Originality is the art of concealing your source.

Myself

Correspondances approchées et obfuscation

Sommaire

Si les clones exacts sont les plus aisés à mettre en évidence, dans la pratique des opérations d'édition sont généralement réalisées entre plusieurs exemplaires de clones. Ces éditions interviennent aussi bien dans un cadre de copie légitime afin d'adapter le code copié à son nouveau contexte que dans un cadre de copie illégitime où les modifications sont potentiellement plus importantes afin de limiter l'efficacité d'outils de détection de similarité. Nous nous intéressons ici principalement aux modifications liées à une volonté d'obfuscation. Nous présentons les opérations de modification de code source les plus fréquentes accompagnées de quelques idées et pistes pouvant les contrecarrer. Un tableau récapitulatif de ces différentes opérations et leurs caractéristiques principales est présenté en figure 4.3 ; des exemples de code obfusqué sont réunis en figure 4.4 en fin de chapitre.

Obfuscation et... obfuscation Il existe deux types principaux de processus obfuscatoire selon l'objectif recherché. Le premier consiste à introduire des transformations dans le code source afin de rendre celui-ci inintelligible pour un humain : il s'agit de rendre les opérations de rétro-ingénierie plus difficiles. Ce type d'obfuscation peut également aller de pair avec une volonté d'optimiser le code. Il peut se caractériser notamment par la suppression des commentaires, le renommage de variables en noms non signifiants, un usage spécifique d'opérations de préprocesseur (pour les langages en disposant) ou le développement de fonctions ou structures de contrôle permettant une optimisation [107, 109, 113]. La détection d'une telle obfuscation est triviale pour un juge humain ; à première vue l'utilisation de certaines métriques [111] pourrait permettre une détection automatisée. Le second type d'obfuscation, auquel nous nous intéressons, concerne le camouflage d'une opération de copie illégitime de code. L'objectif est alors non pas de rendre le code inintelligible, mais de limiter l'efficacité d'une recherche de similitudes par un humain ou un outil automatisé.

Filigranage de code Le filigranage du code (en anglais *watermarking*) est un procédé de modification du code source afin d'y introduire des marqueurs témoignant de l'origine de celui-ci. Le filigranage permet ainsi de retrouver un morceau de code copié suffisamment significatif sans avoir à disposer d'une base de codes de référence. Un procédé de filigranage idéal devrait permettre la reconnaissance de ces marqueurs malgré des opérations d'obfuscation ultérieure sur le code ne dénaturant pas sa sémantique, avec la connaissance publique de l'algorithme de filigranage et un couple clé privé/clé publique permettant respectivement d'ajouter le filigranage et de vérifier son existence. De nombreux travaux ont été réalisés pour le filigranage de fichiers multimédias [126, 133]. Cependant ces fichiers sont finalement destinés à une interprétation humaine et supportent ainsi des procédés destructifs. Dans le cadre de code sources, la sémantique doit être conservée afin que l'exécution du programme ne soit pas affectée. Ainsi, un filigranage automatisé ne peut être réalisé que par des modifications de formatage ou par des modifications structurelles sémantiquement neutres. Une méthode envisageable serait alors le camouflage d'une chaîne de caractères indiquant l'origine par un procédé stéganographique. Un copieur de code attentif peut toutefois systématiser des opérations de normalisation de code afin d'effacer tout filigranage.

4.1 Modifications lexicalement et syntaxiquement neutres

4.1.1 Modifications de formatage

Les opérations d'édition concernant le formatage du code peuvent intervenir aussi bien dans le cadre de copie légitime qu'illégitime. Elles consistent principalement à ajouter ou supprimer des caractères non-significatifs pour l'analyse lexicale tels que, pour la plupart des langages[1] des retours à la ligne, des tabulations ou des espaces. Si certaines métriques se référant au style de programmation peuvent considérer le formatage (style d'indentation, de retours à la ligne, nombre moyen de caractères par ligne...), aucun outil de recherche de similitude ne les utilise. Cependant les méthodes de recherche de similitudes par recherche purement textuelle sans analyse lexicale préalable sont sensibles à des modifications de formatage.

4.1.2 Édition de commentaires

L'édition de commentaires accompagne généralement des copies illégitimes. Il s'agit soit d'ajouter de nouveaux commentaires, soit (pratique plus fréquente) de supprimer des commentaires ou de les modifier. La suppression peut être réalisée systématiquement ou partiellement à l'aide d'une analyse lexicale. La réécriture de commentaires peut être réalisée manuellement par un humain ou alors automatiquement, par exemple en remplaçant certains termes par des synonymes à l'aide d'un corpus approprié. Aucun outil de recherche de similitudes intègre à notre connaissance une recherche de similarité en langue naturelle sur des commentaires : seul le code source utile pour la compilation est considéré. La suppression de commentaires étant une opération simple, il est possible d'attendre d'un plagiaire qu'elle soit réalisée : cependant si celle-ci n'est pas menée, une similarité sur un commentaire est un indice important sur la similarité du code environnant.

4.1.3 Renommage d'identificateurs

Obfuscation Le renommage d'identificateurs est peu fréquent dans un cadre légitime. Un juge humain chargé de rechercher des similitudes possédant généralement une mémoire assez sensible aux noms de variables utilisées, un plagiaire cherchera souvent à les modifier. Le renommage d'identificateurs est alors réalisable manuellement ou systématisable automatiquement. Si le renommage systématique de variables est sans risque à une échelle locale, cette opération est plus délicate pour des membres (variables, méthodes...) d'accessibilité large d'unité de compilation ou de classes, ceux-ci pouvant être évoqués en interne par des mécanismes dynamiques de réflexion ou par du code extérieur non connu. D'autre part, un obfuscateur automatique devra prendre soin de réaliser le renommage avec des termes humainement réalistes (lettres uniques, conversion des noms entre un style *under_score* et *dromaDaire*...).

Parade Une solution pour la détection de similitudes malgré le renommage consiste à abstraire totalement le nom des identificateurs. Un tel procédé peut toutefois augmenter les risques de report de faux-positifs sur des petites portions de code. Une attitude intermédiaire consiste à attribuer des noms normalisés aux variables identiques d'une expression, instruction ou unité

[1] Pour certains (rares) langages, le formatage du code source possède cependant une signification syntaxique. Appartenant à cette catégorie de langage, nous pouvons citer Ruby, Python ainsi que le (plus exotique) langage Whitespace (`http://compsoc.dur.ac.uk/whitespace/`).

syntaxique plus vaste. Par exemple l'instruction `i = i + j + 1` pourra être considérée comme `ID = ID + ID + const` ou plus précisément `ID1 = ID1 + ID2 + const`.

4.2 Transposition de code

4.2.1 Ordre de sous-unités structurelles

Étant donnée une unité structurelle du code source composée d'une séquence de sous-unités $E = e_1, e_2, \cdots, e_n$ (par exemple le corps d'une fonction composé de n instructions), nous déterminons l'ordre imposé sur chaque paire de sous-éléments (e_i, e_j) : soit aucun ordre n'est imposé (les sous-éléments sont indépendants), soit e_j dépend de e_i et doit donc être placé après e_i, soit il s'agit du contraire. La dépendance entre sous-unités peut être obtenue à l'aide d'un graphe de dépendances tel que décrit en 3.5. Il peut donc exister des possibilités de déplacer des sous-unités de l'unité structurelle tout en conservant les relations d'ordre imposées par les dépendances.

4.2.2 Obfuscation par transposition

Obfuscation Le code source peut être obfusqué par transposition en déterminant les relations d'ordre entre sous-unités et en réalisant des déplacements valides selon ces relations. Cela permet donc de conserver la sémantique du code tout en désorganisant sa structure. Ainsi pour la plupart des langages modernes n'accordant pas d'importance à la position des fonctions déclarées ou implantées, il est possible de modifier l'ordre des fonctions. À l'échelle locale, un graphe de dépendances de variables permet de déterminer les déplacements possibles.

Résistance à l'obfuscation Le graphe de dépendances d'une unité étant insensible aux opérations de déplacement de code sémantiquement neutres, il peut être utilisé pour rechercher des similitudes malgré ce type d'obfuscation par détermination de sous-graphes homomorphes.

Pour les méthodes de recherche de similarité basées sur d'autres représentations, il est nécessaire de considérer le volume de code transposé ainsi que la fréquence de transposition. Dans des cas classiques, le volume d'un morceau de code transposé est assez conséquent pour pouvoir être individuellement détecté, en dehors de son avant et après contexte d'origine, eux aussi assez volumineux pour être détectés. Ainsi, des correspondances sont relevées individuellement : celles-ci peuvent être consolidées, malgré leur désordre en utilisant par exemple une méthode d'extension (cf chapitre 11). Lorsque le morceau dupliqué n'est pas suffisamment volumineux pour être détecté, l'avant et après contexte peuvent être reportés individuellement, ou être consolidés en une unique correspondance en utilisant un algorithme d'alignement local (qui signalera le morceau déplacé en tant que morceau supprimé). Quant aux méthodes d'alignement globales, celles-ci doivent être proscrites si des opérations d'édition par transposition de code volumineux sont attendues.

4.3 Insertion et suppression de code inutile

L'insertion et la suppression de code inutile au sein de code copié limite la détection de similarité par des algorithmes de recherches d'arbres ou séquences de lexèmes exactement identiques. Ainsi des portions intactes sont détectées mais séparées par des portions de code sans correspondance que ce soit pour l'exemplaire original (code supprimé) ou pour l'exemplaire

copié (code ajouté). Néanmoins, des algorithmes d'alignement local de séquences peuvent être utilisés car tolérant des zones non concordantes dans les correspondances. Concernant les méthodes d'indexation de code source par génération et sélection d'empreintes (décrites au chapitre 8), elles imposent un seuil minimal de lexèmes identiques t pour le report possible de similitudes : en automatisant l'insertion de code inutile tous les $t-1$ lexèmes, aucune séquence d'au moins t lexèmes identiques ne peut être détectée. Cependant une telle insertion extensive de code inutile est rapidement remarquable par une analyse humaine du code et induit une inflation conséquente du volume du code.

4.3.1 Ajout de code inutile

Une opération d'obfuscation par ajout de code nécessite d'être neutre pour le résultat de l'exécution du code. Cette neutralité peut être atteinte par deux moyens :

1. L'ajout de code non accessible. Dans ce cas, le code n'est jamais exécuté. Ceci peut être obtenu par l'insertion de code quelconque à l'intérieur d'une structure de contrôle garantissant sa non-exécution (bloc *alors* d'une structure conditionnelle de condition invariablement fausse — ou bloc *sinon* d'une structure de condition vraie —, bloc de gestion d'exception jamais levée, ajout après une instruction de retour de fonction, implantation d'une fonction jamais utilisée...).
2. L'ajout de code accessible mais neutre à l'exécution. Il s'agit alors d'insérer du code ne présentant aucune conséquence sur le résultat de l'exécution mais pouvant potentiellement avoir un impact sur les performances du programme. On pourra par exemple définir des variables non utilisées ou pseudo-modifier le contenu de variables par des opérations neutres.

Dans ces deux cas, l'inutilité du code peut être plus ou moins simple à mettre en évidence. Des méthodes d'analyse statiques des chemins d'exécution de programmes peuvent permettre de détecter des morceaux de code trivialement inaccessibles[2]. De la même façon des instructions neutres à l'exécution peuvent être détectées dans des cas triviaux. En supposant l'inexistence d'effets de bord (par exemple avec un style de programmation fonctionnelle), ceci est réalisable par l'utilisation de graphe de dépendances. Il est donc envisageable pour ces cas de normaliser le code source manipulé par suppression du code détecté inutile puis d'utiliser une méthode de détection arbitraire ou alors d'utiliser directement des techniques de recherche de similitudes basées sur des représentations du code en graphe de dépendances de variables. Il demeure cependant que dans le cas général d'un langage Turing-complet, il est impossible de déterminer statiquement si une fonction retourne un résultat constant quels que soient les paramètres d'exécution. Une technique d'obfuscation pourrait ainsi utiliser une bibliothèque de routines algorithmiques de résultat invariant (fonctions retournant 0 dont le résultat est ajouté à la valeur d'une variable, anti-tautologies pour la condition de structures conditionnelles non exécutées...).

Quelques schémas d'insertion de code Nous définissons des schémas d'insertion de code par trois critères principaux :

[2]Nous pouvons remarquer que certains langages tels que Java imposent une discipline au programmeur en interdisant des cas triviaux de code inaccessible (de tels cas génèrent des erreurs à la compilation) ce qui limite les perspectives d'obfuscation par ajout de code inaccessible.

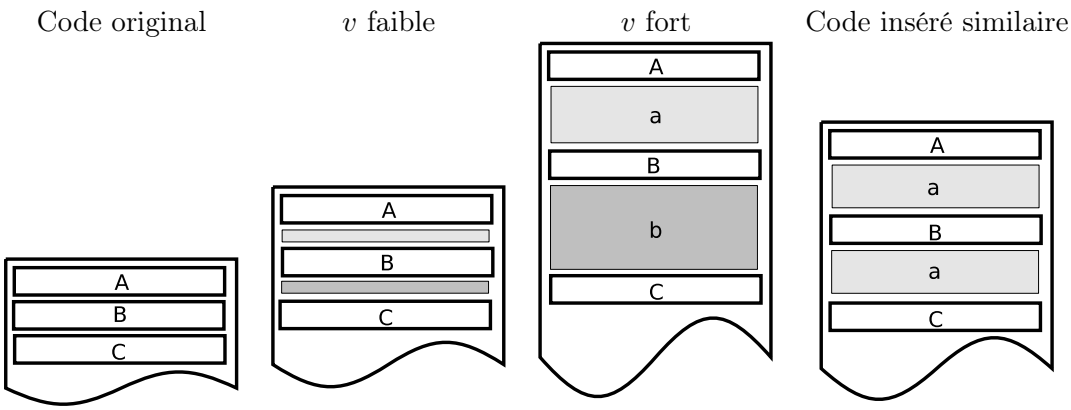

FIG. 4.1 – Quelques schémas singuliers d'insertion de code

1. Le ratio ρ de volume[3] de code inséré par rapport au volume de code original.
2. Le volume unitaire v de chaque morceau de code inséré.
3. Les relations de similarité entre morceaux de code insérés.

En fonction de ces critères, différentes stratégies sont employables pour la recherche de similitudes. Si ρ est important, une grande dissimilarité existe entre la taille du code original et du code copié : une métrique de similarité adaptée quantifiant le volume de l'intersection de code similaire sur le volume de code original doit alors être employée (cf chapitre 12). Si v est petit, les fossés de zones non correspondantes sont faibles et un algorithme d'alignement de séquences (cf chapitre 5) ou de consolidation de correspondances proches (cf chapitre 11) peut être utilisé. En revanche si ρ est élevé et v faible, les systèmes d'indexation et de sélection d'empreintes de séquences peuvent être *aveuglés*. Pour le troisième critère, si les morceaux de codes insérés sont similaires, ils peuvent être détectés comme tels par une recherche de similarité intra-projet : ces morceaux de code peuvent alors se voir attribuer un poids plus faible voire être totalement ignorés dans un processus de recherche de similarité inter-projet.

4.3.2 Suppression de code inutile

La suppression de code inutile présuppose l'existence de tel code au sein de morceaux copiés. Tout comme pour la normalisation de code avec du code inséré, des outils de recherche de couverture de code [115] peuvent être employés. La recherche de code inutile n'est généralement pas menée dans une optique obfuscatoire mais plutôt pour améliorer la qualité du code en recherchant des anomalies potentielles pouvant entraîner des bogues.

4.4 Réécriture d'expressions

La réécriture d'expression peut être utilisée dans un but d'obfuscation. Il s'agit de remplacer une expression par une expression dérivée de valeur identique à l'exécution. Parmi les méthodes de réécriture d'effet neutre, nous pouvons citer non exhaustivement :

[3]Nous définissons le volume d'une portion de code c par une métrique permettant de quantifier la quantité d'information apporté par ce code. Celui-ci peut être exprimé par une mesure théorique telle que la complexité de Kolmogorov du code, ou alors en pratique par le nombre de lignes (code brut), de lexèmes (séquence de lexèmes) ou de nœuds (arbre de syntaxe) utilisés pour la représentation du code.

$$
\begin{array}{lcl}
e & \longrightarrow & e+0 \mid e*1 \mid e-0 \mid e>>0 \mid e<<0 \\
e+f+g & \longrightarrow & (e+f)+g \\
e-f-g & \longrightarrow & e-(f+g) \\
e*(f+g) & \longrightarrow & e*f+e*g \\
 & \dots &
\end{array}
$$

FIG. 4.2 – Quelques règles de réécriture neutres d'arbres d'expressions arithmétiques

- l'usage d'une opération avec une opérande neutre (addition ou soustraction de 0, produit par 1...). L'opérande neutre pourra être masquée en utilisant une fonction de calcul telle que décrite en 4.3.1 ;
- l'usage de la commutativité d'opérateurs en modifiant l'ordre des opérandes ;
- l'usage de propriétés d'associativité et de distributivité de certains opérateurs.

Ces opérations peuvent être systématisées par l'utilisation de règles de réécritures simples[4] sur les arbres syntaxiques d'expressions (quelques règles sont présentées en figure 4.2). Symétriquement, il est possible d'établir une normalisation des arbres syntaxiques d'expressions avant traitement par un détecteur de similitudes afin d'éliminer les effets de cette obfuscation.

4.5 Changements de type

Les changements de types au sein du code copié peuvent intervenir dans une optique de réadaptation de code à un nouveau contexte ou dans un but d'obfuscation. Les types primitifs d'un langage sont souvent hiérarchisés : ainsi en Java un élément de type *byte* peut être représenté par un élément *short*, lui-même représentable par un *int*, représentable par un *long*. Nous pouvons ainsi remplacer le type d'une variable par un *supertype* pouvant le représenter. Concernant des types structurés (structures, classes...), il est possible de réaliser un remplacement en dupliquant une déclaration de type avec un nouveau nom ou alors, pour les langages à classes avec héritage, en remplaçant un type par un type ancêtre (supertype) ou descendant (sous-type). Cette dernière opération peut être accompagnée d'opérations de coercition si nécessaire.

Une solution envisageable afin de résister à ce type d'obfuscation consiste soit à normaliser avec un supertype suffisamment générique, soit à abstraire totalement des types (ce qui revient à la sélection du type racine).

4.6 Modifications de structures de contrôle

La manipulation de structures de contrôle peut être utilisée afin de modifier structurellement le code source avec un impact neutre sur l'exécution. Si les premiers langages de programmation ne comportaient pas de structures de contrôle mais plutôt des étiquettes et instructions de

[4]Il faut toutefois porter une attention particulière aux langages autorisant la surcharge d'un même opérateur pour différents types. Par exemple, l'opérateur * qui serait surchargé en C++ pour implanter la multiplication matricielle ne serait pas commutatif. En Java, l'opérateur + peut être utilisé pour l'addition de nombres mais également pour la concaténation de chaînes où il n'est pas commutatif.

saut conditionnel ou inconditionnel, les langages impératifs récents implantent généralement trois types de structures de contrôle :

1. Des structures conditionnelles permettant d'assortir l'exécution du code à une condition.
2. Des structures de boucle autorisant l'exécution d'une même portion de code un nombre variable de fois.
3. Des structures de capture d'exception permettant de gérer les erreurs inattendues survenant lors de l'exécution d'un bloc de code.

Entourage par une structure d'exécution unitaire Il est aisé d'entourer arbitrairement des groupes d'instructions d'un code source par une structure de contrôle arbitraire d'exécution unitaire. À cet effet, nous pouvons entourer les instructions d'une structure conditionnelle de condition invariablement vraie, d'une structure de boucle exécutée une unique fois via une condition d'arrêt ad-hoc ou alors d'une structure de capture d'exception (non nécessairement levable dans le contexte). Inversement ce procédé d'obfuscation peut également être intéressant afin d'introduire des volumes importants de code inaccessibles avec une condition invariablement fausse de la structure conditionnelle, une boucle exécutée 0 fois ou du code inutile inséré dans le code de capture d'une exception qui n'est jamais levée. Lorsque le code est représenté par un arbre de syntaxe, une étape de normalisation peut être réalisée afin de supprimer certaines de ces structures d'entourage inutiles lorsqu'elles sont statiquement détectables. Enfin, une méthode d'extension (cf chapitre 11) peut permettre la propagation de correspondances au-delà de ces structures d'entourage.

Modification de structures conditionnelles Nous examinons quelques opérations impliquant la modification de structures conditionnelles. Elles se caractérisent par la transposition de code, la création de nouvelles structures ou la transformation en structure de boucle :

- Dissociation/réassociation de structures conditionnelles. Les structures conditionnelles abritant un bloc *alors*, un bloc *sinon* ainsi qu'éventuellement plusieurs autres blocs *sinon* associés à une condition peuvent être éclatées en plusieurs structures conditionnelles individuelles. La condition d'entrée dans les blocs *sinon* transformés en structure *si...alors* doit vérifier si le bloc conditionnel précédent n'a pas été exécuté. Des opérations d'obfuscation peuvent également porter sur des structures de *pattern matching* complexes ou plus basiques (blocs *switch*) par conversion d'une suite de structures conditionnelles dont la condition porte sur les mêmes variables en une structure de *pattern matching* ou vice-versa.
- Inversion de blocs *alors-sinon*. Les blocs *alors* et *sinon* d'une structure conditionnelle peuvent être inversés en négativant la condition initiale. Cette modification est équivalente à une transposition de blocs de code.
- Transformation de structures conditionnelles en boucle ou structure de gestion d'exception. Une structure conditionnelle classique peut être réécrite sous la forme d'une boucle assortie d'une condition permettant une unique itération mais également par l'usage d'une structure de gestion d'exception en levant conditionnellement l'exception dans le bloc de capture et en insérant dans le bloc de gestion d'exception le code exécuté dans la structure conditionnelle originale.

Modification de type de boucle Afin d'offrir des possibilités syntaxiques plus étendues, la plupart des langages proposent dans leur grammaire plusieurs types de boucle (boucle de

type *pour* pour l'exécution itérée sur un ensemble, boucle de type *tant que* avec condition de continuation initiale ou finale, boucle de type *jusqu'à* avec condition de sortie finale...). La modification d'une boucle de type *pour* en boucle *tant que* nécessite l'extraction de la condition d'initialisation de la boucle et l'ajout en fin de boucle de l'instruction d'itération. Les boucles avec condition initiale de continuation peuvent être transformées en boucle avec condition finale de continuation en insérant une structure conditionnelle d'entrée au début de boucle pour la première itération. Préalablement à l'utilisation d'un outil de recherche de similitudes, il est donc possible d'utiliser une normalisation en un seul type de boucle.

Utilisation d'instructions de saut Il est généralement convenu que l'utilisation d'instructions de saut à la place de structures de contrôle adaptées peut dégrader la bonne compréhension du code [127]. Certains langages modernes (tel C++ ou D) conservent cependant la possibilité d'employer des instructions de saut (*goto*) et des étiquettes de branchement. Ainsi, tout code peut être linéarisé avec suppression de ses structures de contrôle et introduction de sauts et étiquettes de branchement. Des modifications moins importantes peuvent concerner également l'utilisation d'instructions de sortie de boucle (*break*) et de saut direct à la prochaine itération (*continue*) à la place d'une structure conditionnelle entourant les instructions de fin de boucle ainsi qu'une modification de la condition de début ou fin de boucle afin de permettre la sortie prématurée.

Déroulage de boucle Le déroulage de boucle consistant à dupliquer un bloc d'instructions au sein d'une boucle afin de limiter le nombre d'itérations de la boucle est généralement utilisé pour l'optimisation du code par les compilateurs afin de réduire les sauts sur le code assembleur. Dans un objectif d'obfuscation sur du code source, une telle méthode est d'effet limité dans la mesure où elle introduit des exemplaires supplémentaires de clones intra-projet facilement détectables.

4.7 Factorisation et développement de fonctions

4.7.1 Factorisation

La factorisation de fonctions est l'objectif principal de la recherche de similarité intra-projet afin de réduire le volume global de code. Elle peut être utilisée au sein d'un code copié dans un but d'obfuscation. Elle consiste alors à externaliser un morceau de code d'une fonction dans une nouvelle fonction indépendante[5]. En notant $\alpha\beta\gamma$ la portion de code originale et $\alpha'\beta'\gamma'$ l'exemplaire copié, l'externalisation de β' conduit un outil de recherche de similitudes au report des trois correspondances $(\alpha, \alpha'), (\beta, \beta'), (\gamma, \gamma')$ au lieu de l'unique correspondance $(\alpha\beta\gamma, \alpha'\beta'\gamma')$. Si un des morceaux α, β ou γ a un volume trop faible, ils peuvent ne pas être reportés pour correspondance par un outil à seuil de détection trop élevé. Un procédé extrême d'obfuscation serait alors d'externaliser individuellement chaque instruction afin d'aveugler l'outil de recherche de similitudes. Ce procédé peut être contré en réintégrant les fonctions dont le corps est de faible volume dans le corps de leurs fonctions appelantes. Cette opération devrait être réalisée en priorité sur les fonctions d'arité entrante (nombre de fonctions appelant cette fonction) basse.

[5]Le morceau de code peut également être externalisé dans une nouvelle macro pour des morceaux de faibles volumes dans des langages offrant un préprocesseur.

Nous remarquons que ce type d'obfuscation conduit à l'obtention de fonctions dont l'arité entrante est unitaire (en excluant des appels récursifs). Dans un souci de normalisation, nous pouvons redévelopper les appels en les remplaçant par le corps de la fonction appelée. Il reste néanmoins envisageable pour un plagiaire de consolider des fonctions externalisées en une unique fonction comportant une structure conditionnelle et un paramètre supplémentaire afin de déterminer le code à exécuter.

4.7.2 Développement

Le développement est l'opération inverse de la factorisation : elle consiste à remplacer un appel de fonction par son implantation complète, moyennant quelques modifications pour son adaptation au contexte local par déclaration et affectation des variables correspondant aux paramètres. Chaque occurrence de paramètre peut également être remplacée par l'expression communiquée comme argument à la fonction. Une obfuscation par développement (sauf à supprimer une fonction d'arité entrante unitaire) alourdit le code et fragmente des correspondances de taille importante comprenant des appels de fonction en leur sein en correspondances plus petites sans appel de fonction. Le remplacement des identificateurs de paramètres par des expressions différentes peut handicaper un outil de recherche de similarités exactes. Lorsqu'une fonction est développée en différents exemplaires analogues, ceux-ci peuvent être considérés comme des clones.

Une approche traitant ce type d'obfuscation (ainsi que la factorisation) est proposée au chapitre 7 sur des fonctions de séquences de lexèmes. Le principe consiste alors à ne considérer que des fonctions ne comprenant pas d'appels de fonctions qui sont ensuite factorisées en itérations successives. Nous obtenons alors finalement un nouveau graphe d'appels de fonctions factorisées commun à plusieurs projets pouvant être utilisé afin de proposer une métrique de similarité sur les fonctions originales.

Une autre approche utilisant des arbres de syntaxe est exposée au chapitre 11. Elle repose sur la recherche de correspondances exactes de sous-arbres. Ces *germes* sont ensuite consolidés et étendus afin de trouver des correspondances sur des entités structurelles plus importantes en suivant à la fois l'arbre de syntaxe mais aussi les liens d'appels de fonctions.

4.7.3 Fonctions récursives et boucles

Afin de s'affranchir de la gestion manuelle d'une pile d'appels, les algorithmes récursifs sont plus facilement codés par une fonction récursive ou un groupe de fonctions mutuellement récursives : le graphe d'appels contient donc une boucle. Un procédé d'obfuscation consiste à supprimer la récursivité d'une fonction (ou d'un groupe) par le remplacement par une fonction gérant manuellement une pile d'appels pour simuler la récursion.

Certaines fonctions récursives présentent dans les faits une récursivité terminale : un seul appel récursif à la fonction est réalisé à son terme. L'utilisation de ce type de fonctions est équivalent à l'usage d'une boucle itérative avec évolution d'une ou plusieurs variables locales. Dans les langages procéduraux, l'utilisation de fonctions récursives terminales est assez rare. Cependant, l'usage de boucles itératives est très fréquent : un obfuscateur peut les externaliser dans des fonctions à récursivité terminale.

4.8 Traduction vers un autre langage

Les langages de programmation généralistes étant dans leur majorité Turing-complets, en faisant abstraction des bibliothèques utilisées, tout algorithme implémenté dans un langage peut théoriquement être traduit dans un autre. Cependant cette traduction peut demander plus ou moins d'efforts en fonction des notions implantées par le langage. Ainsi, par exemple, la traduction d'un programme utilisant un paradigme fonctionnel (écrit par exemple en Haskell ou Caml) vers un langage procédural tel que le C est possible au prix de certaines difficultés. Pour des langages offrant les mêmes notions, en faisant abstraction des bibliothèques standards accompagnant ces langages, la traduction peut être plus simple. Par exemple, traduire un programme en langage Java vers le langage C# peut être réalisé à partir de règles de réécriture de l'arbre de syntaxe du programme en Java. Une méthode d'obfuscation par traduction dans un langage offrant des fonctionnalités proches est particulièrement intéressante car peu d'outils de recherche de similarité réalisent une comparaison inter-langages de projets à partir d'une forme normalisée de séquence de lexèmes ou d'arbre de syntaxe.

4.9 Obfuscation dynamique

Certains langages offrent des possibilités de contrôle dynamique de fonctions ou portions de code exécutées. Il s'agit alors de les exploiter afin de réaliser des opérations d'obfuscation impossibles à déceler via des méthodes d'analyse statique. Nous distinguons ici deux techniques d'obfuscation dynamique. La première se base sur le camouflage des liens d'appels entre fonction, ou alors sur l'extériorisation du code copié à exécuter. Quant à la seconde, elle utilise des possibilités d'introspection de certains langages afin soit de générer du code à l'exécution, soit de modifier dynamiquement du code existant.

4.9.1 Obfuscation des liens d'appel et chargement dynamique

Obfuscation des liens d'appel

L'utilisation de méthodes de recherche et d'extension de zones de similarité utilisant le graphe d'appels du projet est favorisée par une résolution non-ambiguë des liens entre sites d'appel et fonctions appelées. Comme évoqué en 3.4.2, la résolution des liens peut être problématique dans certaines situations telle que l'utilisation de pointeurs de fonctions ou l'existence de fonctions homonymes. Des procédés d'obfuscation consistant à rendre difficile la détermination du graphe d'appels peuvent exploiter des ambiguïtés de liens ne pouvant être levées que lors de l'exécution.

Par exemple en Java, nous pouvons systématiser l'obfuscation d'un lien d'appel vers une fonction appelée m depuis un type statiquement défini C (le type réel à l'exécution dérivant de C est alors inconnu) en créant une superclasse (classe ancêtre) A pour C qui sera utilisée pour la déclaration de l'objet et qui implantera toutes les fonctions de C (dont la fonction m) par du code aléatoire compilable. Dans le cas général, il est alors statiquement impossible de déterminer si la fonction appelée est $A.m$, $C.m$ ou une autre méthode m implantée par un descendant de C. Il est également possible de camoufler le nom de la méthode appelée en l'indiquant dans une chaîne de caractères de contenu statiquement indéterminable : la

méthode *Class.invoke(Object instance, String nomMethode)* est alors utilisée pour l'appeler. Dans le même esprit, un pointeur peut être employé en C pour appeler une fonction.

Chargement dynamique

Le chargement dynamique du code par récupération de code source ou code compilé localement ou à distance permet de masquer ce code source pour un détecteur de similarité tout en permettant son utilisation.

4.9.2 Modification dynamique de code

Certains langages possèdent des possibilités avancées de réflexivité tel que Smalltalk (ou des langages interprétés) permettant de modifier (et compiler) à l'exécution du code source. Un plagieur peut ainsi obfusquer l'utilisation du code en réalisant dans un premier temps des modifications pouvant avoir une portée sémantique sur le code. Ce code modifié, ayant un comportement à l'exécution différent du code original, subit ensuite des modifications symétriques lors de l'exécution afin de retrouver le comportement du code original.

4.10 Modifications sémantiques non-triviales

Plutôt que de copier directement un morceau de code réalisant une tâche spécifique et de réaliser ensuite des opérations d'obfuscation structurelles peu résistantes, un plagieur peut être tenté de réaliser des modifications sémantiques sur le code. Il est ainsi possible de modifier le fonctionnement même de l'algorithme implanté tout en conservant un résultat identique à l'exécution. Ces modifications peuvent être plus ou moins neutres sur la complexité mémorielle et temporelle du code. Toutefois ces opérations, même si elles peuvent s'inspirer du code original, nécessitent un investissement non-négligeable du plagiaire, parfois même supérieur à l'écriture du code *ex-nihilo*. D'autre part, la détermination d'une similarité algorithmique entre deux codes sources est, en règle générale, indécidable car étant un problème plus général que celui de l'arrêt d'un algorithme qui lui-même est indécidable. Si une similarité algorithmique est néanmoins recherchée, il demeure possible de tester les valeurs de sortie des programmes sur un sous-ensemble de l'espace des valeurs d'entrée possibles.

4.11 Récapitulatif

Les opérations d'édition et obfuscation présentées dans ce chapitre sont résumées par la figure 4.3. Nous les classons selon trois critères principaux que sont la facilité de l'automatisation des opérations d'obfuscation, les représentations ainsi que les méthodes adaptées pour la recherche de similitude en présence de ces obfuscation. Enfin, nous exposons pour chaque opération d'obfuscation sa classification selon Bellon [86] et Roy [87]. Bellon, pour son comparatif quantitatif de différents outils de recherche de clones classe les clones en trois grandes catégories : les clones de type 1 comportant pour seules opérations d'édition des modifications de formatage insensibles par analyse lexicale, les clones de type 2 avec des modifications d'identificateurs et de type et les clones de type 3 avec des modifications syntaxiques. Roy et al. quant à eux introduisent pour leur comparatif qualitatif un classement à quatre niveaux dont les deux premiers correspondent à ceux de Bellon. Le troisième niveau concerne les clones avec opérations d'ajout, suppression et modification de code ; le quatrième niveau introduit des

clones algorithmiquement équivalents avec transposition d'instructions, ajout de structures de contrôle ainsi que d'autres opérations syntaxiquement non-neutres.

Opération	Automatisation de l'obfuscation	Représentations adaptées	Méthodes adaptées	Taxonomie [86]/[87]
Modifications de formatage	Possible	Toutes sauf brute		1/1c
Édition de commentaires	Possible	Toutes sauf brute		1/1b
Renommage d'identificateurs	Possible (syntaxe)	Toutes sauf brute, lexicale non abstraite		2/2a
Transposition de code	Possible (PDG)	PDG	PDG homomorphes, alignement local de séquences, extension sur germes exacts	3/4abc
Ins./supp. de code trivialement inutile	Possible	syntaxique normalisée, PDG	PDG homomorphes, alignement de séquences, extension sur germes exacts	3/3d
Ins./supp. de code non-trivialement inutile	Difficile	Trace d'exécution	PDG homomorphes, alignement de séquences, extension sur germes exacts	3
Réécriture triviale d'expressions	Possible (syntaxe)	Syntaxique normalisée	alignement de séquences, hachage dégradé de sous-arbres	3/2bd,3ab
Réécriture non-triviale d'expressions	Difficile	Aucune	Alignement de séquences, hachage dégradé de sous-arbre	3
Changements de types	Possible (syntaxe+sémantique)	Syntaxique abstraite		2/2c
Modifications de structures de contrôle	Possible (syntaxe)	Syntaxique normalisée	Alignement de séquences, extension sur germes exacts	3/3ce,4d
Factorisation/développement de fonctions	Possible	Graphe d'appel	Factorisation, extension sur graphe d'appel	3/4
Traduction vers un autre langage	Possible	Traduction inverse, arbres de syntaxe avec abstraction de langage		3
Obfuscation dynamique	Possible	Trace d'exécution		NA
Modifications sémantiques non-triviales	Impossible (manuel)	Aucune	Aucune	NA

FIG. 4.3 – Opérations d'éditions, représentations et méthodes adaptées pour la recherche de similitudes

Fonction originale

```
int fib(int n)
{
 /* Initialisation des variables */
 int k = 1; int l = 1; int m = 0;
 /* Traitement des cas où n <= 2 */
 if (n == 1) return k;
 if (n == 2) return l;
 /* m contient la somme des deux termes
       précédents */
 for (int i = 3; i < n; i++)
  { m = k + l; k = l; l = m; }
 return m;
}
```

Obfuscation syntaxiquement neutre

```
int fib(n)
{
int a = 1; /* Renommage de variables */
int b = 1; int c = 0;
if (n == 1) return k;
if (n == 2) return l;
for (int i = 3; i < n; i++) { c = a + b; a
      = b; b = c; }
return m;
}
```

Transposition de code

```
int fib(int n)
{
int l = 1; int k = 1;
int m = 0;
for (int i = 3; i < n; i++)
 { m = k + l; k = l; l = m; }
if (n == 2) return l;
if (n == 1) return k;
return m;
}
```

Insertion de code inutile

```
int fib(int n)
{
 int k = 1; int l = 1; int m = 0;
 int inutile = 1; /* Déclaration inutile */
 if (n == 1) return k;
 if (n == 2) return l;
 for (int i = 3; i < n; i++)
  { m = k + l; k = l; l = m; }
 for (int i = 0; i < n; i++)
  inutile++; /* Boucle inutile insérée */
 return m;
}
```

Réécriture d'expression

```
int fib(int n)
{
 int k = 1; int l = 1; int m = 0;
 if (n == 1) return k*1;
 if (n == 2) return l+0;
 for (int i = 3; i < n; i += 1) {
  m = k + 2*l - l;
  k = l/1 + 0;
  l = m * m / (m * 1);
 }
 return pgcd(m,m);
}
```

Changement de types

```
int fib(short n)
{
 long k = 1; long l = 1; long m = 0;
 if (n == 1) return k;
 if (n == 2) return l;
 for (unsigned int i = 3; i < n; i++)
  { m = k + l; k = l; l = m; }
 return (int)m;
}
```

FIG. 4.4 – Fonction de calcul de nombres de Fibonacci en Java obfusquée par différentes méthodes

Modification de structures de contrôle

```
int fib(int n)
{
 int k = 1; int l = 1; int m = 0;
 if (! (n == 1 || n == 2))
 {
  for (int j = 0; j < 1; j++)
  for (int i = 3; i < n; i++)
   { m = k + l; k = l; l = m; }
  return m;
 } else if (n == 1) return k;
 else if (n == 2) return l;
}
```

Externalisation de fonction

```
int fib(int n)
{
 int k = 1; int l = 1; int m = 0;
 int m = fib12(n);
 if (m != -1) return m;
 else
 {
  for (int i = 3; i < n; i++)
   { m = k + l; k = l; l = m; }
  return m;
 }
}

int fib12(int n)
{
 if (n == 1) return 1;
 if (n == 2) return 2;
 return -1;
}
```

Traduction en JavaScript et obfuscation dynamique

```
/* Fonction JavaScript avec chargement
     dynamique */
function fib(n)
{
 var k = 1; var l = 1; var m = 0;
 if (n == 1) return k;
 if (n == 2) return l;
 for (var i = 3; i < n; i++)
  eval("c = a + b; a = b; b = c;".
       replace("a","k").replace("b","l").
       replace("c","m"));
 return m;
}
```

Modification sémantique non-triviale

```
int fib(int n)
{
 /* Réécriture récursive */
 if (n == 1) return 1;
 else if (n == 2) return 2;
 else return fib(n-1) + fib(n-2);
}
```