Recompilation dynamique de codes binaires hostiles

Sébastien Josse

DGA Maîtrise de l'Information

Résumé Les *malware* sont de plus en plus difficiles à analyser, par utilisation des outils conventionnels d'analyse statique et dynamique, dans la mesure où ils s'appuient sur des outils spécialisés du marché pour protéger leur code. Nous présentons dans ce papier un outil adapté à l'analyse de programmes binaires protégés et potentiellement hostiles. Cet outil met en œuvre un émulateur et plusieurs fonctions d'analyse spécialisées pour observer le programme cible et son environnement d'exécution, puis extraire et simplifier sa représentation.

1 Introduction

Les *malware* sont de plus en plus difficiles à analyser, par utilisation des outils conventionnels d'analyse statique et dynamique. Compte tenu du nombre de systèmes d'exploitation et architectures CPU sous-jacentes ciblées par les *malware*, nous pouvons constater l'absence de solution d'analyse multi plate-forme, suffisamment puissante pour faire face aux mécanismes de protection logicielle actuels (code machine auto-modifiant, transformations d'obfuscation fondées sur la virtualisation de code, etc). Les auteurs de *malware* ciblent un large éventail de systèmes d'exploitation et exploitent les progrès réalisés dans le domaine de la protection logicielle et mis à disposition par des produits spécialisés du marché. En outre, en raison de la nature intrinsèquement hostile des *malware*, les analystes ont besoin d'un environnement d'analyse sûr et contrôlé.

Les outils actuels d'analyse statique et dynamique souffrent de certaines limitations lorsqu'ils s'appliquent à l'analyse de *malware*. Il existe pourtant de nombreux outils intéressants pour différents types d'analyse de code, y compris l'analyse de code binaire. Malheureusement, ils viennent souvent avec leur propre représentation intermédiaire (IR) : VEX pour Valgrind [16], VEX/Vine pour BitBlaze [21,4], l'IR d'IDA Pro pour CodeSurfer [24,1], REIL pour BinNavi [7]. En outre, nombre d'entre eux ne sont pas appropriés pour l'analyse de code hostile ou protégé. Lorsqu'ils le sont (TTAnalyze [2], Argos [18], Renovo [13]), ils ne proposent pas de fonctionnalité de réécriture de binaire, fonctionnalité pourtant très utile lorsqu'il s'agit d'analyser des exécutables protégés.

Nous présentons dans ce papier les bases d'un outil de réécriture de binaire (VxStripper), dont la vocation est de traiter les mécanismes de protection logicielle mis en œuvre par les *malware*, dans un environnement contrôlé. Il est conçu pour extraire dynamiquement une représentation intermédiaire d'un binaire et toutes les informations nécessaires pour appliquer certaines simplifications, rendant son fonctionnement interne plus facile à comprendre pour l'analyste.

L'une des principales motivations derrière les choix de conception et de mise en œuvre de notre outil est de contourner les limitations des solutions actuelles d'analyse de *malware* et de programmes binaires. Le but est d'obtenir autant d'informations que possible à partir d'un programme binaire qui utilise toutes les techniques et les outils disponibles pour protéger cette information. L'idée est d'instrumenter le CPU virtuel et le système d'exploitation invité d'une manière non intrusive, pour obtenir dynamiquement les informations nécessaires à la reconstruction du programme et à la simplification de sa représentation.

Cet outil est fondé sur le moteur de traduction dynamique binaire de QEMU et sur la chaîne de compilation LLVM.

LLVM (*Low Level Virtual Machine* [14]) est une chaîne de compilation proposant un nombre conséquent d'optimisations, qui peuvent être appliquées à toutes les étapes du cycle de vie d'un programme. LLVM utilise un jeu d'instructions fortement typé de type RISC et une représentation SSA (*Static Single Assignement*. Dans cette représentation, chaque registre est attribué une seule fois). LLVM supporte un grand nombre de *back-ends* binaires (x86, x86-64, SPARC, PowerPC, ARM, MIPS, CellSPU, XCore, MSP430, MicroBlaze, PTX) et quelques *back-ends* de code source (C, C++).

Le moteur de traduction dynamique binaire (DBT) de QEMU (*Quick EMUlator*, [3]) est utilisé pour traduire dynamiquement le code binaire de l'architecture du processeur invité vers l'architecture du processeur hôte, grâce à l'utilisation d'une représentation intermédiaire appelée TCG (*Tiny Code Generator*, [23]). Ce langage consiste en de simples instructions de type RISC, appelées micro-opérations. La traduction binaire comporte deux étapes : le code cible est d'abord traduit en séquences d'instructions TCG, appelées *translation blocks* (DBT *front-end*). Ensuite, ces *translations blocks* sont convertis en code exécutable par le processeur hôte (DBT *back-end*). Le DBT de QEMU supporte un grand nombre *front-ends* binaires (x86, x86-64, ARM, ETRAX CRIS, MIPS, MicroBlaze, PowerPC, SH4, SPARC).

Notre outil hérite de QEMU les nombreux *front-ends* binaires et de LLVM les nombreux *back-ends*, offrant à un coût raisonnable un framework complet de réécriture de binaire. Les fonctions de réécriture sont mises en œuvre sous la forme de passes LLVM. Sa conception actuelle s'appuie sur les travaux déjà menés pour convertir l'IR TCG vers l'IR LLVM (LLVM-QEMU [20] et S2E [6]), ainsi que sur les algorithmes de conception présentés dans [10,11] et [12]. Ce papier complète cette documentation par une description des nouvelles fonctionnalités et des principales évolutions de cet outil. Notre ambition est que cet outil soit en mesure de collaborer avec les nombreux outils d'analyse de logiciels fondés sur la chaîne de compilation LLVM.

L'originalité de notre approche réside dans la définition d'un outil d'analyse de code binaire spécialement conçu pour résoudre le problème des programmes hostiles et automatiser les tâches souvent fastidieuses et répétitives conduites par un analyste.

Cette chaîne de recompilation s'appuie sur une architecture modulaire et évolutive, permettant d'appliquer les mêmes transformations pour une grande variété d'architectures logicielles et matérielles. Elle est fondée sur une chaîne de compilation moderne, proposant une représentation intermédiaire et des fonctionnalités efficaces. La représentation LLVM a en outre fait l'objet de travaux visant à se doter d'outils formels pour raisonner sur des transformations qui opèrent sur cette représentation intermédiaire. Vellvm (*Verified LLVM* [25]) pourrait nous permettre à terme d'extraire des implémentations vérifiées formellement des passes de désobfuscation implémentées dans notre outil.

Ces choix de conception nous conduisent à l'exploration de nouvelles méthodes d'extraction d'information et d'analyse dynamique de programme. Parmi les techniques qui ne sont pas à notre connaissance décrites dans la littérature, nous trouvons notamment :

- L'extraction dynamique et la reconstruction de l'information de relocation d'un programme binaire, indispensable à la conversion de sa représentation vers la forme SSA de LLVM.
- L'ensemble des techniques utilisées pour projeter la représentation LLVM des *translation blocks*, générée dynamiquement par le *back-end* LLVM de TCG, vers le processeur hôte. En d'autres termes, la méthode utilisée pour « sortir » la représentation intermédiaire du programme de la machine virtuelle et la « projeter » sur la machine hôte.

– Les passes de réécriture de la représentation intermédiaire LLVM utilisées pour débarrasser le programme de ses protections et simplifier sa représentation.

Sur ce dernier point, nous constatons que l'application conjointe de l'évaluation partielle induite par la traduction dynamique du code cible vers la représentation LLVM et l'application des transformations d'optimisation statiques proposées par cette chaîne de compilation suffisent à débarrasser le programme de nombre de ses obfuscations. Ce résultat nous encourage à poursuivre la voie d'étude de méthodes génériques de désobfuscation, pouvant s'appliquer de manière automatique et sans faire d'hypothèse concernant le(s) mécanisme(s) de protection utilisé(s).

La suite du papier est organisée de la manière suivante.
La section 2 présente la conception de notre outil, et ses deux modules d'analyse les plus importants :

– Le module VxUnpack localise le point d'entrée d'origine (OEP) de l'exécutable cible, récupère les informations relatives à ses interactions avec l'API du système d'exploitation et extrait les informations de relocation.
– Le module VxNorm s'appuie sur la représentation intermédiaire LLVM et une évaluation partielle et des transformations de désobfuscation pour normaliser le programme cible vers une représentation plus adaptée à d'autres analyses, comme la décompilation.

La section 3 présente la stratégie développée pour valider l'efficacité de notre outil, les résultats préliminaires et les moyens d'amélioration de sa conception et de sa mise en œuvre.
La section 4 présente les travaux futurs envisagés et conclut cet article.

2 Conception

2.1 Extension du DBT de QEMU

Avant de présenter l'architecture de notre outil, nous allons voir de quelle manière nous avons modifié le fonctionnement du CPU logiciel de QEMU pour permettre l'invocation systématique de notre fonction d'instrumentation et traduire la représentation intermédiaire TCG vers la représentation LLVM.

QEMU est un émulateur de PC utilisant la translation binaire dynamique : le code écrit pour un jeu d'instructions CPU est traduit à la volée en un code pour un autre jeu d'instructions CPU. Nous obtenons une exécution plus rapide que l'émulation simple en utilisant un cache :

l'idée est de traduire un morceau de code, de le mettre en cache, et de le réutiliser si besoin est. Pour accélérer encore l'exécution du CPU logiciel (VPU), ces blocs sont chaînés. Le principal intérêt d'un émulateur de PC fondé sur la traduction binaire dynamique est sa rapidité d'émulation.

Pour chaque processeur émulé par QEMU, la traduction suivante est effectuée : l'ensemble des instructions cible est traduit en une représentation intermédiaire (micro opérations TCG) elle même traduite dans le Jeu d'instruction hôte. Dans QEMU, la représentation intermédiaire est indépendante du jeu d'instruction hôte. Le moteur de traduction binaire dynamique se fonde sur cette représentation. Il est pour cette raison dit portable.

Nous avons vu que le moteur DBT de QEMU effectue la traduction dynamique du code binaire de l'architecture du processeur invité vers l'architecture du processeur hôte en utilisant la représentation intermédiaire TCG.

Voyons sur un exemple simple à quoi ressemble le langage TCG. Considérons l'instruction :

```
0x0040104c:  push   0xa
```

Cette instruction est traduite de la manière suivante dans la représentation TCG de QEMU :

```
(i) movi_i32 tmp0,$0xa
(ii) mov_i32 tmp2,esp
(iii) movi_i32 tmp13,$0xfffffffc
(iv) add_i32 tmp2,tmp2,tmp13
(v) qemu_st32 tmp0,tmp2,$0x1
(vi) mov_i32 esp,tmp2
(vii) movi_i32 tmp4,$0x40104e
(viii) st_i32 tmp4,env,$0x30
(ix) exit_tb $0x0
```

Ce bloc d'instructions TCG émule l'exécution de l'instruction `push` sur le CPU logiciel. Les opérations effectuées sont les suivantes :

L'entier 0xa est stocké dans la variable `tmp0` (i). Cette variable est ensuite sauvegardée sur la pile (ii-vi). L'adresse de d'instruction suivant l'instruction courante est stockée dans `tmp4` (vii) puis sauvegardée dans le registre `cc_op` du VPU (viii). L'instruction (ix) indique la fin du bloc TCG.

Notre outil modifie le mécanisme de traduction binaire dynamique de telle sorte que la fonction d'instrumentation du CPU logiciel soit invoquée systématiquement avant l'exécution d'un *translation block*. Pour y parvenir, nous ajoutons une micro opération supplémentaire qui prends

en opérande l'adresse de la fonction d'instrumentation dans le code de VxStripper. Le code TCG obtenu est le suivant :

```
(i) op_callback @vxs_callback
(ii) movi_i32 tmp0,$0xa
(iii) mov_i32 tmp2,esp
(iv) movi_i32 tmp13,$0xfffffffc
(v) add_i32 tmp2,tmp2,tmp13
(vi) qemu_st32 tmp0,tmp2,$0x1
(vii) mov_i32 esp,tmp2
(viii) movi_i32 tmp4,$0x40104e
(ix) st_i32 tmp4,env,$0x30
(x) exit_tb $0x0
```

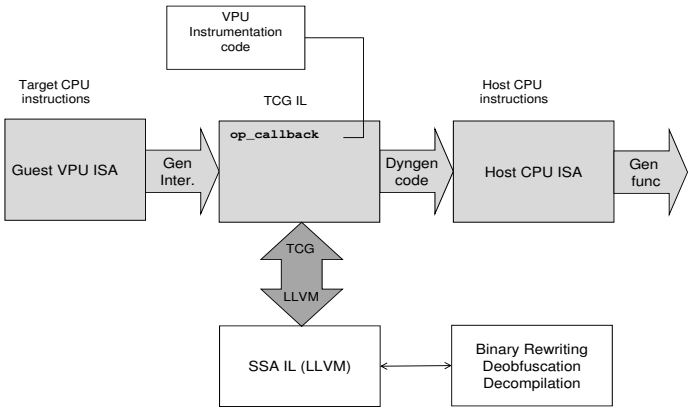

FIGURE 1. Extension du DBT de QEMU

Ce mécanisme nous permet d'exécuter notre code d'instrumentation à chaque cycle du CPU logiciel. Ayant accès aux registres du VPU et à la mémoire du PC logiciel, nous sommes en mesure d'acquérir le contexte d'un processus, et d'extraire l'information d'interaction de ce processus avec les APIs du système d'exploitation invité.

En instrumentant également les instructions TCG *load* et *store*, nous allons pouvoir extraire également une information concernant les interactions du processus cible avec la mémoire du système invité. Cette information va nous permettre de recouvrer l'information de relocation du processus.

Maintenant que nous avons vu comment modifier le fonctionnement du CPU logiciel de QEMU pour permettre l'invocation systématique de

notre fonction d'instrumentation, examinons la traduction de la représentation intermédiaire TCG vers la représentation LLVM. Le résultat de la traduction du bloc TCG précédent est le suivant :

```
(1) %esp_v.i = load i32* @esp_ptr
(2) %tmp2_v.i = add i32 %esp_v.i, -4
(3) %4 = inttoptr i32 %tmp2_v.i to i32*
(4) store i32 10, i32* %4
(5) store i32 %tmp2_v.i, i32* @esp_ptr
(6) store i32 4198478, i32* %next.i
(7) store i32 0, i32* %ret.i
```

L'entier `0xa` est stocké à l'adresse pointée par la variable `%4`, ce qui revient à le sauvegarder sur la pile (1-4). L'adresse de d'instruction suivant l'instruction courante est stockée dans la variable `%next.i` (6). L'instruction (7) termine le bloc LLVM.

Pour information, à l'issue du processus de normalisation, ce bloc LLVM est compilé vers le code assembleur :

```
      401269 !   mov     dword ptr [esp-14h], 0ah
```

Maintenant que nous avons donné un aperçu des principales modifications réalisées sur l'émulateur QEMU, présentées de manière schématique (figure 1), voyons l'architecture générale de l'outil.

2.2 Architecture de l'outil

Notre outil met en œuvre un moteur DBT étendu et plusieurs fonctions d'analyse spécialisées (figure 2), pour observer le programme cible et son environnement d'exécution.
Un gestionnaire de modules coordonne l'activation et la collaboration entre ces fonctions d'analyse, implémentées sous forme de plugins.

Ces fonctions d'analyse ont vocation à extraire une information sémantique du programme cible. Il peut s'agir de la trace de ses interactions avec les APIs du système d'exploitation invité, de la manière dont il modifie les objets et structures de l'exécutif ou du noyau du système d'exploitation invité, ou plus simplement de la trace de son code machine.

L'extraction de cette information s'appuie sur une description du système d'exploitation invité, qui peut être fournie par exemple par un serveur de symboles, comme c'est le cas pour la famille de systèmes d'exploitation Windows.

Parmi les modules déjà implémentés, on trouve notamment :

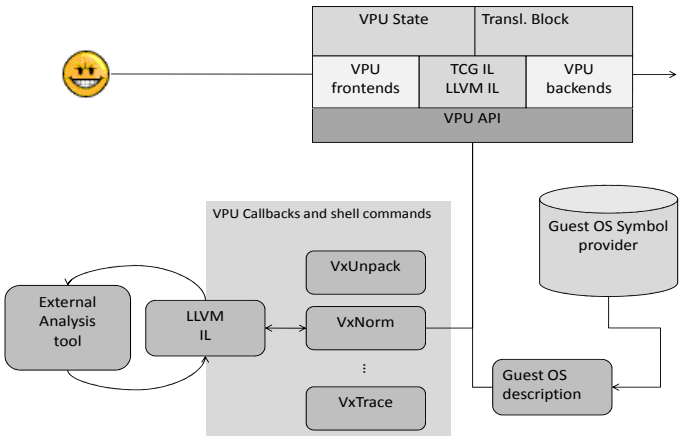

FIGURE 2. Architecture de VxStripper

- Un module d'API *hooking* (VxTrace)
- Un module d'analyse forensique (VxR00t)
- Un module d'*unpacking* (VxUnpack)
- Un module de normalisation (VxNorm)

2.3 API Hooking

Le module d'API *hooking* de notre outil est fondé sur l'analyse forensique de la mémoire du système d'exploitation invité, sans aucune interaction avec ce dernier. La localisation des bibliothèques et des fonctions d'API se fait en parcourant les structures de l'exécutif Windows qui sont utilisées pour représenter un processus.

Les méthodes utilisées pour retrouver et instrumenter dynamiquement les appels aux API Windows sont décrites plus en détail dans [2] et [10].

2.4 Analyse forensique / analyse de rootkit

Le module forensique de notre outil vient avec des fonctionnalités supplémentaires pour surveiller et vérifier l'intégrité de nombreux endroits au sein de la plate-forme invitée où un *hook* peut être installé. Il parcourt les structures exécutives du système d'exploitation afin d'identifier des cibles potentielles d'une attaque *rootkit* et surveille les composants matériels qui pourraient être corrompus par un *rootkit*. Cette information

est cruciale pour l'analyste pour mieux comprendre des attaques virales bas niveau. Les détails de conception et de mise en œuvre du module forensique de notre outil sont donnés dans [11].

Jusqu'à présent, ces caractéristiques sont celles que l'on peut attendre d'un débogueur du noyau. On peut s'attacher à un processus, inspecter l'état du processeur et le code désassemblé, et tracer les interactions du programme cible avec les API du système d'exploitation. Cette inspection est faite dans un environnement sûr et contrôlé, sans aucune interaction intrusive avec le système d'exploitation invité.

Voyons à présent plus en détail le fonctionnement de ses deux modules d'analyse les plus importants : le module d'*unpacking* (VxUnpack) et le module de normalisation (VxNorm).

2.5 Unpacking

Le module VxUnpack localise le point d'entrée d'origine (OEP) de l'exécutable cible, récupère les informations relatives à ses interactions avec l'API du système d'exploitation et extrait les informations de relocation.

L'algorithme utilisé est décrit dans [10]. L'idée sous-jacente est un simple contrôle d'intégrité du code exécutable du programme cible : pour chaque *translation block* du programme, une comparaison entre sa valeur en mémoire virtuelle et sa valeur sur le système de fichiers hôte est effectuée. Tant que les valeurs sont identiques, rien n'est fait. Dès lors qu'une différence est identifiée, le *translation block* courant est écrit dans le fichier à la place de l'ancien *translation block*. La première instruction du *translation block* nouvellement généré est identifiée comme l'OEP du programme protégé. À la fin de l'analyse, les sections de données sont écrites dans le fichier brut à la place des sections de données d'origine.

Le même algorithme de contrôle est appliqué pour chaque *translation block*. Le chargeur de la protection de l'exécutable packé peut avoir plusieurs couches de déchiffrement.

Dès que le dernier *translation block* déchiffré a été atteint, la seule chose qui reste à faire est de réparer le fichier exécutable cible. Afin de récupérer la structure PE (*Portable Executable* [15,17]) d'un exécutable déprotégé, plusieurs tâches doivent être effectuées : définir le point d'entrée d'origine, reconstruire les tables d'imports et de relocations, puis vérifier la cohérence de l'en-tête PE.

La méthode utilisée par notre moteur d'*unpacking* pour reconstruire l'information d'imports est basée sur l'instrumentation des API natives et Win32. Durant le processus d'*unpacking*, tous les appels aux API sont

tracés. Une table triée [1] des appels d'API est initialisée au moment du chargement de l'exécutable, en parcourant les structures de l'exécutif NT.

Ensuite, après reprise de l'exécution du processus, chaque appel d'API est tracé. Ce tableau est mis à jour régulièrement au cours de l'exécution du processus cible, et est utilisé pour résoudre dynamiquement les noms des fonctions d'API. Enfin, après avoir effectué un *dump* de l'espace mémoire du processus cible, ce tableau est utilisé pour fixer la table d'imports dans l'exécutable PE.

L'instrumentation des instructions TCG *load* et *store* nous permet également d'extraire dynamiquement l'information de relocation du programme. Cette information peut également être ajoutée dans une nouvelle section de l'exécutable. Voici à titre d'illustration l'information (utile) extraite lors de l'*unpacking* d'un programme affichant une boîte de dialogue (fonction `MessageBoxA`) :

```
;; [INFO] eip=0x00401000
[RELOC] pc=0x00401002 value=0x00403000 size=0x00000005 va=0x00401003
[RELOC] pc=0x00401007 value=0x0040300f size=0x00000005 va=0x00401008
[RELOC] pc=0x0040100e value=0x00402008 size=0x00000006 va=0x00401010
[APICALL] api_pc=0x77d8050b api_oep=0x77d8050b dll_name=C:\WINDOWS\
    system32\user32.dll func_name=MessageBoxA reloc->va=0x00401010 reloc
    ->value=0x00402008
```

L'information de relocation est constituée des couples (`va`, `value`), donnant respectivement l'adresse virtuelle et la valeur à reloger. Nous pouvons observer que pour ce *packer*, le prologue de la fonction `MessageBoxA` n'est pas émulé par la protection. Dans le cas contraire, l'adresse externe effectivement appelée (`api_pc`) est différente du point d'entrée de la fonction d'API (`api_oep`).

2.6 Normalisation

Désormais, nous sommes en mesure d'obtenir automatiquement des informations sur un *malware* packé. Dans de nombreux cas, nous sommes même en mesure d'obtenir un binaire débarrassé de son chargeur de protection et ne comportant aucun code réinscriptible. Malheureusement, certains mécanismes d'obfuscation (aplatissement du flot de contrôle,

1. La table des imports est triée afin de contourner certaines protections qui émulent les premiers octets des fonctions d'API et donc ne se branchent pas au point d'entrée d'origine des fonctions d'API. En maintenant une table triée des fonctions de l'API, les fonctions d'API ne sont pas indexées par leur point d'entrée, mais par une plage mémoire. Nous sommes maintenant en mesure de tracer les appels d'API, même si leurs premiers octets sont volés (déplacés dans le code de la protection).

transformations d'obfuscation fondées sur la virtualisation de code, etc.) doivent maintenant être traitées, afin de faciliter la compréhension du fonctionnement interne d'un *malware*.

Une première tentative pour apporter une solution à ces problèmes a été mise en œuvre dans notre outil, grâce à l'utilisation de la représentation intermédiaire LLVM. Plutôt que d'essayer de travailler sur le binaire après que son image mémoire ait été dumpée, l'idée est de travailler sur sa représentation intermédiaire et d'augmenter la quantité d'information (qui a été collectée dynamiquement) en embarquant cette information dans le module LLVM. Une telle représentation est plus appropriée pour une analyse plus approfondie. Le module Vxnorm utilise la sortie des analyses précédentes pour générer un module LLVM, sur lequel plusieurs transformations d'optimisations sont appliquées. Examinons cela plus en détails.

Durant l'exécution du programme cible, le *back-end* LLVM de TCG génère la représentation LLVM des *translation blocks*. Ce code LLVM est lié à un module LLVM d'initialisation (figure 3).

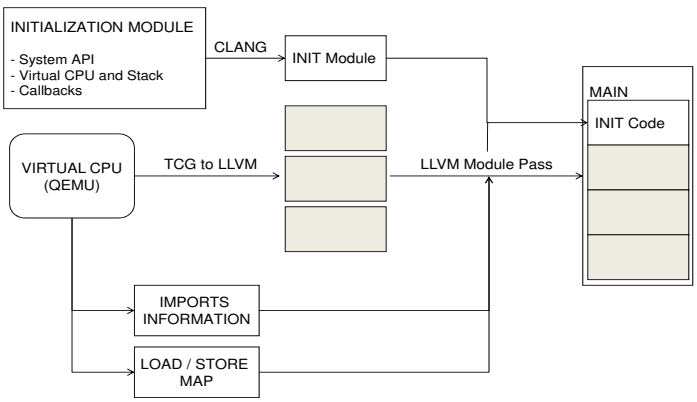

Figure 3. Module de normalisation

Ce module d'initialisation implémente les *callbacks load* et *store*, déclare les prototypes des API système et définit un processeur virtuel et sa pile.

Le module VxNorm utilise l'information collectée dynamiquement lors de l'exécution du programme cible pour résoudre les imports, les relocations du processus, et pour récupérer des sections de données.

L'information de la table d'imports est utilisée pour construire des instructions LLVM d'appels aux APIs. La *map* mémoire des *load / store* est utilisée pour appliquer les relocations et injecter les données du programme cible dans le module LLVM.

Une fois le module LLVM reconstruit, des passes d'optimisations supplémentaires sont appliquées sur sa représentation.

Le module LLVM peut ensuite être compilé vers l'architecture de son choix, en utilisant l'un des *back-ends* LLVM disponible. Il peut également être traduit vers du code C ou C++.

3 Discussion et évaluation

Nous présentons dans cette section la stratégie développée pour valider la sécurité et l'efficacité de notre outil, les résultats préliminaires et nous discutons les limites de l'implémentation actuelle et les moyens d'amélioration de sa conception et de son implémentation.

3.1 Les exigences de sécurité

La principale exigence de sécurité pour un outil d'analyse de *malware* est l'isolement (entrave de toute propagation). Une autre exigence d'un tel outil est que l'information recherchée puisse effectivement être obtenue par l'analyse. En particulier, l'analyse doit être furtive car si l'émulation est détectée [19,8], l'exécutable cible ne fournira plus l'information prévu sur son fonctionnement interne. La conception de notre outil est guidée par ces deux exigences de sécurité.

Pour atteindre ces objectifs, l'implémentation du moteur d'émulation a été modifiée pour rendre l'émulation du matériel plus précise et donc plus difficiles à détecter. Cela comprend le comportement de plusieurs instructions CPU (identification du CPU, lecture du compteur d'horloge, etc), mais aussi la façon dont il gère les fautes successives, par exemple. Toute imprécision relative aux spécifications de l'architecture CPU peut être utilisée pour détecter un environnement d'exécution émulé et faire échouer l'analyse du programme cible.

L'émulateur QEMU a été adapté afin de mettre en œuvre le moteur d'émulation qui est au cœur de notre outil. Dans sa première version [10], le système d'exploitation Windows invité intégrait un service du noyau qui communiquait via une interface réseau virtuelle avec le moniteur de notre outil. Ce canal de communication était utilisé pour télécharger

les binaires cibles dans la machine virtuelle, lancer l'exécution du programme cible principal et obtenir du noyau les informations permettant de diriger l'exécution du processus invité depuis le système hôte.

L'implémentation actuelle n'utilise plus un service noyau embarqué. Le programme cible est directement écrit sur le disque virtuel, et l'information de localisation du processus est récupérée par analyse forensique de la mémoire du système d'exploitation Windows invité, en utilisant les informations fournies par le serveur de symboles de Microsoft. Une telle méthode est probablement déjà utilisée par certains logiciels clients JTAG et par des débogueurs noyau. Nous pouvons nous attacher à un processus cible et acquérir son contexte sans aucune interaction avec le système d'exploitation invité. Nous pouvons alors diriger son exécution via le CPU virtuel.

Un service embarqué est néanmoins nécessaire pour démarrer un programme cible. Cette action peut se faire par injection de code furtif, en écrivant le code d'invocation du chargeur dans la représentation intermédiaire TCG, puis en utilisant le back-end TCG de QEMU pour démarrer (en mode suspendu) et reprendre l'exécution d'un processus cible.

3.2 Considérations de performance

Notre outil peut être utilisé en mode console, batch ou interactif, et éventuellement en mode graphique.

- Le mode *batch* écrit automatiquement le fichier exécutable cible dans la machine virtuelle, le débarrasse de son chargeur de protection et récupère les informations requises sur ses interactions avec le système d'exploitation invité. Le mode *batch* applique les options d'analyse par défaut, données dans un fichier de paramètres.
- Le mode interactif permet à l'analyste de *malware* de piloter dynamiquement l'exécution de l'exécutable cible et d'interagir avec la machine virtuelle, en contrôlant ses états. Il est principalement utilisé lorsque l'exécutable cible exécute un de ses composants comme un nouveau processus, afin d'acquérir le nouveau contexte et de suivre l'exécution du nouveau processus.
- Le programme principal peut également être utilisé en mode graphique. Ce mode est utile lorsque le processus d'analyse nécessite des interactions entre l'utilisateur et le programme cible par le biais d'une interface graphique. Les mêmes options que dans les modes console (*batch*, par défaut, ou interactif) sont disponibles lorsque l'on utilise le mode graphique.

Les performances de notre outil sont liées au type et au nombre de modules d'analyse qui sont enregistrés comme *callbacks*. Lorsque l'on a affaire à un *malware* protégé par un chargeur sécurisé implémentant des centaines de couches de déchiffrement, le coût induit par la traduction dynamique de l'IR TCG vers l'IR LLVM devient prohibitif. C'est la raison principale pour laquelle les modules VxUnpack et VxNorm de notre outils doivent être exécutés séparément. Le module VxUnpack est exécuté avec un minimum de *callbacks* d'extraction d'information activés. L'enregistrement de VxNorm et la génération de code LLVM n'intervient qu'à partir du point d'entrée d'origine supposé.

3.3 Jeu de tests

Les tests sont conduits sur un PC standard (Intel Core 2 Duo T6600, 4Go de RAM), exécutant le système d'exploitation Windows 7. L'environnement de test comprend, outre le logiciel VxStripper, une VM QEMU hébergeant le système d'exploitation Windows XP SP2, ainsi que la chaîne de compilation LLVM 3.1.

La méthodologie de test consiste à valider les principales fonctions de notre outil, par l'élaboration d'un scénario adapté. Le tableau suivant présente la stratégie et les premiers résultats obtenus. Ce jeu de test est prévu pour être appliqué à des exécutables dont nous connaissons le point d'entrée et la sémantique des interactions avec les API du système d'exploitation.

Lorsqu'il s'agit d'un programme malicieux déjà packé, nous identifions le *packer* utilisé avec PEID puis validons la sémantique des interactions avec l'OS, en analysant manuellement le programme résultant de l'exécution de VxUnpack et en comparant le résultat de l'analyse avec l'information mise à disposition par les éditeurs de logiciels antivirus.

L'évaluation des fonctionnalités de notre outil repose sur l'utilisation de plusieurs outils tiers, développés pour valider son efficacité et faciliter sa mise au point :

– un packer modulaire (VxPack), capable d'appliquer de manière sélective des mécanismes de protection de base (incluant notamment des techniques de détection anti-VM et du chiffrement multicouches) que l'on trouve dans les packers du commerce. Le fonctionnement interne du packer VxPack, fondé sur y0da's Crypter [22], est décrit dans [12]. Cet outil devrait nous permettre de tester unitairement des mécanismes de détection et protection décrits

dans la littérature mais pas nécessairement implémentés dans les *packers* du commerce.
- un outil (VxFix) qui va (sur la base de l'information collectée dynamiquement) ajouter deux sections de données supplémentaires, contenant respectivement l'information d'imports et de relocations. Pour valider la correction de l'information de relocation, il modifie l'image base puis applique les relocations. Ce même outil (VxFix) est utilisé pour afficher cette information (imports et relocations) après reconstruction.
- un outil (VxObf) proposant plusieurs transformations d'obfuscation, implémentées sous forme de passes en utilisant le gestionnaire de passes de la chaîne de compilation LLVM. Il s'agit actuellement pour l'essentiel de transformations d'obfuscation du flot de contrôle (insertions de sauts inutiles, de code inutile, aplatissement du flot de contrôle, aplatissement du flot de contrôle renforcé par l'utilisation d'une fonction de hachage [5]). Cet outil a été développé pour éprouver l'efficacité du module de normalisation et faciliter sa mise au point.

3.4 Efficacité du module d'unpacking

Fonction	Scénario	Résultat
Identification du point d'entrée d'origine	L'objectif est ici de récupérer automatiquement le point d'entrée d'origine de l'exécutable protégé. Celui-ci est journalisé.	Lorsque l'exécutable protégé ne comporte pas lui-même de code automodifiable, nous sommes en mesure de retrouver le point d'entrée d'origine.
Extraction de l'information d'imports et de relocations	L'objectif est ici de récupérer dynamiquement l'information d'imports et de relocations. Celle-ci est journalisée.	Lorsque les appels aux fonctions d'API ne sont pas complètement émulés par le chargeur de protection, nous sommes en mesure de les résoudre.

Table 1. test de VxUnpack

Pour vérifier que le module d'*unpacking* génère des résultats précis, plusieurs packers ont été utilisés avec le même programme cible. Dans

chaque cas, nous avons été en mesure de récupérer automatiquement le point d'entrée d'origine de l'exécutable protégé et la *map* mémoire du programme protégé. Les précédents *benchmarks*, qui ont été donnés dans [10], ont été améliorés par deux paramétrages :

- Limiter le nombre de *callbacks* d'extraction d'information activés.
- Activer certaines techniques d'optimisation, telle que l'instrumentation de certaines instructions de terminaison des *translation blocks* (par exemple l'instruction assembleur *REPeat string operation* [9]), qui sont souvent utilisées dans les couches de déchiffrement. Si une telle instruction de terminaison de bloc est rencontrée, tous les *callbacks* d'extraction d'information sont inhibés jusqu'au *translation block* suivant.

Avec ces réglages et avec les *packers* testés, il est possible de débarrasser le programme protégé de son chargeur de protection en moins de soixante secondes sur un ordinateur personnel standard. Observons que les *packers* utilisés lors des tests sont un peu anciens. Nous avons l'intention de compléter cette évaluation en utilisant des *packers* plus récents et en enrichissant les mécanismes de protections disponibles dans notre *packer*.

3.5 Efficacité de la reconstruction

Fonction	Scénario	Résultat
Reconstruction des sections d'imports et de relocations	Ajout des sections d'imports et de relocations. Modification de l'image base puis appliquons les relocations.	Dès lors que l'information d'imports et de relocations collectée dynamiquement est complète, nous sommes toujours en mesure de réinjecter cette information dans l'exécutable.

Table 2. test de VxUnpack avec VxFix

3.6 Efficacité du module de normalisation

L'objectif de l'ensemble de tests suivant est de valider unitairement le bon comportement du module de normalisation vis-à-vis des différents types d'appels (conventions usuelles et moins usuelles, telle que l'utilisation d'un `ret` pour invoquer une fonction d'API).

La représentation LLVM est compilée, en utilisant le back-end `llvm-ld` (ou `llc` puis `gcc`). Le code binaire résultant est édité au moyen d'un désassembleur.

Fonction	Scénario	Résultat
Reconstruction des appels aux fonctions d'API	Test des différentes conventions d'appel pour l'invocation d'une fonction d'API.	Pour les différentes manières d'invoquer une fonction d'API testées, le module de normalisation est en mesure de générer l'appel correspondant dans le module LLVM.

Table 3. test de VxNorm

Le support de plusieurs conventions d'appel est actuellement implémenté dans le module VxNorm. LLVM en supporte actuellement une dizaine, parmi lesquelles ont trouve notamment : `llvm::CallingConv::C`, `llvm::CallingConv::X86_StdCall` et `llvm::CallingConv::X86_FastCall`.

Les premiers résultats montrent que les optimisations standard utilisées en conjonction avec l'évaluation partielle induite par la traduction dynamique du code cible vers la représentation LLVM sont suffisantes pour réduire de façon drastique et simplifier le code sous analyse. Illustrons cela sur un exemple simple : considérons le programme suivant (qui affiche `y = 22`) :

```
#include <stdio.h>
int main(){
    int x = 10, y = 0, pc = 2;
    while (pc < 6)
    {
        switch(pc){
            case 2:
                y = 2; pc = 3; break;
            case 3:
                if (x > 0)
```

Fonction	Scénario	Résultat
Désobfuscation	L'objectif est la suppression des branchements inutiles. Passes utilisées : VxNorm et optimisations standard LLVM.	Ces obfuscations basiques sont défaites par le module de normalisation.
Désobfuscation	L'objectif est la suppression du code inutile. Passes utilisées : VxNorm et optimisations standard LLVM.	Ces obfuscations basiques sont défaites par le module de normalisation.
Désobfuscation	L'objectif est ici de défaire l'aplatissement du flot de contrôle. Passes utilisées : VxNorm et optimisations standard LLVM.	Ces obfuscations basiques sont défaites par le module de normalisation.

TABLE 4. test de VxNorm avec VxObf

```
                    pc = 4;
                else pc = 6;
                break;
            case 4:
                y += 2; pc = 5; break;
            case 5:
                x --; pc = 3; break;
        }
    }
    printf("y = %d\n", y);
    return 0;
}
```

Après compilation, le graphe d'un tel programme possède la propriété d'être aplati (figure 4). L'exécution de VxNorm produit (lorsque l'on

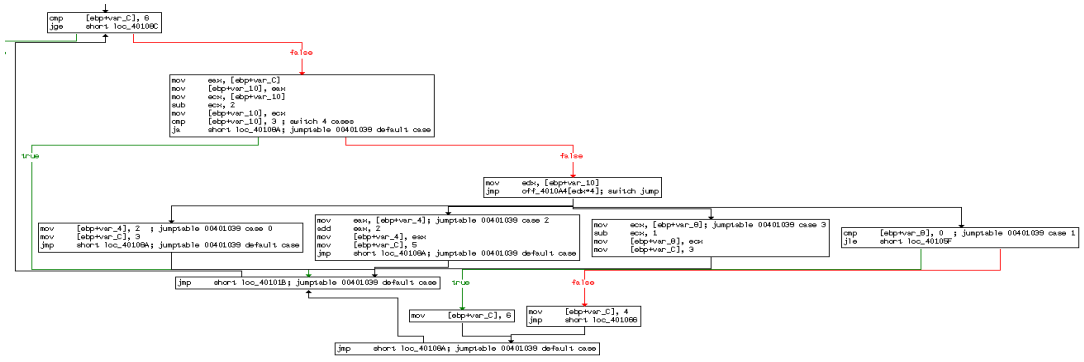

FIGURE 4. Graphe CFG du programme

n'applique pas toutes les optimisations) le code suivant :

```
...... !   push    eax
4011f1 !   mov     dword ptr [esp-0ch], 0ah
4011f9 !   mov     dword ptr [esp-8], 4
401201 !   dec     dword ptr [esp-0ch]
401205 !   add     dword ptr [esp-8], 2
40120a !   dec     dword ptr [esp-0ch]
40120e !   add     dword ptr [esp-8], 2
401213 !   dec     dword ptr [esp-0ch]
401217 !   add     dword ptr [esp-8], 2
40121c !   dec     dword ptr [esp-0ch]
401220 !   add     dword ptr [esp-8], 2
401225 !   dec     dword ptr [esp-0ch]
401229 !   add     dword ptr [esp-8], 2
40122e !   dec     dword ptr [esp-0ch]
401232 !   add     dword ptr [esp-8], 2
401237 !   dec     dword ptr [esp-0ch]
40123b !   add     dword ptr [esp-8], 2
401240 !   dec     dword ptr [esp-0ch]
401244 !   add     dword ptr [esp-8], 2
401249 !   dec     dword ptr [esp-0ch]
40124d !   add     dword ptr [esp-8], 2
401252 !   dec     dword ptr [esp-0ch]
401256 !   mov     eax, [esp-8]
40125a !   mov     [esp-18h], eax
40125e !   mov     dword ptr [esp-1ch], strz_y____d__402010
401266 !   mov     ebp, esp
401268 !   lea     eax, [esp-1ch]
40126c !   mov     esp, eax
40126e !   call    wrapper_crtdll.dll:printf_4012d8
401273 !   mov     esp, ebp
401275 !   mov     ebp, esp
401277 !   lea     eax, [esp+8]
40127b !   mov     esp, eax
40127d !   mov     esp, ebp
40127f !   xor     eax, eax
401281 !   pop     edx
401282 !   ret
```

Nous pouvons observer ici que la génération dynamique de code opérée par VxStripper nous permet naturellement de « déplier » le code aplati. L'application de transformations d'optimisation standard sur le module LLVM nous permet d'obtenir un programme débarrassé de cette obfuscation :

```
...... !   push    eax
4011f1 !   mov     dword ptr [esp-18h], 16h
4011f9 !   mov     dword ptr [esp-1ch], strz_y____d__402010
401201 !   mov     ebp, esp
401203 !   lea     eax, [esp-1ch]
401207 !   mov     esp, eax
401209 !   call    wrapper_crtdll.dll:printf_401268
40120e !   mov     esp, ebp
401210 !   mov     ebp, esp
```

```
401212 !   lea     eax, [esp+8]
401216 !   mov     esp, eax
401218 !   mov     esp, ebp
40121a !   xor     eax, eax
40121c !   pop     edx
40121d !   ret
```

Observons cependant que les transformations d'obfuscation étudiées jusqu'à présent sont assez rudimentaires. Nous avons l'intention de compléter cette évaluation en utilisant les outils de virtualisation de code disponibles sur le marché (et en complétant le jeu de passes d'obfuscation actuellement disponible dans notre obfuscateur).

À présent que le programme est désobfusqué, nous pouvons espérer que l'application d'un décompilateur va nous permettre de retrouver le code haut niveau de notre programme d'exemple (`printf("y=%d\n",22);`). Nous avons donc jugé utile de tester les *back-ends* C et C++ de LLVM.

3.7 Efficacité de la décompilation

Fonction	Scénario	Résultat
Décompilation	Après application des transformations de désobfuscation, l'objectif est ici de valider la qualité du code C ou C++ obtenu après décompilation.	Dans les deux cas, le code obtenu n'a pas la clarté escomptée.

Table 5. test des back-ends C et C++ de LLVM

Les *back-ends* C et C++ implémentés par LLVM ne produisent pas un code de qualité comparable à celle que l'on peut obtenir par utilisation d'autres outils de décompilation.

4 Conclusion

Même s'il reste encore du travail avant de pouvoir obtenir un logiciel supportant l'ensemble des outils de protection logicielle utilisables par les auteurs de *malware*, les premiers résultats obtenus nous encouragent

à poursuivre la voie d'étude de méthodes génériques d'*unpacking* et de désobfuscation, dans le but d'automatiser au maximum les tâches conduites par un analyste.

Cet outil constitue un logiciel d'analyse de *malware* autonome. Cependant, l'un des objectifs futurs de ce projet est de développer la capacité de l'outil à interagir avec d'autres outils d'analyse. De par sa conception, cet outil peut être en mesure de collaborer avec n'importe quel logiciel d'analyse fondé sur la chaîne de compilation LLVM. La chaîne de compilation LLVM, sa représentation intermédiaire bien conçue et les nombreux outils basés sur LLVM fournissent déjà une formidable bibliothèque d'analyses de programmes qui peuvent être utilisées pour défaire les mécanismes de protection des *malware*. Vellvm (*Verified LLVM*) pourrait en outre nous permettre d'extraire des implémentations vérifiées formellement des passes de désobfuscation implémentées dans notre outil.

Nous pouvons enfin envisager d'autres utilisations de cet outil, que celle de l'analyse de la menace : évaluation de la sécurité des solutions de protection logicielles, analyse de résistance des logiciels antivirus.

Plus de travail doit être fait pour rendre les CPU virtuels (et les autres composants d'un ordinateur virtuel) plus résistants face à la vaste gamme des techniques de détection d'émulation. L'émulation doit être aussi précise que possible, et une telle considération n'est pas actuellement une priorité pour la communauté des développeurs d'ordinateurs virtuels.

Actuellement, le seul système d'exploitation que nous avons instrumenté est le système d'exploitation Windows, en s'appuyant sur l'utilisation du serveur de symbole de Microsoft et du module de description de cet OS (figure 2). Même si le système d'exploitation Windows est toujours le plus ciblé par les programmes malveillants, il pourrait être utile d'étendre cet outil à d'autres systèmes d'exploitation, grâce à des modules spécialisés.

Certains mécanismes de protection reposent sur les interactions entre plusieurs threads ou processus d'une application, par le biais de gestionnaires de signaux et de mécanismes de synchronisation sophistiqués. Ces mécanismes de protection représentent un défi intéressant qui pourrait être adressé à l'avenir par notre outil.

Références

1. G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 : a platform for analyzing x86 executables. In *Compiler Construction*, pages 139–139. Springer, 2005.
2. U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic Analysis of Malicious Code. In *Journal in Computer Virology*, volume 2, issue 1, pages 67–77. Springer, 2006.
3. F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
4. BitBlaze : Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.
5. Jan Cappaert and Bart Preneel. A general model for hiding control flow. In *Proceedings of the tenth annual ACM workshop on Digital rights management*, DRM '10, pages 35–42, New York, NY, USA, 2010. ACM.
6. V. Chipounov, V. Kuznetsov, and G. Candea. S2E : A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News*, 39(1) :265–278, 2011.
7. T. Dullien and S. Porst. Reil : A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.
8. P. Ferrie. Attacks on Virtual Machine Emulators. In *Proceedings of the 2006 AVAR Conference, Auckland, New Zealand, December 3-5*, 2006.
9. Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Instruction Set Reference, 2012.
10. S. Josse. Secure and advanced unpacking using computer emulation. In *Proceedings of the AVAR 2006 Conference, Auckland, New Zealand, December 3-5*, pages 174–190, 2006. & In *Journal in Computer Virology*, volume 3, issue 3, pages 221-236. Springer, 2007.
11. S. Josse. Rootkit detection from outside the Matrix. In *Proceedings of the 16th EICAR Conference, Budapest, Hungary, May 5 - 8, 2007, & In Journal in Computer Virology*, volume 3, issue 2, pages 113–123. Springer, 2007.
12. S. Josse. *Dynamic analysis and detection of viral code in a cryptographic context*. PhD thesis, Ecole Polytechnique, 2009.
13. M.G. Kang, P. Poosankam, and H. Yin. Renovo : a hidden code extractor for packed executables. *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53, 2007.
14. C. Lattner and V. Adve. LLVM : a compilation framework for lifelong program analysis & transformation. *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.
15. Microsoft. Microsoft Portable Executable and Common Object File Format Specification, revision 8.0. http ://msdn.microsoft.com/, 2006.
16. N. Nethercote and J. Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6) :89–100, 2007.
17. M. Pietrek. An In-Depth Look into the Win32 Portable Executable File Format. *MSDN Magazine*, 17(2) :80–90, 2002.

18. G. Portokalidis, A. Slowinska, and H. Bos. Argos : an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of the 2006 EuroSys conference*, pages 15–27. ACM Press New York, NY, USA, 2006.
19. J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. http ://invisiblethings.org/papers/redpill.html, 2005.
20. T. Scheller. Llvm-qemu, backend for qemu using llvm, 2007. Google Summer of Code, `http://code.google.com/p/llvm-qemu/`.
21. Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze : A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.
22. Sourceforge. Yoda's Protector, 2012. `http://yodap.sourceforge.net/`.
23. Tiny code generator. `http://wiki.qemu.org/`.
24. T. Teitelbaum. Codesurfer. *ACM SIGSOFT Software Engineering Notes*, 25(1) :99, 2000.
25. Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *ACM SIGPLAN Notices*, volume 47, issue 1, pages 427–440. ACM, 2012.