

# Désobfuscation binaire : Reconstruction de fonctions virtualisées

Jonathan Salwan,  
Sébastien Bardin et Marie-Laure Potet

`jsalwan@quarkslab.com`  
`sebastien.bardin@cea.fr`  
`Marie-Laure.Potet@imag.fr`

Quarkslab, CEA, VERIMAG

**Résumé.** La virtualisation est une technique de protection binaire qui consiste à transformer un *bytecode* original vers un nouveau jeu d'instructions (ISA - Instruction Set Architecture) propriétaire. Cette nouvelle ISA est ensuite interprétée par une machine virtuelle (embarquée dans le binaire) afin de simuler le comportement d'origine. Cette protection a pour but de cacher le code binaire d'origine ainsi que son *control flow graph* (CFG), ce qui implique de devoir comprendre (*reverser*), en premier lieu, la machine virtuelle avant de pouvoir *reverser* le programme ciblé. Nous proposons une méthode permettant de dévirtualiser une fonction (ou un ensemble de fonctions) de façon automatique (et peut-être générique) en s'appuyant sur les concepts de base de l'analyse de teinte, l'exécution symbolique, la décompilation vers LLVM puis la recompilation. Nous illustrons ensuite nos résultats sur 25 virtualisations différentes basées sur la protection Tigress ainsi que leurs challenges. Nous expliquons également dans quels cas cette méthode fonctionne ainsi que ses limitations.

## 1 Introduction

### 1.1 Protection logicielle

La protection logicielle (*obfuscation* [15]) est un enjeu important par exemple dans la lutte pour la conservation des propriétés intellectuelles contenues dans les programmes informatiques. En effet, il est possible d'analyser un programme, d'en comprendre son fonctionnement et donc de pouvoir le reproduire ou même le détourner. Cela pose problème quand une société investit plusieurs centaines de milliers d'euros dans sa recherche et son développement et qu'une société tierce récupère ses algorithmes et donc sa propriété intellectuelle en quelques semaines d'analyse.

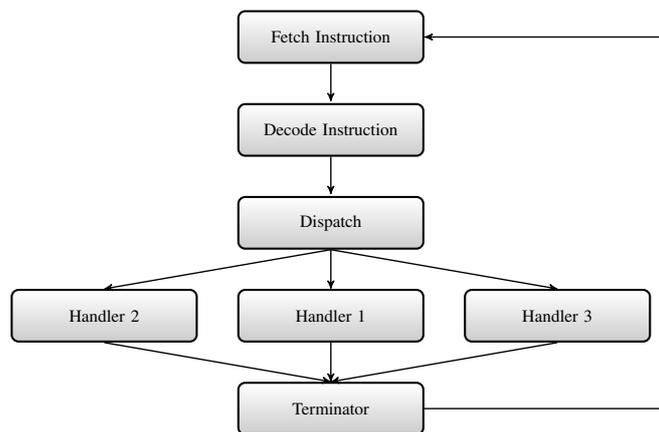
Si nous prenons pour exemple une société de jeux vidéo, une très grande partie de ses bénéfices se fait lors de la sortie d'un jeu attendu.

Si ce dernier se retrouve accessible sur internet dans les 24 heures qui suivent la sortie, la société perd une partie de ses bénéfices. La protection logicielle vise justement à rendre compliquée l'analyse d'un programme afin de ralentir au maximum celui qui l'attaque. Dans l'exemple de la sortie d'un jeu vidéo, l'objectif serait de faire en sorte que la protection tienne au moins 20 jours, le temps que la société puisse tirer profit de son développement.

Vous l'aurez compris, une protection logicielle est faite pour ralentir et non pas garantir à 100% la sécurité d'un programme. C'est pourquoi tester une protection permet d'évaluer le temps et les compétences nécessaires à un individu pour casser la protection<sup>1</sup>.

## 1.2 Protection par virtualisation

La virtualisation [1, 8] est une technique de protection binaire qui consiste à transformer un *bytecode* original vers un nouveau jeu d'instructions (ISA - Instruction Set Architecture) propriétaire. Cette nouvelle ISA est ensuite interprétée par une machine virtuelle (embarquée dans le binaire) afin de simuler le comportement d'origine. Cette protection a pour but de cacher le code binaire d'origine ainsi que son *control flow graph* (CFG), ce qui implique de devoir comprendre (*reverser*), en premier lieu, la machine virtuelle avant de pouvoir *reverser* le programme ciblé.



**Fig. 1.** Architecture d'une machine virtuelle

L'architecture classique d'une machine virtuelle (VM) est proche de celle d'un CPU (d'où son nom - voir figure 1). Cette dernière est composée

<sup>1</sup> Noter également que la désobfuscation est utile pour l'analyse de *malware*

de 4 composants<sup>2</sup> qui constituent l'interprétation de l'architecture ainsi qu'une série de fonctions décrivant la sémantique des *opcodes*.

La machine virtuelle commence par récupérer (*fetch*) les *opcodes* pointés par son VIP (*Virtual Instruction Pointer*), les décode, puis *dispatch* les opérands au bon *handler* qui lui applique la sémantique de l'instruction. Ensuite, c'est au rôle du *terminator* de faire évoluer le VIP afin de déterminer si l'exécution virtuelle doit continuer ou s'arrêter.

Ce genre de protection peut être décrite depuis du code source (listing 1) par le biais de `define`, `macro`, `pragma`... Peu importe la méthode utilisée pour définir la zone à protéger, le but étant d'effectuer une transformation au niveau de la compilation pour que le code à protéger soit remplacé par celui de la VM.

```
1 #pragma clang virtualized
2 ulong f(ulong x) {
3     [...]
4 }
```

**Listing 1.** Tag depuis source

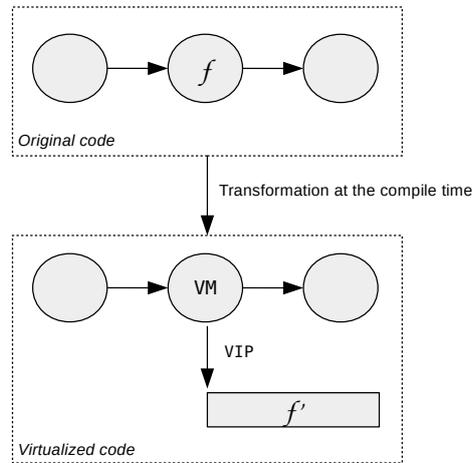
En prenant l'exemple du listing 1 et en partant du principe que nous avons une passe de transformation dans Clang pour effectuer de la virtualisation de fonction, le compilateur transformera la fonction  $f$  en son équivalent virtualisé (dans l'ISA de la machine virtuelle) noté  $f'$ . Puis le compilateur injectera en lieu et place de la fonction  $f$  le code de la VM qui simulera  $f'$ . La figure 2 illustre ce procédé.

Comme nous pouvons le constater, le but de cette protection est de protéger le code d'origine de  $f$  afin de cacher par exemple, des algorithmes de chiffrement, une méthode d'authentification ou encore du *parsing* de formats propriétaires (bref, toutes propriétés intellectuelles voulant être conservées).

### 1.3 Contribution

La désobfuscation binaire sur des protections telles que la virtualisation est un sujet assez récent (environ depuis 2009). Plusieurs travaux existent sur le sujet [5, 12, 16, 20] avec des approches statique et dynamique. Notre approche est proche de celle développée par l'université d'Arizona [5] excepté

<sup>2</sup> *fetch*, *decode*, *dispatch*, *terminator*



**Fig. 2.**  $f \rightarrow f'$

qu'elle repose sur trois sujets en pleine expansion ces dernières années : l'exécution symbolique [3, 11], l'analyse de teinte [18] et la décompilation vers l'IR LLVM [14].

L'approche que nous proposons permet d'isoler les instructions correspondantes au code d'origine sur des traces d'exécution symbolique ainsi que de reconstruire un code valide dévirtualisé. Cette approche fonctionne bien dans le cas de petites fonctions telles que des fonctions de hachage propriétaires.

#### 1.4 Discussion

Retrouver 100% du code d'origine d'une fonction virtualisée est une tâche très complexe pour ne pas dire impossible. C'est pourquoi la dévirtualisation tente de proposer au mieux un code pouvant correspondre à celui d'origine. La question que l'on peut se poser est :  *finalement, est-il vraiment nécessaire de retrouver 100% du code d'origine tant que la sémantique du programme dévirtualisé est conservée ?* Notre objectif est de proposer un binaire dévirtualisé sémantiquement identique au code d'origine permettant au *reverser* de mieux comprendre le programme. Cependant, notre approche possède plusieurs limitations décrites dans la section 4.

## 2 Désobfuscation, notre approche

Comme vous pouvez l’imaginer, le but d’une désobfuscation de protection comme la virtualisation serait de retrouver (au plus proche) le code d’origine de la fonction  $f$  initiale. Nos expérimentations se sont portées sur des fonctions de hachage propriétaires au travers de la protection Tigress<sup>3</sup>. Le reste de l’article expose notre approche.

### 2.1 Aperçu de la méthode

Notre approche repose sur un fait essentiel au fonctionnement d’une machine virtuelle : quoi qu’il arrive, la machine virtuelle devra exécuter l’instruction sur le CPU *host* correspondant à la sémantique de l’*opcode* à simuler. Autrement dit, une addition de deux opérandes dans l’ISA propriétaire restera quoi qu’il arrive un ADD sur x86 (sans prendre en compte que les opérateurs peuvent être obfusqués, nous ne parlons ici que de virtualisation).

*Étape 1 - Analyse par teinte* : L’objectif est donc d’arriver à isoler ces instructions (que nous appelons instructions de fin de séquence de virtualisation ou encore instructions pertinentes), qui, mises bout à bout, nous permettront de nous rapprocher du code d’origine. Pour cela, nous effectuons une analyse de teinte où les entrées de la fonction virtualisée sont teintées (dans le cas du listing 1 nous teintons l’argument  $x$ ).

*Étape 2 - Exécution symbolique* : Une fois les instructions pertinentes isolées, nous devons leur donner un sens sémantique (leurs contextes). Pour cela nous effectuons une exécution symbolique dynamique guidée par l’analyse de teinte. Toute expression non teintée est donc concrétisée. Ce qui revient à se débarrasser des instructions concernant le fonctionnement interne de la machine virtuelle.

*Étape 3 - Couverture de code* : Afin d’extraire la sémantique complète d’une fonction virtualisée, nous devons prendre en compte l’ensemble de ses chemins. Pour cela, nous effectuons une couverture de code basée sur les expressions symboliques récupérées lors de l’étape 2.

---

<sup>3</sup> <http://tigress.cs.arizona.edu>

*Étape 4 - Reconstruction binaire* : Pour finir, une fois la couverture de code effectuée, nous convertissons notre *Abstract Syntax Tree* (AST) vers celui de LLVM [14]. Ceci nous permet par la suite de recompiler et donc de reconstruire du code valide. Noter qu'en bonus il est également possible d'appliquer en même temps les optimisations du compilateur et de recompiler vers une autre architecture.

## 2.2 Étape 1 - Analyse par teinte

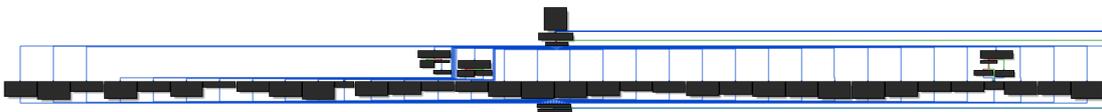
**Fonction virtualisée sans branchement** Prenons comme exemple une simple fonction qui prend en argument un nombre et retourne un dérivé de ce nombre (une fonction de hachage simple)  $f(x) \rightarrow x'$ . Le calcul effectué par la fonction  $f$  est virtualisé. Le listing 2 illustre le code d'origine.

```

1 void f(ulong input[1], ulong output[1]) {
2     ulong a1 = input[0] >> 5UL;
3     ulong a2 = a1 ^ 810798164723513605UL;
4     ulong a3 = input[0] - 275339905UL;
5     ulong a4 = a3 + a2;
6
7     output[0] = a4;
8 }
```

**Listing 2.** Fonction de hachage simple

Comme expliqué précédemment, à la compilation, cette fonction  $f$  est virtualisée et est donc remplacée par le code de la VM. On note cette transformation  $f \rightarrow f'$ . Le *control flow graph* (CFG) de  $f'$ , illustré par la figure 3, est donc celui de la VM.



**Fig. 3.** Control Flow Graph de  $f'$

Le CFG contient les 4 composants de virtualisation expliqués précédemment (**fetch**, **decode**, **dispatch** et **terminator**) ainsi que les opérateurs de la VM illustrés par la séquence de *basic blocks* horizontale.

En utilisant l'analyse de teinte dynamique [4]<sup>4</sup>, il nous est possible de rapidement identifier le rôle de chaque *handler* de la machine virtuelle

<sup>4</sup> Nous utilisons ici notre propre moteur de teinte.

(ses opérateurs). Par exemple, si nous teintons l'entrée `input[0]` de la fonction  $f'$  et que nous affichons toutes les instructions teintées sur une trace d'exécution concrète dans IDA, nous avons le résultat illustré par la figure 4.

```

00000000004008E4
00000000004008E4 loc_4008E4: ; jumtable 000000000040072E case 2
00000000004008E4 mov rax, [rbp+vip]
00000000004008E8 add rax, 1
00000000004008EC mov [rbp+vip], rax
00000000004008F0 mov rax, [rbp+var_80]
00000000004008F4 lea rdx, [rax-8]
00000000004008F8 mov rax, [rbp+var_80]
00000000004008FC sub rax, 8
0000000000400900 mov rsi, [rax]
0000000000400903 mov rax, [rbp+var_80]
0000000000400907 mov rax, [rax]
000000000040090A mov rbx, rsi
000000000040090D mov ecx, eax
000000000040090F shr rbx, cl
0000000000400912 mov rax, rbx
0000000000400915 mov [rdx], rax
0000000000400918 mov rax, [rbp+var_80]
000000000040091C sub rax, 8
0000000000400920 mov [rbp+var_80], rax
0000000000400924 jmp loc_400E6B

```

Fig. 4. Handler d'opérateur SHR

Comme nous pouvons le constater, il nous est facile de déterminer que *ce handler* effectue un SHR. Pour donner un autre exemple de *handler* identifiable par la teinte, la figure 5 illustre l'opération XOR du *bytecode* de la machine virtuelle. L'adresse 400854 est le chargement du contexte, 40085E l'opération XOR et 400861 la sauvegarde du contexte.

```

0000000000400838
0000000000400838 loc_400838: ; jumtable 000000000040072E case 4
0000000000400838 mov rax, [rbp+vip]
000000000040083C add rax, 1
0000000000400840 mov [rbp+vip], rax
0000000000400844 mov rax, [rbp+var_80]
0000000000400848 lea rdx, [rax-8]
000000000040084C mov rax, [rbp+var_80]
0000000000400850 sub rax, 8
0000000000400854 mov rcx, [rax]
0000000000400857 mov rax, [rbp+var_80]
000000000040085B mov rax, [rax]
000000000040085E xor rax, rcx
0000000000400861 mov [rdx], rax
0000000000400864 mov rax, [rbp+var_80]
0000000000400868 sub rax, 8
000000000040086C mov [rbp+var_80], rax
0000000000400870 jmp loc_400E6B

```

Fig. 5. Handler d'opérateur XOR

Pour résumer, si on extrait toutes les instructions teintées lors de l'exécution de  $f'$ , on obtient le résultat illustré par le listing 3. Pour une

meilleure lisibilité, nous avons séparé les *basic blocks* des *handlers* par des sauts de lignes.

Comme nous pouvons le constater, nous retrouvons tous les opérateurs de la fonction d'origine  $f$  (SHR, XOR, SUB et ADD). En d'autres mots, en utilisant l'analyse de teinte on se débarrasse de 1039 instructions (sur notre exemple) qui représentent le processus de virtualisation pour se concentrer sur 30 instructions qui définissent l'exécution d'origine (*instructions pertinentes*).

```

0x40066c: mov qword ptr [rbp - 0x10], rax
0x400675: mov rdx, qword ptr [rbp - 0x10]
0x400679: mov qword ptr [rbp + rax*8 - 0x30], rdx

0x4007c9: mov rdx, qword ptr [rdx]
0x4007cc: mov qword ptr [rax], rdx

0x400900: mov rsi, qword ptr [rax]
0x40090a: mov rbx, rsi
0x40090f: shr rbx, cl ; Original operation
0x400912: mov rax, rbx
0x400915: mov qword ptr [rdx], rax

0x400e19: mov rdx, qword ptr [rdx]
0x400e1c: mov qword ptr [rax], rdx

0x4007c9: mov rdx, qword ptr [rdx]
0x4007cc: mov qword ptr [rax], rdx

0x400854: mov rcx, qword ptr [rax]
0x40085e: xor rax, rcx ; Original operation
0x400861: mov qword ptr [rdx], rax

0x4007c9: mov rdx, qword ptr [rdx]
0x4007cc: mov qword ptr [rax], rdx

0x400ca2: mov rcx, qword ptr [rax]
0x400cb0: mov rbx, rcx
0x400cb3: sub rbx, rax ; Original operation
0x400cb6: mov rax, rbx
0x400cb9: mov qword ptr [rdx], rax

0x400be4: mov rcx, qword ptr [rax]
0x400bef: mov rax, qword ptr [rax]
0x400bf2: add rax, rcx ; Original operation
0x400bf5: mov qword ptr [rdx], rax

0x4006a9: mov rax, qword ptr [rbp + rax*8 - 0x20]
0x4006ae: mov rsi, rax

```

**Listing 3.** Analyse par teinte sur une exécution concrète de  $f'$

**Fonction virtualisée avec branchements** Étant donné qu'un code d'origine virtualisé peut contenir des branchements, il est nécessaire que la machine virtuelle supporte les conditions de branchement. Pour pouvoir reconstruire un CFG au plus proche de celui d'origine, nous devons donc isoler les conditions de branchement virtualisées. Ce qui importe, en vérité, est de déterminer si notre entrée influence des conditions de branchement. Pour cela, nous utilisons toujours l'analyse de teinte. La figure 6 nous montre qu'une condition est teintée, ce qui implique que le code d'origine utilise son entrée comme condition de branchement. En d'autres termes, à chaque fois que nous exécutons une instruction de branchement teintée, cela signifie qu'il existe une condition dans le code d'origine.

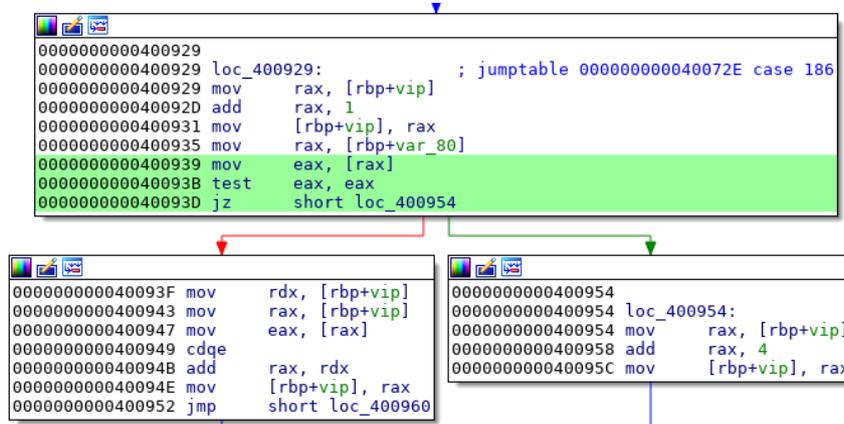


Fig. 6. Branche virtualisée

Ce concept de détection de branchement est important pour la reconstruction du CFG d'origine (du moins, avoir un CFG au plus proche de celui d'origine).

### 2.3 Étape 2 - Exécution symbolique

Faire une analyse de teinte n'est pas suffisant pour reconstruire un code valide. En effet, si nous plaçons nos instructions teintées les unes à la suite des autres, cela n'a aucun sens. Il nous faut donc reconstruire le contexte d'interaction entre ces instructions (donner un sens aux instructions teintées).

C'est pourquoi nous appliquons une exécution symbolique dynamique [9, 10, 13, 19] (DSE) uniquement sur les instructions teintées. Ce qui nous permet, dans un premier temps, de récupérer les expressions sémantiques correspondant à l'évolution de notre entrée (dans  $f^l$ ) tout au

long de l'exécution. Notons que les expressions décrivent l'évolution de l'entrée pour un seul chemin emprunté.

Afin de combiner l'analyse de teinte et l'exécution symbolique facilement, nous utilisons notre propre *framework* d'analyse binaire présenté au SSTIC 2015 [17]. Cet article peut également être vu comme un point sur l'avancement du projet Triton depuis la dernière publication au SSTIC.

**Notre modèle de DSE** Le modèle de DSE utilisé dans Triton est classique (voir [18] pour une intro en douceur). Nous partons d'un état concret (peu importe où) et à partir de ce point, des expressions arithmétiques et logiques sont assignées aux registres et aux cellules mémoires en fonction de la sémantique de l'instruction exécutée. Chacune de ces expressions possède un `id` unique qui lui est propre et la construction des expressions se fait sous la forme *Static Single Assignment* (SSA). Le listing 4 illustre un exemple d'expressions Triton sous la forme SSA pour l'exécution de deux instructions.

```
; Expressions Triton pour la séquence assembleur :
; mov rax, 1
; add rax, 2

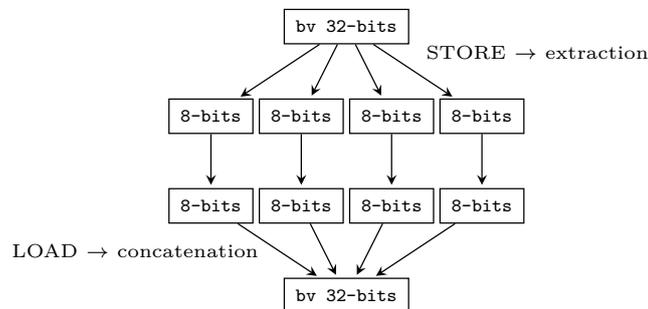
ref!0 = (_ bv1 64) ; MOV operation
ref!1 = (_ bv7 64) ; Program Counter
ref!2 = (bvadd
        ((_ extract 63 0) ref!0)
        (_ bv2 64)
      ) ; ADD operation
ref!3 = ... ; Adjust flag
ref!4 = ... ; Carry flag
ref!5 = ... ; Overflow flag
ref!6 = ... ; Parity flag
ref!7 = ((_ extract 63 63) ref!2) ; Sign flag
ref!8 = (ite
        (=
          ((_ extract 63 0) ref!2)
          (_ bv0 64)
        )
        (_ bv1 1)
        (_ bv0 1)
      ) ; Zero flag
ref!9 = (_ bv11 64) ; Program Counter
```

**Listing 4.** Exemple d'expressions Triton

La concrétisation [10] est le fait de casser cette forme SSA en réassignant une expression contenant la valeur concrète au lieu de la référence de son prédécesseur. Ainsi il nous est possible de couper facilement certaines branches de notre arbre d'expressions pour réduire sa complexité.

Le modèle mémoire est le suivant : chaque cellule mémoire est représentée par un vecteur de 8 bits dans le domaine des expressions. La logique des tableaux (QF\_ABV) n'est pas utilisée.

Comme certaines ISA (telle que x86) permettent l'accès non aligné à la mémoire, nous devons représenter la mémoire comme des vecteurs de 8 bits distincts et utiliser des concaténations ainsi que des extractions pour effectuer des `LOAD` et `STORE`. Autrement dit, si une expression effectue un `LOAD` de 4 octets, l'expression sera la concaténation de 4 cellules mémoire de 8 bits. À l'inverse si on `STORE` l'expression de `RAX` dans une zone mémoire, nous effectuons 8 extractions de 8 bits qui seront assignées à chacune des cellules mémoires correspondantes.



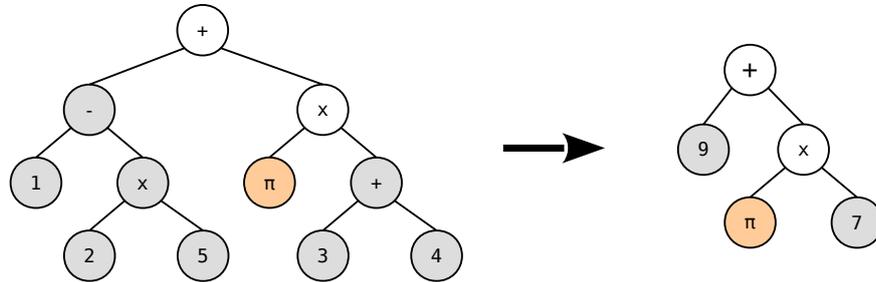
Notre politique de concrétisation (C/S [6] policy) est  $CC^5$  quand une adresse (`STORE` ou `LOAD`) est calculée (`LEA` - `Load Effective Address`). Cependant, leurs cellules mémoire peuvent, quant à elles, être symboliques. Ce modèle implique une taille fixe des tableaux et donc nous obtenons des expressions ne contenant qu'une seule logique. Ainsi, cette forme canonique nous permet d'effectuer plus facilement des transformations.

**Politique de concrétisation basée sur l'analyse de teinte** Comme dit précédemment, l'analyse de teinte nous permet de nous focaliser sur les instructions pertinentes. Notre exécution symbolique dynamique est donc effectuée uniquement sur les instructions teintées.

Si une instruction n'est pas teintée, les expressions assignées aux registres et aux cellules mémoires se basent sur les valeurs concrètes. Dans le cas contraire, si une instruction est teintée, les expressions sont assignées en se référant à leur prédécesseur (forme SSA) comme expliqué ci-dessus. En d'autres termes, nous concrétisons tout ce qui n'est pas en rapport avec notre entrée utilisateur, ce qui peut être illustré par la figure 7.

Notons que cela permet de concrétiser tout le processus d'exécution de la machine virtuelle (car celui-ci est constant). C'est donc cette concrétisation

<sup>5</sup>  $CC$  = concrétisation des `LOAD` et `STORE`



**Fig. 7.** Concrétisation basée sur notre entrée

tisation qui fait le plus gros du travail de simplification. Cela nous permet, par exemple, de passer de 1039 instructions exécutées à 30 instructions (dans le cas du listing 2). Sur d'autres exemples plus conséquents (que nous verrons dans la section 3) nous passons par exemple d'environ 140 millions d'expressions à environ 320.

**Expression sémantique de la fonction virtualisée** À la fin de l'exécution de la fonction virtualisée  $f'$ , nous effectuons du *backward slicing* sur l'expression de RAX et nous obtenons comme résultat l'expression sémantique de la transition  $f'(x) \rightsquigarrow x'$ . Cette transition représente la sémantique de la fonction d'origine  $f$  (sans la virtualisation). Le listing 5 représente le pseudo-code de cette transition dans lequel nous pouvons identifier les quatre opérateurs d'origine SHR, XOR, SUB et ADD ainsi que les trois constantes 5, 810798164723513605 et 275339905.

À ce stade, nous avons donc désobfusqué la fonction virtualisée  $f'$  pour arriver à un pseudo code au plus proche du code d'origine. Il faut également noter que, dans notre exemple, la fonction d'origine n'avait pas de condition de branchement ce qui simplifie la désobfuscation. Dans la section suivante, nous allons voir ce qu'il en est avec une fonction  $f$  contenant des branchements.

## 2.4 Étape 3 - Couverture de code

Précédemment, nous avons décrit notre approche pour dévirtualiser une fonction ne contenant aucune condition de branchement. Cependant, il est également possible, même si cela reste plus complexe, de dévirtualiser une

```

ref!228 := SymVar_0
ref!243 := (((_ extract 63 0) ref!228)) ; MOV operation
ref!1131 := (((_ extract 63 0) ref!243)) ; MOV operation
ref!1317 := (((_ extract 63 0) ref!1131)) ; MOV operation
ref!1323 := (((_ extract 63 0) ref!1317)) ; MOV operation
ref!1327 := (
  (bvlsr
    ((_ extract 63 0) ref!1323)
    (bvand
      ((_ zero_extend 56) (_ bv5 8))
      (_ bv63 64)
    )
  )
) ; SHR operation
ref!1334 := (((_ extract 63 0) ref!1327)) ; MOV operation
ref!2204 := (((_ extract 63 0) ref!1334)) ; MOV operation
ref!3066 := (((_ extract 63 0) ref!2204)) ; MOV operation
ref!3252 := (((_ extract 63 0) ref!3066)) ; MOV operation
ref!3258 := (
  (bvxor
    (_ bv810798164723513605 64)
    ((_ extract 63 0) ref!3252)
  )
) ; XOR operation
ref!4149 := (((_ extract 63 0) ref!243)) ; MOV operation
ref!4218 := (((_ extract 63 0) ref!4149)) ; MOV operation
ref!4232 := (((_ extract 63 0) ref!4218)) ; MOV operation
ref!4234 := (
  (bvsub
    ((_ extract 63 0) ref!4232)
    (_ bv275339905 64)
  )
) ; SUB operation
ref!4242 := (((_ extract 63 0) ref!4234)) ; MOV operation
ref!4331 := (((_ extract 63 0) ref!4242)) ; MOV operation
ref!4343 := (((_ extract 63 0) ref!3258)) ; MOV operation
ref!4345 := (
  (bvadd
    ((_ extract 63 0) ref!4343)
    ((_ extract 63 0) ref!4331)
  )
) ; ADD operation

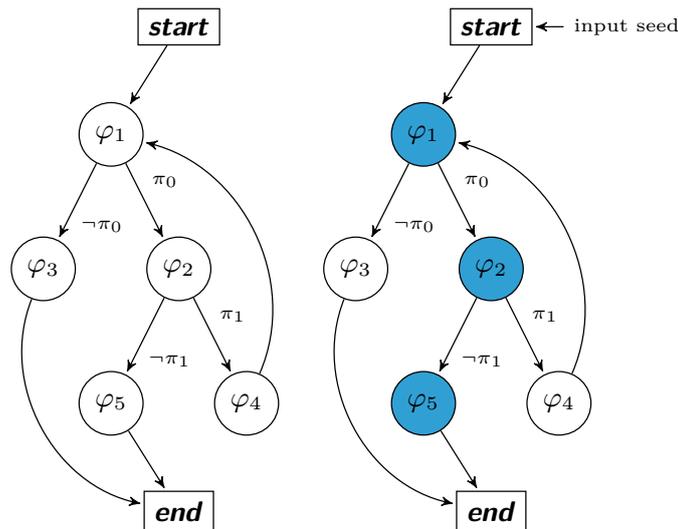
```

**Listing 5.** Expression sémantique de  $f'(x) \rightsquigarrow x'$

fonction contenant des conditions de branchement. Pour cela, il faut donc reconstruire son CFG et par conséquent connaître toutes les expressions sémantiques des sorties de la fonction (dans notre cas, la valeur de retour) pour tous les chemins qu'elle contient. On parle donc ici, de couverture de code.

Une couverture de code est applicable [3] dès lors que la fonction ciblée est simple. Dans le cas contraire, on atteint très rapidement un problème ouvert qui est propre à l'exécution symbolique, nommé *l'explosion de chemins* (nous en parlerons plus en détail dans la section 4).

Dans le cadre des challenges Tigress, l'explosion de chemins n'a pas lieu et il nous est donc possible de couvrir rapidement les fonctions de hachage. Pour cela, nous commençons par injecter une première valeur aléatoire  $x$  en entrée à la fonction ciblée  $f'(x)$ . La fonction prendra donc un premier chemin basé sur cette entrée (figure 8).

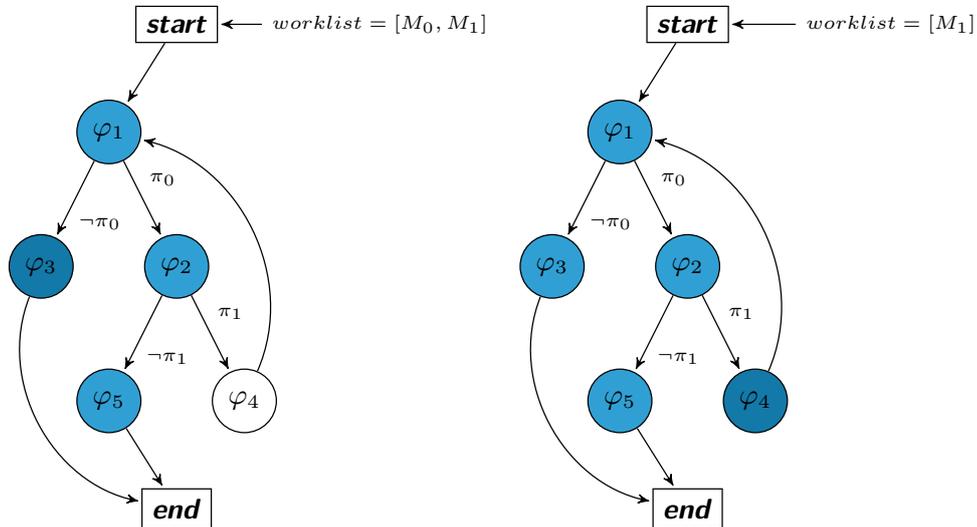


**Fig. 8.** Couverture de code

Sur cette première exécution, nous effectuons l'exécution symbolique basée sur l'analyse de teinte comme expliqué précédemment et appliquons le principe exposé à la fin de la section 2.2 pour isoler les conditions de branchement dans le code d'origine.

À la fin de la première exécution, nous demandons au *SMT solver* [11] les modèles nécessaires pour explorer toutes les nouvelles branches découvertes et non explorées. Ces modèles sont ensuite mis dans une *worklist*. Cette *worklist* est ensuite utilisée comme ensemble d'*inputs* et pour tous les *inputs* dans cet ensemble, une nouvelle exécution est faite qui, elle

aussi, rajoutera des nouveaux modèles à la *worklist*. Cette opération est répétée tant que la *worklist* n'est pas vide (figure 9).



**Fig. 9.** Couverture de code

**Fusion des traces** Une fois la couverture de code effectuée, nous obtenons un ensemble de traces sémantiques. Ces traces doivent maintenant être fusionnées entre elles pour reconstruire un CFG. La reconstruction d'un CFG basée sur les adresses des instructions est triviale, cependant ici nous ne pouvons appliquer cette méthode car on reconstruirait le CFG de la virtualisation. Il nous faut donc travailler sur l'unification des sémantiques (*merger* les traces d'exécution symbolique) et ceci est plus complexe qu'il n'y paraît. Actuellement, nous sommes obligés de convertir ces traces en IR LLVM afin de bénéficier des optimisations du compilateur pour effectuer ce *path merging* (la transformation vers l'IR LLVM est décrite dans la section 2.5).

```
A
B
C
if C:
  A
  B
  C
  D
  H
  I
else:
  A
```

```

B
C
E
F
G
H
I

```

**Listing 6.** CFG avant LLVM

Le listing 6 représente le CFG que nous envoyons à LLVM. Dans chacun des *basic blocks* du CFG nous avons la trace d'exécution depuis son origine. Une fois l'optimisation LLVM `-O2` appliquée, LLVM effectue son *path merging* et nous obtenons un CFG épuré (listing 7).

```

A
B
C
if C:
    D
else:
    E
    F
    G
H
I

```

**Listing 7.** CFG après le `-O2` d'LLVM

Bien que cette méthode fonctionne parfaitement bien et qu'elle nous ait permis de dévirtualiser la protection Tigress, nous reconnaissons qu'elle est plus que moyenâgeuse et sommes d'ores et déjà en train de travailler sur une méthode un peu plus sexy pour l'unification de traces basée sur leurs sémantiques. Nous avons plusieurs pistes comme l'utilisation de l'algorithme de Levenshtein, mais nous ne sommes actuellement qu'au stade de *PoC*. Noter également que nous n'arrivons pas à reconstruire les boucles étant donné qu'en dynamique elles sont déroulées (plus de détails dans la section 4).

## 2.5 Étape 4 - Reconstruction binaire

Bien que le pseudo code (qui peut être affiché en SMT2 ou Python) soit suffisant pour comprendre le code d'origine, nous avons poussé l'idée un peu plus loin en essayant de reconstruire du code machine valide. Pour cela, rien de mieux que de traduire notre AST (qui décrit nos expressions) vers la représentation intermédiaire de LLVM [14]. La conversion vers LLVM nous apporte plusieurs avantages :

1. **Recompilation** : la possibilité de recompiler du code depuis des expressions sémantiques pour reconstruire un binaire désobfusqué, et pourquoi pas, recompiler vers d'autres architectures.
2. **Optimisations** : bénéficier de toutes les analyses, optimisations et transformations du *framework* LLVM. Par exemple, il nous est possible de recompiler du code en utilisant l'option `-O2` ce qui nous permet de nettoyer encore un peu plus le code désobfusqué.

Pour pouvoir convertir l'AST Triton vers la représentation intermédiaire de LLVM sans trop d'efforts, nous avons choisi d'utiliser le *framework* Arybo [7] pour deux raisons :

1. **Solidarité** : étant donné qu'Arybo est développé dans le bureau à côté de celui de l'équipe Triton, il nous est facile d'échanger et de se synchroniser pour les besoins de chacun.
2. **Tout en un** : Arybo est un point central entre plusieurs représentations de plusieurs *frameworks* (*hey les loulous, il serait peut-être temps qu'on se synchronise sur une représentation commune à tout le monde, non ?*), ce qui nous permet de faire des aller-retour sans limite entre les représentations et donc de bénéficier de toutes les fonctionnalités de chacun. L'architecture d'Arybo est illustrée par la figure 10.

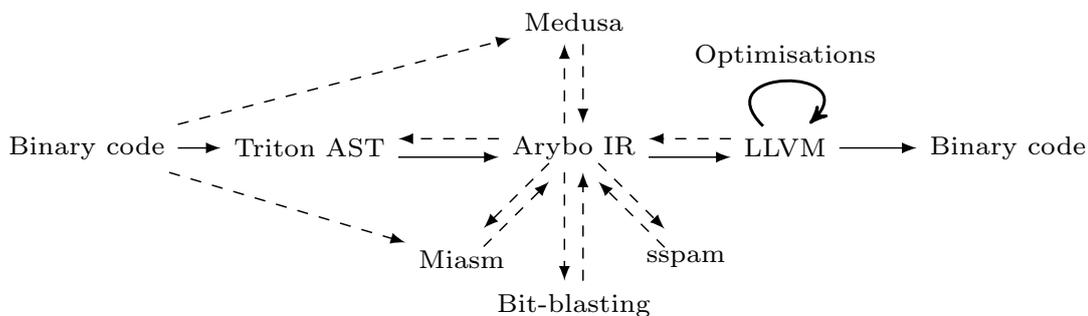


Fig. 10. Arybo comme point central

### 3 Expérimentations

Nous avons testé notre approche sur la protection Tigress<sup>6</sup>. L'équipe de Tigress a mis à disposition une série de challenges avec différents niveaux de difficulté<sup>7</sup>.

<sup>6</sup> <http://tigress.cs.arizona.edu>

<sup>7</sup> <http://tigress.cs.arizona.edu/challenges.html#current>

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

**Fig. 11.** Challenges Tigress

Chaque challenge contient 5 machines virtuelles différentes. Nous nous sommes concentrés sur les challenges 0000 à 0004 ce qui fait un total de 25 machines virtuelles (dont un challenge contenant deux niveaux de virtualisation). Les challenges 0005 et 0006 n'ont pas été analysés dû à la protection de *jitting* qui est hors de la portée de cet article<sup>8</sup>.

Chaque challenge virtualise une fonction de hachage et le but est de retrouver l'algorithme d'origine (algorithme propriétaire). Nous avons donc créé un script générique<sup>9</sup> pour tous les challenges en appliquant notre approche expliquée précédemment afin de reconstruire tous les challenges sans leurs protections. Le déroulement chronologique de l'analyse est le suivant :

1. Teinte des entrées de la fonction virtualisée (voir section 2.2)
2. Émulation symbolique des binaires obfusqués (voir section 2.3)
3. Concrétisation basée sur la teinte (voir paragraphe « Politique de concrétisation basée sur l'analyse de teinte » dans la section 2.3)
4. Couverture de code s'il existe des branches (voir 2.4)
5. Conversion vers Arybo puis LLVM (voir section 2.5)
6. Optimisation LLVM -O2 pour le *path merging* (voir paragraphe « Fusion des traces » dans la section 2.4)
7. Reconstruction du binaire désobfusqué
8. Apéro

<sup>8</sup> en vérité, nous n'avons pas regardé ces challenges car, étant donné qu'il faut émuler les binaires ciblés et que ceux utilisant le *jit* utilisaient des fonctions de la *libc* pénibles à émuler, nous n'avons pas poussé :)

<sup>9</sup> [http://github.com/Jonathandesobfuscation-binaire/Tigress\\_protection/blob/master/solve-vm.py](http://github.com/Jonathandesobfuscation-binaire/Tigress_protection/blob/master/solve-vm.py)

Une fois les binaires désobfusqués, il nous a fallu vérifier que l'algorithme de transformation obtenu reste correct pour toute entrée entière. Pour cela nous avons utilisé une batterie de tests qui exécute nos deux binaires (celui obfusqué et celui désobfusqué) avec des entrées aléatoires afin de vérifier que les sorties sont identiques entre elles. La figure 12 illustre les résultats obtenus en se focalisant sur une seule exécution. Toutes les cases orange signifient donc qu'il existe une branche non couverte lors de l'extraction de l'algorithme de hachage.

Une fois la couverture de code appliquée pour récupérer la sémantique complète de la fonction virtualisée (voir section 2.4), nous obtenons les résultats illustrés par la figure 13.

Deux challenges n'ont pas pu être reconstruits dû au fait qu'il y avait des boucles sur l'entrée utilisateur (ce qui fait actuellement partie de nos limites expliquées dans la section 4).

Les analyses ont été faites sur un ordinateur portable de bureau (i7-6560U, 16Go de RAM, disque SSD). Les temps d'analyse, ainsi que la quantité de RAM consommée pour une seule trace d'exécution, sont illustrés par la figure 14. Nous pouvons constater que, pour la plupart des machines virtuelles ne contenant qu'un seul niveau de virtualisation, le temps d'analyse ainsi que la RAM consommée sont tout à fait raisonnables.

La protection demandant le plus de ressources est celle contenant deux niveaux de virtualisation (VM 4). Pour donner une idée, le challenge `VM-4-chall-3` avait 44 millions d'instructions exécutées par trace, soit environ 140 millions d'expressions symboliques et environ 1 milliard de nœuds dans notre représentation sous forme d'AST. Pour pouvoir résoudre cette série, nous avons été contraints d'installer une distribution sur un disque SSD avec une grande partition de SWAP (800Go de swap SSD...).

*Conclusion* La question maintenant est : *est-ce que 170Go de RAM et  $\leq 3h$  d'analyse sont vraiment aberrants pour une entité (ou même un individu) voulant vraiment casser ce genre de protection ?*

Toute cette analyse a été rendue publique<sup>10</sup>. On peut y trouver le script de désobfuscation<sup>11</sup>, ainsi que les binaires désobfusqués<sup>12</sup>, les expressions

<sup>10</sup> [http://github.com/Jonathandesobfuscation-binaire/Tigress\\_protection](http://github.com/Jonathandesobfuscation-binaire/Tigress_protection)

<sup>11</sup> [http://github.com/Jonathandesobfuscation-binaire/Tigress\\_protection/blob/master/solve-vm.py](http://github.com/Jonathandesobfuscation-binaire/Tigress_protection/blob/master/solve-vm.py)

<sup>12</sup> [http://github.com/Jonathandesobfuscation-binaire/Tigress\\_protection/tree/master/deobfuscated\\_binaries](http://github.com/Jonathandesobfuscation-binaire/Tigress_protection/tree/master/deobfuscated_binaries)

	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	100.00%	100.00%	34.70%	100.00%	89.60%
VM 1	100.00%	62.55%	100.00%	100.00%	100.00%
VM 2	53.83%	70.25%	100.00%	76.55%	100.00%
VM 3	100.00%	26.35%	92.12%	100.00%	100.00%
VM 4	97.90%	100.00%	79.62%	100.00%	100.00%
VM 5	not analyzed	not analyzed	not analyzed	not analyzed	not analyzed
VM 6	not analyzed	not analyzed	not analyzed	not analyzed	not analyzed
<b>F</b>	Full expressions of the hash algorithm extracted <b>with</b> 100.00% of success				
<b>P</b>	Partial expressions of the hash algorithm extracted <b>without</b> 100.00% of success				

Fig. 12. Résultat avec une seule exécution

	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	100.00%	100.00%	loop on input	100.00%	100.00%
VM 1	100.00%	100.00%	100.00%	100.00%	100.00%
VM 2	loop on input	100.00%	100.00%	100.00%	100.00%
VM 3	100.00%	100.00%	100.00%	100.00%	100.00%
VM 4	100.00%	100.00%	100.00%	100.00%	100.00%
VM 5	not analyzed	not analyzed	not analyzed	not analyzed	not analyzed
VM 6	not analyzed	not analyzed	not analyzed	not analyzed	not analyzed
<b>F</b>	Full expressions of the hash algorithm extracted <b>with</b> 100.00% of success				
<b>P</b>	Partial expressions of the hash algorithm extracted <b>without</b> 100.00% of success. Loops on input are not trivial to reconstruct – we need more time to work on it.				

Fig. 13. Résultat avec deux exécutions

	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	3.85 seconds	9.20 seconds	3.27 seconds	4.26 seconds	1.58 seconds
VM 1	1.26 seconds	1.42 seconds	3.27 seconds	2.49 seconds	1.74 seconds
VM 2	6.58 seconds	2.02 seconds	2.63 seconds	4.85 seconds	3.82 seconds
VM 3	45.59 seconds	11.30 seconds	8.84 seconds	4.84 seconds	21.64 seconds
VM 4	361 seconds	315 seconds	588 seconds	8040 seconds	1680 seconds
	Few seconds to extract the equation and less than 200 MB of RAM used				
	Few minutes to extract the equation and ~4 GB of RAM used				
	Few minutes to extract the equation and ~5 GB of RAM used				
	Few minutes to extract the equation and ~9 GB of RAM used				
	Few minutes to extract the equation and ~21 GB of RAM used				
	Few hours to extract the equation and ~170 GB of RAM used				

Fig. 14. Temps d'analyse et RAM consommée par exécution

LLVM<sup>13</sup> et les expressions Triton<sup>14</sup>. L'équipe de Tigress a été contactée et il semblerait qu'ils étaient au courant qu'avec une exécution symbolique il était possible de casser certaines de leurs protections. Quelques semaines après notre contact avec eux, ils publient une nouvelle protection anti exécution symbolique [2] à ACSAC'16 et nous annoncent qu'il va y avoir une nouvelle série de challenges.

## 4 Limitations

Bien que cette analyse ait été un franc succès sur la protection Tigress, nous arrivons vite aux limites connues de l'exécution symbolique qui sont l'explosion de chemins ainsi que la complexité des expressions. Les limites sont en réalité fixées par le contenu virtualisé plus que la protection en elle-même.

*Exécution symbolique* : En effet, si nous virtualisons un programme complexe, sa reconstruction sera d'autant plus complexe du fait qu'il faut arriver à couvrir l'ensemble de ses chemins (sans parler du fait que les programmes virtualisés peuvent être asynchrones, multi-threadés, contenant des IPC, etc.). On tient également à souligner que dans le cas de ces challenges, il s'agissait de reconstruire une simple transition  $f(x) \rightsquigarrow x'$  qui se concrétise par une seule paire entrée sortie et donc une simple expression. Dans le cas de fonctions virtualisées complexes, nous devons traiter un grand nombre d'entrées et de sorties ainsi que des CFG conséquents.

*Propagation de la teinte* : Actuellement, le moteur de teinte ne propage pas cette dernière en prenant en compte le CFG. Ce qui, dans certains cas, implique une perte de l'entrée utilisateur et donc une concrétisation d'expressions qui n'aurait pas lieu d'être.

*Structure d'exécution (les boucles)* : C'est d'autant plus complexe que les boucles sont difficiles à reconstruire du fait que nous travaillons sur une trace d'exécution et que les boucles sont déroulées (sans parler des boucles dans des boucles).

---

<sup>13</sup> [http://github.com/Jonathandesobfuscation-binaire/Tigress\\_protection/tree/master/llvm\\_expressions](http://github.com/Jonathandesobfuscation-binaire/Tigress_protection/tree/master/llvm_expressions)

<sup>14</sup> [http://github.com/Jonathandesobfuscation-binaire/Tigress\\_protection/tree/master/symbolic\\_expressions](http://github.com/Jonathandesobfuscation-binaire/Tigress_protection/tree/master/symbolic_expressions)

*Outils* : Notons que les limites peuvent également être fixées par Triton lui-même. Par exemple, notre représentation sémantique d'une trace d'exécution ne fait pas de distinction entre des données intra et interprocédurales ce qui ne nous permet pas de reconstruire des appels de sous-fonctions. Dans notre modèle, toute sous-fonction est gérée comme étant une seule et même fonction. Triton ne supporte pas encore les flottants et consomme encore un peu trop de RAM. Bref, des limites dans Triton comme dans n'importe quel autre projet peuvent être listées sans fin, c'est pourquoi nous privilégions les contributions plutôt que les trolls :).

## 5 Conclusion

Nous avons exposé les principes de la virtualisation comme protection binaire et donné notre approche pour essayer de reconstruire au plus proche le code d'origine. Nous avons expérimenté l'approche sur 25 machines virtuelles utilisant la protection Tigress. L'approche repose sur une analyse de teinte pour isoler les instructions pertinentes, l'exécution symbolique pour donner un sens à l'interaction entre ces instructions teintées et LLVM pour la reconstruction d'un code désobfusqué valide. Sur Tigress, les résultats sont plutôt concluants, mais cette approche souffre de plusieurs limites fixées par le domaine de l'exécution symbolique lui-même. Nous estimons cependant que cette approche mérite d'être continuée et poussée un peu plus loin notamment sur les limites définies précédemment ainsi que sur d'autres protections.

## Remerciements

Merci à **Adrien Guinet** pour son support Arybo, **Romain Thomas** pour son idée sur l'union des traces avec l'algorithme de Levenshtein, **Marion Videau** pour sa relecture aguerrie ainsi que **Quarkslab** pour le temps accordé sur le sujet.

## Références

1. Bertrand Anckaert, Mariusz Jakubowski et Ramarathnam Venkatesan. Proteus : virtualization for diversified tamper-resistance. In *Proceedings of the ACM workshop on Digital rights management*, pages 47–58. ACM, 2006.
2. Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham et Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200. ACM, 2016.

3. Cristian Cadar et Koushik Sen. Symbolic execution for software testing : three decades later. *Commun. ACM*, 56(2) :82–90, 2013.
4. James Clause, Wanchun Li et Alessandro Orso. Dytan : a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.
5. Kevin Coogan, Gen Lu et Saumya Debray. Deobfuscation of virtualization-obfuscated software : a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 275–284. ACM, 2011.
6. Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta et Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis*, ISSTA 2016. ACM, 2016.
7. Ninon Eyrolles, Adrien Guinet et Marion Videau. Arybo : Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions. In *Arybo : Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions*. GreHack, France, Grenoble, 2016.
8. Sudeep Ghosh, Jason D Hiser et Jack W Davidson. A secure and robust approach to software tamper resistance. In *International Workshop on Information Hiding*, pages 33–47. Springer, 2010.
9. Patrice Godefroid, Jonathan de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann et Michael Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5) :30–37, 2008.
10. Patrice Godefroid, Nils Klarlund et Koushik Sen. DART : directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.
11. Patrice Godefroid, Michael Y. Levin et David A. Molnar. SAGE : whitebox fuzzing for security testing. *Commun. ACM*, 55(3) :40–44, 2012.
12. Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 61–70. IEEE, 2012.
13. James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976.
14. Chris Lattner et Vikram Adve. LLVM : A compilation framework for lifelong program analysis and transformation. In *LLVM : A Compilation Framework for Lifelong Program Analysis and Transformation*, pages 75–88, San Jose, CA, USA, Mar 2004.
15. Jasvir Nagra et Christian Collberg. *Surreptitious software : obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.
16. Rolf Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.
17. Florent Saudel et Jonathan Salwan. Triton : A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.
18. Edward J. Schwartz, Thanassis Avgerinos et David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, pages 317–331, 2010.

19. Koushik Sen, Darko Marinov et Gul Agha. CUTE : a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.
20. Babak Yadegari, Brian Johannsmeyer, Ben Whitely et Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 674–691. IEEE, 2015.