L’usage de l’exécution symbolique pour la déobfuscation binaire en milieu industriel

Soutenance de thèse

Jonathan Salwan
13 février 2020

Jury de thèse

Vincent Nicomette (Rap.)
Jean-Yves Marion (Rap.)
Philippe Elbaz-Vincent (Exa.)
Sarah Zennou (Exa.)
Robin David (Exa.)
Marie-Laure Potet (Dir.)
Sébastien Bardin (Enc.)
Cédric Tessier (Inv.)

Contexte : mission d’audit de sécurité

- La société **X** veut mettre en ligne une application **Y**
- L’application **Y** reflète l’image de la société **X**
- La société **X** fait donc faire tester l'application **Y**

Le rôle de l’analyste “expert”

Rôle : Analyser des programmes informatiques pour en comprendre leur fonctionnement et trouver des comportements suspects.

- Ex. niveau d’analyse
  - Source
    - Partielle
    - Complète
  - **Binaire**
    - i686
    - AArch64
    - ARM
- Ex. d’objectifs
  - Analyse de *malware*
  - Recherche de vulnérabilités
  - Analyse de protections
  - Compréhension d’algorithmes

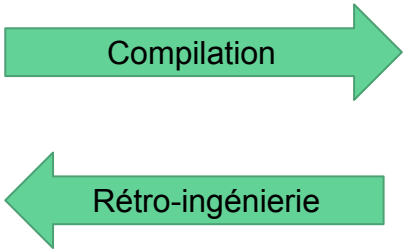
La rétro-ingénierie

Définition : La rétro-ingénierie, ou ingénierie inverse, est l'activité qui consiste à étudier un objet pour en déterminer le fonctionnement interne ou la méthode de fabrication.

Exemple :

```
int foo(int a, int b) {
  return a * b + 1;
}
```

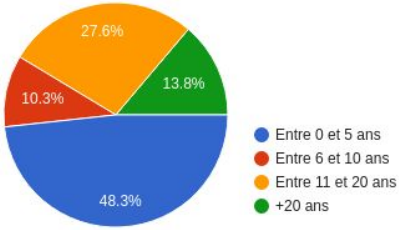
Une fonction qui prend deux entiers **a** et **b** et qui renvoie le résultat de **a * b + 1**




```
55 48 89 e5 89 7d fc
89 75 f8 8b 45 fc 0f
af 45 f8 83 c0 01 5d
c3 55 48 89 e5 53 48
83 ec 18 89 7d ec 48
89 75 e0 48 8b 45 e0
48 83 c0 08 48 8b 00
```

Défis de la rétro-ingénierie

- Liste non exhaustive des défis auxquels les analystes sont confrontés
  - D1. Le manque d'information
  - D5. Résolution des contraintes
  - D6. Le suivi des données
  - D9. Taille des programmes
  - Etc.
- Il y a 20 ans la rétro-ingénierie était principalement manuelle
  - Un art cultivé par des petits groupes de personnes
- Aujourd'hui on passe de l'art à l'ingénierie (industrialisation)
  - Automatisation des analyses et création d'outils
    - Extraire des informations rapidement (gagner du temps)
    - Rejouer et d'expliquer des résultats plus facilement



Années d'expérience des analystes interrogés

Liens entre les défis et l'outillage

D1	Le manque d'informations
D2	Distinguer les zones de DATA et de CODE
D3	Côtoyer l'assembleur
D4	Identification d'algorithmes connus
D5	Résolution des contraintes
D6	Suivi de données
D7	Extraction et réutilisation de code binaire
D8	Stabilité de l'exploitation logicielle
D9	Taille des programmes
D10	L'analyse statique et le milieu de l'embarqué
D11	Prise en main des outils dans le temps imparti

O1. Désassembleur	IDA, Ghidra, Capstone, Binary Ninja, Radare2, Hopper
O2. Débogueur	GDB, Ollydbg, lldb, x64dbg
O3. IR	BIL, VEX, REIL, ESIL, Vine, DBA
O4. Exec. symbolique	Miasm, Angr, Manticore, S2E, PathGrind, BINSEC
O5. Analyse de teinte	BinCAT, TaintGrind
O6. Instrumentation	Pin, QBDI, DynamoRIO, Valgrind
MC. Méthodologie	Méthodologie d'analyse et conception des outils

	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11
O1		✓		✓			✓			✓	
O2	✓						✓				
O3			✓				✓				
O4				✓	✓			✓			
O5				✓		✓					
O6	✓										
MC									✓		✓

Les contributions de la thèse

1. La rétro-ingénierie et ses problématiques
   a. Recueil des problématiques auprès d'analystes du milieu industriel
   b. Description des différents défis en liens avec l'outillage
2. Formalisation d'une bibliothèque d'analyse binaire
   a. Formalisation et limitation des algorithmes
   b. Impact de nos choix de conception sur les performances
3. Dévirtualisation binaire [0, 1]
   a. Méthode automatique de dévirtualisation
   b. Définition de métriques et expérimentations

[0] *Deobfuscation of VM based software protection, SSTIC 2017*
[1] *Symbolic Deobfuscation: From Virtualized Code Back to the Original, DIMVA 2018*

Plan

1. Présentation du travail fait sur Triton
    a. Quels sont les défis auxquels on s’attaque
    b. Description formelle des algorithmes d’analyse et des optimisations
2. Présentation d’une méthode de dévirtualisation basée sur Triton
3. Présentation des travaux futurs

Triton : bibliothèque d’analyse binaire

Objectifs

- Apporter une aide aux analystes lors de rétro-ingénierie
- Fonctionnalités axées sur les défis
  - D3 : Côtoyer l’assembleur
    - Définir des algorithmes d’analyse indépendamment des architectures
  - D5 : Résolution des contraintes
    - Trouver les entrées nécessaires pour arriver à un point précis
  - D6 : Suivi des données
    - Définir l’impact d’une donnée sur l’exécution
  - D9 : Taille des programmes
    - Analyser des traces de plusieurs centaines de millions d’instructions
  - D11 : Prise en main des outils
    - Intégration avec d’autres outils

Ingrédients clés de Triton

- Exécution concrète
- Analyse de teinte dynamique
- Exécution symbolique dynamique
- Concrétisation basée sur l’état de teinte
    - Passage à l’échelle

Background : Analyse de teinte [30, 86]

- La teinte est le marquage d'une donnée
- Permet de suivre la propagation d'une donnée au travers l'exécution
- Granularité de la teinte : bit, byte, word, dword, ..., objet

Trace d'exécution	État de teinte
	{esi}
(1) mov eax, esi	{esi, eax}
(2) inc eax	{esi, eax, of, sf, zf, af, pf}
(3) mov ebx, eax	{esi, eax, of, sf, zf, af, pf, ebx}
(4) mov eax, 0xbfffffff7	{esi, of, sf, zf, af, pf, ebx}
(5) mov [eax], ebx	{esi, of, sf, zf, af, pf, ebx, @eax:32}
(6) mov edi, [eax]	{esi, of, sf, zf, af, pf, ebx, @eax:32, edi}

Exemple d'analyse de teinte dynamique avec
une teinte initiale sur esi

Background : Exécution symbolique [47, 48, 49, 51]

- Représenter une trace d'exécution par un ensemble de contraintes
  - On appelle les contraintes menant à un point : le prédicat de chemin
- Principale utilisation de l'exécution symbolique
  - Trouver les entrées nécessaires pour suivre un chemin
  - Explorer de nouveaux chemins

Code source	**Valeurs concrètes**	**Construction du prédicat de chemin**
int f(int x, int y, int z) {	$\{x = 1, y = 10, z = 20\}$	
if (x == 0) {	$\{x = 1, y = 10, z = 20\}$	$\phi_{pc2^1} \triangleq (x_0 \neq 0)$
x = y + z ;	Non exécuté	
}		
else if (x == 1) {	$\{x = 1, y = 10, z = 20\}$	$\phi_{pc2^2} \triangleq (x_0 \neq 0) \wedge (x_0 = 1)$
x = y - z ;	$\{x = -10, y = 10, z = 20\}$	$\phi_{pc2^3} \triangleq (x_0 \neq 0) \wedge (x_0 = 1) \wedge x_1 = y_0 - z_0$
}		
else {	Non exécuté	
x = -1 ;	Non exécuté	
}		
return x ;	$\{x = -10, y = 10, z = 20\}$	$\phi_{pc2^4} \triangleq (x_0 \neq 0) \wedge (x_0 = 1) \wedge x_1 = y_0 - z_0$
}		

Construction du prédicat de chemin suivant
une exécution concrète

Background : Notions de correction et de complétude

- Suivant la définition donnée par Patrice Godefroid [47]

Correction : On note *PW* le prédicat de chemin d'une trace d'exécution *W* d'un programme *P* et on considère que le prédicat de chemin est correct si toutes les entrées satisfaisant *PW* suivent le chemin *W*.

Complétude : On considère que le prédicat de chemin est complet si toutes les entrées qui suivent la trace *W* vérifie le prédicat de chemin *PW* .

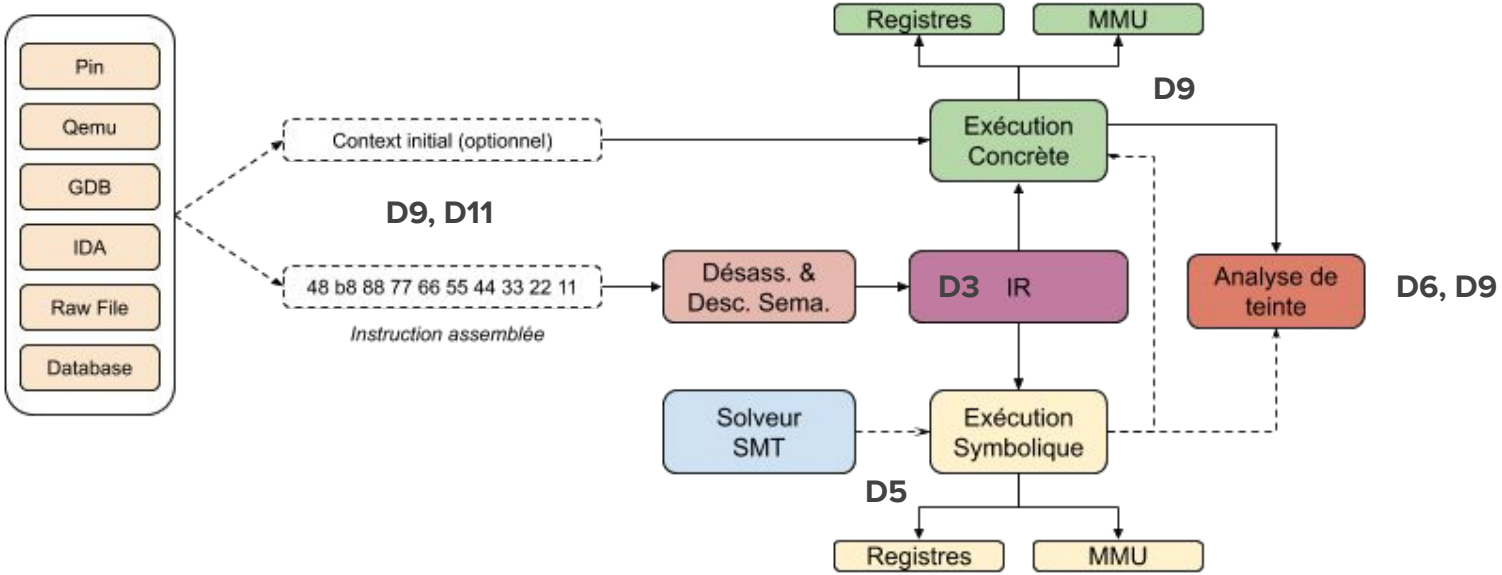
Background : Notion de concrétisation [34, 47]

- Remplacer une expression symbolique par une valeur concrète
- Avantage
  - Simplification des formules (ex. calcul d'une adresse d'allocation)
- Inconvénients
  - On perd potentiellement en correction
  - On perd en complétude

	Valeurs concrètes	**Prédicat de chemin**
ex.1	$\{x = -10, y = 10, z = 20\}$	$\phi_{pc2^4} \triangleq (x_0 \neq 0) \wedge (x_0 = 1) \wedge x_1 = 10 - z_0$
ex.2	$\{x = -10, y = 10, z = 20\}$	$\phi_{pc2^4} \triangleq (x_0 \neq 0) \wedge (x_0 = 1) \wedge x_1 = y_0 - z_0 \wedge y_0 = 10$

Triton

- D3 : Côtoyer l'assembleur
- D5 : Résolution des contraintes
- D6 : Suivi des données
- D9 : Taille des programmes
- D11 : Prise en main des outils



Les algorithmes

- Exécution concrète
  - On note Σ l'état concret à chaque point du programme
  - $\Sigma : Reg \cup Mem \mapsto Val$
  - **Algorithme classique**
- Analyse de teinte
  - On note Σ^t l'état de la teinte à chaque point du programme
  - $\Sigma^t : Reg \cup Mem \mapsto bool$
    - Granularité de l'ordre de l'objet
  - **Algorithme classique**

Algorithme d’exécution symbolique

- On note $\sum^*$ l’état symbolique à chaque point du programme
- $\Sigma^* : Reg \cup Mem \mapsto \varphi$
- **Particularité** pour le passage à l’échelle
  - Concrétisation des accès mémoire
  - Pas de modèle mémoire abstrait

Algorithme d’exécution symbolique

Expressions

$$constante\ \frac{}{\Sigma, \Sigma^*, bv \vdash_s bv} \qquad registre\ \frac{r \in Reg}{\Sigma, \Sigma^*, r \vdash_s \Sigma^*[r]} \qquad unaire\ \frac{\Sigma^*, e \vdash_s \varphi_e \quad \varphi \triangleq \diamond_u \varphi_e}{\Sigma, \Sigma^*, \diamond_u e \vdash_s \varphi}$$

$$binaire\ \frac{\Sigma^*, e_1 \vdash_s \varphi_1 \quad \Sigma^*, e_2 \vdash_s \varphi_2 \quad \varphi \triangleq \varphi_1 \diamond_b \varphi_2}{\Sigma, \Sigma^*, (e_1 \diamond_b e_2) \vdash_s \varphi} \qquad @\ \frac{\Sigma, e \vdash_e addr}{\Sigma, \Sigma^*, @e \vdash_s \Sigma^*[addr]}$$

$$@_k\ \frac{\Sigma, e \vdash_e addr}{\Sigma, \Sigma^*, @_k e \vdash_s (\Sigma^*[addr + (k-1)] \parallel ... \parallel \Sigma^*[addr + 1] \parallel \Sigma^*[addr + 0])}$$

$$ite\ \frac{\Sigma^*, c \vdash_s \varphi_c \quad \Sigma^*, e1 \vdash_s \varphi_1 \quad \Sigma^*, e2 \vdash_s \varphi_2}{\Sigma, \Sigma^*, ite(c, e_1, e_2) \vdash_s ite(\varphi_c, \varphi_1, \varphi_2)}$$

Règles d’évaluation symbolique des expressions

Algorithme d’exécution symbolique

𝕀nstructions

$$lhs - reg\ \frac{r \in Reg \quad r \neq pc \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow \varphi]}{\phi, \Sigma, \Sigma^*, r := src \rightsquigarrow \phi, \Sigma, \Sigma^*_{new}}$$

$$lhs - reg_{h..l}\ \frac{r \in Reg \quad r \neq pc \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow (\Sigma^*[r]_{s-1..h+1} \parallel \varphi \parallel \Sigma^*[r]_{l-1..0})]}{\phi, \Sigma, \Sigma^*, r_{h..l} := src \rightsquigarrow \phi, \Sigma, \Sigma^*_{new}}$$

$$lhs - @\ \frac{\Sigma^*, src \vdash_s \varphi \quad \Sigma, e \vdash_e addr \quad \Sigma^*_{new} \triangleq \Sigma^*[addr \leftarrow \varphi]}{\phi, \Sigma, \Sigma^*, @e := src \rightsquigarrow \phi, \Sigma, \Sigma^*_{new}}$$

$$lhs - @_k\ \frac{\Sigma^*, src \vdash_s \varphi \quad \Sigma, e \vdash_e addr \quad \Sigma^*_{new} \triangleq \Sigma^*[(\bigcup_{i=0}^{k-1} addr + i \leftarrow \varphi_{((i*8)+7)..(i*8)})]}{\phi, \Sigma, \Sigma^*, @_k e := src \rightsquigarrow \phi, \Sigma, \Sigma^*_{new}}$$

Règles d’évaluation symbolique des instructions

Correction et complétude

- La concrétisation des accès mémoire nous permet de passer à l'échelle
- Conséquences :
  - Incorrect et incomplet en cas d'accès mémoire symboliques durant l'exécution

Accès mémoire sur la trace	**Correct**	**Complet**
Accès mémoire symbolique	✗	✗
Accès mémoire constant	✓	✓

Correction et complétude du modèle symbolique de Triton

Optimisation : Exécution symbolique guidée par la teinte

- Garde symbolique les données teintées, concrétise le reste
- Objectifs :
  - Réduire la taille des expressions en se basant sur l’état de teinte
    - C’est l’utilisateur qui définit la teinte initiale
  - Réduire davantage la consommation mémoire
  - Passage à l’échelle
    - Analyse de trace contenant des centaines de millions d’instructions
- L’efficacité de l’optimisation dépend de la teinte définie par l’utilisateur
  - Tout teindre est équivalent à une exécution symbolique
  - Ne rien teindre est équivalent à une exécution concrète

Exécution symbolique guidée par la teinte

Instructions

$$T-lhs-reg\ \frac{r \in Reg \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^t, src \vdash_t True \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow \varphi]}{\phi, \Sigma^*, r := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$F-lhs-reg\ \frac{r \in Reg \quad \Sigma, src \vdash_e v \quad \Sigma^t, src \vdash_t False \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow v]}{\phi, \Sigma^*, r := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$T-lhs-reg_{h..l}\ \frac{r \in Reg \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^t, src \vdash_t True \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow (\Sigma^*[r]_{s-1..h+1} \parallel \varphi \parallel \Sigma^*[r]_{l-1..0})]}{\phi, \Sigma^*, r_{h..l} := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$F-lhs-reg_{h..l}\ \frac{r \in Reg \quad \Sigma, src \vdash_e v \quad \Sigma^t, src \vdash_t False \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow (\Sigma[r]_{s-1..h+1} \parallel v \parallel \Sigma[r]_{l-1..0})]}{\phi, \Sigma^*, r_{h..l} := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$T-lhs-@\ \frac{\Sigma, e \vdash_e addr \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^t, src \vdash_t True \quad \Sigma^*_{new} \triangleq \Sigma^*[addr \leftarrow \varphi]}{\phi, \Sigma^*, @e := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$F-lhs-@\ \frac{\Sigma, src \vdash_e v \quad \Sigma, e \vdash_e addr \quad \Sigma^t, src \vdash_t False \quad \Sigma^*_{new} \triangleq \Sigma^*[addr \leftarrow v]}{\phi, \Sigma^*, @e := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

Règle d’évaluation symbolique des instructions guidée par la teinte

Exécution symbolique guidée par la teinte

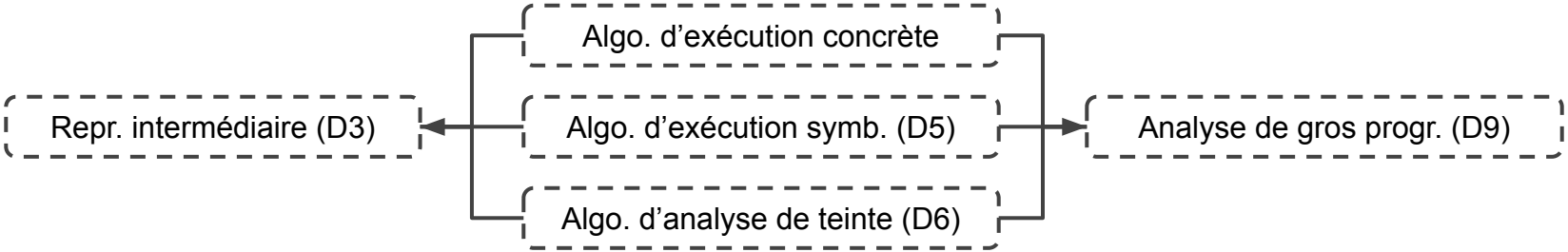
Instructions

$$T - lhs - reg\ \frac{r \in Reg \quad \Sigma^*, src \vdash_s \varphi \quad \Sigma^t, src \vdash_t True \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow \varphi]}{\phi, \Sigma^*, r := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

$$F - lhs - reg\ \frac{r \in Reg \quad \Sigma, src \vdash_e v \quad \Sigma^t, src \vdash_t False \quad \Sigma^*_{new} \triangleq \Sigma^*[r \leftarrow v]}{\phi, \Sigma^*, r := src \rightsquigarrow \phi, \Sigma^*_{new}}$$

Zoom sur les règles d’affectation sur registre

Conclusion sur Triton



- Conçu pour être intégré avec d'autres outils (D11)
  - Outils dynamiques : **QBDI**+Triton, **Pin**+Triton, **GDB**+Triton *(HITCON 2017)*
  - Outils statiques : **IDA**+Triton *(Hex-Rays contest 2016)*, **Radar**+Triton *(r2con 2017)*, **LIEF**+Triton
- Exemples des possibilités d'utilisation :
  - Couverture de code pour la recherche de vulnérabilités
  - Analyser des protections logicielles
    - Ex. détecter la présence de prédicats opaques
    - Permet d'améliorer les deux parties (attaque et défense)

Dévirtualisation par exécution symbolique

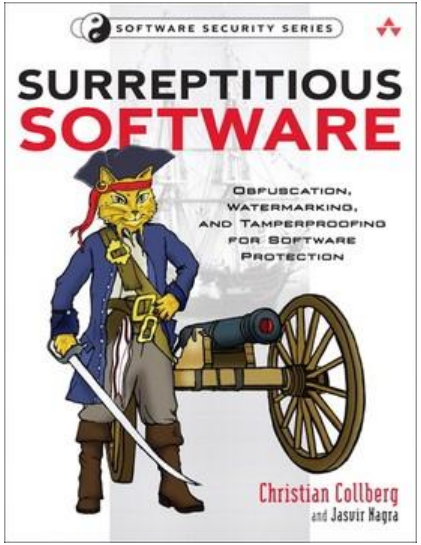
La protection logicielle

Définition : On considère qu'un secret **S** est caché dans un programme **P**. La protection logicielle est le fait de transformer le code initial du programme **P**, tout en conservant sa sémantique, pour rendre la détection et la compréhension du secret **S** plus difficile pour un analyste.

- Exemple de secret :
  - Clé de chiffrement (ex. clé AES)
  - Algorithme
  - Calcul d'intégrité
  - Mécanisme d'authentification
  - Données
  - Etc.

Différentes* protections logicielles

- Protection contre l’analyse
  - Anti-debug
  - Anti-VM
  - Vérification d’intégrité
- Protection de donnée
  - Chiffrement et déchiffrement
  - Constantes opaques
- Protection de code
  - CFG aplati (code flattening)
  - Code mort
  - Opérations arithmétiques complexifiées (ex. MBA)
  - **Machine virtuelle**

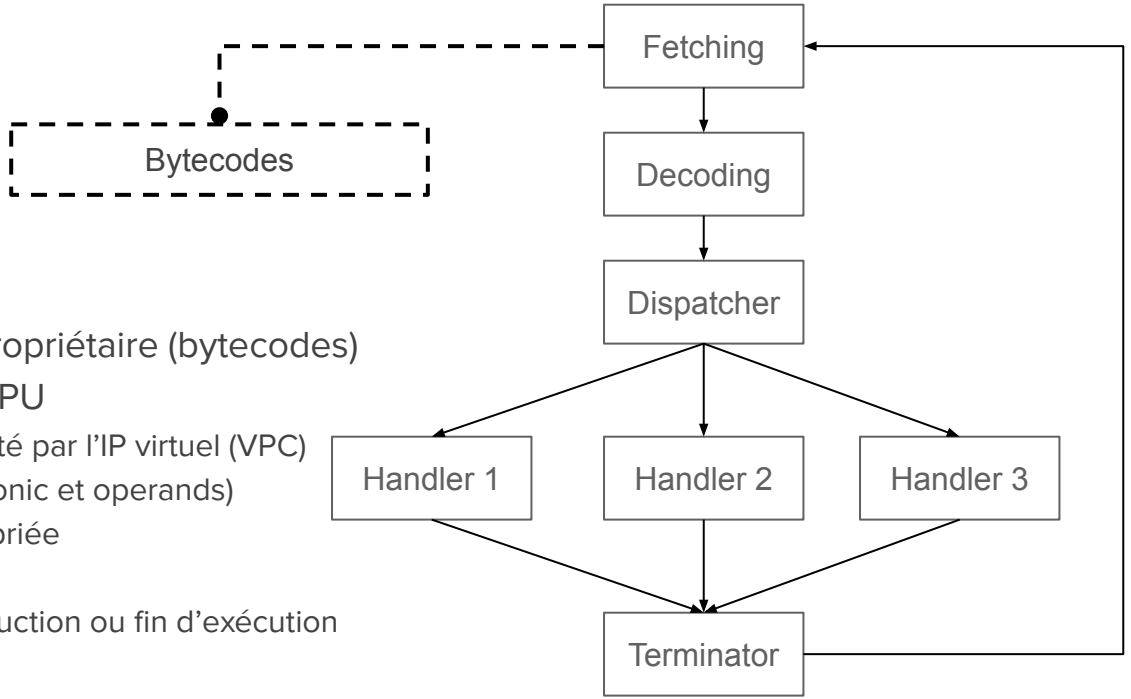


* : Liste non exhaustive

Protection logicielle par virtualisation

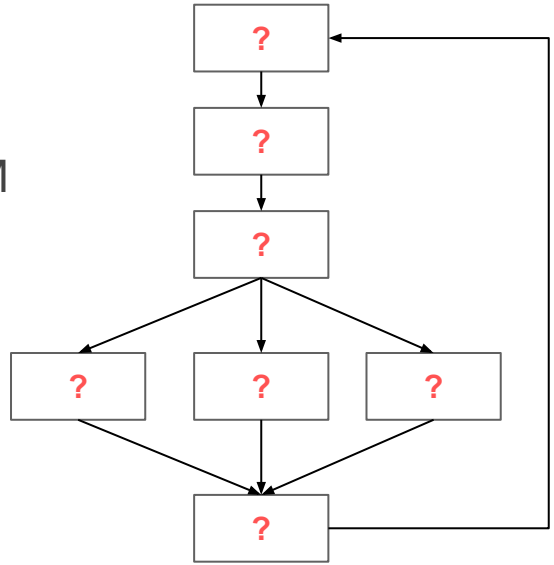
Protection logicielle - Exemple de Machine Virtuelle

- Aussi appelé virtualisation
- Virtualise un jeu d'instructions propriétaire (bytecodes)
- Structure proche de celle d'un CPU
  - Récupération de l'opcode pointé par l'IP virtuel (VPC)
  - Décodage de l'opcode (mnemonic et operands)
  - Appel de la sémantique appropriée
  - Exécution de la sémantique
  - Exécution de la prochaine instruction ou fin d'exécution



Virtual Machine - Les défis pour un analyste

1. Identifier que le programme est virtualisé et identifier ses entrées
2. Identifier chacun des composants de la machine virtuelle
3. Comprendre le fonctionnement de ses composants
4. Comprendre comment le VPC est calculé
5. Créer un désassembleur pour le bytecode de la VM
6. Commencer à analyser le code virtualisé



État de l’art

Position de notre approche

	Manuelle	**Coogan** (2011)	**Kinder** (2012)	**Yadegari** (2015)	**Notre approche**
Identification entrées **Compr. le VPC** **Compr. le dispatcher** **Compr. le bytecode**	Requis Requis Requis Requis	Requis Non Non Non	Requis Requis Non Non	Requis Non Non Non	Requis Non Non Non
Sortie	CFG simplifié	Trace simplifiée	CFG + invariants	CFG simplifié	Code simplifié
Analyse(s) utilisée(s)	-	*Value-based slicing*	Analyse Statique (interp. abstraite)	Teinte, symbolique et simpl. de code	Teinte, symbolique, simpl. de formules et simpl. de code.
xp: type de code **xp: #exemple** **xp: metrics evaluation**	- - -	*Toys + malware* 12 % Simplification	*Toy* 1 Invariants connus	*Toys + malware* 44 Similarité	Algorithmes 945 Taille, Equi. Sem.

Notre approche
Dévirtualisation automatisée

Notre approche - Dévirtualisation automatisée

Intuition clef :

trace obfusquée = **instructions d'origine** + **instructions virtualisée**

0. Identification des entrées de la fonction virtualisée
1. Sur une trace, isoler les instructions d'origine avec une analyse de teinte
2. Construire la représentation symbolique de ces instructions teintées
3. Effectuer une DSE pour découvrir les chemins de la fonction virtualisée
4. Reconstruire un binaire à partir de l'ensemble des traces parcourues

Étape 1 - Analyse de teinte

- **But :** Séparer les **instructions originales** des **instructions de la VM**
- **Entrée :** Une fonction virtualisée
- **Analyse : Teinte** des entrées utilisateur (découvertes en étape 0) puis exec.
- **Sortie :** Deux sous-traces d'instructions
  - Instructions teintées = **instructions pertinentes**
    - ⚠ Instructions réellement pertinentes que si le flot de données ne dépend pas des entrées

Étape 2 - Représentation symbolique

- **But :** Représenter la sous-trace d'instructions pertinentes avec des expressions symboliques
  a. Appliquer une exploration de chemins (étape 3)
  b. Faciliter la traduction vers une représentation intermédiaire de compilation (étape 4)
- **Entrée :** Une sous-trace d'instructions pertinentes
- **Analyse :** Représentation symbolique de la trace d'exécution avec la politique de concrétisation basée sur l'état de teinte
- **Sortie :** Une sous-trace d'instructions pertinentes sous la forme d'expressions symboliques

Étape 3 - Exploration de chemins

- **But :** Reconstruire le comportement complet de la fonction virtualisée
- **Entrée :** Une sous-trace d’instructions pertinentes sous la forme d’expressions symboliques
- **Analyse :** Application d’une **exploration symbolique** de chemins
- **Sortie :** Un arbre d’exécution des instructions pertinentes sous la forme d’expressions symboliques

Étape 4 - Génération d’un programme compilé

- **But :** Fournir un binaire exécutable sans mécanisme de virtualisation
- **Entrée :** Un arbre d’exécution des instructions pertinentes sous la forme d’expressions symboliques
- **Analyse :** Traduction des expressions symboliques vers la représentation intermédiaire de LLVM, puis **compilation avec optimisations**
- **Sortie :** Un nouveau binaire

Expérimentations

Expérimentations : Nos objectifs et critères

- Nos objectifs :
  - Vérifier que le comportement du programme après dévirtualisation est conservé
  - Vérifier que la taille du programme dévirtualisé est similaire à celle d'origine
  - Vérifier la performance de notre analyse
  - Vérifier l'impact des protections sur le résultat de notre analyse

- Nos critères :
  - C1. Précision : correction et concision
  - C2. Efficacité
  - C3. Robustesse

Expérimentations : Banc de test

- Première expérimentation : Environnement contrôlé
  - Choix des sources
  - Contrôle des combinaisons des protections
  - Banc de test : 920 binaires protégés avec différentes combinaisons de protections

- Deuxième expérimentation : Environnement non contrôlé (challenges Tigress)
  - Aucune connaissance des sources
  - Aucun contrôle sur le choix des protections appliquées
  - Challenge non résolu publiquement
  - Banc de test : 25 binaires protégés

Les fonctions de hachage analysées

Hash	Loops	Binary Size (inst)	# executable paths
Adler-32	✓	78	1
CityHash	✓	175	1
Collberg-0001-0	✓	167	1
Collberg-0001-1	×	177	2
Collberg-0001-2	×	223	1
Collberg-0001-3	✓	195	1
Collberg-0001-4	✓	183	1
Collberg-0004-0	×	210	2
Collberg-0004-1	×	143	1
Collberg-0004-2	✓	219	2
Collberg-0004-3	✓	171	1
Collberg-0004-4	✓	274	1
Fowler-Noll-Vo Hash (FNV1a)	×	110	1
Jenkins	✓	79	1
JodyHash	✓	90	1
MD5	✓	314	1
SpiHash	✓	362	1
SpookyHash	✓	426	1
SuperFastHash	✓	144	1
Xxhash	✓	182	1

Les protections combinées

Protecticons	Options
Anti Branch Analysis	goto2push, goto2call, branchFuns
Max Merge Length	0, 10, 20, 30
Bogus Function	0, 1, 2, 3
Kind of Operands	stack, registers
Opaque to VPC	true, false
Bogus Loop Iterations	0, 1, 2, 3
Super Operator Ratio	0, 0.2, 0.4, 0.6, 0.8, 1.0
Random Opcodes	true, false
Duplicate Opcodes	0, 1, 2, 3
Dispatcher	binary, direct, call, interpolation, indirect, switch, ifnest, linear
Encode Byte Array	true, false
Obfuscate Decode Byte Array	true, false
Nested VMs	1, 2, 3

Résultats : Précision (C1)

- Critères :
  - **Correction :** Vérifier que le comportement du programme est conservé
  - **Concision :** Vérifier que la taille du programme dévirtualisé est similaire à celle d'origine
- Métriques utilisées :
  - **Correction :** P(seed) == P'(seed)
  - **Concision :**
    - Ratio du nombre d'instructions **Originale** → **Virtualisé**
    - Ratio du nombre d'instructions **Originale** → **Dévirtualisé**

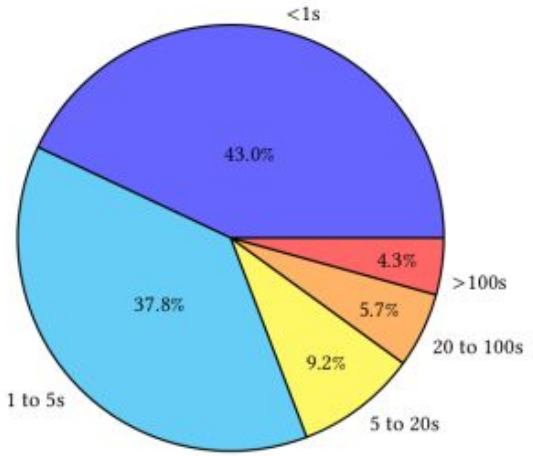
	Original → Obfuscate	Original → Deobfuscate
Correctness	100% (920 successes)	
Binary Size	avg : x6	avg : x0.71
Trace Size	avg : x424	avg : x0.39

Résultats : Efficacité (C2)

- Critère :
  - **Efficacité :** Vérifier la performance de notre analyse
- Métriques utilisées :
  - Mesure du temps d'analyse sur l'ensemble des échantillons

	Obfuscated
Binary Size	min : 468 max : 5,424 avg : 1,205
Trace Size	min : 1,349 max : 47,927,795 avg : 229,168

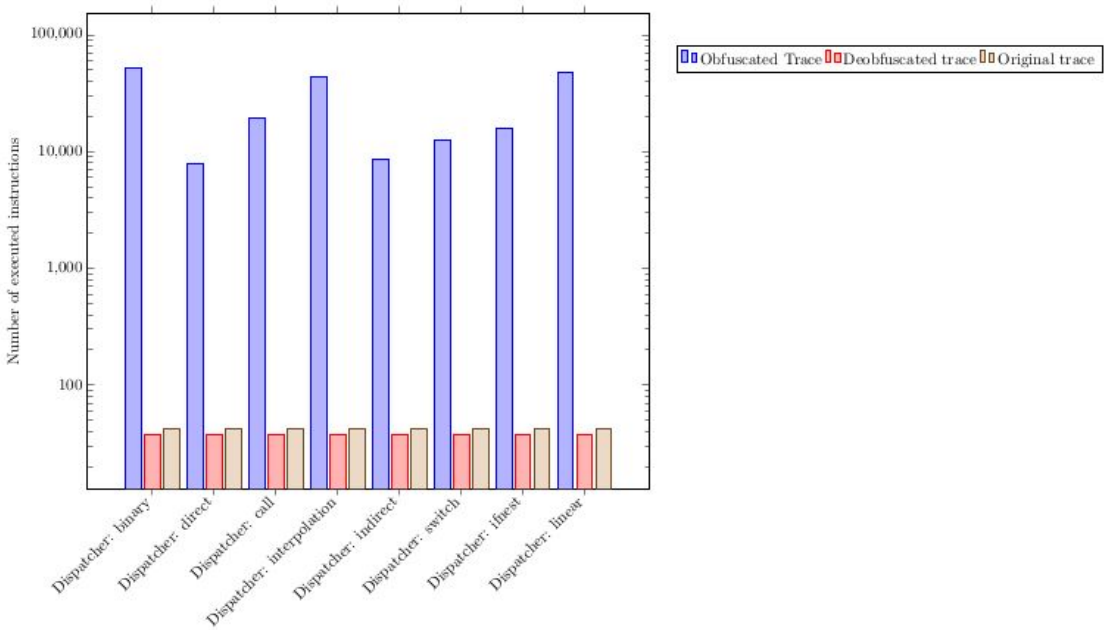
Taille de nos échantillons en instructions



Palier de temps pour nos 920 échantillons

Résultats : Influence des protections (C3)

- Critère :
  - **Robustesse :** Vérifier l'impact des protections sur le résultat de notre analyse
- Métriques utilisées :
  - On considère les résultats de la métrique sur la **concision**



Expérimentation : Le challenge Tigress

Challenge	Description	Number of binaries	Difficulty (1-10)	Script	Prize	Status
0000	One level of virtualization, random dispatch.	5	1	script	Certificate issued by DAPA	Solved
0001	One level of virtualization, superoperators, split instruction handlers.	5	2	script	Signed copy of Surreptitious Software.	Solved
0002	One level of virtualization, bogus functions, implicit flow.	5	3	script	Signed copy of Surreptitious Software.	Solved
0003	One level of virtualization, instruction handlers obfuscated with arithmetic encoding, virtualized function is split and the split parts merged.	5	2	script	Signed copy of Surreptitious Software.	Solved
0004	Two levels of virtualization, implicit flow.	5	4	script	USD 100.00	Solved
0005	One level of virtualization, one level of jitting, implicit flow.	5	4	script	USD 100.00	Solved
0006	Two levels of jitting, implicit flow.	5	4	script	USD 100.00	Open

Expérimentation : Le challenge Tigress

	Challenges Tigress				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	3.85s	9.20s	3.27s	4.26s	1.58s
0001	1.26s	1.42s	3.27s	2.49s	1.74s
0002	6.58s	2.02s	2.63s	4.85s	3.82s
0003	45.6s	11.3s	8.84s	4.84s	21.6s
0004	361s	315s	588s	8049s	1680s

Temps d'analyse (secondes)

	Challenges Tigress				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	x0.85	x1.09	x0.73	x0.89	x1.4
0001	x0.41	x0.60	x0.26	x0.22	x0.53
0002	x0.29	x0.28	x0.51	x1.4	x0.42
0003	x1.10	x1.17	x1.57	x0.46	x0.44
0004	x0.81	x0.38	x0.70	x0.37	x0.53

Ratio (taille)
originale → dévirtualisée

Limitations de notre analyse

Limitations liées à Triton	**Résultat de notre analyse**
Présence d'index symbolique	incorrect & incomplet

Limitations liées à la méthode	**Résultat de notre analyse**
Timeout des contraintes	incomplet
Entrées entrelacées avec le flot de données	peu concis
Présence de boucle	CFG déroulé

Travaux futurs

Travaux futurs

- Dans les tuyaux :
  - Support ARM v7 et v8
  - Accès mémoire symbolique avec différentes politiques de concrétisation
    - C/C, C/S, S/C, S/S *[34]*
- Proposer deux formes de concrétisation possibles
  - Forme actuelle : incorrect mais réduction de la taille des expressions
  - Forme future : correct mais sans réduction de la taille des expressions
- Granularité de la teinte
  - Déterminer les inconvénients et les avantages d'une granularité au niveau bit
- Reconstruction d'un CFG à partir d'un ensemble de traces

[34] *Specification of concretization and symbolization policies in symbolic execution*, *ISSTA 2016*

Références

[30] *Dytan : a generic dynamic taint analysis framework, ACM 2007.*

[34] *Specification of concretization and symbolization policies in symbolic Execution, ISSTA 2016.*

[47] *Higher-order test generation, ACM SIGPLAN 2011.*

[48] *Automating software testing using program analysis, IEEE 2008.*

[49] *DART : directed automated random testing, ACM SIGPLAN 2005.*

[51] *SAGE : whitebox fuzzing for security testing, ACM 2012.*

[86] *All you ever wanted to know about dynamic taint analysis and forward symbolic execution, SP 2012.*