
# Miasm

## *Framework de reverse engineering*

Fabrice Desclaux

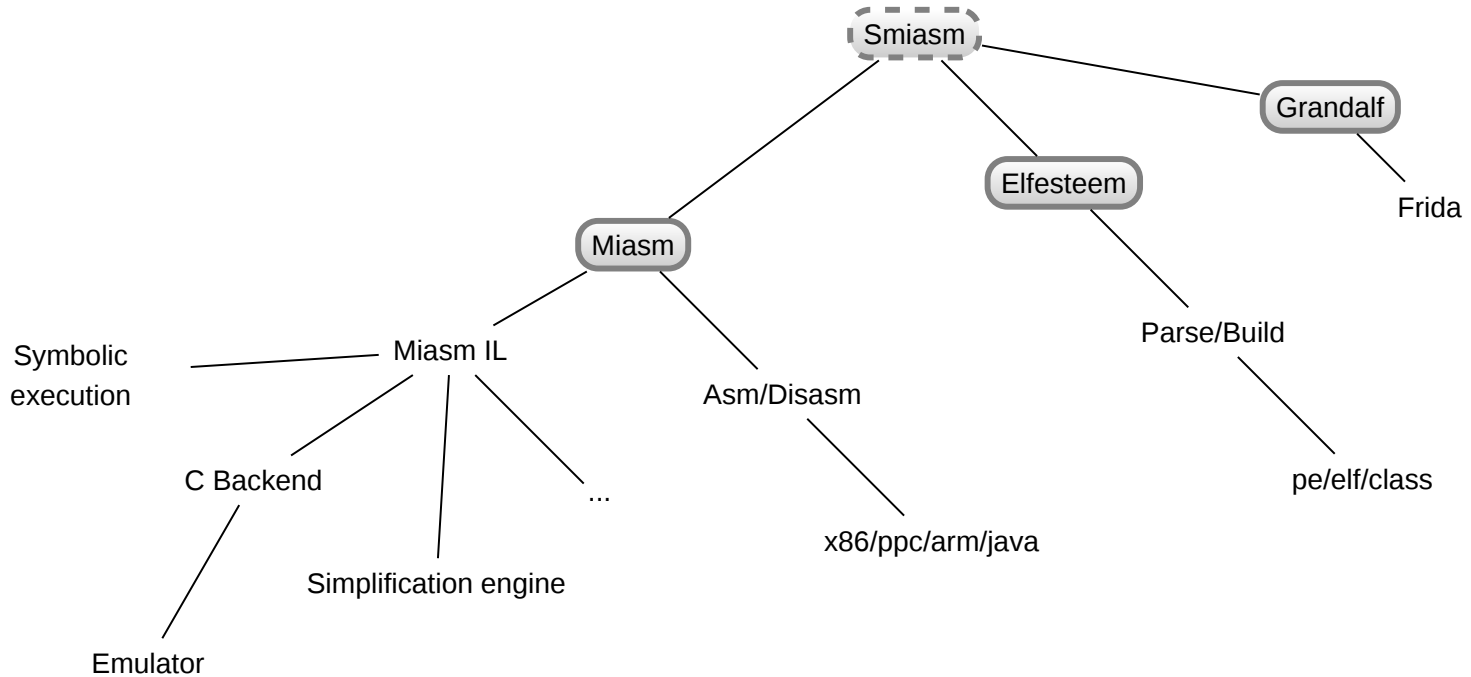
Commissariat à l'énergie atomique et aux énergies alternatives
Direction des applications militaires

SSTIC 2012
8 juin 2012

energie atomique • energies alternatives

## Qu'est ce que Miasm ?

Outil développé pour l'aide au *reverse engineering*, GPL, en Python.




## Permet les manipulations

- de sections
- de ressources/attributs/répertoires
- des imports/exports/
- par adressage virtuel/rva/offset
- parsing et *reconstruction* des binaires
- …

energie atomique • energies alternatives

## Modification des ressources de la calculatrice Windows

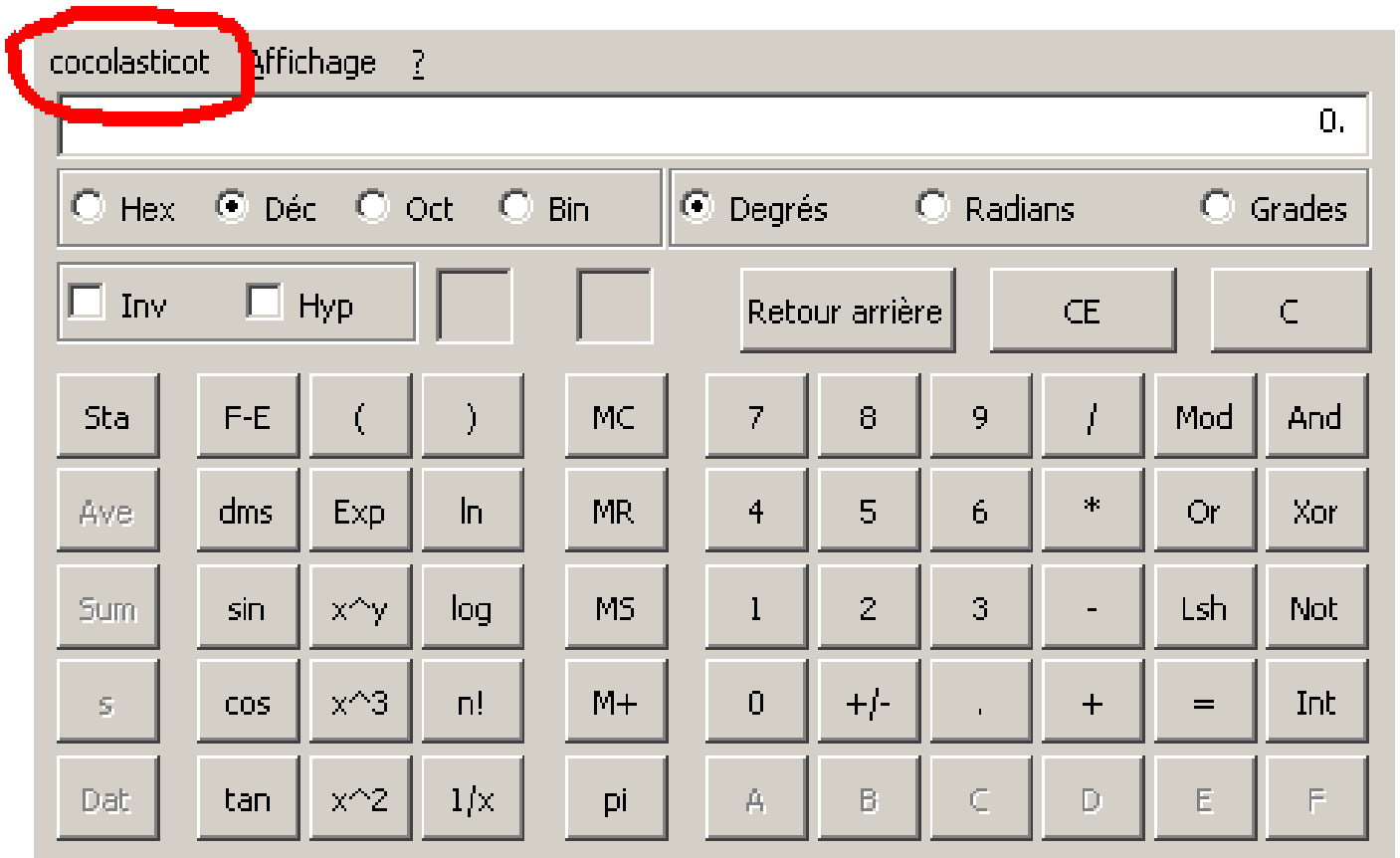
```
import sys
from elfesteem.pe_init import PE
import struct

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
    off = 6 + struct.unpack('H', menu.resentries[0].data.s[4:6])[0]
    end = menu.resentries[0].data.s[off:]
    menu.resentries[0].data.s = menu.resentries[0].data.s[:4]
    menu.resentries[0].data.s += struct.pack('H', len(name))
    menu.resentries[0].data.s += name + end

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

energie atomique • energies alternatives




## Modification de binaire

Binaire protégé avec un simple XOR

```
loc_4ce02e:                          ;xref j4ce036
  xor           byte ptr [ecx], 33h
  add           ecx, 1
  cmp           ecx, edx
  jc            loc_4ce02e
```

energie atomique • energies alternatives

## Déchiffrement de binaire

```
import sys
from elfesteem.pe_init import PE
from array import array

e = PE(open(sys.argv[1], 'rb').read())
start, stop = XXX, YYY

for i in xrange(start, stop):
    e.virt[i]=chr(ord(e.virt[i])^0x33)

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

energie atomique • energies alternatives

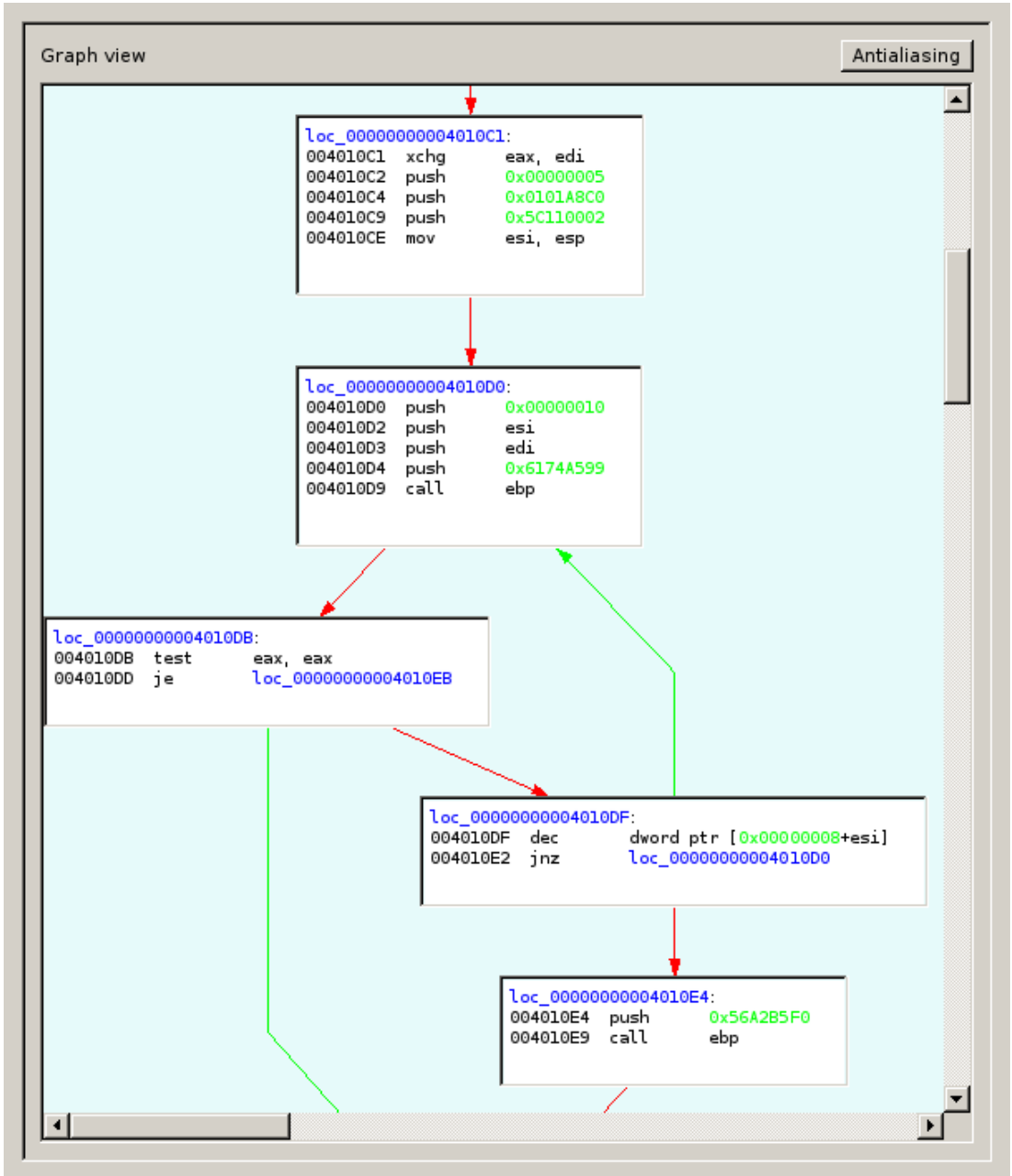
Ça y est, vous êtes des fakirs de la manipulation PE/ELF/CLASS !


Fakir niveau 1


```
all_bloc, symbol_pool = parse_asm.parse_txt(x86_mn, r'''
main:
    push 0
    push title
    push msg
    push 0
    call [ MessageBoxA ]
    ret
title:
.string "Hello!"
msg:
.string "World!"
''')
...
resolved_b, patches = asmbloc.asm_resolve_final(x86_mn,
                                    all_bloc[0], symbol_pool)
for p in patches:
    e.virt[p] = patches[p]
open('msg.exe', 'wb').write(str(e))
```


Hello!
World!
OK

energie atomique • energies alternatives

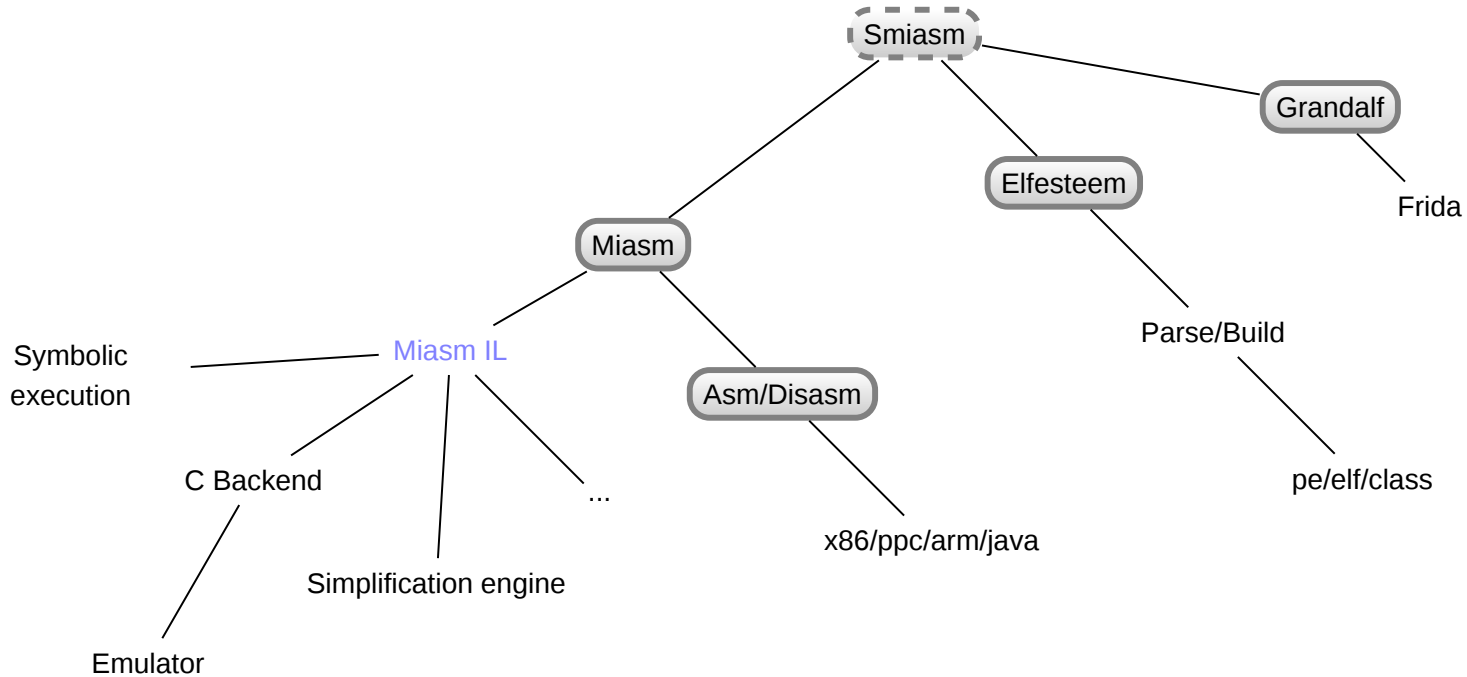


energie atomique • energies alternatives

Ça y est, vous êtes des fakirs de la manipulation assembleur !


Fakir niveau 2

energie atomique • energies alternatives



energie atomique • energies alternatives

## But

- travailler sur l'assembleur natif est difficile
- (plusieurs façons d'exprimer la même chose)
- dépendant de l'architecture
- simplicité (non spécialiste de l'assembleur)
- développer des algorithmes d'analyse de code
  - déobscurcissement de code
  - recherche de vulnérabilités
  - différence entre binaires (virus, patch, …)
  - preuve de code (nécessite des compétences de fakir)
  - …

⇒ Solution: traduire les assembleurs natif dans un langage intermédiaire, et travailler sur ce langage.

energie atomique • energies alternatives

## Mots du langage

- *ExprInt(valeur, taille)*
- *ExprId(nom, taille)*
- *ExprAff(dst, src)*
- *ExprCond(condition, valeur_si_vrai, valeur_si_faux)*
- *ExprMem(adresse, taille)*
- *ExprOp(nom_de_l_opérateur, opérande1, opérande2, ...)*
- *ExprSlice(src, bit_start, bit_stop)*
- *ExprCompose(expr1, expr2, ...)*

energie atomique • energies alternatives

## exemple

```
>>> from expression import *
>>> a = ExprId('A', 32)
>>> b = ExprId('B', 32)
>>> c = ExprId('C', 32)
>>> i = ExprInt(uint32(42))
>>> o = a+(b^i)
>>> o
<ExprOp 0x14909e0>
>>> print o
(A + (B ^ 0x2A))
>>> d = ExprAff(c, o)
>>> print d
C = (A + (B ^ 0x2A))
```

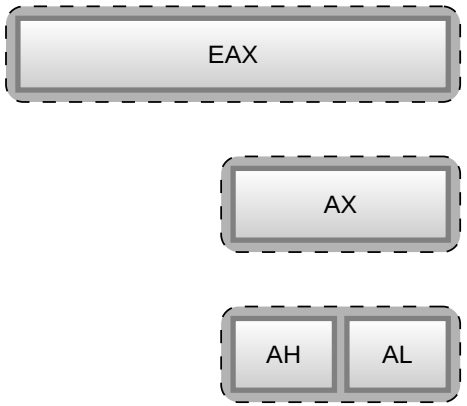
```
>>> m = ExprMem(b + a, 32)
>>> print m
@32[(B + A)]
>>> [str(x) for x in d.get_r()]
['A', 'B']
>>> [str(x) for x in m.get_r()]
['A', 'B', '@32[(B␣+␣A)]']
>>> print m.canonize()
@32[(A + B)]
```

```
a+(b^i) = ExprOp('+', a, ExprOp('^', b, i))
```

energie atomique • energies alternatives

## ExprSlice

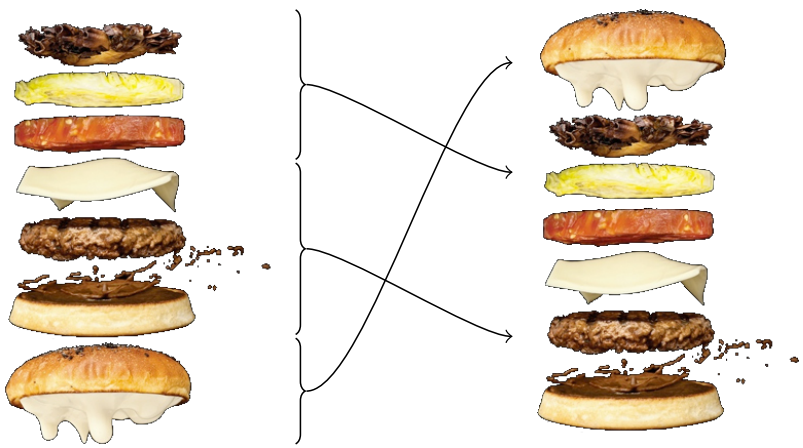
- Le *Slice* est une sous-partie des bits d'une expression
- Ici, AX est une tranche de EAX des bits de 0 à 15
- AH est la tranche des bits de 8 à 15 de EAX



energie atomique • energies alternatives

## ExprCompose

- le ExprCompose est l'inverse de ExprSlice
- il permet de prendre plusieurs expressions et positions
- puis de recomposer une expression
- un peu comme les tranches d'un hamburger




## Représentation des instructions

- une instruction est représentée par une liste d'effets de bord
- un effet de bord est représenté par une expression
- ces expressions sont exécutées en *parallèle*
  - ceci permet d'éviter des variables temporaires
  - ce point peut être discutable

energie atomique • energies alternatives

## Définition de quelques instructions

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

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

def update_flag_zf(a):
    return [ExprAff(zf, ExprOp('==', a, ExprInt_from(a, 0)))]

def update_flag_nf(a):
    return [ExprAff(nf, ExprOp('&',
                        get_op_msb(a), ExprInt_from(a, 1)))]
```

energie atomique • energies alternatives

## Points particuliers

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

Les expressions sont ici exécutées en parallèle : aucune variable temporaire n'est utilisée.

## Dans l'implémentation

- les variables *écrites* sont annotées comme sorties
- les variables *lues* sont annotées comme entrées

energie atomique • energies alternatives

## Le code devient

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

À la fin de l'instruction, l'état de sortie n'est exprimé qu'en fonction de l'état d'entrée des variables.


## Vérifier la sémantique : le cycle en V

- écrire un émulateur (JiT) qui utilise cette sémantique
- émuler des codes
- comparer avec une vraie trace d'exécution
- boucler dans la spirale de l'échec

## Autre utilisation

On peut émuler des API des bibliothèques en Python et réaliser une sandbox.

energie atomique • energies alternatives

## Exemple de fonction simulée

```
def ws2_32_WSASocketA():
    ret_ad = vm_pop_uint32_t()
    af = vm_pop_uint32_t()
    t = vm_pop_uint32_t()
    proto = vm_pop_uint32_t()
    protoinfo = vm_pop_uint32_t()
    g = vm_pop_uint32_t()
    flags = vm_pop_uint32_t()

    print whoami(), hex(ret_ad)

    regs = vm_get_gpreg()
    regs['eip'] = ret_ad
    regs['eax'] = 0x1337  # paye ton numero de socket
    vm_set_gpreg(regs)
```

energie atomique • energies alternatives

## Émulation d'un shellcode metasploit

```
serpilliere@pruneau:~/ python unpack_gen_light.py  -s -d -l -e msf_sc_connect_back.exe
call api ws2_32_WSAStartup 0x4010b2
ws2_32_WSAStartup 0x4010b2
call api ws2_32_WSASocketA 0x4010c1
ws2_32_WSASocketA 0x4010c1 0x2 0x1 0x0 0x0 0x0 0x0
call api ws2_32_connect 0x4010db
ws2_32_connect 0x4010db 0x1337 0x123fe4c 0x10
call api ws2_32_recv 0x4010f8
ws2_32_recv 0x4010f8 0x1337 0x123fe4c 0x4 0x0
call api kernel32_VirtualAlloc 0x40110b
kernel32_VirtualAlloc 0x40110b ( 0x0 0x10 0x1000 0x40 )
call api ws2_32_recv 0x401119
ws2_32_recv 0x401119 0x1337 0x20000000 0x10 0x0
hlt
```

energie atomique • energies alternatives

## Règles de simplifications

- $X + 0 \rightarrow X$
- $X + Y - Y \rightarrow X$
- $X \oplus X \rightarrow 0$
- $X + int1 + int2 \rightarrow X + int3$
- ...

## Exemple

```
>>> print o
(A + B)
>>> p = o - b
>>> print p
((A + B) - B)
>>> print expr_simp(p)
A
```

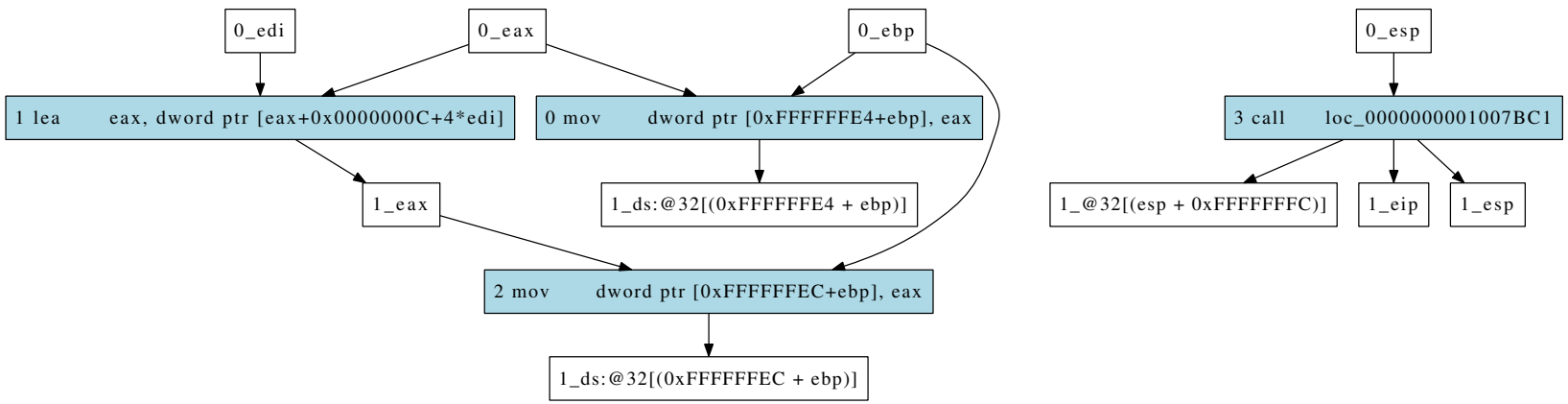
```
>>> q = (a - ExprInt32(1)) + ExprInt32(3)
>>> print q
((A - 0x1) + 0x3)
>>> print expr_simp(q)
(A + 0x2)
```

energie atomique • energies alternatives

## Flot de données

Utilisation de la sémantique pour retrouver le flot de données.

```
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
```




## État mémoire

- l'état de la machine est un dictionnaire
- les clefs sont soit des identifiants soit des cases mémoire
- les valeurs du dictionnaire sont les expressions qui représentent la valeur de ces clefs

## État initial : On définit les valeurs par défaut (éventuellement inconnues)

```
machine = eval_abs({esp:init_esp, ebp:init_ebp, eax:init_ecx,
  ebx:init_ebx, ecx:init_ecx+1, edx:ExprInt32(42),
  esi:init_esi, edi:init_edi, cs:ExprInt32(9), ...},
mem_read_wrap,
mem_write_wrap, )
```


## Exemple

```
>>> machine = eval_abs(dict(init_state))
>>> print machine.pool[ebx]
init_ebx
>>> my_eip, mem_dst = emul_full_expr_x(x86_mn.dis("\x43"), machine)
>>> print machine.pool[ebx]
(init_ebx + 0x1)
>>> print machine.pool[zf]
((init_ebx + 0x1) == 0x0)
```


## Principe

- déclaration des symboles sous notre contrôle
- ajout des informations à l'état de la machine
- émulation symbolique de *tout* le code par bloc de X instructions
- tests des conditions désirées sur l'état obtenu pour chaque bloc

energie atomique • energies alternatives

## Création de l'état mémoire

```
arg1 = ExprId('ARG1', 32, True)
arg2 = ExprId('ARG2', 32, True)
ret1 = ExprId('RET1', 32, True)
```

```
m = eval_abs({eax:init_eax,
              esp:init_esp,
              edi:init_edi, ...
```

## Contexte et vérification

```
eval_instr(mov(eax, arg2))
eval_instr(mov(ebp, esp))
eval_instr(sub(esp,
               ExprInt32(0x14)))
...
```

```
myesp = machine.pool[esp]
if arg2 in myeip and \
   arg2 in myesp:
    print 'w00t'
```

## Résultat

```
0x10002ccf
eip @32[ARG2]
esp (ARG2 + 0x4)
Instructions:
xchg        eax, esp
ret
```

energie atomique • energies alternatives

## Code obscurci

```
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
...
0x9652b0 xchg      ebp, dword ptr [esp]
0x9652b3 pop       esp
0x9652b4 movzx     eax, al
0x9652b7 jmp       dword ptr [4*eax+edi]
```

energie atomique • energies alternatives

## Code extrait

```
eax ((((@8[(@32[init_esp] - 0x9AAFFEC)] ^ @8[init_esp]) ^ 0xA4) + 0x15)_to[0:8], 0x0_to[8:32]
ebx (((@8[init_esp] - (((@8[(@32[init_esp] - 0x9AAFFEC)] ^ @8[init_esp]) ^ 0xA4) + 0x62)) + 0x
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)]
```

(note : aucun accès mémoire en écriture)

energie atomique • energies alternatives

## Mécanisme

- les chaînes de caractères sont ”chiffrées”
- la fonction de déchiffrement est appelée avec un pointeur en paramètre
- le compilateur ne génère pas toujours le même code

$\Rightarrow$ difficile à détecter

## Exemple de code équivalents

```
push    0x10331144

mov     dword ptr [esp], 0x10318F80

push    0x1031C160
mov     dword ptr [0x1033EB94], eax
mov     dword ptr [0x1033EB98], 0x00000001
add     esi, 0x00000044
```

energie atomique • energies alternatives

## Analyse automatique

- détecter les appels à la fonction
- trouver le *basic bloc* contenant l'appel
- exécuter en symbolique
- noter si l'argument sur la pile est bien un pointeur (entier)
- émuler la fonction avec cet argument
- patcher le binaire

## Récupération et test de l'argument

```
emul_bloc(machine, b_candidate)
if not esp in machine.pool:
    continue
arg_value = machine.eval_expr(ExprMem(machine.pool[esp]), {})
if isinstance(arg_value, ExprInt):
    arg_v = int(arg_value.arg)
    if e.is_in_virt_address(arg_v):
        break
```

energie atomique • energies alternatives

## Principe

- plus il y a de code, plus il y a de bugs
- on veut couvrir le plus de code possible
- on exécute symboliquement chaque *basic bloc*
- on extrait les conditions nécessaires pour passer par le plus de branches possibles
- TODO : on essaye de satisfaire ces conditions et on recommence (pour le moment, c'est artisanal)

energie atomique • energies alternatives

## Informations sur le contexte

```
argc = ExprId('argc')
argv = ExprId('argv')
ret_addr = ExprId('ret_addr')
machine.eval_instr(push(('u32', 'u32'), argv))
machine.eval_instr(push(('u32', 'u32'), argc))
machine.eval_instr(push(('u32', 'u32'), ret_addr))
```

## Code analysé :

```
push   ebp                                  | mov    eax, [ebp+0x8]
mov    ebp, esp                             | add    eax, 0x3
sub    esp, 0x34                            | mov    [ebp-00000024], eax
mov    dword ptr [ebp-4], 0x0               | mov    eax, [ebp-00000024]
cmp    dword ptr [ebp+0x8], 0x0             | cmp    eax, 0x45
jnz    0x1c                                 | jnz    0x90
```

## Conditions extraites

```
((argc == 0x0) == 0x0) = 0x0
(((argc & 0x30) == 0x0) == 0x1) = 0x1
((argc == 0x0) == 0x0) = 0x1
(((argc - 0x42) == 0x0) == 0x0) = 0x0
((((argc + argc) - 0xE) == 0x0) == 0x0) = 0x1
(((argc - 0x42) == 0x0) == 0x0) = 0x1
...
```

energie atomique • energies alternatives

Ça y est, vous êtes des fakirs de l'analyse de code !

Fakir niveau 8

energie atomique • energies alternatives

## Voies de développement

- amélioration sur le LI (opérateur - unaire, natif, …)
- scripts de recherche de vulnérabilités
- décompilation (retrouver le typage, les entrées d'une fonction)
- preuve de code : implémentation de domaines abstraits sur le LI (fakir++)

energie atomique • energies alternatives

## Miasm : le clou du fakir

Questions ?