# Techniques d’obfuscation de code : chiffrer du clair avec du clair

La plupart des infections informatiques (vers, virus, chevaux de Troie, etc.) utilisent désormais le chiffrement pour changer de peau. Elles utilisent également d'autres types de transformations pour rendre la vie plus difficile aux " laborantins " des sociétés éditrices de produits antivirus. Vous découvrirez dans ces pages quelques-unes de ces techniques.

# Introduction

Les transformations d'obfuscation sont utilisées et donc étudiées depuis longtemps dans les cartes à puce. Elles sont depuis peu utilisées pour la protection (purement logicielle) de contenu et la gestion numérique des droits.
Cette famille de mécanismes, à défaut d'être purement cryptographique, est présente dans la cryptographie opérationnelle pour assurer la protection des clés et autres secrets. Nous expliquons son utilité pour les virus.
Après avoir donné une définition formelle de la notion d'obfuscation, nous examinons quelques techniques d'obfuscation, tentons d'en établir une classification et donnons quelques critères pour mesurer leur pertinence. De nombreux exemples illustrent l'exposé (en C dans la mesure du possible).

## Définition : obfuscation

Un obfuscateur $\mathcal{T}$ peut être vu comme un compilateur particulier. Il prend en entrée un programme ou un circuit P et produit un nouveau programme $\mathcal{T}(P)$ qui possède deux propriétés :

- $\mathcal{T}(P)$ a les mêmes fonctionnalités que P. En d'autres termes, $\mathcal{T}(P)$ calcule les mêmes fonctions que P.
- $\mathcal{T}(P)$ est inintelligible, dans le sens où $\mathcal{T}(P)$ constitue une boîte noire virtuelle.

Une telle boîte noire virtuelle $\mathcal{T}(P)$ est caractérisée, du point de vue de la théorie de la complexité, par le fait que quiconque est capable de réaliser un calcul effectif au moyen de $\mathcal{T}(P)$, peut alors réaliser ce même calcul en ayant accès par le biais d'un oracle au programme P (c'est-à-dire en pouvant disposer d'une infinité de couples (entrée/sortie) du programme). Cela signifie simplement que si quelqu'un est capable de faire des calculs avec $\mathcal{T}(P)$, alors il peut réaliser ces mêmes calculs en observant uniquement les entrées/sorties du programme P.
Nous parlerons plus loin des implications théoriques de cette définition.

## Un mécanisme d'utilité virale

Pour les virus, l'obfuscation de code (permet de rendre le code viral difficile à comprendre) fait partie des mécanismes de base, au même titre que la furtivité (permet de rendre le code viral difficile à localiser sur le système) ou le polymorphisme (permet de rendre difficile l'identification par signature unique d'un programme viral).
Les transformations permettant de remplacer du code en clair avec du code en clair fonctionnellement équivalent, présentent un intérêt évident pour les virus :

- Elles permettent de contourner les moteurs d'analyse spectrale ou tout autre moteur fondé sur une mesure de l'entropie (ces transformations conservent le plus souvent la distribution des

opcodes ainsi que l'entropie. Le code résultant d'une opération de chiffrement classique sera identifié comme suspect par de tels moteurs de détection).
- Elles permettent de protéger les clés et l'algorithme de chiffrement utilisé, le cas échéant. Le résultat de ces transformations est un camouflage de certaines structures de données et certaines zones de code (ces transformations participent alors à la gestion des clés (et peuvent être assimilées à ce titre à des primitives de nature cryptographique).
- De manière générale, les transformations d'obfuscation de code permettent de protéger les données critiques statiques du virus, c'est-à-dire les valeurs qui ne doivent pas pouvoir être modifiées, qui doivent rester secrètes ou qui sont indispensables à l'exécution sécurisée des routines vitales du virus (contrôle d'intégrité, détection d'une exécution en mode pas à pas, de la présence d'un débogueur sur le système ou d'une exécution sous émulateur ou système de virtualisation, etc.).

Le code permettant de réaliser une transformation d'obfuscation peut facilement être modifié afin d'assurer également une fonction de diversification (réalisation d'instances équivalentes fonctionnellement au programme original et difficiles à analyser, fondée sur l'utilisation conjointe d'une ou plusieurs sources d'aléa du système hôte). Cette amélioration permet de participer au polymorphisme et d'augmenter l'effort requis pour lever les protections et automatiser le processus de désinfection. Elle permet donc d'augmenter les chances de survie du programme viral.

# Un mécanisme cryptographique ?

La cryptographie malicieuse ou cryptovirologie peut se définir comme l'étude des mécanismes cryptographiques dans le contexte de leur utilisation par des infections informatiques [21]. Des mécanismes cryptographiques classiquement dédiés à la sécurité ou à la sûreté de fonctionnement sont pervertis dans leur utilisation et appliqués au développement d'infections informatiques toujours plus robustes.
Des techniques de génération environnementale de clés de chiffrement, fondées sur l'utilisation de fonctions à sens unique et initialement développées pour augmenter la résistance des agents logiciels mobiles [18], vont également pouvoir s'appliquer au développement de programmes viraux hautement résistants à l'analyse par rétro-conception (les virus dirigés [10]).
Ces techniques purement cryptographiques permettent de résoudre le problème de la protection des clés de chiffrement, mais au prix d'une perte d'indépendance du programme viral (le virus est dirigé par son concepteur). Pour pallier ce type de problème, les techniques de protection de contenu comme les transformations d'obfuscation apparaissent indispensables, d'un point de vue opérationnel.
On distingue en général le chiffrement de l'obfuscation : une transformation d'obfuscation, même si elle est le plus souvent paramétrée par une clé (voir par exemple [2]), effectue par le biais de substitutions successives, la transformation d'un message clair en un autre message clair.
Les primitives cryptographiques d'obfuscation des données, telles que les permutations paramétrées par une clé, sont des contre-mesures naturelles aux attaques non intrusives logiques (voir par exemple [12] pour une description de ces attaques). Ces primitives sont très adaptées à la réalisation de brouilleurs matériels dans les cartes à puce : brouillage du bus de données entre le microprocesseur et la mémoire ou entre le CPU et le cryptoprocesseur, brouillage de la RAM. Ces primitives peuvent être incorporées dans des fonctions de brouillage de données non linéaires plus évoluées. Appliquant le paradigme confusion/diffusion de Shannon, ces primitives peuvent être utilisées sous la forme de petites matrices de substitution en couches alternées avec des fonctions affines. Ce type de construction n'assure cependant pas le même niveau de sécurité que l'utilisation d'un système de chiffrement par blocs classique.

## Virologie et protection de contenu : un combat unifié contre la rétro-ingénierie

La virologie informatique et la problématique de protection de contenu sont étrangement symétriques dans l'utilisation de la cryptographie :

- Un programme viral implémente des mécanismes cryptographiques pour contourner les logiciels antivirus et ralentir l'analyse de son code.
- Un logiciel implémente des mécanismes comparables pour protéger une licence d'utilisation ou protéger le contenu ou les secrets de conception de l'application contre l'analyse.

La communauté des développeurs de virus montre une grande motivation quant à l'utilisation de techniques d'obfuscation de code. Leur approche est aujourd'hui encore très empirique, mais nous pouvons sentir d'ores et déjà une volonté d'augmenter le niveau de modularité et de sophistication des programmes viraux [22].
De nouvelles techniques cryptographiques naissent du besoin plus impérieux d'assurer la protection d'une application par voie purement logicielle. Nous pouvons prédire que ces techniques seront perverties par les concepteurs de virus.

# Qualification des transformations d'obfuscation

Après avoir rappelé les résultats théoriques les plus importants concernant les transformations d'obfuscation, nous donnons des critères théoriques et empiriques permettant de qualifier une transformation d'obfuscation.

## Obfuscation : la théorie

Nous avons vu en introduction qu'un obfuscateur $\mathcal{T}$ doit rendre le programme $\mathcal{T}(P)$ inintelligible dans le sens où tout ce que l'on peut apprendre du programme $\mathcal{T}(P)$, en ayant accès à son code et à ses états internes lors de son exécution, peut être obtenu uniquement à partir des entrées/sorties du programme P. En d'autres termes, tout ce que l'on peut espérer apprendre lors d'une analyse du programme protégé, on peut l'apprendre en exécutant le programme et en observant ses entrées/sorties.
Avec une telle définition, et en construisant une famille de fonctions qui ne sont pas obfuscables, il est démontré qu'un obfuscateur parfait, c'est-à-dire qui puisse rendre un programme inintelligible au sens défini précédemment, n'existe pas [1].
Cela ne signifie pas qu'il n'y a pas de méthodes permettant de rendre un programme inintelligible dès lors qu'un sens moins absolu est donné à ce terme. Une définition affaiblie de la notion d'obfuscation, c'est-à-dire une définition qui évite le paradigme de la boîte noire virtuelle, peut être donnée (en remplaçant le paradigme de la boîte noire virtuelle par une propriété plus faible, assurant l'existence d'un obfuscateur $\mathcal{T}$ qui, étant donné deux circuits C1 et C2 calculant la même fonction, assure que $\mathcal{T}(C1)$ et $\mathcal{T}(C2)$ sont impossibles à distinguer l'un de l'autre).

## Qualification : critères théoriques

Même si un obfuscateur parfait n'est théoriquement pas réalisable, une évaluation au regard de la théorie de la complexité/décidabilité reste indispensable pour qualifier complètement une

transformation d'obfuscation.
L'idée est de prouver que la transformation permet d'augmenter suffisamment la difficulté d'une analyse de type boîte noire (reconstruction du graphe des états par observation des entrées/sorties) en diversifiant les sorties. Chaque transition d'état lors de l'exécution d'un programme dissémine une certaine quantité d'information dans son environnement. La somme de ces informations peut être suffisante à un observateur pour déterminer l'espace des états du programme. La quantité moyenne d'information fournie par chaque observation doit être la plus petite possible, afin d'augmenter l'effort nécessaire à la désobfuscation.
Par exemple, dans le cas des transformations inter-procédurales utilisant des tableaux de pointeurs de fonctions (voir ci-dessous), il est possible d'apporter une preuve théorique de leur efficacité : la complexité de l'analyse inter-procédurale d'un programme protégé par ce type de primitive croît de manière exponentielle avec la taille du programme [16].

# Qualification : critères empiriques

De nombreuses transformations d'obfuscation sont très utiles, même s'il n'est pas toujours possible de prouver leur pertinence au regard de la théorie de la complexité. Des critères plus souples permettent de décrire et de classifier ces transformations. Ces critères, même s'ils reposent sur des mesures de complexité, sont empiriques.
Nous donnons ci-dessous quelques critères permettant d'évaluer l'efficacité d'une transformation d'obfuscation.

## La puissance

Étant donné une transformation d'obfuscation T, nous pouvons définir le niveau d'obfuscation comme le degré de gêne imposée par cette transformation à un attaquant humain. Étant donné une métrique μ, notons cμ(P) la complexité du programme P au regard de cette mesure de la complexité. Pour fixer les idées, la métrique μ considérée pour mesurer la complexité d'un programme peut être par exemple la longueur du programme. cμ(P) augmente alors avec le nombre d'instructions (opérateurs et opérandes) dans P. D'autres métriques, empruntées à la théorie de l'optimisation, pourront être trouvées dans [5].
Nous définissons alors la puissance d'une transformation d'obfuscation T relativement à un programme P par :

$$\mathcal{T}_{pot}(P) = \frac{c_\mu(\mathcal{T}(P))}{c_\mu(P)} - 1$$

## La résistance

Étant donné une transformation d'obfuscation T, nous pouvons définir le niveau d'obfuscation comme une mesure de l'effort requis pour défaire la transformation. Nous pouvons identifier deux axes d'effort :

- L'effort de développement d'un outil automatique de désobfuscation capable de réduire effectivement la puissance de la transformation.
- La quantité de mémoire et de cycles CPU requis par l'outil automatique de désobfuscation.

Ces deux axes permettent de définir une matrice **R** dont les valeurs correspondent aux résistances

d'une transformation d'obfuscation T relativement à un programme P :

$$\mathcal{T}_{res}(P) = R(\mathcal{T}_1(P), \mathcal{T}_2(P)$$

où T1(P) donne une mesure de l'effort de développement requis pour mettre au point l'outil, en fonction du périmètre de la transformation (locale à un bloc d'instruction du graphe CFG, globale, inter-procédurale, inter-processus) et T2(P) une mesure de l'effort requis par l'outil (polynomial, exponentiel).

## Le coût

Le coût Tcost(P) d'une transformation d'obfuscation T (appliquée à un programme P) correspond à une mesure de la consommation supplémentaire, en termes de ressources (temps CPU/espace mémoire), induite par la protection. Le coût augmente avec la complexité induite (constante, linéaire, polynomiale, exponentielle).

## La furtivité

La définition de ce critère vient de l'observation suivante : si une transformation introduit du code nouveau dans le programme protégé T(P), un attaquant pourra facilement le détecter et exploiter cette faiblesse pour lever la protection. Nous définissons donc un ensemble de caractéristiques du langage utilisé par un programme, (P). Notons (T) l'ensemble des caractéristiques du langage introduites par la transformation d'obfuscation T. Nous définissons la furtivité de la transformation T relativement au programme P par :

$$\mathcal{T}_{ste}(P) = 1 - \frac{|\mathcal{L}(T) - \mathcal{L}(P)|}{|\mathcal{L}(T)|}$$

Si la transformation n'enrichit pas le langage des programmes qu'elle protège (on a alors (T)=0), la furtivité Tste(P) est maximale et égale à 1.

## La qualité

Nous pouvons à présent définir la qualité d'une transformation d'obfuscation relativement à un programme P par :

$$\mathcal{T}_{qual}(P) = f(\mathcal{T}_{pot}(P), \mathcal{T}_{res}(P), \mathcal{T}_{cost}(P), \mathcal{T}_{ste}(P))$$

où f est une fonction décroissante du coût (les autres paramètres étant fixés) et croissante des autres critères.

## Exemple de mesure de qualité

Nous pouvons par exemple définir la qualité d'une transformation par :

$$\mathcal{T}_{app}(P) = \frac{\omega_1\mathcal{T}_{pot}(P) + \omega_2\mathcal{T}_{res}(P) + \omega_3\mathcal{T}_{ste}(1}{\mathcal{T}_{cost}(P)}$$

où ωi,i=1,2,3 sont des constantes fixées en fonction du contexte opérationnel d'utilisation des transformations d'obfuscation
(ici, nous avons choisi :

$$f(x_1, x_2, x_3, x_4) = x_3^{-1}(\omega_1 x_1 + \omega_2 x_2 + \omega_3 x_4)$$

# Exemples de transformations

Nous avons vu que la définition de transformations d'obfuscation se fonde pour partie sur des problèmes ouverts. Il est donc difficile dans ces conditions de dégager les spécifications précises et universelles de ce que doit être une bonne transformation. Nous avons pris le parti de présenter le principe de quelques transformations, sans détailler le fonctionnement du compilateur sous-jacent.
Un moteur d'obfuscation peut être vu comme un compilateur/décompilateur : le code à brouiller doit être décompilé vers une représentation intermédiaire permettant de représenter le programme sous forme adéquate (sous la forme d'un arbre, sous la forme de pseudo-code assembleur). Cette représentation intermédiaire doit être pensée de manière à faciliter l'application des transformations successives. Une fois ces transformations appliquées, le programme doit le plus souvent être recompilé.
Lorsqu'un programme viral mute, il peut utiliser un moteur d'obfuscation. Le code de ce moteur peut agir en divers endroits du code viral : au sein de son loader, lors du chargement du code viral en mémoire (pour rendre difficile l'analyse de la mémoire), au niveau de la routine de recopie (par exemple pour dissimuler le code viral au sein du programme cible), etc. Le code du moteur d'obfuscation sera pensé différemment en fonction de sa tâche au sein du code viral.
Le lecteur intéressé par les spécifications et l'implémentation d'un moteur d'obfuscation virale pourra par exemple se reporter au code du moteur viral Mistfall [22]. Ce moteur permet de désassembler, modifier (en insérant des instructions supplémentaires) puis recompiler un programme cible au format PE. Il procède en plusieurs étapes :

- analyse de la structure PE ;
- désassemblage du binaire, instruction par instruction ;
- conversion des informations dans un langage de plus haut niveau, de façon à obtenir une matière plus facile à remodeler ;
- invocation de la transformation proprement dite, dont le but est ici de modifier, par ajout ou suppression, cette liste d'instructions ;
- ré-assemblage du code.

Nous présentons maintenant un petit catalogue des transformations d'obfuscation les plus connues. Ces transformations sont librement adaptées de [8], [9], [15] et [20].

## Brouillage élémentaire du flux des instructions

Un programme difficile à analyser doit posséder au moins les propriétés suivantes :

- Le début et la fin du programme doivent être difficiles à localiser.
- Le code ne possède pas de motif facilement identifiable.
- La structure des données n'est pas parlante. Aucune cohérence ne ressort de la séquence des données.

L'idée est donc de convertir toute séquence d'instruction pouvant servir à se repérer dans le programme, c'est-à-dire toute séquence d'instruction identifiée à partir de laquelle il est possible de forger une signature, en une séquence d'instruction rendant impossible la recherche de motif, ce sans modifier l'algorithme original. Les techniques sont nombreuses :

- Des instructions inutiles sont insérées.
- Le programme se déchiffre lui-même.

Pour rendre un flot d'instructions difficile à comprendre, plusieurs méthodes existent. Parmi elles, on a :

- le remplacement d'une instruction par d'autres ;
- le brouillage du flot des instructions.

Ces opérations permettent de rendre le programme très difficile à tracer.

## Optimisation du code

Le but est ici de supprimer les redondances de code, qui fournissent une aide à l'analyse. Le processus de protection par obfuscation doit être non réversible. De manière générale, l'optimisation de code lors de la compilation d'un programme laisse moins d'informations à l'attaquant.
Ce type de transformation étant à sens unique (les informations sont supprimées du programme et ne pourront pas être retrouvées), la résistance Tres face à un outil de désobfuscation est maximale. La puissance Tpot opposée à un analyste est variable, en fonction de la nature des informations supprimées. Le coût Tcost est nul.

## Mélange des instructions

Voyons un exemple de brouillage élémentaire d'un flot d'instructions. Prenons le bloc d'instruction suivant :

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit               db "hi!",0
res               db 0
.CODE
main:
      JMP   STEP
      call  ExitProcess, 0
STEP:
      call  MessageBoxA, 0, offset tit, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 1.1 : Code avant transformation.

On remplace certaines instructions par une séquence équivalente d'instructions.

```
option casemap:none
include r1.inc
.DATA
tit               db "hi!",0
res               db 0
.CODE
main:
      mov   eax, offset STEPI + 135
```

```
      add   eax, -135
      jmp   eax
      call  ExitProcess, 0
STEPI:
      mov   ebx, offset STEP - 156
      add   ebx, 4
      add   ebx, 156
      sub   ebx, 4
      jmp   ebx
      call  ExitProcess, 0
STEP:
      call  MessageBoxA, 0, offset tit, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 1.2 : Code après remplacement de certaines des instructions.

L'étape suivante consiste à mélanger les instructions.

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit               db "hi!",0
res               db 0
.CODE
main:
      mov   ebx, offset STEP - 156        ; j1
      mov   eax, offset STEPI + 135       ; ji1
      add   ebx, 4                        ; j2
      add   ebx, 156                      ; j3
      add   eax, -135                     ; ji2
      sub   ebx, 4                        ; j4
      jmp   eax                           ; ji3
      call  ExitProcess, 0                ; ji4
STEPI:
      jmp   ebx                           ; j5
      call  ExitProcess, 0                ; j6
STEP:
      call  MessageBoxA, 0, offset tit, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 1.3 : Code après mélange des instructions.

Les opérations sur le code compilé sont donc les suivantes :

- optimisation du code, afin de supprimer le plus possible d'informations ;
- remplacement des instructions complexes par des instructions plus fondamentales ;
- mélange des instructions.

Ces trois opérations ont pour effet d'effacer les empreintes d'un programme et la fréquence d'utilisation des registres, qui devient plus uniforme.

La résistance Tres de ce type de transformation face à un outil de désobfuscation est élevé, dans la mesure où ces transformations sont difficilement réversibles. La puissance Tpot opposée à un analyste est également assez élevée (l'uniformisation du jeu des instructions et le mélange de celles-ci rendent la compréhension du programme plus laborieuse). Le coût Tcost est généralement faible.

## Insertion de portions de code inutiles

L'insertion de portions de code inutiles permet également de générer un code plus difficile à tracer. Il y a deux manières d'insérer du code factice :

- en ajoutant du code supplémentaire ;
- en insérant un branchement conditionnel vers un bloc de code recopié depuis le programme lui-même.

Un code factice doit être suffisamment complexe pour ne pas être supprimé lors d'une phase d'optimisation du code compilé. L'insertion d'un branchement conditionnel vers un bloc recopié depuis le programme doit être masqué par les opérations de brouillage vues précédemment.

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit               db "a=",0
res               db 0
.CODE
main:
      mov   eax, 1
      push  eax
      mov   ebx, eax
      shl   ebx, 1
      add   eax, ebx
      pop   ebx
      add   eax, '0'
      mov   dword ptr [res], eax
      call  MessageBoxA, 0, offset res, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 2.1 : Code avant transformation.

Nous pouvons insérer du code factice par ajout d'instructions supplémentaires.

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit               db "a=",0
res               db 0
.CODE
main:
      mov   eax, 1
      push  eax
      mov   ebx, eax
```

```
      inc   ecx
      shl   ebx, 1
      sub   eax, ecx
      add   eax, ebx
      add   ecx, -1
      add   eax, ecx
      add   eax, 1
      pop   ebx
      add   eax, '0'
      mov   dword ptr [res], eax
      call  MessageBoxA, 0, offset res, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 2.2 : Code après ajout d'instructions supplémentaires.

Nous pouvons également insérer du code factice par insertion d'un branchement conditionnel vers un bloc de code recopié depuis le programme lui-même (cf figure 2.3 ci-contre).
Modifier l'ordre d'instructions qui n'interfèrent pas entre elles, altère la structure du code interne du programme compilé, mais n'affecte pas le résultat de son exécution.
Pour ces transformations, la résistance Tres face à un outil de désobfuscation; la puissance Tpot opposée à un analyste et le coût Tcost dépendent fortement de la profondeur à laquelle la transformation est appliquée.

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit               db "a=",0
res               db 0
.CODE
main:
      mov   eax, 1
      push  eax
      mov   ebx, eax
      shl   ebx, 1
      jc    L
      add   eax, ebx
      pop   ebx
      add   eax, '0'
      mov   dword ptr [res], eax
      call  MessageBoxA, 0, offset res, offset tit, MB_OK
      call  ExitProcess, 0
L:
      add   eax, ebx
      pop   ebx
      add   eax, '0'
      mov   dword ptr [res], eax
      call  MessageBoxA, 0, offset res, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 2.3 : Code après insertion d'un bloc recopié depuis le programme.

### Utilisation d'une machine virtuelle

Le principe inhérent à ce type de transformation est de convertir le code en un code à destination d'un processeur virtuel, par utilisation d'une table d'interprétation. Le nouveau code est exécuté par l'interpréteur de la machine virtuel.
Le lecteur intéressé trouvera dans [3] un exemple d'application de cette technique de protection logicielle, parmi d'autres (gestionnaire d'exception anti-débogage, chiffrement par couche, obfuscation par insertion de code inutile).
Pour ce type de transformation, la puissance Tpot opposée à un analyste et la résistance Tres face à un outil de désobfuscation sont élevées, mais au prix d'un coût Tcost qui peut être élevé lui aussi.
Nous allons examiner d'autres types de transformations, qui, comme les précédentes, affectent la représentation interne du programme. Elles ne modifient pas le comportement externe du programme. Ces transformations ont seulement pour objectif de rendre l'analyse du flux de contrôle et celle du flux de données plus difficiles.

# Transformations intra-procédurales

Ces transformations visent à rendre l'analyse intra-procédurale plus difficile. L'analyse intra-procédurale consiste à :

- construire le graphe CFG (Control-Flow Graph) du flux de contrôle. Cette étape correspond donc à l'analyse du flot de contrôle d'une procédure. Un graphe CFG correspond à des nœuds (les blocs d'instructions de base) et des arêtes (indiquent les transferts de contrôle entre blocs).
- analyser des flux de données sur les données relativement à l'arbre CFG.

Ces transformations reposent essentiellement sur deux techniques :

- dégénérescence du flux de contrôle du programme via transformation des branches statiques en instructions de branchements dynamiques sur la base des valeurs de registres (cette opération permet d'augmenter la complexité d'une analyse statique du flux de contrôle);
- introductions intensives d'alias pour les données.

### Dégénérescence du flux de contrôle

Considérons le programme suivant :

```
#include <stdio.h>
int main() {
int a=1, b=1;
while (a<10) {
   b+=a;
   if (b>10)
        b--;
   a++;
}
printf("b=%d\n", b);
}
```

Figure 3.1 : Code avant transformation.

Les structures de contrôles de haut niveau sont converties en leur équivalant **if-then-goto**

```
#include <stdio.h>
int main() {
int a=1, b=1;
L1:
  if (a>=10)
       goto L4;
  b+=a;
  if (b<=10)
       goto L2;
  b--;
L2:
  a++;
  goto L1;
L4:
printf("b=%d\n", b);
}
```

Figure 3.2 : Code après remplacement des structures de contrôles de haut niveau.

Les adresses de branchement des instructions **goto** sont déterminées dynamiquement. Cette méthode peut être implémentée en remplaçant chaque **goto** par une instruction **switch** et en affectant la variable de contrôle du **switch** dynamiquement dans chaque sous-bloc de code pour décider quel bloc sera exécuté ensuite.

```
#include <stdio.h>
int main() {
int a, b , sw;
sw=1;
s:
switch(sw) {
case 1:
   a=1, b=1;
   sw=2;
   goto s;
case 2:
   if (a>=10)
        sw=6;
   else
        sw=3;
   goto s;
case 3:
   b+=a;
   if (b<=10)
        sw=5;
   else
        sw=4;
   goto s;
case 4:
   b--;
   sw=5;
   goto s;
case 5:
```

```
   a++;
   sw=2;
   goto s;
case 6:
   break;
}
printf("b=%d\n", b);
}
```

Figure 3.3 : Code après dégénérescence du flux de contrôle.

La résistance Tres de ce type de transformation face à un outil de désobfuscation automatique est forte mais pas maximale. La puissance Tpot opposée à un analyste est élevée. Le coût Tcost induit est également élevé.

## Utilisation de tableaux dynamiques

Le principe est d'utiliser une indexation afin de perturber les outils d'analyse statique de code. Nous introduisons un tableau global g[]. Ce tableau est initialisé de la manière suivante :

- Quel que soit k, g[k.n] est congru à c modulo j.
- Les entiers n, j et c sont placés de manière aléatoire dans le tableau g[].
- Les autres éléments du tableau contiennent des valeurs aléatoires.

Après chaque étape de calcul :

- Tout ou partie des g[k.n] sont remplacés par des entiers appartenant à la même classe de congruence.
- Tout ou partie des autres éléments du tableau sont remplacés par des valeurs aléatoires.
- Une fonction de redistribution réorganise le tableau en modifiant la valeur de n et en recalculant les g[k.n]. Cette fonction est appelée périodiquement au cours de l'exécution du programme.

Avec ces modifications, le programme peut sélectionner les branches dynamiquement en utilisant des expressions de complexité arbitraire dans le calcul des valeurs d'index. Le fait que les valeurs du tableau changent périodiquement permet de mettre en échec les outils d'analyse statique de code. La résistance Tres de ce procédé face à un outil de désobfuscation peut être élevée. En revanche, la puissance Tpot opposée à un analyste humain est assez faible, dès lors qu'il a identifié de quelle manière s'effectue le recalcul des valeurs du tableau. Le coût Tcost d'une telle transformation n'est pas élevé.

## Utilisation d'alias

L'utilisation d'alias permet d'augmenter de manière conséquente la complexité de l'analyse d'un flux de données. Nous parlons d'alias lorsque plusieurs noms désignent un même emplacement mémoire. Considérons les deux programmes suivants :

```
#include <stdio.h>
int main(){
int i, *p;
```

```
p=&i;
i=0;
*p=1;
while (i<5) {
      *p+=i;
      i++;
}

printf("i=%d\n", i);
}
#include <stdio.h>
int main(){
int i, *p;
p=(int *)malloc(sizeof(int));
i=0;
*p=1;
while (i<5) {
      *p+=i;
      i++;
}
free(p);
printf("i=%d\n", i);
}
```

Figure 4.0 : Exemple d'utilisation des alias.

Dans le premier cas, qui correspond à l'utilisation d'un alias, i vaut 7. Dans le second cas, i vaut 5. Considérons à présent le programme suivant :

```
#include <stdio.h>
int main(){
int a=1, b=1, *p;
p=&a;
a+=b;
p=&b;
b=3;
printf("a=%d, b=%d, *p=%d\n", a, b, *p);
}
```

Figure 4.1 : Code avant transformation.

Auquel nous appliquons la transformation dont le résultat est :

```
#include <stdio.h>
int main(){
int a=1, b=1, *p, sw;
sw=1;
s:
switch(sw) {
case 1:
   p=&a;
   a+=b;
```

```
   sw=2;
   goto s;
case 2:
   p=&b;
   b=3;
   sw=3;
   goto s;
case 3:
   break;
}
printf("a=%d, b=%d, *p=%d\n", a, b, *p);
}
```

Figure 4.2 : Code après utilisation d'alias.

Un analyseur statique ne sachant pas quel est l'ordre d'exécution des blocs d'instructions sera incapable de décider, à l'issue de son analyse, quelle est la dernière relation d'alias en vigueur. La transition entre les deux blocs d'instructions étant brouillée par utilisation de la technique de calcul d'index décrite précédemment. Toutes les assignations de pointeurs seront donc reportées indifféremment. L'utilisation des alias intra-procéduraux, couplée avec l'utilisation de techniques de dégénérescence du flux de contrôle statique, permet d'induire des erreurs de diagnostics ou des diagnostics imprécis de la part des analyseurs statiques de code.
La résistance $T_{res}$ de ce type de transformation face à un outil de désobfuscation est assez forte. La puissance $T_{pot}$ du procédé face à un analyste humain est cependant plus moyenne. Le coût $T_{cost}$ supplémentaire induit par l'utilisation d'alias n'est pas élevé.

# Transformations inter-procédurales

Ces transformations visent à rendre l'analyse inter-procédurale plus difficile.

## Modification des appels de fonctions

Considérons le programme suivant :

```
#include <stdio.h>
int x;
void fonct2(){printf("fonct2\n");}
void fonct3(){printf("fonct3\n");}
void fonct1() {
      if (x>4)
            fonct2();
      else
            fonct3();
}
int main () {
      x=4; fonct1();
      x=5; fonct1();
}
```

Figure 5.1 : Code avant transformation.

Les appels de fonctions sont transformés en appels indirects via l'utilisation de pointeurs :

```
#include <stdio.h>
int x;
void fonct2(){printf("fonct2\n");}
void fonct3(){printf("fonct3\n");}
void fonct1(void (*fptr1)(),
            void (*fptr2)()) {
      void (*ptr)();
      if (x>4)
            ptr=fptr1;
      else
            ptr=fptr2;
      (*ptr)();
}
int main () {
      x=4; fonct1(fonct2, fonct3);
      x=5; fonct1(fonct2, fonct3);
}
```

Figure 5.2 : Code après utilisation des pointeurs.

L'utilisation de pointeurs de fonctions permet de compliquer l'analyse statique du code. La résistance Tres face à un outil de désobfuscation reste cependant moyenne. La puissance Tpot opposée à un analyste est faible. Le coût Tcost induit n'est pas élevé. L'utilisation des pointeurs de fonction trouve toute sa puissance lorsqu'elle est utilisée dans des tableaux de pointeurs de fonctions (voir ci-dessous).

## Unification des signatures de fonctions

Les signatures (ou prototypes) de fonctions sont unifiées (cela signifie simplement que les prototypes des fonctions sont homogénéisés, au niveau des valeurs de retour et des arguments, dont on normalise le nombre et le type).
Le programme suivant :

```
#include <stdio.h>
int p(int x) {
      return(2*x);
}
int q(int x, float f) {
      return(2*x+1000*f);
}
int main() {
    int x=1, y, z; float f=0.001;
    y = p(x);
    z = q(y, f);
    printf("z=%d\n", z);
}
```

Figure 6.1 : Code avant transformation.

devient, après unification des prototypes de fonction, tel qu'en figure 6.2, présentée ci-contre.
La résistance Tres de ce type de transformation face à un outil de désobfuscation est faible. La puissance Tpot opposée à un analyste est moyenne. L'analyste aura cependant plus de mal à identifier le rôle des fonctions en se basant uniquement sur leur prototype. Le coût Tcostd'une telle transformation est nul.

```
#include <stdio.h>
int p(int x, float f) {
      return(2*x);
}
int q(int x, float f) {
      return(2*x+1000*f);
}
int main() {
    int x=1, y, z; float f=0.001, g;
    y = p(x, g);
    z = q(y, f);
    printf("z=%d\n", z);
}
```

## Modification des signatures de fonctions

Les signatures de fonctions sont modifiées puis unifiées.
Le programme suivant :

```
#include <stdio.h>
typedef struct rec {
  int  f1;
  char f2;
} rec ;
int fonct1(int x, rec r, char *c) {
  printf("%s : %d\n", c,
         x+r.f1+r.f2);
  return(x+r.f1+r.f2);
}
int fonct2(int x, char *c) {
  printf("%s : %d\n", c, 2*x);
  return(2*x);

}
int main() {
  rec r;
  int x=1, z;
  r.f1=1; r.f2='1'-'0';
  z=fonct1(x, r, "f1");
  fonct2(z, "f2");
}
```

Figure 7.1 : Code avant transformation.

devient, après modification et unification des prototypes de fonctions, tel que présenté en figure 7.2. La résistance Tres de ce type de transformation face à un outil de désobfuscation est faible. La puissance Tpot opposée à un analyste est plus forte que précédemment, dans la mesure où les prototypes des fonctions n'apportent plus d'information sur leur rôle dans le programme. Le coût Tcost d'une telle transformation n'est pas élevé.

```
#include <stdio.h>
typedef struct rec {
  int  f1;
  char f2;
} rec ;
int fonct1(int x, void *r, char *c) {
  rec *re=(rec *)r;
  printf("%s : %d\n", c,
         x+re->f1+re->f2);
  return(x+re->f1+re->f2);
}
int fonct2(int x, void *r, char *c) {
  printf("%s : %d\n", c, 2*x);
  return(2*x);

}
int main() {
  rec r;
  int x=1, z;
  r.f1=1; r.f2='1'-'0';
  z=fonct1(x, &r, "f1");
    fonct2(z, &r, "f2");
}
```

Figure 7.2 : Code après modification des prototypes des fonctions.

## Utilisation des vecteurs de pointeurs de fonctions

Considérons le programme suivant :

```
#include <stdio.h>
int main(){
    int a=1, b=2;
    if (a<b)
        a=b;
    b=a+1;
    printf("a=%d, b=%d\n", a, b);
}
```

Figure 8.1 : Code avant transformation.

Nous introduisons les fonctions **fonct1()** et **fonct2()** :

```
#include <stdio.h>
int a=1, b=2;
fonct1(){
  a=b;
}
fonct2(){
  b=a+1;
}
int main(){
   if (a<b)
         fonct1();
   fonct2();
   printf("a=%d, b=%d\n", a, b);
}
```

Figure 8.2 : Code après introduction de fonctions.

Les appels de fonctions sont transformés en appels indirects via l'utilisation de pointeurs :

```
#include <stdio.h>
int a=1, b=2;
int (*fp)();
fonct1(){
  a=b;
}
fonct2(){
  b=a+1;
}
int main(){
   if (a<b) {
         if (a*(a+1)%2==0)
            fp=fonct1;
         else
            fp=fonct2;
         (fp)();
   }
   if ((b-2)*(b-1)*b%6!=0)
         fp=fonct1;
   else
         fp=fonct2;
   (fp)();
   printf("a=%d, b=%d\n", a, b);
}
```

Figure 8.3 : Code après utilisation de pointeurs.

Un tableau **A[10]** de pointeurs de fonctions achève notre construction :

```
#include <stdio.h>
int a=1, b=2;
int (*fp)();
int (*A[10])();
fonct0(){
  return(a*(a-1));
```

```
}
fonct1(){
  a=b;
}
fonct2(){
  b=a+1;
}
int main(){
   A[0]=A[1]=fonct0;
   A[2]=fonct1 ;
   A[3]=fonct2 ;
   A[4]=A[6]=fonct0;
   A[5]=A[9]=fonct1;
   A[7]=A[8]=fonct2;
   fp=A[(fonct0()%2)*a*b];
   if (a<b) {
         if (a*(a+1)%2==0)
            fp= A[(fp()%2)+2];
         else
            fp= A[(fp()%2)+4];
         (fp)();
   }
   if ((b-2)*(b-1)*b%6 !=0)
         fp= A[(fp()%2)+5];
   else
         fp= A[(fp()%2)+3];
   (fp)();
   printf("a=%d, b=%d\n", a, b);
}
```

Figure 8.4 : Code après utilisation d'un tableau de pointeurs de fonctions.

Notons qu'il est possible d'apporter une preuve théorique de l'efficacité de cette dernière famille de transformation : la complexité de l'analyse inter-procédurale d'un programme protégé par ce type de primitive croît de manière exponentielle avec la taille du programme [16].

# Conclusion

J'espère que ces quelques pistes et exemples de transformations d'obfuscation de code pourront être utiles aux concepteurs de programmes dédiés à la protection logicielle.
Les transformations d'obfuscation présentées ci-dessus sont conçues d'abord pour rendre difficile l'analyse statique d'un programme. Nous n'avons pas examiné ici, faute de place, les techniques d'obfuscation spécifiquement destinées à entraver l'analyse dynamique (analyse boîte noire et boîte blanche d'un programme en cours d'exécution au moyen d'un traceur ou d'un débogueur). Le lecteur intéressé pourra se référer à la bibliographie.
Je vous invite à tester les différents types de protection par obfuscation proposés dans cet article et à vous faire votre opinion sur leur résistance face à vos outils de décompilation ou de désassemblage favoris.