Désobscurcissement de prédicats opaques

Dylan Marinho

STAGE DE FIN D'ANNÉE

Sous la direction de Jean-Yves Marion

Désobscurcissement de prédicats opaques

Rapport de stage

Dylan Marinho

Master II: Sécurité, Défense et Intelligence Stratégique

Version du 20 juin 2023

Table des matières

Partie 1

Préambule

1.1 Contexte

Ce stage a été réalisé au sein du Laboratoire Lorrain de Recherche en Informatique et ses Applications (LORIA). Sous la direction de Jean-Yves Marion, responsable d'équipe et directeur du LORIA, il s'est déroulé au sein de l'équipe Carbone. Cette équipe s'intéresse notamment à des aspects de sécurité, en particulier de la détection de logiciels malveillants.

Ce stage de recherche s'est orienté sur les questions de désobscurcissement de code, outil nécessaire afin de procéder à l'analyse des *malware* [1]. Le rapport présenté en Partie 2 prend la forme d'un article de recherche, avec le plan usuel présentant l'état de l'art, puis la méthode proposée et enfin la validation. Une section supplémentaire (Section 2.2) permet de présenter des éléments essentiels à la compréhension des enjeux et du travail réalisé à un public non initié à la science informatique.

En raison de l'épidémie de COVID-19, ce stage a été entièrement réalisé en télétravail.

1.2 Avertissement

Ce rapport de stage étant à destination des enseignants de Sciences Po Rennes et prenant la forme d'un article de recherche, le choix d'ajouter une partie sup-

1. C'est-à-dire des logiciels malveillants. On ne parlera pas de « virus », ceux-ci ne représentant qu'une sous-famille des *malware*.

plémentaire (Section 2.2) permettant de présenter simplement des bases informatiques et une meilleure compréhension a été fait. Celui-ci se justifie notamment par le souhait d'avoir à la fois une explication simple des outils utilisés mais aussi de garder un contenu formel – et non simplifié – dans la suite du raisonnement.

Cette partie, spécifiquement destinée aux lecteurs de ce rapport n'ayant pas de formation en science informatique, présente donc de façon simplifiée les prérequis nécessaires à la compréhension de la suite du document. Ces différents points sont volontairement grandement simplifiés et peuvent paraître parfois erronés pour un public différent. Il est important de noter que les notions alors introduites ne seront pas forcément définies de façon formelle dans la suite du document.

En outre, dans cette partie (Section 2.2), on ne distinguera volontairement pas les notions de « programme », « algorithme » ou de « code ». Bien que l'outil s'adresse à des programmes rédigés en C/C++, les codes présentés en exemple seront, dans tout le rapport et sauf mention contraire, présentés dans un pseudo-code, inspiré de Python, afin de permettre une lecture plus aisée.

Enfin, les termes d'« *obfuscation* » et de « *déobfuscation* » seront utilisés tout au long du rapport, alors que les termes « obscurcissement » et « désobscurcissement » sont plus appropriés en langue française. Cet anglicisme est conservé volontairement afin de concorder avec le vocabulaire utilisé dans ce milieu.

Autant que possible, tous les termes techniques ont été traduits en langue française et utilisés en lieu et place des noms anglais (sauf dans certains cas). Cependant, afin de permettre une compréhension par les informaticiens habitués à ces outils, les noms originels (en anglais) sont systématiquement donnés et, dans quelques cas, préférés à leur traduction.

Partie 2

Rapport de recherche

Désobscurcissement de prédicats opaques

Dylan Marinho

LORIA, Nancy

Mots-clés : Sécurité, Obfuscation, Obscurcissement, Prédicats, Opaques

Référence : Stage réalisé du 20/04 au 24/07/2020 au sein de l'équipe Carbone, LORIA, Nancy sous la supervision de Jean-Yves Marion.

2.1 Introduction

L'obfuscation de code est aujourd'hui utilisée comme une méthode permettant de protéger un logiciel, notamment face à des enjeux de propriété intellectuelle. En effet, il permet d'empêcher – ou de rendre plus difficile – des analyses par rétro-ingénierie tout en conservant un comportement identique. Cette fonction est également utilisée par les auteurs de logiciels malveillants : en rendant difficile l'analyse de leur code, ces derniers peuvent se propager puisqu'ils ne pourront pas être détectés comme *malware*.

L'obfuscation pose donc un problème scientifique dès lors qu'il est question de détecter de tels programmes : sans palier à cette étape de leur conception, il est impossible de déterminer si un logiciel est malveillant. Des méthodes de déobfuscation ont alors été mises en oeuvre, mais sont aujourd'hui encore insuffisamment performantes.

Ce travail de recherche s'est alors intéressé à permettre la déobfuscation d'un certain type d'obfuscation, dit par « prédicat opaque », en s'interdisant de se concentrer sur des modèles spécifiques de constructions. Les contributions mises en avant sont donc :

— une analyse de l'état actuel des méthodes d'obfuscation et de déobfuscation par la réalisation d'une analyse bibliographique poussée ;
— la proposition d'une méthode de déobfuscation, statique et basée sur une classification par *machine learning*, de prédicats opaques variants (*two-ways opaque predicates*).

Ce rapport de recherche commencera par présenter des pré-requis en informatique (Section 2.2) et sera ensuite articulé autour des trois étapes du travail de

recherche. La Section 2.3 introduira donc les notions spécifiques à la déobfuscation de prédicats opaques. Ensuite, la Section 2.4 détaillera la méthode choisie afin de résoudre le problème évoqué. Enfin, la Section 2.5 proposera des pistes afin de procéder à la validation de ce procédé.

2.2 Pré-requis en Informatique

Rappel. Cette section est destinée à un public non-informaticien afin de présenter les bases nécessaires à la compréhension du problème posé ou des notions introduites par la suite. Les points évoqués sont donc volontairement simplifiés et peuvent parfois paraître erronés pour un public initié.

2.2.1 Notion de programmes, boucles, prédicats

Programme

Un programme est une suite d'instructions communiquées à l'ordinateur. Par exemple, la Figure 2.1 représente un programme [1] qui permet d'ouvrir le fichier `file.txt`, le lire puis le fermer.

```
1 pathname = 'file.txt'
2 f = open(pathname)
3 content = f.read()
4 f.close()
```

FIGURE 2.1 – Exemple de programme simple

Boucle

Une boucle est une instruction qui permet d'obtenir un comportement non-séquentiel d'un programme. On distingue alors :

— les boucles conditionnelles, qui permettent de choisir une suite d'instruction ou une autre ;

— les boucles itératives, qui permettent d'itérer une partie du programme.

On donne, en Figure 2.2, un exemple de programme plus complexe, incluant des boucles.

1. Ce programme est ici écrit dans un pseudo-code, inspiré de Python. Pour rappel, dans cette partie on ne fait aucune distinction entre « programme », « algorithme » et « code ».

```
1 tab = (1,20,5,4,17)
2 maxi = 0
3 for k in tab:
4   if k>maxi:
5       maxi=k
```

FIGURE 2.2 – Exemple de programme comportant des boucles (une conditionnelle et une itérative)

Ce programme va exécuter les instructions suivantes :
— définir une liste de nombres et une variable `maxi`, initialisée à 0 ;
— pour chaque élément de la liste,
  — si cet élément est plus grand que la valeur stockée dans `maxi`,
    — fixer la valeur de `maxi` à la valeur de l'élément.

On cherche donc ici à trouver la valeur maximum parmi les nombres stockés dans la liste.

Ce programme comporte deux types de boucles :

En ligne 3 on trouve une boucle itérative, afin de répéter les lignes suivantes pour chaque élément ;

En ligne 4 on trouve une boucle conditionnelle, afin d'exécuteur la ligne suivante si la condition k>maxi est vraie.

Prédicat

Un prédicat est une formule logique que l'on peut évaluer, soit à `Vrai` soit à `Faux`. Celui-ci est exprimé dans une syntaxe mathématique.

Par exemple, $x > y$ est un prédicat, qui est Vrai si x est (strictement) plus grand que y, Faux sinon.

Dans la suite du document, on donnera des exemples de prédicats. Le Tableau 2.1 donne donc la signification des symboles utilisés.

Remarque. Une boucle conditionnelle est donc constituée d'un prédicat (**if** predicat).
Il est aussi possible de définir une boucle itérative à partir d'un prédicat. On la note `while` et elle exécute les lignes incluses *tant que* le prédicat est vrai.

Symbole	Nom (Prononciation)	Exemple
$=$	Égal	$x = y$ si x et y ont la même valeur
$\neg$	Négation (« non »)	$\neg x$ est Vrai si x est Faux (et inversement)
$\wedge$	Et	$x \wedge y$ est Vrai si x *et* y sont Vrais
$\vee$	Ou	$x \vee y$ est Vrai si x *ou* y est Vrai

TABLEAU 2.1 – Définition de quelques symboles logiques

2.2.2 *Machine Learning*, Intelligence Artificielle

Le « *machine learning* » regroupe des méthodes d'apprentissage automatique. On ne présente ici que des pistes afin de donner une compréhension large de l'utilisation des modèles dans le cas d'une classification.

Remarque. Ces notions étant couramment utilisées dans le domaine, les définitions rigoureuses ne seront pas introduites dans la suite du rapport.

La classification

Les méthodes de *machine learning* permettent, entre autres, de créer des « classifieurs ». On peut définir un classifieur comme un objet qui cherche à associer, à un élément, une classe. On peut, par exemple, vouloir créer un classifieur qui, à un nombre, associe l'une des classes « est pair » ou « est impair » [2].

La notion de modèles

Il existe un grand nombre de modèles utilisables pour faire du « *machine learning* ». On peut citer les « *random forests* », les « *neural networks* » (ou « réseaux de neurones »), les « *decision trees* » (ou « arbres de décision »), les SVM, etc. Afin d'illustrer leur fonctionnement, on décrit ici les arbres de décisions [3].

Un arbre de décision est un objet qui contient des noeuds symbolisant une question. Les « fils » de ces noeuds représentent alors les différentes réponses possibles. Une fois arrivé « en bas » de cet arbre (ce qu'on appelle les « feuilles »), une classe est proposée.

2. Dans le cas d'un classifieur à deux classes on peut aussi parler de « détecteur ». Dans cet exemple, on peut considérer que l'on cherche à « détecter » les nombres pairs (ou impairs).

3. En pratique, ce modèle est rarement utilisé dans le contexte que nous décrirons, pour préférer les réseaux de neurones ou les *random forests*. Il permet cependant une représentation claire et un schéma d'apprentissage facilement compréhensible.

Exemple 1. *Considérons un exemple où l'on souhaite créer un arbre de décision qui permet de déterminer si un véhicule est une voiture (on crée donc un « détecteur » – ou classifieur dans le cas général – pour la classe « voiture »). La Figure 2.3 (page 12) donne un arbre possible* [4].

Dans cet exemple, on base la décision sur un certain nombre de propriétés (que l'on appelle « features »*), à savoir :*

- *si le véhicule comporte un volant ;*
- *le nombre de roues ;*
- *le poids du véhicule ;*
- *le nombre de portes du véhicule.*

On peut donc représenter ces variables sous la forme d'un « vecteur », c'est à dire une suite de nombres (on choisira 0 pour représenter Faux, 1 pour Vrai). Ainsi :

- *Une voiture de cinq portes aura pour vecteur* $(1;4;1,2;5)$ *(contient un volant, a quatre roues, pèse 1,2 tonne et a cinq portes) ;*
- *un vélo aura pour vecteur* $(0;2;0,01;0)$ *(n'a pas de volant, a deux roues, pèse 10 kg et n'a pas de portes).*

Les vecteurs ainsi formés sont appelés « vecteurs de *features* ». Ce sera l'argument élémentaire pour permettre le « *machine learning* ». Dans le cas général, la définition de ces vecteurs est toujours réalisée par l'« humain ».

L'apprentissage automatisé d'un modèle

Dans la partie précédente, l'arbre de décision a été créé à la main. Cependant, en pratique, les modèles sont générés par ordinateur (ce que l'on appelle « appris »).

Phase d'apprentissage. Dans le cas d'un arbre de décision, le calcul est réalisé de la façon suivante :

1. on choisit la « *feature* » qui discrimine le plus le résultat (en cas d'égalité, on en choisit une au hasard [5]) ;
2. pour chaque valeur possible de la *feature*, on cherche la *feature* suivante qui discrimine le plus (on notera que la *feature* peut être différente en fonction de la réponse prise à l'étape précédente) ;

4. On y choisit de représenter la réponse « Oui » par « Y », pour éviter toute confusion avec « zéro ».

5. On choisit généralement de prendre la « *feature* » aléatoirement parmi les cas d'égalité. On pourrait également prendre la *feature* qui arrive en premier parmi ces cas.

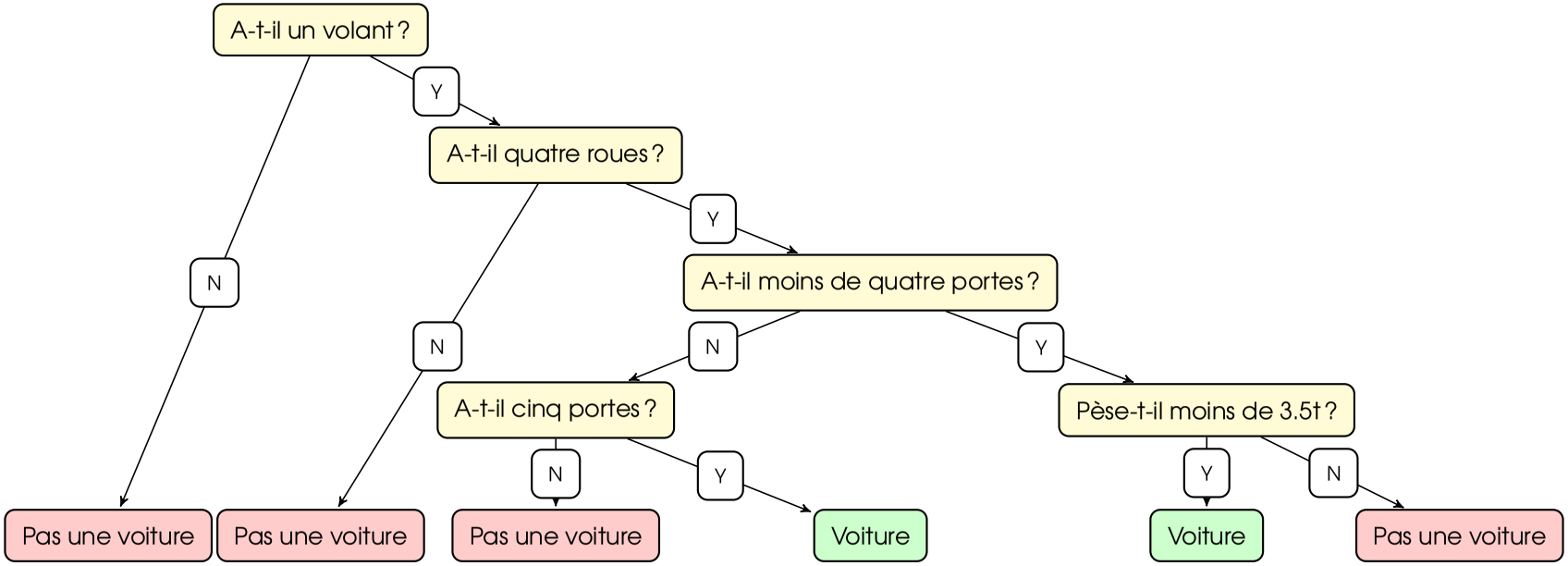


FIGURE 2.3 – Exemple d'un arbre de décision déterminant si un véhicule est une voiture

3. on recommence jusqu'à n'obtenir que des éléments de la même classe.

Afin de procéder à cette étape, un premier jeu de données est nécessaire (un « *dataset* »), que l'on appelle « dataset d'apprentissage ». On remarquera aussi que le choix des éléments de ce dataset peut impacter grandement le résultat de l'arbre. Pour illustrer cet aspect, on choisit deux exemples extrêmes.

— on choisit un dataset comportant de nombreuses voitures (vecteurs de la forme $(1;4;..;..)$) et des vélos (vecteurs de la forme $(0;2;..;0)$). L'algorithme d'apprentissage choisira, de façon aléatoire (puisqu'ils sont autant discriminants) :
  — si le véhicule comporte un volant ;
  — si le véhicule a 2 ou 4 roues ;
  — si le véhicule a des portes.
  Dans tous ces cas, on obtiendra un arbre de hauteur 1 (un seul « étage » de question). Par exemple, la Figure 2.4 montre un arbre de décision de hauteur un : pour distinguer un vélo d'une voiture, il suffit de regarder le nombre de roues.
— on choisit un dataset comportant des voitures (vecteurs de la forme $(1;4;..;..)$) et des camions (vecteurs de la forme $(1;4;..;..)$). Cette fois, l'algorithme discriminera ces exemples par rapport au poids du véhicule [6] (tous comportent un volant et quatre roues). La Figure 2.5 en donne une représentation.

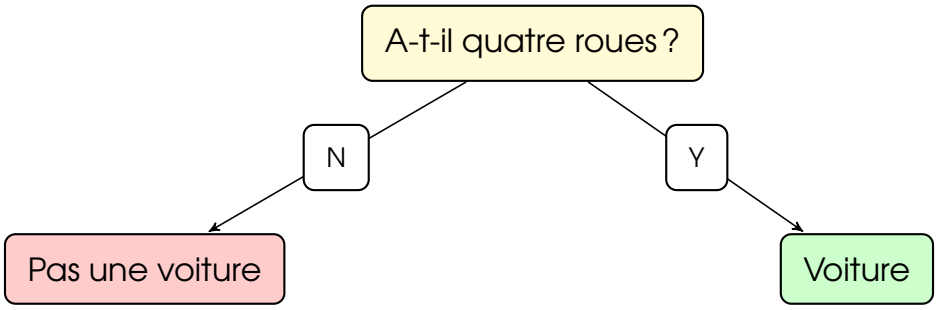


FIGURE 2.4 – Cas d'un apprentissage avec un seul niveau : vélo ou voiture

Phase de classification. Une fois le modèle appris, on suit l'arbre avec le vecteur de l'objet que l'on cherche à classifier. Une fois arrivé à une feuille, on obtient une réponse.

Exemple 2. *Reprenons l'arbre de la Figure 2.3. On cherche à déterminer si un objet de*

6. ou éventuellement par le nombre de portes, en fonction des exemples

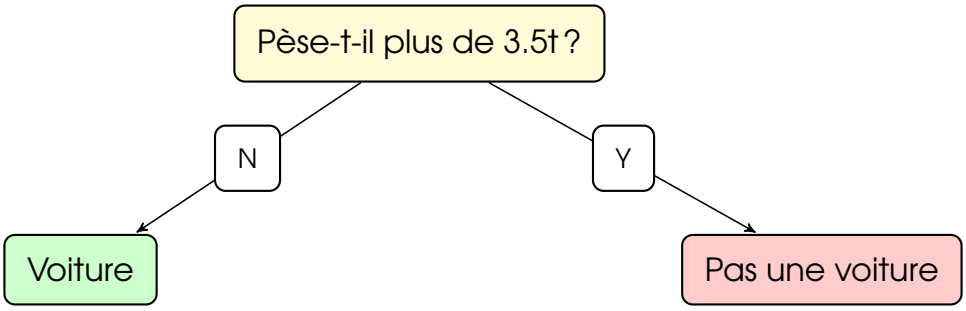


FIGURE 2.5 – Cas d'un apprentissage avec un seul niveau : voiture ou camion

vecteur (1;4;4;3) *(véhicule avec un volant, quatre roues, pèse 4t et a trois portes : un camion).*

- *A l'étage 1 (A-t-il un volant ?), la réponse est « Oui », on suit le chemin à droite.*
- *A l'étage 2 (A-t-il quatre roues ?), la réponse est « Oui », on suit le chemin à droite.*
- *A l'étape 3 (A-t-il moins de quatre portes ?), la réponse est « Oui », on suit à droite.*
- *A l'étape 4 (Pèse-t-il moins de 3.5t ?), la réponse est « Non » : ce n'est pas une voiture.*

Exemple 3 (Avec un mauvais jeu d'apprentissage)**.** *Reprenons le premier dataset donné précédemment où l'apprentissage n'est fait qu'avec un jeu contenant des voitures et des vélos (Figure 2.4). Avec le même vecteur, on se demande si le véhicule a quatre roues, ce qui est le cas. On en déduira donc que c'est une voiture.*

Remarque. En pratique, le jeu n'est pas mauvais au sens strict, mais il n'est pas adapté pour créer un classifieur où l'on souhaite distinguer un camion d'une voiture.

2.2.3 Code binaire et *Control Flow Graph*

Afin d'être exécuté par l'ordinateur, le programme sera transformé en code binaire.

Pour permettre l'analyse de ce code, nous le transformons en un « *control flow graph* » (ou « graphe de flow de contrôle »), abrégé en CFG. Celui-ci représente les liens existants entre les différents blocs du code.

Exemple 4 (Conjecture de Syracuse)**.** *Considérons le programme suivant :*

```
1  s = 10
2  while s ≠ 1:
3      if s % 2 = 0:        %si s est pair
4          s = s/2
```

```
5        else :                    %si  s  est  impair
6            s  =  3*s+1
```

On y calcule les termes de la suite dite « de Syracuse » : à chaque étape,

— *si s est pair, le terme suivant est $s/2$;*

— *si s est impair, le terme suivant est $3 * s + 1$.*

La conjecture de Syracuse énonce alors que cette suite atteindra 1.

Le CFG correspondant à ce programme est présenté en Figure 2.6.

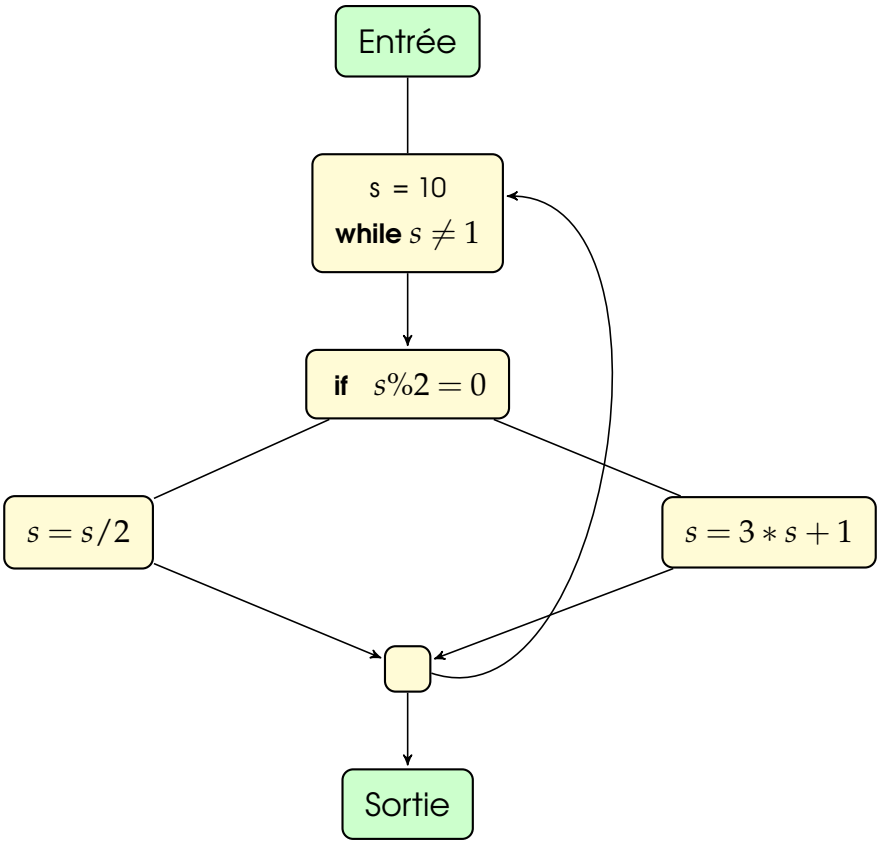


FIGURE 2.6 – CFG du code présenté en Exemple 4

On remarque alors que l'utilisation d'un prédicat (généralement au sein d'une boucle conditionnelle `if` ou d'une boucle itérative `for` ou `while`) provoque deux sorties possibles à un bloc (dans le cas où le prédicat est Vrai et dans le cas où il est Faux). Ces différents instants du programmes sont nommés « sauts conditionnels ».

2.3 Notions préliminaires

2.3.1 *Control Flow Graph*

Afin de permettre l'analyse de codes binaires, ceux-ci sont représentés sous la forme de « graphes de flots de contrôle ». Ces derniers permettent de donner une représentation sous forme de graphe des liens entre les blocs d'instructions consécutives. On donne ici une définition formelle de ces graphes et un exemple.

Définition 1 (Graphe de flot de contrôle – *Control Flow Graph* (CFG))**.** *Un graphe de flot de contrôle (CFG) d'un code $\mathcal{C}$ est un graphe CFG($\mathcal{C}$) tel que :*

- *un sommet représente un bloc d'instructions séquentiel maximal,* ie. *un bloc d'instruction de taille maximale sans saut ni cible de sauts ;*
- *il existe deux sommets spéciaux, représentant l'entrée et la sortie du programme ;*
- *les arêtes représentent les sauts de flot.*

Exemple 5. *Considérons le programme suivant :*

```
1  t0 = input()                    % entrée au clavier
2  if t0 mod 2 == 0:               %  si t0 est pair
3      print(t0 + "␣est␣pair")
4  else:
5      print(t0 + "␣est␣impair")
```

Celui-ci correspond à la syntaxe binaire suivante :

```
1  t0 = read_num
2  if t0 mod 2 == 0
3      print t0 + "␣est␣pair"
4      goto 6
5  print t0 + "␣est␣impair"
6  end program
```

Ce programme sera représenté par le CFG présenté en Figure 2.7.

2.3.2 La notion de prédicats opaques

La notion de prédicat

On s'intéresse tout d'abord à la notion de prédicat. On considérera une notion élargie : on désigne comme prédicat l'instruction qui permet de créer plusieurs chemins au sein d'un CFG.

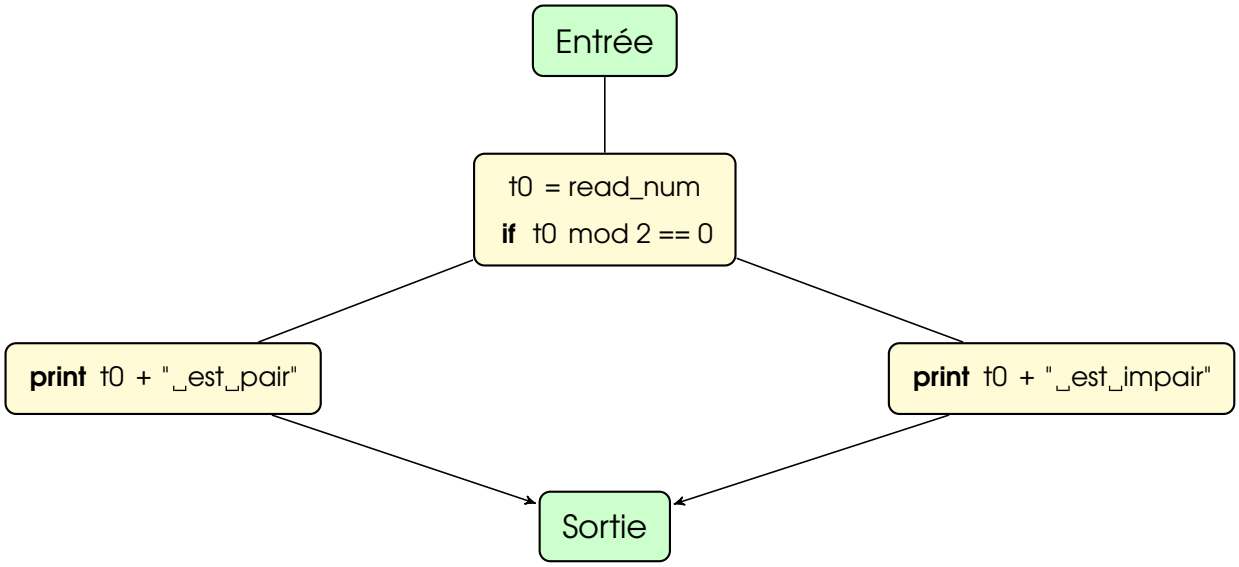


FIGURE 2.7 – CFG du code présenté en Exemple 5

Définition 2 (Prédicat)**.** *Un prédicat φ est un saut conditionnel au sein d'un code binaire.*
Celui-ci peut être évalué à $\top$ (vrai) ou à $\bot$ (faux).

Remarque. En pratique, on aura un prédicat dès lors que le CFG correspondant au code binaire présente deux sorties d'un bloc (*ie.* deux flèches qui sortent d'un même bloc).
Dans le cas général (*ie.* sauf cas particuliers de certains langages), on trouve des sauts conditionnels dans le binaire correspondant aux parties du programme incluant des boucles (conditionnelles ou itératives).

La notion de prédicat opaque

De façon intuitive, on peut alors définir un prédicat opaque comme un prédicat dont la construction permet une évaluation *difficile*.

Définition 3 (Fonction d'obfuscation)**.** *Une fonction d'obfuscation $\mathcal{O}$ renvoie, à partir d'un prédicat φ, un prédicat opaque $\mathcal{O}(\varphi)$. $\mathcal{O}$ doit vérifier les propriétés suivantes :*

furtivité (*stealth*) *c'est-à-dire que $\mathcal{O}(\varphi)$ doit être indistinguable d'un autre prédicat ;*

résistance (*resilience*) *c'est-à-dire que sa valeur ne doit pas être facilement connue.*

Remarque. On s'assure donc :

— qu'on ne peut pas (*facilement*) savoir si un prédicat est un prédicat opaque (furtivité) ;

— qu'on ne peut pas (*facilement*) trouver une version simple (la valeur d'origine) d'un prédicat opaque (résistance).

Prédicats variants et invariants

On peut déterminer deux types de prédicats, variants et invariants.

Définition 4 (Prédicats invariants (*Invariant opaque predicates*))**.** *Un prédicat opaque invariant est un prédicat φ qui est toujours évalué à $\top$ (vrai) ou à $\bot$ (faux).*

Formellement,

$$\exists v \in \{\top, \bot\}, \forall x, \mathcal{O}(\varphi)(x) = v$$

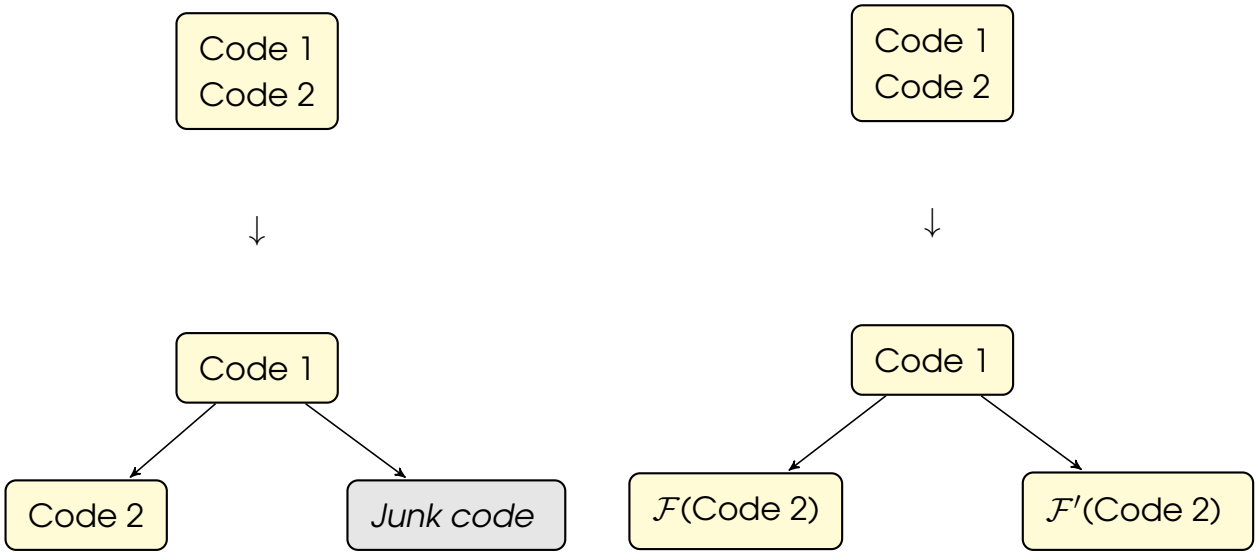
Définition 5 (Prédicats variants (*Two-ways opaque predicates*))**.** *Un prédicat opaque variant est un prédicat φ qui n'est pas invariant.*

Formellement,

$$\exists x_1, x_2, \mathcal{O}(\varphi)(x_1) = \top \textit{ et } \mathcal{O}(\varphi)(x_2) = \bot$$

Remarque. Cette définition est différente de la notion introduite dans la bibliographie [TS19, TSAEVL19] et est propre à ce document vis à vis de ces articles.

En fonction du type de prédicat choisit, la transformation du code sera différente. Les deux cas sont présentés au sein de la Figure 2.8.



(a) Cas d'un prédicat invariant (b) Cas d'un prédicat variant

FIGURE 2.8 – Transformations de code en fonction du type de prédicat

La Figue 2.8a présente le cas d'un prédicat invariant. Il n'y a alors qu'une seule branche qui est effectivement accessible, le reste du code y sera donc présent.

La Figure 2.8b présente le cas d'un prédicat variant. Cette fois, les deux branches sont accessibles. On notera alors la présence d'une fonction d'obfuscation $\mathcal{F}$ qui permet d'obtenir deux codes différents [7] de part et d'autre du prédicat : sinon, il suffirait de comparer de façon syntaxique les deux blocs d'instruction et obtenir un résultat immédiatement.

2.3.3 L'obfuscation de programmes

L'obfuscation, telle que définie dans la littérature [OBBM19, TS19, TSAEVL19, MLPQ19], a pour objectif de cacher le comportement d'un programme ou de protéger des informations importantes, comme des clés de cryptographie. Pour cela, un programme $\mathcal{P}$ est transformé en un programme $\mathcal{P}'$ tel que :

— $\mathcal{P}$ et $\mathcal{P}'$ soient sémantiquement équivalents ;
— $\mathcal{P}'$ est « aussi efficace » que $\mathcal{P}$;
— $\mathcal{P}'$ est *difficile* à comprendre.

On peut donc donner les définitions suivantes.

Définition 6 (Équivalence)**.** *On dit que deux programmes $\mathcal{P}$ et $\mathcal{P}'$ sont équivalents si, pour toute entrée et pour toute exécution, les résultats de $\mathcal{P}$ et $\mathcal{P}'$ sont identiques.*
On note alors $\mathcal{P} \equiv \mathcal{P}'$.

Définition 7 (Fonction d'obfuscation)**.** *Une fonction $\mathcal{F}$ est dite* d'obfuscation *si pour un programme $\mathcal{P}$,*

— *$\mathcal{F}(\mathcal{P}) \equiv \mathcal{P}$;*
— *$\mathcal{P}$ et $\mathcal{F}(\mathcal{P})$ sont de même complexité ;*
— *le problème définit par « Connaissant $\mathcal{F}(\mathcal{P})$, trouver $\mathcal{P}$ » est difficile.*

2.3.4 Méthodes d'obfuscation et de déobfuscation

Méthodes d'obfuscation

On s'intéresse maintenant aux méthodes qui permettent de construire de tels prédicats. Aujourd'hui, on peut distinguer cinq familles de constructions [TS19, OBBM19]. En pratique, l'idée est de créer des problèmes qui peuvent sembler évidents, mais difficilement résolubles par un ordinateur en phase d'analyse (et non d'exécution).

7. Ces deux codes doivent vérifier la propriété de codes obfusqués : avoir un comportement identique et une syntaxe différente.

Arithmétique (*Arithmetic-based*) Ce sont des formules mathématiques difficiles à résoudre :

$$7y^2 - 1 \neq x^2$$

Arithmétique et booléen (*Mixed-boolean and arithmetic based*) Ce sont des formules qui utilisent des propriétés booléennes [Eyr17] :

$$x + y \qquad \longrightarrow \qquad (x \vee y) + y - (\neg x \wedge y)$$

Alias (*Alias-based*) Ce sont des formules qui utilisent l'état d'un programme ou un emplacement mémoire : par exemple, en utilisant des pointeurs.

Environnement (*Environment-based*) Ce sont des formules qui utilisent des invariants du système ou des librairies :

```
1 if(strcpy(s2,s1)==s2)
```

Ici, par définition de `strcpy`, on trouvera toujours `s2` en sortie et donc une réponse positive.

Bi-opaque (*Bi-opaque*) Cette méthode, introduite récemment [XZK+18], permet de construire des prédicats en opposition à certaines méthodes d'analyse.

Méthodes de déobfuscation

Face à ces cinq méthodes pour construire des prédicats opaques, six méthodes sont usuellement utilisées pour permettre de déobfusquer un prédicat. Rappelons qu'au moment de l'analyse, il est impossible de savoir si un prédicat est opaque, variant ou invariant, ni quelle méthode a permis de le construire.

Analyse probabilistique (*Probabilistic check*)

Cette méthode consiste à évaluer un prédicat sur des valeurs aléatoires puis à conclure sur un cas d'invariant (si possible).

Cette méthode cause un grand nombre de cas faux-positifs [8] : elle est totalement inutilisable face à des prédicats invariants ou face à des prédicats qui ne sont pas basés sur des entrées de l'utilisateur.

8. Prédicat variant détecté invariant

Correspondance de modèles (*Pattern matching*)

Cette méthode consiste à observer la construction du prédicat et à la comparer à des prédicats connus. Face à de nombreux cas, cette méthode peut se révéler efficace : la littérature [EGV16] a montré qu'il n'existait qu'un faible nombre de prédicats qui sont utilisés dans la majorité des programmes.

Cependant, il serait très facile de contourner cette analyse en construisant des variantes ou de nouveaux modèles qui ne seront jamais détectés : cela implique un fort taux de cas faux-négatifs [9].

Interprétation abstraite (*Abstract interpretation*)

Cette méthode consiste en une attaque statique et sémantique du code. On cherche donc, statiquement, à comprendre sa sémantique. De façon générale, cette méthode n'est efficace que pour certaines classes de prédicats arithmétiques.

Preuve automatisée (*Automated proving*)

Cette méthode consiste à utiliser des solveurs SMT : ces outils permettent d'évaluer la valeur d'un prédicat (à vrai, à faux ou invariant).

Cependant, cette méthode est pénalisée (souvent) par les limites de l'analyse dynamique (notamment d'un point de vue sécurité et temps d'exécution) et récupère les limites des solveurs.

Synthèse de programme (*Program synthesis*)

Cette méthode consiste à synthétiser le code en chemins et à apprendre à partir de synthèses. Cependant, cette méthode est très focalisée sur certains modèles et ne peut pas être étendue.

Obfuscation et déobfuscation : limites de la littérature

Maintenant que les méthodes d'obfuscation et de déobfuscation ont été définie, il convient de s'interroger sur ce qu'il est possible d'analyser actuellement. Le Tableau 2.2 montre alors si une méthode permet de contrer un certain modèle de construction [TS19, TSAEVL19].

Il est important alors de noter que des cas considérés comme « traités »

9. Prédicats opaques détectés comme non-opaques

Constructions	Probabilistic	Pattern matching	Abstract interpretation	Automated proving	Program synthesis
Arithmetic-based	✓(*) [UDM05]	✗(*)	✓[DPMDBG06]	✓[BDM17]	✗
MBA-based	✗(*)	✓[EGV16]	✗	✗(A)	✓[BDM17]
Alias-based	✗(*)	✗	✗	✗(B)	✗
Concurrence-based	✗(*)	✗	✗	✗(B)	✗
Environment-based	✗(*)	✗	✗	✓[BDM17]	✗
Bi-opaque	✗(*)	✗(*)	✗	✗(*)	✗

TABLEAU 2.2 – Intersection des méthodes d'obfuscation et de déobfuscation

souffrent en réalité de mauvais taux de cas faux-positifs ou faux-négatifs. Ces cas sont indiqués par (*). D'autres sont limités par les solveurs SMT (A) ou par la méthode d'exécution symbolique (B).

Limites. Il est ainsi possible de mettre en évidence quatre limites principales, bien que toutes les méthodes ne sont pas concernées par chacune d'elles.

Spécificité *(Specificity)* Les techniques sont souvent concentrées sur des constructions spécifiques.

Couverture de code *(Code coverage)* La plupart des techniques sont basées sur une analyse dynamique : il est donc impossible de couvrir tous les chemins possibles.

Passage à l'échelle *(Scalability)* L'utilisation d'exécution dynamique empêche un passage à une large échelle : ces exécutions sont bien plus longues qu'une analyse statique.

Utilisation de solveurs L'utilisation de solveurs implique que l'analyse récupère les limites qu'ils possèdent, que ce soit les temps d'analyse, les faux-positifs ou les faux-négatifs.

Objectif. On se fixe donc comme objectif d'obtenir une méthode basée au maximum sur de l'analyse statique, n'exclure aucun modèle de construction et de n'avoir recours aux solveurs que dans de rares cas.

2.4 Méthode d'analyse proposée

2.4.1 Objectifs

On se donne donc comme objectif de proposer une méthode permettant la deobfuscation de prédicats opaques qui n'aurait pas les limites mises en évidences précédemment (Section 2.3.4), c'est-à-dire que notre outil :

— ne doit pas être spécifié, c'est-à-dire qu'il ne doit pas se concentrer sur certaines constructions ;
— doit permettre une couverture totale du code ;
— doit permettre un passage à l'échelle aisé ;
— doit limiter l'utilisation de solveurs.

En pratique, on se fixera donc comme objectif d'obtenir un outil basé uniquement sur de l'analyse statique en limitant l'utilisation de solveurs (mais sans interdire leur recours).

On propose donc un schéma avec deux phases de détection, illustré en Figure 2.9, à l'aide de *machine learning* :

— pour détecter si un prédicat est invariant ;
— pour détecter si un prédicat non invariant est un prédicat opaque.

Ce travail permet alors de réduire l'analyse des branches au minimum utile :

— soit la partie effectivement exécutée dans le cas d'un prédicat invariant ;
— soit une des deux branches (puisque équivalentes) dans le cas d'un prédicat (opaque) variant ;
— soit les deux branches, lorsqu'il ne s'agit pas d'un prédicat opaque.

On remarque aussi l'utilisation d'un solveur SMT uniquement afin de choisir la branche à analyser d'un prédicat invariant. Leur utilisation est donc limitée aux seuls cas où ils sont effectivement utiles.

La première étape étant déjà traitée dans la littérature [TSAEVL19, TS19], on s'intéresse ici à la deuxième étape de *machine learning* : détecter statiquement si un prédicat est un prédicat *(opaque)* variant. Notons également que la détection d'un prédicat variant a été réalisé dans la littérature [TSCEVL18] à partir d'une analyse sémantique, mais qui ne permet pas d'atteindre les objectifs fixés précédemment.

2.4.2 Étapes de l'analyse

Comme pour tout schéma de détection basé sur du *machine learning*, il est nécessaire de procéder à plusieurs étapes (voir Section 2.2.2 – pré-requis) :

— définition d'un modèle représentant les données ;
— définition d'une méthode d'extraction des « *features* » ;
— phase d'apprentissage (avec choix du modèle d'apprentissage).

Les données initiales seront des programmes rédigés en C++, compilés avec g++ dans sa dernière version et exécutables sur un environnement Unix (en pratique, Ubuntu). On obtiendra le processus d'extraction de *features* décrit en Figure 2.10.

Modèle de données : les *Control Flow Graph*

Afin de représenter les binaires de notre jeu de données, le modèle retenu est celui des CFG. On se base pour cette transformation sur l'outil Gorille, développé

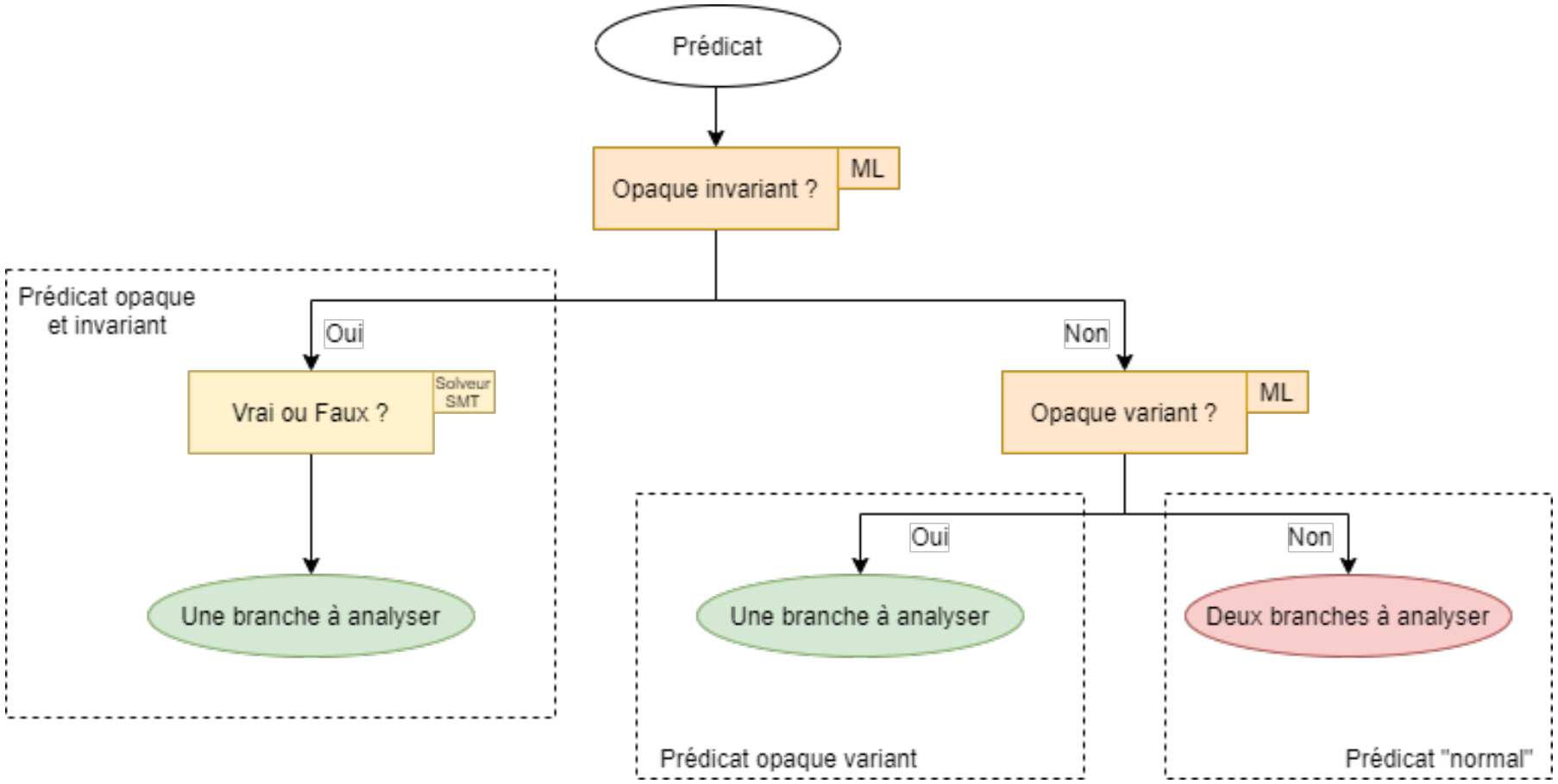


FIGURE 2.9 – Proposition de schéma de détection

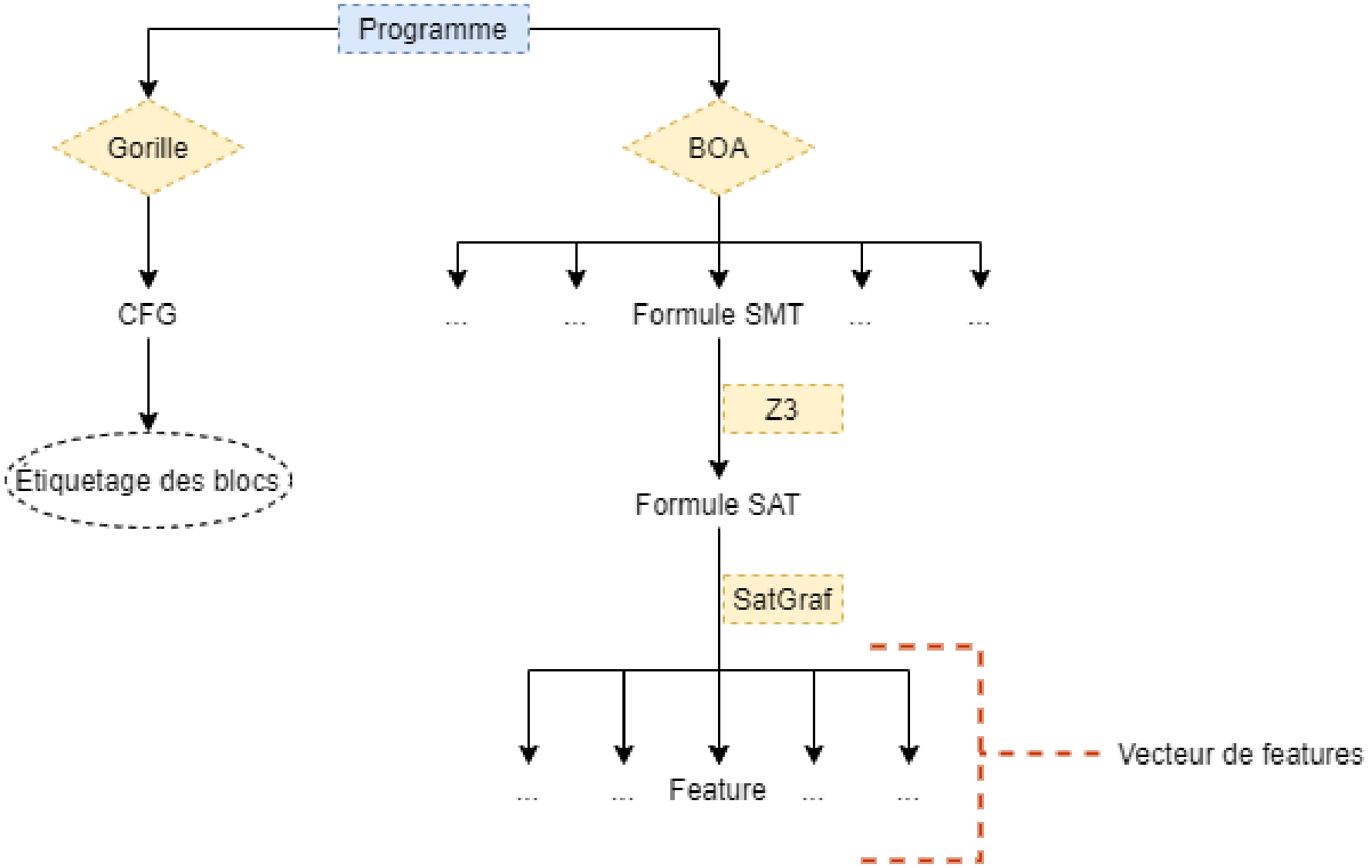


FIGURE 2.10 – Schéma représentant les étapes de l'extraction des *features*

par Cyber-detect [CD], une entreprise créée par des chercheurs du LORIA, dont Jean-Yves Marion. Pour des raisons de propriété industrielle, il est impossible ici de détailler les moyens mis en oeuvre afin de générer ces CFG.

Représentation des blocs : formules SMT

A l'aide de l'outil BOA [Cec], développé par Sylvain Cecchetto, chaque bloc d'instructions du CFG est transformé en une formule logique, exprimée en une syntaxe SMT. BOA étant un outil indépendant, il génère lui-même un CFG dont il extrait ces blocs.

Extraction des *features*

A partir des modèles extraits précédemment, il est nécessaire de définir des *features* (voir Section 2.2.2 du pré-requis). Malgré la diversité de la littérature sur le sujet [AAJ+20, MLPQ19, OS19], nous avons choisi de reprendre les modèles utilisés par Sebastian Banescu et Christian Collberg [BCP17].

Afin d'extraire des propriétés, leur travail se base sur une transformation des formules SMT :

— la formule logique, exprimée en SMT, est traduite en SAT, à l'aide de Microsoft's Z3 [dMB08] ;
— de la formule SAT obtenue, sont extraits des *features* à l'aide de SatGraf [NLG+15].

Notons alors que Banescu et Collberg [BCP17] ont également travaillé avec l'outil *Unified Code Counter* (UCC) [Ngu10] afin d'obtenir plus de propriétés statiques. Cet outil n'a pas été repris ici, leur travail ayant montré qu'en pratique ces *features* étaient moins pertinentes.

2.5 Pistes de validation

Rappel. Pour les raisons évoquées précédemment, la partie de validation n'a pas pu être menée. Cette partie vise donc seulement à présenter des pistes.

2.5.1 La validation croisée

Afin de limiter les aléas liés aux jeux de données [10], nous utilisons une méthode dite de « validation croisée ». Pour cela, le jeu complet est divisé en parts égales, qui serviront tour à tour de jeu de test.

La Figure 2.11 illustre le cas d'une validation croisée à cinq « *folds* » ; le jeu complet est donc divisé en cinq parties égales. Lors de la première expérience, la sous-partie 1 est utilisée comme test alors que les quatre autres permettent l'apprentissage. Pour l'expérience suivante, ce sera la sous-partie 2, etc.

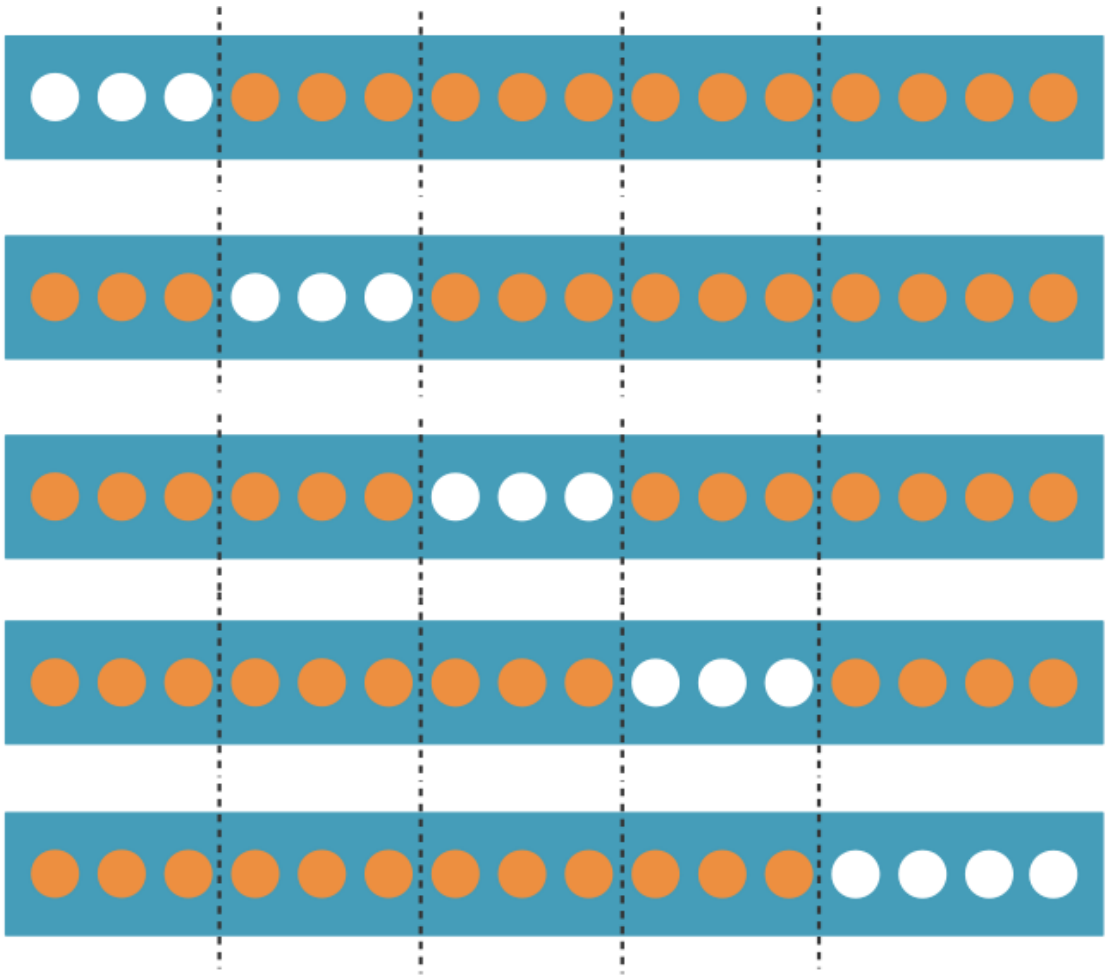


FIGURE 2.11 – Une validation croisée à 5 « folds », image d'après [Roo]

10. Tel que dans 2.2.2 avec un jeu qui ne dispose que de voitures et de vélos et qui ne permet pas de détecter des camions

2.5.2 Construction des datasets

La première étape pour mener les expériences de validation est la construction des jeux d'apprentissage et de test.

Programmes initiaux

Afin de pouvoir obtenir un jeu initial de programmes en C++, nous utilisons la base de données proposée par CodeForces [For]. Cette plate-forme propose des défis à relever à des programmeurs. Les réponses étant publiques, elles permettent d'obtenir une grande variété de programmes disponibles.

Obfuscation

Afin d'obfusquer de façon automatique et indépendante des programmes, nous utilisons l'outil Tigress [Tig] qui dispose d'une grande variété d'options. Pour le travail présenté ici (détection de prédicats opaques), sont seulement utilisées les options permettant une obfuscation par prédicats (principalement `AddOpaque` et `OpaquePredicate`).

2.6 Conclusion

La déobfuscation de programmes reste un enjeu majeur dans le domaine de la rétro-ingénierie. Alors qu'une recherche active propose de nouvelles méthodes d'obfuscation face au besoin croissant de protection intellectuelle et industrielle, cet outil est également utilisé afin de protéger des logiciels malveillants. Or, la littérature actuelle ne dispose pas de méthodes fiables et généralisables permettant de contourner ces protections.

Face à ce constat, la recherche autour des questions de méthodes de déobfuscation est apparue, notamment dans le contexte de la détection ou l'analyse de *malware*. Au cours de ce rapport, l'étude bibliographique a montré que ce problème est encore loin d'être résolu.

Afin d'obtenir un outil complet et utilisable dans un contexte à large échelle, un cadrage fort est défini : la méthode doit reposer sur une analyse statique, limiter le recours à des solveurs SMT et ne préférer aucun modèle de construction. Une telle méthodologie est proposée, mais au vu du contexte particulier, elle n'a pas

pu être validée à l'aide d'expériences.

Bibliographie

[AAJ+20] Hisham Alasmary, Ahmed A. Abusnaina, RhongHo Jang, M. Abuhamad, Afsah Anwar, DaeHun Nyang, and David Mohaisen. Soteria : Detecting adversarial examples in control flow graph-based malware classifiers. 2020.

[BCP17] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 661–678, Vancouver, BC, August 2017. USENIX Association.

[BDM17] S. Bardin, R. David, and J.-Y. Marion. Backward-Bounded DSE : Targeting Infeasibility Questions on Obfuscated Codes. In *2017 IEEE Symposium on Security and Privacy*, pages 633–651, San Jose, United States, May 2017. Institute of Electrical and Electronics Engineers Inc. Conference of 2017 IEEE Symposium on Security and Privacy, SP 2017 ; Conference Date : 22 May 2017 Through 24 May 2017 ; Conference Code :128430.

[CD] Cyber-Detect. `https://www.cyber-detect.com/index-fr.html`.

[Cec] Sylvain Cecchetto. Analyse du flot de données dans les codes binaires malveillants et applications à la détection de comportements malveillants. Thèse en préparation – `http://www.theses.fr/s187095`.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3 : an efficient smt solver. volume 4963, pages 337–340, 04 2008.

[DPMDBG06] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. Opaque predicates detection by abstract interpretation. pages 81–95, 07 2006.

[EGV16] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In ACM, editor, *2nd International Workshop on Software PROtection*, Vienna, Austria, October 2016.

[Eyr17] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools*. Theses, Université Paris-Saclay, June 2017.

[For] Code Forces. Problemset. `https://codeforces.com/problemset`.

[MLPQ19] Luca Massarelli, Giuseppe Luna, Fabio Petroni, and Leonardo Querzoni. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. 01 2019.

[Ngu10] Vu Nguyen. Improved size and effort estimation models for software maintenance. pages 1–2, 09 2010.

[NLG+15] Zack Newsham, William Lindsay, Vijay Ganesh, Jia Liang, Sebastian Fischmeister, and Krystof Czarnecki. Satgraf : Visualizing the evolution of sat formula structure in solvers. 09 2015.

[OBBM19] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to kill symbolic deobfuscation for free ; or unleashing the potential of path-oriented protections, 2019.

[OS19] Angelo Oliveira and Renato Sassi. Behavioral malware detection using deep graph convolutional neural networks, 11 2019.

[Roo] Open Class Room. Evaluez les performances d'un modele de machine learning. `https://openclassrooms.com/fr/courses/4297211-evaluez-les-performances-dun-modele-de-machine-learning/4308241-mettez-en-place-un-cadre-de-validation-croisee`.

[Tig] Tigress. Publications. Liste des publications : `https://tigress.wtf/publications.html`.

[TS19] Ramtine Tofighi-Shirazi. *Evaluation des méthodes d'obscurcissement de binaire*. PhD thesis, 2019. Thèse de doctorat dirigée par Philippe Elbaz-Vincent, Mathématiques et informatique Université Grenoble Alpes (ComUE) 2019.

[TSAEVL19] Ramtine Tofighi-Shirazi, Irina Mariuca Asavoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis. In *3rd International Workshop on Software PROtection*, London, United Kingdom, November 2019.

[TSCEVL18] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh-Ha Le. DoSE : Deobfuscation based on Semantic Equivalence. In *SSPREW-8*, San Juan, United States, December 2018.

[UDM05] S.K. Udupa, S.K. Debray, and Matias Madou. Deobfuscation : reverse engineering obfuscated code. volume 2005, pages 10 pp.–, 12 2005.

[XZK+18] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. Manufacturing resilient bi-opaque predicates against symbolic execution. pages 666–677, 06 2018.