Transformations de programmes pertinentes pour la sécurité du logiciel

Stéphanie Riaud

Université de Rennes 1 - Master Recherche en Informatique
École Supérieure d'ingénieur de Rennes

**INRIA – Institut de Recherche en Informatique et Automatisme –
Rennes - Bretagne Atlantique**

Encadrante : Sandrine Blazy
Equipe : Celtique

Transformations de programmes pertinentes pour la sécurité du logiciel

Stéphanie Riaud

Rennes, le 1er juin 2011

Remerciements

Je remercie Sandrine Blazy pour ses conseils et son soutien.
Je remercie également André Oliveira Maroneze et Pierre-Emmanuel Cornilleau pour leurs nombreux conseils techniques.
D'une façon plus générale, je tiens à remercier l'équipe Celtique pour son accueil.

Résumé

Avec le développement des outils d'analyse tels que les décompilateurs ou les désassembleurs, la nécessité de protéger les codes contre le risque d'attaque de rétro-ingénierie est apparue. Un moyen efficace de prévention est d'obfusquer le code source, assembleur ou binaire. Un obfuscateur de code est une application qui convertit un programme afin de le rendre moins compréhensible et donc plus difficile à rétro-concevoir. Pour l'utilisateur final, l'application d'origine et l'application obfusquée doivent sembler identiques, le comportement général ne doit pas être altéré. Le principe d'obfuscation repose sur des transformations de code, quelques unes d'entre elles sont étudiées, leur efficacité et leur coût d'exécution sont évalués dans ce rapport.

Table des matières

Introduction

La rétro-ingénierie en informatique est une activité qui consiste à reconstruire un programme source à partir d'un code binaire. Deux approches peuvent être utilisées, l'analyse statique et l'analyse dynamique. L'analyse statique consiste à obtenir des informations sur le comportement d'un programme lors de son exécution sans réellement l'exécuter. L'analyse dynamique consiste à étudier le comportement du programme pendant son exécution à l'aide d'un débogueur. Contrairement à l'analyse statique, l'analyse dynamique utilise les données d'entrées, elle nécessite d'exécuter le programme de nombreuses fois. Cette technique reste très restrictive mais beaucoup plus simple à mettre en œuvre que l'analyse statique. Le développement d'outils d'analyse comme les décompilateurs ou les désassembleurs rend plus aisé la rétro-conception et oblige les concepteurs de logiciels à élaborer des techniques de défense.

Ce rapport expose la mise en place d'un mécanisme de protection contre la rétro-conception, l'obfuscation de code. Cette technique consiste à appliquer diverse transformations sur le code d'un programme afin de le rendre plus difficile à comprendre et donc plus compliqué à rétro-concevoir, sans pour autant modifier le fonctionnement de l'application finale. L'obfuscation de code ne rend pas impossible la rétro-ingénierie mais elle fait en sorte que le temps nécessaire à un ingénieur pour rétro-concevoir une application soit plus long que le temps nécessaire pour construire sa propre version. Lors d'une analyse dynamique, un adversaire exécute le programme obfusqué de façon répétitive afin d'observer l'image du programme dans la mémoire, pour tracer et analyser son flot de contrôle et son flot de données, et ainsi, rétro-concevoir le programme dans un langage plus haut niveau. Il faut donc obfusquer le code à la fois au niveau lexical, des données et du flot de contrôle.
L'obfuscation de code peut-être effectuée au niveau du code source, du code assembleur ou du code binaire. Le travail décrit dans ce rapport étant essentiellement basé sur des travaux exposant des obfuscations appliquées aux langages C ou JAVA [1, 2, 3], les obfuscations ont été implémentées au niveau du code source et du code assembleur. Les techniques d'obfuscation au niveau binaire sont parmi les plus difficiles à déjouer automatiquement, mais leur mise en œuvre est très différente de celle des obfuscations effectuées à plus haut niveau, la décision a été prise de ne pas les analyser dans ce travail.

La première partie de ce rapport est consacrée au rappel des objectifs du stage , le contexte de travail puis les transformations d'obfuscation sont présentés enfin les moyens qui existent pour évaluer ces transformations sont définis. La seconde partie de ce rapport est consacrée à la réalisation des transformations d'obfuscation et à l'étude des résultats. Dans un premier temps, les obfuscations au niveau du langage assembleur sont étudiées puis les transformations au niveau du langage C.

Première partie

Présentation du sujet

Chapitre 1

Contexte de travail

Dans cette partie le compilateur CompCert est présenté, ainsi que le problème de la rétro-ingénierie, puis les différents moyens de protection contre cette pratique sont exposés, les transformations d'obfuscation et les prédicats opaques sont définis et finalement les différentes étapes de travail sont décrites. Les sections 1.1, 1.2 et 1.3 sont reprises de l'étude bibliographique préalable au stage [4].

1.1 Le compilateur CompCert

Un compilateur doit normalement produire un code en adéquation avec la sémantique du code source. Cependant les compilateurs sont devenus très complexes, il est donc très difficile de prouver qu'ils fonctionnent correctement. Malgré de nombreux tests, il arrive que lors de la compilation des erreurs soient détectées alors que le code source est correct, de la même façon il peut arriver qu'un exécutable erroné soit généré à partir d'un code source valide. Pour la majorité des logiciels cela ne pose pas de réel problème. Les logiciels classiques sont validés par des tests rigoureux qui permettent de mettre en évidence les erreurs, qu'elles soient originaires du code source ou du compilateur lui-même. Pour les logiciels critiques, dont le fonctionnement doit être vérifié de façon très rigoureuse, ces tests sont complétés par l'utilisation de méthodes formelles : le *model checking*, *l'analyse statique* ou la *preuve de programme*, sur le code source. Le maillon faible de la chaîne reste donc le compilateur. Leroy [6, 5] et son équipe ont développé CompCert, un compilateur **réaliste du langage C et vérifié formellement** avec l'assistant à la preuve coq. Il peut être utilisé pour le logiciel embarqué critique. A l'origine, il avait pour unique langage cible l'assembleur PowerPC, auquel, sont venus s'ajouter récemment les langages assembleurs ARM et x86. Une des évolutions envisagée est d'ajouter au compilateur CompCert un outil d'obfuscation afin de protéger les codes générés de la rétro-ingénierie. L'obfuscation se fera à plusieurs niveaux : sur le code source et sur le code assembleur.

Le compilateur CompCert est composé de deux parties : une partie prouvée (en coq) et une partie non prouvée (figure 1.1). La partie non prouvée comprend l'analyse syntaxique, la construction d'arbres de syntaxe abstraite, l'affichage de la syntaxe assembleur, l'assemblage et l'édition de liens. Le stage porte sur les langages source et cible de la partie prouvée du compilateur. Le compilateur CompCert traduit des programmes sources codés en Clight[1], en langage assembleur. Parmi les langages assembleur cibles PowerPC, ARM et x86, le choix a été fait de travailler sur le langage PowerPC disponible dans toutes les versions (même les plus anciennes)

[1] large sous ensemble du langage C.

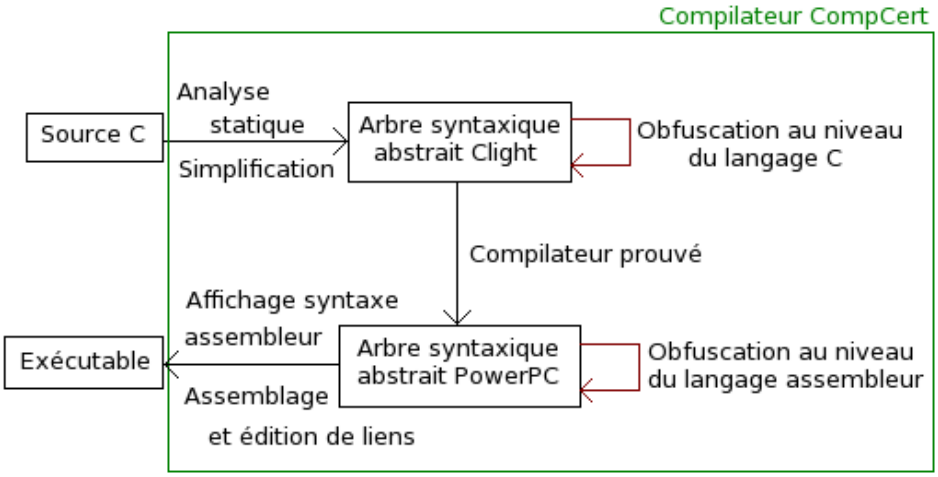


FIG. 1.1 – **Schéma du compilateur CompCert**

du compilateur. La partie prouvée du compilateur CompCert est écrite dans le langage de spécification de coq. Ces spécifications sont ensuite automatiquement traduites en code Caml. Le code Caml ainsi généré est le code du compilateur. Le travail effectué durant ce stage a consisté à modifier dans ce code Caml, les fichiers traitants les langages Clight et PowerPC.

Le langage Clight possède tous les types et les opérateurs, y compris les opérateurs arithmétiques de pointeurs, toutes les instructions, y compris l'instruction goto du langage C (figure 1.2). Les types long long et double double, les fonctions à nombre variable d'arguments, le passage par valeur des structures et les structures de type union ne sont pas supportées par le langage Clight. Les différences entre le langage Clight et le langage C étant minimes, cela n'a pas d'incidence sur le sujet du stage. Le compilateur CompCert est modérément optimisant. Les seules optimisations sont la propagation de constantes, l'élimination de sous-expressions communes et l'allocation de registres. Il est important de préciser qu'aucune optimisation de boucle n'a été implémentée dans le compilateur.

1.2 Les problèmes liés à la rétro-ingénierie

La rétro-ingénierie consiste à étudier un code binaire pour en déterminer le fonctionnement. Cette technique est souvent utilisée à des fins malhonnêtes, l'exemple le plus répandu est le détournement des protections anti-copie des jeux vidéo. Certaines entreprises peuvent avoir recours à la rétro-conception pour accéder aux technologies mises au point par d'autres sociétés ou rechercher des vulnérabilités pour les exploiter et les distribuer. Cependant l'obfuscation de code peut aussi être utilisée de matière légitime notamment pour comprendre des programmes dont le code source n'est pas connu, tels que les virus informatiques. La rétro-ingénierie pose beaucoup de problèmes aux développeurs de logiciels. La décompilation d'un logiciel en elle-même n'est pas illégale. « L'utilisateur régulier d'un logiciel peut, sans autorisation de l'auteur, procéder à la reproduction et à la traduction de la forme de son code, c'est-à-dire sa décompilation, lorsque ces opérations sont indispensables pour obtenir les informations nécessaires à l'interopérabilité de ce logiciel avec d'autres logiciels » (Code de la Propriété Intellectuelle art L.122-6-1-IV). Une entreprise peut choisir, pour diverses raisons d'avoir recours à la rétro-conception. Comme in-

Signedness:	*signedness*	::=	`Signed` | `Unsigned`
Integer sizes:	*intsize*	::=	`I8` | `I16` | `I32`
Float sizes:	*floatsize*	::=	`F32` | `F64`
Types:	τ	::=	`int`(*intsize*, *signedness*)
		|	`float`(*floatsize*)
		|	`void`
		|	`array`(τ, n)
		|	`pointer`(τ)
		|	`function`(τ^*,τ)
		|	`struct`(id,φ)
		|	`union`(id,φ)
		|	`comp_pointer`(id)
Field lists:	φ	::=	$(id, \tau)^*$

FIG. 1.2 – **Syntaxe abstraire du type Clight [7]**

Expressions:	a	::=	id — variable identifier
		|	n — integer constant
		|	f — float constant
		|	`sizeof`(τ) — size of a type
		|	$op_1\ a$ — unary arithmetic operation
		|	$a_1\ op_2\ a_2$ — binary arithmetic operation
		|	$\star a$ — pointer dereferencing
		|	$a.id$ — field access
		|	`&`a — taking the address of
		|	$(\tau)a$ — type cast
		|	a_1 ? a_2 : a_3 — conditional expressions
Unary operators:	op_1	::=	- | ~ | !
Binary operators:	op_2	::=	+ | - | * | / | % — arithmetic operators
		|	« | » | & | | | ^ — bitwise operators
		|	< | <= | > | >= | == | != — relational operators

FIG. 1.3 – **Syntaxe abstraire des expressions Clight [7]**

```
Registres flottants:    freg ::= FPR0
                               | FPR1
                               | FPR2
                               .
                               .
                               .
                               | FPR30
                               | FPR31

Registres entiers:      ireg ::= GPR0
                               | GPR1
                               | GPR2
                               .
                               .
                               .
                               | GPR30
                               | GPR31

Instructions:     i::= Padd of ireg * ireg * ireg
                     | Paddi of ireg * ireg * constant
                     | Paddis of ireg * ireg * constant
                     | Paddze of ireg * ireg
                     | Pand_ of ireg * ireg * ireg
                     | Pandc of ireg * ireg * ireg
                     | Pandi_ of ireg * ireg * constant
                     | Pandis_ of ireg * ireg * constant
                     | Pfmul of freg * freg * freg
                     | Pfneg of freg * freg
                     | Pb of label
                     | Pbctr
                     | Pbctrl
                     | Pbl of ident
                     .
                     .
                     .
                     | Plabel of label
```

FIG. 1.4 – **Syntaxe abstraire PowerPC**

diqué dans l'article de loi, elle peut vouloir assurer l'interopérabilité entre une application et ses installations. Une autre raison, serait de vouloir s'assurer que l'application ne pose pas de problèmes de sécurité, qu'elle fait bien ce qu'elle est censée faire [8]. Il est légitime pour un développeur (ou un éditeur) de logiciels de souhaiter se prémunir par divers moyens techniques de la rétro-conception.

1.3 Les solutions de lutte contre la rétro-ingénierie

Lorsqu'un développeur est victime de rétro-ingénierie le premier réflexe est de se tourner vers la loi. En effet, la loi sur le droit d'auteur protège les développeurs. Malheureusement, il est très compliqué pour les petites structures de faire valoir leurs droits face à des sociétés plus importantes. De plus, le délit est souvent impossible à mettre en évidence, un développeur qui a recours à la rétro-ingénierie ne copie évidemment pas l'intégralité d'un programme, il se contente des parties critiques voire même de ne voler que l'idée et non le code lui-même. La solution judiciaire n'apportant pas réellement de solution, il est préférable d'envisager des solutions amont. L'une d'entre elle serait de ne pas fournir à l'utilisateur final les sections de code critique. L'accès à l'application se ferait alors via un site Internet. Pour limiter les problèmes liés à la bande passante et à la latence du réseau, l'application serait divisée en deux parties. La première contenant le code non critique fonctionnerait localement sur la machine du client, l'autre partie fonctionnerait à distance.

Une autre approche consisterait à chiffrer le code avant qu'il soit envoyé à l'utilisateur. Malheureusement cette solution n'est pas viable dans tous les cas, si les étapes de déchiffrage et d'exécution ne prennent pas toutes les deux place au niveau matériel mais que le code est exécuté dans la partie logicielle via une machine virtuelle, il devient très aisé d'intercepter et de décompiler le code déchiffré. La suite logique serait donc de ne transmettre à l'utilisateur final que le code natif correspondant à son architecture et à son système d'exploitation. Il devient alors très compliqué de rétro-concevoir le code. Malheureusement, de nouveaux problèmes se posent lors de la transmission de code natif. Il est inconcevable d'exécuter sur une machine un code natif douteux. Le développeur serait dans l'obligation d'ajouter une signature numérique qui garantirait qu'il est bien le propriétaire du code et que personne ne l'a altéré.

Finalement, une approche plus simple consiste à modifier le code afin qu'il soit beaucoup plus difficile à comprendre tout en conservant son fonctionnement, cette technique est appelée l'obfuscation de code.

L'objectif du projet est d'ajouter au compilateur CompCert ce type d'outil afin de permettre aux utilisateurs de protéger efficacement et simplement leurs programmes.

Chapitre 2

Les transformations d'obfuscation

Cette partie est consacrée aux transformations d'obfuscation. Les obfuscations de mise en page, du contrôle de flux d'exécution et de données sont décrites. Les sections 2.1 et 2.3.1 sont reprises de l'étude bibliographique préalable au stage [4].

2.1 Définition

L'hypothèse principale lors de l'obfuscation d'un programme est que le programme initial et le programme final doivent avoir exactement le même comportement. Si le programme original crée un fichier ou envoie un message via Internet, le programme obfusqué doit faire de même. Dans l'article [2], il est expliqué que certaines libertés peuvent être prises en autorisant le programme final à avoir des effets secondaires différents ou à ne pas se terminer dans des cas où le programme d'origine se terminait avec des erreurs. L'important cependant est que les comportements observables soient parfaitement identiques. Décrire la sémantique d'un programme consiste à définir quels sont ces comportements observables, par exemple, quels sont les valeurs calculées, les appels de fonctions, les accès mémoires etc. Voici la définition formelle des transformations obfuscantes [1, 2, 3] :

Définition 1 (Transformation d'obfuscation) *Soit :*
$P \xrightarrow{T} P'$ *: la transformation d'un programme source P en un programme cible P'.*
$P \xrightarrow{T} P'$ *est une transformation d'obfuscation, si P et P' ont le même comportement observable* et si P' est plus compliqué** à comprendre que P (pour un humain ou un désobfuscateur). Plus précisément, pour que* $P \xrightarrow{T} P'$ *soit une transformation d'obfuscation les conditions suivantes doivent-être respectées :*

- *Si P échoue à terminer ou termine avec une erreur, alors P' peut terminer ou non.*
- *Par ailleurs, P' doit terminer et produire la même sortie que P'.*

* Cette notion est définie précisément dans CompCert comme ceci : deux programmes ont le même comportement observable si les valeurs calculées par la fonction *main()* sont identiques et si les deux programmes possèdent la même séquence d'appels aux fonctions externes.

** Il n'existe pas de définition standard qui permet d'évaluer le fait qu'un programme soit plus difficile à comprendre qu'un autre, c'est l'un des objectifs du stage.

(1) $x = 0$
(2) $y = 1$
(3) if $(7y^2 - 1 \neq x^2)$ --→ P_3^T
(4) ...
(6) $x(x+1)(mod2) \equiv 0$ --→ P^T
(7) ...
(8) $x = 1$
(9) if $(x + y = y^2)$ --→ P_9^F (P_7^V)

FIG. 2.1 – **Exemple d'utilisation de prédicats opaques**

2.2 Obfuscation de la mise en page

Les transformations les plus basiques ciblent la mise en page, certaines suppriment le style de codage pour que le code ne soit plus qu'un unique bloc incompréhensible, d'autres suppriment les commentaires enfin certaines modifient les identifiants. Ces transformations triviales sont peu résistante face à l'attaque d'un désobfuscateur automatique mais permettent de rendre le programme beaucoup moins compréhensible pour un attaquant humain. Lors du stage, l'impact d'une transformation est étudiée au niveau du code assembleur, parmi les transformations listées précédemment la seule transformation intéressante est donc la modification des identifiants. Cette modification de code a été implémenté au niveau assembleur (section 4.1). L'implémenter au niveau du langage C n'aurait eu aucun sens, en effet, il est inutile de modifier les identifiants qui n'apparaissent pas dans le code généré. Travailler au niveau du langage assembleur garantit l'efficacité de cette transformation.

2.3 Obfuscation du contrôle du flux d'exécution

Cette section est consacrée à l'étude des transformations de code visant à obfusquer le contrôle de flux d'exécution.

2.3.1 Les prédicats opaques

Le vrai challenge lorsque l'on cherche à mettre au point des transformations est de les rendre, à la fois peu couteuses en ressources (temps d'exécution, mémoire) et résistantes aux attaques des désobfuscateurs. Dans cet objectif, beaucoup de transformations reposent sur des variables et des prédicats opaques. Un prédicat est un énoncé dont le sens logique peut être vrai ou faux en fonction de la valeur de ses arguments ($ex : x(x+1)(mod2) \equiv 0$). Une variable V (ou un prédicat P) est définie comme opaque si les propriétés q connues par l'obfuscateur sont difficilement déductibles pour le désobfuscateur [1, 2, 3]. La définition formelle d'une construction opaque est la suivante.

Définition 2 (Construction opaque) *Une variable V est opaque en un point du programme p, si en ce point du programme, V possède une propriété qui est connue au moment de l'obfuscation. Nous écrivons alors V_p^q ou V^q si p est clairement défini dans le contexte (s'il est inutile de repréciser à quel point de programme p, q est connue au moment de l'obfuscation). Un prédicat P est opaque au point de programme p si son résultat est connu lors de l'obfuscation. Nous écrivons*

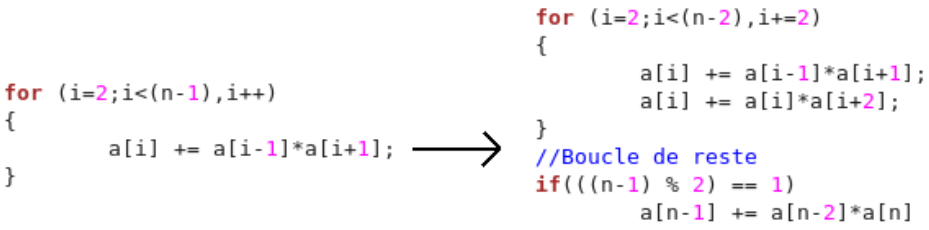


FIG. 2.2 – **Deroulage de boucle**

P_p^F (P_p^V) si P est évalué Faux (Vrai) au temps p et $P_p^?$ si P peut-être évalué à Vrai ou Faux (exemple figure 2.1).

2.3.2 Modification des expressions

De nombreuses transformations reposent sur l'utilisation de prédicats opaques, il est possible par exemple d'obfusquer les boucles en rendant leur condition de terminaison plus complexe sans augmenter leur intervalle d'itération ou d'ajouter des opérateurs redondants pour compliquer les opérations arithmétiques. Au niveau du compilateur, les conditions de terminaison de boucles et les opérations arithmétiques sont gérées sous forme d'expressions (figure 5.9), il a donc suffi de modifier ces expressions, en ajoutant des prédicats opaques, pour obfusquer les conditions de terminaison des boucles et les opérations arithmétiques.

2.3.3 Insertion de code mort

Un code plus long est plus difficile à comprendre par un désobfuscateur ou un être humain, qu'un code plus court. Une des transformations de base consiste donc à ajouter du code inutile (ou code mort) au programme d'origine. Au niveau du langage C cela revient à ajouter des prédicats opaques, au niveau de l'assembleur cela consiste simplement à ajouter des instructions inutiles au code initial. Des prédicats opaques ayant déjà été inséré au langage C (section 2.3.2), la transformation d'obfuscation par insertion de code mort a uniquement consisté à ajouter des instructions dans le code assembleur (section 4.2).

2.3.4 Obfuscation des boucles for

Les transformations de boucles sont fréquemment utilisées pour optimiser des programmes, notamment en programmation parallèle, pour des applications multi-threads. Le principe du déroulage de boucle est simple : plusieurs itérations de boucle sont exécutées simultanément (figure 2.2). Une fission de boucle consiste à placer chaque instruction du corps de boucle initial dans une nouvelle boucle ayant le même espace d'itération (figure 2.3). Ces transformations possèdent l'avantage d'augmenter la taille globale du code et le nombre de conditions qu'il contient, ce qui le rend plus obscur et plus difficile à désobfusquer. Appliquées séparément ces transformations sont peu résistantes fasse à l'attaque d'un désobfuscateur mais combinées leur résilience augmente drastiquement. Le compilateur CompCert ne possédant pas d'optimisation de boucle, ces transformations ont été implémentées durant le stage (section 5.2).

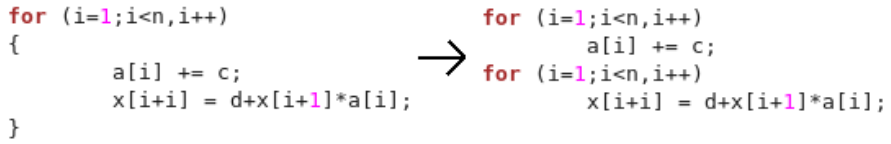


FIG. 2.3 – **Fission de boucle**

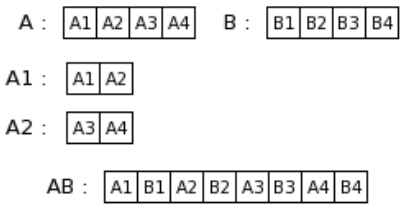


FIG. 2.4 – **Partage / Rassemblement de tableaux**

2.4 Obfuscation des données

Cette section décrit les transformations qui visent le stockage des données. Il est possible de modifier le stockage, l'encodage, l'agrégation ou l'ordonnancement des données.

2.4.1 Modification du stockage et de l'encodage des données

Il y a des manières naturelles de stocker des données en mémoire, par exemple lorsqu'un tableau est crée, l'espace mémoire est alloué avec un type approprié à la taille des données manipulées. Les transformations obfuscantes visant le stockage et l'encodage des données imposent un stockage et un encodage inhabituel. Un exemple simple consiste à changer la valeur d'une variable i par i'=i*c1+c2. Des transformations de ce type sont effectuées lorsque les expressions sont modifiées (partie 2.3.2).

2.4.2 Modification de l'agrégation des données

Une partie importante du travail du rétro-ingénieur consiste à retrouver les structures de données du programme. Afin de rendre cette tâche plus compliquée, une transformation consiste à restructurer les tableaux. Quatre opérations de bases peuvent être effectuées : (1) un tableau peut être partagé en plusieurs tableaux, (2) plusieurs tableaux peuvent être réunis en un seul, (3) le nombre de dimensions d'un tableau peut être augmenté ou (4) diminué. La figure 2.4 montre un exemple de partage et de rassemblement de tableaux. Le tableau A est partagé en deux nouveaux tableaux A1 et A2. Les tableaux A et B sont fusionnés pour créer le tableau AB. La figure 2.5 montre un exemple de redimensionnement de tableaux. Les données du tableau D à une dimension sont stockées dans le tableau D1 à deux dimensions, inversement, les données du tableau C à deux dimensions sont stockées dans le tableau C1 à une dimension. L'implémentation de ces transformations est décrite section 4.3.

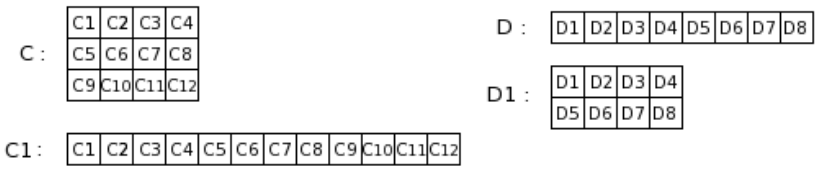


FIG. 2.5 – **Redimensionnement de tableaux**

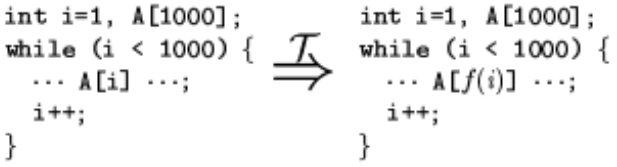


FIG. 2.6 – **Réorganisation des éléments d'un tableau [1]**

2.4.3 Modification de l'ordonnancement des données

Il est très simple de réorganiser les déclarations dans l'application source, par exemple en rendant aléatoire l'ordre des méthodes et la déclaration des variables à l'intérieur de ces méthodes. Mais ce type de transformation, bien qu'irréversible, n'est pas résistante face à l'attaque d'un désobfuscateur. Il est aussi possible de réorganiser les éléments d'un tableau en ajoutant une fonction d'encodage f(i) qui fait correspondre les éléments du tableau original avec ceux du nouveau tableau (figure 2.6).

2.5 Implémentation, étude et tests des transformations

D'après l'étude bibliographique quelques transformations de code à implémenter ont été sélectionnées. La transformation visant les identifiants a été la première implémentée. Cette technique d'obfuscation est triviale et elle permet de comprendre la syntaxe du compilateur. La modification des identifiants ayant été effectuée au niveau du langage assembleur, il a été décidé d'implémenter la transformation suivante au niveau du langage C. La transformation qui consiste à modifier les expressions du langage C a donc été choisie. Elle repose sur l'utilisation de prédicats opaques, technique très répandue pour l'obfuscation de code et elle regroupe à elle seule plusieurs autres techniques (voir 2.3.2, 2.3.3 et 2.4.1). Afin de compléter l'obfuscation par ajout de code mort au niveau du langage C, une transformation qui consiste à insérer aléatoirement des instructions au niveau du code assembleur a été implémentée (section 4.2). La transformation suivante qui a été choisi vise les boucles (section 5.2). Une fois de plus, les transformations de type déroulage/fission de boucles sont des classiques de l'obfuscation de code et possèdent un coût d'exécution très faible pour une efficacité importante. Finalement, afin de diversifier les structures sur lesquelles les transformations sont appliquées, la dernière obfuscation de code vise les tableaux (section 4.3).

Afin d'étudier l'efficacité des transformations, le code PowerPC issu du compilateur a été examiné. Les modifications doivent être prises en compte, il y a un risque qu'elles soient supprimées, par exemple, par une optimisation du compilateur. Pour chaque transformation, le code assembleur initial et le code assembleur obfusqué sont comparés. En effet, lors d'une décompila-

tion le rétro-ingénieur ne cherche pas à remonter jusqu'au langage source, la plupart du temps il se contente de retrouver le code assembleur. Enfin des tests ont été effectués sur des programmes comportant les cas de base du langage source afin de vérifier que les modifications ne perturbent pas l'exécution de l'application. Afin de faciliter ces tests, une interface très basique a été ajoutée, elle permet de sélectionner les transformations à appliquer au code, les informations à afficher et d'évaluer automatiquement ces modifications de code.

Le chapitre suivant décrit les outils d'évaluation des transformations d'obfuscation.

Chapitre 3

Evaluation des transformations

L'objectif de cette partie est de définir des outils permettant d'évaluer l'efficacité des transformations d'obfuscation. Les transformations sont évaluées selon trois critères : (1) la puissance, (2) la résilience et (3) le coût. La notion de qualité d'une transformation est définie. Cette partie est reprise de l'étude bibliographique antérieure au stage [4] et s'appuie sur les travaux de Collberg et al. [1][2][3]

3.1 Mesure de la puissance

Quelle est la signification pour un programme P' d'être plus obscur qu'un programme P ? En se basant sur l'hypothèse générale suivante issue des recherches dans le domaine de la mesure de la complexité des programmes (théorie de la complexité des algorithmes) : si le programme P et le programme P' sont identiques, excepté le fait que P' contient en plus une propriété q, alors P' est plus complexe que P. Afin de construire une transformation d'obfuscation, de nouvelles propriétés sont ajoutées au programme de base, augmentant ainsi sa complexité. La mesure de la complexité permet de formaliser la notion de puissance d'une transformation. De façon non formelle, la puissance d'une transformation mesure à quel point un code obfusqué est plus difficile à comprendre pour un humain que le code original. Ce concept est formalisé avec la définition suivante.

Définition 3 (Puissance d'une transformation) *Soit T une transformation d'obfuscation, telle que $P \xrightarrow{T} P'$ transforme un programme source P en un programme cible P'. Soit $E(P)$, la complexité de P*[1] *et soit, $T_{pot}(P)$, la puissance avec laquelle T change la complexité de P, nous avons :*

$$T_{pot}(P) \stackrel{def}{=} E(P')/E(P) - 1$$

T est une transformation efficace si $T_{pot}(P)>0$.

Certaines propriétés augmentent la puissance d'une transformation [1], pour obtenir une transformation d'obfuscation puissante, il est primordial :

- d'augmenter la taille globale du programme et d'introduire de nouvelles méthodes,
- d'introduire de nouveaux prédicats, d'augmenter le niveau de conditionnelles et le nombre de boucles,

[1] Les auteurs [1] ont laissé cette notion volontairement abstraite, cela n'a aucun impact sur le reste de l'étude.

– d'augmenter le nombre d'arguments dans les méthodes et les dépendances entre les variables.

Dans la suite de ce rapport, la puissance d'une transformation est mesurée comme pouvant être <basse,moyenne,haute>.

Ces critères ont été utilisé pour définir quelles transformations implémenter. L'ajout de code mort et la modification des expressions augmentent la taille du programme, le nombre de prédicat et le niveau de conditionnelles. Transformer les boucles augmente leur nombre.

3.2 Mesure de la résilience

Certaines transformations sont inutiles (ex : modification des identifiants) puisqu'elles peuvent être défaites très simplement par un désobfuscateur. Il est donc primordial d'introduire le concept de résilience, qui mesure la résistance des transformations face aux attaques d'un désobfuscateur automatique. Voici la définition formelle de la résilience :

Définition 4 (Résilience d'une transformation) *Soit T une transformation d'obfuscation, telle que $P \xrightarrow{T} P'$ transforme un programme source P en un programme cible P'. $T_{res}(P)$, est la résilience de P. $T_{res}(P)$ est irréversible si l'information ciblée par la transformation est supprimée et que P ne peut pas être reconstruit à partir de P'.*

$$T_{res}(P) \stackrel{def}{=} \textit{Résilience } (T_{EffortDésobfuscateur}, T_{EffortProgrammeur})$$

Où Résilience () est une fonction définie figure 3.1.

L'effort du programmeur correspond au temps nécessaire pour construire un désobfuscateur efficace. L'effort du désobfuscateur correspond aux ressources en temps et en espace mémoire nécessaires pour désobfusquer la transformation. Une transformation est locale si elle n'affecte qu'un bloc basique (une entrée, une sortie) du graphe de flot de contrôle (GFC). Une transformation est globale si elle affecte entièrement le GFC, elle est inter-procédurales si elle affecte le flot d'informations entre procédures et elle est inter-processus si elle affecte l'interaction entre processus (thread) indépendants.

La résilience est classée en cinq catégories <trivale, faible, moyenne, forte, irréversible >. Une transformation irréversible ne peut jamais être défaite.

Il est important de distinguer la différence entre puissance et résilience. Une transformation est puissante si un lecteur humain ne parvient pas à déchiffrer le code, elle possède une forte résilience si elle résiste aux attaques d'un désobfuscateur.

3.3 Mesure du coût

Le coût d'une transformation correspond aux pénalités de temps et d'espace qu'une transformation occasionne à une application obfusquée. Le coût est classé en quatre catégories <gratuit, bon marché, élevé, très élevé>.

Définition 5 (Coût d'une transformation) *Soit T une transformation d'obfuscation, telle que :*
$P \xrightarrow{T} P'$ transforme un programme source P en un programme cible P'. $T_{cost}(P)$, est le coût supplémentaire en temps et en espace d'exécution de P' par rapport à P. Le coût est :

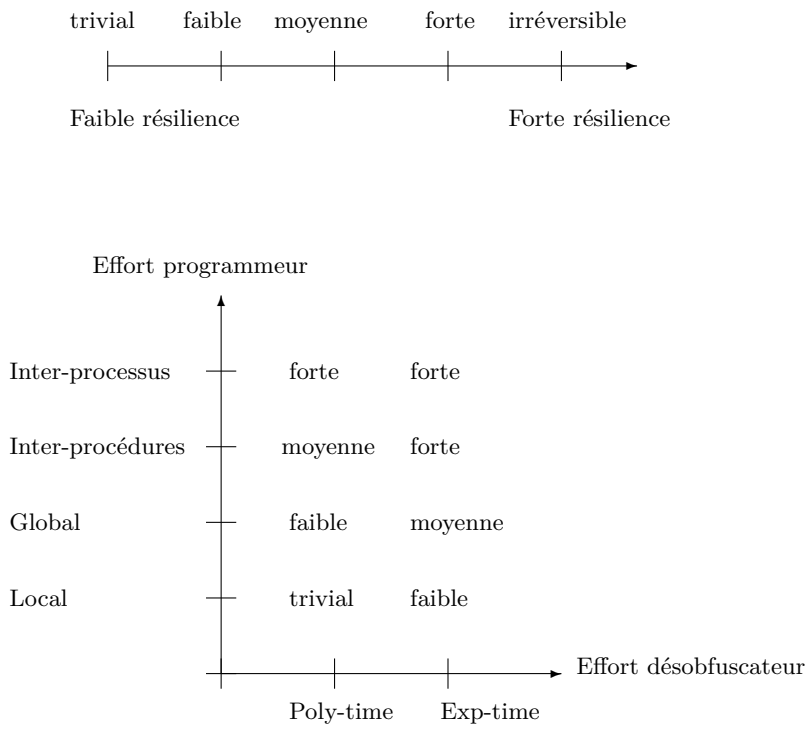


FIG. 3.1 – **La résilience d'une transformation d'obfuscation [1]**

- *Très élevé : si l'exécution de P' requiert exponentiellement plus de ressources que celle de P.*
- *Élevé : si l'exécution de P' requiert $O(n^p)$, avec p>1, plus de ressources que celle de P.*
- *Bon marché : si l'exécution de P' requiert O(n) plus de ressources que celle de P.*
- *Gratuit : si l'exécution de P' requiert O(1) plus de ressources que celle de P.*

Le coût réel associé à une transformation dépend de l'environnement dans lequel elle est exécutée.

3.4 Définition de la qualité

Voici la définition formelle de la qualité d'une transformation d'obfuscation :

Définition 6 (Qualité d'une transformation) *$T_{qual}(P)$, la qualité de la transformation T, est définie par une combinaison de la puissance, la résilience et du coût de T :*

$$T_{qual}(P) = (T_{pot}(P), T_{res}(P), T_{cost}(P))$$

3.5 Interface d'évaluation

Les outils définis par Collberg [1][2][3] apportent des éléments pour évaluer les transformations d'obfuscation. Malheureusement ces outils ne sont pas précis, la puissance et la résilience d'une transformation sont difficiles à quantifier. Une interface d'évaluation a été ajouté au projet, cette interface évalue automatiquement la qualité des transformations. Aucun critère pratique n'ayant été trouvé durant l'étude bibliographique [4], pour évaluer la résilience d'une transformation, les propriétés influençant la puissance des transformations ont été utilisées. Pour évaluer les transformations sont pris en compte : le nombre, le type de structures modifiées et l'évolution de la taille globale du programme. L'interface d'évaluation n'est pas aboutie, elle sera améliorée ultérieurement.

La partie suivante de ce rapport est consacrée à la présentation des transformations d'obfuscation qui ont été implémentées au niveau du langage assembleur et du langage C. Leur mise en œuvre et les résultats obtenus sont décrits précisément.

Deuxième partie

Réalisation et résultats

Chapitre 4

Obfuscation du langage assembleur

Cette partie est consacrée aux trois transformations de code implémentées au niveau du langage assembleur. L'obfuscation du langage assembleur est traitée avant l'obfuscation du langage C car les transformations sont plus simple à implémenter. La première transformation permet simplement de modifier le nom des fonctions afin de les rendre plus obscurs. La seconde transformation consiste à insérer des instructions inutiles dans le code d'origine et la troisième consiste à modifier les tableaux.

4.1 Modification des identifiants

Avant d'explorer des transformations plus complexes, une transformation triviale, couramment utilisée par des obfuscateurs JAVA comme Crema [9], la modification d'identifiants, est décrite en détails.

4.1.1 Implémentation

Les seuls identifiants encore visibles au niveau du langage assembleur sont le nom des fonctions. Ils ne sont pas stockés sous forme de chaînes de caractères mais sous forme d'unités lexicale. Des tables de hachage permettent de récupérer la chaîne de caractères correspondant à une unité lexicale donnée, ou vis-versa. Le nom des fonctions est modifié directement dans ces tables. Avant tout changement, un test est effectué afin de vérifier que les fonctions ne sont pas des fonctions externes (ex : bibliothèques). En effet changer le nom d'une telle fonction empêche au compilateur de la reconnaître, la séquence d'instruction ajoutée lors de son appel ne sera pas générée. L'exemple de la figure 4.1 permet de comparer deux codes assembleur, l'un est modifié, l'autre non. La partie de gauche correspond au code non obfusqué. Deux méthodes *printf* et *calcul* sont appelées dans la fonction *main*, les identifiants sont parfaitement lisibles. Dans la version obfusquée du code, le nom de la fonction *printf* est toujours visible alors que celui de la fonction *calcul* est modifié.

4.1.2 Evaluation

Cette transformation possède un coût totalement nul. Bien qu'elle soit *irréversible* (le nom original des méthodes ne peut pas être retrouvé), cette méthode n'est pas résistante face à

FIG. 4.1 – **Code assembleur avec modification des identifiants**

l'attaque d'un désobfuscateur. Néanmoins, elle permet de supprimer des informations utiles pour un rétro-ingénieur, elle possède donc une puissance *moyenne*.

4.2 Ajout de code mort

La façon la plus courante d'obfusquer un code assembleur est d'ajouter de façon aléatoire du code mort. Sont définies comme « *code mort* » des instructions qui n'ont aucune utilité et qui ne modifient pas le comportement du code d'origine.

4.2.1 Implémentation

L'objectif est d'ajouter aléatoirement des instructions dans le code d'origine. L'implémentation repose sur l'utilisation de la fonction *random()*, disponible en programmation *Caml*.

Première étape : Le code assembleur est représenté sous forme d'une liste d'instructions. Le nombre d'instructions *ni* est calculé par un parcours de liste. Une valeur aléatoire *va* comprise entre 0 et *ni* est déterminée grâce à la fonction *random()*. La liste des instructions est parcourue jusqu'à cette valeur aléatoire et la sous-liste d'instructions correspondante *sl* est extraite. La liste des registres non utilisés *lr* par les instructions de la sous-liste *sl* est calculée. Dans un premier temps une liste contenant tous les registres du langage *ltr* est crée. Les registres utilisés par chaque instruction de la liste *sl* sont relevés et retirés de la liste *ltr*. On obtient ainsi la liste *lr* contenant tous les registres non employés par le code initial. Cette liste permet d'utiliser, lors d'ajout d'instructions, uniquement des registres qui ne contiennent pas de valeur utile à l'exécution du code, diminuant ainsi le risque de modifier le fonctionnement du programme final. Le jeu d'instructions n'offrant aucune possibilité pour déterminer si un registre est utilisé, aucune alternative à l'utilisation de cette méthode peu élégante n'a pu être trouvée. De plus, ayant dépassé la phase d'optimisation du compilateur, un minimum de registres est utilisé, certains seront donc disponibles lors de l'obfuscation pour l'ajout d'instructions.

Deuxième étape : Si des registres sont disponibles, deux instructions sont insérées dans le code à l'emplacement préalablement déterminé par la variable aléatoire *va*. La première instruction utilise des registres de type entiers et la seconde utilise des registres de type flottants. Les

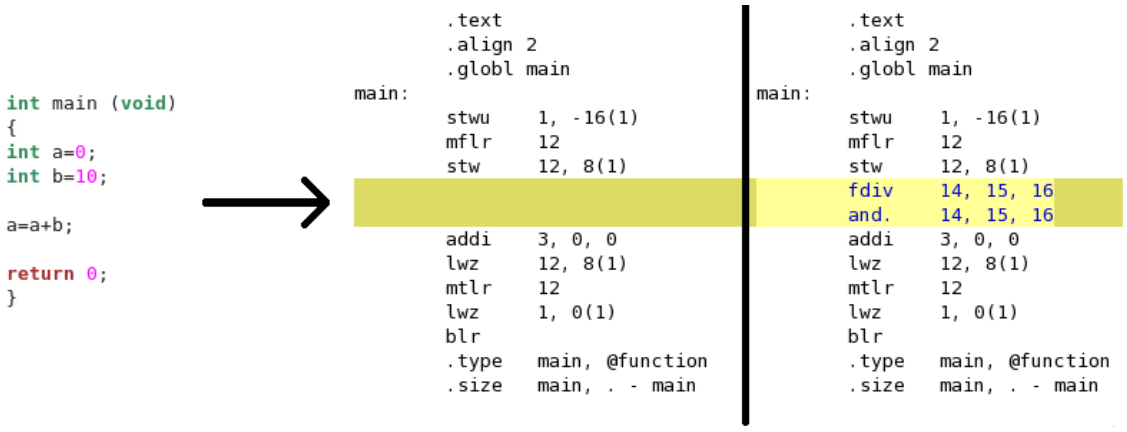


FIG. 4.2 – **Exemple d'insertion de code mort**

instructions sont choisies aléatoirement parmi une liste d'instructions qui ont été préalablement sélectionnées comme n'ayant pas d'impact sur le code principal. Le choix de ces instructions a été effectué suite à l'étude du langage PowerPC [10, 11].

Troisième étape : Lors de la construction de la liste d'instructions et donc du code assembleur, la méthode *obfusque()* est appelée, elle regroupe les deux étapes préalablement décrites, modifiant ainsi le code assembleur selon nos souhaits. L'opération d'insertion d'instructions est répétée un nombre de fois égal à la longueur du code divisé par quatre. Cette valeur peut être modifiée néanmoins, augmenter la taille du code de 25% semble être raisonnable.

La figure 4.2 présente un exemple simple d'insertion de code.

4.2.2 Tests

Six programmes de représentatifs ont été sélectionnés. L'objectif était de couvrir tous les cas de bases de la programmation en langage C. Le programme *P1* contient par exemple de nombreuses conditions *if* et de nombreuses comparaisons entre nombres entiers. Chaque programme renvoie un résultat (utilisation de fonctions d'affichage) afin de déterminer s'il s'exécute correctement ou non. Les particularités de chaque programme sont résumées dans le tableau 4.3. Dans un premier temps aucun de ces programmes de base ne s'exécutait correctement, la plupart s'exécutaient à l'infini et les autres pratiquaient de mauvais accès mémoire. Le jeux d'instructions ajouté aléatoirement dans le code [10] a dû être revu plusieurs fois. Finalement, uniquement des instructions très basiques (ex : and, add etc...) sont insérées. De plus, les registres ne doivent pas être des registres réservés, en effet sur les 31 registres flottants et les 31 registres entiers, seule la moitié sont généraux [11]. Finalement pour plus de précaution, aucune instruction n'est insérée avant un branchement, afin d'éviter toute modification des conditions de sauts. Une fois ces corrections effectuées les programmes de test s'exécutent correctement.

4.2.3 Résultats

Pour des tailles de code initiales comprises entre 61 et 349 lignes, les pourcentages de code ajoutés sont compris entre 14% et 42% (figure 4.4). Il n'y a pas de lien observable entre la taille

	P1	P2	P3	P4	P5	P6
Boucle while			X		X	X
Boucle for						X
Condition if	X					X
Appels de bibliothèques	X	X	X	X	X	X
Appels de fonctions						X
Tableaux			X			X
Switch/Case		X				
Variables globales				X		
Opérations sur des entiers/flottants	X		X	X	X	
Comparaisons d'entiers / de flottants	X		X	X	X	X
Boucles For imbriquées				X		
Conditions if imbriquées					X	

FIG. 4.3 – **Particularités des programmes de tests**

initiale et le pourcentage de code ajouté. Néanmoins les codes ayant un nombre important de structures de type if, for ou de comparaisons entre entiers/flottants présentent des valeurs d'obfuscation moins élevées. Cela s'explique par le choix qui a été fait lors de l'implémentation de ne pas autoriser l'ajout d'instructions avant un branchement. Notre panel ne contenant que neuf instructions, la même instructions peut être ajoutée de nombreuses fois lorsque le programme est de taille importante. De même, tous les registres disponibles sont utilisés mais les combinaisons sont souvent les mêmes, les instructions insérées peuvent donc être aisément détectées .

Ces premiers résultats sont encourageants, en effet même si le manque de diversité des instructions insérées et le manque de combinaisons de registres rend le code mort facilement identifiable, la transformation est viable puisque les applications s'exécutent normalement. De petites corrections pourront aisément corriger le manque de diversité d'instructions et de combinaisons de registres. De plus cette technique a pour avantage d'être adaptable, l'utilisateur pourra choisir le niveau d'obfuscation qu'il souhaite, en fonction des performances nécessaires au bon fonctionnement de son application. Une technique d'obfuscation efficace et malléable a été mise en évidence.

4.3 L'obfuscation des tableaux

L'objectif de ce type de transformations est d'obfusquer les opérations effectuées sur des tableaux.

4.3.1 Présentation

Afin d'implémenter des obfuscations plus complexes, des transformations qui visent le stockage des données ont été réalisées. Les modifications de tableaux font parties de ces transformations. Quatre transformations différentes existent [1] : le partage, le rassemblement, la diminution et l'augmentation du nombre de dimensions de tableaux. L'étape antérieure à l'implémentation de ces transformations a été de comprendre comment les tableaux sont gérés par le compilateur.

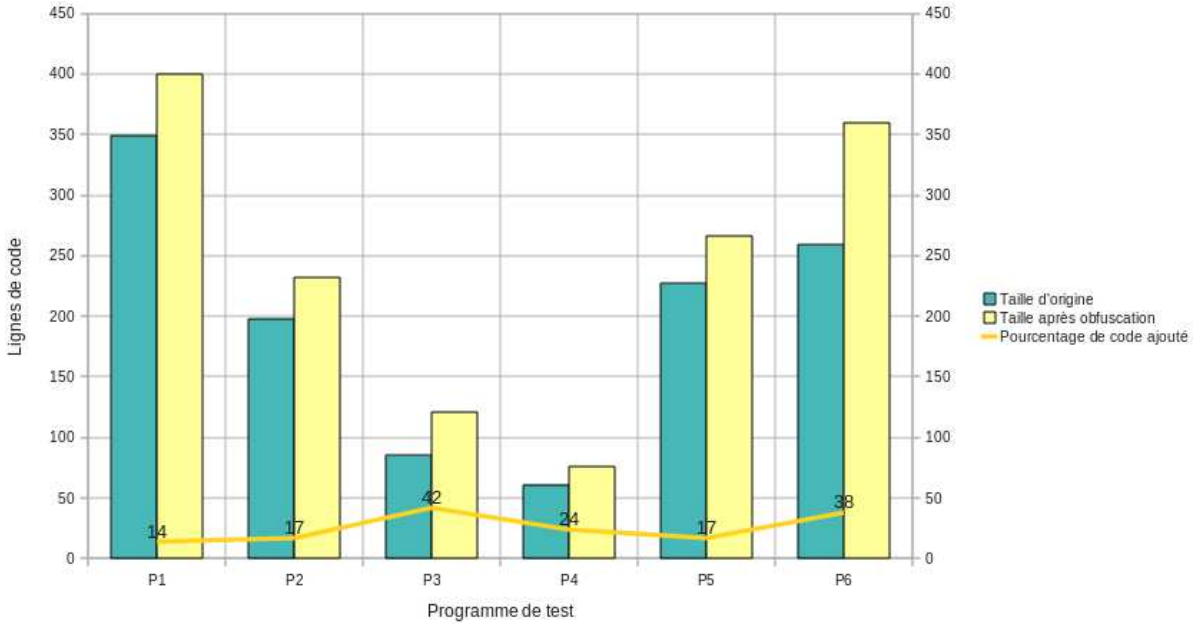


FIG. 4.4 – **Résultats du test de l'obfuscation par ajout de code mort**

4.3.2 Gestion des tableaux par le compilateur

La logique veut que si deux tableaux sont utilisés au lieu d'un seul, deux espaces mémoires différents seront réservés, des données qui auraient du être stockées les unes à la suite des autres ne le seront pas. Mais lorsque deux tableaux sont déclarés, les emplacements alloués sont contigus (figure 4.5). Il est alors évident qu'une transformation de type partage ou rassemblement de tableaux est inefficace. Pour que la transformation soit efficace, il faut simplement changer l'ordre dans lequel les accès mémoires sont effectués. Le compilateur gère un tableau à deux dimensions comme un tableau de tableau. De même que précédemment, si un tableau à une dimension est transformé en un tableau à deux dimensions (ou vice-versa), la transformation est inefficace (figure 4.6), les espaces mémoires réservés sont identiques.

4.3.3 Implémentation

L'étude du fonctionnement du compilateur a permis de déduire qu'effectuer les transformations présentée dans la section 2.4.2, au niveau du langage C, ne présente aucune utilité, ces transformations n'auront aucun impact sur le code assembleur généré. Au niveau du langage assembleur, le fichier d'édition du code a été modifié afin que les emplacements mémoires choisis ne soient plus contigus. La technique d'obfuscation développée par Collberg [1] a ainsi été adaptée au projet. Afin de ne pas modifier le fonctionnement de l'application finale, la correspondance entre les anciens emplacements mémoires et les nouveaux est enregistrée dans une table de hachage. Toutes les lectures ou écritures de données de tableaux sont ainsi obfusquées lors de la génération du code assembleur.

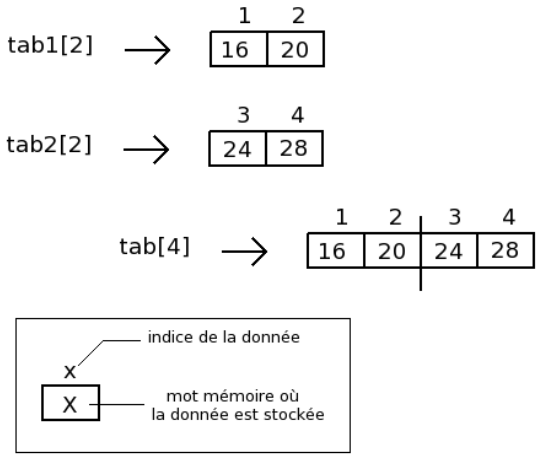


FIG. 4.5 – **Partage / Fusion de tableaux**

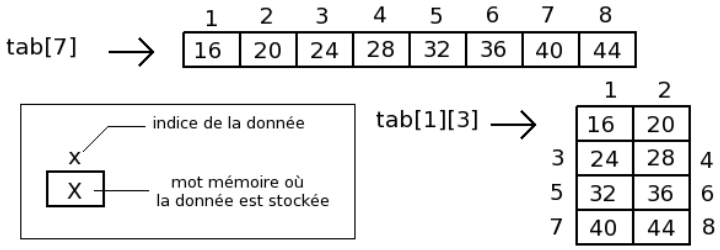


FIG. 4.6 – **Changement de dimension de tableaux**

```
...                          ...                          ...
tab1[0]=0;          --> addi 3, 0, 0  -->   addi 3, 0, 0
                        stw  3, 16(1) -->   stw  3, 128(1)

tab1[1]=1;          --> addi 3, 0, 1  -->   addi 3, 0, 1
                        stw  3, 20(1) -->   stw  3, 162(1)

tab2[0]=tab1[0]; --> lwz  3, 16(1) -->   lwz 3, 128(1)
                        stw  3, 24(1) -->   stw 3, 120(1)

tab2[1]=tab1[1]; --> lwz  3, 20(1) -->   lwz 3, 162(1)
                        stw  3, 28(1) -->   stw 3, 100(1)
...                          ...                          ...
      (1)                         (2)                          (3)
```

FIG. 4.7 – **Obfuscation de tableau :** (1) code source en langage C, (2) code assembleur compilé non obfusqué et (3) code assembleur compilé obfusqué

4.3.4 Résultats

La figure 4.7 présente un exemple d'obfuscation de tableaux. Deux tableaux *tab1* et *tab2* ont été déclarés préalablement. Les deux premières instructions permettent de placer deux valeurs aux emplacements zéro et un du premier tableau. Les emplacements alloués sont contigus (16 et 20) lorsque le code n'est pas obfusqué et aléatoires (128 et 162) lorsque le code est obfusqué. Les deux instructions suivantes placent respectivement la valeur présente à l'emplacement zéro du premier tableau au même endroit dans le second tableau et la valeur présente à l'emplacement un du premier tableau au même endroit dans le second tableau. Les nouveaux emplacements alloués pour le tableau *tab2* sont aléatoires lorsque le code est obfusqué. Les données sont récupérées aux bons emplacements dans les deux versions du code assembleur.

La question posée est de savoir si ces transformations sont valables ou non. Le stockage en mémoire est le même en ayant un tableau d'une ou plusieurs dimensions, un ou plusieurs tableaux. L'objectif était d'ajouter ou de supprimer des structures pour compliquer la tâche d'un désobfuscateur mais cette finalité n'est pas atteinte, les tableaux n'ont pas été supprimé mais uniquement masqué. Néanmoins un programmeur qui utilise un tableau à deux dimensions le fait car ce type de structure fait correspondre proprement les données manipulées. Le rétro-ingénieur (ou le désobfuscateur) perd des informations essentielles lorsque cette correspondance est brouillée.

4.4 Conclusion

Dans cette partie, trois obfuscations de code basiques ont été implémentées. Les obfuscations des identifiants de fonctions et des tableaux présentent une résilience très faible, néanmoins, ces transformations permettent de dissimuler des informations essentielles lorsque l'attaque est menée par un rétro-ingénieur. De plus, leur coût étant nul, elles restent tout de même intéressantes à mettre en œuvre.

L'obfuscation par ajout de code mort possède une résilience et une puissance beaucoup plus importante. La solution implémentée dans le compilateur CompCert est viable mais il est indispensable de diversifier les instructions et les combinaisons de registres utilisées. De plus l'utilisateur doit pouvoir choisir quel pourcentage de code supplémentaire il désire.

Les obfuscations au niveau du langage assembleur sont compliquées à développer. Les structures du langage C ne sont plus visibles. Le travail est effectué directement au niveau des instructions. Les obfuscations les plus robustes devront être développées dans les langages de plus haut niveau.

La partie suivante est consacrée à l'étude des transformations d'obfuscation implémentées au niveau du langage C.

Chapitre 5

Obfuscation du langage C

Cette partie décrit les obfuscations au niveau du langage C. La première transformation de code consiste à modifier les expressions du langage. La seconde obfuscation consiste à modifier les boucles *for*.

5.1 Modification des expressions

L'idée générale mise en œuvre pour obfusquer le code au niveau du langage C est de modifier les arbres syntaxiques du langage définis dans le compilateur. L'objectif est de créer de nouveaux nœuds sur ces arbres syntaxiques afin de les compliquer un maximum, un point important est de veiller à ne pas modifier la valeur réelle de l'arbre et de conserver le type initial de l'expression.

5.1.1 Définition des types du langage C

Les expressions du langage Clight sont représentées au niveau du compilateur par le type *expr*. Ce type correspond lui-même à une expression *expr_descr* annotée par son type *type*. Le type *expr_descr* représente tous les types d'expressions disponibles dans le langage et le type *type* donne le type de l'expression. Les déclarations de ces types sont données figures 1.2 et 1.3.

5.1.2 Implémentation

Pour chaque type d'expression défini dans le compilateur, une transformation est mise au point.

Entiers et flottants Une expression de type *Econst_float* f deviendra une expression de type :

f op1 (f2 op2 f3)

Ainsi la valeur flottante $f = 3.2$ est remplacée par l'opération $f = 3.2 - (2.0 - 2.0)$ (figures 5.1 et 5.2). La valeur de la constante f reste inchangée mais dix-huit nœuds et sept niveaux ont été ajoutés à l'arbre syntaxique de cette expression. L'utilisateur n'a pas accès aux arbres syntaxique mais les modifier ainsi devrait avoir un impact sur le code assembleur généré. Le même genre de transformation est appliqué aux types *Econst_int* pour les entiers.

Opérations arithmétiques L'opération d'addition de type *op1 + op2* devient une expression de type :

*op1 + (op2 * op2) / (op2 * 1)*

Comme pour l'exemple précédent, le résultat de l'opération arithmétique n'est pas modifié mais l'arbre obfusqué possède dix-sept nœuds et quatre niveaux supplémentaires (figures 5.3 et 5.4)). Les opérateurs arithmétiques de soustraction, multiplication et division sont modifiés de façon similaire.

Opérations binaires Les opérateurs binaires « et », « ou » et « ou exclusif » de type : *Op1 opérateur Op2* deviennent des expressions de type :

(Op1 opérateur Op2) opérateur (Op1 opérateur Op2)

L'hypothèse que quel que soit le résultat de l'opération $X = A.B$[1], l'opération $X = (A.B).(A.B)$ présentera un résultat similaire est utilisée. L'opération $X = A \oplus B$[2] deviendra $X = (A \oplus B).(A \oplus B)$ et l'opération $X = A + B$ deviendra $X = (A + B) + (A + B)$[3]. L'arbre syntaxique ainsi obtenu possède vingt et un nœuds et deux niveaux supplémentaires (figure 5.5, 5.6).

Opérations de comparaison Les opérateurs de comparaison « égale à », « différent de », « inférieur à », « supérieur à », « inférieur ou égal à » et « supérieur ou égal à » sont obfusqués suivant le même principe. En effet, quel que soit le résultat de l'opération $X = A < B$, l'opération $X = (A < B).(A < B)$ rendra un résultat identique. Les expressions de type *Op1 Opérande Op2)* deviennent alors des expressions de type :

(Op1 Opérande Op2) Oand/Oor (Op1 Opérande Op2)

Toujours sans modifier le résultat de l'opération de comparaison, l'arbre syntaxique obtenu possède vingt et un nœuds et deux niveaux supplémentaires (figures 5.7 et 5.8).

Les expressions étant construites récursivement, très peu d'entre elles (environ une sur dix) sont obfusquées, le code est ainsi grandement obscurci, tout en gardant un coût raisonnable. De plus, afin de rendre le processus d'obfuscation aléatoire, les expressions obfusquées sont choisies en s'appuyant sur la fonction aléatoire *random()*. Les expressions sont plus complexes en effet de nombreux nœuds et niveaux ont été ajoutés aux arbres syntaxiques. La section suivante permettra de déterminer si ces transformations complexifies ou non le code assembleur.

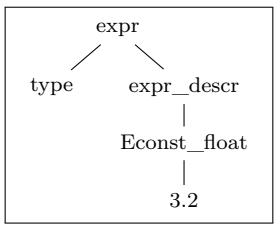


FIG. 5.1 – **Arbre syntaxique flottant non-obfusqué**

[1]L'opérateur "et" est représenté par le symbole "."
[2]L'opérateur "ou exclusif" est représenté par le symbole "⊕"
[3]L'opérateur "ou" est représenté par le symbole "+"

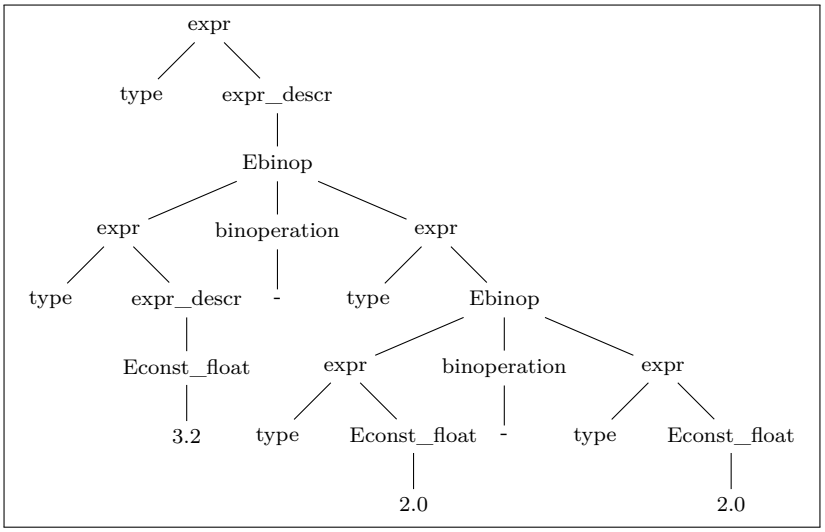


FIG. 5.2 – **Arbre syntaxique flottant obfusqué**

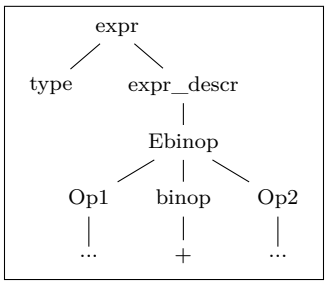


FIG. 5.3 – **Arbre syntaxique de l'opération d'addition non-obfusqué**

5.1.3 Tests et résultats

Ces transformations ont été appliquées aux six programmes de tests. Les programmes s'exécutent tous correctement. Cette transformation est plus fiable qu'une tranformation appliquée directement au niveau du langage assembleur puisque les structures fournies dans le code du compilateur sont directement modifiées. Les codes assembleurs des programmes de test P2, P3 et P4 ne sont pas modifiés par cette obfuscation.

P2 : Le programme P2 ne contient que très peu d'expressions, le code se résumant à une structure de type switch/case, notre interface d'évaluation indique qu'aucune de ces expressions n'a été modifiée. L'obfuscation est donc valide uniquement si le code source dépasse une certaine taille. Une solution envisageable serait d'augmenter le pourcentage d'expressions obfusquées mais cela augmenterait de façon trop importante la taille des codes, ce qui pourrait occasionner une trop grande pénalité en terme de temps d'exécution.

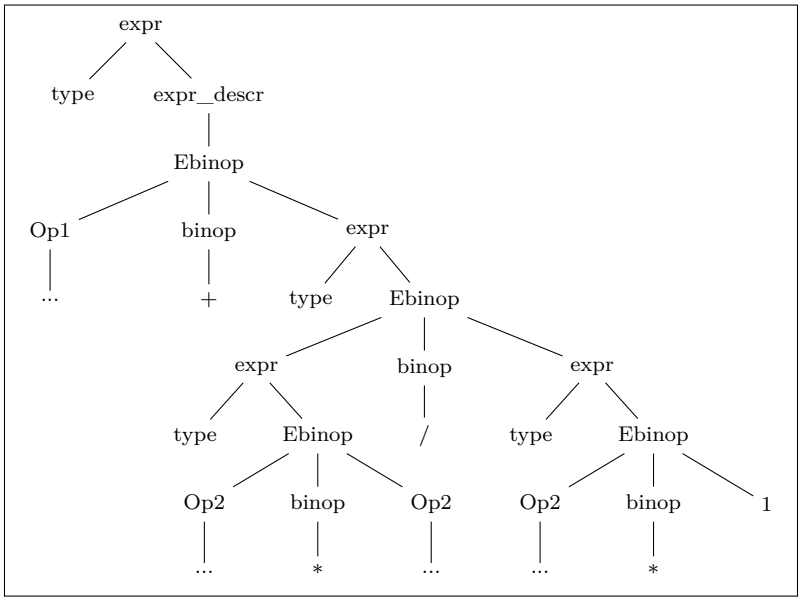


FIG. 5.4 – **Arbre syntaxique de l'opération d'addition obfusqué**

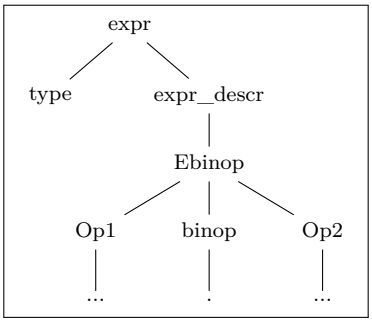


FIG. 5.5 – **Arbre syntaxique de l'opération "et" non-obfusqué**

P3 et P4 : Les code assembleurs des programmes P3 et P4 ne sont pas modifiés, contrairement au programme précédent quelques expressions de type *Econst_int* (constantes entières) ont été obfusquées. L'obfuscation n'est donc pas conservée par le compilateur.

P5 : Le code obfusqué du programme P5 contient dix lignes supplémentaires. Ces lignes sont regroupées en deux blocs, deux expressions ont donc été modifiées. L'interface d'évaluation indique en fait que treize constantes entières, une expression booléenne « et », une opération de comparaison « différent de », deux opérations arithmétiques de soustraction et une multiplications ont été modifiées. Les résultats précédents indiquent que la modification des constantes n'a pas d'impact sur le code. De plus, l'étude du code assembleur et la comparaison avec les autres programmes de tests permet de déterminer que les deux expressions obfusquées, visibles dans le code assembleur, sont l'expression booléenne « et » et l'opération de comparaison « différent de ».

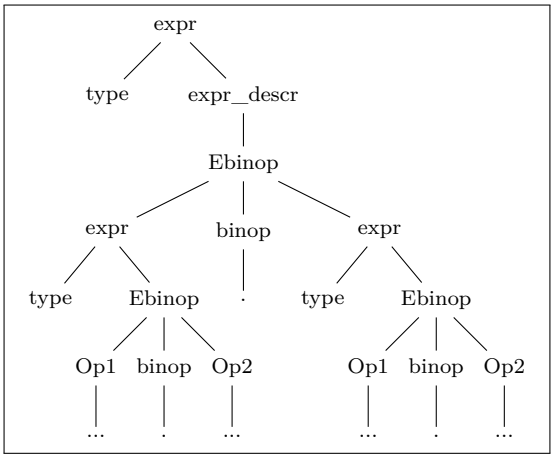


FIG. 5.6 – **Arbre syntaxique de l'opération "et" obfusqué**

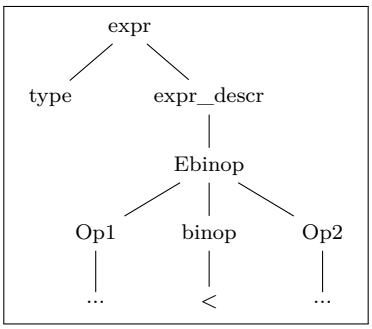


FIG. 5.7 – **Arbre syntaxique de l'opération "inférieur à" non-obfusqué**

P6 : Le code assembleur obfusqué du programme P6 contient huit lignes supplémentaires, divisées en trois blocs. L'étude de ce code et la comparaison avec les autres programmes de test, permet de déduire que les expressions obfusquées sont une expression de comparaison « différent de » et deux opérations arithmétiques d'addition. Les opérations arithmétiques de soustraction, multiplication et de division ne sont pas modifiées.

P1 : Le programme P1 présente des résultats intéressants, en effet, le code obfusqué comporte seize lignes de code supplémentaires. Elles sont regroupées en cinq blocs, deux opérations arithmétiques de multiplication et trois opérations de comparaison de type « égal à » obfusquées sont visibles dans le code. Il reste des expressions de type constantes et des opérations arithmétiques qui sont obfusquées sans incidence visible sur le code assembleur.

Peu de transformations sont conservées par le compilateur (figure 5.9). Les opérations de comparaison et les opérations logiques obfusquées sont intégralement visibles dans le code assembleur, alors que seulement certaines des transformations concernant les opérations arithmétiques sont conservées. L'obfuscation des constantes n'a aucun impact sur le code assembleur. Ces transformations sont moyennement résistantes et moyennement puissantes, les résultats obtenus

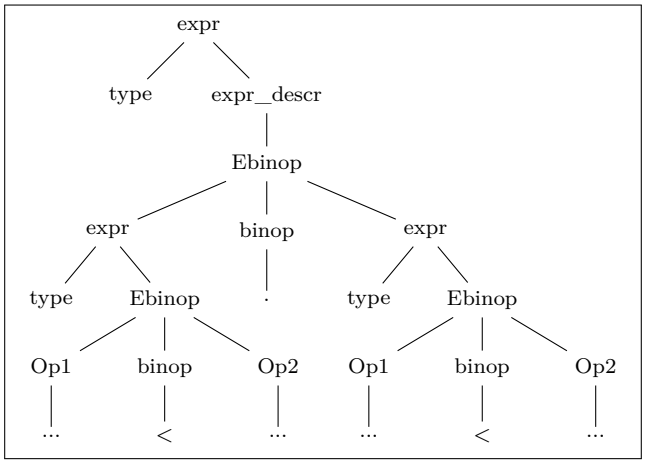


FIG. 5.8 – **Arbre syntaxique de l'opération "inférieur à" obfusqué**

sont aléatoires mais le nombre d'expressions obfusquées reste assez faible. Ces premiers résultats sont très instructifs, les transformations qui ont le plus grand impact sur le code assembleur et celles qui sont inefficaces sont maintenant identifiables. Encore une fois, il faudrait développer ces exemples en ne conservant que les transformations valides, en diversifiant les modifications d'arbres et en ajustant le pourcentage d'expressions obfusquées.

5.2 L'obfuscation des boucles For

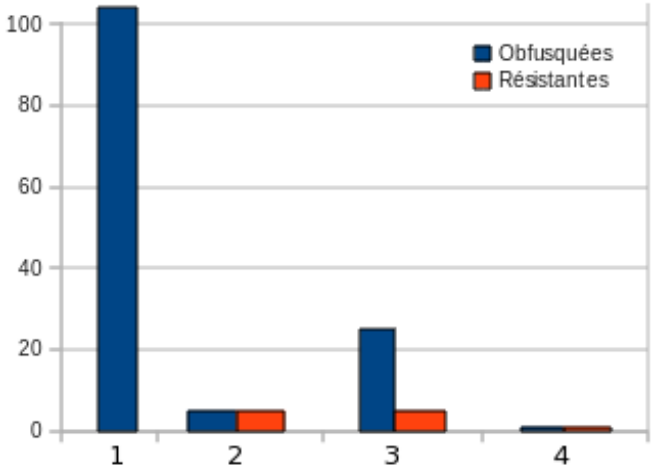
Des méthodes de déroulage et de fission de boucle ont été implémentées afin d'obfusquer les codes contenant ce type de structures.

5.2.1 Déroulage de boucle

La première étape de l'obfuscation de boucle consiste à déterminer si la boucle peut être obfusquée ou non. Pour le déroulage de boucle, le seul critère est d'avoir une définition de boucle classique, afin de pouvoir retrouver aisément la variable et l'intervalle d'itération. Afin de simplifier l'implémentation, un facteur de déroulage de deux a été choisi dans un premier temps. Si la boucle est confirmée comme obfuscable, une nouvelle boucle est implémentée avec un intervalle d'itération divisé par deux. La nécessité d'inclure, ou non, une boucle de reste est déterminée. Finalement, le corps de la boucle est copié et dans la copie la variable d'itération est remplacée par elle-même incrémentée de un. La nouvelle boucle ainsi construite est insérée dans le code. Le principe reste le même pour des déroulages de boucle avec des facteurs différents.

Tests et résultats

Deux de nos programmes de tests contiennent des boucles for, le programme P4 et le programme P6. Le programme P6 contient trois boucles classiques avec des corps contenant deux, trois et quatre instructions. Ces instructions peuvent être des conditionnelles de type if, des opérations sur des tableaux, des appels de fonctions et des opérations sur des variables. Ces boucles sont placées dans la fonction *main()* et dans d'autres fonctions du programme. Le programme



		Obfusquées	Résistantes
1	Constantes Entières/Flottantes	104	0
2	Comparaisons	5	5
3	Opérations arithmétiques	25	5
4	Opérations logiques	1	1

FIG. 5.9 – **Expressions obfusquées et impact sur le code**

```
char tab[];

for(i=0;tab[i]!='\0';i++) { }
```

FIG. 5.10 – **Boucle non obfuscable**

P6 contient deux boucles imbriquées. Le corps commun des deux boucles est composé d'une opération arithmétique entre deux variables et d'un appel de fonction.

P6 : Le code assembleur du programme de test P6 est conforme aux attentes, les deux blocs de code supplémentaires sont présents. Ces blocs sont les duplicatas du code correspondant au corps de la boucle. La dernière boucle n'est pas obfusquée car elle ne possède pas un intervalle d'itération classique, le nombre d'itérations ne peut pas être déterminé. (figure 5.10).

P4 : Des transformations similaires sont présentes dans le code du programme P4. Un premier bloc de code supplémentaire correspond au déroulage de la boucle intérieure puis un second bloc correspond au déroulage de la boucle extérieure. Cette transformation est très efficace sur ce type de boucle, la taille du code est augmenté de façon conséquente et le programme est grandement complexifié tout en gardant un coût en temps d'exécution raisonnable.

Comme déclaré précédemment, une transformation de type déroulage de boucle ne possède ni une grande résilience et ni une forte puissance. Ce type de transformation est identifié très rapidement par les outils automatiques (désobfuscateur ou désassembleur), elles peuvent donc être

renversées très simplement. Cette lacune devrait être comblée en appliquant une transformation de type fission sur les boucles préalablement déroulées.

5.2.2 Fission de boucle

De même que pour le déroulage de boucle, la première étape consiste à déterminer si les boucles compilées sont obfuscables ou non. Il est indispensable de s'assurer qu'il n'y a pas de dépendance de variables entre les expressions afin que nos transformations ne faussent pas les calculs. Prenons le cas d'une variable modifiée dans la première expression de la boucle *for*, si la valeur est réutilisée dans l'expression suivante, ces deux instructions sont dépendantes, elles ne peuvent donc pas être placées dans deux boucles différentes (fission impossible). Comme pour le déroulage de boucle, la définition de la boucle doit avoir une forme classique : il est nécessaire de connaître l'intervalle d'itération et de récupérer la variable de boucle afin de pouvoir la réutiliser.

Tests et résultats

Le programme de test P6 contient trois boucles différentes, mais ces boucles étant constituées d'un unique énoncé (une condition if), une transformation de boucle de type fission est impossible : deux énoncés sont nécessaires au minimum. Le programme P4 contient une boucle qui peut être partagée en deux. Le code assembleur obfusqué contient sept lignes supplémentaires, le corps de la boucle n'est pas recopié, seulement les instructions correspondant à la boucle. Comme pour le déroulage de boucle, la fission de boucle est identifiée très rapidement, cette transformation ne possède donc ni une grande résilience, ni une grande puissance.

5.2.3 Fission et déroulage de boucle

L'objectif ensuite était de tester l'efficacité de ces transformations lorsqu'elles sont appliquées ensemble. Afin de comprendre parfaitement comment le compilateur gérait ces deux techniques, un programme de test supplémentaire a été implémenté. Il est simplement composé d'une boucle *for* et de deux instructions (figure 5.11). Le code assembleur issu de ce programme en langage C est facilement compréhensible, les deux initialisations de variables sont visibles de même que les composantes de la boucle et les deux opérations arithmétiques (figure 5.12).

La structure du code obfusqué par déroulage de boucle (figure 2.2) est similaire (figure 5.13). Les labels et les instructions de saut sont identiques, par contre la variable d'itération est incrémentée de deux. Comme prévu, le corps de la boucle est différent, deux itérations sont réalisées en même temps. L'espace d'itération étant impair, une boucle de reste est présente à la fin du programme.

Le code assembleur obfusqué par fission de boucle (2.2) contient deux labels supplémentaires, pour le début et la fin de la nouvelle boucle. La deuxième expression, $b = b + i$ a été déplacée dans le corps de la seconde boucle (5.14)

Le code assembleur obfusqué à la fois par fission et par déroulage (5.15) est beaucoup plus complexe, le nombre de labels et d'instructions de saut est le même que dans le code assembleur obfusqué par fission. Les corps de boucle sont plus compliqués, les deux itérations du déroulage de boucle sont présentes, l'espace d'itération est modifié et la variable d'itération est incrémentée de deux. Finalement, pour chaque nouvelle boucle, une boucle de reste a été ajoutée.

Cet exemple confirme le fait que les transformations appliquées seules, ne sont pas très efficaces mais combinées leur puissance et leur résilience deviennent beaucoup plus importantes. Le coût de ce type de transformation reste faible.

```
a=0;
b=10;

for(i=0;i<5;i++)
{
    a=a+i;
    b=b+i;
}
```

FIG. 5.11 – **Obfuscation de boucle : code du programme de test**

```
main:
    ...
    addi    4, 0, 0  --> a=0
    addi    5, 0, 10 --> b=10
    addi    3, 0, 0  --> i=0
.L100:
    cmpwi   0, 3, 5  --> si i>=5
    bf      0, .L101 --> saut L101 (sortie de boucle)
    add     4, 4, 3  --> a=a+i
    add     5, 5, 3  --> b=b+i
    addi    3, 3, 1  --> i++
    b       .L100    --> sauf L100 (début de boucle)
.L101:
    ...
```

FIG. 5.12 – **Obfuscation de boucle : code assembleur non obfusqué**

```
main:
    ...
    addi    4, 0, 0  --> a=0
    addi    5, 0, 10 --> b=0
    addi    3, 0, 0  --> i=0
.L100:
    cmpwi   0, 3, 4  --> si i>=4
    bf      0, .L101 --> saut L101 (sortie de boucle)
    add     4, 4, 3  --> a=a+i
    add     5, 5, 3  --> b=b+i
    add     6, 4, 3  --> t=a+i (a =
    addi    4, 6, 1  --> a=t+1      a+i+1)
    add     5, 5, 3  --> b=b+i (b =
    addi    5, 5, 1  --> b=b+1      b+i+1)
    addi    3, 3, 2  --> i=i+2
    b       .L100    --> saut L100 (début de boucle)
.L101:
    add 4, 4, 3      --> a=a+i (boucle
    add 5, 5, 3      --> b=b+i         de reste)
    ...
```

FIG. 5.13 – **Obfuscation de boucle : code assembleur obfusqué par déroulage de boucle**

```
main:
    ...
    addi    4, 0, 0     --> a=0
    addi    5, 0, 10    --> b=10
    addi    3, 0, 0     --> i=0
.L100:
    cmpwi   0, 3, 5     --> si i>=5
    bf      0, .L101    --> saut L101 (sortie de boucle)
    add     4, 4, 3     --> a=a+i
    addi    3, 3, 1     --> i++
    b       .L100       --> sauf L100 (début de boucle)
.L101:
    addi    3, 0, 0     --> i=0
.L102:
    cmpwi   0, 3, 5     --> si i>=5
    bf      0, .L103    --> saut L103 (sortie de boucle)
    add     5, 5, 3     --> b=b+i
    addi    3, 3, 1     --> i++
    b       .L102       --> saut L102 (début de boucle)
.L103:
    ...
```

FIG. 5.14 – **Obfuscation de boucle : code assembleur obfusqué par fission de boucle**

5.3 Conclusion

Les obfuscations au niveau du langage C sont plus simples à mettre en œuvre, les structures fournies par le compilateur sont utilisées mais l'impact sur le langage assembleur est difficilement prévisible. Les transformations d'expressions sont rarement conservées. Lorsqu'elles le sont, l'expression d'origine est difficilement retrouvable. De nouvelles modifications d'arbres doivent être mises au point afin de rendre le code assembleur généré plus aléatoire.

Les transformations de boucles appliquées simultanément possèdent une résilience et une puissance importantes. Leur coût est quasiment nul. Les déroulages de boucles avec des valeurs supérieurs à deux doivent être implémentés mais les transformations sont viables et efficaces.

Les transformations de code doivent être optimisées, les modifications d'arbres syntaxiques doivent être diversifiées et l'utilisateur doit avoir la possibilité de dérouler les boucles d'un facteur supérieur à deux mais ces modifications sont efficaces et possèdent un coût faible.

```
main:
    ...
    addi    4, 0, 0     --> a=0
    addi    5, 0, 10    --> b=0
    addi    3, 0, 0     --> i=0
.L100:
    cmpwi   0, 3, 4     --> si i>=4
    bf      0, .L101    --> saut L101 (sortie de boucle)
    add     4, 4, 3     --> a=a+i
    add     6, 4, 3     --> t=a+i (a =
    addi    4, 6, 1     --> a=t+1      a+i+1)
    addi    3, 3, 2     --> i=i+2
    b       .L100       --> saut L100
.L101:
    add     4, 4, 3     --> a=a+i (boucle de reste)
    addi    3, 0, 0     --> i=0
.L102:
    cmpwi   0, 3, 4     --> si i>=4
    bf      0, .L103    --> saut L103 (sortie de boucle)
    add     5, 5, 3     --> b=b+i
    add     5, 5, 3     --> b=b+i (b =
    addi    5, 5, 1     --> b=b+1      b+i+1)
    addi    3, 3, 2     --> i=i+2
    b       .L102       --> saut L102 (début de boucle)
.L103:
    add 5, 5, 3         --> b=b+i (boucle de reste)
    ...
```

FIG. 5.15 – **Obfuscation de boucle : code assembleur obfusqué par fission et déroulage de boucle**

Conclusion

La première partie de ce document explique qu'il est indispensable de protéger les codes sources contre les attaques de type rétro-ingénierie. L'une des techniques suggérée est l'obfuscation de code. L'objectif était donc d'implémenter des obfuscations de code dans le compilateur CompCert et d'étudier les comportements des programmes transformés par rapport à ceux des programmes dont ils sont issus.

La seconde partie de ce rapport décrit la façon dont ces transformations ont été réalisées et analyse leurs impacts sur le code assembleur généré. Les transformations implémentées au niveau assembleur sont plus compliquées à mettre en œuvre. Le risque d'altérer l'application est plus important. Néanmoins, l'emprise sur le code assembleur généré est plus importante. Les résultats obtenus avec les transformations effectuées au niveau du langage C sont plus incertains. L'impact réel qu'elles auront sur le code généré n'est pas connu à l'avance.

Même si ces transformations ne sont pas abouties, les résultats sont prometteurs. Il reste de nombreuses transformations à développer, il faudrait étudier l'interaction de ces transformations entre elles et déterminer quelle séquence de transformations est idéale en fonction des circonstances. La relation efficacité / coût doit être analysée plus attentivement, il doit être possible de déterminer pour une catégorie de code, quelle transformation permet d'obtenir la meilleure efficacité avec le coût le plus raisonnable possible.

Enfin, la dernière étape du stage sera consacrée à l'étude sémantique des transformations d'obfuscation. Ce travail doit permettre de vérifier de façon plus rigoureuse que les transformations ne modifient pas le comportement des applications. L'outil d'étude sémantique sera implémenté directement dans le compilateur et devra fonctionner automatiquement lors de la compilation.

La lutte contre la rétro-conception n'est pas la seule application de l'obfuscation de code. Cette technique peut être utilisée pour tracer les logiciels piratés. Le développeur distribue une version différente du code de son application aux utilisateurs. Il conserve un registre dans lequel est indiqué à qui chaque version du code a été vendue. Si la copie piratée est distribuée sur Internet, il peut connaître très simplement qui a acheté la version originale. Évidement les pirates peuvent eux aussi utiliser l'obfuscation de code, ils différencient de la sorte le code original et leur copie. Ils peuvent ainsi prétendre avoir développé leur propre version de l'application.

Bibliographie

[1] C.Collberg, C.Thomborson, et D.Low. ***A Taxonomy of Obfuscation Transformation*** . 1997. Technical report 148.

[2] C.Collberg, C.Thomborson, et D.Low. ***Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs***. POPL 1998 : 184-196.

[3] C.Collberg, C.Thomborson, et D.Low. ***Abstractions and Unstructuring Data Structures***. ICCL 1998 : 28-38.

[4] Stéphanie Riaud. ***Transformations de programmes pertinentes pour la sécurité du logiciel***. Bibliographie Master Recherche en Informatique, Rennes 2011. ftp ://ftp.irisa.fr/local/caps/DEPOTS/BIBLIO2011/Riaud_Stephanie.pdf

[5] Xavier Leroy. ***Formal Verification of a Realistic Compiler***. Communications of the ACM 52(7), 2009.

[6] Xavier Leroy. ***The CompCert compiler***. Version 1.8, Septembre 2010, http ://compcert.inria.fr/

[7] Sandrine Blazy, Xavier Leroy. ***Mechanized semantics for the Clight subset of the C language***. JAR 2009. 43(3), October 2009.

[8] G.Balakrishnan, T.Reps, D.Melski, et T.Teitelbaum. ***WYSINWYX : What You See Is Not What You eXecute*** . VSTTE 2005 : 202-213

[9] H.Peter et V.Vliet ***Crema–The Java obfuscator***.
http ://web.inter.nl.net/users/H.P.van.Vliet/crema.html,January 1996.

[10] IBM ***PowerPC Microprocessor Family : The programming Environments for 32-Bit Microprocessors***.
https ://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2.
G522-0290-01, Février 2000

[11] IBM ***Developing PowerPC Embedded Application Binary Interface (EABI). Compliant Programs*** .
Microcontroller Applications, IBM Microelectronics, Research Triangle Park, NC, ppcsupp@us.ibm.com Version : 1.0, September 21, 1998

Table des figures