Miasm : *Framework* de *reverse engineering*

Fabrice Desclaux

Commissariat à l'énergie atomique et aux énergies alternatives
Direction des applications militaires
fabrice.desclaux(@)cea.fr,
https://code.google.com/p/smiasm

Résumé *Miasm* est un *framework* de *reverse engineering* open-source. Son but est de permettre l'analyse, la modification et la regénération de programmes binaires. Il embarque plusieurs sous-composants, comme un manipulateur de (PE/ELF/CLASS) et un assembleur/désassembleur. *Miasm* définit également un langage intermédiaire sur lequel il s'appuie pour faciliter l'analyse de programmes.

1 Introduction

Miasm est un *framework* de *reverse engineering* écrit en python et open-source. Son but est de permettre l'analyse, la modification et la regénération de programmes binaires (PE/ELF/CLASS)

Pour cela, il embarque son propre assembleur/désassembleur ainsi qu'un langage intermédiaire.

Ce document décrira l'architecture de *Miasm*. Nous nous attarderons alors sur la définition de son langage intermédiaire. Ce dernier est la base pour les manipulations d'analyse de code, de désobscurcissement ou même de recherche de vulnérabilités. Il permet en effet de s'affranchir de la disparité des différents assembleurs natifs (x86, ARM, MIPS, ...) et du fait qu'ils sont peu adaptés à cet usage.

Enfin, des exemples illustreront l'utilisation de ce langage dans l'émulation de programmes, ainsi que dans l'analyse statique de binaires.

2 Architecture et principaux composants de Miasm

Cette partie décrit les divers modules de *Miasm*. Des exemples de manipulations du format PE ainsi que de l'assembleur sont donnés plus loin.

2.1 Architecture

Miasm est un *framework* de *reverse enginnering*. Il est composé de plusieurs modules :

- *Elfesteem* : un module de manipulation de binaire : ce module est utilisé pour disséquer des fichiers PE/ELF/CLASS, les modifier puis les regénérer. Ce module tente de s'occuper automatiquement de tout ce qui peut l'être lors des regénérations de programmes, comme les entrées des relocations, des imports ou des ressources d'un binaire.
- *Miasm ASM/DISASM* : un module pour assembler et désassembler supportant pour l'instant les architectures x86/arm/ppc/java.
- *Mi-asm IL* : un module de langage intermédiaire ainsi qu'un traducteur des langages assembleurs natifs vers ce langage. Le but est ici d'appliquer divers algorithmes sur un programme binaire. Travailler sur l'assembleur directement est assez fastidieux et non portable d'un assembleur à un autre : l'idée est de définir un langage intermédiaire dans lequel les assembleurs peuvent être traduits, et d'appliquer des algorithmes sur ce nouveau langage, plus simple et plus adapté.
- un module de simplification et d'exécution symbolique de ce langage intermédiaire. Il utilise le langage intermédiaire et permet de résoudre des problèmes simples posés par l'obscurcissement de programme en faisant par exemple de la propagation de constante.
- *Miasm JiT* : un module d'émulation, faisant de la traduction de code à la volée en utilisant la sémantique des instructions décrite dans le langage intermédiaire.
- *Grandalf* : un placeur de graphe, permettant d'afficher des graphes de flots d'exécution.

La figure 1 présente un schéma qui récapitule l'ensemble des modules de *Miasm*

2.2 Manipulation PE/ELF/Class

Elfesteem[1] permet de faire des manipulations sur les ELF, PE et CLASS à la manière de [4], c'est-à-dire à tenter de faire automatiquement tout ce qui peut l'être : les entrées des relocations, des imports ou des ressources d'un binaire.

Premièrement, le module *Elfesteem* permet la manipulation de PE/ELF/CLASS. Par exemple nous allons écrire un programme qui prend un fichier contenant un *shellcode* et crée un exécutable windows contenant une section (le *shellcode*) qui sera exécutable.

```
#! /usr/bin/env python
```

1. développé avec Philippe Biondi

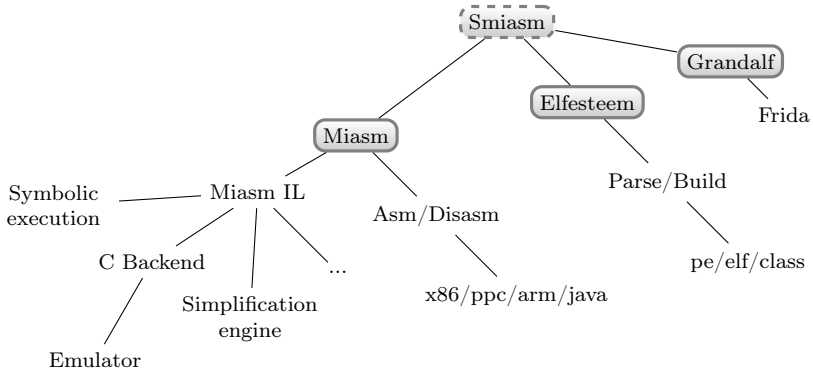


FIGURE 1. Composants de *Miasm*

```
import sys
from elfesteem.pe_init import PE

myshellcode =  open(sys.argv[1]).read()

e = PE()
s_text = e.SHList.add_section(name = "text", addr = 0x1000,
                              data = myshellcode)
e.Opthdr.AddressOfEntryPoint = s_text.addr

open('myshellcode.exe', 'wb').write(str(e))
```

Le code suivant illustre la modification d'entrées du répertoire des ressources dans un binaire Windows. Ces modifications sont prises en compte par *Miasm* lors de la génération du programme, qui calcule automatiquement leurs tailles et leurs positions finales.

Ici, nous allons changer le nom du menu *Fichier* de la calculatrice.

```
#! /usr/bin/env python

import sys
from elfesteem.pe_init import PE
import struct

e = PE(open(sys.argv[1], 'rb').read())
name = "\x00".join("cocolasticot") +"\x00"

for i in xrange(2):
    menu = e.DirRes.resdesc.resentries[1].subdir.resentries[i].
        subdir
    # read menu len & data
    off = 6 + struct.unpack('H', menu.resentries[0].data.s[4:6])[0]
    end = menu.resentries[0].data.s[off:]
    # modify
```

```
    menu.resentries[0].data.s = menu.resentries[0].data.s[:4]
    menu.resentries[0].data.s += struct.pack('H', len(name)) + name
        + end
    print repr(menu.resentries[0].data.s)

open('calc_menu_mod.exe', 'wb').write(str(e))
```

Si on exécute le binaire généré, on peut se rendre compte que le nom du menu est effectivement modifié.

2.3 Assembleur/Désassembleur

Miasm est également composé d'un assembleur/désassembleur. Ceux-ci sont écrits dans le but de pouvoir aussi bien travailler sur des instructions que sur un petit code source assembleur complet. Un mécanisme de multiplexeur permet également de désassembler une source pouvant être une chaîne de caractères, un binaire PE/ELF/CLASS ou même la machine d'émulation de *Miasm*.

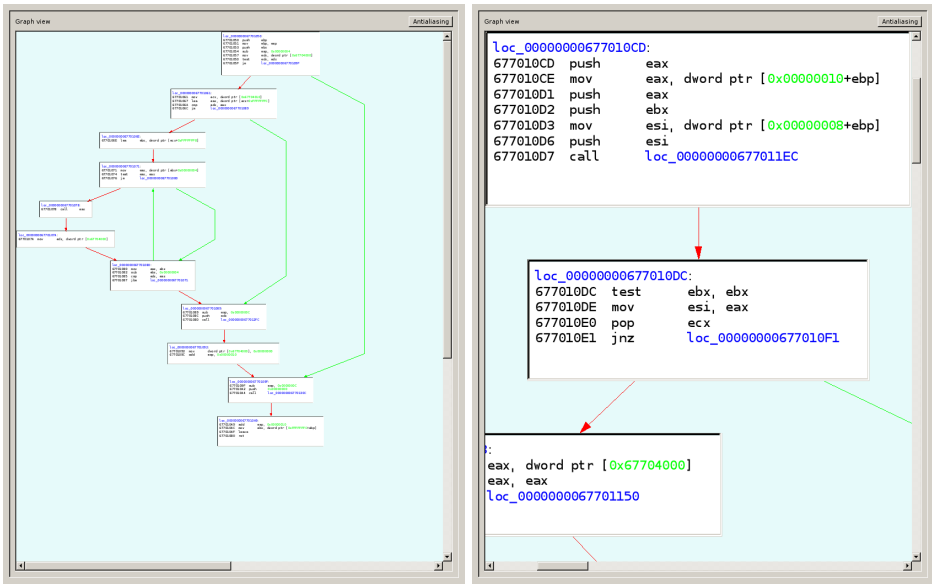
Voici quelques manipulations de bases de l'assembleur/désassembleur :

```
from miasm.arch.ia32_arch import x86_mn
>>> from miasm.arch.ia32_arch import x86_mn
>>> # disasm nop
... l = x86_mn.dis('\x90')
>>> print str(l)
nop
>>> # asm nop
... print x86_mn.asm('nop')
['\x90']
>>> # asm inc eax
... print x86_mn.asm('inc eax')
['@', '\xff\xc0']
>>> # disasm int eax (both forms)
... print str(x86_mn.dis('@'))
inc        eax
>>> print str(x86_mn.dis('\xff\xC0'))
inc        eax
```

Miasm inclut également une gestion des symboles, ce qui lui permet d'assembler des codes complets. Il est possible d'écrire directement des codes assembleurs qui seront par la suite injectés dans un binaire, servir de *shellcode*, ... Des exemples sont disponibles dans le fichier `example/asm_x86.py` dans le code de *Miasm*[2].

Pour terminer, *Miasm* embarque également un mini désassembleur *Frida*, offrant une vue graphique d'un listing assembleur.

2. http://code.google.com/p/miasm



3 Langage intermédiaire

Cette partie décrit le langage intermédiaire *Mi-asm* ainsi que ses API de manipulations.

3.1 Description

État de l'art *Miasm* utilise un langage intermédiaire pour représenter la sémantique des instructions d'un architecture donnée. Le but premier est de représenter les instructions dans une forme indépendante de l'architecture. Son faible nombre de mots évite les instructions redondantes et conserve une certaine simplicité dans les algorithmes qui l'utilisent.

L'écriture de *Miasm*[3] a été motivée par le constat de l'absence de langage simple pouvant représenter la sémantique des instructions assembleur. Aujourd'hui de nombreux autres langages permettant cette représentation. Par exemple :

– [6] qui est un assembleur *RISC* composé de 17 instructions, et dont le point fort est de permettre la traduction x86 vers REIL et inversement. Le point faible est peut être le nombre encore un peu important d'instructions, déplaçant une partie de la complexité dans les algorithmes de traitement (ce point peut se discuter).

3. aux alentour de 2007

- [7], dont le langage intermédiaire est assez proche de celui de *Miasm*. Il a été utilisé avec succès aussi bien dans l'analyse de code que la décompilation semi-automatique. Il n'a toutefois pas encore de *backend* permettant de faire de l'émulation à partir de ce langage.
- [2] est le langage intermédiaire de *BAP*. Ce langage est également proche de celui de *Miasm*. Une des différences se situe dans le fait que la représentation des instructions dans ce langage utilise des variables temporaires, alors que *Mi-asm* tente de les décrire par des opérations réalisées en parallèle (ceci sera décrit en détail par la suite).
- [3] utilisé principalement pour la décompilation. Il n'est pas *open-source*.
- [1] qui implémente une représentation simple des effets de bord d'une instruction servant surtout à l'émulation simple d'instructions.

Définition Le langage intermédiaire de *Miasm* tente de répondre à plusieurs problématiques : il a été conçu pour pouvoir être utilisé dans le désobscurcissement, l'analyse de code, la recherche de vulnérabilités dans des programmes binaires. C'est un langage d'expressions, défini comme suit :

- *ExprAff(dst, src)* c'est la seule expression qui modifie l'environnement d'exécution symbolique (ou d'émulation). Elle affecte l'expression *src* à l'expression *dst*.
- *ExprInt(valeur, taille)* cette expression représente un entier dont la taille en bits est passée en paramètre. Si la taille est omise, la variable est considérée sur 32 bits.
- *ExprId(nom, taille)* c'est une variable dont le nom et la taille en bits sont passés en paramètre. Si la taille est omise, la variable est considérée sur 32 bits.
- *ExprCond(condition, valeur_si_vrai, valeur_si_faux)* cette expression est équivalente à l'opérateur ternaire du langage C. Elle prend en paramètre 3 sous-expressions représentant respectivement la condition à tester, l'expression renvoyée si cette condition est vraie et l'expression renvoyée si cette condition est fausse.
- *ExprMem(adresse, taille)* cette expression représente un déréférencement mémoire, pointé par l'adresse donnée par une expression en paramètre, ainsi qu'une taille passée en paramètre. Si la taille est omise, la zone mémoire est considérée sur 32 bits.
- *ExprOp(nom_de_l_opérateur, opérande1, opérande2, ...)* cette expression représente l'application de l'opérateur dont le nom est passé

en paramètre sur les opérandes représentés par les expressions suivantes. Le nom de l'opérateur est de type : ['+', '-', '*', '/', '^', '|', '&', '<<', '>>', '<<<', 'a>>', 'parity', ...]. '<<<' représente une rotation à gauche, 'a>>' le décalage à droite arithmétique. Par exemple, *ExprOp('+', ExprInt(1), ExprInt(2))* est l'addition entre les entiers 1 et 2 (codés sur 32 bits).

- *ExprSlice(src, bit_start, bit_stop)* : extrait une tranche de *bits* de l'expression passée en paramètre. Cela peut par exemple extraire la portion *ah* du registre *eax* en x86, ou le *zero flag* du *eflag*.
- *ExprCompose(expr1, expr2, ...)* crée une expression composée de la concaténation des bits de sous-expressions. Ces sous-expressions sont des *ExprSliceTo*.
- *ExprSliceTo(expression, start, stop)* cette expression est utilisée uniquement dans le *ExprCompose*. Elle permet de donner l'information de placement de l'expression dans la concaténation d'expressions. Dans le futur, cette expression devrait être supprimée, car ces informations seront stockées dans le *ExprCompose*.

La simplicité du langage tente de représenter l'état d'esprit de l'auteur[4]. Les tailles des expressions sont incluses lors de leur définition : Cela offre la possibilité de faire des vérifications de compatibilité lors de la création des expressions.

Le choix a été fait de ne pas créer autant d'expressions opérateur que d'opérateurs. La nature de l'opérateur reste un paramètre de l'expression opérateur.

Cela a certains avantages : les algorithmes qui traitent le langage intermédiaire restent simples et ne manipulent que ces 9 entités. Même si on rajoute un opérateur qui n'est pas connu de l'algorithme, les traitements de flots de données restent valables.

Exemple de manipulation du langage intermédiaire :

```
>>> from miasm.expression.expression import *
>>>
>>> # define 2 ID
... a = ExprId('eax', 32)
>>> b = ExprId('ebx', 32)
>>> print a, b
eax ebx
>>> # add those ID
... c = ExprOp('+', a, b)
>>> print c
(eax + ebx)
>>> # + automaticaly generates ExprOp('+', a, b)
... c = a + b
```

4. Le langage a été défini avec la participation d'Axel "Capitaine Igloo" Tillequin.

```
>>> print c
(eax + ebx)
>>> # ax is a slice of eax
... ax = a[:16]
>>> print ax
eax[0:16]
>>> #memory deref
... d = ExprMem(c, 32)
>>> print d
@32[(eax + ebx)]
```

À partir de là, on peut définir les instructions de l'assembleur x86. Pour cela, on traduit une instruction par une liste d'expressions écrites dans le langage précédent. Chacune de ces expressions représente une entité (un registre, un *flag*, une case mémoire) modifiée par l'instruction. Par exemple, l'instruction *xchg* qui échange le contenu de ses deux arguments est représentée par :

```
def xchg(info, a, b):
    e = []
    e.append(ExprAff(a, b))
    e.append(ExprAff(b, a))
    return e
```

Cette instruction est décomposée en deux affectations :
- on place le contenu de la variable b dans la variable a
- puis on place le contenu de a dans la variable b

La première remarque pouvant être émise est que si l'on exécute ces affectations séquentiellement, le résultat est erroné. Il faudrait introduire une variable temporaire qui conserve la valeur de a avant la première affectation. Ici, une autre solution est adoptée : on considère que les expressions de cette liste sont exécutées simultanément.

Notez également que des informations sur le contexte de l'instruction sont passées en paramètre. Ceci est nécessaire car dans certains cas comme dans l'exécution 16 bits, les arguments n'apportent pas assez d'information, et on ne sait pas s'il faut soustraire au pointeur de pile 16 ou 32 bits.

Pour se représenter cette idée, on peut considérer une instruction comme une fonction de transfert entre un état d'entrée *in* et un état de sortie *out*. L'instruction *xchg* peut alors être écrit :

```
def xchg(info, a_in, b_in):
    e = []
    e.append(ExprAff(a_out, b_in))
    e.append(ExprAff(b_out, a_in))
    return e
```

Maintenant, si on effectue ces affectations séquentiellement on obtient :

```
a_out = b_in
b_out = a_in
```

On a bien les valeurs *a_out* et *b_out* qui valent respectivement *b_in* et *a_in*, et ainsi obtenir le comportement attendu. Pour finir, à la fin de l'évaluation d'une instruction, son état de sortie devient l'état d'entrée de l'instruction suivante.

```
a_in = a_out
b_in = b_out
```

On peut alors passer à l'évaluation de la prochaine instruction en utilisant le même principe.

Dans la représentation sémantique d'une instruction, chaque variable ne sera écrite qu'une fois (pour une instruction donnée). Pour que cette idée soit totalement applicable, quelques manipulations sont toutefois nécessaires, notamment pour l'affectation dans des *Slices*. Par exemple, l'affectation à *ax* est normalement représentée par :

```
eax[0:16] = ExprInt(uint16(42))
```

Lors de la création de cette expression, une transformation est appliquée et donnera :

```
eax = ExprCompose(ExprSliceTo(ExprInt(uint16(42)),
                              0, 16),
                  ExprSliceTo(eax[16:32],
                              16, 32))
```

Ici, on voit que les bits de 0 à 15 sont composés de l'entier 42, et les bits 16 à 31 composés des bits originaux de *eax*.

Voilà quelques exemples de représentation d'instructions du langage x86 :

```
def mov(info, a, b):
    return [ExprAff(a, b)]

def xchg(info, a, b):
    e = []
    e.append(ExprAff(a, b))
    e.append(ExprAff(b, a))
    return e

def push(info, a):
    e= []
    s = a.get_size()
```

```
    if not s in [16,32]:
        raise 'bad size stacker!'
    c = ExprOp('-', esp, ExprInt(uint32(s/8)))
    e.append(ExprAff(esp, c))
    e.append(ExprAff(ExprMem(c, s), a))
    return e
```

Le *mov* devient une simple affectation. Le *xchg* est représenté par deux affectations. Le *push* est représenté par le stockage en mémoire de l'argument, ainsi que la mise à jour du pointeur de pile.

Mi-asm permet également la rédaction de macro-instructions pour simplifier l'implémentation d'instructions comportant beaucoup d'effets de bord :

```
def add(info, a, b):
    e= []
    c = ExprOp('+', a, b)
    e+=update_flag_arith(c)
    e+=update_flag_af(c)
    e+=update_flag_add(a, b, c)
    e.append(ExprAff(a, c))
    return e
```

Ces macro-instructions sont implémentées avec les mêmes manipulateurs ; Par exemple, pour *update_flag_zf* :

```
def update_flag_zf(a):
    cast_int = tab_uintsize[a.get_size()]
    return [ExprAff(zf, ExprOp('==', a, ExprInt(cast_int(0))))]
```

Utilisation simple On peut alors très simplement retrouver pour une instruction donnée les registres lus/écrits :

```
from miasm.arch.ia32_sem import *

def get_rw(exprs):
    o_r = set()
    o_w = set()
    for e in exprs:
        o_r.update(e.get_r())
    for e in exprs:
        o_w.update(e.get_w())
    return o_r, o_w

a = ExprId('eax')
b = ExprMem(ExprId('ebx'), 32)

exprs = add(('u32', 'u32'), a, b)
o_r, o_w = get_rw(exprs)
# read ID
```

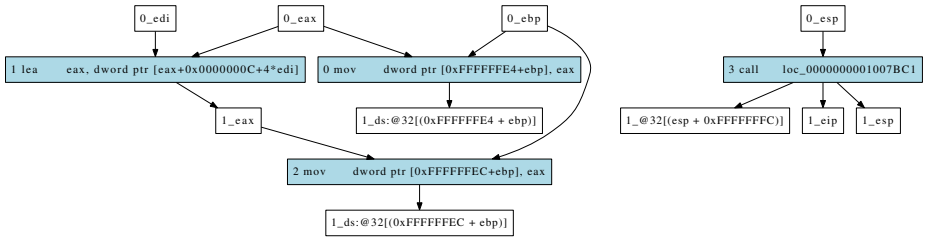
```
print [str(x) for x in o_r]
# ['eax', '@32[ebx]']

# written ID
print [str(x) for x in o_w]
# ['eax', 'pf', 'af', 'of', 'zf', 'cf', 'nf']
```

Un algorithme simple peut permettre alors de retrouver le flot de donnée d'un *basic bloc*[5]. Par exemple pour :

```
0 mov       dword ptr [0xFFFFFFE4+ebp], eax
1 lea       eax, dword ptr [eax+0x0000000C+4*edi]
2 mov       dword ptr [0xFFFFFFEC+ebp], eax
3 call      loc_0000000001007BC1
```

On obtient le graphe suivant :



D'autres algorithmes peuvent être utilisés ici, comme l'analyse de registres morts, la détection des variables locales, la dépendance de données inter-blocs, la détermination de la nature des arguments d'une fonction, ...

3.2 API de manipulations du langage intermédiaire

Évaluation symbolique L'exécution symbolique de *Miasm* peut servir à trouver la fonction de transfert d'un *basic bloc*. Cette fonction donne l'équation qui lie les registres et la mémoire après l'exécution d'un *basic bloc* à l'état des registres et de la mémoire en entrée de ce bloc.

Pour cela, *Miasm* part d'une machine dont l'état est représenté par un dictionnaire. Les clefs de ce dictionnaire peuvent être soit des identifiants (*ExprId*), soit des déréférencements mémoire (*ExprMem*). On associe à chaque clef l'expression représentant sa valeur.

Par exemple, dans le dictionnaire suivant :

5. Un *basic bloc* est une suite de lignes assembleurs dont aucun flot d'exécution n'arrive entre ces instructions, et dont le seul flot d'exécution sortant ne peut être que de sa dernière instruction

```
ExprId('eax'): ExprId('init\_eax')
ExprId('ebx'): ExprInt(4)
ExprMem('eax_init'+ExprInt(42)): ExprInt(1337)
```

Le registre *eax* vaut *eax_ init* ; le registre *ebx* vaut l'entier 4 et la case mémoire pointée par *eax_ init*+42 vaut l'entier 1337.

Pour émuler une instruction, *Miasm* applique les transformations décrites par la sémantique de cette instruction à l'état de la machine, et le met à jour. L'émulation de la prochaine instruction se fait de même en partant de ce nouvel état.

L'exécution symbolique d'un *basic bloc* complet est l'exécution symbolique répétée sur les instructions qui le composent. On obtient l'état de la machine en sortie de bloc, qui lie les registres et les cases mémoire à leurs valeurs en entrée de bloc.

Miasm implémente la sémantique de la plupart des instructions x86, ainsi qu'une petite partie de l'architecture ARM.

Voilà un code qui va désassembler un bloc d'un binaire de type PE, et l'exécuter symboliquement :

```
import sys
from miasm.arch.ia32_arch import *
from miasm.tools.emul_helper import *
from miasm.core.bin_stream import bin_stream

e = pe_init.PE(open(sys.argv[1]).read())
in_str = bin_stream(e.virt)

job_done = set()
symbol_pool = asmbloc.asm_symbol_pool()
l = asmbloc.asm_label('toto')
b = asmbloc.asm_bloc(l)

ad = 0x10120fa
asmbloc.dis_bloc(x86_mn, in_str, b, ad, job_done, symbol_pool)
print b
machine = x86_machine()
emul_bloc(machine, b)
print dump_reg(machine.pool)
print dump_mem(machine.pool)
```

et voilà le résultat :

```
# code assembleur
"""
xor         ecx, ecx
mov         ebx, 0x00001337
push        eax
add         eax, 0x00000005
pop         eax
"""
```

```
# Resultat
eax = init_eax
ebx = 00001337
ecx = 00000000
edx = init_edx
...
zf  = ((init_eax + 0x5) == 0x0)
```

L'exemple choisi met en œuvre plusieurs mécanismes :
- le registre *edx* n'est pas touché par ce *basic bloc*, il conserve donc sa valeur d'origine *init_ edx*
- le registre *ebx* est bien positionné à la constante 0x1337
- comme le registre ecx est XORé avec lui-même, son équation devrait être $ecx = ecx\_init \wedge ecx\_init$. Le moteur de simplification réduit cette expression à 0. La valeur finale de *ecx* est donc 0.
- pour finir, le registre eax est poussé sur la pile, puis on lui ajoute 5 ; Son équation est alors $eax = eax\_init + 5$. Mais plus loin, le pop *eax* repositionne sa valeur à sa valeur d'origine. Son équation finale est donc $eax = eax\_init$. On retrouve donc bien automatiquement le fait que le *push eax/pop eax* revient à ne rien faire, modulo les effets de bords générés par des instructions encadrées par les *push/pop*
- on note donc que le *zero flag* positionné par le *add* prend bien en compte la nullité de $eax\_init + 5$

Moteur de simplification *Miasm* inclut également un moteur de simplification d'expressions. Ce dernier est une suite de règles de réductions simples, qui seront appliquées si leurs conditions d'applications sont respectées. Par exemple :

```
# A + 0 => A
if op in ['+', '-', '|', "^", "<<", ">>"]:
    if isinstance(args[1], ExprInt) and args[1].arg == 0:
        return expr_simp(args[0])

# ((a >>> b) <<< b) => a
if op in ['<<<', '>>>'] and isinstance(args[0], ExprOp) and args[0].
    op in ['<<<', '>>>'] and args[1] == args[0].args[1]:
    if (op, args[0].op) in [('<<<', '>>>'), ('>>>', '<<<')]:
        e = expr_simp(args[0].args[0])
        return e
```

La description de ces simplifications se fait pour le moment en python, mais l'utilisation d'un module spécialisé tiers ou bien l'utilisation d'un langage permettant des desciptions simples est à l'étude.

Voilà une illustration d'utilisation [6] :

```
>>> from miasm.arch.ia32_sem import *
>>> from miasm.expression.expression_helper import *
>>> a = ExprId('eax')
>>> b = ExprId('ebx')
>>> c = a + b
>>> print c
(eax + ebx)
>>> d = c - a
>>> print d
((eax + ebx) - eax)
>>> print expr_simp(d)
ebx
>>> e = ExprInt(uint32(0x12)) + ExprInt(uint32(0x30)) - a
>>> print e
((0x12 + 0x30) - eax)
>>> print expr_simp(e)
(0x42 - eax)
```

À ce mécanisme de simplification, est ajouté (si on le désire) un module permettant d'utiliser certaines heuristiques pour simplifier le code résultant d'une émulation symbolique. Par exemple, en x86, tout objet situé au delà du pointeur de pile n'a normalement plus de raison d'être [7].

Pour détecter ces variables, le moteur recherche après chaque émulation symbolique d'une instruction les variables mémoires référencées par un pointeur de pile, et soustrait le pointeur de pile courant à cette adresse :

```
def del_above_stack(state, esp_val = None):
    if esp_val == None:
        esp_val = state[esp]
    kk = state.keys()
    for k in kk:
        if not isinstance(k, ExprMem):
            continue
        e_diff = expr_simp(ExprOp('-', expr_simp(k.arg), expr_simp(
            esp_val)))
        machine = eval_abs(state,
                           mem_read_wrap,
                           mem_write_wrap,
                           )
        ee = machine.eval_expr(e_diff, {})
        ee = expr_simp(ee)
        if not isinstance(ee, ExprInt):
            continue
        if int32(ee.arg) < 0:
            del(state[k])
```

6. Le lecteur d'un certain âge pourra faire le rapprochement avec les calculs symboliques des *TI-92*

7. sauf dans certains obscurcissements où leur utilisation déroute des analyseurs automatiques

Par exemple, si après l'exécution symbolique d'une instruction l'état de la machine est :

```
esp = ExprOp('+', ExprId(init_esp), ExprInt(0x4))
case_memoire = ExprMem(ExprOp('-', ExprId(init_esp), ExprInt(0x10)))
```

alors le code tentera d'évaluer l'expression représentée par la soustraction du pointeur mémoire et de *esp* :

```
ExprOp('-', ExprId(init_esp), ExprInt(0x10))  - ExprOp('+', (ExprId(
    init_esp), ExprInt(0x4)))
```

Ce qui donnera après simplification :

```
-0xC
```

Ceci permet d'ordonner deux expressions symboliques, et voir que la case mémoire est située au dessus du pointeur de pile et doit donc être supprimée.

Bonus : moteur de *Just-in-time compilation* Pour valider la correction de l'implémentation de la sémantique du x86, ainsi que sa couverture, l'idée d'implémenter un émulateur s'est imposée d'elle même. Pour cela, *Miasm* opère les opérations suivantes :

- il désassemble le x86.
- il le traduit en langage intermédiaire en utilisant la description de la sémantique de chaque instruction.
- le langage intermédiaire est alors passé à un *Backend* générant du C à la volée
- ce code C est ensuite compilé à l'aide de *TCC*.

le code C généré applique les effets de bord qu'aurait eu l'instruction sur les registres et la mémoire, émulant donc son comportement. Ce mécanisme est effectué sur des *basic blocs*.

Si le résultat donné par un programme exécuté sur un vrai CPU et le résultat d'un programme émulé par ce mécanisme sont identiques, on sait que l'émulation s'est bien passée, et donc que la sémantique décrite dans *Miasm* est valide.

Pour le x86, ceci a été testé sur des programmes joués, ainsi que sur de vrais virus. Certains de ces virus étaient également packés/obscurcis (upx/aspack/expression/...). Ces tests ont permis de vérifier une large palette d'instructions du x86.

La figure 2 est un graphique représentant cette mécanique interne.

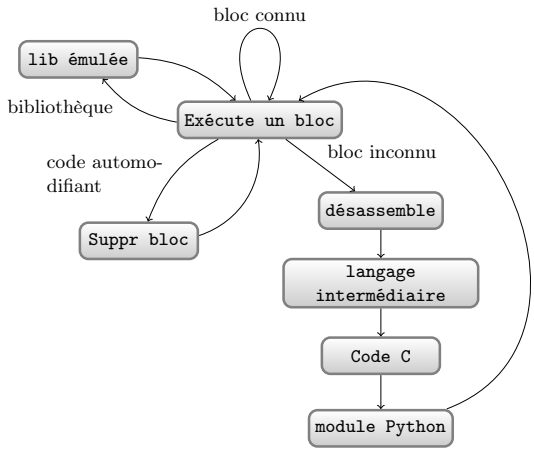


Figure 2. Mécanisme d'émulation concrète

Cet émulateur est également associé à *Elfesteem* qui permet de mapper un ELF/PE en mémoire. Les fonctions simples des bibliothèques utilisées par les programmes visés sont émulées en python. Ce mécanisme est détaillé plus tard et notamment son utilisation dans l'*unpacking*/analyse de *malware*.

L'émulateur est au final un émulateur de *CPU* scriptable en python. Néanmoins, certains mécanismes de l'*OS* peuvent et ont été implémentés, par exemple le mécanisme *SEH* de rattrapage d'erreurs sous Windows, un bout du mécanisme de chaînage des bibliothèques chargées en mémoire, ...

4 Exemples d'utilisation de *Miasm*

Cette partie décrit le mécanisme de traduction de code à la volée en utilisant le langage intermédiaire *Mi-asm*. Ceci permet par exemple l'exécution cloisonnée de code.

4.1 Exemple 1 : émulation d'un *malware* à partir du langage intermédiaire

On va utiliser ici le module d'émulation de *Miasm* dans le but de faire de l'étude de *malwares*. *Miasm* intègre un gestionnaire de mémoire virtuelle qui permet de manipuler les plages mémoires du programme visé. La mémoire du programme émulé est réservée sur le système hôte. Un mécanisme de traduction d'adresses est mis en place pour transformer

les adressages du programme émulé vers la mémoire de l'hôte. Les pages mémoire émulées ont exactement les mêmes propriétés que celles d'un système d'exploitation (*Read, Write, Execute*). Si le programme émulé sous *Miasm* écrit dans une page en lecture seule, *Miasm* déclenchera une faute.

Miasm remplit le même travail que le *loader* du système d'exploitation. Pour charger le binaire cible, il parse les sections du binaire (contenant le code, les données, ...) et va créer des pages mémoire sur le système hôte représentant ces sections.

Un binaire a en général besoin de bibliothèques pour assurer son bon déroulement. *Miasm* peut opérer de façon identique pour charger ces bibliothèques dans la mémoire émulée du programme cible.

L'émulation des bibliothèques pouvant être lourde (allant parfois jusqu'au *kernel*), on peut utiliser un autre mécanisme : un programme qui a besoin de bibliothèques embarque une liste des noms et des fonctions de ces bibliothèques (c'est cette liste qui est utilisée par le *loader* pour savoir quoi charger). Il décrit également où placer les adresses des fonctions nécessaires lorsqu'elles seront résolues par le *linker*. *Miasm* note toutes ces informations. L'astuce est de conserver la correspondance entre ces adresses et les noms des fonctions qu'elles représentent, ce qui servira plus tard lors de l'émulation : quand le pointeur d'instruction vaudra une de ces adresses, l'émulateur aura le choix entre émuler cette fonction ou appeler un *callback* d'une fonction python qui simulera le comportement de cette fonction. Elle doit prendre les arguments sur la pile, faire son traitement faire pointer l'adresse de la prochaine instruction à exécuter sur l'adresse de retour de la fonction.

Voilà un exemple de fonction simulée en python :

```
def kernel32_GetVersion():
    ret_ad = vm_pop_uint32_t()

    regs = vm_get_gpreg()
    regs['eip'] = ret_ad
    regs['eax'] = win_api.getversion
    vm_set_gpreg(regs)
```

L'exemple suivant est un *schellcode* Metasploit. On peut observer dans les logs les appels des API Windows de ce *shellcode* :

- le chargement de la bibliothèque *ws2_32.dll*
- la création d'une *socket* (appel à ws2_32_WSASocketA)
- la connexion vers le site malveillant (appel à ws2_32_connect)

Dans le code d'exemple, on simule les fonctions de connexion et d'envoi et réception de données. On arrive ainsi à simuler le *shellcode* jusqu'au téléchargement et l'exécution du second *stage*.

```
start emulation
0x4010a2
kernel32_LoadLibraryA 0x4010a2 0x123ffec
'ws2_32'
ws2_32_WSAStartup 0x4010b2
ws2_32_WSASocketA 0x4010c1 0x2 0x1 0x0 0x0 0x0 0x0
ws2_32_connect 0x4010db 0x1337 0x123fe4c 0x10
'\x02\x00\x11\\\xc0\xa8\x01\x01\x05\x00\x00\x00\\\xfe#\x01...'
0x4010f8
ws2_32_recv 0x4010f8 0x1337 0x123fe4c 0x4 0x0
...
```

4.2 Exemple 2 : émulation d'un Bootloader

Le but est ici d'analyser le déroulement d'un *bootloader*, pour valider l'absence de *malwares* pouvant infecter le *MBR*. Nous allons utiliser utiliser le module d'émulation de *Miasm*, comme dans la partie précédente.

Quelques différences sont notables : il n'y a pas ici de fonctions de bibliothèques à simuler mais les services du BIOS de la machine ; L'émulation se déroulera en 16 bits et il faut également supporter la segmentation. Le *bootloader* n'a pas non plus de descripteurs de section comme dans un binaire PE/ELF. Pour savoir comment charger le *bootloader* en mémoire, on peut se référer au comportement d'un *PC*, à savoir charger le premier secteur du disque dur à l'adresse `0x7C00`.

Voilà la petite partie de code responsable de la simulation de l'interruption 0x13, et notamment le service 0x41, qui permet de demander au BIOS les informations de géométrie du disque dur de la machine.

Dans cette fonction, il faut entre autre récupérer les paramètres tel que le numéro du disque dur physique dans les registres du programme émulé. Il faut alors positionner ces mêmes registres comme le feraient les services du BIOS de façon à renvoyer une information sur un disque factice.

```
def deal_disk_call():
    sector_size = 512
    cyl_num = 1
    regs = vm_get_gpreg()
    segms = vm_get_segm()
    bx = (regs['ebx'] ) & 0xFFff
    func = (regs['eax'] >>  8) & 0xFF
    drv_index = (regs['edx'] ) & 0xFF
    print 'DISK ACCESS', 'func', hex(func), 'drv', drv_index, hex(
        func),
```

```
    print 'ds:si', hex(segms['ds']), hex(regs['esi']), 'es', hex(
        segms['es'])
    if func == 0x41:
        print "DISK GET EXT"
        if bx!= 0x55AA:
            raise ValueError('bad func format')
        regs['cf'] = 0
        regs['ebx'] = 0xaa55
        regs['ecx'] = 0x1
        regs['eax'] = 0x100
        regs['eip'] +=2
        vm_set_gpreg(regs)
    ...
```

Après avoir simulé la partie du BIOS qui charge le premier secteur du disque en mémoire et place le pointeur d'instruction en 0x7C00, l'émulation peut commencer, on a alors les logs :

```
DISK ACCESS func 0x42 drv 128 0x42 ds:si 0x0 0x7bde es 0x0
DISK READ size 0x10 rez 0x0 num_sect_to_read 0x1 ptr 0x7c00
    seek_sect 0x3f
read 0x200
0x7c00
'\xebR\x90NTFS    \x00\x02\x08\x00\x00\x00\x00\x00\x00\xf8...
...
DISK ACCESS func 0x42 drv 128 0x42 ds:si 0x0 0x7bc8 es 0xd20
DISK READ size 0x10 rez 0x0 num_sect_to_read 0x1 ptr 0xd200000
    seek_sect 0x40
read 0x200
0xd200
'\x05\x00N\x00T\x00L\x00D\x00R\x00\x04\x00$\x00I\x003...
DISK ACCESS func 0x42 drv 128 0x42 ds:si 0x0 0xcfa2 es 0xd00
DISK READ size 0x10 rez 0x0 num_sect_to_read 0x1 ptr 0xd003000
    seek_sect 0x60003f
read 0x200
0x10000
'FILE0\x00\x03\x00Z\x14\x95\x08\x00\x00\x00\x00\x01\x00\x01\x008
```

Bien évidement, l'émulation complète d'un système d'exploitation est difficilement envisageable : cela nécessiterait l'implémentation de tous les périphériques d'un PC. [8]

4.3 Exemple 3 : analyse statique utilisant le langage intermédiaire

Cette partie illustre l'utilisation du langage intermédiaire dans le désobscurcissement de programmes malveillants.

8. De plus des outils plus adaptés comme *Ramooflax* permettent le débuggage de ces systèmes à tous les niveaux

Pour cela, on va utiliser l'exécution symbolique pour extraire des propriétés sur le code obscurci : des informations comme les registres morts, la hauteur de pile, et l'utilisation de calculs menant toujours à des constantes permettront de simplifier et de comprendre le code original. L'exécution symbolique utilise le mécanisme décrit en 3.2.

Quelques heuristiques sont toutefois apportées à ce principe : la pile est naturellement représentée dans l'état mémoire par des cases mémoires (indexées par une expression basée sur *esp*), auxquelles sont associées les valeurs poussées. Cependant, après l'exécution de chaque instruction, une procédure s'assure qu'aucun objet mémoire adressé avec *esp* n'est au dessus de la valeur courante de *esp*, auquel cas il est détruit. Cette heuristique modélise le comportement d'une pile.

L'exemple suivant concerne l'étude d'une machine virtuelle. Des travaux ont déjà été faits dans ce domaine en utilisant un langage intermédiaire. Un très bon article est celui de [5]. L'exemple décrit suit les pistes de ce document en utilisant les capacités de *Miasm*.

L'utilisation du langage intermédiaire se fera à travers l'étude d'un *malware* protégé avec une couche de code virtualisé. On peut remarquer que le code est obscurci, ce qui rend son analyse manuelle fastidieuse :

```
...
0x951c62 lodsb      byte ptr ds:[esi]
0x951c63 xor        al, bl
0x951c65 push       dx
0x951c67 mov        word ptr [esp], cx
0x951c6b mov        ch, 0x000000A4
0x951c6d xor        al, ch
0x951c6f jmp        loc_0000000000956169
0x956169 mov        cx, word ptr [esp]
0x95616d push       esi
0x95616e push       0x00005589
0x956173 mov        dword ptr [esp], ebp
0x956176 mov        ebp, esp
0x956178 add        ebp, 0x00000004
0x95617e sub        ebp, 0x00000004
...
0x9652b0 xchg       ebp, dword ptr [esp]
0x9652b3 pop        esp
0x9652b4 movzx      eax, al
0x9652b7 jmp        dword ptr [4*eax+edi]
```

En combinant l'émulation symbolique et le moteur de simplification, on obtient la fonction de transfert du bloc. Voilà les valeurs des registres ainsi que les cases mémoires touchées :

```
eax ((((@8[(@32[init_esp] - 0x9AAFFEC)] ^ @8[init_esp]) ^ 0xA4) + 0
    x15)_to[0:8], 0x0_to[8:32])
```

```
ebx (((@8[init_esp] - (((@8[(@32[init_esp] - 0x9AAFFEC)] ^ @8[
    init_esp]) ^ 0xA4) + 0x62)) + 0x4D)_to[0:8], @32[init_esp][8:32]
    _to[8:32])
ecx 00000001
edx init_edx
esi (@32[init_esp] - 0x9AAFFEB)
edi 00950E8C
esp (init_esp - 0x24)
ebp init_ebp
eip @32[(((((((@8[(@32[init_esp] - 0x9AAFFEC)] ^ @8[init_esp]) ^ 0xA4
    ) + 0x15)_to[0:8], 0x0_to[8:32]) * 0x4) + 0x950E8C)]
```

On voit sur la sortie un empilement de tous les registres x86 qui est caractéristique du passage du monde x86 au monde de la machine virtuelle. Ils seront modifiés plus tard par l'exécution successive des mnémoniques virtuelles :

```
@32[(init_esp - 0x8)]    init_eax
@32[(init_esp - 0x24)]    init_edi
@32[(init_esp - 0x1C)]    init_ebp
@32[(init_esp - 0x14)]    init_ebx
@32[(init_esp - 0x4)]    0x202
@32[(init_esp - 0xC)]    init_ecx
@32[(init_esp - 0x10)]    init_edx
@32[(init_esp - 0x20)]    init_esi
@32[(init_esp - 0x28)]    init_edx
@32[(init_esp - 0x30)]    (init_esp - 0x28)
@32[(init_esp - 0x18)]    (init_esp - 0x14)
@32[(init_esp - 0x2C)]    (init_esp - 0x24)
@32[(init_esp - 0x38)]    0x1
@32[(init_esp - 0x3C)]    0xE0408823
@32[(init_esp - 0x34)]    (init_esp - 0x30)
```

Ici la condition de terminaison de l'analyseur est conditionnée par la résolution du pointeur d'instruction. Si un obscurcissement introduit un faux saut conditionnel, l'analyseur [9] saura résoudre sa condition (par exemple par propagation de constante) et trouvera l'adresse du prochain pointeur d'instruction. Dans le cas contraire, il s'arrête et affiche son équation.

Ici, une rapide analyse du pointeur d'instruction montre qu'il est de la forme $eip = @32[base\_address + 4 * X]$. Il s'agit de la fonction qui *parse* les opcodes des mnémoniques de la machine virtuelle et qui va utiliser ceci comme une table de saut pour exécuter le code responsable de l'émulation de cette mnémonique : on pourra alors extraire ce tableau pour analyser chaque mnémonique individuellement.

La partie de code responsable du parsing des instructions a les effets de bords suivant :

9. s'il est assez performant

```
eax = ((((@8[init_esi] ^ init_ebx[0:8]) ^ 0xA4) + 0x15)_to[0:8], 0
    x0_to[8:32])
ebx = (((init_ebx[0:8] - (((@8[init_esi] ^ init_ebx[0:8]) ^ 0xA4) +
    0x62)) + 0x4D)_to[0:8], init_ebx[8:32]_to[8:32])
ecx = init_ecx
edx = init_edx
esi = (init_esi + 0x1)
edi = init_edi
esp = init_esp
ebp = init_ebp
zf = (init_esp == 0x0)
@32[(init_esp - 0xC)]    (init_esp - 0x4)
@32[(init_esp - 0x8)]    init_esp
@32[(init_esp - 0x14)]    init_ecx
@32[(init_esp - 0x4)]    init_edx
@32[(init_esp - 0x10)]    (init_esp - 0xC)
eip = @32[(((((((@8[init_esi] ^ init_ebx[0:8]) ^ 0xA4) + 0x15)_to
    [0:8], 0x0_to[8:32]) * 0x4) + init_edi)]
```

Par des raisons de clarté, on supprimera les registres qui ont une valeur en sortie de bloc égale à leur valeur en entrée de bloc, par exemple $eax = init\_eax$. De plus, on supprimera les cases mémoire obsolètes de la pile en utilisant l'heuristique décrite précédemment, ce qui donne ici :

```
eax = ((((@8[init_esi] ^ init_ebx[0:8]) ^ 0xA4) + 0x15)_to[0:8], 0
    x0_to[8:32])
ebx = (((init_ebx[0:8] - (((@8[init_esi] ^ init_ebx[0:8]) ^ 0xA4) +
    0x62)) + 0x4D)_to[0:8], init_ebx[8:32]_to[8:32])
esi = (init_esi + 0x1)
eip = @32[(((((((@8[init_esi] ^ init_ebx[0:8]) ^ 0xA4) + 0x15)_to
    [0:8], 0x0_to[8:32]) * 0x4) + init_edi)]
```

On peut noter plusieurs choses :

- *edi* n'est pas modifié
- *eax* est *mort*, c'est-à-dire qu'il est écrit sans être lu
- *esi* est incrémenté, et utilisé après déréférencement dans le calcul du prochain eip. Ceci pourrait être le *program counter* de la machine virtuelle
- *ebx* est une clef qui évolue, permettant de déchiffrer l'instruction courante.

Connaissant ces informations, on peut appliquer le même mécanisme pour retrouver la sémantique des mnémoniques de la machine virtuelle. Ici, la mnémonique :

```
0x95b2ce mov       ecx, dword ptr [esp]
0x95b2d1 push      0x00001CB5
0x95b2d6 mov       dword ptr [esp], ebp
0x95b2d9 push      0x00006EC5
0x95b2de mov       dword ptr [esp], esp
0x95b2e1 push      edx
```

```
 0x95b2e2 mov       edx, 0x00000004
 0x95f39c add       dword ptr [esp+0x00000004], edx
 0x961695 pop       edx
...
 0x957dd2 xchg      ebp, dword ptr [esp]
 0x957dd5 pop       esp
 0x957dd6 mov       dword ptr [esp], edi
 0x957dd9 push      dword ptr [esp+0x00000004]
 0x957ddd mov       edi, dword ptr [esp]
 0x95d44e add       esp, 0x00000004
 0x9615f5 pop       dword ptr [esp]
 0x9615f8 mov       esp, dword ptr [esp]
 0x9615fb mov       dword ptr [esp], edx
 0x953bd6 push      eax
 0x953bd7 pushfd
 0x953bd8 jmp       loc_0000000000951C62
```

se simplifie en :

```
ecx = @32[init_esp]
edx = (@32[(init_esp + 0x4)] umul32_hi @32[init_esp])
esp = (init_esp - 0x4)
@32[(init_esp - 0x4)] = (((0x1 == ((((init_esp - 0x4) ^ 0x4) ^
    init_esp) >>  ...
@32[(init_esp + 0x4)] = (@32[(init_esp + 0x4)] umul32_hi @32[
    init_esp])
@32[init_esp] = (@32[(init_esp + 0x4)] umul32_lo @32[init_esp])
```

En analysant les effets de bord de cette mnémonique, on peut voir :
- la pile a augmenté de 4 octets ;
- les deux avant-derniers éléments sur la pile correspondent aux parties haute et basse d'une multiplication 32 bits ;
- le dernier élément de la pile représente le *eflags*, c'est-à-dire les bits représentant les états du dernier calcul (négatif, nul, ...).

Les opérandes de la multiplication (*@32[init_ esp]* et *@32[(init_ esp + 0x4)]*) proviennent des deux éléments de tête de la pile en entrée de bloc. Nous pouvons en déduire que la machine virtuelle est une machine à pile. Tous les calculs ont le même schéma :
- empilement des opérandes par l'instruction précédente ;
- dépilement, calcul de l'opération, empilement des résultats, ainsi que du *eflag* résultant.

Pour cette mnémonique, on a donc le pseudo-code suivant :

```
pop a
pop b
c = a*b
push hipart(c)
push lowpart(c)
push eflags
```

Voilà une deuxième mnémonique :

```
esp = (init_esp - 0x2)
@32[(init_esp - 0x2)] = (0x2_to[0:2], (parity (@8[(init_esp + 0x2)]
    ^ @8[init...
@8[(init_esp + 0x2)] = (@8[(init_esp + 0x2)] ^ @8[init_esp])
```

Ici, il calcule un XOR sur deux éléments 16 bits. Voilà le pseudo-code :

```
pop16 a
pop16 b
push16 a^b
push16 eflags
```

Le dernier travail serait, comme souligné dans le document [5], de retraduire les mnémoniques simplifiées dans le langage x86 [10].

5 Conclusion

Après avoir succinctement décrit les divers modules de *Miasm*, nous nous sommes attardés sur la description de son langage intermédiaire ainsi que les divers composants permettant de l'exploiter. Celui-ci permet de représenter un code assembleur de façon plus simple et plus utilisable pour un analyseur de code.

Les exemples fournis ici illustrent le moteur d'émulation basé sur la traduction à la volée des instruction x86 en utilisant leurs sémantiques, ainsi que l'utilisation du langage intermédiaire et le désobscurcissement de code.

L'utilisation du langage intermédiaire promet de grandes possibilités pour automatiser l'analyse de code dans ces domaines, mais aussi dans la recherche de vulnérabilités.

Les futures directions de recherche s'orienteront surtout dans des algorithmes répondant à des questions simples, comme la découverte des variables locales d'une fonction et la distinction entre entier et pointeur. Ces algorithmes pourront s'appuyer sur les briques de base décrites dans ce document.

Références

1. . Radare. radare.org, 2007.
2. David brumley and Ivan Jager. BAP intermediate langage. bap.ece.cmu.edu/doc/bap.pdf, 2011.

10. Et permettant ainsi de retomber dans le langage maternel de l'analyste.

3. Ilfak Guilfanov. Hex Ray Microcode. hex-rays.com, 2009.
4. Philippe Biondi. scapy. *blabla*, 2012.
5. Rolf Rolles. Unpacking Virtualization Obfuscators. http://www.usenix.org/event/woot09/tech/full_papers/rolles.pdf, 2009.
6. Thomas Dullien, Sebastian Porst. REIL. www.zynamics.com/download/csw09.pdf, 2009.
7. Yoann Guillot. Metasm. metasm.cr0.org, 2007.