligne]. Disponible : https://blog.eccouncil.org/how-strong-is-your-anti-phishing-strategy/

[9] Imperva, *Our Analysis of 1,019 Phishing Kits*, 2018. [En ligne]. Disponible : https://www.imperva.com/blog/our-analysis-of-1019-phishing-kits/

[10] A. Oest *et al.*, "Inside a phisher's mind : Understanding the anti-phishing ecosystem through phishing kit analysis," dans *2018 APWG Symposium on Electronic Crime Research (eCrime)*, mai 2018, p. 1–12, iSSN : 2159-1245.

[11] PhishLabs, *How to Fight Back against Phishing*, 2013. [En ligne]. Disponible : https://info.phishlabs.com/hs-fs/hub/326665/file-558105945-pdf/White_Papers/How_to_Fight_Back_Against_Phishing_-_White_Paper.pdf

[12] K. Thomas *et al.*, "Data breaches, phishing, or malware ? : Understanding the risks of stolen credentials," dans *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, p. 1421–1434.

[13] S. Zawoad *et al.*, "Phish-Net : Investigating phish clusters using drop email addresses," dans *2013 APWG eCrime Researchers Summit*, sept. 2013, p. 1–13.

[14] X. Han, N. Kheir et D. Balzarotti, "Phisheye : Live monitoring of sandboxed phishing kits," dans *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, p. 1402–1413.

[15] P. Zhang *et al.*, "CrawlPhish : Large-scale Analysis of Client-side Cloaking Techniques in Phishing," dans *42nd IEEE Symposium on Security & Privacy*, 2021.

[16] D. Manky, "Cybercrime as a service : A very modern business," *Computer Fraud & Security*, vol. 2013, p. 9–13, 2013.

[17] M. Chandrasekaran, K. Narayanan et S. Upadhyaya, "Phishing email detection based on structural properties," dans *NYS cyber security conference*, vol. 3. Albany, New York, 2006.

[18] I. Fette, N. Sadeh et A. Tomasic, "Learning to detect phishing emails," dans *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, p. 649–656.

[19] D. Miyamoto, H. Hazeyama et Y. Kadobayashi, "An evaluation of machine learning-based methods for detection of phishing sites," dans *International Conference on Neural Information Processing*. Springer, 2008, p. 539–546.

[20] I. R. A. Hamid et J. Abawajy, "Hybrid feature selection for phishing email detection," dans *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2011, p. 266–275.

[21] R. Verma, N. Shashidhar et N. Hossain, "Detecting phishing emails the natural language way," dans *European Symposium on Research in Computer Security*. Springer, 2012, p. 824–841.

[22] G. Stringhini et O. Thonnard, "That ain't you : Blocking spearphishing through behavioral modelling," dans *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, p. 78–97.

[23] Y. Zhang, J. Hong et C. Lorrie, "Cantina : a content-based approach to detecting phishing web sites," dans *Proceedings of the 16th International Conference on World Wide Web*, Banff, AB, 2007, p. 639–648.

[24] A. P. E. Rosiello *et al.*, "A layout-similarity-based approach for detecting phishing pages," dans *Proceedings of the 3rd International Conference on Security and Privacy in Communication Networks, SecureComm*, Nice, 2007, p. 454–463.

[25] T.-C. Chen, S. Dick et J. Miller, “Detecting Visually Similar Web Pages : Application to Phishing Detection,” *ACM Trans. Internet Technol.*, vol. 10, n°. 2, p. 5 :1–5 :38, juin 2010, place : New York, NY, USA Publisher : ACM.

[26] S. Afroz et R. Greenstadt, “Phishzoo : Detecting phishing websites by looking at them,” dans *Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on*. IEEE, 2011, p. 368–375.

[27] W. Liu *et al.*, “Antiphishing through Phishing Target Discovery,” *IEEE Internet Computing*, vol. 16, n°. 2, p. 52–61, 2012.

[28] E. H. Chang *et al.*, “Phishing detection via identification of website identity,” dans *2013 International Conference on IT Convergence and Security, ICITCS 2013*. IEEE, 2013, p. 1–4.

[29] G.-G. Geng *et al.*, “Favicon - a clue to phishing sites detection,” dans *eCrime Researchers Summit (eCRS), 2013*, sept. 2013, p. 1–10.

[30] G. Ramesh, I. Krishnamurthi et K. S. S. Kumar, “An efficacious method for detecting phishing webpages through target domain identification,” *Decision Support Systems*, vol. 61, n°. 1, p. 12–22, 2014.

[31] Q. Cui *et al.*, “Tracking Phishing Attacks Over Time,” dans *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, p. 667–676.

[32] M. Kim *et al.*, “An Empirical Study of Code Clone Genealogies,” dans *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2005.

[33] M. W. Godfrey, “Understanding Software Artifact Provenance,” *Sci. Comput. Program.*, vol. 97, n°. P1, p. 86–90, janv. 2015, place : USA Publisher : Elsevier North-Holland, Inc. [En ligne]. Disponible : https://doi.org/10.1016/j.scico.2013.11.021

[34] L. Barbour, F. Khomh et Y. Zou, “Late propagation in software clones,” dans *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, p. 273–282.

[35] R. Saha, C. Roy et K. Schneider, “gCad : A Near-Miss Clone Genealogy Extractor to Support Clone Evolution Analysis,” dans *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2013, p. 488–491.

[36] M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEE*, vol. 68, n°. 9, p. 1060–1076, 1980.

[37] K. Rekouche, “Early phishing,” *arXiv preprint arXiv :1106.4692*, 2011. [En ligne]. Disponible : http://arxiv.org/abs/1106.4692

[38] F. Cryptography. (2005) Gp4.3 - growth and fraud - case #3 - phishing. [En ligne]. Disponible : https://financialcryptography.com/mt/archives/000609.html

[39] K. Sangani, "The battle against identity theft," The Banker, p. 53–54, sept. 2003.

[40] I. Management. (2004, déc.) In 2005, organized crime will back phishers. [En ligne]. Disponible : https://www.webcitation.org/5w9YVyMYn?url=http://itmanagement.earthweb.com/secu/article.php/3451501

[41] A. K. Sood et R. J. Enbody, "Crimeware-as-a-service—a survey of commoditized crimeware in the underground market," *International Journal of Critical Infrastructure Protection*, vol. 6, nº. 1, p. 28–38, 2013. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S1874548213000036

[42] A. K. Jain et B. B. Gupta, "A survey of phishing attack techniques, defence mechanisms and open research challenges," *Enterprise Information Systems*, vol. 0, nº. 0, p. 1–39, 2021. [En ligne]. Disponible : https://doi.org/10.1080/17517575.2021.1896786

[43] R. Wash, "How experts detect phishing scam emails," *Proc. ACM Hum.-Comput. Interact.*, vol. 4, nº. CSCW2, oct. 2020. [En ligne]. Disponible : https://doi.org/10.1145/3415231

[44] S. Ducasse, O. Nierstrasz et M. Rieger, "On the Effectiveness of Clone Detection by String Matching," *International Journal on Software Maintenance and Evolution : Research and Practice - Wiley InterScience*, nº. 18, p. 37–58, 2006.

[45] T. Kamiya, S. Kusumoto et K. Inoue, "CCFinder : A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code," dans *IEEE Transactions on Software Engineering*, vol. 28. IEEE Computer Society Press, 2002, p. 654–670, issue : 7.

[46] H. Basit *et al.*, "Efficient Token Based Clone Detection with Flexible Tokenization," dans *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2007.

[47] C. K. Roy et J. R. Cordy, "NICAD : Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," dans *International Conference on Program Comprehension*. IEEE Computer Society Press, 2008, p. 172–181.

[48] N. Göde et R. Koschke, "Incremental Clone Detection," dans *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2009, p. 219–228.

[49] J. Mayrand, C. Leblanc et E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," dans *Proceedings of the International*

Conference on Software Maintenance - IEEE Computer Society Press, Monterey, CA, nov. 1996, p. 244–253.

[50] E. Merlo *et al.*, "Linear Complexity Object-Oriented Similarity for Clone Detection and Software Evolution Analysis," dans *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, 2004, p. 412–416.

[51] E. Merlo et T. Lavoie, "Detection of Structural Redundancy in Clone Relations," Ecole Polytechnique of Montreal, Rapport technique EPM-RT-2009-05, 2009.

[52] I. Baxter *et al.*, "Clone detection using abstract syntax trees." dans *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, 1998, p. 368–377.

[53] R. Tiarks, R. Koschke et R. Falke, "An assessment of type-3 clones as detected by state-of-the-art tools," dans *Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society Press, 2009, p. 67–76.

[54] T. Lavoie et E. Merlo, "Automated Type-3 Clone Oracle Using Levenshtein Metric," dans *IWSC11 Proceedings of the 5th International Workshop on Software Clones*, 2011, p. 25–32.

[55] ——, "An Accurate Estimation of the Levenshtein Distance Using Metric Trees and Manhattan Distance," dans *Proceedings of the 6th International Workshop on Software Clones*, ser. IWSC '12. New York, NY, USA : ACM, 2012, p. 1–7. [En ligne]. Disponible : http://doi.acm.org/10.1145/1985404.1985411

[56] P. Ciaccia, M. Patella et P. Zezula, "M-tree : An efficient access method for similarity search in metric spaces," dans *Proc. of 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers, 1997, p. 426–435.

[57] D. Rattan, R. Bhatia et M. Singh, "Software clone detection : A systematic review," *Information and Software Technology*, vol. 55, n°. 7, p. 1165 – 1199, 2013. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S0950584913000323

[58] C. K. Roy et J. R. Cordy, "A Survey on Software Clone Detection Research," School of Computing, Queen's University, Rapport technique Technical Report 2007-541, nov. 2007.

[59] A. Sæbjørnsen *et al.*, "Detecting Code Clones in Binary Executables," dans *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA : Association for Computing Machinery, 2009, p. 117–128, event-place : Chicago, IL, USA. [En ligne]. Disponible : https://doi.org/10.1145/1572272.1572287

[60] W. M. Khoo, A. Mycroft et R. J. Anderson, “Rendezvous : a search engine for binary code,” dans *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, San Francisco, CA, USA, May 18-19, 2013*. IEEE Computer Society, 2013, p. 329–338. [En ligne]. Disponible : https://doi.org/10.1109/MSR.2013.6624046

[61] M. Egele *et al.*, “Blanket Execution : Dynamic Similarity Testing for Program Binaries and Components,” dans *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14. USA : USENIX Association, 2014, p. 303–317, event-place : San Diego, CA.

[62] J. Ming, D. Xu et D. Wu, “Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference,” dans *ICT Systems Security and Privacy Protection*, H. Federrath et D. Gollmann, édit. Cham : Springer International Publishing, 2015, p. 416–430.

[63] ——, “Malware Hunt : Semantics-Based Malware Diffing Speedup by Normalized Basic Block Memoization,” *Journal of computer virology and hacking techniques*, août 2017. [En ligne]. Disponible : http://par.nsf.gov/biblio/10066920

[64] M. Lindorfer *et al.*, “Lines of Malicious Code : Insights into the Malicious Software Industry,” dans *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC ’12. New York, NY, USA : Association for Computing Machinery, 2012, p. 349–358, event-place : Orlando, Florida, USA. [En ligne]. Disponible : https://doi.org/10.1145/2420950.2421001

[65] D. Wang *et al.*, “A malware similarity analysis method based on network control structure graph,” dans *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, Oct 2020, p. 295–300.

[66] S. Venkatraman, M. Alazab et R. Vinayakumar, “A hybrid deep learning image-based analysis for effective malware detection,” *Journal of Information Security and Applications*, vol. 47, p. 377–389, Aug 2019.

[67] J. Jang, M. Woo et D. Brumley, “Towards Automatic Software Lineage Inference,” dans *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC’13. USA : USENIX Association, 2013, p. 81–96, event-place : Washington, D.C.

[68] T. Dumitraş et I. Neamtiu, “Experimental Challenges in Cyber Security : A Story of Provenance and Lineage for Malware,” dans *Proceedings of the 4th Conference on Cyber Security Experimentation and Test*, ser. CSET’11. USA : USENIX Association, 2011, p. 9, event-place : San Francisco, CA.

[69] W. M. Khoo et P. Lio’, “Unity in Diversity : Phylogenetic-inspired Techniques for Reverse Engineering and Detection of Malware Families,” dans *First SysSec Workshop,*

SysSec@DIMVA, Amsterdam, The Netherlands, July 6, 2011. IEEE, 2011, p. 3–10. [En ligne]. Disponible : http://doi.ieeecomputersociety.org/10.1109/SysSec.2011.24

[70] A. Gupta *et al.*, "An Empirical Study of Malware Evolution," dans *Proceedings of the First International Conference on COMmunication Systems And NETworks*, ser. COMSNETS'09. IEEE Press, 2009, p. 356–365, event-place : Bangalore, India.

[71] C. Darmetko, S. Jilcott et J. Everett, "Inferring Accurate Histories of Malware Evolution from Structural Evidence," dans *Proceedings of the Florida Artificial Intelligence Research Society Conference*, 2013. [En ligne]. Disponible : https://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS13/paper/view/5884/6082

[72] G. Canfora *et al.*, "How I Met Your Mother ? - An Empirical Study about Android Malware Phylogenesis," dans *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4 : SECRYPT, Lisbon, Portugal, July 26-28, 2016*. SciTePress, 2016, p. 310–317. [En ligne]. Disponible : https://doi.org/10.5220/0005968103100317

[73] A. Cimitile *et al.*, "Model Checking for Mobile Android Malware Evolution," dans *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2017, p. 24–30.

[74] C. Soto-Valero et M. González, "Empirical study of malware diversity in major android markets," *Journal of Cyber Security Technology*, vol. 2, n°. 2, p. 51–74, 2018. [En ligne]. Disponible : https://doi.org/10.1080/23742917.2018.1483876

[75] S. G. Sireci, "Review of modern multidimensional scaling : Theory and applications," *Journal of Educational Measurement*, vol. 40, n°. 3, p. 277–280, 2003.

[76] F. Pedregosa *et al.*, "Scikit-learn : Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, p. 2825–2830, 2011.

ANNEXE A ALGORITHME DE COMPARAISON DE KITS PAR COUVERTURE STRICTE

```
// la couverture stricte est recherchée
// les clones internes sont pris en considération
fonction JUSTIFIERCOUVERTURE(kit1, kit2, vecteursMétrique)
    kitDist ← 0
    fragsCouvertsKit2 ← {}
    fragsNonCouvertsKit1, fragsNonCouvertsKit2 ← {}, {}
    couvertureMap1, couvertureMap2 ← {}, {}
    mtree ← kit2.mtree // l'arbre métrique du kit 2, contenant les vecteur de métriques de ses fragments

    pour tout f1 ∈ kit1 faire // on considère tous les fragments du kit 1
        // Calcul des paramètres de la range query de l'arbre métrique
        rayon ← seuil * cardinalitéJetons(f1)
        ensembleRQ ← mtree.rangeQuery(f1, rayon)
        minDist, minF2 ← +∞, UNDEF_VAL // UNDEF_VAL signifie valeur indéfinie
        pour tout f2 ∈ ensembleRQ faire
            si f2 ∉ fragsCouvertsKit2 alors
                // En pratique fournie par l'arbre métrique, pas calculé
                dist ← l1(vecteursMétrique(f1), vecteursMétrique(f2))
                // La distance est en dessous du seuil pour les deux fragments
                si dist < seuil * cardinalitéJetons(f2) alors
                    // Priorité d'appariement : identique, paramétrique, similaire
                    si identique(f1, f2) alors
                        si non identique(f1, minF2) alors
                            minDist, minF2 ← 0, f2
                        fin si
                    sinon si paramétrique(f1, f2) alors
                        si non paramétrique(f1, minF2) alors // identique implique paramétrique
                            minDist, minF2 ← 0, f2
                        fin si
                    sinon si dist < minDist alors // similaire
```

```
                        minDist, minF2 ← dist, f2
                    fin si
                fin si
            fin si// fragment non couvert
        fin pour// boucle de la range query
        si minF2 ≠ UNDEF_VAL alors
            couvertureMap1[f1] ← (f2, dist)
            couvertureMap2[f2] ← (f1, dist)
            kitDist ← kitDist + dist
        sinon
            fragsNonCouvertsKit1.ajouter(f1)
            kitDist ← kitDist + cardinalitéJetons(f1)
        fin si
    fin pour// f1 dans kit1

    pour tout f2 ∈ kit1 faire
        si f2 ∉ dom(couvertureMap2) alors
            fragsNonCouvertsKit2.ajouter(f2)
            kitDist ← kitDist + cardinalitéJetons(f2)
        fin si
    fin pour
    renvoie kitDist, couvertureMap1, couvertureMap2, fragsNonCouvertsKit1, fragsNon-
CouvertsKit2
fin fonction

kitDist, couvertureMap1, couvertureMap2, fragsNonCouvertsKit1, fragsNonCouvertsKit2
← justifierCouverture(kit1, kit2, vecteursMétrique)
pour tout f1 ∈ dom(couvertureMap1) faire
    f2, dist ← couvertureMap1[f1]
    si dist ≤ 0 alors
        si identique(f1, f2) alors
            // fragments identiques f1, f2
        sinon si paramétrique(f1, f2) alors
            // fragments paramétriques f1, f2
        sinon
            // fragments similaires f1, f2, avec les mêmes vecteurs de métriques
```

```
            fin si
        sinon
            // fragments similaires f1, f2, avec des vecteurs de métriques différents
        fin si
    fin pour
pour tout f1 ∈ fragsNonCouvertsKit1 faire
    // fragment original f1
fin pour
pour tout f2 ∈ fragsNonCouvertsKit2 faire
    // fragment original f2
fin pour
```

ANNEXE B ALGORITHME DE COMPARAISON DE KITS PAR COUVERTURE RELÂCHÉE

```
1: fonction CouvertureRelachee(kit1, kit2)
2:     // asymétrique, il faut appeler cette fonction deux fois par paire
3:     distance ← 0
4:     pour tout f1 ∈ kit1 faire
5:         f2 ← le fragment du kit2 le plus proche de f1
6:         si f2 existe alors
7:             si f1 et f2 sont identiques ou paramétriques alors
8:                 marquer f1 et f2 comme tels
9:                 // distance inchangée, puisque d(f1, f2) = 0
10:            sinon si d(f1, f2) < α|f1| et d(f1, f2) < α|f2| alors
11:                marquer f1 et f2 comme similaires
12:                distance ← distance + d(f1, f2)
13:            sinon
14:                marquer f1 comme original, distance ← distance + |f1|
15:            fin si
16:        sinon
17:            marquer f1 comme original, distance ← distance + |f1|
18:        fin si
19:    fin pour
20:    renvoie distance, relations des fragments
21: fin fonction
```

FIGURE B.1 Algorithme simplifié de la comparaison de kits par couverture relâchée

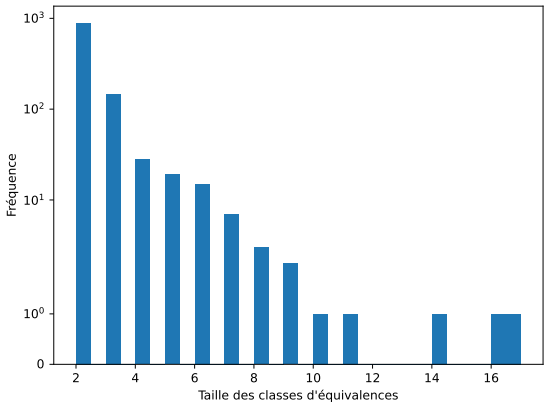
Algorithme détaillé :

```
// les clones internes sont plus ou moins pris en compte, pas de couverture stricte
// un fragment dans le kit 1 peut être similaire à de nombreux fragments du kit 2
fonction compSim(kit1, kit2, vecteursMétrique)
    kitDist ← 0
5:  couvertureMap ← {}
    mtree ← kit2.mtree // l'arbre métrique du kit 2, contenant les vecteur de métriques de ses fragments

    pour tout f1 ∈ kit1 faire
        // Calcul des paramètres de la range query de l'arbre métrique
10:     rayon ← seuil * cardinalitéJetons(f1)
```

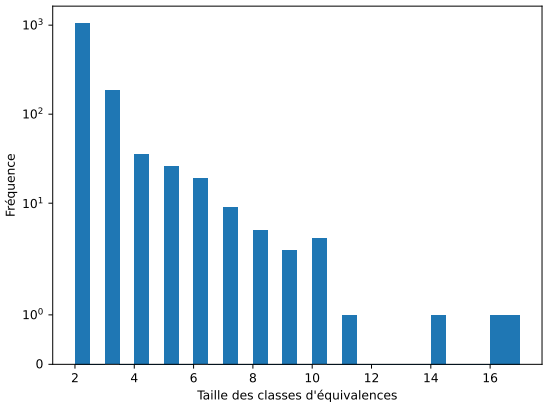
```
        ensembleRQ ← mtree.rangeQuery(f1, rayon)
        minDist, minF2 ← +∞, UNDEF_VAL // UNDEF_VAL signifie valeur indéfinie
        pour tout f2 ∈ ensembleRQ faire
            // En pratique fournie par l'arbre métrique, pas calculé
15:         dist ← l1(vecteursMétrique(f1), vecteursMétrique(f2))
            // La distance est en dessous du seuil pour les deux fragments
            si dist < seuil * cardinalitéJetons(f2) alors
                // Priorité d'appariement : identique, paramétrique, similaire
                si identique(f1, f2) alors
20:                 si non identique(f1, minF2) alors
                        minDist, minF2 ← 0, f2
                    fin si
                sinon si paramétrique(f1, f2) alors
                    si non paramétrique(f1, minF2) alors // identique implique paramétrique
25:                     minDist, minF2 ← 0, f2
                    fin si
                sinon si dist < minDist alors // similaire
                    minDist, minF2 ← dist, f2
                fin si
30:         fin si
        fin pour// boucle de la range query
        si minF2 ≠ UNDEF_VAL alors
            couvertureMap[f1] ← (f2, dist)
            kitDist ← kitDist + dist
35:     sinon
            kitDist ← kitDist + cardinalitéJetons(f1)
        fin si
    fin pour// f1 dans le kit 1

40: renvoie kitDist, couvertureMap
fin fonction

fonction justifierCouverture(kit, couvertureMap)
    pour tout f1 ∈ kit faire
45:     si f1 ∈ dom(couvertureMap) alors
```

```
            f2, dist ← couvertureMap[f1]
            si dist ≤ 0 alors
                si identique(f1, f2) alors
                    // fragments identiques f1, f2
50:             sinon si paramétrique(f1, f2) alors
                    // fragments paramétriques f1, f2
                sinon
                    // fragments similaires f1, f2, avec les mêmes vecteurs de métriques
                fin si
55:         sinon
                // fragments similaires f1, f2, avec des vecteurs de métriques différents
            fin si
        sinon
            // fragment original f1
60:     fin si
    fin pour
fin fonction

fonction justifierSim(kit1, kit2, vecteursMétrique)
65:     kitDist1, couvertureMap1 ← compSim(kit1, kit2, vecteursMétrique)
    justifierCouverture(kit1, couvertureMap1)
    kitDist2, couvertureMap2 ← compSim(kit2, kit1, vecteursMétrique)
    justifierCouverture(kit2, couvertureMap2)

70:     renvoie kitDist1, kitDist2, couvertureMap1, couvertureMap2
fin fonction

justifierSim(kit1, kit2, vecteursMétrique)
```

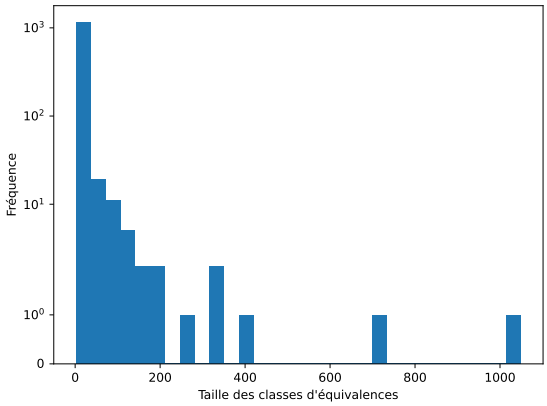
ANNEXE C DISTRIBUTION DES FRÉQUENCES DE TAILLES DE CLASSES D'ÉQUIVALENCE DE KITS



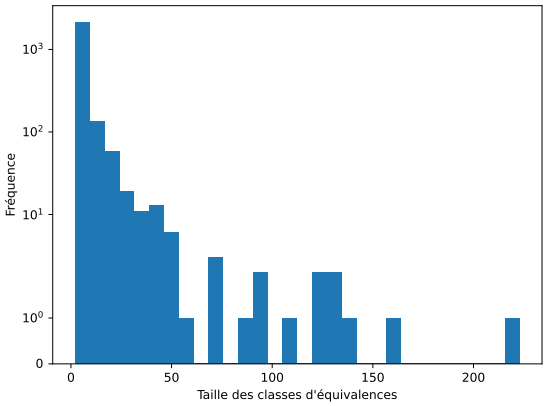
(a) Multi-langage



(b) PHP

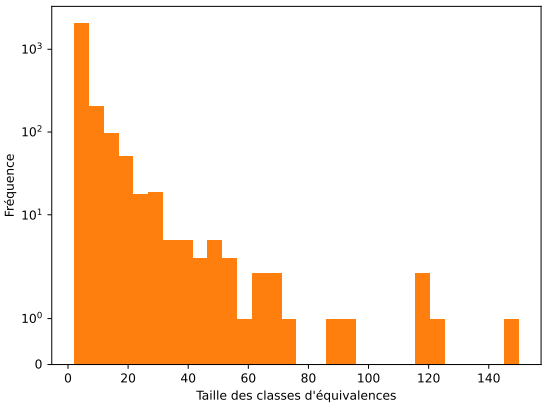


(c) JS

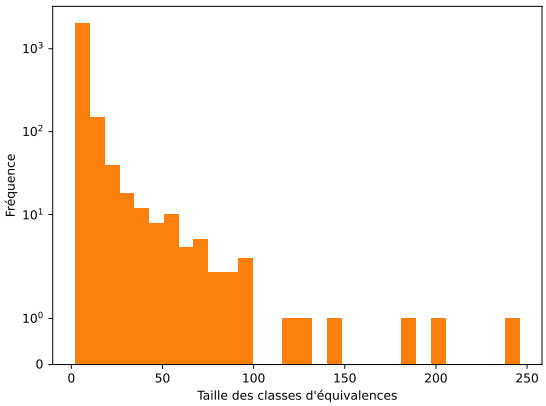


(d) HTML

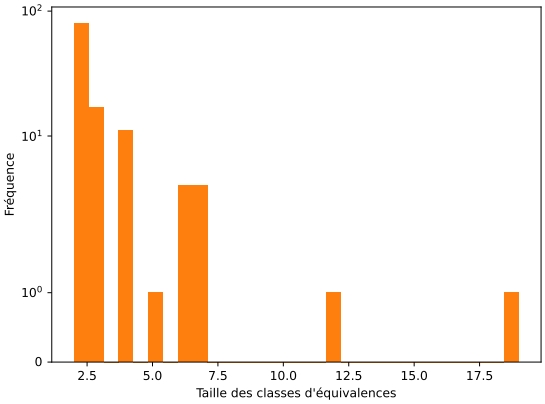
FIGURE C.1 Histogrammes des tailles de classes de kits identiques



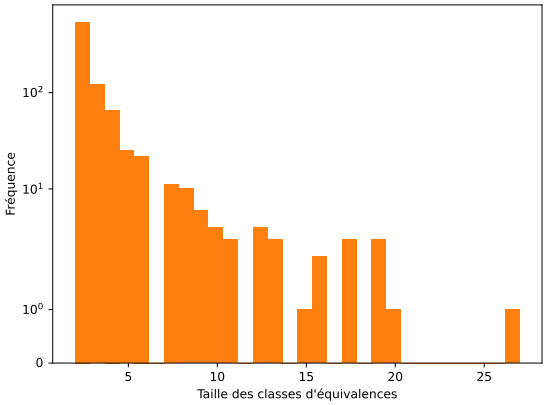
(a) Multi-langage



(b) PHP



(c) JS



(d) HTML

FIGURE C.2 Histogrammes des tailles de classes de kits paramétriques