Désassemblage et détection de logiciels malveillants auto-modifiants

Aurélien Thierry

11 mars 2015

Introduction

Objectif : détection de logiciels malveillants (programmes binaires)

Introduction

Objectif : détection de logiciels malveillants (programmes binaires)

Analyse morphologique (Bonfante, Kaczmarek, Marion, 2009) :

- Comparaison des graphes de flot de contrôle des programmes

Introduction

Objectif : détection de logiciels malveillants (programmes binaires)

Analyse morphologique (Bonfante, Kaczmarek, Marion, 2009) :

- Comparaison des graphes de flot de contrôle des programmes

Première étape :

- Désassemblage d'un programme
- Reconstruction d'un graphe de flot de contrôle

Introduction

Objectif : détection de logiciels malveillants (programmes binaires)

Analyse morphologique (Bonfante, Kaczmarek, Marion, 2009) :

- Comparaison des graphes de flot de contrôle des programmes

Première étape :

- Désassemblage d'un programme
- Reconstruction d'un graphe de flot de contrôle

Obscurcissement :

- Chevauchement de code
- Auto-modification

Introduction

Objectif : détection de logiciels malveillants (programmes binaires)

Analyse morphologique (Bonfante, Kaczmarek, Marion, 2009) :

- Comparaison des graphes de flot de contrôle des programmes

Première étape :

- Désassemblage d'un programme
- Reconstruction d'un graphe de flot de contrôle

Obscurcissement :

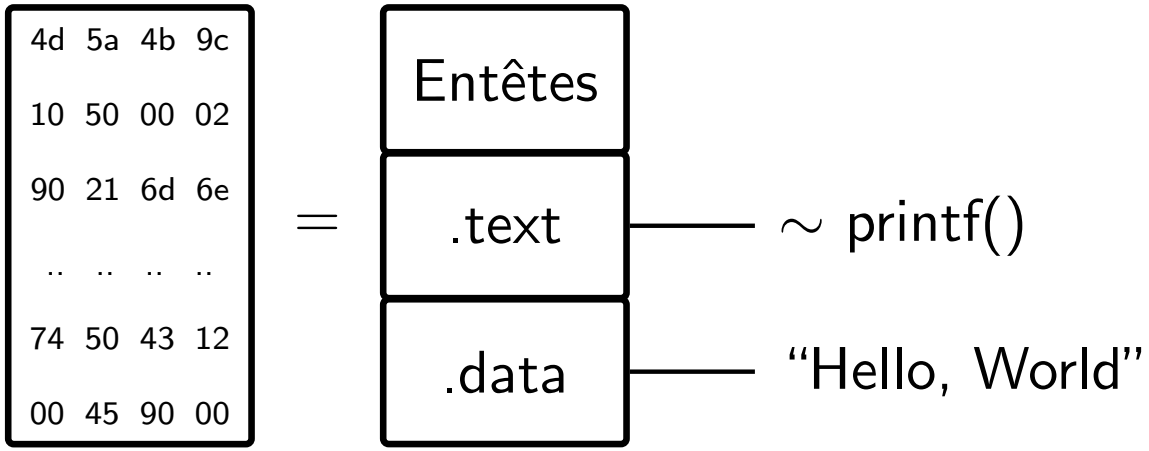
- Chevauchement de code
- Auto-modification

Seconde étape :

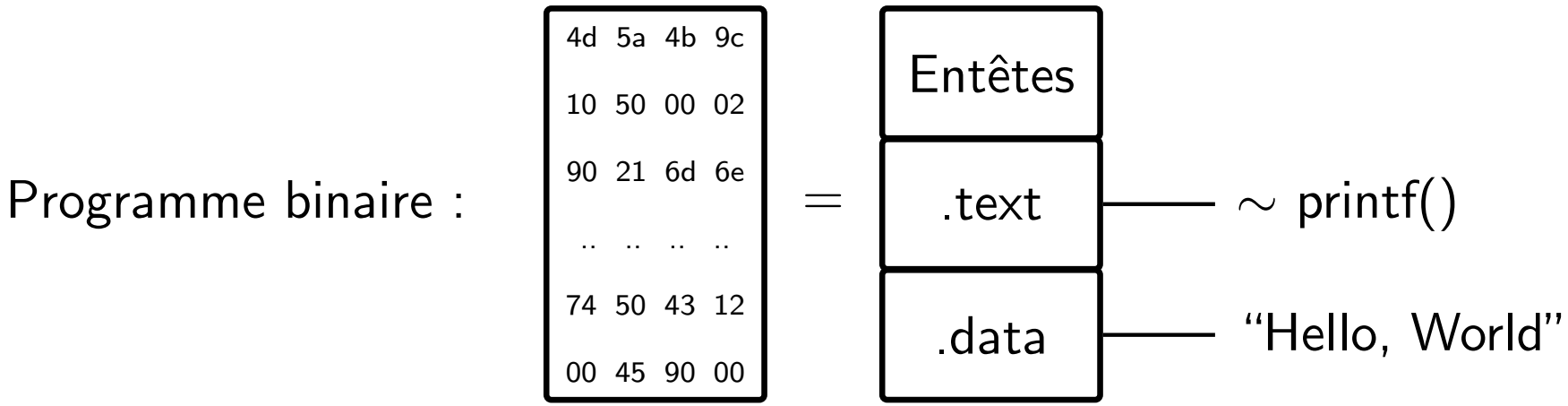
- Utilisation de l'analyse morphologique
- Formalisation et optimisation de la méthode
- Adaptation à la détection de similarités logicielles

Binaires et désassemblage

Programme binaire :



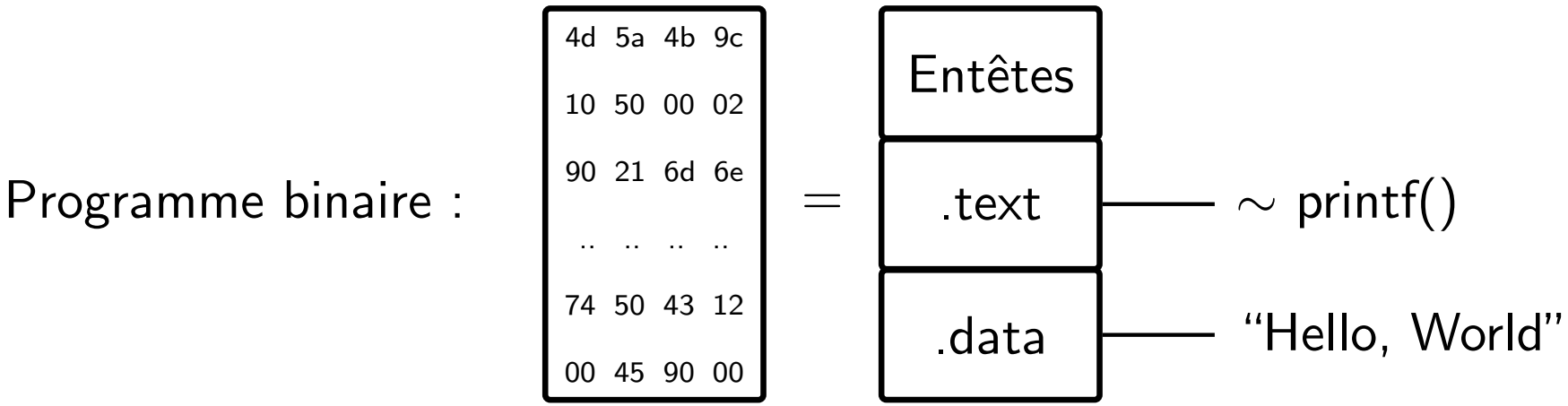
Binaires et désassemblage



Désassemblage :

- Programme binaire → code assembleur
- Code et données peuvent être présents dans les mêmes sections
- Difficile de séparer les données du code

Binaires et désassemblage



Désassemblage :

- Programme binaire → code assembleur
- Code et données peuvent être présents dans les mêmes sections
- Difficile de séparer les données du code

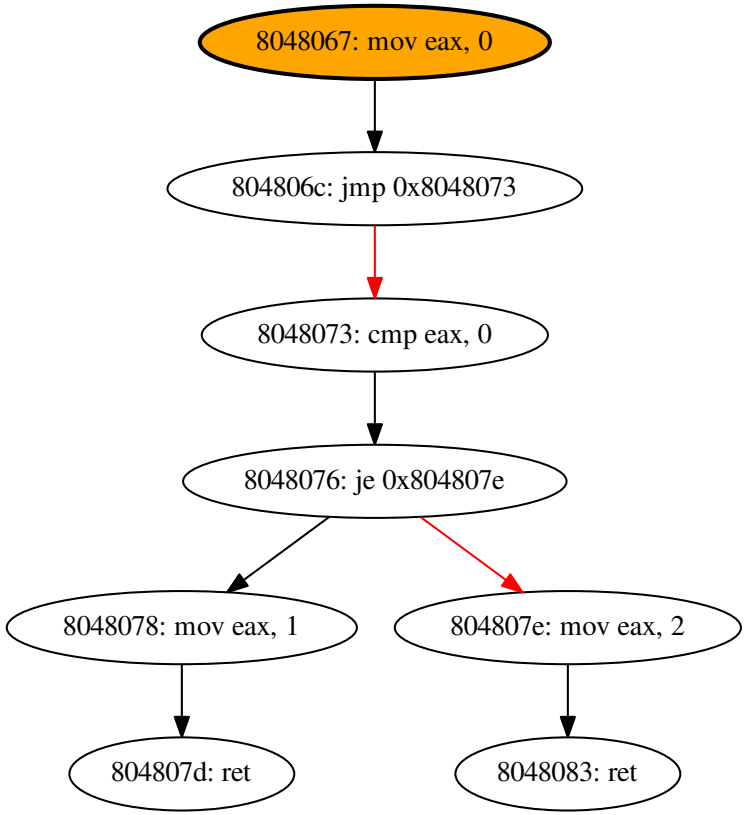
Définition

Le désassemblage parfait d'un programme binaire est la donnée de l'ensemble de ses instructions atteignables.

Il s'agit d'un problème indécidable.

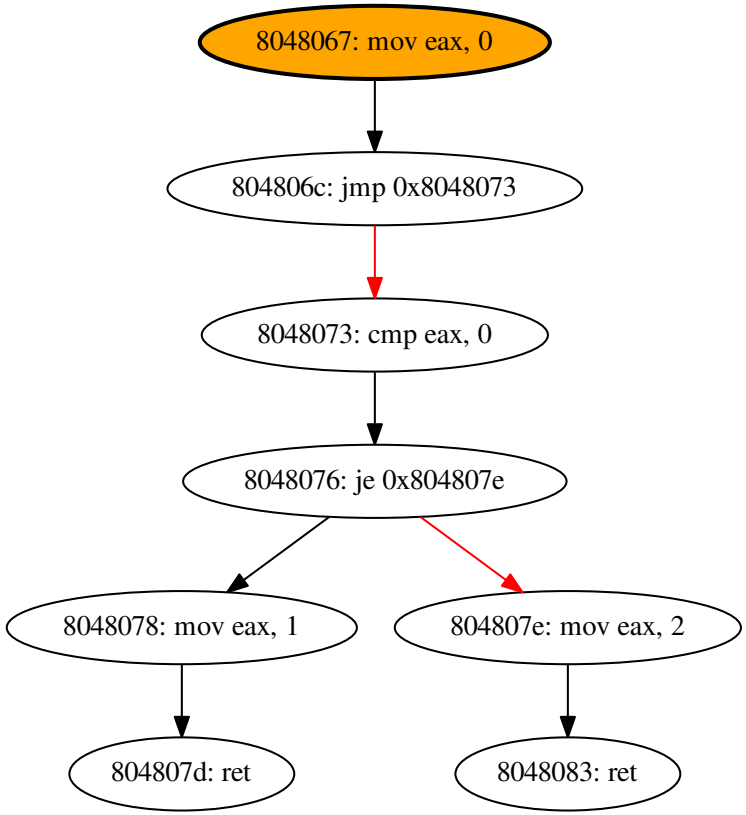
Désassemblage et graphe de flot de contrôle

Adresse	Octets	Instruction
8048067	b8 00 00 00 00	mov eax,0x0
804806c	eb 05	jmp 0x8048073
804806e	b8 03 00 00 00	mov eax,0x3
8048073	83 f8 00	cmp eax,0x0
8048076	74 06	je 0x804807e
8048078	b8 01 00 00 00	mov eax,0x1
804807d	c3	ret
804807e	b8 02 00 00 00	mov eax,0x2
8048083	c3	ret



Désassemblage et graphe de flot de contrôle

Adresse	Octets	Instruction
8048067	b8 00 00 00 00	mov eax,0x0
804806c	eb 05	jmp 0x8048073
804806e	b8 03 00 00 00	mov eax,0x3
8048073	83 f8 00	cmp eax,0x0
8048076	74 06	je 0x804807e
8048078	b8 01 00 00 00	mov eax,0x1
804807d	c3	ret
804807e	b8 02 00 00 00	mov eax,0x2
8048083	c3	ret



Désassemblage linéaire :

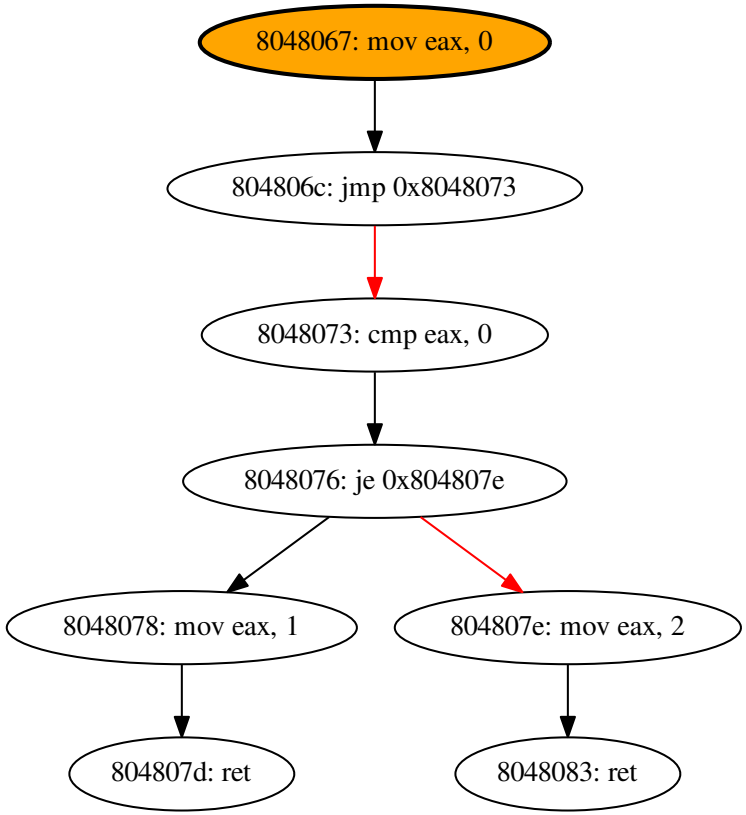
- Désassemblage adresse après adresse

Désassemblage récursif :

- Suit le flot de contrôle des instructions
- Jusqu'à un retour (`ret`) ou un saut dynamique (`jmp eax`)

Désassemblage et graphe de flot de contrôle

Adresse	Octets	Instruction
8048067	b8 00 00 00 00	mov eax,0x0
804806c	eb 05	jmp 0x8048073
804806e	b8 03 00 00 00	mov eax,0x3
8048073	83 f8 00	cmp eax,0x0
8048076	74 06	je 0x804807e
8048078	b8 01 00 00 00	mov eax,0x1
804807d	c3	ret
804807e	b8 02 00 00 00	mov eax,0x2
8048083	c3	ret

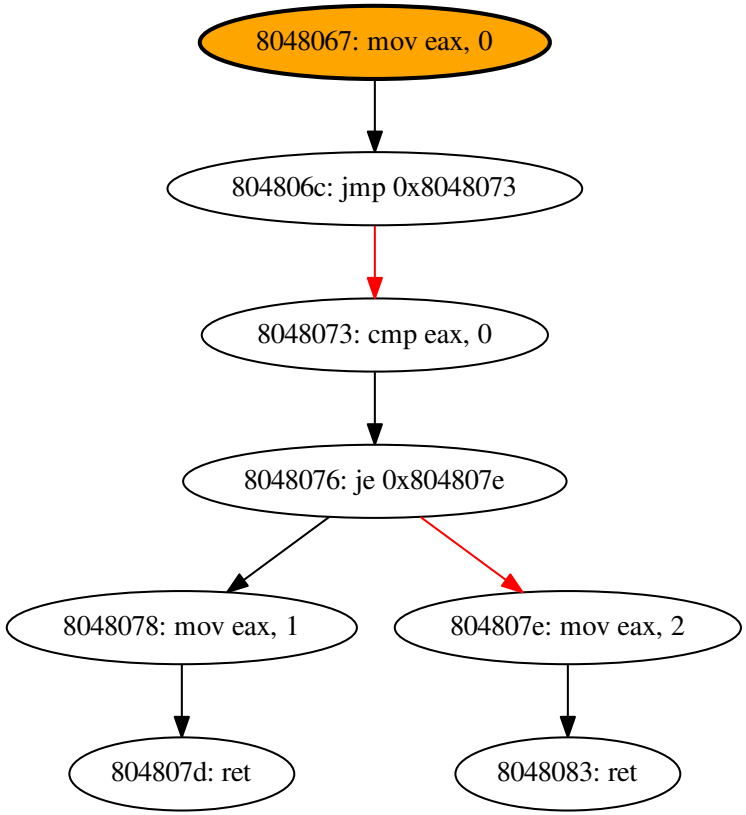


Graphe de flot de contrôle parfait (théorique) :

- On suppose qu'on dispose de toutes les exécutions possibles
- Sommets : ensemble des couples (adresse, instruction) atteignables
- Arc entre a et b ssi il existe un chemin d'exécution au sein duquel b suit immédiatement a

Désassemblage et graphe de flot de contrôle

Adresse	Octets	Instruction
8048067	b8 00 00 00 00	mov eax,0x0
804806c	eb 05	jmp 0x8048073
804806e	b8 03 00 00 00	mov eax,0x3
8048073	83 f8 00	cmp eax,0x0
8048076	74 06	je 0x804807e
8048078	b8 01 00 00 00	mov eax,0x1
804807d	c3	ret
804807e	b8 02 00 00 00	mov eax,0x2
8048083	c3	ret



Objectif : désassemblage et reconstruction automatique du GFC

Obscurcissement

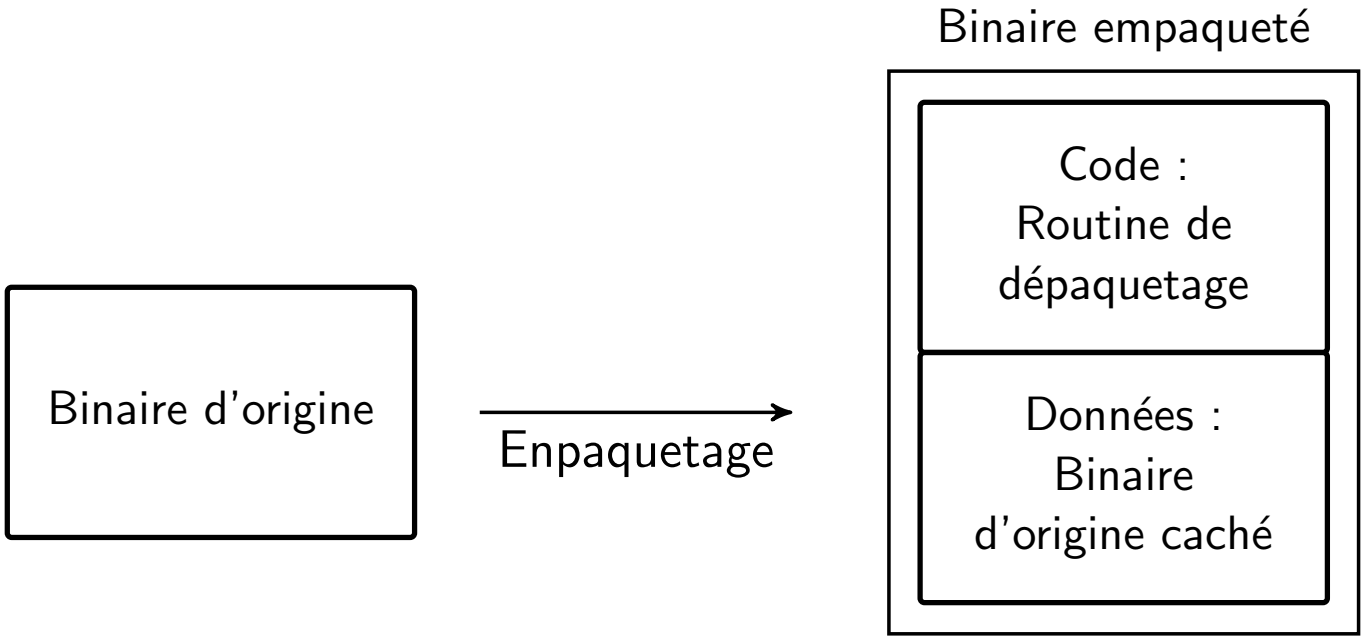
But de l'attaquant : rendre le désassemblage plus difficile

- Compression, chiffrement, ajout de code inutile...

Obscurcissement

But de l'attaquant : rendre le désassemblage plus difficile

- Compression, chiffrement, ajout de code inutile...
- En pratique appliqué par un logiciel d'empaquetage



tElock :

- Chevauchement de code, auto-modification

Chevauchement de code et analyse statique

Chevauchement de code

- Plusieurs instructions codées sur des adresses qui se chevauchent
- Assembleur x86 : taille des instructions variable (entre 1 et 15 octets)
- Les désassembleurs (IDA) font généralement l'hypothèse que ce n'est pas le cas

Exemple (tElock)

Octets à désassembler : `eb ff c9 7f e6`

```
01006e7d    eb ff          jmp +1
01006e7e       ff c9       dec ecx
01006e80    7f e6          jg 01006e68
```

Chevauchement de code

- Plusieurs instructions codées sur des adresses qui se chevauchent
- Assembleur x86 : taille des instructions variable (entre 1 et 15 octets)
- Les désassembleurs (IDA) font généralement l'hypothèse que ce n'est pas le cas

Exemple (tElock)

Octets à désassembler : `eb ff c9 7f e6`

```
01006e7d    eb ff          jmp +1
01006e7e       ff c9       dec ecx
01006e80    7f e6          jg 01006e68
```

- Possibilité de cacher une séquence de code de longueur arbitraire au milieu d'un autre code[1]

[1] Jämthagen, Lantz et Hell (2013)

Chevauchement de code : sémantique

Exemple (tElock)

Octets à désassembler : `eb ff c9 7f e6`

```
01006e7d    eb  ff          jmp +1
01006e7e        ff  c9      dec ecx
01006e80    7f  e6          jg 01006e68
```

- Séparation des instructions en couches de code

Adresses	01006e7d	01006e7e	01006e7f	01006e80	01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1		leave	jg 0x1006e68	
Couche 2		dec ecx			

Chevauchement de code : sémantique

Couche de code :

- Ensemble d'instructions qui ne se chevauchent pas.

Définition

Étant donné un ensemble d'instructions E, un découpage cohérent est un ensemble de couches

- *deux à deux disjointes,*
- *recouvrant l'ensemble des instructions E.*

Chevauchement de code : sémantique

Couche de code :

- Ensemble d'instructions qui ne se chevauchent pas.

Définition

Étant donné un ensemble d'instructions E, un découpage cohérent est un ensemble de couches

- *deux à deux disjointes,*
- *recouvrant l'ensemble des instructions E.*

Quelle stratégie pour construire un découpage cohérent en couches ?

- Couches linéaires
- Couches désassemblées par parcours récursif

Chevauchement de code : couches linéraires

- Une couche démarre à chaque adresse et consiste en un désassemblage linéaire
- Les couches inférieures redondantes sont supprimées

Adresses	01006e7d	01006e7e	01006e7f	01006e80	01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1		leave	jg 0x1006e68	
Couche 2		dec ecx		jg 0x1006e68	
Couche 3			leave	jg 0x1006e68	
Couche 4				jg 0x1006e68	
Couche 5					(invalide)

Chevauchement de code : couches linéraires

- Une couche démarre à chaque adresse et consiste en un désassemblage linéaire
- Les couches inférieures redondantes sont supprimées

Adresses	01006e7d	01006e7e	01006e7f	01006e80	01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1		leave	jg 0x1006e68	
Couche 2		dec ecx		jg 0x1006e68	

Chevauchement de code : couches linéraires

- Une couche démarre à chaque adresse et consiste en un désassemblage linéaire
- Les couches inférieures redondantes sont supprimées

Adresses	01006e7d	01006e7e	01006e7f	01006e80	01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1		leave	jg 0x1006e68	
Couche 2		dec ecx		jg 0x1006e68	

- La première couche correspond au désassemblage linéaire
- L'ensemble des couches est une généralisation du désassemblage linéaire

Couches désassemblées par parcours récursif

- On suit le parcours réalisé lors du désassemblage
- On ajoute chaque instruction désassemblée à la première couche existante pouvant la contenir, ou à une nouvelle couche

Adresses	01006e7d	01006e7e	01006e7f	01006e80	01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1			jg 0x1006e68	
Couche 2		dec ecx			

- Permet d'observer les difficultés liées au chevauchement de code lors du désassemblage récursif

Chevauchement de code : Caractérisation

Adresses	01006e7d	01006e7e	01006e7f	01006e80	01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1			jg 0x1006e68	
Couche 2		dec ecx			

Métriques :

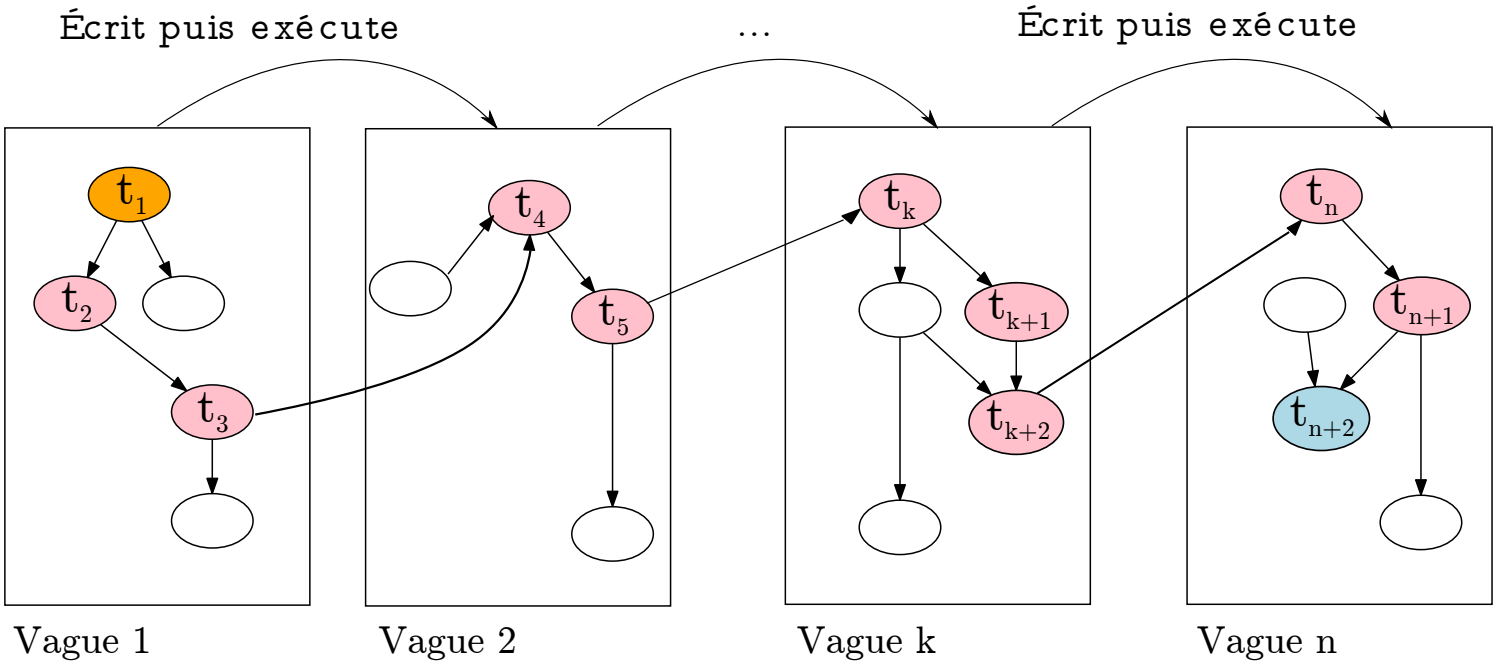
- Binaire non obscurci : désassemblage récursif constitué d'une seul couche
- Nombre de couches : indique la complexité des chevauchements
- Nombre de sauts de changement de couches : indique la fréquence d'utilisation de cette technique

Auto-modification et analyse dynamique

Auto-modification

Problème : certains programmes se modifient pendant l'exécution

- Découpage en niveaux d'exécution (Reynaud, Calvet)
- Utilisation d'une analyse dynamique (trace d'exécution)



- Instantané *i* : état de la mémoire au début de la vague *i*
- Au sein d'une vague, il n'y pas d'auto-modification

Auto-modification : graphe de flot de contrôle parfait

Sans auto-modification :

- Donnée : ensemble des traces possibles
- Sommet : (adresse, instruction)
- Arc : s'il existe une trace dans laquelle les instructions se suivent

Avec auto-modification :

- Données : ensemble des traces possibles et découpées en niveaux d'exécution
- Représenter les niveaux d'exécution dans le GFC ?

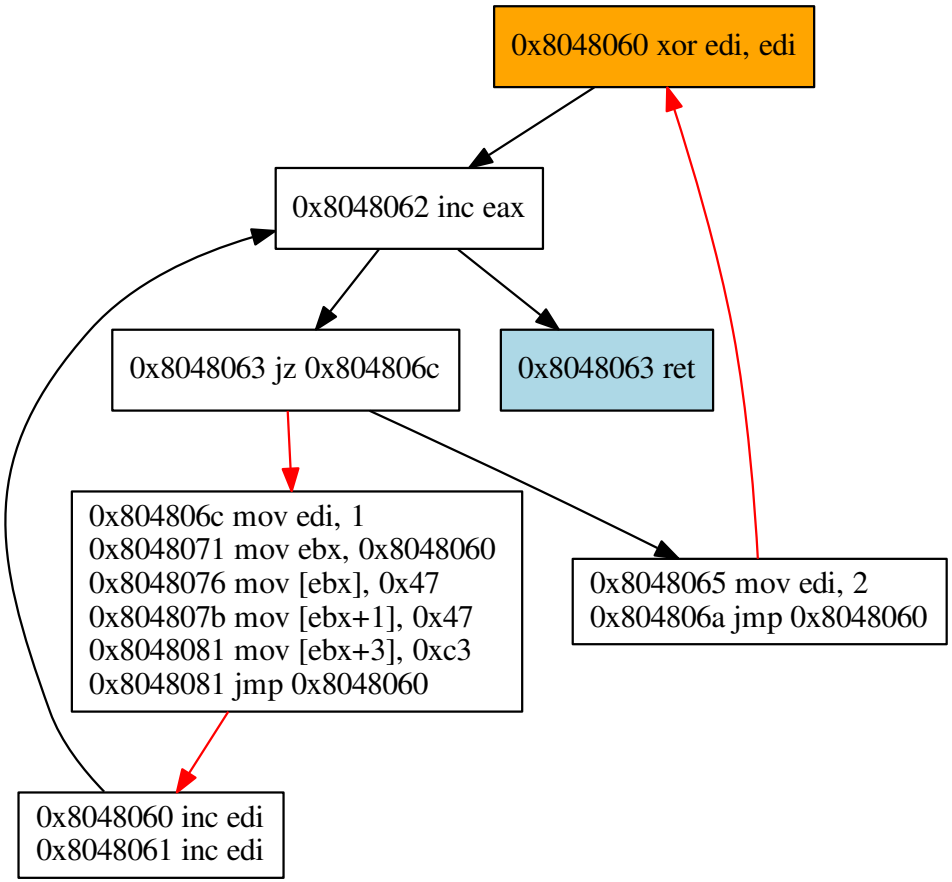
Auto-modification : exemple

Adresse	Octets	Instruction	Écriture
8048060 (debut)	31 ff	xor edi,edi	
8048062	40	inc eax	
8048063	74 07	je 804806c (si_zero)	
8048065	bf 02 00 00 00	mov edi, 0x2	
804806a	eb f4	jmp 8048060 (debut)	
804806c (si_zero)	bf 01 00 00 00	mov edi, 0x1	
8048071	bb 60 80 04 08	mov ebx, 0x8048060	
8048076	66 c7 03 47 00	mov [ebx], 0x47	inc edi
804807b	66 c7 43 01 47 00	mov [ebx+0x1], 0x47	inc edi
8048081	66 c7 43 03 c3 00	mov [ebx+0x3], 0xc3	ret
8048087	eb d7	jmp 8048060 (debut)	

- Valeur de `edi` lorsque `ret` est atteint ?

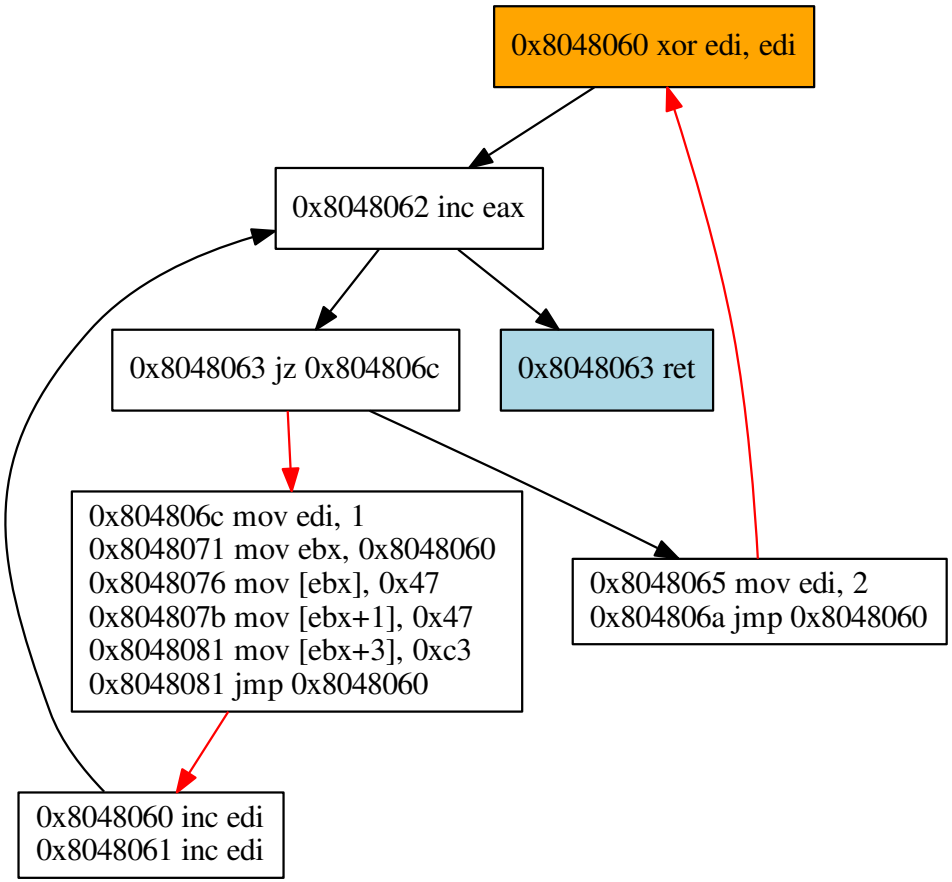
GFC ne traitant pas l'auto-modification

- Sommet : (adresse, instruction)



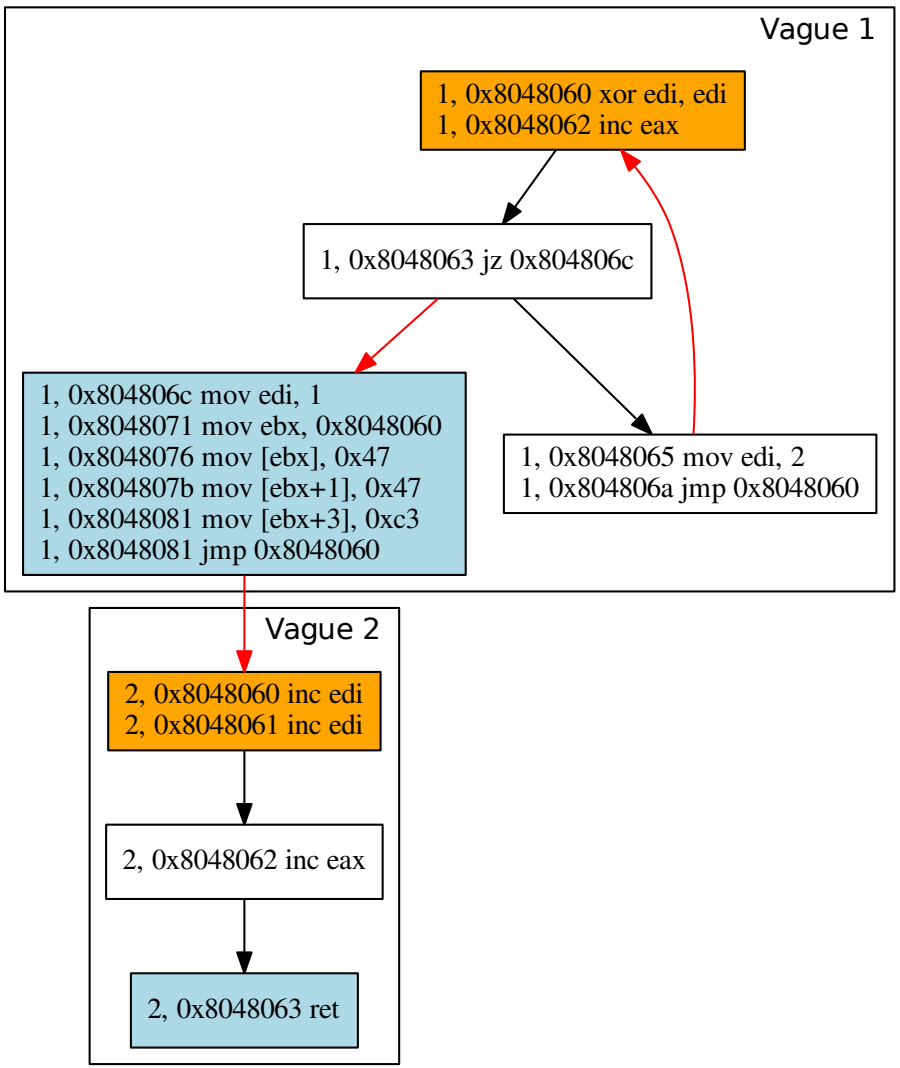
GFC ne traitant pas l'auto-modification

- Sommet : (adresse, instruction)



- Certains chemins sont impossibles
- On n'observe pas l'enchaînement des niveaux d'exécution

Auto-modification : graphe de flot de contrôle paramétré



- Sommet : (niveau d'exécution, adresse, instruction)

Graphe de flot de contrôle parfait

- On considère toutes les exécutions possibles
- Au sein d'une trace, un instantané de la mémoire est associé à chaque niveau d'exécution

Graphe de flot de contrôle parfait

Niveau d'exécution	1	2	3	4	5
Trace 1	s_1	s_2	s_3	s_4	
Trace 2	s_1	s_2	s_3	s_5	s_6
Trace 3	s_1	s_2	s_7	s_5	s_6
Trace 4	s_1	s_2	s_4		

Instantanés de la mémoire

Graphe de flot de contrôle parfait

Niveau d'exécution	1	2	3	4	5
Trace 1	s_1	s_2	s_3	s_4	
Trace 2	s_1	s_2	s_3	s_5	s_6
Trace 3	s_1	s_2	s_7	s_5	s_6
Trace 4	s_1	s_2	s_4		

Instantanés de la mémoire

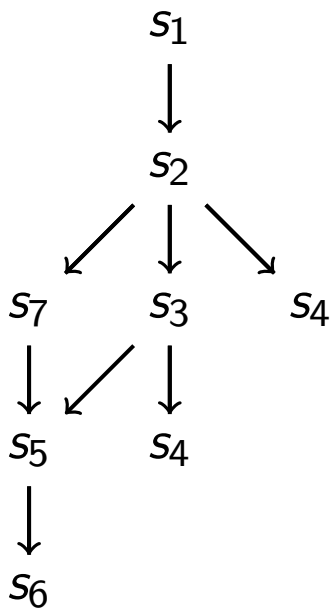
Niveau d'exécution (X) 1

Niveau d'exécution (X) 2

Niveau d'exécution (X) 3

Niveau d'exécution (X) 4

Niveau d'exécution (X) 5

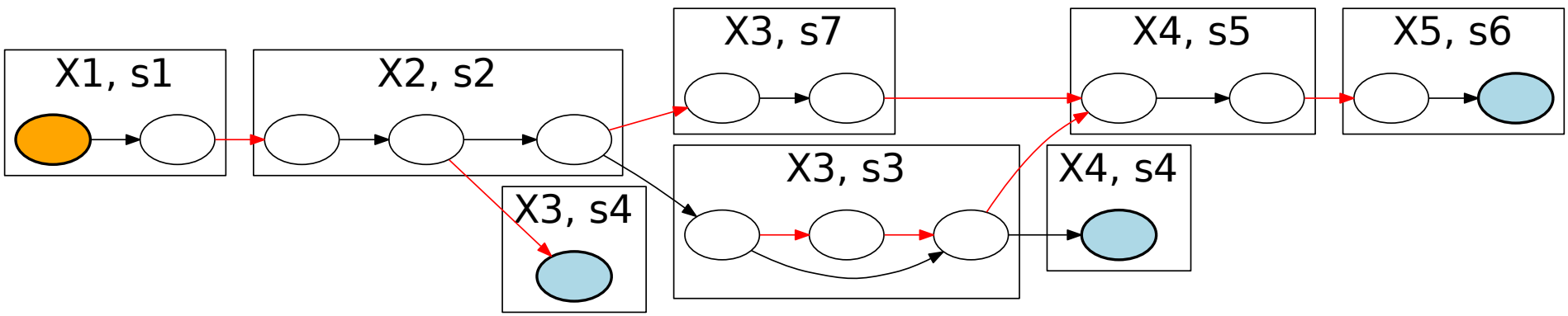


Graphe de flot de contrôle parfait

Niveau d'exécution	1	2	3	4	5
Trace 1	s_1	s_2	s_3	s_4	
Trace 2	s_1	s_2	s_3	s_5	s_6
Trace 3	s_1	s_2	s_7	s_5	s_6
Trace 4	s_1	s_2	s_4		

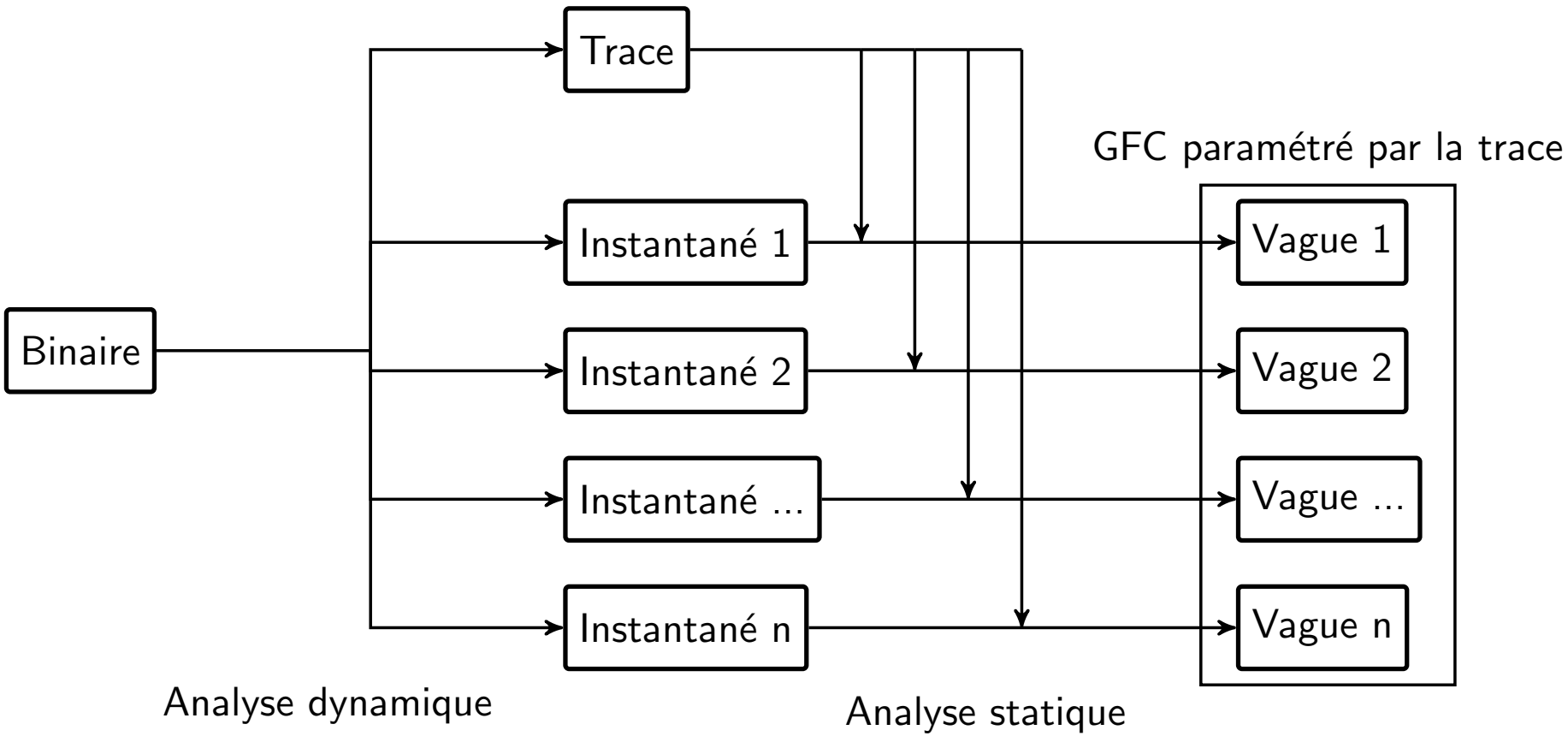
Instantanés de la mémoire

- Sommet : (niveau d'exécution, instantané, adresse, instruction)



Désassemblage hybride

Analyse dynamique puis statique par parcours récursif



Exemple sur `hostname.exe` obscurci

- Utilisation d'un binaire (`hostname.exe`) et de 34 versions obscurcies

Exemple sur `hostname.exe` obscurci

- Utilisation d'un binaire (`hostname.exe`) et de 34 versions obscurcies

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches	
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas
(aucun)	/	154	354	/	/	/	/	/	/
	0	154	354	0	0	1	1	0	0
UPX	/	322	523	/	/	/	/	/	/
	0	168	169	4	4	2	2	2	2
	1	154	354	0	0	1	1	0	0
tElock99	/	2044	3022	/	/	/	/	/	/
	0	46	130	0	42	1	3	0	16
					...				
	2	19	20	2	2	2	2	2	2
	3	25	25	8	8	2	2	8	8
					...				
	17	226	430	0	0	1	1	0	0

- 28/34 binaires ont atteint la charge finale (`hostname.exe`)
- 27/28 sont auto-modifiants : de 2 à 79 vagues

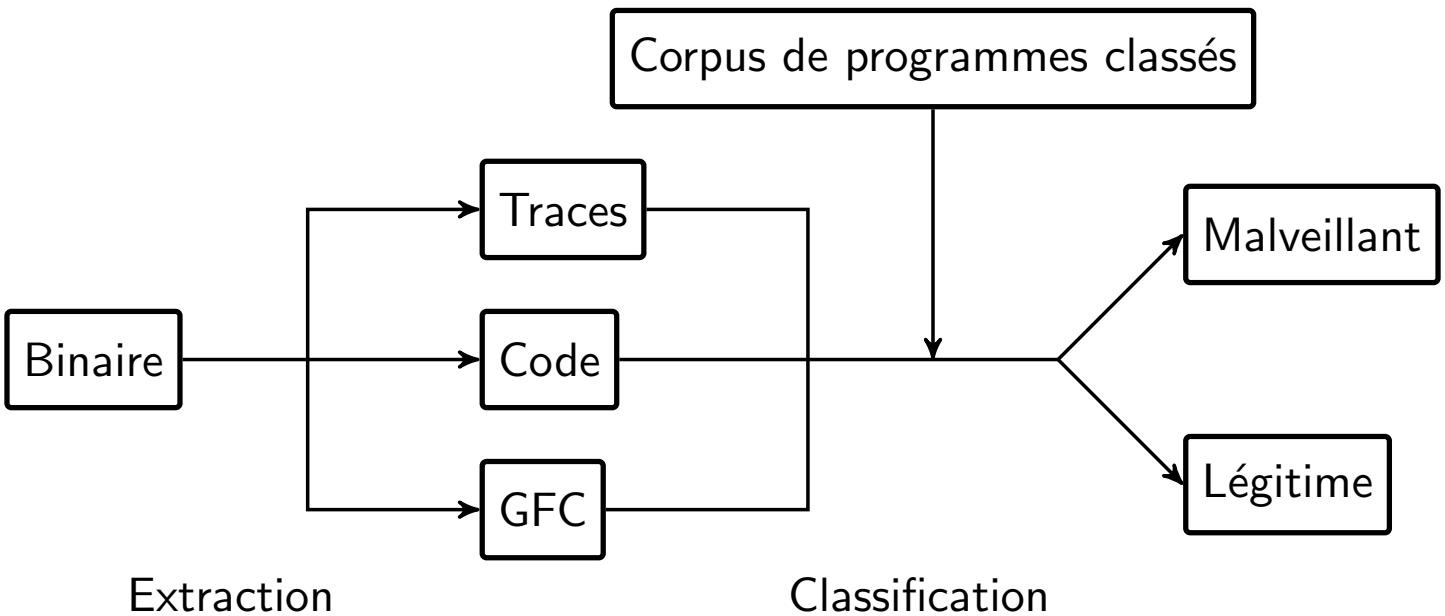
Exemple sur `hostname.exe` obscurci

- 22/28 ont plusieurs couches de code dans le désassemblage
  - de 2 à 4 couches
  - de nombreux sauts entre couches (jusqu'à 1 817 pour jdpack)
- 9/28 (tElock, UPX, pespin) exécutent réellement des instructions en chevauchement
  - uniquement 2 couches
  - jusqu'à 12 sauts entre couches

Analyse morphologique et algorithmes de comparaison de graphes

Analyse morphologique

- Comparaison des graphes de flot de contrôle
- Détection par signatures



- Les signatures sont des graphes de flot de contrôle
- On cherche des isomorphismes de sous-graphes

Isomorphisme appliqué aux graphes de flot

Les graphes sont réduits :

- But : rendre les signatures plus génériques
- Instructions : `call eax` $\rightarrow$ `CALL`
- Regroupement des instructions séquentielles : bloc $\rightarrow$ `INST`

Isomorphisme appliqué aux graphes de flot

Les graphes sont réduits :

- But : rendre les signatures plus génériques
- Instructions : `call eax` → `CALL`
- Regroupement des instructions séquentielles : bloc → `INST`

Isomorphisme de sous-graphes :

- Appliqué entre des graphes de flot extraits d'un malware (signatures) et le graphe à analyser
- Problème NP complet
- Algorithme classique (Ullmann[2]) trop lent

Objectif : définir un sous-problème → solutions en temps polynomial ?

[2] J.R. Ullmann, "An algorithm for Subgraph Isomorphism" (1976)

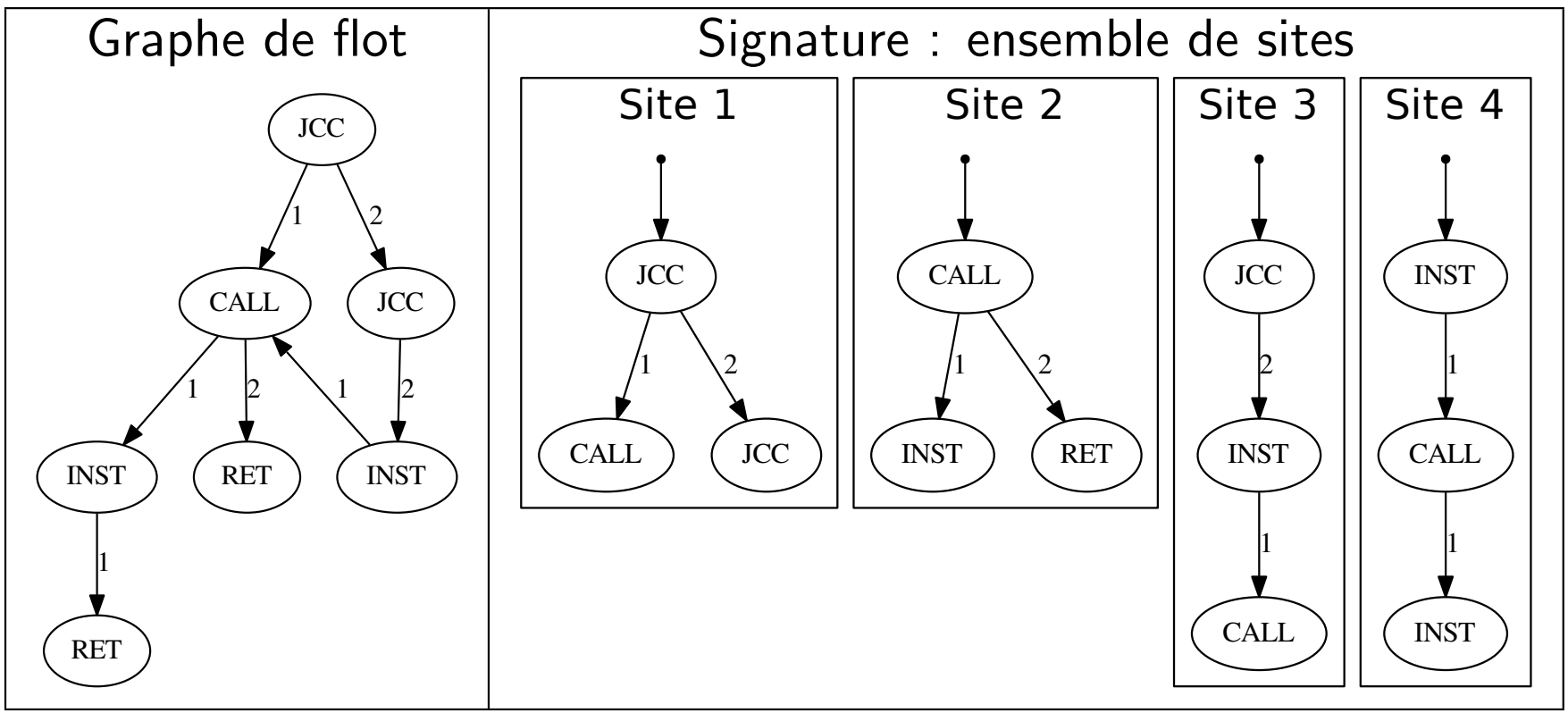
Simplification : graphes de flot et sites

Graphe de flot :

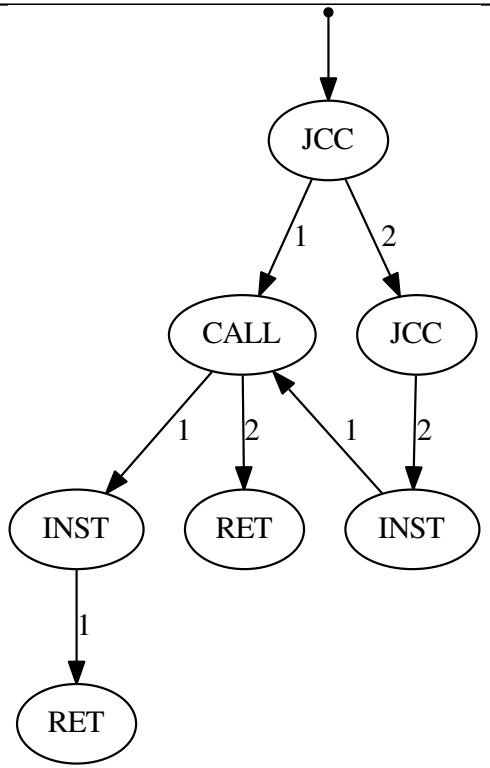
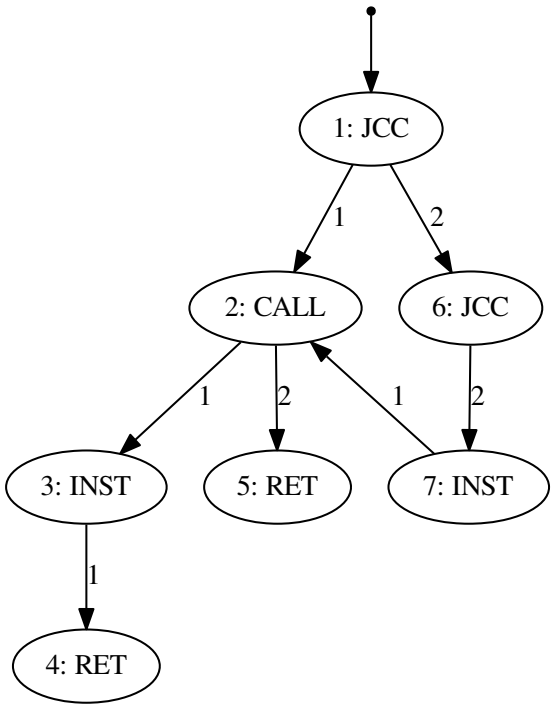
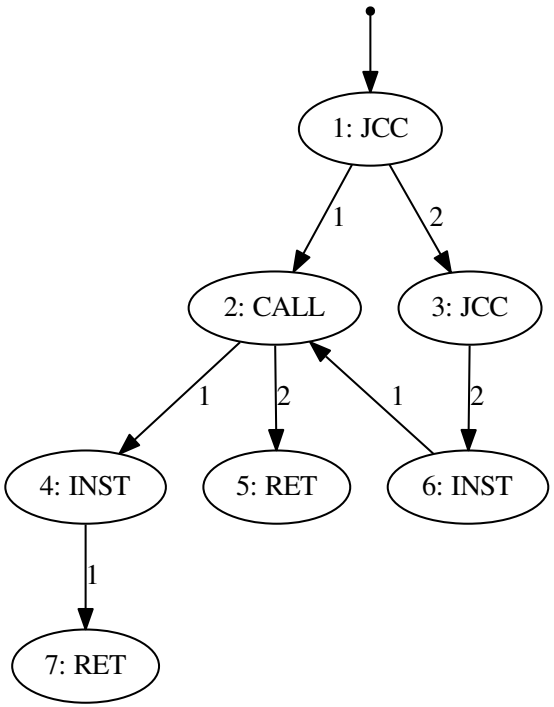
- Le nombre de fils est borné (par 2)
- Les fils sont ordonnés

On génère des sous-graphes (sites) par parcours en largeur de taille fixe

- Site : graphe de flot avec racine

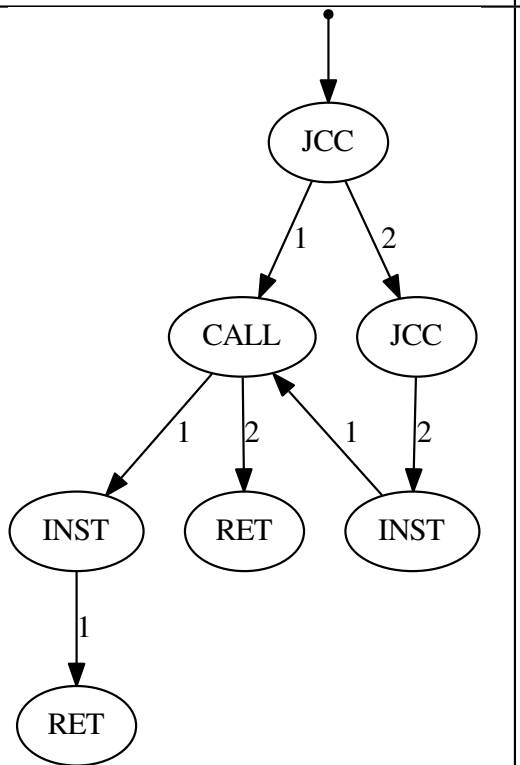


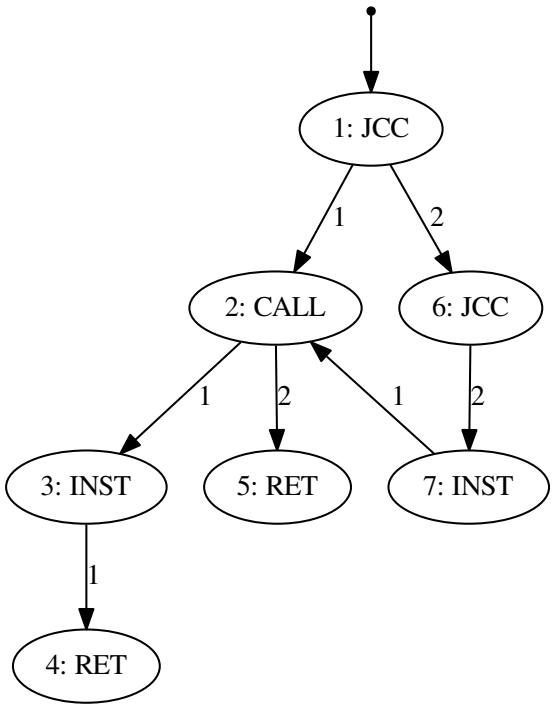
Parcours d'un site

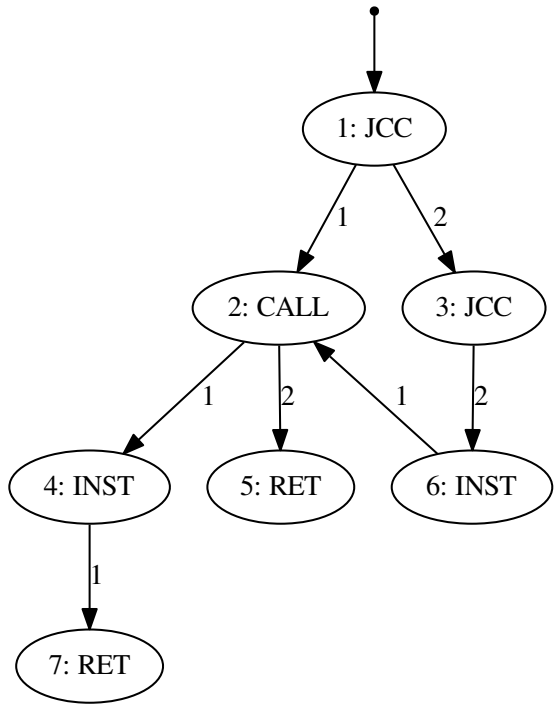
Site	Numérotation en profondeur	Numérotation en largeur
		

Parcours d'un site

Site	Numérotation en profondeur	Numérotation en largeur





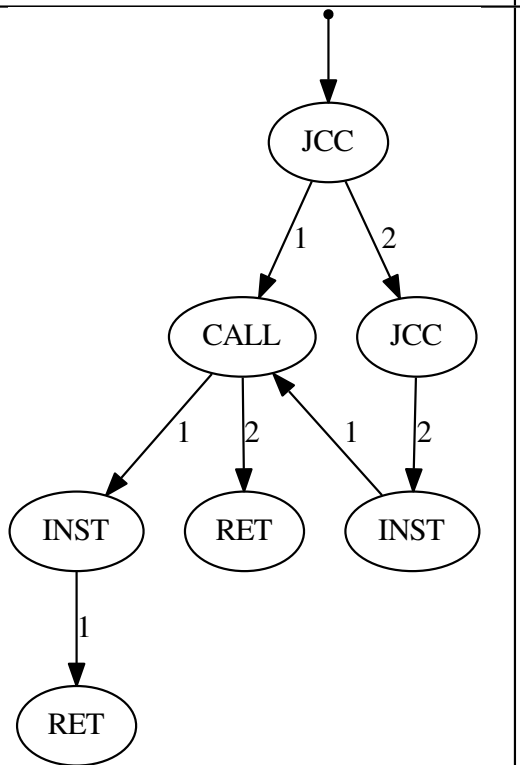


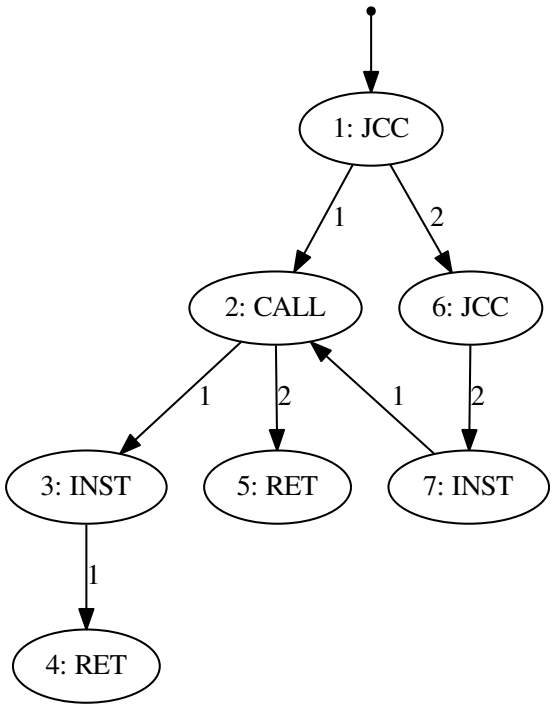
Codage de parcours en profondeur :

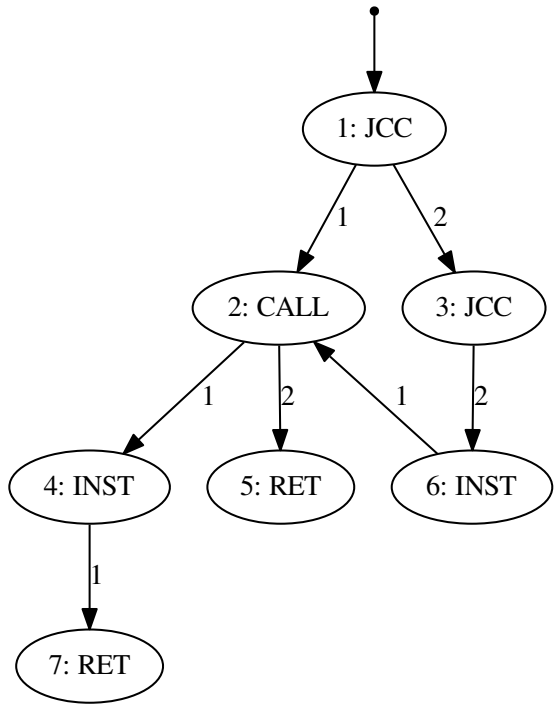
$1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{1} 3 : INST \xrightarrow{1} 4 : RET \xrightarrow{R} 2 \xrightarrow{2} 5 : RET \xrightarrow{R} 1 \xrightarrow{2} 6 : JCC \xrightarrow{2} 7 : INST \xrightarrow{1} 2.$

Parcours d'un site

Site	Numérotation en profondeur	Numérotation en largeur







Codage de parcours en largeur :

$1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{R} 1 \xrightarrow{2} 3 : JCC \xrightarrow{R} 2 \xrightarrow{1} 4 : INST \xrightarrow{R} 2 \xrightarrow{2} 5 :$
$RET \xrightarrow{R} 3 \xrightarrow{2} 6 : INST \xrightarrow{R} 4 \xrightarrow{1} 7 : RET.$

Codage de parcours

Codage de parcours en largeur :
$1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{R} 1 \xrightarrow{2} 3 : JCC \xrightarrow{R} 2 \xrightarrow{1} 4 : INST \xrightarrow{R} 2 \xrightarrow{2} 5 :$
$RET \xrightarrow{R} 3 \xrightarrow{2} 6 : INST \xrightarrow{R} 4 \xrightarrow{1} 7 : RET.$

Définition : Mot de parcours

Un mot de parcours est soit :

- de type m_1, c'est à dire $1 : L$ où L est une étiquette, ou
- de type m_2, c'est à dire $\xrightarrow{\alpha} i[: L]$ où i est un entier, L est une étiquette et α est soit un entier k, soit R.

Un codage de parcours est de la forme $m_1(m_2)^*$

Codage de parcours

Définition : Codage de parcours bien formé

Un codage de parcours est dit bien formé s'il vérifie :

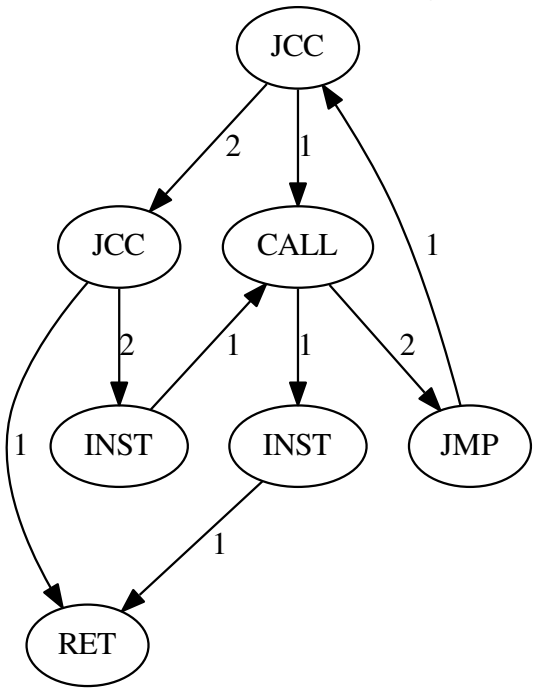
1. L'étiquette de chaque sommet i est spécifiée une unique fois, lors la première référence à ce sommet
2. Pour tout mot de la forme $\xrightarrow{R} i$, un mot $i : L$ est présent précédemment dans le codage
3. Chaque fils de chaque sommet est défini au plus une fois

- Les algorithmes fournissant les parcours en profondeur et en largeur sont simples
- Ils produisent des codages de parcours bien formés
- On peut reconstruire le site d'origine à partir du codage de parcours bien formé

Parcours d'un codage au sein d'un graphe de flot

Le parcours d'un codage bien formé est-il possible dans un graphe de flot ?

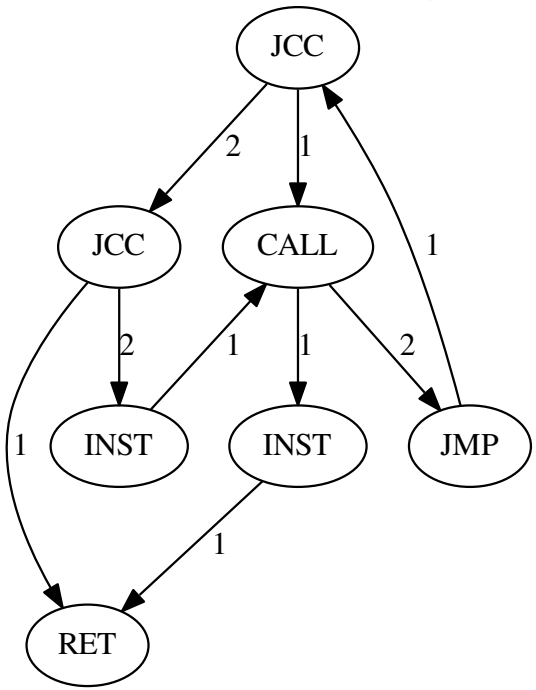
$$1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{R} 1 \xrightarrow{2} 3 : JCC$$



Parcours d'un codage au sein d'un graphe de flot

Le parcours d'un codage bien formé est-il possible dans un graphe de flot ?

$1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{R} 1 \xrightarrow{2} 3 : JCC$



- On part de chaque sommet du graphe à analyser
- On cherche à réaliser le parcours à partir de ce sommet
- Le codage est traité mot après mot

Codages et isomorphisme de sous-graphes

Lien entre isomorphisme de sous-graphes et parcours dans un graphe de flot ?

Codages et isomorphisme de sous-graphes

Lien entre isomorphisme de sous-graphes et parcours dans un graphe de flot ?

Théorème

Soit P un site et T un graphe de flot. Les propositions suivantes sont équivalentes.

- *P est isomorphe à un sous-site de T.*
- *Tout codage de parcours bien formé de P peut être parcouru dans T.*
- *Il existe un codage de parcours bien formé de P qui peut être parcouru dans T.*

Codages et isomorphisme de sous-graphes

Lien entre isomorphisme de sous-graphes et parcours dans un graphe de flot ?

Théorème

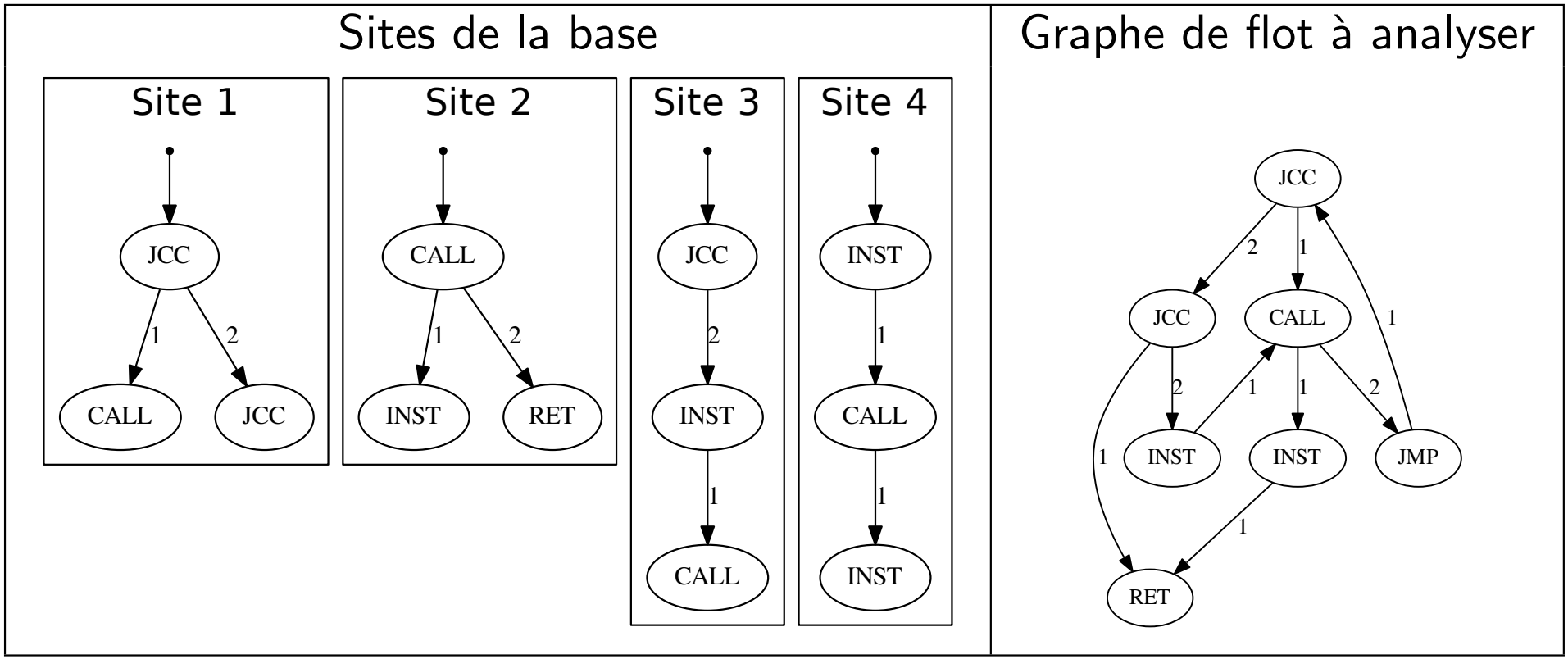
Soit P un site et T un graphe de flot. Les propositions suivantes sont équivalentes.

- *P est isomorphe à un sous-site de T.*
- *Tout codage de parcours bien formé de P peut être parcouru dans T.*
- *Il existe un codage de parcours bien formé de P qui peut être parcouru dans T.*

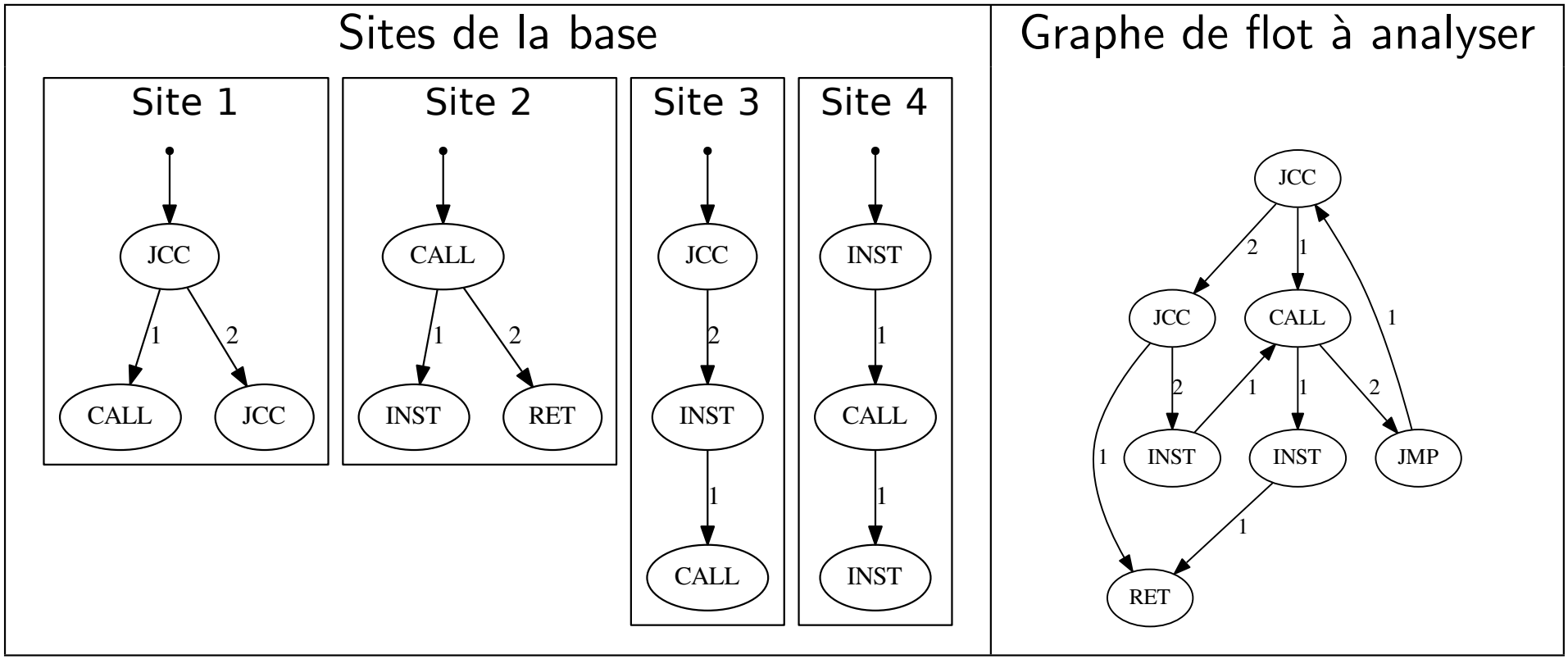
Conséquence : équivalence entre

- P est isomorphe à un sous-site de T
- Le codage de parcours en profondeur de P peut être parcouru dans T
- Le codage de parcours en largeur de P peut être parcouru dans T

Détection d'isomorphismes par parcours de codage



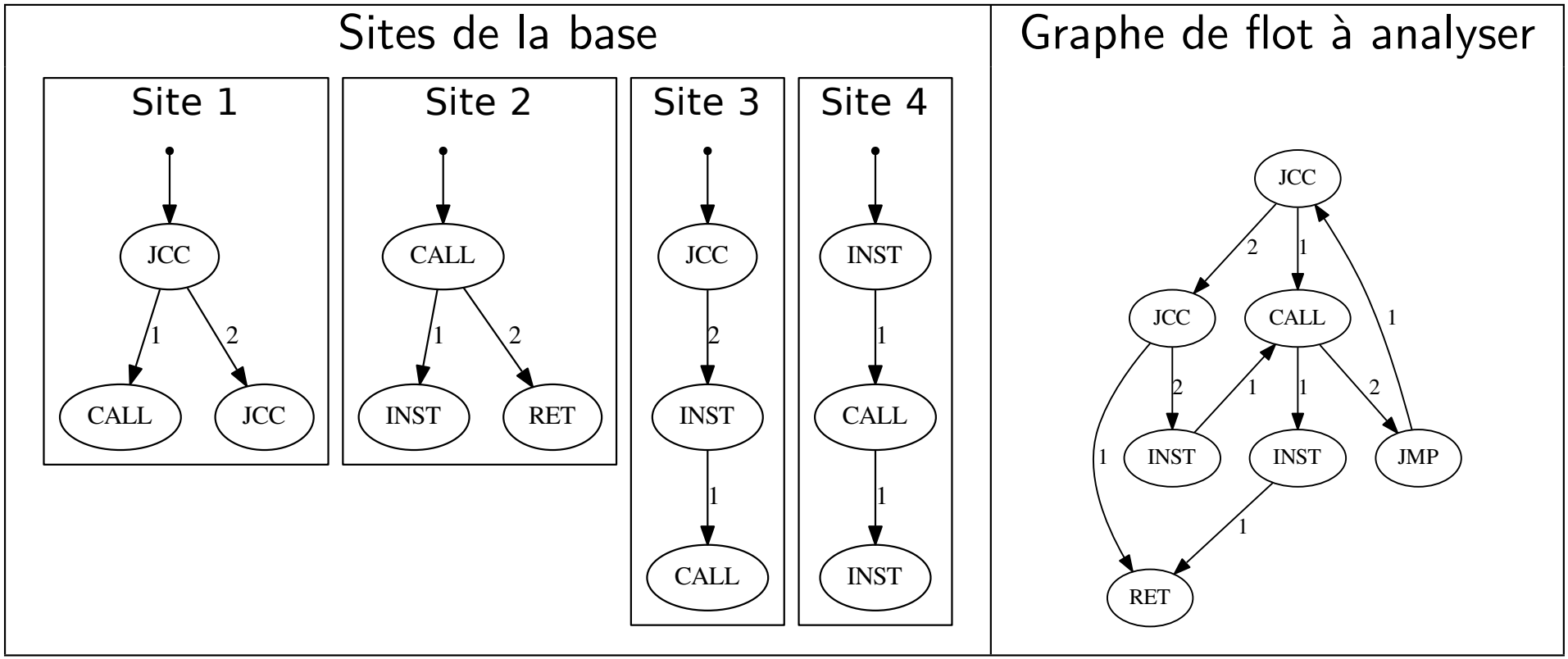
Détection d'isomorphismes par parcours de codage



Parcours en profondeur :

- $1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{R} 1 \xrightarrow{2} 3 : JCC$
- $1 : CALL \xrightarrow{1} 2 : INST \xrightarrow{R} 1 \xrightarrow{2} 3 : RET$
- $1 : JCC \xrightarrow{2} 2 : INST \xrightarrow{1} 3 : CALL$
- $1 : INST \xrightarrow{1} 2 : CALL \xrightarrow{1} 3 : INST$

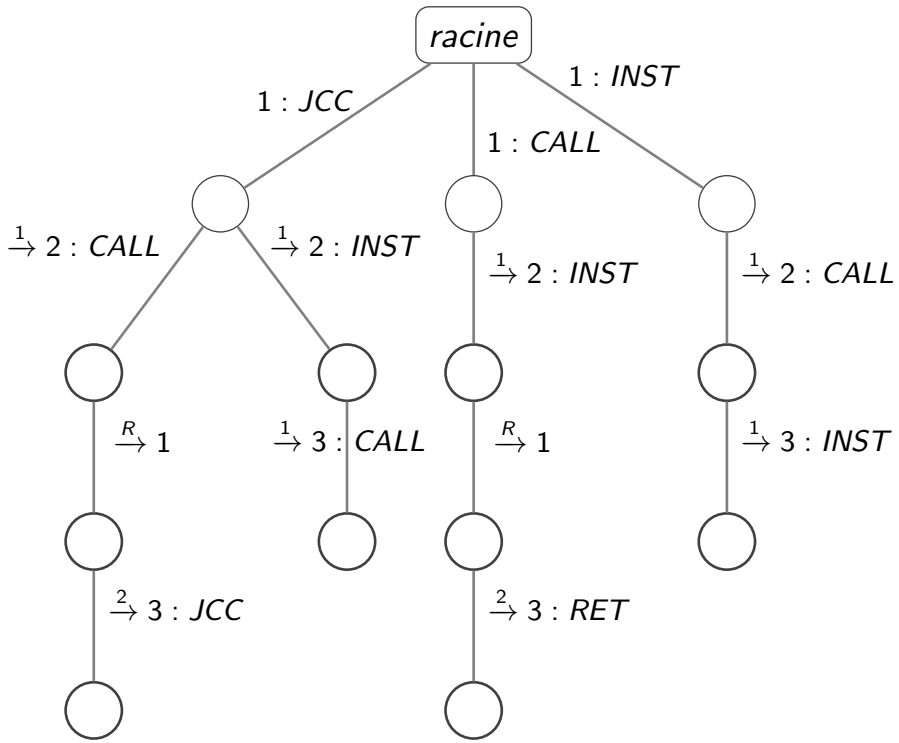
Détection d'isomorphismes par parcours de codage



$1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{R} 1 \xrightarrow{2} 3 : JCC$

- La possibilité du parcours d'un codage indique s'il y a un isomorphisme
- Chaque site est testé indépendamment des autres

Détection d'isomorphismes par parcours de codage



Codages rangés dans un arbre :

- Certains parcours sont factorisés
- On doit parcourir tous les sommets de l'arbre

Approche par découpage

Autre approche :

- Découper également le graphe de flot du programme à analyser en sites
- Recherche d'isomorphismes de sites

Approche par découpage

Autre approche :

- Découper également le graphe de flot du programme à analyser en sites
- Recherche d'isomorphismes de sites

Dans ce cas :

- Un site a une représentation unique (matricielle)
- On peut ranger les sites dans un arbre de hauteur fixe
- La détection d'un site revient à la recherche dans un arbre ordonné

Approche par découpage

Autre approche :

- Découper également le graphe de flot du programme à analyser en sites
- Recherche d'isomorphismes de sites

Dans ce cas :

- Un site a une représentation unique (matricielle)
- On peut ranger les sites dans un arbre de hauteur fixe
- La détection d'un site revient à la recherche dans un arbre ordonné

Conséquences :

- La complexité ne dépend plus du nombre de sites dans la base
- Mais l'algorithme n'est pas complet

Complexité en temps dans le pire des cas

Problème :

- Isomorphisme de sous-graphes
- Dans le cas des graphe de flot et des sites

- n : taille du graphe de flot
- S : nombre de sites dans la base
- W : taille des sites considérés (en pratique 24)

Algorithme	Ajout d'un graphe de flot	Analyse d'un graphe de flot	Complet
Recherche exhaustive	/	$O(S.n^3.n!)$	Oui
Ullmann	/	$O(S.(n/3)^W)$	Oui
Parcours	$O(n.W^2)$	$O(S.n.W)$	Oui
Découpage	$O(n.W^3)$	$O(n.W^2)$	Non

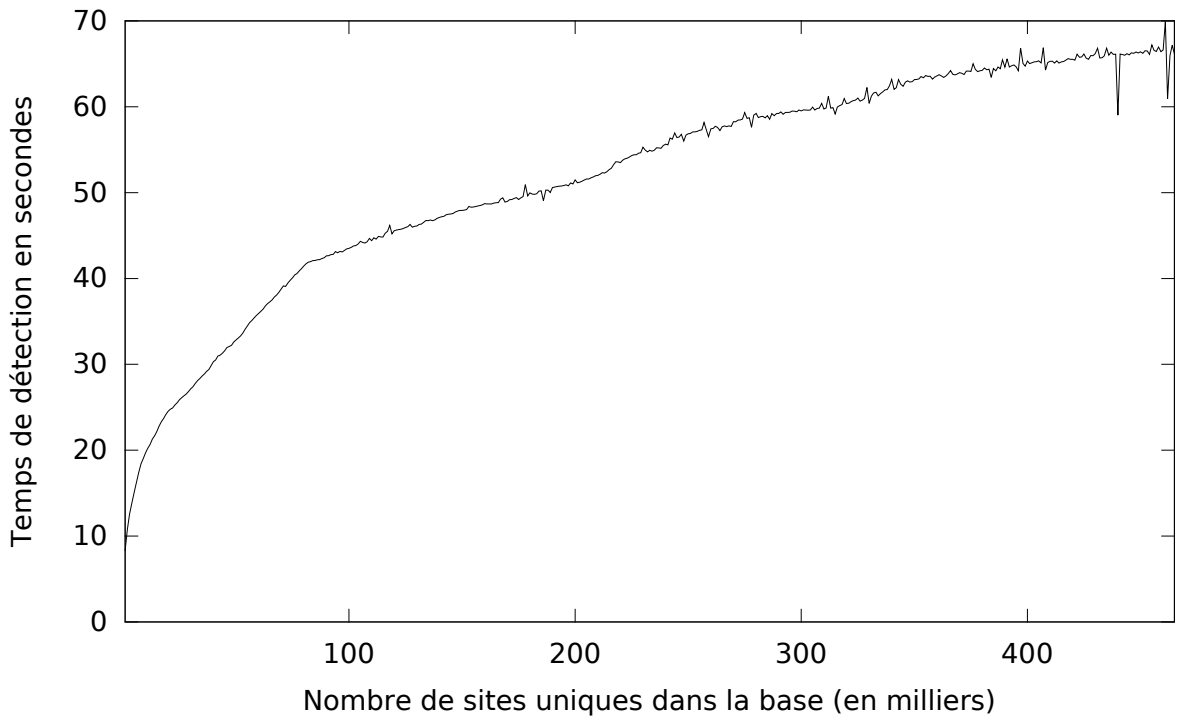
Implémentations : performance

- Base de programmes malveillants : 465 000 sites uniques
- Apprentissage : croissance linéaire, $<$ 6 minutes

Implémentations : performance

- Base de programmes malveillants : 465 000 sites uniques
- Apprentissage : croissance linéaire, $<$ 6 minutes
- Détection de 5 programmes (18 160 sites) en fonction du nombre de sites appris

Parcours :



Découpage : $<$ 1 seconde

Retour sur `hostname.exe` protégé

- Utilisation d'un binaire (`hostname.exe`) et de 34 versions protégées
- Extraction des graphes de flot de contrôle par analyse hybride
- Comparaison des graphes de flot par analyse morphologique

Retour sur `hostname.exe` protégé

- Utilisation d'un binaire (`hostname.exe`) et de 34 versions protégées
- Extraction des graphes de flot de contrôle par analyse hybride
- Comparaison des graphes de flot par analyse morphologique

Résultats :

- 28/34 binaires ont pu être analysés dynamiquement
- Dans 24/28 cas, on détecte plus de 75% des sites de `hostname.exe` dans la version protégée

Retour sur `hostname.exe` protégé

- Utilisation d'un binaire (`hostname.exe`) et de 34 versions protégées
- Extraction des graphes de flot de contrôle par analyse hybride
- Comparaison des graphes de flot par analyse morphologique

Résultats :

- 28/34 binaires ont pu être analysés dynamiquement
- Dans 24/28 cas, on détecte plus de 75% des sites de `hostname.exe` dans la version protégée

Comparaison des algorithmes de détection :

Algorithme	Apprentissage	Détection	Sites uniques détectés	Détection à + de 75%
Parcours	$<$ 1 seconde	70 secondes	6 739	24/28
Découpage	$<$ 1 seconde	1.6 seconde	6 045	22/28

Détection de similarités logicielles

Similarité logicielle

Problème : comparaison de deux programmes

Objectif : détecter de parties de code en commun

Similarité logicielle

Problème : comparaison de deux programmes

Objectif : détecter de parties de code en commun

Analyse morphologique :

- Découpage entre sites
- Correspondance entre sites

Similarité logicielle

Problème : comparaison de deux programmes

Objectif : détecter de parties de code en commun

Analyse morphologique :

- Découpage entre sites
- Correspondance entre sites

Adaptation :

- Correspondance fine entre instructions
- Identification automatique de fonctions

Implémentation :

- Avec l'algorithme par découpages
- Plugin pour IDA : affiche les similarités détectées

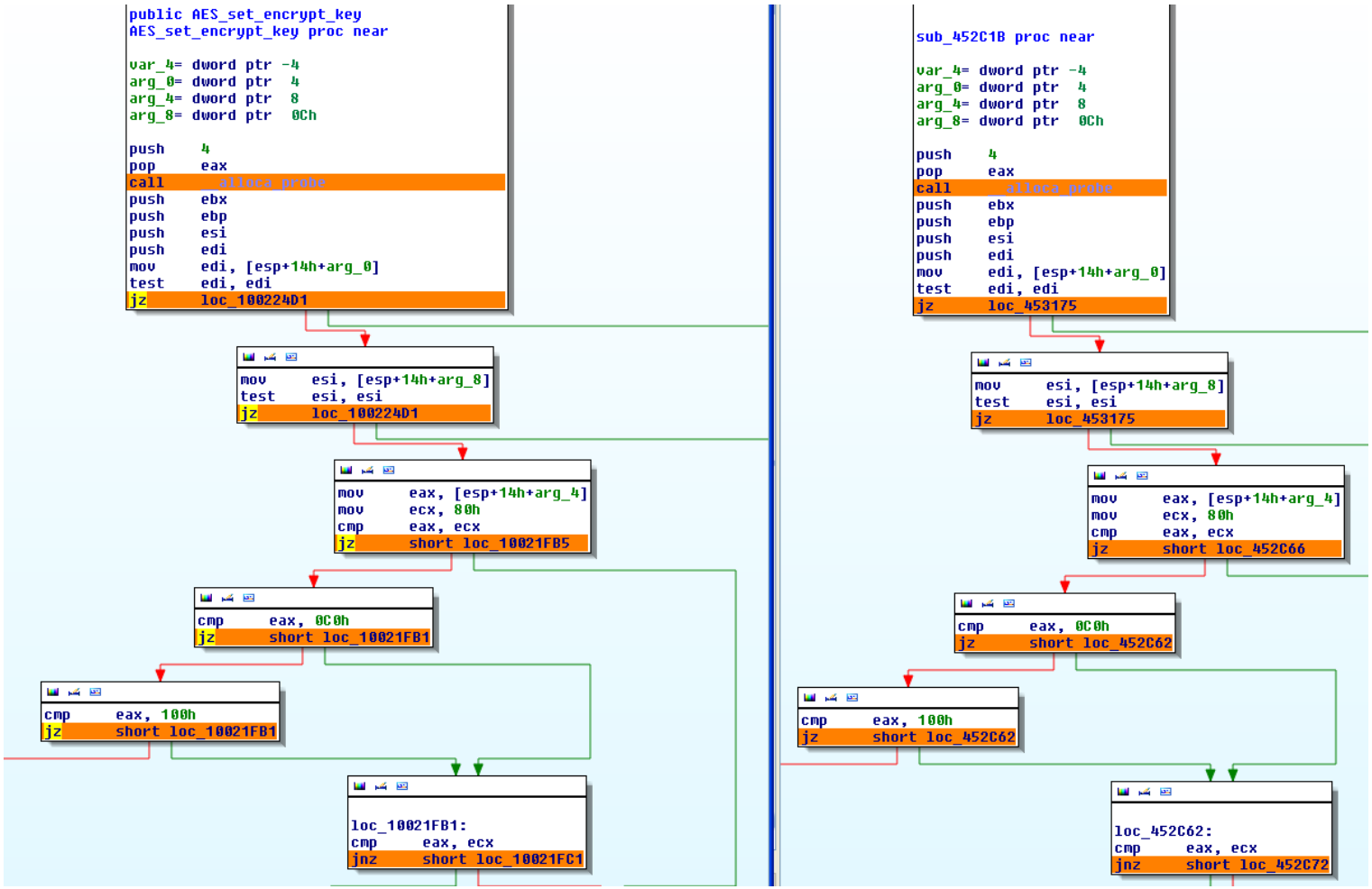
Waledac et OpenSSL

Problème :

- Analyser un programme malveillant : Waledac
- Pas d'informations de compilation (nom des fonctions)
- Utilise OpenSSL : Quelles fonctions sont utilisées ?

Waledac et OpenSSL

- Analyse morphologique et alignement du code



- IDA : OpenSSL à gauche, Waledac à droite

Waledac et OpenSSL[3]

Problème :

- Analyser un programme malveillant : Waledac
- Pas d'informations de compilation (nom des fonctions)
- Utilise OpenSSL : Quelles fonctions sont utilisées ?

Fonctions	Fonctionnalité
AES_set_encrypt_key, AES_set_decrypt_key	Chiffrement AES
RSA_free, DSA_size, DSA_new_method	Chiffrement RSA / DSA
X509_PUBKEY_set, X509_PUBKEY_get	Certificats X509
BN_is_prime_fasttest_ex, BN_ctx_new	Gestion des grands entiers
CRYPTO_lock, CRYPTO_malloc	Fonctions génériques

[3] Bonfante, Calvet, Marion, Sabatier, Thierry (REcon 2012, Malware 2012)

Similarité logicielle : Duqu et Stuxnet[4]

Application à Duqu (2011) et Stuxnet (2010)

[4] Bonfante, Marion, Sabatier, Thierry (SSTIC 2013, Malware 2013)

Similarité logicielle : Duqu et Stuxnet[4]

Application à Duqu (2011) et Stuxnet (2010)

- 26.5% des sites de Duqu en commun avec Stuxnet
- 60.3% des sommets du graphe de flot de Duqu correspondent à des sommets de Stuxnet

[4] Bonfante, Marion, Sabatier, Thierry (SSTIC 2013, Malware 2013)

Similarité logicielle : Duqu et Stuxnet[4]

Application à Duqu (2011) et Stuxnet (2010)

- 26.5% des sites de Duqu en commun avec Stuxnet
- 60.3% des sommets du graphe de flot de Duqu correspondent à des sommets de Stuxnet

La charge finale de Duqu est similaire à Stuxnet

- Présente uniquement en mémoire
- Comment la détecter avant l'infection ?
- Duqu surveille les processus pour s'y injecter

[4] Bonfante, Marion, Sabatier, Thierry (SSTIC 2013, Malware 2013)

Similarité logicielle : Duqu et Stuxnet[4]

Application à Duqu (2011) et Stuxnet (2010)

- 26.5% des sites de Duqu en commun avec Stuxnet
- 60.3% des sommets du graphe de flot de Duqu correspondent à des sommets de Stuxnet

La charge finale de Duqu est similaire à Stuxnet

- Présente uniquement en mémoire
- Comment la détecter avant l'infection ?
- Duqu surveille les processus pour s'y injecter

Rétroingénierie du pilote de Duqu

- Construction d'une version défensive
- Capable de détecter une infection par Duqu

[4] Bonfante, Marion, Sabatier, Thierry (SSTIC 2013, Malware 2013)

Conclusion

Conclusion

Désassemblage :

- Analyse et désassemblage de programmes binaires obscurcis
- Traitement du chevauchement de code
- Graphe de flot de contrôle pour programmes auto-modifiants
- Implémentation d'un désassembleur hybride

Détection :

- Comparaison des graphes de flot de contrôle
- Formalisation et optimisation de l'analyse morphologique
- Application à la détection de similarités logicielles

Perspectives

Désassemblage :

- Évaluer les faux positifs sur un grand nombre d'échantillons
- Utiliser des techniques d'interprétation abstraite
- Incorporer plusieurs traces dans le même graphe de flot de contrôle

Analyse morphologique :

- Utiliser d'autres critères du graphe de flot (chevauchements)
- Développer des méthodes de réductions résistantes à certains obscurcissements
- Étudier en détail la précision (faux positifs, faux négatifs) et l'influence des constantes (taille des sites, seuil de détection)

Merci !