Étude en Coq de sémantiques d'obfuscations de programmes

Alix Trieu

Rapport de stage de 1ère année du Magistère Informatique et Télécommunications

Stage du 03 Juin au 26 Juillet dans l'équipe Celtique au centre INRIA Rennes

Encadrant : Sandrine Blazy

Résumé

Nous présentons ici une étude de la « correction » d'obfuscations de programmes. Nous commençons par spécifier la sémantique d'un langage impératif minimaliste dans l'assistant de preuve Coq, puis nous prouvons que les obfuscations que nous étudions préservent la sémantique voulue du programme originel. Nous montrons ici la démarche suivie afin d'accomplir cela, nous pouvons notamment observer qu'il y a une certaine correspondance entre la « sécurité » d'une obfuscation et la complexité de la preuve de la préservation de la sémantique.

Introduction

L'équipe Celtique dans lequel le stage s'est effectué s'intéresse à la sémantique des langages de programmation, et en particulier à la preuve en Coq de la correction de transformations de programmes en vue d'utiliser ces techniques dans la sécurité informatique. C'est dans ce cadre là, que mon stage s'est intéressé à l'obfuscation de code. En effet, cela est devenue un domaine important de la sécurité informatique durant ces dernières années. En effet, il est devenu indispensable pour les développeurs de se protéger du piratage logiciel et de protéger leur propriété intellectuelle. Il s'agit de modifier le code source ou binaire d'un programme afin de le rendre plus difficile à comprendre ou à déterminer son fonctionnement. L'obfuscation est notamment utilisée pour rendre les virus informatiques moins détectables par les logiciels anti-virus, mais aussi par des entreprises comme Apple ou Microsoft afin de protéger leurs logiciels, ou encore dans le domaine militaire afin d'éviter que les données embarquées dans des missiles ou drones puissent être utilisées par des ennemis par exemple. Ainsi, l'obfuscation est un domaine d'avenir, de part l'omniprésence de logiciels dans la vie quotidienne (lecture des cartes de transports...) et son importance. Cependant, au contraire de la cryptographie qui peut fournir des certificats électroniques, il n'existe pas d'obfuscations fiables.

Par conséquent, il est intéressant de s'intéresser à la « qualité » d'une obfuscation. Nous étudions donc la correction de cette obfuscation, nous entendons par là, qu'il faut vérifier qu'elle conserve bien la sémantique souhaitée du programme obfusqué. Il est aussi nécessaire d'essayer de quantifier la difficulté à désobfusquer [1] un programme afin d'évaluer sa « qualité ». Il existe plusieurs métriques de qualité d'obfuscation, mais elles sont principalement syntaxiques (augmentation de la taille du code, du nombre de boucles imbriquées, etc) et trop générales pour être utilisée, nous avons choisi de suivre une voie sémantique. Ainsi, notre démarche a été d'utiliser Coq, un assistant de preuve, afin de spécifier un langage impératif minimaliste et de prouver la « correction » de certaines obfuscations. Ces obfuscations touchent notamment les données ainsi que le flot de contrôle des programmes. Les preuves peuvent alors être vues comme les étapes pour désobfusquer le programme, ce qui permet alors de quantifier la difficulté par le nombre de lemmes nécessaires qu'il est aussi possible de voir comme étant les étapes que doivent traverser un attaquant pour comprendre

1. c'est-à-dire effectuer la transformation inverse de l'obfuscation.

le code obfusqué.

Nous commencerons par présenter la sémantique utilisée, puis les obfuscations étudiées qui sont le Variable Splitting, l'Array Merging et le Control Flow Flattening, ainsi que la démarche utilisée pour prouver la préservation de la sémantique.

1 Syntaxe et sémantique d'un petit langage impératif

1.1 Présentation

Nous présentons ici un petit langage impératif avec tableaux consistant en des expressions arithmétiques, booléennes et des instructions. Sa syntaxe est la suivante :

Expression arithmétique :	a := x	variable
	| n	constante ($\in \mathbb{Z}$)
	| t.[a]	tableau
	| length t	taille de tableau
	| $a_1 + a_2$	addition
	| $a_1 - a_2$	soustraction
	| $a_1 * a_2$	multiplication
	| a_1/a_2	division
Expression booléenne :	b := true	vrai
	| false	faux
	| $a_1 == a_2$	test d'égalité
	| $a_1 \leq a_2$	inférieur ou égal
	|!b	négation
	| $b_1 \wedge b_2$	conjonction
	| $b_1 \vee b_2$	disjonction
Instruction :	c := skip	instruction vide
	| x = a	assignation
	| new t l a	création de tableau [1]
	| $c_1; c_2$	séquence
	| if (b) c_1 else c_2	conditionnelle
	| while (b) c	boucle tant que

Intuitivement, un programme est une instruction qui d'un état donné de la machine, amène à un autre état. Ainsi, les états peuvent être

1. tableau t de taille l initialisé à la valeur de a de $t.[1]$ à $t.[l]$

modélisés comme des fonctions de Variable $\to \mathbb{Z} \cup \{\varnothing\}$ ($\varnothing$ pour variable non définie). En effet, un tableau de taille n peut être vu comme $n+1$ variables, l'une d'entre elle étant la longueur de tableau (nous utiliserons t.[0] pour stocker la longueur du tableau t).

Pour un état s et un programme c, nous écrivons $(s, c) \Downarrow s'$ pour signifier qu'en partant de l'état s, le programme c termine dans l'état s'. De même, $(s, e : v)$ signifie que l'expression arithmétique (respectivement booléenne) e vaut la valeur entière (respectivement booléenne) v dans l'état s. De plus, $s[x \mapsto v]$ sert à dénoter l'état s où la valeur de x a été mise à jour à v. Il est alors possible de définir la sémantique à grands pas suivante [2] :

Les expressions arithmétiques :

1. $(s, n : n)$
2. $\dfrac{\mathrm{s(x)} = \mathrm{v}}{(s, x : v)}$
3. $\dfrac{\mathrm{s(t.[0])} = \mathrm{v}}{(s, \text{length } t : v)}$
4. $\dfrac{(s, a_1 : v_1) \qquad (s, a_2 : v_2)}{(s, a_1 + a_2 : v_1 + v_2)}$
5. $\dfrac{(s, a_1 : v_1) \qquad (s, a_2 : v_2)}{(s, a_1 - a_2 : v_1 - v_2)}$
6. $\dfrac{(s, a_1 : v_1) \qquad (s, a_2 : v_2)}{(s, a_1 * a_2 : v_1 * v_2)}$
7. $\dfrac{(s, a_1 : v_1) \qquad (s, a_2 : v_2) \qquad v_2 \neq 0}{(s, a_1/a_2 : v_1/v_2)}$

Les expressions booléennes :

1. $(s,$ true : true$)$
2. $(s,$ false : false$)$
3. $\dfrac{(s, b_1 : v_1) \qquad (s, b_2 : v_2) \qquad v_1 = v_2}{(s, b_1 == b_2 : \text{true})}$
4. $\dfrac{(s, b_1 : v_1) \qquad (s, b_2 : v_2) \qquad v_1 \neq v_2}{(s, b_1 == b_2 : \text{false})}$
5. $\dfrac{(s, b_1 : v_1) \qquad (s, b_2 : v_2) \qquad v_1 \leq v_2}{(s, b_1 \leq b_2 : \text{true})}$

2. les règles sont de la forme (état, elément syntaxique : valeur) et sont très proches de la manière dont elles sont écrites en Coq

6. $\dfrac{(s, b_1 : v_1) \qquad (s, b_2 : v_2) \qquad v_1 > v_2}{(s, b_1 \leq b_2 : \text{false})}$

7. $\dfrac{(s, b : v)}{(s, !b : \neg v)}$

8. $\dfrac{(s, b_1 : v_1) \qquad (s, b_2 : v_2)}{(s, b_1 \wedge b_2 : v_1 \wedge v_2)}$

9. $\dfrac{(s, b_1 : v_1) \qquad (s, b_2 : v_2)}{(s, b_1 \vee b_2 : v_1 \vee v_2)}$

Les instructions :

1. $(s, \text{skip}) \Downarrow s$

2. $\dfrac{(s, a : v)}{(s, x = a) \Downarrow s[x \mapsto v]}$

3. $\dfrac{(s, l : m) \qquad (s, a : v)}{(s, \text{new t l a}) \Downarrow s[t.[0] \mapsto m][t.[1] \mapsto v] \ldots [t.[m] \mapsto v]}$

4. $\dfrac{(s, c_1) \Downarrow s' \qquad (s', c_2) \Downarrow s''}{(s, c_1; c_2) \Downarrow s''}$

5. $\dfrac{(s, b : \text{true}) \qquad (s, c_1) \Downarrow s'}{(s, \text{if (b) } c_1 \text{ else } c_2) \Downarrow s'}$

6. $\dfrac{(s, b : \text{false}) \qquad (s, c_2) \Downarrow s'}{(s, \text{if (b) } c_1 \text{ else } c_2) \Downarrow s'}$

7. $\dfrac{(s, b : \text{false})}{(s, \text{while (b) } c) \Downarrow s}$

8. $\dfrac{(s, b : \text{true}) \qquad (s, c) \Downarrow s' \qquad (s', \text{while (b) } c) \Downarrow s''}{(s, \text{while (b) } c) \Downarrow s''}$

2 Variable Splitting

Étant donnée une application injective de $\mathbb{Z} \to \mathbb{Z}^2$, il est possible de « séparer en deux » une variable. En effet, soit $f : x \mapsto (a(x), b(x))$ une telle application, il est alors possible de créer deux variables a et b telles que $x = f^{-1}(a, b)$. Par souci de simplicité, nous avons fixé un entier n non nul et défini f de la manière suivante $f(x) = (\lfloor x/n \rfloor, x - \lfloor x/n \rfloor \times n)$. Par conséquent, afin d'obfusquer un programme, il suffit simplement de remplacer toutes les occurrences d'un certain x choisi à l'avance par $a \times n + b$ s'il apparaît dans une expression arithmétique, et remplacer les instructions de la forme $x = e$ par la séquence $a = e/n; b = e - a \times n$.

Cela se généraliser à toutes les variables du programme, et ainsi rendre le programme initial bien plus difficile à comprendre.

Plus formellement, nous définissons 3 fonctions σ_{aexpr}, σ_{bexpr} et σ de la manière suivante.

σ_{aexpr} :

1. $\forall n \in \mathbb{Z}, \sigma_{\text{aexpr}}(n) = n$
2. $\forall y \in \text{Variable}, x \neq y \Rightarrow \sigma_{\text{aexpr}}(y) = y$
3. $\sigma_{\text{aexpr}}(x) = a * n + b$
4. $\sigma_{\text{aexpr}}(t.[e]) = t.[\sigma_{\text{aexpr}}(e)]$
5. $\sigma_{\text{aexpr}}(\text{length } t) = \text{length } t$
6. $\forall \circ \in \{+, -, *, /\}, \sigma_{\text{aexpr}}(a_1 \circ a_2) = \sigma_{\text{aexpr}}(a_1) \circ \sigma_{\text{aexpr}}(a_2)$

σ_{bexpr} :

1. $\forall v \in \{\text{true},\ \text{false}\}, \sigma_{\text{bexpr}}(v) = v$
2. $\forall \circ \in \{==, \leq\}, \sigma_{\text{bexpr}}(a_1 \circ a_2) = \sigma_{\text{aexpr}}(a_1) \circ \sigma_{\text{aexpr}}(a_2)$
3. $\sigma_{\text{bexpr}}(!b) = !\sigma_{\text{bexpr}}(b)$
4. $\forall \circ \in \{\wedge, \vee\}, \sigma_{\text{bexpr}}(b_1 \circ b_2) = \sigma_{\text{bexpr}}(b_1) \circ \sigma_{\text{bexpr}}(b_2)$

σ :

1. $\sigma(\text{skip}) = \text{skip}$
2. $\forall y \in \text{Variable}, x \neq y \Rightarrow \sigma(y = e) = (y = \sigma_{\text{aexpr}}(e))$
3. $\sigma(x = e) = (a = e/n; b = e - a * n)$
4. $\sigma(t.[e] = a) = t.[\sigma_{\text{aexpr}}(e)] = \sigma_{\text{aexpr}}(a)$
5. $\sigma(\text{new } t\ l\ e) = \text{new } t\ \sigma_{\text{aexpr}}(l)\ \sigma_{\text{aexpr}}(e)$
6. $\sigma(c_1; c_2) = \sigma(c_1); \sigma(c_2)$
7. $\sigma(\text{if } (b)\ c_1 \text{ else } c_2) = \text{if } \sigma_{\text{bexpr}}(b)\ \sigma(c_1) \text{ else } \sigma(c_2)$
8. $\sigma(\text{while } b\ c) = \text{while } \sigma_{\text{bexpr}}(b)\ \sigma(c)$

Nous avons alors défini l'obfuscation d'un programme c comme étant $\sigma(c)$. Il nous reste donc à trouver un théorème de préservation sémantique liant $(s, c) \Downarrow\ s'$ à $\sigma(c)$. Pour cela, définissons une fonction d'obfuscation de l'état mémoire θ : état $\rightarrow$ état par $\theta(s) = s[a \mapsto s(x)/n][b \mapsto s(x) - (s(x)/n) * n][x \mapsto \varnothing]$ (avec pour convention $\varnothing/n = \varnothing$ et $\varnothing - \varnothing * n = \varnothing$).

Finalement, le théorème recherché est :

$$\forall s, \forall s', \forall c, (s, c) \Downarrow s' \rightarrow (\theta(s), \sigma(c)) \Downarrow \theta(s')$$

En effet, seul ce sens nous intéresse, car nous cherchons uniquement à montrer que le programme obfusqué se comporte comme le programme initial et non l'inverse. De plus, bien qu'implicite, lors de la preuve en Coq, il faut néanmoins indiquer que a et b sont des « variables fraîches » (n'apparaissent pas dans c lors de son exécution) et distinctes l'une de l'autre. La preuve de la préservation de la sémantique se fait alors assez facilement par induction sur la structure du programme.

3 Array Merging

L'Array Merging, comme son nom l'indique, est une obfuscation qui agit sur les données du programme et fusionne les tableaux. En effet, étant donnés deux tableaux u et v respectivement de tailles l et m, il est possible de créer un tableau t de taille $l + m$, dont les l premières valeurs correspondent aux valeurs de u, tandis que les suivantes correspondent aux valeurs de v. À nouveau, afin d'obfusquer le code, il suffit de supprimer les occurrences de u et de v et de les remplacer de la manière adéquate. Néanmoins, contrairement au cas précédent, cette obfuscation ne fonctionne pas sur n'importe quel programme. En effet, que faire si seulement u est déclaré et non v, ou inversement ?

Par conséquent, nous considérons uniquement le code après que u et v aient été déclarés, c'est-à-dire qu'il n'y a pas d'occurrences de la forme new $u\ l\ e$ ou new $v\ l\ e$. De la même manière que précédemment, nous définissons σ_{aexpr}, σ_{bexpr} et σ.

σ_{aexpr} :

1. $\forall n \in \mathbb{Z}, \sigma_{\text{aexpr}}(n) = n$
2. $\forall x \in \text{Variable}, \sigma_{\text{aexpr}}(x) = x$
3. $\sigma_{\text{aexpr}}(u.[e]) = t.[\sigma_{\text{aexpr}}(e)]$
4. $\sigma_{\text{aexpr}}(v.[e]) = t.[\sigma_{\text{aexpr}}(e) + l]$
5. $\sigma_{\text{aexpr}}(a.[e]) = a.[\sigma_{\text{aexpr}}(e)]$
6. $\sigma_{\text{aexpr}}(\text{length } u) = l$
7. $\sigma_{\text{aexpr}}(\text{length } v) = \text{length } t - l$
8. $\sigma_{\text{aexpr}}(\text{length } a) = \text{length } a$
9. $\forall \circ \in \{+, -, *, /\}, \sigma_{\text{aexpr}}(a_1 \circ a_2) = \sigma_{\text{aexpr}}(a_1) \circ \sigma_{\text{aexpr}}(a_2)$

σ_{bexpr} :

1. $\forall v \in \{\text{true},\ \text{false}\}, \sigma_{\text{bexpr}}(v) = v$
2. $\forall \circ \in \{==, \leq\}, \sigma_{\text{bexpr}}(a_1 \circ a_2) = \sigma_{\text{aexpr}}(a_1) \circ \sigma_{\text{aexpr}}(a_2)$
3. $\sigma_{\text{bexpr}}(!b) = !\sigma_{\text{bexpr}}(b)$
4. $\forall \circ \in \{\wedge, \vee\}, \sigma_{\text{bexpr}}(b_1 \circ b_2) = \sigma_{\text{bexpr}}(b_1) \circ \sigma_{\text{bexpr}}(b_2)$

σ :

1. $\sigma(\text{skip}) = \text{skip}$
2. $\sigma(u.[e] = a) = (t.[\sigma_{\text{aexpr}}(e)] = \sigma_{\text{aexpr}}(a))$
3. $\sigma(v.[e] = a) = (t.[\sigma_{\text{aexpr}}(e) + l] = \sigma_{\text{aexpr}}(a))$
4. $\sigma(w.[e] = a) = (w.[\sigma_{\text{aexpr}}(e)] = \sigma_{\text{aexpr}}(a))$
5. $\sigma(x = e) = (x = \sigma_{\text{aexpr}}(e))$
6. $\sigma(\text{new } t\ l\ e) = \text{new } t\ \sigma_{\text{aexpr}}(l)\ \sigma_{\text{aexpr}}(e)$
7. $\sigma(c_1; c_2) = \sigma(c_1); \sigma(c_2)$
8. $\sigma(\text{if } (b)\ c_1 \text{ else } c_2) = \text{if } \sigma_{\text{bexpr}}(b)\ \sigma(c_1) \text{ else } \sigma(c_2)$
9. $\sigma(\text{while } b\ c) = \text{while } \sigma_{\text{bexpr}}(b)\ \sigma(c)$

De même, nous définissons θ comme étant la fonction prenant un état en argument et renvoie l'état correspondant où le tableau t est initialisé aux valeurs adéquates et u et v effacés [3]. Ainsi, en supposant que c vérifie les conditions précédentes, nous pouvons alors prouver par induction que :

$$\forall s, \forall s', (s, c) \Downarrow s' \rightarrow (\theta(s), \sigma(c)) \Downarrow \theta(s')$$

Ce théorème est le même que le précédent et il en est de même pour sa preuve, nous en déduisons que ces deux obfuscations sont d'une qualité équivalente, ce qui peut sembler intuitif puisqu'elles complémentaires l'une de l'autre.

4 Control Flow Flattening

La dernière obfuscation étudiée concerne le flot de contrôle du programme, c'est-à-dire que l'ordre d'exécution des instructions est modifié, mais de manière à ce que globalement le résultat soit le même. Pour cela, nous avons tout d'abord introduit une nouvelle instruction :

3. Plus formellement, $\theta(s) = s[t.[0] \mapsto l + m][t.[1] \mapsto u.[1]] \dots [t.[l] \mapsto u.[l]][t.[l+1] \mapsto v.[1]] \dots [t.[l+m] \mapsto v.[m]][u.[0] \mapsto \varnothing] \dots [u.[l] \mapsto \varnothing][v.[0] \mapsto \varnothing] \dots [v.[m] \mapsto \varnothing]$

$\ll$ switch e l $\gg$ [4] où e est une expression arithmétique et l une liste de couple $\mathbb{Z}\times$Instruction. Sa sémantique est la suivante :

1. $\dfrac{(s, e : n)}{(s, \text{switch } e \text{ nil}) \Downarrow s}$
2. $\dfrac{(s, e : n) \qquad n = k \qquad (s, c) \Downarrow s'}{(s, \text{switch } e \ (\text{cons } (k, c) \ l)) \Downarrow s'}$
3. $\dfrac{(s, e : n) \qquad n \neq k \qquad (s, \text{switch } e \ l) \Downarrow s'}{(s, \text{switch } e \ (\text{cons } (k, c) \ l)) \Downarrow s'}$

L'obfuscation modifie les boucles while de la manière suivante :

Code original

```
1  x = 2;
2  y = 5;
3  z = 0;
4  while (0 ≤ y) {
5  z = z + x;
6  y = y − 1;
7  }
```

Code obfusqué

```
1   x = 2;
2   y = 5;
3   z = 0;
4   pc = 3;
5   while (4 ≤ pc) {
6   switch pc
7   4 : if (0 ≤ y) then
8       (pc = 5) else (pc = −1);
9   5 : z = z + x; pc = 6;
10  6 : y = y − 1; pc = 4;
11  }
```

Plus concrètement, chaque instruction est étiquetée dans l'ordre croissant du parcours du programme original (numéro de ligne dans l'exemple), la variable pc (program counter) est alors initialisée à la valeur de l'étiquette n du while ($n = 4$ dans l'exemple) qui doit être obfusqué. Ensuite, la condition de la boucle est modifiée à (n $\leq$ pc), ainsi que le corps de la boucle afin de faire apparaître l'instruction switch de telle manière que le cas (pc = n) permet de décider si la boucle doit terminer ou non. Si la boucle doit terminer, il suffit alors d'assigner -1 à pc, dans le cas contraire, pc est mis à jour à la valeur n_1 correspondant à la première instruction contenu dans le corps de la boucle original. Le cas (pc = n_1) exécute alors son instruction correspondant et met à jour adéquatement pc pour atteindre l'instruction suivante et continuer ainsi de suite ou alors revenir au cas (pc = n) s'il s'agissait de la dernière instruction du corps de boucle.

4. qui n'est rien d'autre qu'une disjonction de cas de la forme if ($e == n_1$) then ... else if ($e == n_k$) then ... else skip

Notons étiquette(c) la première étiquette rencontrée dans le programme c. Nous définissons alors une fonction $\alpha : (\mathbb{N} \times \text{Instruction}) \rightarrow (\mathbb{Z} \times \text{Instruction})$ list [5] par :

- $\alpha(n, (c_1; c_2)) = \text{cons}$ (étiquette(c_1), (c_1 ; pc = étiquette(c_2))) ($\alpha(n, c_2)$)
- $\alpha(n, c) =$ cons (étiquette(c), (c; pc = n)) nil si c n'est pas une séquence.

Avec la notation $n =$ étiquette(while (b) c) et $m =$ étiquette(c) [6], nous pouvons finalement définir l'obfuscation par :

σ(while (b) c) = (pc = n) ; while (n $\leq$ pc) (switch pc (cons (n, if (b) (pc = m) else (pc = -1)) $\alpha(n,$ c))).

Le théorème recherché est alors :

$$(s, \text{while (b) } c) \Downarrow s' \rightarrow (s, \sigma(\text{while (b) } c)) \Downarrow s'[\text{pc} \mapsto -1]$$

Néanmoins, la démonstration ne se fait plus aussi bien par induction comme auparavant, mais nécessite plusieurs lemmes intermédiaires. Avec les mêmes notations pour n et m que précédemment, le premier lemme s'écrit alors : $(s, \text{while (b) } c) \Downarrow s' \rightarrow (s[\text{pc} \mapsto n], \text{while } (n \leq \text{pc}) \text{ (if (pc == } n) \text{ (if (b) (pc} = m) \text{ else pc} = -1) \text{ else } (c; \text{pc} = n))) \Downarrow s'[\text{pc} \mapsto -1]$. Ou plus informellement, les deux codes suivants s'exécutent « à peu près » de la même manière :

```
1  while (b) {
2  c
3  }
```

```
1  while (n ≤ pc) {
2  if (pc == n) then
3          if (b) then
4          pc = m else pc = −1
5      else c; pc = n
6  }
```

Ce lemme est relativement facile à démontrer, en effet, nous pouvons remarquer qu'un passage dans la boucle de gauche correspond à deux passages dans la boucle de droite.

Le second lemme [7] affirme que les deux codes suivants sont équivalents :

5. il s'agit de la fonction qui prend le corps de boucle et le transforme de la manière adéquate à utiliser dans le switch

6. nous avons $n < m$ avec ces notations

7. plus formellement, $(s[\text{pc} \mapsto n], \text{while } (n \leq \text{pc}) \text{ (if (pc == } n) \text{ (if (b) (pc} = m) \text{ else pc} = -1) \text{ else } (c; \text{pc} = n))) \Downarrow s' \rightarrow (s[\text{pc} \mapsto n], \text{while } (n \leq \text{pc}) \text{ (if (pc == } n) \text{ (if (b) (pc} = m) \text{ else pc} = -1) \text{ else (while } (n < \text{pc}) \text{ switch pc } \alpha(n, c)))) \Downarrow s'$.

```
1  while (n ≤ pc) {
2  if (pc == n) then
3     if (b) then
4          pc = m else
5          pc = −1
6  else c; pc = n
7  }
```

```
1  while (n ≤ pc) {
2  if (pc == n) then
3       if (b) then
4       pc = m else
5       pc = −1
6  else while (n < pc) {
7            switch pc
8            m : ...
                 ⋮
9
10           k : ...; pc = n;
11      }
12 }
```

Il suffit de remarquer qu'exécuter (c; pc = n) revient aussi à exécuter (while (n < pc) switch pc ...).

Le troisième lemme[8] affirme que les deux codes suivants sont équivalents :

```
1  while (n ≤ pc) {
2  if (pc == n) then
3       if (b) then
4       pc = m else
5       pc = −1
6  else while (n < pc) {
7            switch pc
8            m : ...
                 ⋮
9
10           k : ...; pc = n;
11      }
12 }
```

```
1  while (n ≤ pc) {
2  if (pc == n) then
3       if (b) then
4       pc = m else
5       pc = −1
6  else switch pc
7            m : ...
                 ⋮
8
9            k : ...; pc = n;
10 }
```

La démonstration se fait par induction structurelle. Finalement, la dernière étape s'appuie simplement sur le fait que l'instruction switch n'est qu'une disjonction de cas :

8. plus formellement, $(s[\text{pc} \mapsto n], \text{while } (n \leq \text{pc}) \text{ (if (pc == } n\text{) (if (b) (pc} = m\text{) else pc} = -1\text{) else (while } (n < \text{pc}) \text{ switch pc } \alpha(n, c)))) \Downarrow s' \rightarrow (s[\text{pc} \mapsto n], \text{while } (n \leq \text{pc}) \text{ (if (pc == } n\text{) (if (b) (pc} = m\text{) else pc} = -1\text{) else (switch pc } \alpha(n, c))))$.

```
1  while (n ≤ pc) {
2  if (pc == n) then
3       if (b) then
4       pc = m else
5       pc = −1
6  else switch pc
7          m : ...
8                ⋮
9          k : ...; pc = n;
10 }
```

```
1  while (n ≤ pc) {
2  switch pc
3    n : if (b) then
4        pc = m else
5        pc = −1
6    m : ...
7            ⋮
8    k : ...; pc = n;
9  }
```

Par rapport aux deux obfuscations précédentes, la preuve a été cette fois ci bien plus difficile. En effet, il a été nécessaire de décomposer l'obfuscation en transformations élémentaires tout en réussissant à trouver celles dont la correction pouvaient être prouvée. La démarche pour réaliser cette preuve ressemble à celle qu'aurait suivi un attaquant, en partant du code obfusqué et remonter jusqu'à un code plus facilement compréhensible. Cependant, nous avons l'avantage de connaître le programme initial afin de réaliser la preuve, ce qui nous a permis de plus facilement trouver certaines transformations.

Conclusion

Lors de ce stage, j'ai pu découvrir divers domaines de l'informatique allant de la sécurité aux méthodes formelles, ainsi qu'apprendre à apprivoiser l'assistant de preuve Coq. Les obfuscations étudiées sont relativement simples, mais nous pouvons tout de même remarquer que dans le dernier cas, la preuve fournie peut ressembler à la démarche que doit accomplir quelqu'un pour comprendre le code obfusqué. En généralisant cette observation, nous pouvons alors considérer que la difficulté de la preuve est liée à la complexité de l'obfuscation.

Finalement, le langage utilisé ici pourrait être remplacé par un sous ensemble du langage C, le CompCert C, dont la sémantique est déja spécifiée en Coq, afin d'avoir un langage utilisé professionnellement. La difficulté serait alors de gérer des entiers machines (32 bits ou 64 bits) au lieu de valeurs dans $\mathbb{Z}$. De plus, il serait intéressant d'étudier d'autres propriétés telles que la minimalité des obfuscations étudiées par exemple.

Remerciement

Je tiens tout d'abord à remercier Sandrine Blazy pour m'avoir donné un stage passionnant, et permis d'avoir la chance d'assister à une vraie conférence scientifique lors de l'ITP. Je remercie aussi toute l'équipe Celtique qui m'a accueilli, et plus particulièrement Vincent Laporte et André Oliveira Maroneze pour toute l'aide qu'ils m'ont apportée dans les arcanes de Coq et les idées qu'ils m'ont fournies lors de mes tentatives de démonstration.

Références

[BG12] Sandrine Blazy and Roberto Giacobazzi. Towards a formally verified obfuscating compiler. In Christian Collberg, editor, *SSP 2012 - 2nd ACM SIGPLAN Software Security and Protection Workshop*, Beijing, Chine, June 2012. ACM SIGPLAN, ACM SIGPLAN.

[Coq] Coq Development Team. *Reference Manual*. `http://coq.inria.fr/refman/index.html`.

[GJM12] Roberto Giacobazzi, Neil D. Jones, and Isabella Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 63–72, New York, NY, USA, 2012. ACM.

[NC09] Jasvir Nasgra and Christian Collberg. *Surreptitious Software : Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[Pie] Benjamin C. Pierce. *Software Foundations*. `http://www.cis.upenn.edu/~bcpierce/sf/`.