Désobfuscation automatique de binaire - The Barbarian Sublimation

Yoann Guillot, Alexandre Gazet

Sogeti - ESEC

Résumé Ce papier présente l'état de notre recherche dans le domaine de l'automatisation du contournement de techniques de protection logicielle, concentrée ici plus particulièrement sur la problématique de désobfuscation.

Notre approche actuelle consiste à analyser la sémantique locale du code binaire afin de la reformuler de manière simplifiée en langage machine. Ceci permet notamment de s'affranchir de la recherche manuelle de « patterns » d'obfuscation, en mettant en place un mode de fonctionnement au final assez similaire à celui du module d'optimisation d'un compilateur.

Les résultats obtenus sur des exemples concrets sont très encourageants.

Dans la suite de nos travaux de l'an dernier[1], nous travaillons sur l'automatisation du contournement de techniques de protection logicielle, en nous concentrant plus particulièrement sur la problématique de l'obfuscation.

Ce travail est important, car beaucoup des logiciels malveillants (virus, chevaux de Troie, etc). utilisent ce type de protections afin de ralentir leur analyse. L'automatisation est essentielle pour faire face à la multiplication de ces codes d'année en année.

Notre précédent papier était principalement axé sur l'attaque et la suppression de mécanismes de protection logicielle au moyen du framework Metasm. Celui-ci fournit plusieurs primitives facilitant la manipulation de code protégé : modification du graphe de contrôle, recompilation, possibilité d'utiliser un pseudo-processeur filtrant... Il s'appuyait toutefois sur un fastidieux travail d'identification manuelle des schémas utilisés par la protection, préalable à la phase de suppression.

Nous allons présenter ici l'évolution de ces techniques, reposant maintenant sur une analyse sémantique du code binaire afin d'en extraire une représentation minimale. L'objectif n'est plus la recherche puis la suppression de schémas connus, mais une ré-écriture automatisée du code sous une forme optimisée[1].

Nous montrerons les résultats obtenus sur quelques exemples, et nous donnerons également un aperçu des pistes que nous creusons actuellement pour progresser davantage encore dans ce domaine.

[1] Optimisée dans le sens de la simplicité d'expression.

1 Metasm

Metasm[2][2] est un framework libre de manipulation de binaires, que nous avons utilisé pour résoudre deux challenges de reverse-engineering l'an dernier. Suite à ces travaux, nous y avons intégré plusieurs méthodes et techniques afin de faciliter un certain nombre de tâches récurrentes à l'avenir.

1.1 Désassemblage

Metasm permet de désassembler un fichier binaire à partir d'un ou plusieurs points d'entrée. Il va alors suivre le flot d'exécution, et utiliser son moteur d'émulation intégré pour résoudre les sauts indirects et les *cross-references* vers la mémoire (quelle instruction lit ou écrit à quelle adresse). Cette technique est appelée *backtracking* dans le framework, et est similaire à la notion de *slicing*[3][4][5] que l'on peut trouver dans la littérature.

Ce parcours du programme permet la construction de son graphe d'exécution. Les nœuds du graphe sont des blocs basiques d'instructions (séquences d'instructions exécutées séquentiellement et atomiquement lors de l'exécution du programme, en l'absence d'exceptions).

Ces nœuds sont organisés selon deux graphes entremêlés :
- le graphe des blocs d'une fonction,
- le graphe des fonctions et sous-fonctions.

Le graphe peut être visualisé au moyen d'une interface graphique interactive.

1.2 Mises à jour

Deux additions principales ont été faites au framework suite à nos travaux.

La première est une méthode permettant de modifier le graphe en en remplaçant une partie.

Cette fonction est paramétrée par trois éléments :

1. l'adresse de début du premier bloc à supprimer,
2. l'adresse de fin du dernier bloc,
3. et la liste des instructions qui serviront de remplacement (éventuellement vide).

Un nouveau bloc est alors construit à partir de la liste d'instructions, et est inséré dans le graphe en lieu et place des blocs sélectionnés.

[2] http ://metasm.cr0.org/

La seconde addition est une méthode permettant de récupérer la sémantique d'une section de code.

Celle-ci réutilise le moteur de backtracking qui est au cœur du désassembleur, en l'appelant à de multiples reprises afin de déterminer l'ensemble des effets dont la section de code est responsable :
- les modifications apportées aux différents registres du processeur,
- et la liste des écritures en mémoire effectuées.

Cette analyse est pour le moment restreinte à des séquences de code ayant une structure linéaire simple (sans boucle ou saut conditionnel). Comme nous le verrons dans la suite de ce papier, elle reste néanmoins au cœur de la plupart de nos applications. L'obtention de la sémantique du code permet par exemple de surmonter le niveau d'abstraction ajouté par l'implémentation d'une machine virtuelle en facilitant l'analyse des portions de codes responsables de l'exécution des instructions virtuelles (*handlers*).

Enfin, l'instrumentation du moteur de désassemblage a été facilitée par la mise en place de chausse-trappes (*callbacks*) permettant de prendre la main à différents moments et d'intercepter les données manipulées par le framework, par exemple :
- au désassemblage d'une nouvelle instruction,
- à la détection d'un saut (conditionnel ou non),
- à la détection de code automodifiant,
- ou encore à la fin du désassemblage.

2 Analyse d'une protection

Nous utilisons le terme *packer* pour désigner un logiciel de protection applicable à un programme binaire et/ou un code source afin d'obscurcir sa forme finale et d'en ralentir l'analyse. Les packers dits « classiques », du type `AsProtect`, `PeCompact`, etc. sont le plus souvent assez bien supportés par les outils de sécurité, par exemple les anti-virus ou outils de classification automatique. Différentes méthodes existent pour les contrer :
- Des unpackers statiques/dynamiques, qui nécessitent toutefois une analyse initiale poussée du fonctionnement de la protection.
- Des méthodes d'unpacking génériques (instrumentation de code[6], ou émulation comme Pandora's Bochs[7] ou Renovo[8].)

Le cœur de ces protections s'appuie principalement sur des notions de compression et de chiffrement, auxquelles viennent souvent s'adjoindre des fonctions d'anti-debug[9],

de gestion de licence, etc. La grande faiblesse de cette famille de protections est qu'à un moment ou un autre, le code doit être décompressé/déchiffré en mémoire (au moins partiellement) avant de pouvoir être exécuté par le processeur. Il est dès lors possible de le récupérer puis de l'analyser.

À côté de ces packers « classiques », nous trouvons une autre forme de protection, reposant non plus sur le couple chiffrement/compression mais sur la virtualisation du code. Cette catégorie n'est pas vulnérable aux attaques précédemment citées et il n'existe pas, à notre connaissance, de techniques génériques d'analyse y faisant face. Lors de nos travaux, nous avons rencontré plusieurs instances d'une même protection, et avons décidé d'en mener l'analyse.

En comparant très rapidement les différentes instances de cette protection, nous avons découvert que nous allions principalement manipuler deux grandes notions :

- **Le polymorphisme**. Cette notion est apparue au grand jour dans les années 1990, avec pour principal objectif de mettre en échec les mécanismes de détection par signatures utilisés par les antivirus. En changeant la forme du code, il est ainsi possible de contourner une signature utilisée par un produit antivirus. Pour ce faire, il est par exemple possible, d'exprimer la même sémantique, à l'aide d'une séquence d'instructions différentes. Par la suite, les différents éditeurs de logiciels anti-virus ont tenté de réagir pour contrecarrer cette technique et plus généralement l'ensemble des techniques d'obfuscation de code. Des travaux plus formels ont été publiés sur le sujet ; en 2003 notamment Frédéric Perriot a proposé une approche fondée sur l'utilisation d'optimisations afin d'aider à la détection des virus polymorphiques[10] ; mais aussi des travaux présentés en 2008 de Matt Webster et Grant Malcolm[11]. Dans le même esprit, nous pouvons citer le papier de Mihai Christodorescu[12]. Dans les deux cas, l'idée principale est d'automatiser le processus de désobfuscation afin d'améliorer la détection des codes viraux.
- **La virtualisation**. Pour rappel, dans le domaine de la protection logicielle, le terme *machine virtuelle* désigne une brique logicielle simulant le comportement d'un processeur, une sorte de processeur virtuel. Ce dernier est pourvu de son propre jeu d'instructions et exécute un programme spécialement écrit dans le code machine lui correspondant. Cela revient à ajouter un niveau d'abstraction entre le code machine tel qu'il est perçu lors de l'analyse (à l'aide d'un debugger ou d'un désassembleur) et sa sémantique. Pour plus de détails sur le fonctionnement interne d'une protection à base de machine virtuelle, le lecteur intéressé pourra se référer au papier proposé l'an dernier[1].

L'approche que nous allons présenter ici se veut didactique. À chaque étape de l'analyse, nous indiquerons quels sont les problèmes rencontrés et comment les résoudre.

2.1 Découverte de l'architecture de la protection

La présence d'une machine virtuelle apparaît rapidement. De grandes zones mémoires indéfinies sont utilisées. De plus certains appels de fonctions ont une forme très caractéristique (fig. 1) : le code original a été remplacé par un appel au stub d'initialisation de la machine virtuelle ; l'adresse poussée sur la pile correspond à l'adresse du bytecode implémentant l'algorithme protégé.

```
023E6C VM_CALL:
023E6C                 push    offset vm_call_1
023E71                 jmp     VM_INIT
023E76 ; ---------------------------------------
023E76                 push    offset vm_call_2
023E7B                 jmp     VM_INIT
023E80 ; ---------------------------------------
```

Fig. 1. Appels à la machine virtuelle

Autre construction caractéristique (fig. 2) : la présence du contexte de la machine virtuelle, ainsi qu'une table de handlers.

Jusque là, pas de grand défi, le code est relativement classique.

2.2 Optimiser pour mieux régner

Nous avons vu que la protection utilise une table de handlers, l'étape suivante est l'analyse de ces handlers. Cette analyse permet d'identifier le jeu d'instructions dont est pourvue la machine virtuelle (un de ces handlers est visible fig. 3).La première chose que nous remarquons est le découpage en de très nombreux blocs élémentaires reliés par des sauts inconditionnels. Ce type de code est parfois appelé « *code spaghetti* ». Cette méthode est peu efficace : lors de nos précédents travaux, nous avons déjà développé les méthodes nécessaires pour regrouper les blocs basiques et reconstruire le flot d'instructions.

Dans un second temps nous remarquons que la plupart des blocs ont pour caractéristique frappante le nombre d'opérations arithmétiques mises en jeu ainsi que l'utilisation « abusive » de mouvements ou opérations sur la pile. Ce comportement est clairement issu d'un processus d'obfuscation et nous allons devoir le corriger avant

```
0B800 VM_CONTEXT       dd 0
0B804                  dd 0
0B808                  dd 0
0B80C                  dd 0
0B810                  dd 0
0B814                  dd 0
0B818                  dd 0
0B81C                  dd 0
0B820                  dd 0
0B824                  dd 0
0B828                  dd 0
0B82C                  dd 0
0B830                  dd 0
0B834                  dd 0
0B838                  dd 0
0B83C                  dd 0
0B840                  dd 0
0B844 HANDLER_TABLE    dd offset loc_1000C1FA
0B844                  dd offset loc_1000F5E7
0B844                  dd offset loc_1000D2A1
0B844                  dd offset loc_1000F81E
0B844                  dd offset loc_1000DF2C
0B844                  dd offset loc_1000CF3C
0B844                  dd offset loc_1000BD9C
0B844                  dd offset loc_1000EFEA
0B844                  dd offset loc_1000F7D7
0B844                  dd offset loc_1000F58F
```

Fig. 2. Contexte de la machine virtuelle

d'être en mesure d'analyser simplement le handler.

Le problème se pose en ces termes : comment supprimer l'obfuscation en minimisant au possible l'analyse manuelle ? La réponse apportée repose sur l'utilisation de techniques d'optimisation, celles-là mêmes utilisées par les compilateurs. L'optimisation est une transformation du code, pour laquelle de nombreux objectifs contradictoires peuvent être choisis, comme la vitesse d'exécution ou la taille finale du code. Notre processus d'optimisation a pour objectif (*critère de performance*) de réduire du code obfusqué à une forme normalisée, d'expression simple. Contrairement à un compilateur standard, la performance du code (en termes de temps d'exécution) n'est pas notre objectif principal, mais elle sera néanmoins améliorée par effet de bord des optimisations réalisées.

Un des points les plus surprenants des techniques d'optimisation que nous avons utilisées est leur relative simplicité. Algorithmiquement très abordables, elles sont pourtant d'une efficacité redoutable. Pour cette phase, nous nous sommes inspirés des travaux proposés sur une cible équivalente[13]. En revanche, nous n'adoptons pas le même angle d'attaque, à savoir travailler sur une représentation textuelle du code à l'aide d'un parser/lexer. En effet, nous possédons déjà une représentation des instructions désassemblées sur lesquelles nous allons travailler directement. Les objets

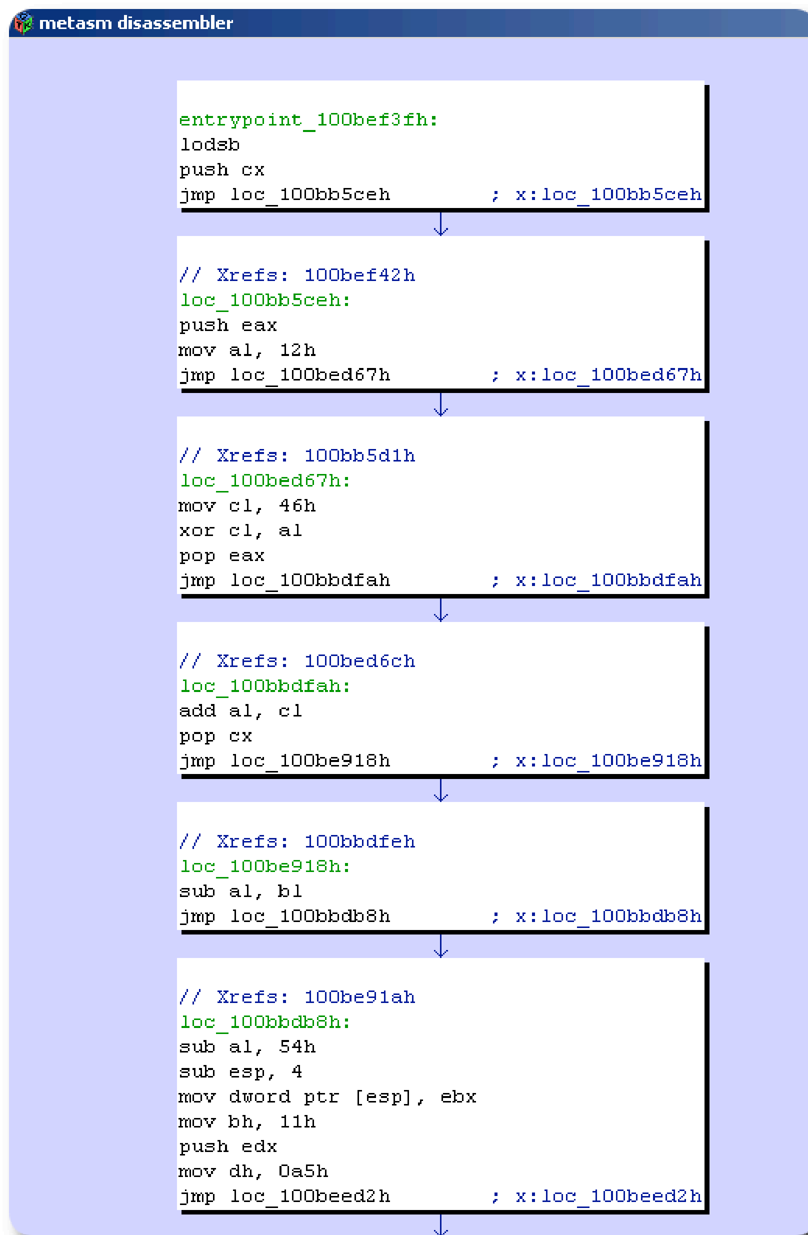

Fig. 3. Structure classique d'un handler.

`Metasm` manipulés seront principalement de type `Instruction` (on saluera au passage l'habile choix du nom des classes).

Nous avons mis en place les méthodes suivantes (les noms anglais ont été conservés, afin que le lecteur intéressé puisse facilement se référer à la littérature spécialisée sur le sujet) :

1. **Peephole optimisation.** Il s'agit de remplacer certains schémas connus par une forme simplifiée. Cette technique est, pour nous, la moins intéressante car c'est la plus proche de l'analyse manuelle des patterns. Elle reste toutefois nécessaire pour certains patterns particuliers, et permet également d'éviter, de temps en temps, l'utilisation de méthodes plus génériques mais beaucoup plus complexes (et de réduire ainsi le temps d'analyse).

2. **Constant propagation.** Cette technique consiste à propager la valeur d'une variable dans les expressions l'utilisant ensuite (fig. 4).

```
100bb5cfh mov al, 12h
100bed67h mov cl, 46h
100bed69h xor cl, al
```

```
100bb5cfh mov al, 12h
100bed67h mov cl, 46h
100bed69h xor cl, 12h
```

Fig. 4. Propagation de 12h à travers *al*.

La valeur propagée est `12h`, que nous trouvons ligne 1 assignée au registre *al*. Sur la ligne 3, *al* qui n'a pas été modifié depuis son initialisation est alors remplacé par sa valeur numérique.

3. **Constant folding.** La valeur initiale d'une variable est simplifiée en résolvant statiquement certaines opérations arithmétiques superflues (fig. 5).

   À la ligne 2, l'initialisation de *cl* à la valeur `46h` suivi à la ligne 3 d'un xor par une autre constante sans modification ni branchement intermédiaire est optimisée en calculant une fois pour toute le résultat de cette opération et en l'assignant directement. *cl* est alors initialisé avec le résultat de `46h xor 12h = 54h` ; la ligne 3, maintenant inutile, est supprimée.

```
100bb5cfh mov al, 12h
100bed67h mov cl, 46h
100bed69h xor cl, 12h
```

```
100bb5cfh mov al, 12h
100bed67h mov cl, 54h
```

Fig. 5. Réduction de *cl*.

4. **Operation folding.** Une nouvelle fois, un calcul est simplifié de manière statique, mais sans permettre ici de calculer une valeur finale à affecter à une variable.

```
100bb34fh add al, -7fh
100bb351h add al, bl
100bb353h add al, -70h
```

```
100bb34fh add al, 11h
100bb351h add al, bl
```

Fig. 6. Réduction de l'opération *add*.

Dans l'exemple présenté sur la figure 6, nous regroupons les opérations `add`[3] `al, -7fh` et `add al, -70h`. L'instruction résultante est `add al, (-7fh + -70h)`. De plus, notre moteur d'optimisation gère la commutativité des opérateurs, ce qui lui permet ici de réordonner librement une séquence d'*add* de la manière la plus utile aux opérations qui vont suivre.

5. **Stack optimisation** L'optimisation réalisée ici de la pile d'exécution est des plus basiques : nous supprimons simplement les couples *push*-*pop*[4] inutiles.

```
100bbdbah push ebx
100bb569h sub al, 56h
100bb56bh pop ebx
```

```
100bb569h sub al, 56h
```

Fig. 7. Nettoyage de la pile.

[3] *add* est une instruction qui modifie le registre passé en premier argument en lui ajoutant la valeur du second argument

[4] L'instruction *push* pousse une valeur au sommet de la pile d'exécution, et *pop* fait l'inverse

Sur la figure 7, il est possible de supprimer le couple *push ebx-pop ebx* car le registre référencé (`ebx`) n'est pas modifié (pas d'accès en écriture) entre les deux opérations considérées ; seul *al* l'est, ainsi que les *flags* du processeur.

Ces différentes méthodes sont intégrées dans un processus itératif : tant qu'une des méthodes arrive à optimiser une portion du code, le processus est appelé à nouveau sur le résultat. Les résultats ont été au-delà de nos espérances initiales.

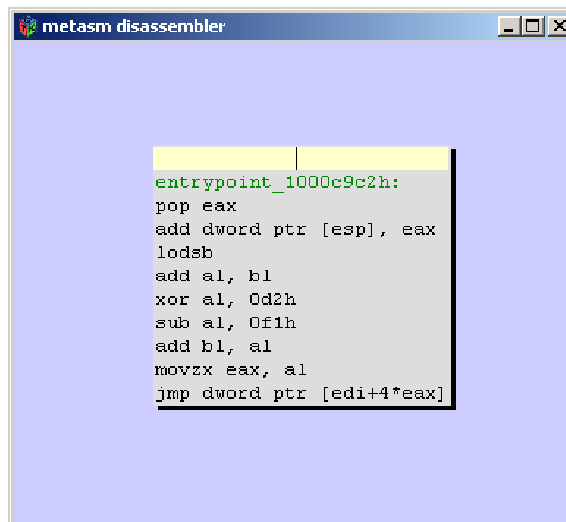

Fig. 8. Handler optimisé.

Le résultat est étonnant, car le code du handler a été réduit de manière drastique. La majorité des handlers sont initialement composés de 100 à 200 instructions réparties dans un grand nombre de blocs basiques. Le code optimisé ne comporte plus qu'une petite dizaine d'instructions pour un unique basique (en fait, une poignée de handlers de conception plus complexe déroge dérogent à cette règle). Une partie de ce code est commune à tous les handlers : il s'agit d'un stub (fig. 9), qui permet de calculer l'adresse du handler suivant. Ce choix découle du décodage du flot d'instructions virtuelles à l'aide d'une clé stockée dans le registre `ebx`. Le pointeur sur le bytecode est toujours situé dans `esi`.

Au final, nous pouvons conclure que la sémantique du handler initialement considéré n'est portée que par un nombre très réduit d'instructions (fig. 10).

```
1000bb4eh lodsb
1000eadfh add al, bl
1000d5f2h xor al, 0d2h
1000d4fch sub al, 0f1h
1000c222h add bl, al
1000f063h movzx eax, al
1000f066h jmp dword ptr [edi+4*eax]
```

Fig. 9. Déchiffrement de l'index du prochain handler.

```
1000c9c2h pop eax
1000e295h add dword ptr [esp], eax
```

Fig. 10. Code optimisé du handler.

Du point de vue de la « défense », l'apport de l'obfuscation, et donc du polymorphisme, est intéressant. Il peut s'envisager à deux niveaux :
– au niveau local : il augmente la complexité du code de chacun des handlers et donc la résistance à l'analyse.
– au niveau global : chaque instance générée de la machine virtuelle sera dotée de handlers différents. Il est donc nécessaire pour un éventuel attaquant ne possédant pas les outils adéquats de recommencer son analyse à zéro pour chaque instance rencontrée.

Du point de vue de l'« attaque », l'obfuscation utilisée ici est trop faible. Reconstruire le flot d'instruction au travers du code spaghetti ne nous pose aucun problème. Nous travaillons à un niveau d'abstraction très bas, à l'intérieur des blocs basiques. Pourtant, les résultats sont déjà probants. Le module d'optimisation ajouté produit code extrêmement simple. L'investigation a été réduite au strict minimum. Ce module réécrit progressivement le code. Chacune des méthodes d'optimisation est une règle de réécriture éventuellement associée à une condition. Chacune de ces transformations doit être correcte : le code transformé doit calculer la même fonction que le code original. Enfin, il faut s'assurer de la terminaison du système de réécriture.

2.3 Analyse des handlers

L'étape précédente nous a permis d'obtenir un code clair de chacun de nos handlers. C'est certes déjà positif mais nous sommes encore loin de notre objectif. Comme nous l'avons signalé plus tôt, une méthode a été ajoutée à Metasm. Elle permet

de récupérer la sémantique d'une section de code. Nous appelons donc la méthode `code_binding` en lui donnant l'adresse de la première instruction du handler comme point de départ, et l'adresse du début du stub de transition comme adresse de fin (tous les handlers aboutissent sur ce stub).

```
dword ptr [esp+4] := dword ptr [esp+4]+dword ptr [esp]
eax := dword ptr [esp]&0ffffffffh
esp := (esp+4)&0ffffffffh
```

Fig. 11. Sémantique du handler.

Nous obtenons ainsi immédiatement la sémantique du handler (fig. 11), que nous appelerons *binding* par la suite.

2.4 Exécution symbolique

Les étapes précédentes de l'analyse sont totalement automatiques. Lorsque l'outil rencontre un handler qu'il ne connaît pas, il est désassemblé, optimisé et son binding extrait. Cette étape est un peu coûteuse en temps, de l' ordre de quelques minutes. C'est pourquoi nous allons placer toutes ces informations dans un cache. Un fichier contenant la description (le code et la sémantique) de chaque handler et donc par extension du processeur virtuel est ainsi créé, et enrichi au fur et à mesure de la progression de l'analyse.

Ce que nous obtenons n'est ni plus ni moins que l'interpréteur du bytecode utilisé par la machine virtuelle.

Cette donnée est fondamentale. En effet, d'un point de vue théorique, pour deux langages L_a et L_b, nous savons qu'il est possible de trouver un compilateur de L_b vers L_a, si on connaît un interpréteur de L_b écrit en L_a. Ce résultat est connu sous le nom de *deuxième projection de Futamura*[14]. Nous verrons dans les prochaines étapes comment traduire de manière pratique ce résultat théorique.

Nous avons vu que chacun des handlers exécute le stub de déchiffrement de l'index du handler suivant, puis donne le contrôle à celui-ci après avoir récupéré son adresse dans la table des handlers. Cette information (le numéro du prochain handler) est

stockée chiffrée dans le bytecode du programme. Le chiffrement est basique (constitué de trois opérations), mais il fonctionne en déchiffrement est nécessaire pour calculer correctement le résultat suivant.

Pour être en mesure de suivre le flot d'exécution, il faut ainsi connaître la valeur de la clé et le pointeur sur le bytecode. Cette problématique n'est pas sans rappeler le challenge T2'07 que nous avons résolu l'an dernier. Nous allons cette fois attaquer le problème en utilisant une forme d'exécution symbolique.

```
vmctx = {
    :eax => :eax,
    :ecx => 1,
    :edx => :edx,
    :ebx => @key,
    :esp => :esp,
    :ebp => :ebp,
    :esi => @virt_addr,
}
```

Fig. 12. Déclaration d'un contexte symbolique

Nous déclarons un contexte symbolique (fig. 12) : il s'agit d'une simple vue partielle du processeur « hôte », à savoir une architecture `Ia32` classique. L'exécution symbolique n'est réalisée que sur les éléments nous intéressant ; les registres non significatifs ne sont pas pris en compte dans cette vue. Certains registres du contexte sont initialisés avec des valeurs numériques essentielles telles que *key* et *virt_addr*, qui sont liées au déchiffrement des opcodes du processeur virtuel (respectivement la valeur initiale de la clé de déchiffrement et la valeur initiale du pointeur sur le bytecode). Les autres registres se voient assigner une valeur symbolique, telle que *:eax*.

Nous avons déjà obtenu le binding de chacun de nos handlers. Le *binding* n'est que la représentation des valeurs de sortie en fonction des valeurs d'entrée d'une portion de code. Une étape importante est la contextualisation (ou spécialisation) du handler. En effet, un handler peut être vu comme un opcode, vide de tout opérande. Il est nécessaire de déchiffrer et de suivre le bytecode pour contextualiser le handler et ainsi obtenir la sémantique réelle du couple handler-instruction virtuelle. L'obtention de chacune des instructions, associées à un graphe de contrôle, aboutit au recouvrement

de l'algorithme implémenté.

Comment contextualiser un handler ? À l'aide du contexte symbolique du processeur hôte ! Pour résoudre ou réduire les différentes expressions qui composent le binding, il suffit d'injecter l'état du contexte avant exécution dans le binding calculé. Nous disposons de méthodes permettant de résoudre les indirections faisant référence à des données du programme en mémoire, ce qui permet de simplifier d'autant plus les expressions faisant intervenir des pointeurs connus. L'exécution symbolique d'un handler consiste simplement en l'application de son binding au contexte ; et l'exécution du programme à l'enchaînement de ces handlers.

Pour mieux visualiser les choses, regardons tout d'abord deux adorables papillons puis l'exemple suivant, dans lequel chacune des étapes est détaillée :

1. Récupération du code du handler (désassemblage).
2. Récupération du contexte courant (fig. 13).
3. Optimisation du code du handler (fig. 14).
4. Calcul du binding brut (fig. 15).
5. Simplification du binding par injection du contexte courant (fig. 16).
6. Mise à jour du contexte (fig. 17).

Les étapes 1 et 2 sont automatiques. Nous avons déjà vu comment elles étaient réalisées. Si la description du handler n'est pas présente dans le cache, elle est calculée.

Voici le contexte courant avant exécution de l'instruction virtuelle (fig. 13) :

```
eax := 2eh
ebp := ebp
ebx := 10016743h
ecx := 1
edx := edx
esi := 10016716h
esp := esp
```

Fig. 13. Contexte courant

Le code désassemblé puis optimisé du handler est visible (fig. 14). Le stub calculant l'adresse du prochain handler n'est pas affiché.

```
lodsb
sub al, bl
sub al, 63h
add bl, al
movzx eax, al
lea eax, dword ptr [edi+4*eax]
push eax
```

Fig. 14. Handler optimisé

La figure 15 présente le binding brut. Il est facile de constater que tous les éléments sémantiques sont effectivement présents : l'écriture au sommet de la pile issue de l'instruction `push`, l'incrément du registre `esi` lié à l'instruction `lodsb`, etc.

```
dword ptr [esp] := edi+4*((((byte ptr[esi]+-(ebx&0ffh))&0ffh)-63h)&0ffh)
eax := (edi+4*((((byte ptr [esi]+-(ebx&0ffh))&0ffh)-63h)&0ffh))
ebx := (ebx&0ffffff00h)|(((ebx&0ffh)+((((byte ptr[esi]+ -(ebx&0ffh))
&0ffh)-63h)&0ffh))&0ffh)
esi := (esi+1)&0ffffffffh
esp := (esp-4)&0ffffffffh
ip := 1000da0eh
```

Fig. 15. Binding brut

Le contexte est alors injecté dans le binding brut et les expressions sont résolues (fig. 16). Sur cet exemple, nous découvrons que le handler pousse la valeur `edi+1C` sur la pile.

Enfin, les écritures réalisées par le handler sont répercutées sur le contexte symbolique. La figure 17 montre la valeur de celui-ci après exécution du handler.

En pratique, les trois premières étapes sont immédiates, car ces informations sont mises en cache dans le fichier de description du processeur virtuel. Dès lors, nous sommes en mesure de :

- calculer le numéro du prochain handler ;
- l'adresse des opcodes virtuels suivants ;
- et donc de déchiffrer l'intégralité du bytecode en suivant le flot d'exécution de la machine virtuelle.

```
dword ptr [esp] := edi+1ch
eax := edi+1ch
ebx := 1001674ah
esi := 10016717h
esp := esp-4
ip := 1000da0eh
```

Fig. 16. Binding résolu

```
dword ptr [esp] := edi+1ch
eax := edi+1ch
ebp := ebp
ebx := 1001674ah
ecx := 1
edx := edx
esi := 10016717h
esp := esp-4
ip := 1000da0eh
```

Fig. 17. Contexte après exécution symbolique

2.5 D'un langage l'autre

En capitalisant les résultats de l'étape précédente, nous avons fait en sorte d'être en mesure de générer très facilement le code correspondant au handler contextualisé (fig. 18). Ces quelques lignes d'assembleur sont la représentation textuelle d'un opcode virtuel totalement déchiffré et contextualisé dans son flot d'instruction.

```
lea eax, dword ptr [edi+4]
push eax
```

Fig. 18. Assembleur généré pour le handler

Implicitement, nous avons réalisé une étape très importante lors de la génération de l'assembleur. À l'aide du binding contextualisé du handler, c'est-à-dire la sémantique de l'interpréteur du bytecode, nous avons traduit une instruction virtuelle en une

instruction exprimée en assembleur `Ia32` natif. Metasm disposant d'un compilateur, nous générerons alors automatiquement le code machine `x86` correspondant. Ce résultat est l'application directe de la deuxième projection de Futamura citée précédemment. Par analyse de la sémantique de l'interpréteur, nous avons réalisé un compilateur $L_{bytecode} \rightarrow L_{Ia32}$. Plus exactement, nous avons procédé à la spécialisation de l'interpréteur. Comme tout compilateur qui se respecte, nous allons maintenant optimiser le code produit.

2.6 Compilation : le tube de l'été

L'étape précédente nous a permis de convertir le bytecode de la machine virtuelle, en code machine `x86` natif. Cette utilisation des techniques de compilation est à mettre en perspective avec les travaux proposés par Rolf Rolles[15] visant à mettre en échec des machines virtuelles telles que `VMProtect` à l'aide de techniques de compilation. Dans son approche, le bytecode de la machine virtuelle étudiée était tout d'abord traduit dans une représentation intermédiaire destinée à être optimisée puis compilée. Nous avons fait le choix de passer directement du bytecode à un assembleur natif, en utilisant la sémantique et le code contextualisé du handler.

La compilation de l'assembleur obtenu se déroule sans encombre, en revanche, à la vue des très (trop) nombreuses occurrences des instruction `push` et `pop`, il apparaît assez clairement que le fonctionnement de la machine virtuelle est assez proche d'un automate à pile (fig. 19). Ce point est problématique, car générateur d'un *surplus* d'instructions et de mouvements de données compliquant l'analyse du code. D'une certaine manière, cet aspect du code est un résidu de la virtualisation.

Une nouvelle fois, nous allons utiliser nos méthodes d'optimisation. Elles vont ainsi nous permettre de totalement gommer cet aspect du code (fig. 20).

2.7 Everybody's gone surfin

Si toutes les étapes précédentes sont automatisées et globalement génériques, celle qui vient requiert une part d'analyse manuelle du code. Le code présenté sur les figures 19 et 20 sert d'introduction à une nouvelle étape.

En effet, ce code est présent dans tous les prologues de fonctions virtualisées. Tous les registres natifs du processeur hôte sont poussés sur la pile, puis *popés* dans des zones mémoires pointées par des indirections du type `dword ptr [edi+xx]`. Ces indirections pointent en fait vers le contexte de la machine virtuelle. Pour résumer, les registres natifs sont directement mappés dans les registres virtuels. Le processus inverse est réalisé dans le prologue de chacune de ces portions de code. Cette analyse,

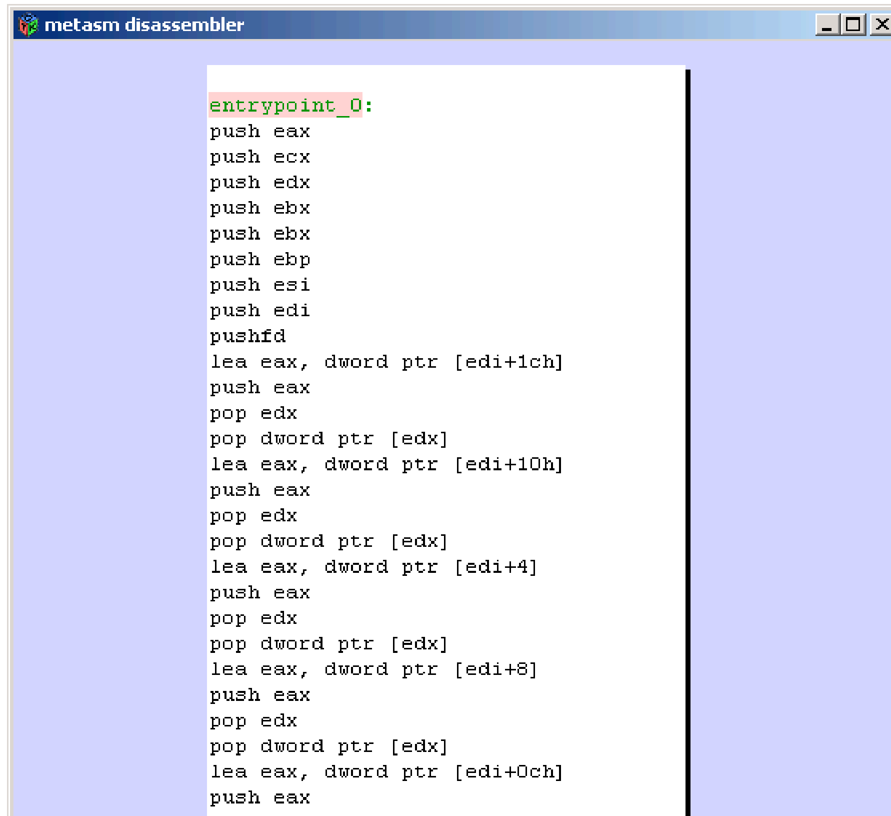

Fig. 19. Point d'entrée du code *dé-virtualisé*.

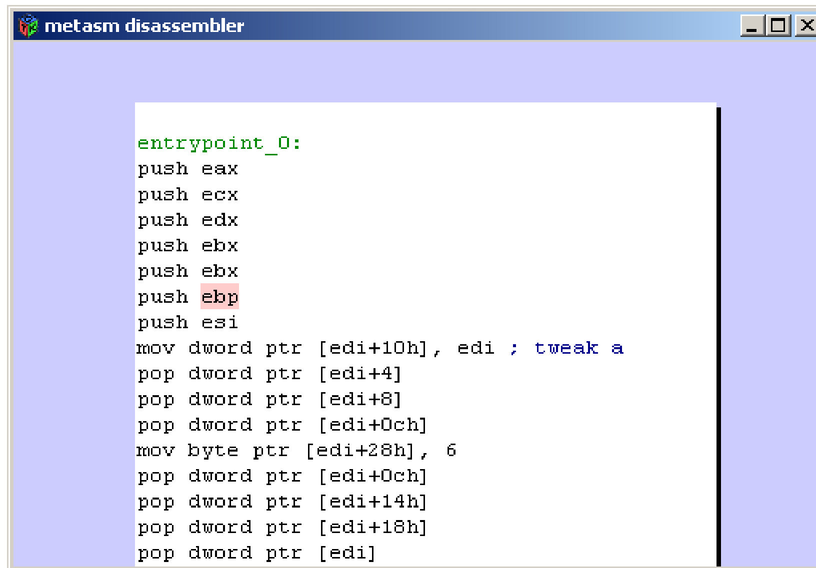

Fig. 20. Point d'entrée du *dé-virtualisé* légèrement optimisé.

bien que précieuse, voire essentielle, est spécifique à la cible d'où la perte de généricité. Dans le code final, ce code lié à la gestion de la machine virtuelle aura disparu.

Maintenant que nous sommes conscients du contexte virtuel, une question se pose : comment abstraire les registres de la machine virtuelle dans notre désassembleur `Ia32` ?

La réponse est d'une grande simplicité : il suffit d'étendre le processeur `Ia32` utilisé par Metasm pour lui adjoindre ces registres virtuels. Le code `Ruby` permettant de réaliser cette extension est présenté sur la figure 21 à titre indicatif. Nous retrouvons ici les qualités premières de Metasm : c'est un framework de manipulation de binaires, destiné à être aisément scriptable et adaptable à tous les besoins. Ici, un registre virtuel « étendu » sera vu et manipulé exactement comme un registre natif.

Cette extension réalisée, il suffit de parcourir nos objets instructions pour y *injecter* les registres virtuels à la volée. Ainsi, toutes nos méthodes d'optimisation s'appliqueront automatiquement et de manière transparente aux registres virtuels. Ceci est le coup de grâce porté au code virtualisé. Regardons en pratique sur un petit exemple le déroulement image par image du processus d'optimisation :

1. Code original, issu de la compilation du bytecode vers un assembleur `Ia32` natif :

```
def extend_ia32cpu

  Ia32::Reg.i_to_s[32].concat( %w[virt_eax virt_ecx ])
  Ia32::Reg.s_to_i.clear
  Ia32::Reg.i_to_s.each { |sz, hh|
     hh.each_with_index { |r, i|
        Ia32::Reg.s_to_i[r] = [i, sz]
     }
   }
   Ia32::Reg::Sym.replace Ia32::Reg.i_to_s[32].map { |s| s.to_sym }

end
```

Fig. 21. Extension du processeur `Ia32`

```
71h push 1000a2b4h
76h pop edx
77h mov eax, dword ptr [edi+2ch]
7ah add edx, dword ptr [edi+2ch]
7dh push dword ptr [edx]
7fh mov eax, dword ptr [esp]
82h pop ecx
83h xor dword ptr [esp], eax
86h pushfd
87h pop dword ptr [edi+1ch]
8ah lea eax, dword ptr [edi+18h]
8dh push eax
8eh pop edx
8fh pop dword ptr [edx]
91h lea eax, dword ptr [edi+18h]
94h push eax
95h lea eax, dword ptr [edi]
97h push eax
98h pop edx
99h pop edx
9ah push dword ptr [edx]
9ch push 1000a2d4h
0a1h pop edx
0a2h mov eax, dword ptr [edi+2ch]
0a5h add edx, dword ptr [edi+2ch]
0a8h pop dword ptr [edx]
```

2. Après la première passe : les registres virtuels ont été « injectés » (cf. `virt_ecx` ou `virt_eax`) dans le code manipulé. D'autres résidus de virtualisation, comme les calculs d'adresses relatives vers des adresses absolues, ont été supprimés (références à `dword ptr [edi+2ch]`). Le nombre d'opérations réalisées sur la pile est également réduit.

```
61h push dword ptr [1000a2d4h]
63h lea eax, virt_ecx
68h pop dword ptr [eax]
6ah lea eax, virt_ecx
6fh push dword ptr [eax]
7dh push dword ptr [1000a2b4h]
82h pop ecx
83h xor dword ptr [esp], ecx
8ah lea eax, virt_ecx
8fh pop dword ptr [eax]
91h lea eax, virt_ecx
94h push eax
95h lea eax, virt_eax
99h pop edx
9ah push dword ptr [edx]
0a8h pop dword ptr [1000a2d4h]
```

3. Le nombre d'opérations sur la pile (calculs ou simples mouvements de données) est de nouveau réduit.

```
61h push dword ptr [1000a2d4h]
68h pop virt_ecx
6fh push virt_ecx
82h mov ecx, dword ptr [1000a2b4h]
83h xor dword ptr [esp], ecx
8fh pop virt_ecx
91h lea eax, virt_ecx
99h mov edx, eax
9ah push dword ptr [edx]
0a8h pop dword ptr [1000a2d4h
```

4. Nous assistons progressivement à la reconstruction des mouvements de données initiaux. Pour l'instant, au moins deux lectures en mémoire apparaissent clairement (aux adresses 68h et 83h).

```
68h mov virt_ecx, dword ptr [1000a2d4h]
83h xor virt_ecx, dword ptr [1000a2b4h]
91h lea eax, virt_ecx
9ah push dword ptr [eax]
0a8h pop dword ptr [1000a2d4h]
```

5. La dernière indirection mettant en jeu l'instruction `lea` (*Load effective address*) est maintenant résolue. Nous avons alors un code très concis.

```
68h mov virt_ecx, dword ptr [1000a2d4h]
83h xor virt_ecx, dword ptr [1000a2b4h]
0a8h mov dword ptr [1000a2d4h], virt_ecx
```

Sur ce petit exemple, les 26 lignes du code original ont été réduites à seulement 3 lignes de code. Les mouvements ou opérations inutiles ont disparu, rendant le code très lisible. Cet exemple est tout à fait représentatif de l'efficacité globale du processus d'optimisation. De plus, nous retrouvons, dans le code que nous avons déprotégé, des constructions communes avec des parties de code qui n'avaient pas été virtualisées. Cette ultime étape apporte une validation supplémentaire à l'ensemble des transformations que nous avons appliquées sur le code.

2.8 Conclusion

Nous avons présenté ici une approche concrète utilisant le framework de manipulation de binaires Metasm. Les notions employées reposent sur des bases théoriques largement éprouvées : les concepts d'évaluation partielle et de spécialisation. Le lecteur intéressé par ces sujets pourra trouver une étude très intéressante de ces thèmes dans [16]. L'idée générale de la spécialisation est de supprimer tous les éléments interprétés. Lors de notre analyse, nous tirons parti de toutes les données statiques permettant de pré-calculer le résultat : précalcul des opérations arithmétiques ou des mouvements de données mis en jeu par le processus d'obfuscation, précalcul du résultat de l'application de l'interpréteur au bytecode de la machine virtuelle. Au final, une fois les règles de réécriture appliquées, nous possédons un programme spécialisé suivant nos propres critères de performance, qui sont prioritairement la concision du code et sa facilité d'analyse. De plus, cette spécialisation est quasiment optimale : nous avons totalement supprimé l'interpréteur et nous obtenons du code qui est du même ordre que le code original[17].

Au-delà de la simple démonstration technique, ce résultat est aussi un aperçu d'une des pistes vers laquelle devront tendre les moteurs d'analyse d'outils antiviraux, ou encore les outils de classification automatique. Comme nous l'avons rappelé précédemment, les technologies de types packers sont relativement bien gérées, mais la virtualisation reste un souci majeur. Un outil comme celui que nous avons développé pour l'occasion permet de recouvrer de manière automatique un code équivalent à celui avant virtualisation, et cela pour n'importe quelle instance de la protection. Il deviendrait ainsi relativement aisé de recouper les variantes d'une même souche virale par exemple. Le traitement à réaliser pour l'analyse est assez coûteux, c'est pourquoi ce type d'analyse pourrait être réservé à des serveurs dédiés disposant des ressources suffisantes, ou à des technologies de type *cloud computing*.

Les résultats que nous avons obtenus sont très positifs, en particulier sur le code obfusqué. Il convient toutefois de modérer la portée de l'outil que nous avons

développé. En effet, ne travaillant que localement à l'intérieur des blocs basiques, il est relativement aisé de le mettre en échec par l'ajout de boucles ou faux sauts conditionnels. Ceux-ci seront également faciles à circonvenir, mais nécessiteront une nouvelle analyse manuelle, et nous rentrerions ici dans le jeu du chat et de la souris.

Toutefois, nos méthodes d'optimisation travaillent à un très bas niveau d'abstraction : les instructions assembleur. Il pourrait être tentant de conserver nos méthodes d'optimisation tout en apportant une vision de plus haut niveau : gestion du graphe de contrôle, des boucles, des fonctions, etc. Il nous manque en fait une représentation intermédiaire de plus haut niveau du code assembleur. Dans la lignée de nos travaux de l'an dernier, l'axe choisi pour répondre à cette problématique est la génération de code C, autrement dit de la décompilation de code.

3 Décompilation

Lors des travaux présentés dans la partie précédente, une étape importante est la reconstitution de code assembleur natif à partir d'un *binding*, c'est à dire d'un ensemble d'affectations.

Or, cette phase est problématique, car le langage assembleur, en plus d'être évidemment lié à une architecture fixée, impose de fortes contraintes ; il est notamment impossible en `x86` d'avoir une instruction qui référence deux adresses mémoires. Il est donc nécessaire de connaître et de contourner ces limitations, par exemple en générant deux instructions pour traduire un effet donné.

Cela nécessite en outre de savoir précisément quelle instruction doit être utilisée pour additionner 2 registres, ou bien 1 registre et une constante numérique...

Utiliser du code C est par contre beaucoup plus facile. En effet, le binding est un simple ensemble d'affectation d'expressions, ce qui peut se traduire immédiatement en langage C. La traduction en C fait également disparaître toute la gestion des flags du processeur, ce qui allège d'autant plus les données à manipuler. Enfin, l'usage du C apporte également une gestion simplifiée du graphe de contrôle : il est en effet plus simple de parcourir un *if/else* que d'interpréter et de suivre dans un graphe un saut conditionnel assembleur.

Bien entendu, tout ceci ne sera possible que quand l'ensemble du code considéré sera correctement décompilé, ce qui peut être en soi un challenge difficile.

3.1 Mécanisme de la décompilation

Le module de décompilation est actuellement un des gros chantiers au sein de Metasm. Nous présentons ici son mode de fonctionnement actuel ; la version « finale »

est susceptible de présenter de nombreuses divergences par rapport à cette version.

À partir d'un point donné dans le graphe de désassemblage, le graphe est parcouru afin de déterminer les dépendances entre les blocs basiques en termes de lecture-écriture (*producteur/consommateur*). Chaque bloc est ainsi annoté avec la liste des registres/flags qui seront utilisés par un autre bloc, plus tard au cours de l'exécution, sans avoir été réécrit entre temps.

Le binding de chacun des blocs est ensuite calculé. Dans ce binding, les accès en lecture ou écriture à la pile sont remplacés par une variable symbolique représentant le pointeur de pile au point d'entrée, ce qui permet de faire ressortir l'usage de variables locales telles qu'utilisées traditionnellement par les compilateurs.

Le calcul du binding dépend de l'émulation correcte de l'intégralité des instructions du bloc. Si une des instructions n'est pas parfaitement émulée, elle est reportée telle quelle dans le langage C, au moyen d'une construction *assembler inline*. Ceci donnera du code probablement incorrect, et sera sans doute fatal pour les phases d'analyse automatique à suivre.

À partir du binding, la liste des dépendances d'un bloc est transformée en une séquence d'expressions C. Cette phase doit être menée avec précaution, car le binding représente un ensemble d'affectations simultanées, qui doivent être traduites en séquence ; il faut donc prendre particulièrement garde aux dépendances entre les valeurs. Il peut être nécessaire d'introduire des variables temporaires à ce stade.

L'instruction finale du bloc est ensuite inspectée afin de déterminer le mécanisme à utiliser pour traduire les transitions du flot d'exécution original : appel de sous-fonction, utilisation d'un *goto*...

Dans le cas d'un appel de sous-fonction, la décompilation est récursive, et l'on suppose donc que l'on connaît le prototype et l'ABI[5] de la fonction appelée. Les valeurs correspondant aux arguments sont alors mises en forme conformément au standard C. Les sauts conditionnels sont quant à eux traduits sous la forme d'un *if () goto*.

L'ensemble de ces expressions est stocké au sein d'une fonction C, qui contient la traduction de l'intégralité du code accessible depuis le point d'entrée choisi.

Cette limitation est nécessaire (le standard C n'autorise pas d'expressions au *scope* global), mais peut être trompeuse : une fonction C présuppose certaines propriétés (conservation du pointeur de pile, etc) qui ne sont pas forcément vérifiées par le code

[5] Application Binary Interface, c'est le « contrat » qui décrit notamment comment sont passés les arguments à la fonction

assembleur sous-jacent.

À partir de ce point, nous ne manipulerons plus du tout d'assembleur.

Il reste tout de même plusieurs tâches à accomplir avant de pouvoir conclure la décompilation ; notamment la reconstruction de structures de contrôle C, et la reconstruction des types des variables utilisées.

Voici un exemple de code manipulé à cette étape :

```
void sub_48h()
{
    register int eax, frameptr;
sub_48h:
    *(__int8*)(frameptr-12) = 0;
    *(__int32*)(frameptr-16) = 0x8051248;
    eax = 8;

loc_57h:
    eax = eax-1;
    if (eax == 0)
        goto loc_124h;
    *(__int8*)(frameptr-12) = *(__int8*)(*(__int32
        *)(frameptr-16));
    sub_244h(frameptr-16);
    goto loc_57h;

loc_124h:
}
```

Pour la suite, une première phase de nettoyage consiste à simplifier le graphe de contrôle, notamment en supprimant certains *goto*s inutiles (un goto pointant sur un autre goto par exemple). Ensuite, le code est parcouru afin de remplacer tous ces goto par les plus agréables *if* et *while*. Enfin, les labels inutilisés sont effacés.

En général à ce stade, sur du code effectivement compilé par un compilateur standard, on ne trouve plus de *goto*, et l'on distingue assez bien la structure du code.

Les types des variables sont alors inférés du code existant.

La principale source pour cela est le prototype des sous-fonctions appelées, qui permet de donner une indication sur le type de ses arguments. Les affectations sont également utilisées pour propager les types détectés.

Une fois cette passe faite, on possède une association liant un type supposé à l'offset d'une variable sur la pile. En cas de conflit, par exemple avec un membre d'une structure, on conservera la structure, et on modifiera le code afin de faire apparaître des *cast*s où cela est nécessaire.

Cette étape est la plus complexe de la démarche de décompilation, et peut se heurter à de nombreux problèmes. Le principal est l'aliasing : c'est lorsqu'un même emplacement mémoire est utilisé à différents moments pour stocker différentes variables. Ce problème est systématique pour les registres, qui se voient affecter de très nombreuses valeurs provenant de variables de types différents tour à tour au sein d'une même fonction. L'utilisation d'*union*s C est également problématique, car elle se manifeste de la même façon.

Ceci peut être résolu par une analyse de l'usage des variables, afin de délimiter des segments de code où une même variable prend des valeurs indépendantes (similaire à une analyse de *liveness*), qui pourrait se traduire par la création de variables C distinctes pour chaque domaine, ayant chacune leur propre type.

Le prototype actuel est très orienté *x86* mais donne des résultats encourageants ; il reste cependant un peu de travail avant de pouvoir l'utiliser comme nous le souhaiterions pour la désobfuscation de code.

3.2 Utilisation

Nous supposerons pour la suite que l'on dispose d'un décompilateur fonctionnel.

Cela nous permet d'utiliser un des nombreux outils existants pour effectuer un traitement optimisant du code (LLVM[6] par exemple). Le problème ici est que ces outils sont généralement orientés vers la génération de code optimisé pour la vitesse d'exécution, ce qui n'est pas forcément notre but ; certains algorithmes très efficaces produisent un code très difficile à lire ; quasiment du niveau d'une couche d'obfuscation.

Il est sans doute probable qu'une réutilisation des techniques que nous avons vues sur l'assembleur donnerait déjà de bons résultats, d'autant plus qu'ici le traitement pourrait être fait plus globalement, par exemple au niveau de la fonction tout entière ; on pourrait même envisager des optimisations inter-procédurales.

Si l'on parvient à intégrer le mécanisme de contextualisation discuté précédemment à la phase de décompilation, il semble possible de pouvoir décompiler directement le bytecode de certaines machines virtuelles, comme celle étudiée dans la dernière partie. Il est actuellement trop tôt pour se prononcer sur ce sujet, mais c'est une piste qui semble très prometteuse, et que nous ne manquerons pas d'investiguer.

[6] http ://llvm.org/

Étant donné la grande simplicité des handlers de cette machine virtuelle, il pourrait également être possible d'émuler directement certaines parties de ceux-ci une fois traduits en C, et de réaliser la phase de contextualisation à ce moment.

À ce stade, ces possibilités restent lointaines, mais ne semblent pas irréalisables.

4 Concolisme bucolique

En conclusion de ce papier, nous souhaitons présenter une autre voie d'extension de ces différents travaux. Nous ne nous concentrerons pas ici sur le résultat, mais sur la démarche de l'analyse et l'utilisation de Metasm. Le contexte est le suivant : nous étudions un binaire protégé, nous savons qu'il met en jeu plusieurs machines virtuelles, sans toutefois être en mesure d'obtenir ou de simuler l'intégralité de leur initialisation. De quelles voies de recours disposons-nous pour tenter de mener à bien une analyse ?

L'approche que nous allons proposer ici est dite « concolique ». Cet adjectif, couramment utilisé dans le monde du testing, désigne une approche couplant une exécution réelle avec l'exécution symbolique d'un programme. C'est le fond de notre démarche : nous allons utiliser une partie d'analyse dynamique (à l'aide d'un debugger) et une d'analyse statique telle que celles dont nous avons largement discuté jusqu'à présent.

4.1 La dynamique des corps

Metasm dispose d'un wrapper (très basique dans la version actuelle) sur l'API de debug standard de Windows, que nous allons mettre à profit. Nous sommes intéressés par l'analyse de la machine virtuelle, donc nous allons tout naturellement mettre un point d'arrêt matériel à l'entrée de cette machine virtuelle, et procéder alors à une analyse statique.

Un squelette de code est présenté sur la figure 22. Certaines fonctions, comme `update_eip`, ne sont pas détaillées ici. Néanmoins, on pourra apprécier la simplicité de mise en œuvre de ce script.

L'élément fondamental se situe sur les lignes 26 et 27. La variable `ctx` n'est rien d'autre que le contexte du processus débuggé, c'est-à-dire l'interface permettant d'accéder à la valeur actuelle des registres du thread qui est débuggué. La variable `remote_mem` permet d'accéder de manière transparente à l'intégralité de la mémoire

```
def debugloop
   debugevent = debugevent_alloc
      while not @mem.empty?
         if WinAPI.waitfordebugevent(debugevent, 500)
            debugloop_step(debugevent)
         else
            require 'starter'
      end
   end
end

def handler_newthread(pid, tid, info)
   super
   puts "Setting break on vm entry\n\n"
   set_hbp(@vm_entry, pid, tid)
   WinAPI::DBG_CONTINUE
end

def handler_exception(pid, tid, info)
   case info.code
      when Metasm::WinAPI::STATUS_SINGLE_STEP
         case get_context(pid, tid)[:eip]
         when @vm_entry
            puts "\n######   BREAK ON VM ENTRY   ######\n"

            ctx = get_context(pid, tid)
            remote_mem = OS.current.findprocess(pid).mem

            sa = Static_analyzer.new(remote_mem, ctx[:esp])
            sa.followHandlers

            update_eip()
         end
      end
   super
   WinAPI::DBG_CONTINUE
end
```

Fig. 22. Couplage debugger / analyse statique.

du processus. Ce sont ces deux éléments que nous allons injecter dans notre analyseur statique afin d'améliorer l'analyse.

4.2 Définition d'un processeur virtuel

Une rapide analyse dynamique de la machine virtuelle nous permet d'extraire les propriétés de différents éléments : différentes clés utilisées pour déchiffrer les opcodes, un jeu de flags, etc. En particulier, nous savons où ces éléments se situent en mémoire (nous avons leurs positions relatives par rapport au pointeur de pile `esp`). Nous décrivons ainsi le binding symbolique de notre machine virtuelle (fig. 23).

```
#------------------------   VM symbolic binding ------------------------
@symbolic_binding = {
  Indirection[Expression[:esp, :+, 0x10], 4, nil] =>  Expression[:key_a],
  Indirection[Expression[:esp, :+, 0x14], 4, nil] =>  Expression[:key_b],
  Indirection[Expression[:esp, :+, 0x18], 4, nil] =>  Expression[:key_c],

  Indirection[Expression[:esp, :+, 0x58], 4, nil] =>  Expression[:delta],
  Indirection[Expression[:esp, :+, 0x5c], 4, nil] =>  Expression[:delta_false],
  Indirection[Expression[:esp, :+, 0x60], 4, nil] =>  Expression[:delta_true],

  Indirection[Expression[:esp, :+, 0x134], 1, nil] =>  Expression[:flag8],
  Indirection[Expression[:esp, :+, 0x135], 1, nil] =>  Expression[:flag7],
  Indirection[Expression[:esp, :+, 0x136], 1, nil] =>  Expression[:flag6],
  Indirection[Expression[:esp, :+, 0x137], 1, nil] =>  Expression[:flag5],
  Indirection[Expression[:esp, :+, 0x138], 1, nil] =>  Expression[:flag4],
  Indirection[Expression[:esp, :+, 0x139], 1, nil] =>  Expression[:flag3],
  Indirection[Expression[:esp, :+, 0x13a], 1, nil] =>  Expression[:flag2],
  Indirection[Expression[:esp, :+, 0x13b], 1, nil] =>  Expression[:flag1],

  Indirection[Expression[:esp, :+, 0x13c], 4, nil] => Expression[:nrHandler],
}
#------------------------   VM symbolic binding ------------------------
```

Fig. 23. Définition du binding symbolique.

À titre d'illustration, la première ligne de ce hash signifie qu'en `dword ptr [esp+10h]` nous trouvons le symbole `key_a`. Nous définissons ainsi le mapping mémoire de chacun de nos éléments symboliques/virtuels.

4.3 Exécution symbolique simplifiée

Nous souhaitons maintenant procéder à l'exécution symbolique du bytecode sur le processeur virtuel. Pour rappel, nous avions fait l'hypothèse que nous n'étions pas en mesure d'analyser l'initialisation de la machine virtuelle. Comment initialiser le contexte du processeur virtuel ?

La réponse est des plus simple. Nous disposons déjà de tous les éléments nécessaires : le contexte et la mémoire du processus. Nous avons développé une petite méthode qui tient dans le main (fig. 24) :

```
def vm_ctx_init()
   vmctx = {}
   @symbolic_binding.each_value{ |key|
      vmctx[key.reduce_rec] = solve_ind_partial(
            @symbolic_binding.dup.invert[Expression[key.reduce_rec]],
            true
         )
   }
   vmctx
end
```

Fig. 24. Initialisation automatique du contexte du processeur virtuel.

Le résultat de l'appel à `vm_ctx_init` est le suivant (fig. 25) :

À chaque élément symbolique a été associée sa valeur numérique initiale. La méthode `solve_ind_partial` a déjà été évoquée précédemment dans le papier, elle permettait initialement de résoudre les indirections faisant référence à des données du programme en mémoire, c'est-à-dire présent dans sections de code ou données du programme. Nous avons étendu ses fonctionnalités afin qu'elle considère la mémoire effective du processus en cours d'exécution et non simplement sa valeur statique. Le contexte virtuel en main, nous allons pouvoir réaliser l'exécution symbolique du bytecode de la machine virtuelle.

4.4 Injection de symbolisme

Voici le code d'un handler de la machine virtuelle (fig 26)

```
delta := 250210h
delta_false := 0
delta_true := 25d568h
flag1 := 74h
flag2 := 35h
flag3 := 43h
flag4 := 7eh
flag5 := 60h
flag6 := 5fh
flag7 := 25h
flag8 := 0
key_a := 110h
key_b := 2
key_c := 2595a8h
nHandler := 0ce3h
```

Fig. 25. Contexte virtuel initialisé.

Nous calculons automatiquement son binding. Pour rappel, Metasm propose une méthode pour faire cela automatiquement. Dans le cadre de cette machine virtuelle, nous allons nous intéresser seulement aux accès en écriture en mémoire. Le résulat est présenté sur la figure 27.

Il y a du mieux, mais il est toujours difficile de comprendre ce qui se passe derrière le code. Des vies sont en jeu, nous ne pouvons plus reculer maintenant : l'avenir du monde libre dépend de nos actions. Si nous détaillons un peu la constitution des éléments du binding, nous y retrouvons des figures connues. Nous savons par exemple que le symbole `key_a` se cache derrière `dword ptr [esp+10h]`.

La solution ici est encore une fois très simple : il nous suffit d'injecter le binding symbolique dans le binding du handler. En `Ruby`, nous écrirons simplement :

```
expression.bind(@symbolic_binding)
```

Nous obtenons alors le résultat intermédiaire suivant (fig. 28) :

Le résultat est encourageant, il nous suffit alors de repasser les expressions obtenues à travers notre méthode de résolution des indirections. Voici tout d'abord la trace produite par la méthode (fig. 29) :

```
412eb3h mov esi, dword ptr [esp+14h]
412eb7h mov ecx, dword ptr [esp+18h]
412ebbh mov ebx, dword ptr [esp+10h]
412ebfh mov eax, dword ptr [436000h+4*esi]
412ec6h mov edi, dword ptr [436000h+4*ecx]
412ecdh mov edx, dword ptr [436000h+4*ebx]
412ed4h mov ebp, dword ptr [436004h+4*ecx]
412edbh xor eax, edi
412eddh mov edi, dword ptr [esp+10h]
412ee1h xor eax, edx
412ee3h mov ebx, dword ptr [esp+140h+4*eax]
412eeah mov eax, dword ptr [436004h+4*esi]
412ef1h mov edx, dword ptr [436004h+4*edi]
412ef8h mov esi, dword ptr [esp+14h]
412efch xor eax, ebp
412efeh xor eax, edx
412f00h mov ebp, dword ptr [esp+10h]
412f04h mov edx, dword ptr [esp+140h+4*eax]
412f0bh mov eax, dword ptr [436008h+4*esi]
412f12h mov esi, dword ptr [esp+18h]
412f16h mov edi, dword ptr [436008h+4*ebp]
412f1dh mov ecx, dword ptr [436008h+4*esi]
412f24h xor eax, ecx
412f26h xor eax, edi
412f28h mov ecx, dword ptr [esp+140h+4*eax]
412f2fh mov edi, dword ptr [esp+10h]
412f33h movzx eax, byte ptr [edx]
412f36h mov edx, dword ptr [43600ch+4*edi]
412f3dh test al, al
412f3fh mov byte ptr [ebx], al
412f41h mov eax, dword ptr [esp+14h]
412f45h setz byte ptr [ecx]
412f48h mov ecx, dword ptr [43600ch+4*esi]
412f4fh add edi, 4
412f52h mov dword ptr [esp+10h], edi
412f56h mov ebp, dword ptr [43600ch+4*eax]
412f5dh add esi, 4
412f60h mov dword ptr [esp+18h], esi
412f64h add eax, 4
412f67h mov dword ptr [esp+14h], eax
412f6bh xor ecx, ebp
412f6dh xor ecx, edx
412f6fh mov dword ptr [esp+13ch], ecx
412f76h jmp loc_401f20h
```

Fig. 26. Code d'un handler.

```
byte ptr [dword ptr [esp+4*(dword ptr [4*dword ptr [esp+14h]+436000h]^
(dword ptr [4*dword ptr [esp+18h]+436000h] ^dword ptr [4*dword
ptr [esp+10h]+436000h]))+140h]] := (byte ptr [dword ptr [esp+4*(dword ptr
[4*dword ptr [esp+14h]+436004h]^(dword ptr [4*dword ptr [esp+18h]+436004h]^
    dword
ptr [4*dword ptr [esp+10h]+436004h]))+140h]]&0ffh)

byte ptr [dword ptr [esp+4*(dword ptr [4*dword ptr [esp+14h]+436008h]^(dword
    ptr
[4*dword ptr [esp+18h]+436008h]^dword ptr [4*dword ptr
[esp+10h]+436008h]))+140h]] := ((byte ptr [dword ptr [esp+4*(dword ptr [4*dword
ptr [esp+14h]+436004h]^(dword ptr [4*dword ptr [esp+18h]+436004h]^dword ptr
[4*dword ptr [esp+10h]+436004h]))+140h]]&0ffh)==0)

dword ptr [esp+13ch] := (dword ptr [4*dword ptr [esp+18h]+43600ch]^(dword ptr
[4*dword ptr [esp+14h]+43600ch]^dword ptr [4*dword ptr [esp+10h]+43600ch]))

dword ptr [esp+10h] := dword ptr [esp+10h]+4
dword ptr [esp+14h] := dword ptr [esp+14h]+4
dword ptr [esp+18h] := dword ptr [esp+18h]+4
```

Fig. 27. Binding brut du handler.

```
byte ptr [dword ptr [esp+4*(dword ptr [4*key_b+436000h]^(dword
ptr [4*key_c+436000h]^dword ptr [4*key_a+436000h]))+140h]] := byte ptr [dword
ptr [esp+4*(dword ptr [4*key_b+436004h]^(dword ptr [4*key_c+436004h]^dword
ptr [4*key_a+436004h]))+140h]]&0ffh

byte ptr [dword ptr [esp+4*(dword ptr [4*key_b+436008h]^(dword
ptr [4*key_c+436008h]^dword ptr [4*key_a+436008h]))+140h]] := (byte ptr
[dword ptr [esp+4*(dword ptr [4*key_b+436004h]^(dword ptr
[4*key_c+436004h]^dword ptr [4*key_a+436004h]))+140h]]&0ffh)==0

key_a := key_a+4
key_b := key_b+4
key_c := key_c+4

nHandler := dword ptr [4*key_c+43600ch]^(dword ptr [4*key_b+43600ch]^dword
ptr [4*key_a+43600ch])
```

Fig. 28. Binding intermédiaire.

```
 - solve read access to arg: (dword ptr [4371ech]^(dword ptr [437888h]^dword
     ptr
[43b780h]))&0ffffffffh
 - solved key: 184dh

 - solve write access to arg: byte ptr [dword ptr [esp+4*(dword ptr
[4*key_b+436000h]^(dword ptr [4*key_c+436000h]^dword ptr
[4*key_a+436000h]))+140h]]
 - make stack variable <esp+136h> from stack address 23e9a6h
 - solved key: flag6

 - solve read access to arg: byte ptr [dword ptr [esp+4*(dword ptr
[437880h]^(dword ptr [4371e4h]^dword ptr [43b778h]))+140h]]&0ffh
 - solved key: flag5&0ffh

 - solve write access to arg: byte ptr [dword ptr [esp+4*(dword ptr
[4*key_b+436008h]^(dword ptr [4*key_c+436008h]^dword ptr
[4*key_a+436008h]))+140h]]
 - make stack variable <esp+135h> from stack address 23e9a5h
 - solved key: flag7

 - solve read access to arg: ((byte ptr [dword ptr [esp+4*(dword ptr
[437880h]^(dword ptr [4371e4h]^dword ptr [43b778h]))+140h]]&0ffh)==0)&0
    ffffffffh
 - solved key: ((flag5&0ffh)==0)&0ffffffffh
```

Fig. 29. Trace de la méthode de résolution des indirections.

Les éléments de symboliques sont progressivement injectés. Les indirections sont résolues, parfois jusqu'à obtenir des valeurs numériques. La gestion de l'*aliasing* (différents pointeurs sur une même zone mémoire) des zones de mémoire devient alors un jeu d'enfant. Les valeurs numériques sont transformées en leur équivalent symbolique lorsque cela est possible. Le résultat final du binding (fig. 30) devient alors tout à fait lisible et facilement interprétable.

```
flag6 := flag5&0ffh
flag7 := ((flag5&0ffh)==0)&0ffffffffh
key_a := 15e1h
key_b := 623h
key_c := 47ch
nHandler := 184dh
```

Fig. 30. Binding affiné final.

4.5 Pour la beauté du geste

Dans cette partie, nous avons souhaité illustrer une approche concolique d'une protection par machine virtuelle. Les résultats sont particulièrement encourageants. Nous disposons de tous les éléments nécessaires à l'analyse : le code et la mémoire. En nous affranchissant des limites et contraintes de l'analyse statique pure, nous avons pu simplifier grandement l'analyse des handlers de cette machine virtuelle. La principale difficulté étant souvent de savoir quand résoudre et quand rester en symbolique. Sur cet exemple, nous sommes en mesure d'intercepter les appels à la machine virtuelle, en réaliser l'exécution symbolique, mettre à jour le contexte et la mémoire, puis rendre la main au programme original. Une nouvelle fois nous nous rapprochons des concepts de compilations, ici nous réalisons la compilation à la volée du bytecode de la machine virtuelle vers notre interpréteur. Il serait tout à fait envisageable de compiler à la volée vers du code binaire `x86` natif.

5 Conclusion

Par rapport aux travaux présentés l'an dernier, nous avons franchi une nouvelle étape dans l'automatisation de l'analyse de code protégé par des mécanismes d'obfuscation ou de virtualisation. Les optimisations que nous avons utilisées dans la

première partie du papier sont, pour la plupart, tout à fait génériques, et d'une simplicité déconcertante. Notre outil reste très simple, nous avons néanmoins vu que nous manquions d'une représentation intermédiaire digne de ce nom et capable de supporter des optimisations de plus haut niveau ; étape sur laquelle nous travaillons au travers de la décompilation. Ceci permettra d'atteindre un degré de généricité encore supérieur.

Nous avons également franchi une nouvelle étape dans l'utilisation de la sémantique des instructions. L'utilisation du binding a révélé une fois de plus toute sa puissance. Nous avons été en mesure de mettre en échec une machine virtuelle complexe sans même analyser ses handlers. Mieux, l'extraction de la sémantique de l'interpréteur nous a permis de générer un compilateur bytecode virtuel vers code `x86` natif. Ce type d'approche, outre le cadre évident de l'analyse des protections logicielles, pourrait être une étape préparatoire à la recherche d'un motif de détection dans un code potentiellement malveillant[12].

En bémol, toutes ces approches supposent que nous soyons en mesure de désassembler la majorité du code étudié. Or, des techniques telles que l'aliasing de zones mémoire peuvent compliquer le recouvrement du graphe de contrôle du binaire, en exploitant les limitations du moteur de backtracking et d'émulation du framework.

Pour contrer cela, la troisième partie ouvre également des perspectives tout à fait intéressantes. L'utilisation d'une approche concolique permet d'avoir une vue et un contrôle total sur tous les éléments de l'analyse. Cette capacité, combinée la manipulation des éléments symboliques, permet de reprendre la main et mener des analyses toujours plus poussées sur le code.

Il reste de nombreuses questions en suspens :
- Python vaincra-t-il Ruby ?
- Jack Bauer sauvera-t-il le monde libre ?
- Est-ce que Péchabou obtiendra l'indépendance ?

Nul ne le sait, mais croisons les doigts !

Vous pouvez maintenant refermer vos actes et reprendre une activité normale. À tchao bon dimanche !

Références

1. Guillot, Y., Gazet, A. : Déprotection semi-automatique de binaire. In : 6ème Symposium sur la Sécurité des Technologies de l'Information et des Communicatins (SSTIC'08). (2008)

2. Guillot, Y. : Metasm. In : 5ème Symposium sur la Sécurité des Technologies de l'Information et des Communicatins (SSTIC'07). (2007)
3. Tip, F. : A survey of program slicing techniques. Journal of Programming Languages **3** (1995) 121–189
4. Wroblewski, G. : General method of program code. obfuscation (2002)
5. Beck, J., Eichmann, D. : Program and interface slicing for reverse engineering. In : In IEEE/ACM 15 th Conference on Software Engineering (ICSE'93), IEEE Computer Society Press (1993) 509–518
6. Quist, D., Valsmith : Covert debugging - circumventing software armoring techniques. (2007)
7. Böhne, L. : Pandora's bochs : Automated malware unpacking (2008)
8. Kang, M.G., Poosankam, P., Yin, H. : Renovo : A hidden code extractor for packed executables. (2007)
9. Ferrie, P. : Anti-unpacker tricks
10. Perriot, F. : Defeating polymorphism through code optimization (2003)
11. Webster, M., Malcolm, G. : Detection of metamorphic and virtualization-based malware using algebraic specification. In : EICAR. (2008)
12. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H. : Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA (nov 2005)
13. http://orange-bat.com
14. Futamura, Y. : Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls **2** (1971) 45–50
15. Rolles, R. : Optimizing and compiling (2008)
16. Marlet, R. : Vers une formalisation de l'évaluation partielle. PhD thesis, L'Université de Nice - Sophia Antipolis, École Doctorale - Sciences pour l'Ingénieur (1994)
17. Hartmann, L., Jones, N.D., Simonsen, J.G. : Interpretive overhead and optimal specialisation