

Techniques d'obscurcissement de code pour virus métamorphes

Jean-Marie Borello, Ludovic Mé

15 mars 2007

Résumé

Cet article a pour objet l'étude des virus métamorphes. Plus précisément, nous étudions ici l'utilisation de techniques d'obscurcissement de code avancées applicables aux virus métamorphes afin d'en estimer l'impact sur une détection virale statique. Nous généralisons le résultat de Spinellis [Spi03] concernant la difficulté de détection des virus polymorphes aux cas des virus métamorphes. Ainsi, nous démontrons que la détection fiable d'un virus métamorphe en analyse statique est un problème \mathcal{NP} -difficile dans le cas général. Puis, nous montrons de manière empirique une application de ce résultat à travers la construction d'un obscurcisseur de code qui pourrait être employé par un virus métamorphe.

1 Introduction

Depuis les travaux de Cohen [Coh90] formalisant la notion de virus comme des codes auto-reproducteurs, les techniques virales n'ont cessé d'évoluer afin d'échapper aux outils de détection. Cette évolution a donné naissance à des virus de plus en plus complexes, dont l'une des formes les plus abouties fait appel au métamorphisme.

Contrairement aux autres techniques virales qui ont précédé, telles que le chiffrement ou le polymorphisme, les virus métamorphes se distinguent par leur processus de réplication. Celui-ci modifie l'intégralité du code viral afin de produire des instances syntaxiquement les plus différentes possible des précédentes. Ainsi, chaque binaire résultant de l'exécution d'un virus métamorphe est totalement différent du binaire qui l'a produit. Le métamorphisme désigne la technique d'auto-reproduction propre aux virus métamorphes. Il procède en deux étapes : dans une première étape, le programme viral extrait la sémantique de son propre code afin de se modéliser. Par la suite, cette étape sera désignée sous le terme de modélisation ou de «méta-représentation» et sous le terme archétype le modèle obtenu. Dans une deuxième étape, le programme viral applique des transformations d'obscurcissement à son modèle afin de produire un code le plus différent possible du précédent, tout en conservant sa sémantique.

Les instances d'un virus métamorphe étant toutes de formes différentes, une analyse statique par signature ne peut constituer une méthode de détection fiable. Partant du constat qu'un virus métamorphe est capable de se modéliser afin de se reproduire, plusieurs travaux récents traitant de la détection virale ont émis l'hypothèse qu'un autre programme doit aussi pouvoir modéliser un

tel virus afin de le détecter [LKK04, BMM06, WMCL06]. Notre objectif est d'étudier le bien fondé de cette hypothèse suivant les deux approches possibles pour un outil de détection : analyses statique et dynamique de code. Nous présentons ici la première partie de notre travail relatif à une analyse purement statique. L'analyse statique est ici vue comme un processus visant à extraire la sémantique d'un programme sans recourir à l'exécution de la moindre instruction. L'analyse globale du cycle de reproduction d'un virus métamorphe dépasse largement la cadre d'un seul article. C'est pourquoi, le présent article aborde uniquement la deuxième étape du processus d'auto-reproduction, à savoir les techniques d'obscurcissement de code envisageables pour un code métamorphe, et en présente les conséquences en terme de détection virale.

Le reste de l'article s'organise comme suit : une première partie présentera les techniques d'obscurcissement et de détection des virus métamorphes. Dans une deuxième partie, plus formelle, seront exposées les limitations théoriques auxquelles sont confrontés les outils de détection dans le cas d'une analyse statique de binaire. Enfin, une dernière partie proposera une approche d'obscurcissement qui pourrait être employée par des codes métamorphes, qui tire profit des limitations identifiées pour échapper à une détection par analyse statique.

2 Techniques d'obscurcissement de code et de détection des virus métamorphes

Cette section présente dans un premier temps des techniques d'obscurcissement utilisées par des virus métamorphes. Elle expose ensuite les techniques utiles à la détection de tels virus.

2.1 Techniques d'obscurcissement de code

De manière informelle, l'obscurcissement de code consiste à rendre un programme le plus «in-intelligible» possible. Cette technique s'est principalement développée avec l'utilisation de plus en plus répandue des langages de hauts niveau tel que .NET et JAVA afin de protéger la propriété intellectuelle des algorithmes implémentés. En effet, pour ces langages de haut niveau, le format du byte code produit lors de la compilation conserve toutes les informations permettant de facilement remonter aux sources et donc aux algorithmes implémentés.

L'obscurcissement de code agit à la fois sur les flux de contrôle et de données d'un programme. Une taxonomie des différentes techniques d'obscurcissement est présentée dans [CTL97]. La description qui suit présente celles utilisées dans le cadre du métamorphisme viral :

- obscurcissement du flux de données (substitution d'instructions, permutation d'instructions, insertion de «code mort», et substitution de variables) ;
- obscurcissement du flux de contrôle (altération du flux de contrôle).

Nous présentons ces cinq techniques dans la suite. Le tableau 1 expose des virus utilisant ces transformations :

	EVOL (2000)	ZMIST (2001)	ZPERM (2000)	REGSWAP (2000)	METAPHOR (2001)
Substitution d'instructions					✓
Permutation d'instructions	✓	✓			✓
Insertion de code mort	✓	✓			✓
Substitution de variable	✓	✓		✓	✓
Altération du flux de contrôle		✓	✓		✓

TAB. 1 – *Techniques d'obscurcissement utilisées par des virus métamorphes connus.*

2.1.1 Substitution d'instructions

La substitution d'instructions consiste à changer une séquence d'instructions par une autre séquence d'instructions tout en conservant la même sémantique pour le code.

Le tableau 2 présente un exemple de règles de substitutions (appelées aussi règles de ré-écriture) employées par le virus WIN32.METAPHOR. Dans ce tableau, `Reg`, `Reg2`, `Mem` et `Imm` représentent respectivement un registre quelconque, un autre registre, une référence mémoire et une valeur numérique. La première ligne expose deux possibilités pour affecter la valeur 0 au registre `Reg` : soit `Reg ← (Reg xor Reg)`, soit `Reg ← 0`. La deuxième ligne présente deux possibilités pour affecter une valeur `Imm` au registre `Reg` : soit l'affectation directe `Reg ← Imm`, soit l'utilisation de la pile (l'instruction `PUSH Imm` empile la valeur `Imm`, puis l'instruction `POP Reg` affecte au registre `Reg`, en la dépilant, la valeur `Imm` précédente). Enfin, la dernière ligne montre deux séquences équivalentes de l'opération `OP` sur les registres `Reg` et `Reg2` : soit le calcul directe `Reg ← (Reg OP Reg2)`, soit par utilisation d'une référence mémoire `Mem` intermédiaire : `Mem ← Reg`, puis `Mem ← (Mem OP Reg2)` et enfin `Reg ← Mem`.

Instruction simple	Séquence d'instructions
<code>XOR Reg,Reg</code>	<code>MOV Reg,0</code>
<code>MOV Reg,Imm</code>	<code>PUSH Imm</code> <code>POP Reg</code>
<code>OP Reg,Reg2</code>	<code>MOV Mem,Reg</code> <code>OP Mem,Reg2</code> <code>MOV Reg,Mem</code>

TAB. 2 – *Exemples de substitutions d'instructions extraits du virus WIN32.METAPHOR.*

2.1.2 Permutation d'instructions

La permutation d'instructions consiste à modifier l'ordre d'exécution des instructions d'un programme sans pour autant en altérer la sémantique. Permuter des instructions est possible lorsque deux séquences d'instructions sont indépendantes l'une de l'autre.

Un exemple de permutation d'instructions est présenté dans le tableau 3. Dans ce tableau, toute la sémantique du code exposé est contenue dans la dernière instruction. En effet, cette instruction `REPNZ MOVSB` copie `ecx` caractères à partir de l'adresse contenue dans le registre `esi` vers l'adresse contenue dans

le registre `edi`. Comme l'ordre d'affectation des registres `ecx`, `esi` et `edi` ne modifie pas le résultat de la dernière instruction, il est possible de permuter l'ordre d'affectation de ces registres (lignes 1 à 3).

Instance 1	Instance 2
<code>MOV ecx,104h</code>	<code>MOV edi,dword ptr [ebp+08h]</code>
<code>MOV esi,dword ptr [ebp+0Ch]</code>	<code>MOV ecx,104h</code>
<code>MOV edi,dword ptr [ebp+08h]</code>	<code>MOV esi,dword ptr [ebp+0Ch]</code>
<code>REPZ MOVSB</code>	<code>REPZ MOVSB</code>

TAB. 3 – Exemples de permutation d'instructions.

2.1.3 Insertion de «code mort»

L'insertion de «code mort» consiste à insérer du code inutile dans un programme. Le «code mort» se présente sous la forme d'instructions sémantiquement inutiles. Il peut aussi être constitué d'une séquence d'instructions plus complexes qui n'est jamais atteinte lors d'une exécution du programme.

Règles	Significations
<code>ADD Reg,0</code>	$\text{Reg} \leftarrow \text{Reg} + 0$
<code>MOV Reg,Reg</code>	$\text{Reg} \leftarrow \text{Reg}$
<code>OR Reg,0</code>	$\text{Reg} \leftarrow \text{Reg} \mid 0$
<code>AND Reg,-1</code>	$\text{Reg} \leftarrow \text{Reg} \& -1$

TAB. 4 – Exemples de «code mort» extraits du virus WIN32.METAPHOR.

Le tableau 4 présente des exemples de «codes morts» avec dans la colonne de gauche, des instructions inutiles qui sont sémantiquement équivalentes à l'instruction `NOP` (No Operation) et dans la colonne de droite, la signification précise de chaque instruction. La première ligne correspond à l'ajout de la valeur 0 au registre `Reg`. La deuxième ligne affecte la valeur d'un registre `Reg` à lui même. La troisième ligne effectue un «OU» logique entre un registre `Reg` et la valeur 0. Enfin, la dernière ligne calcule le résultat de l'opérateur logique «ET» entre un registre `Reg` et la valeur -1.

2.1.4 Substitution de variables

Au niveau d'un binaire, la substitution de variables consiste soit à échanger les registres utilisés, soit à modifier la portée des variables en passant de l'utilisation des registres à l'utilisation de variables locales ou globales.

Dans le tableau 5, les instructions sont conservées, seuls les registres ont été échangés. Les registres `edx`, `edi` et `esi` ont été respectivement remplacés par les registres `eax`, `ebx` et `edi`.

Instance 1	Instance 2
<code>POP edx</code>	<code>POP eax</code>
<code>MOV edi,04h</code>	<code>MOV ebx,04h</code>
<code>MOV esi,ebp</code>	<code>MOV edx,ebp</code>
<code>MOV eax,0Ch</code>	<code>MOV edi,0Ch</code>
<code>ADD edx,088h</code>	<code>ADD eax,088h</code>

TAB. 5 – Exemples de substitutions de variables par échange de registres extraits du virus W95.REGSWAP.

2.1.5 Altération du flux de contrôle

L'altération du flux de contrôle consiste à modifier le flux d'exécution d'un programme en introduisant des instructions de branchements conditionnels ou inconditionnels sans pour autant modifier le résultat du programme.

Un exemple de modification du flux de contrôle par insertion de branchements inconditionnels est présenté dans la figure 1, qui expose un code constitué d'une suite d'instructions séquentielles (programme original) et deux autres codes dont le flux d'exécution a été modifié pour ne plus être séquentiel (première et deuxième altérations). Dans tous les cas, la même suite d'instructions est exécutée.

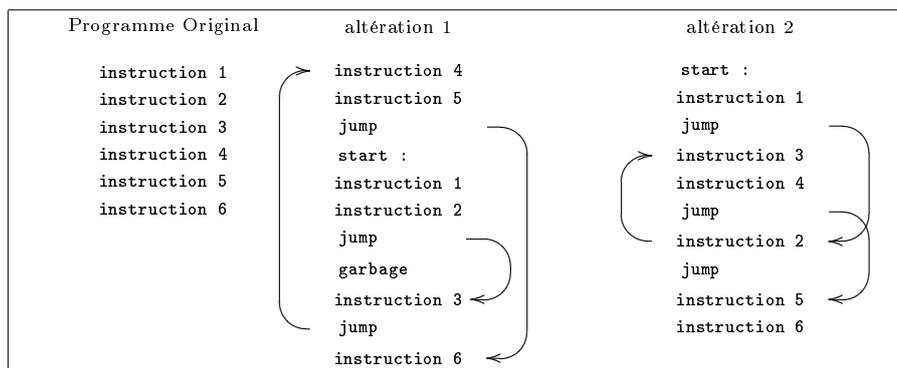


FIG. 1 – Exemples de modifications du flux de contrôle.

2.2 Techniques utilisées pour la détection des virus métamorphes

Nous nous intéressons maintenant aux techniques employées par les logiciels anti-virus afin de détecter les virus métamorphes.

Peu de détails techniques précis sont communiqués sur ces outils de détection viral. Cependant, des travaux portant sur la fiabilité de la détection de virus connus ont révélés les limitations des logiciels anti-virus du marché [CJ03, CJ04]. En effet, les logiciels testés se sont avérés vulnérables aux techniques d'obscurcissement les plus simples, telles que l'insertion de code mort ou la substitution d'instructions. Ces résultats laissent à penser que l'analyse statique par signature, qui consiste à vérifier si un programme comporte ou non un motif binaire donné qui correspond à la signature du virus, reste encore la technique la plus utilisée. Ceci est d'ailleurs confirmé par des travaux récents [Fil06].

Cependant, obscurcir un programme de sorte que sa sémantique ne puisse pas être retrouvée est impossible [BGI⁺01, BF06]. Autrement dit, la sémantique d'un programme peut toujours être retrouvée et ce quelles que soient les techniques d'obscurcissement employées. Pour ce faire, plusieurs prototypes de détection tentent de construire un modèle optimisé du programme. La construction de ce modèle fait appel à des optimisations largement documentées dans le cas de la compilation de code [ARU86]. Comme montré dans [Cif94, DEMS00]

les même techniques peuvent être directement employées pour la modélisation d'un binaire. Un tel processus de modélisation s'effectue en deux temps :

1. Dans un premier temps, un graphe du flux de contrôle du programme est construit. Ce graphe correspond à un modèle des flux d'exécutions possibles du programme.
2. Dans un deuxième temps, le flux de données est analysé et simplifié, ce qui permet ainsi de revenir sur le graphe de flux de contrôle pour le simplifier lui aussi.

Le graphe de flux de contrôle optimisé représente un modèle du programme considéré. Détecter un virus métamorphe revient alors à vérifier si le modèle ainsi obtenu correspond ou non au virus métamorphe en question. La construction d'un tel modèle est largement détaillée dans la thèse de C. Cifuentes [Cif94]. Comme les approches de comparaison de modèles sont spécifiques à chaque prototype [BMM06, CJ03, SXCM04], nous concentrons notre étude sur l'obtention du modèle optimisé support de la détection. Les sous-sections suivantes présentent brièvement les deux étapes de la modélisation.

2.2.1 Présentation du graphe de flux de contrôle

Afin de définir brièvement un graphe de flux de contrôle (GFC), dont une définition plus précise est fournie dans [ARU86], deux types d'instructions sont considérées suivant le fait qu'elles modifient ou non le flux d'exécution du programme. Ainsi, nous désignerons par instruction séquentielle une instruction ne modifiant pas le flux de contrôle d'un programme et par instruction de transfert une instruction modifiant le flux de contrôle d'un programme.

Définition : un graphe de flux de contrôle est un graphe orienté (N, E) avec N l'ensemble des sommets du graphe et E l'ensemble des arêtes. Chaque sommet représente un «basic block». Un «basic block» est un ensemble ordonné d'instructions séquentielles se terminant :

- soit par une instruction de transfert ;
- soit par une instruction séquentielle immédiatement suivie d'une instruction séquentielle appartenant à un autre «basic block».

Chaque arête e issue d'un «basic block» correspond à une sortie conditionnelle ou incondionnelle de ce bloc.

2.2.2 Optimisations en analyse du flux de données

Afin de manipuler le code plus simplement, chaque instruction assembleur est représentée sous la forme d'une instruction abstraite exprimant la sémantique de l'instruction. Cette représentation abstraite permet aussi de s'affranchir de l'architecture matérielle. Quatre types d'instructions abstraites sont généralement utilisées afin de retranscrire la sémantique du jeu d'instructions d'un CPU [Cif94] : l'affectation (notée $=$), l'appel et le retour de procédure (CALL et RET), le branchement conditionnel (JCOND), et enfin le branchement incondionnel (GOTO).

Le tableau 6 illustre les différentes étapes d'optimisations lors de l'analyse du flux de données sur un exemple extrait du virus WIN32.METAPHOR. La première colonne présente un «basic block» d'origine. La deuxième colonne montre

une représentation abstraite de chaque instruction du même bloc afin d'en exprimer la sémantique. Enfin, la troisième colonne montre le résultat obtenu après les différentes optimisations. Sans rentrer dans les détails de chaque instruction, ce bloc correspond juste à un appel à l'API WIN32 : `ExitProcess(0)`. La valeur 0 est empilée (instructions 2, 3 et 4), puis l'adresse de la fonction `ExitProcess` est propagée jusqu'à l'appel de la ligne 9.

Code d'origine	Méta-représentation	Après optimisations
<code>MOV esi,esi</code>	<code>esi=esi</code>	
<code>MOV dword_40BB49, 0</code>	<code>dword_40BB49=0</code>	<code>dword_40BB49=0</code>
<code>MOV esi, dword_40BB49</code>	<code>esi=dword_40BB49</code>	<code>esi=0</code>
<code>PUSH esi</code>	<code>esp=esp-4</code> <code>[esp]=esi</code>	<code>[esp-4]=0</code>
<code>MOV dword_40BCBB,offset ExitProcess</code>	<code>dword_40BCBB=&ExitProcess</code>	<code>dword_40BCBB=&ExitProcess</code>
<code>MOV ebx,dword_40BCBB</code>	<code>ebx=dword_40BCBB</code>	<code>ebx=&ExitProcess</code>
<code>PUSH dword ptr [ebx+0]</code>	<code>esp=esp-4</code> <code>[esp]=[ebx]</code>	<code>[esp-8]=ExitProcess</code>
<code>POP dword_40B92A</code>	<code>dword_40B92A=[esp]</code>	<code>dword_40B92A=ExitProcess</code>
<code>CALL dword_40B92A</code>	<code>esp=esp+4</code> <code>call dword_40B92A</code>	<code>call ExitProcess</code>

TAB. 6 – Exemple d'optimisations du flux de données.

L'analyse du flux de données consiste en plusieurs optimisations :

1. **Propagation des données** : lors de cette étape, les valeurs ou expressions algébriques affectées aux variables du programme (registres, variables locales et globales) sont propagées. Toutes les occurrences d'une variable sont remplacées par sa valeur ou son expression. Par exemple, dans le tableau 6, la valeur de la variable globale `dword_40BB49` en ligne 2 est affectée au registre `esi` la ligne suivante.
2. **Élimination du code mort** : les instructions dont le résultat n'est jamais utilisé sont supprimées. À la première ligne du tableau précédent la définition du registre `esi` n'est jamais utilisé avant d'être redéfini deux lignes plus loin. La définition de la première ligne est donc inutile.
3. **Simplifications algébriques** : la majorité des instructions d'un CPU correspond à des opérateurs arithmétiques et logiques. La simplification algébrique vise alors à simplifier au maximum une expression mathématique obtenue par propagation des données.
4. **Compression du graphe de flux de contrôle** : la compression ou optimisation du graphe de flux de contrôle a pour objectif d'inverser les modifications de ce graphe afin d'en obtenir une version la plus réduite et la plus simple possible. Elle permet entre autres de distinguer des structures de contrôle de haut niveau telles que les boucles (`for`, `do`, `while`).

La figure 2, représente un modèle optimisé de l'amorce du virus METAPHOR dans le cas où le corps du virus est chiffré (15 fois sur 16). Les trois premiers blocs servent à allouer un espace mémoire d'un peu plus de 3,4Mo où sera déchiffré le corps du virus. La boucle représentée par les blocs 6 à 8 correspond à la routine de déchiffrement. Le bloc 9 exécute le corps du virus puis le bloc 10 termine le programme. Ce modèle peut constituer un motif de haut niveau permettant d'identifier l'amorce du virus en question.

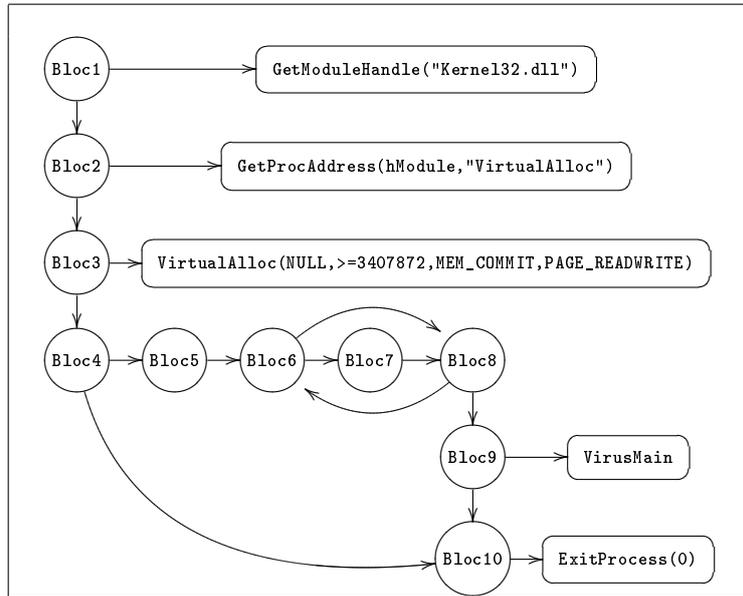


FIG. 2 – Exemple d’archétype pour l’amorce du virus WIN32.METAPHOR.

3 Limites d’une détection fiable des virus métamorphes par analyse statique

Cette section a pour but de déterminer la difficulté d’une détection fiable d’un virus métamorphe d’un point de vue théorique. Pour cela, plusieurs notations et définitions sont précisées.

Notations : Soient deux programmes (algorithmes) A et B dont les ensembles d’entrées possibles respectifs sont \mathcal{D}_A et \mathcal{D}_B . Le fait que le calcul de A pour l’entrée x ne termine jamais est représenté par $A(x) = \perp$. A et B sont dit fonctionnellement équivalents, ce qui est noté $A \equiv B$, si et seulement si ils produisent les mêmes résultats pour les mêmes entrées, soit :

$$\begin{cases} \mathcal{D}_A = \mathcal{D}_B \\ \forall x \in \mathcal{D}_A, A(x) = B(x) \end{cases}$$

Définition 1 : Le programme D_V détecte de manière fiable le virus métamorphe V (avec V tel que $\forall x, V(x) \neq \perp$) si et seulement si pour tout programme P ,

$$\begin{cases} D_V(P) \text{ retourne } \langle \text{true} \rangle \text{ si } P \equiv V \\ D_V(P) \text{ retourne } \langle \text{false} \rangle \text{ sinon.} \end{cases}$$

Proposition 1 : Il n’existe pas d’algorithme capable de déterminer si, pour deux programmes donnés P et P' non nuls, $P' \equiv P$.

Preuve :

Supposons par l’absurde qu’un tel algorithme D existe. Par définition, D est tel

que pour tous programmes A et B ,

$$\begin{cases} D(A, B) \text{ retourne } \langle \text{true} \rangle \text{ si } A \equiv B \\ D(A, B) \text{ retourne } \langle \text{false} \rangle \text{ sinon} \end{cases}$$

Le programme C est défini pour tous programmes A et B , et pour toute entrée $x \in \mathcal{D}_B$ par,

```
1 C(A,B,x){
2   if (D(A,B)==true) then { if (B(x)==⊥) then ω ; else ⊥ ;}
3   else B(x) ;}
```

Le symbol ω désigne une valeur de retour quelconque autre que \perp . ω existe nécessairement car tout programme admet au moins \emptyset et \perp comme valeurs de retour possibles. Pour tout programme P et pour toute entrée $x \in \mathcal{D}_P$. Deux cas sont à envisager pour le calcul de $C(C, P, x)$:

1. Premier cas : $D(C, P)$ retourne $\langle \text{true} \rangle$, ce qui, par définition de D signifie que $C \equiv P$. Or, par construction de C , si $D(C, P)$ retourne $\langle \text{true} \rangle$ (ligne 2) alors le programme C ne retourne pas $P(x)$. Cela signifie que dans ce cas, $C \not\equiv P$, ce qui est en contradiction avec l'hypothèse de départ.
2. Deuxième cas : $D(C, P)$ retourne $\langle \text{false} \rangle$, ce qui, par définition de D signifie que $C \not\equiv P$. Or, d'après la construction de C , si $D(C, P)$ retourne $\langle \text{false} \rangle$ (ligne 3) le programme retourne alors $P(x)$. Il en résulte que $C(C, P, x) = P(x)$ soit $C \equiv P$, ce qui est en contradiction avec l'hypothèse de départ.

Corollaire de la proposition 1 : La détection fiable d'un virus métamorphe telle que présentée par la définition 1 est un problème indécidable.

La proposition 1 n'est qu'un cas particulier du théorème de Rice [Ric53]. En ce qui nous concerne, ce théorème a pour conséquence directe l'indécidabilité de nombreux problèmes d'analyse statique [Lan92]. Pour s'affranchir de cette limitation, nous supposons que tous les chemins d'un programme P donné sont potentiellement exécutables. Même si elle n'est pas toujours valide, cette hypothèse couramment faite en analyse statique [ARU86], permet de simplifier notre problème.

Proposition 2 : Dans l'hypothèse où tous les chemins d'un programme sont potentiellement exécutables, pour tous programmes P et P' , tels que $\forall x \in \mathcal{D}_P, P(x) \neq \perp$, savoir si il existe un chemin de P' pour lequel $P' \equiv P$ est un problème \mathcal{NP} -difficile.

La preuve suivante s'inspire directement des nombreux problèmes démontrés \mathcal{NP} -difficiles par Landi [Lan92]. Elle s'apparente plus précisément aux démonstrations des travaux suivants [OSSM02, WHKD00] et sert de base à la construction d'un obscurcisseur de code présenté dans la section suivante.

Preuve :

Étant donné une instance S du problème 3-SAT, problème démontré \mathcal{NP} -Complet, et un programme P , nous démontrons qu'il est possible de construire en temps polynomial par rapport à la taille de S un programme P' tel que, S est satisfiable si et seulement si il existe un chemin de P' tel que $P' \equiv P$.

Une instance S du problème 3-SAT est donnée sous la forme :

$$S = \bigwedge_{i=1}^n (l_{i,1} \vee l_{i,2} \vee l_{i,3}) \text{ avec } \begin{cases} \{v_1, v_2, \dots, v_m\} \text{ un ensemble de variables booléennes} \\ \forall (i, j) \in [1, n] \times [1, 3], \exists k \in [1, m], l_{i,j} = v_k \text{ ou } l_{i,j} = \overline{v_k} \end{cases}$$

Nous construisons la famille des u_i , notée $(u_i)_{i \in [1, k]}$ telle qu'elle soit une partition d'éléments consécutifs de l'ensemble $[1, n]$ où $\forall i \in [1, k]$, $\text{card}(u_i) > 0$. Une représentation possible de la famille (u_i) est de la forme :

$$\underbrace{1, 2, 3, 4}_{u_1}, \underbrace{5, 6, \dots, 9, 10}_{u_2}, \dots, \underbrace{n-2, n-1, n}_{u_k}$$

S s'écrit alors :

$$S = \bigwedge_{i=1}^k S_i \text{ avec } S_i = \bigwedge_{j=\min(u_i)}^{\max(u_i)} (l_{j,1} \vee l_{j,2} \vee l_{j,3})$$

Le programme P constitué d'une suite d'instructions quelconques $I_1 I_2 \dots I_n$ est représenté sous la forme d'une famille $(P_i)_{i \in [1, k]}$ du type :

$$\begin{cases} P_1 = I_1 \dots I_\alpha \\ P_2 = I_{\alpha+1} \dots I_\beta \\ \vdots \\ P_k = I_{\gamma+1} \dots I_n \end{cases}, \text{ avec } (1 < \alpha < \beta < \gamma < n)$$

Toutes les instructions sont supposées **atteignables**, c'est-à-dire que le programme P ne contient pas de code mort. De plus, le découpage est effectué de telle sorte que pour chaque bloc P_i , l'exécution de la dernière instruction de ce bloc conduit à la première instruction du bloc P_{i+1} (le bloc P_i ne se termine pas par un saut ou un retour de procédure).

Nous construisons maintenant en temps polynomial par rapport à la taille de S le programme P' constitué de la famille des $(P'_i)_{i \in [0, k+1]}$ comme le montre le schéma de la figure 3.

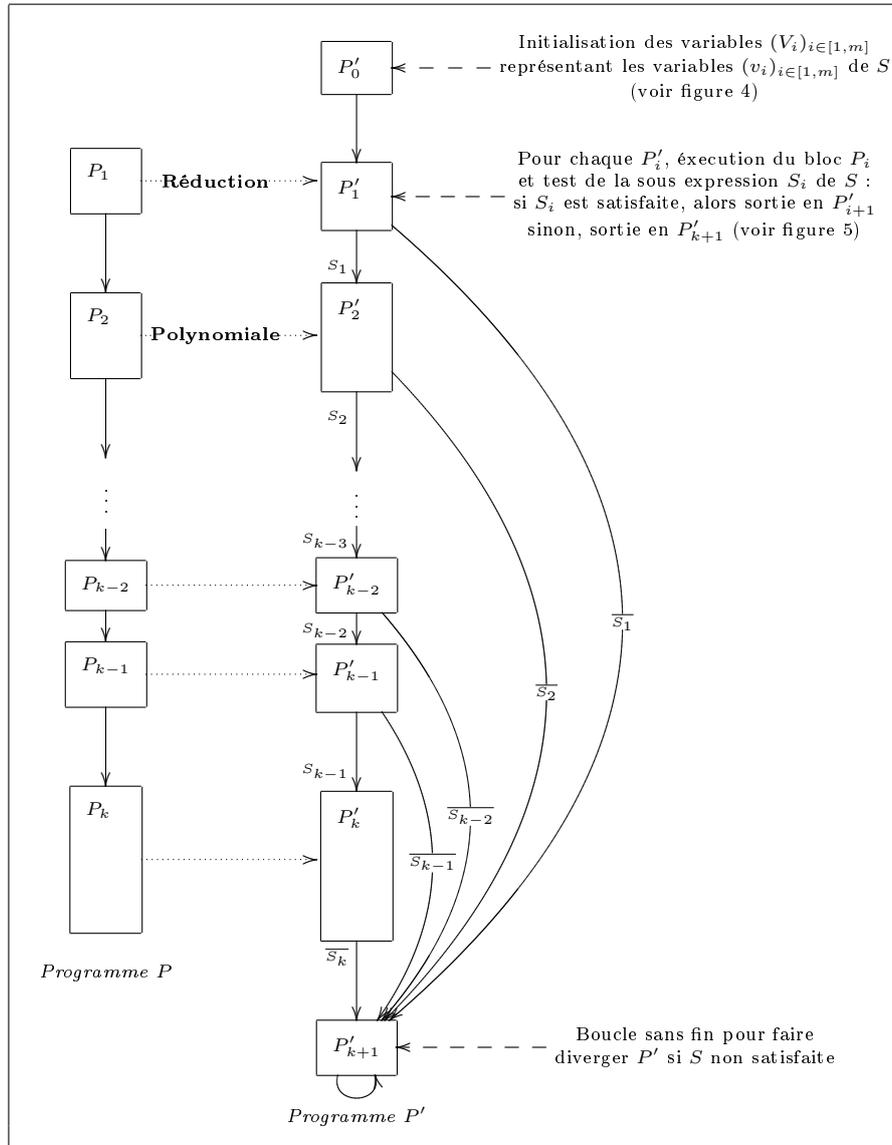


FIG. 3 – Schéma de construction du programme obscurci P' (à droite) à partir du programme P (à gauche).

Comme tous les chemins sont supposés exécutables, nous ne tenons pas compte des conditions au niveau des branchements : `if (-) {code1} else {code2}`

La figure 4 présente le pseudo-code contenu à l'intérieur du bloc P'_0 . Ce code sert à l'initialisation du programme P' . Chaque variable V_i et sa négation \bar{V}_i sont déclarées en première ligne comme des pointeurs de pointeurs de fonctions. Chacune de ces variables peut pointer sur `true` ou `false` déclarées en ligne 2 comme des pointeurs de fonctions. Chaque V_i et \bar{V}_i représentent respectivement une variable booléenne de S et sa négation. Ainsi, une affectation de v_i à «vrai»

pour S correspond à l'affectation $*V_i=\text{true}$ pour P' . Un chemin d'exécution de P'_0 (lignes 4 à 6) initialise donc de toutes les variables V_i et \overline{V}_i , ce qui correspond à fixer les valeurs des variables booléennes v_i pour notre équation S . Deux cas vont alors influencer le comportement du reste du programme P' : soit l'affectation des v_i satisfait S , soit elle ne satisfait pas S .

```

1 void (**V1)(),(**V1)(),...,(**Vm)(),(**Vm)();
2 void (*true)(),(*false)();

3 if (-) {V1=&true;V1=&false;} else {V1=&false;V1=&true;}
4 if (-) {V2=&true;V2=&false;} else {V2=&false;V2=&true;}
...
5 if (-) {Vm=&true;Vm=&false;} else {Vm=&false;Vm=&true;}
6 goto P1';

```

FIG. 4 – Pseudo-code du bloc P'_0 .

Le bloc P'_{k+1} est une boucle sans fin ($P'_{k+1} : \text{goto } P'_{k+1}$). Ce bloc fait diverger le programme P' du programme P dans le cas où l'affectation des $(v_i)_{i \in [1,m]}$ ne satisfait pas S (voir figure 3).

La figure 5 définit, pour tout i variant de 1 à $k-1$, le bloc P'_i à partir du bloc P_i . Le pointeur `false` est initialisé avec l'adresse du bloc P'_{i+1} . La deuxième ligne représente l'insertion du code contenu à l'intérieur du bloc P_i de sorte qu'exécuter le bloc P'_i conduit à exécuter P_i . La notation $l_{i,j}$ dans les expressions de la forme $*l_{i,j}=\&P'_{k+1}$ (lignes 3 à 4) est une notation abstraite qui représente une variable V_k ou sa négation \overline{V}_k (comme dans la formulation de S). Les lignes numérotées de 3 à 4 permettent de modifier la destination du pointeur `false` en fonction des valeurs des V_i et \overline{V}_i initialisées en P'_0 comme nous allons le voir un peu plus loin. Finalement, la fonction `false` est appelée en ligne 5.

```

1 false=&P_{i+1};
2 .../* insertion du code de P_i */
3 if (-) {*l_{min(u_i),1}=&P'_{k+1};}
   else if (-) {*l_{min(u_i),2}=&P'_{k+1};}
   else {*l_{min(u_i),3}=&P'_{k+1};}
...
4 if (-) {*l_{max(u_i),1}=&P'_{k+1};}
   else if (-) {*l_{max(u_i),2}=&P'_{k+1};}
   else {*l_{max(u_i),3}=&P'_{k+1};}
5 false();
6 return;

```

FIG. 5 – Pseudo-code des blocs du programme P' .

Le bloc P'_k diffère légèrement des autres blocs P'_i , $i \in [1, k-1]$, pour ne pas atteindre le bloc P'_{i+1} (qui dans ce cas serait la boucle sans fin en P'_{k+1}) mais terminer le programme.

Pour la suite, sauf précision, nous nous référerons toujours à la figure 5. De plus, nous désignons par *sortie* du bloc P'_i une des deux destinations possibles :

P'_{i+1} ou P'_{k+1} . Nous montrons maintenant l'équivalence entre notre problème exprimé dans la proposition 2 et le problème 3-SAT.

1. Premier cas : **S est satisfiable**, ce qui signifie que $\forall i \in [1, k]$, S_i est satisfiable. S_i satisfiable correspond au niveau du pseudo-code à $\forall j \in [\min(u_i), \max(u_i)]$, $l_{j,1}$ ou $l_{j,2}$ ou $l_{j,3}$ pointe sur la variable **true**. Si nous représentons la ligne correspondante de P'_i sous la forme d'un GFC, nous obtenons la figure 6.

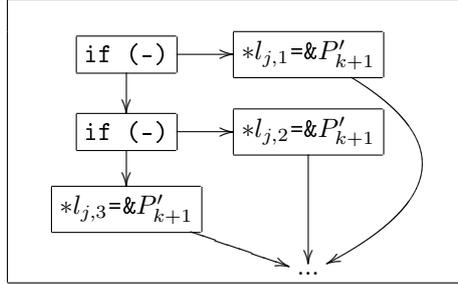


FIG. 6 – GFC d'une des lignes de 3 à 4 de P'_i

Comme au moins un littéral de la forme $l_{j,t}$, $t \in [1, 3]$ pointe sur la variable **true** alors il existe au moins un chemin dans ce graphe pour lequel la destination du pointeur **true** est affectée à l'adresse P'_{k+1} . Cette propriété étant vraie pour toutes les lignes de 3 à 4, il existe donc un chemin de P'_i pour lequel seule la variable **true** a été réaffectée à l'adresse du bloc P'_{k+1} à la ligne 5 et donc pour lequel la variable **false** n'a pas été redéfinie. Dans ce cas, la *sortie* du bloc P'_i est le bloc P'_{i+1} . Cette propriété étant vérifiée pour tous les P'_i , il en résulte que $P' \equiv P$ par construction de P' . Nous obtenons donc que, **si S est satisfiable alors il existe un chemin de P' pour lequel $P' \equiv P$.**

2. Deuxième cas : **Il existe un chemin de P' tel que $P' \equiv P$** , ce qui signifie que $\forall x \in \mathcal{D}_P$, $P'(x) \neq \perp$ d'après les hypothèses de la proposition 2. Par construction de P' , pour tout i , la *sortie* du bloc P'_i n'est alors jamais P'_{k+1} (sinon nous aurions $P'(x) = \perp$). Autrement dit pour chaque P'_i , **false** pointe sur le bloc P'_{i+1} en ligne 5. Or **false** pointe sur le bloc P'_{i+1} uniquement si, pour le chemin de P'_i considéré, chaque littéral de la forme $l_{j,t}$ pointe sur la variable **true**. En effet, si pour ce chemin un seul littéral $l_{j,t}$ pointait sur **false** alors **false** se verrait affectée l'adresse du bloc P'_{k+1} dans une des lignes de 3 à 4 et pointerait donc sur la boucle sans fin en ligne 5. Ce résultat pour le bloc P'_i implique que S_i est satisfiable. Comme cette relation doit être vérifiée pour tout i de 1 à k , S est satisfiable, d'où le résultat attendu : **s'il existe un chemin de P' tel que $P' \equiv P$ alors S est satisfiable.**

Nous obtenons donc l'équivalence entre notre problème et le problème 3-SAT. Comme nous ne faisons pas d'hypothèse sur la classe de complexité d'appartenance de notre problème, il en résulte que ce dernier est \mathcal{NP} -difficile.

Corollaire de la proposition 2 : La détection fiable d'un code métamorphe de taille finie telle que présentée en définition 1 est un problème \mathcal{NP} -difficile.

Ce résultat apporte un complément par rapport à celui de Spinellis [Spi03] concernant la difficulté de détection des codes polymorphes de tailles bornées. Il constitue une généralisation de ce résultat aux codes métamorphes à la nuance près que nous ne démontrons que la complexité minimale de détection. La borne supérieure reste un problème ouvert qui n'est pas traité ici. Nous dégageons deux conséquences immédiates à ce corollaire :

1. Seule une détection statique approximative est, de manière théorique, calculatoirement acceptable. Ce résultat conforte donc l'approche par optimisation envisagée par les derniers prototypes de détection évoqués en première partie. [BMM06, CJ03, SXCM04]
2. La mise en place de techniques d'obscurcissement plus abouties que celles présentées en première partie, en agissant notamment sur le flux de contrôle d'un programme, augmente considérablement la difficulté de la détection d'un code métamorphe. C'est ce point que nous développons en section 4.

4 Obscurcisseur de code pour des virus métamorphes difficilement détectables par analyse statique

Cette section présente une approche possible d'obscurcisseur qui pourrait être employé dans le cadre d'un virus métamorphe. Cet obscurcisseur s'appuie sur la démonstration de la proposition 2.

4.1 Approche d'obscurcissement de flux de contrôle

L'approche consiste à modifier le flux de contrôle d'un programme P afin de produire un programme obscurci P' . Elle s'apparente à l'approche détaillée dans [CGJZ01] par automate à états finis et celle exposée dans [OSSM02] utilisant des pointeurs de fonctions. Deux aspects sont toutefois spécifiques à notre approche :

1. L'obscurcisseur proposé travaille au niveau d'un source assembleur. Ce premier aspect permet d'appliquer, indépendamment des modifications du flux de contrôle, toutes les techniques d'obscurcissement du flux de données présentées en première section. Ces transformations seront conservées lors de l'assemblage du source. Or, ce n'est pas forcément le cas pour un source écrit dans un langage de plus haut niveau où la phase de compilation applique des optimisations qui peuvent inverser les transformations d'obscurcissement réalisées préalablement et donc nuire à la qualité de l'obscurcissement.
2. L'obscurcisseur proposé est fortement contraint car il devrait permettre au virus de se modéliser tout en assurant que l'analyse statique de son code soit la plus difficile possible. Cet aspect est abordé à la fin de la sous-section ?? mais n'est pas détaillé dans cet article.

L'approche globale de construction du programme obscurci P' peut se résumer de la façon suivante :

- Comme dans la démonstration de la proposition 2, nous découpons le programme P , constitué d'une suite de n instructions $(I_j)_{j \in [1, n]}$, en k ensembles de tailles aléatoires non nulles d'instructions consécutives pour former les blocs $(P_i)_{i \in [1, k]}$. Ce découpage est fait de telle sorte que chaque bloc P_i ne se termine ni par un saut incondtionnel (GOTO) ni par un retour de procédure (RET). Cette restriction est nécessaire car autrement, la condition de passage de P_i à P_{i+1} serait inutile.

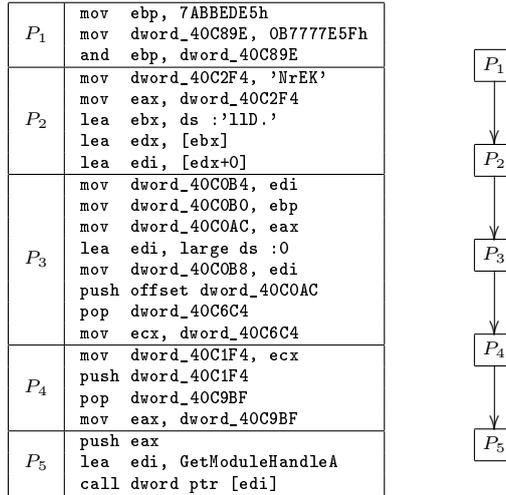


FIG. 7 – Exemple de découpage en (P_i) pour une amorce possible du virus WIN32.METAPHOR.

La figure 7, présente un exemple de découpage de code assembleur correspondant au premier «basic block» du GFC du virus WIN32.METAPHOR. L'explication détaillée du code n'est pas nécessaire pour la compréhension de la suite. L'enchaînement des blocs est ici linéaire (P_1, P_2, \dots, P_5) .

- Nous construisons $(G_i)_{i \in [1, p]}$ une famille de séquences d'instructions cohérentes de tailles aléatoires non nulles. Par cohérente, nous entendons que la séquence d'instructions provient d'un programme réel et non pas d'instructions aléatoires. Cette famille représente des blocs de code mort. Le but de ces blocs est uniquement d'augmenter la complexité de l'analyse statique du programme P' . Pour la suite de l'explication, nous utiliserons des blocs de code mort illustrés par la figure 8 dont la compréhension n'est pas nécessaire, à la lecture de la suite.



FIG. 8 – Exemple de blocs de code morts (G_i) .

- Nous construisons maintenant le programme obscurci P' constitué des blocs $(P'_i)_{i \in [0, k+p]}$ de la façon suivante :
 - $P'(0)$ constitue le bloc d'initialisation du programme obscurci. Il initialise une information K qui représente l'unique aiguillage en sortie

de chaque bloc P'_i de sorte que $P' \equiv P$. Cette information K constitue donc l'élément clé de l'obscurcisseur qui sera désignée par la suite sous le terme de paramètre d'obscurcissement.

- (b) $\forall i \in [1, k+p]$, deux choix sont possibles pour la construction du bloc P'_i :
- soit P'_i est un bloc légitime, c'est-à-dire que $\exists! s \in [1, k]$ tel que P'_i contient le code de P_s . Dans ce cas, nous définissons une sortie légitime vers le bloc P'_{i+1} et une autre sortie choisie aléatoirement parmi les autres blocs.
 - soit P'_i est un bloc illégitime (code mort), c'est-à-dire que $\exists! t \in [1, p]$ tel que P'_i contient le code de G_t . Dans ce cas, nous définissons deux sorties aléatoires.

Dans tous les cas, les sorties sont conditionnées par K de telles sortes que l'exécution du programme P' corresponde à celle de P .

4.1.1 Initialisation du paramètre d'obscurcissement K

Comme K est l'élément essentiel à une reconstruction statique du programme P à partir de P' , il est nécessaire de montrer comment compliquer l'obtention de cette information. Nous distinguons ici deux approches :

1. la première s'appuie sur des difficultés d'ordre mathématiques présentées dans [CTL97].
 - À ce titre, l'emploi de conjectures mathématiques peut grandement compliquer le bon déroulement d'une analyse statique. Considérons par exemple la suite de Syracuse $(u_n)_{n \in \mathbb{N}}$ définie par :

$$\begin{cases} u_0 \in \mathbb{N} \\ \forall n \in \mathbb{N}^*, u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases} \end{cases}$$

La conjecture de Syracuse affirme que pour tout u_0 initial, la suite $(u_n)_{n \in \mathbb{N}}$ finit par boucler sur les valeurs 4, 2, 1. K pourrait alors être initialisé à partir du plus petit entier i vérifiant $u_i = 1$;

- Il est aussi possible d'utiliser des expressions algébriques difficilement vérifiables [OSSM02]. Par exemple, l'initialisation d'un bit k_i de K pourrait provenir du code suivant :

`if (a*(a+1)%2==0) {k_i=1 ;} else {k_i=0 ;}` où a désigne un entier quelconque.

Dans ce cas, si $a(a+1)$ est pair le bit en question vaut 1, sinon il vaut 0. De manière générale, la détermination d'une telle condition nécessite l'emploi d'algorithmes complexes tels que ceux utilisés par logiciels de calcul formel.

2. des difficultés induites par l'aspect statique de l'analyse qui ne peut pas prendre en compte le contexte d'exécution d'un programme. D'un point de vue statique, le résultat de tout appel extérieur au programme (autres programmes, API ...) demeure une inconnue qui ne peut être résolue, de manière exacte, qu'au moment de l'exécution.

Il en est de même pour l'obtention statique des résultats de calculs itératifs ou récursifs. À titre d'exemple, l'utilisation de fonctions de hachage

ou de chiffrement peut sensiblement augmenter la difficulté d’une analyse statique. Supposons par exemple que $K = (x_m, x_{m-1}, \dots, x_2, x_1)$ soit défini par $\forall i \in [1, m], \exists j \in [1, m], x_i = \mathcal{H}(P_j)$ où \mathcal{H} représente une fonction de hachage quelconque retournant 1 bit. La détermination d’un x_i nécessite alors l’exécution de la fonction \mathcal{H} difficilement calculable de manière statique.

4.1.2 Exemple illustrant la difficulté d’analyse statique d’un programme obscurci

La mesure de l’efficacité d’un obscurcisseur de code s’évalue suivant trois critères [CTL97] présentés ici de manière informelle :

- le **potentiel**, qui mesure la difficulté de compréhension du code obscurci pour un humain ;
- la **résilience**, qui mesure la difficulté de l’optimisation de code par un outils automatique ;
- le **coût**, qui mesure la pénalité du programme obscurci en terme de temps et d’espace par rapport au programme d’origine.

Dans le cadre d’un virus métamorphe, la métrique la plus pertinente est la résilience qui mesure alors la difficulté, pour un programme donné, de modélisation d’un virus afin de le détecter.

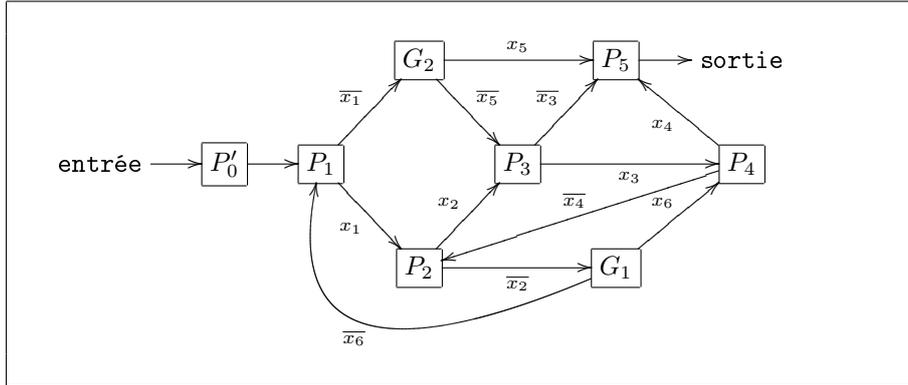


FIG. 9 – Exemple d’obscurcissement possible du programme P .

À titre d’exemple, la figure 9 montre un programme obscurci P' à partir du programme P . Le bloc P'_0 initialise les variables booléennes $(x_i)_{i \in [1,6]}$ conditionnant les sorties de chaque bloc. Par exemple, pour le bloc P_1 , si x_1 vaut 1 alors le prochain bloc exécuté est P_2 sinon le bloc suivant est G_2 . Pour que l’exécution du programme P' corresponde effectivement au programme P , il faut conserver l’enchaînement des blocs comme dans la figure 7, ce qui correspond dans ce cas, à $\forall i \in [1, 4], x_i = 1$. **Sans la connaissance du contexte K , le nombre de chemins d’exécutions possibles est de 2^{k+p-1} .**

Les variables x_5 et x_6 conditionnent uniquement les sorties des blocs de codes morts G_1 et G_2 , blocs qui ne sont jamais exécutés dans le cas où $P' \equiv P$. Le tableau suivant ne présente donc que les 16 premières valeurs de K possibles en supposant que x_5 et x_6 sont nuls. Au lieu de développer les optimisations d’analyse statique pour les 16 cas possibles d’affectation des $(x_i)_{i \in [1,4]}$, nous considé-

rons le résultat de l'exécution du programme obscurci P' comme le montre le tableau 7. Cette considération est légitime dans ce cas précis puisque détecter ce «basic block» (figure 7) revient à vérifier qu'il aboutit à l'exécution de l'appel `GetModuleHandle("Kernel132.dll")` comme déjà vue en figure 2.

K	Affectations des (x_i)	Chemins d'exécution	Résultats de P'
0	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, \overline{x_2}, \overline{x_1}$	P1,G2,P3,P5	Erreur fatale
1	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, \overline{x_2}, x_1$	(P1,P2,G1)*	Boucle sans fin
2	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, x_2, \overline{x_1}$	P1,G2,P3,P5	Erreur fatale
3	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, x_2, x_1$	P1,P2,P3,P5	Erreur fatale
4	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, \overline{x_2}, \overline{x_1}$	(P1,G2,P3,P4,P2,G1)*	Boucle sans fin
5	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, \overline{x_2}, x_1$	(P1,P2,G1)*	Boucle sans fin
6	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, x_2, \overline{x_1}$	P1,G2,(P3,P4,P2)*	Boucle sans fin
7	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, x_2, x_1$	P1,(P2,P3,P4)*	Boucle sans fin
8	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, \overline{x_2}, \overline{x_1}$	P1,G2,P3,P5	Erreur fatale
9	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, \overline{x_2}, x_1$	(P1,P2,G1)*	Boucle sans fin
10	$\overline{x_6}, \overline{x_5}, x_4, x_3, x_2, \overline{x_1}$	P1,G2,P3,P5	Erreur fatale
11	$\overline{x_6}, \overline{x_5}, x_4, x_3, x_2, x_1$	P1,P2,P3,P5	Erreur fatale
12	$\overline{x_6}, \overline{x_5}, x_4, x_3, \overline{x_2}, \overline{x_1}$	P1,G2,P3,P4,P5	Incorrect
13	$\overline{x_6}, \overline{x_5}, x_4, x_3, \overline{x_2}, x_1$	(P1,P2,G1)*	Boucle sans fin
14	$\overline{x_6}, \overline{x_5}, x_4, x_3, x_2, \overline{x_1}$	P1,G2,P3,P4,P5	Incorrect
15	$\overline{x_6}, \overline{x_5}, x_4, x_3, x_2, x_1$	P1,P2,P3,P4,P5	Correct

TAB. 7 – Résultat de l'exécution du programme P' pour différentes valeurs de K .

Le tableau 7 expose quatre types de résultats regroupés comme suit :

- 7 cas ($K = 1, 4, 5, 6, 7, 9, 13$) correspondent à une boucle sans fin pour laquelle l'instruction `call` n'est jamais atteinte. Ce qui implique une non détection du bloc comme appartenant au virus.
- 6 cas ($K = 0, 2, 3, 8, 10, 11$) entraînent une erreur de l'application lors de l'appel `WIN32 GetModuleHandle`, erreur due à un paramètre incorrect qui ne constitue pas un pointeur valide.
- 2 cas ($K = 12, 14$) renvoient 0 après l'appel. Ici, le pointeur passé en paramètre est bien valide mais ne pointe pas sur la chaîne recherchée `"Kernel132.dll"`.
- un seul cas ($K = 15$) correspond à l'appel recherché conformément au choix de la valeur de K ayant servi à la construction de P' .

Cet exemple montre l'impact concret du résultat théorique sur une détection statique dans le cas où le paramètre d'obscurcissement K n'est pas connu.

5 Conclusion

Dans cet article nous avons démontré qu'une détection fiable par analyse statique d'un virus métamorphe est un problème \mathcal{NP} -difficile dans le cas général. En application à cette démonstration nous présentons une approche d'obscurcissement de code à résilience prouvée face à un outils d'analyse statique. Cette approche par altération du flux de contrôle d'un programme a été conçue pour pouvoir être appliquée aux virus métamorphes afin d'empêcher une détection

virale statique efficace. En effet, cette approche s'appuie sur le constat qu'un virus métamorphe peut, lors de son exécution, résoudre des problèmes reconnus comme difficiles en analyse statique. Par contre, l'obscurcissement exposé ici n'est pas directement applicable aux virus actuels. En effet, le processus de réplication métamorphe procède à la phase d'obscurcissement de code après celle de modélisation. Pour employer directement notre approche sur un virus métamorphe, il faudrait aussi que ce virus soit capable d'inverser cette transformation lors de son exécution afin de se modéliser. La modélisation ainsi que les modifications qu'elle sous-entend sur l'approche ici présentée ne sont pas abordées dans cet article. Nos travaux futurs s'orientent dans cette direction afin d'étudier le processus de reproduction dans sa globalité : modélisation du virus, puis obscurcissement de son code.

Références

- [ARU86] A.V. Aho, R.Sethi, and J.D. Ullman. *Compilers : Principles, Techniques and, Tools*. Addison-Wesley, 1986.
- [BF06] P. Beaucamps and E. Filiol. On the possibility of practically obfuscating programs - towards a unified perspective of code protection. *WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds, Journal in Computer Virology, 2 (4)*, 2006.
- [BGI⁺01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *in Crypto '01, LNCS No.2139* :pages 1–18, 2001.
- [BMM06] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of the International Symposium of Secure Software Engineering*, Arlington, VA, U.S.A., 2006. IEEE Computer Society.
- [CGJZ01] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *ISC '01 : Proceedings of the 4th International Conference on Information Security*, pages 144–155, London, UK, 2001. Springer-Verlag.
- [Cif94] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, School of Computing Science, 1994.
- [CJ03] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.
- [CJ04] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *ISSTA '04 : Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44, New York, NY, USA, 2004. ACM Press.
- [Coh90] Fred Cohen. Computational aspects of computer viruses. *Rogue programs : viruses, worms and Trojan horses*, pages 324–355, 1990.
- [CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, New Zealand, 1997.

- [DEMS00] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2) :378–415, 2000.
- [Fil06] Eric Filiol. Malware pattern scanning schemes secure against black-box analysis. *EICAR 2006 Special Issue, V. Broucek And Paul Tuner ed, Journal in Computer Virology*, 2 (1), 2006.
- [Lan92] William Alexander Landi. *Interprocedural aliasing in the presence of pointers*. PhD thesis, New Brunswick, New Jersey, USA, 1992.
- [LKK04] A. Lakhotia, A. Kapoor, and E. U. Kumar. Are metamorphic viruses really invincible? *Virus Bulletin*, 2004.
- [OSSM02] Toshio Ogiso, Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Software tamper resistance based on the difficulty of interprocedural analysis. In *In The Third International Workshop on Information Security Applications*, pages 437–452, 2002.
- [Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, pages pp. 358–366, 1953.
- [Spi03] D. Spinellis. Reliable identification of boundedlength viruses is np-complete. *IEEE Transactions on Information Theory*, 49(1), pages 280–284, 2003.
- [SXCM04] A.H. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static analyzer of vicious executables(save). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*. IEEE, 2004.
- [WHKD00] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance : Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 2000.
- [WMCL06] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. Normalizing metamorphic malware using term rewriting. *SCAM 2006 : The 6th IEEE Workshop Source Code Analysis and Manipulation*, pages 75–84, 2006.