# Binary deprotection with metasm

and stuff

Alexandre Gazet
Sogeti / ESEC R&D
alexandre.gazet(at)sogeti.com

Yoann Guillot
Sogeti / ESEC R&D
yoann.guillot(at)sogeti.com



## HITB 2009

# Plan



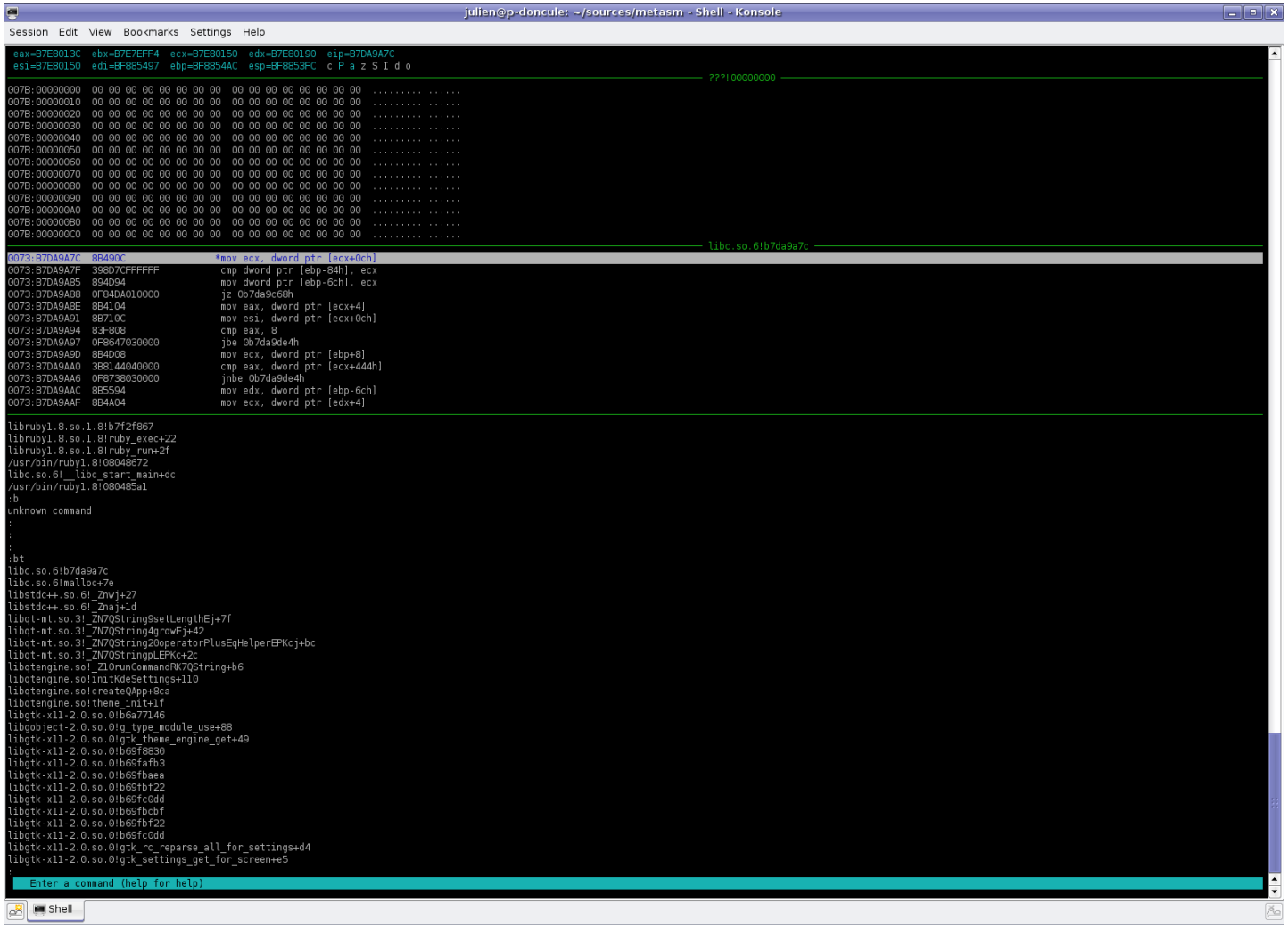
# Metasm

- a pure ruby opensource framework
- assembler/dissassembler
  - Ia32 (16/32/64bits), mips
  - Even supports cr7
- debugger
  - linux, windows, remote
- compiler/decompiler (more or less :)
- GUI included !

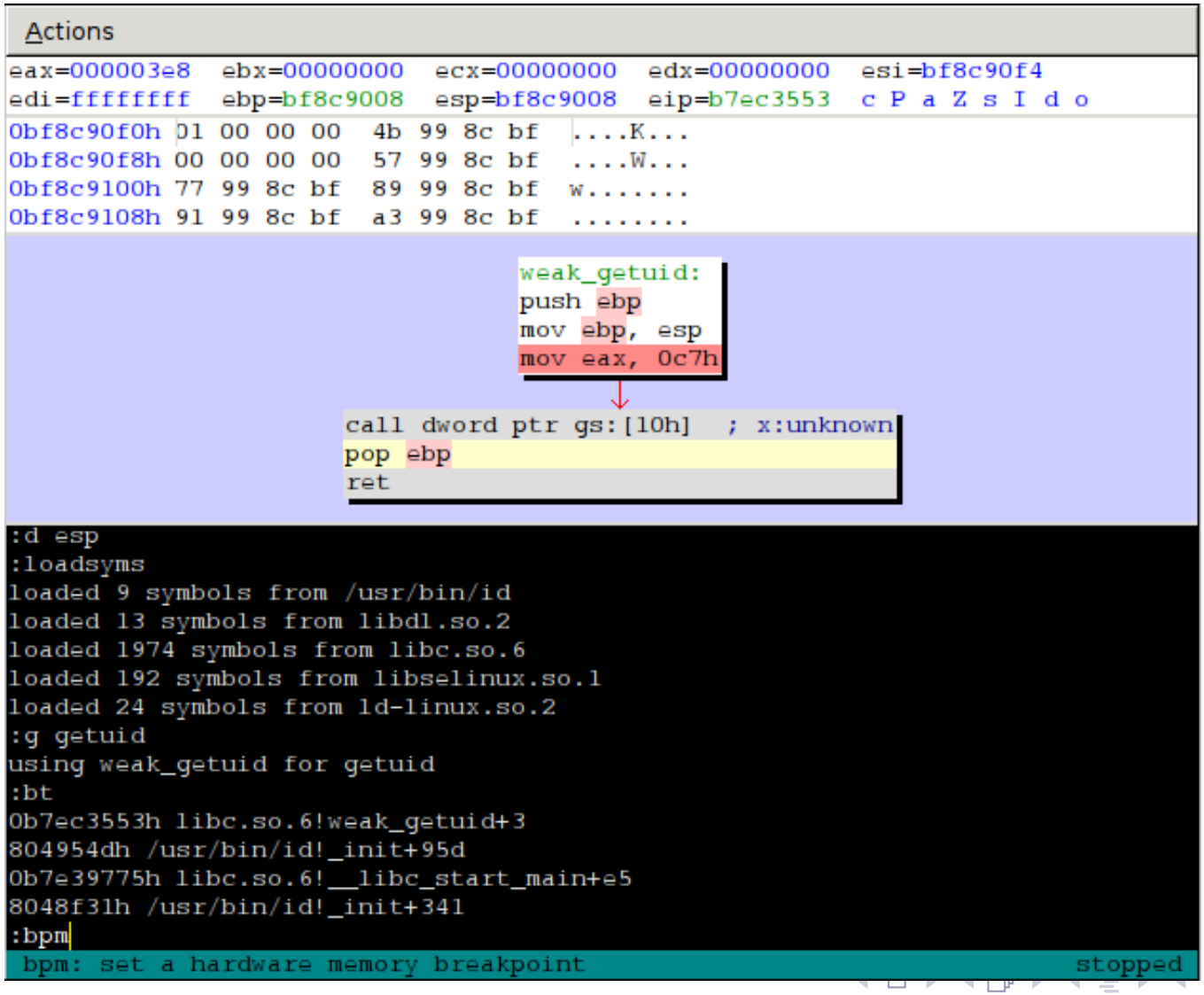
# Plan



# Debugger


julien@p-doncule: ~/sources/metasm - Shell - Konsole
Session Edit View Bookmarks Settings Help
eax=B7E8013C ebx=B7E7EFF4 ecx=B7E80150 edx=B7E80190 eip=B7DA9A7C
esi=B7E80150 edi=BF885497 ebp=BF8854AC esp=BF8853FC c P a z S I d o
???!00000000
libc.so.6!b7da9a7c
0073:B7DA9A7C 8B490C *mov ecx, dword ptr [ecx+0ch]
0073:B7DA9A7F 398D7CFFFFFF cmp dword ptr [ebp-84h], ecx
0073:B7DA9A85 894D94 mov dword ptr [ebp-6ch], ecx
0073:B7DA9A88 0F84DA010000 jz 0b7da9c68h
0073:B7DA9A8E 8B4104 mov eax, dword ptr [ecx+4]
0073:B7DA9A91 8B710C mov esi, dword ptr [ecx+0ch]
0073:B7DA9A94 83F808 cmp eax, 8
0073:B7DA9A97 0F8647030000 jbe 0b7da9de4h
0073:B7DA9A9D 8B4D08 mov ecx, dword ptr [ebp+8]
0073:B7DA9AA0 3B8144040000 cmp eax, dword ptr [ecx+444h]
0073:B7DA9AA6 0F8738030000 jnbe 0b7da9de4h
0073:B7DA9AAC 8B5594 mov edx, dword ptr [ebp-6ch]
0073:B7DA9AAF 8B4A04 mov ecx, dword ptr [edx+4]
libruby1.8.so.1.8!b7f2f867
libruby1.8.so.1.8!ruby_exec+22
libruby1.8.so.1.8!ruby_run+2f
/usr/bin/ruby1.8!08048672
libc.so.6!__libc_start_main+dc
/usr/bin/ruby1.8!080485a1
:b
unknown command
:bt
libc.so.6!b7da9a7c
libc.so.6!malloc+7e
libstdc++.so.6!_Znwj+27
libstdc++.so.6!_Znaj+1d
libqt-mt.so.3!_ZN7QString9setLengthEj+7f
libqt-mt.so.3!_ZN7QString4growEj+42
libqt-mt.so.3!_ZN7QString20operatorPlusEqHelperEPKcj+bc
libqt-mt.so.3!_ZN7QStringpLEPKc+2c
libqtengine.so!_Z10runCommandRK7QString+b6
libqtengine.so!initKdeSettings+110
libqtengine.so!createQApp+8ca
libqtengine.so!theme_init+1f
libgtk-x11-2.0.so.0!b6a77146
libgobject-2.0.so.0!g_type_module_use+88
libgtk-x11-2.0.so.0!gtk_theme_engine_get+49
libgtk-x11-2.0.so.0!b69f8830
libgtk-x11-2.0.so.0!b69fafb3
libgtk-x11-2.0.so.0!b69fbaea
libgtk-x11-2.0.so.0!b69fbf22
libgtk-x11-2.0.so.0!b69fc0dd
libgtk-x11-2.0.so.0!b69fbcbf
libgtk-x11-2.0.so.0!b69fbf22
libgtk-x11-2.0.so.0!b69fc0dd
libgtk-x11-2.0.so.0!gtk_rc_reparse_all_for_settings+d4
libgtk-x11-2.0.so.0!gtk_settings_get_for_screen+e5
Enter a command (help for help)
Shell

# Debugger


Actions
eax=000003e8 ebx=00000000 ecx=00000000 edx=00000000 esi=bf8c90f4
edi=ffffffff ebp=bf8c9008 esp=bf8c9008 eip=b7ec3553 c P a Z s I d o
0bf8c90f0h 01 00 00 00 4b 99 8c bf ....K...
0bf8c90f8h 00 00 00 00 57 99 8c bf ....W...
0bf8c9100h 77 99 8c bf 89 99 8c bf w.......
0bf8c9108h 91 99 8c bf a3 99 8c bf ........
weak_getuid:
push ebp
mov ebp, esp
mov eax, 0c7h
call dword ptr gs:[10h] ; x:unknown
pop ebp
ret
:d esp
:loadsyms
loaded 9 symbols from /usr/bin/id
loaded 13 symbols from libdl.so.2
loaded 1974 symbols from libc.so.6
loaded 192 symbols from libselinux.so.1
loaded 24 symbols from ld-linux.so.2
:g getuid
using weak_getuid for getuid
:bt
0b7ec3553h libc.so.6!weak_getuid+3
804954dh /usr/bin/id!_init+95d
0b7e39775h libc.so.6!__libc_start_main+e5
8048f31h /usr/bin/id!_init+341
:bpm
bpm: set a hardware memory breakpoint stopped

# Features

- Direct manipulation of the OS primitives
  - sys_ptrace
  - WaitForDebugEvent
- Very fine & low-level control
- Unified high-level interface
  - Linux, Windows, GDBserver
  - Conditionnal breakpoints, callback...

# Plan



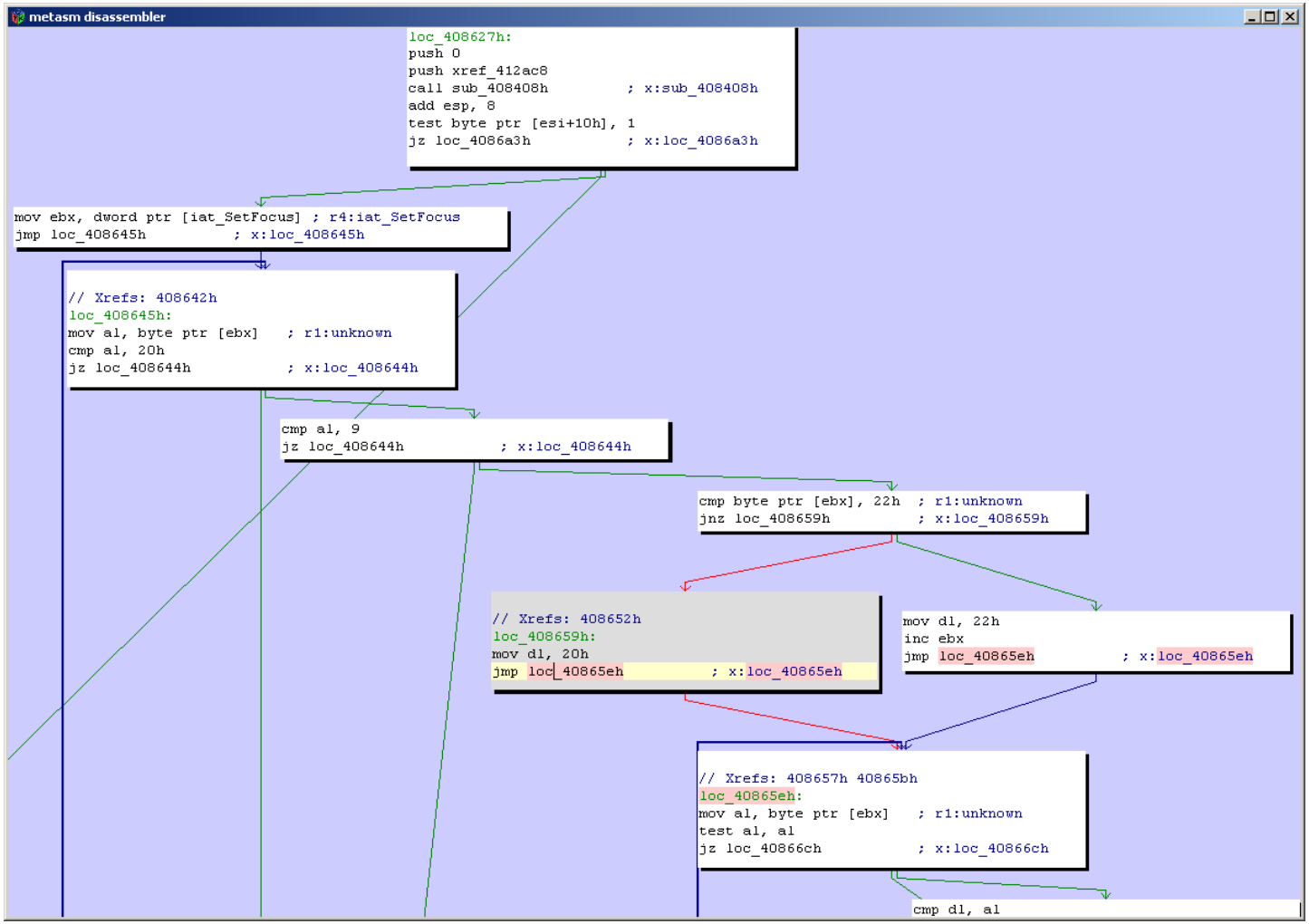
# C Compiler

- Rudimentary C compiler
- x86 only
- Framework integration easy to leverage
  - Easy to customize e.g. dynamic symbol resolution

# Plan



# Disassembler



# Disassembly

## The reference: **IDA Pro**

- Excellent on unobfuscated binaries
- Not so useful on protected code
  - No code interpretation
  - Strong hypothesis

## Hypothesis

- Both branches are taken on a conditionnal jump
- Two instructions never overlap
- A subfunction call returns

# Disassembly

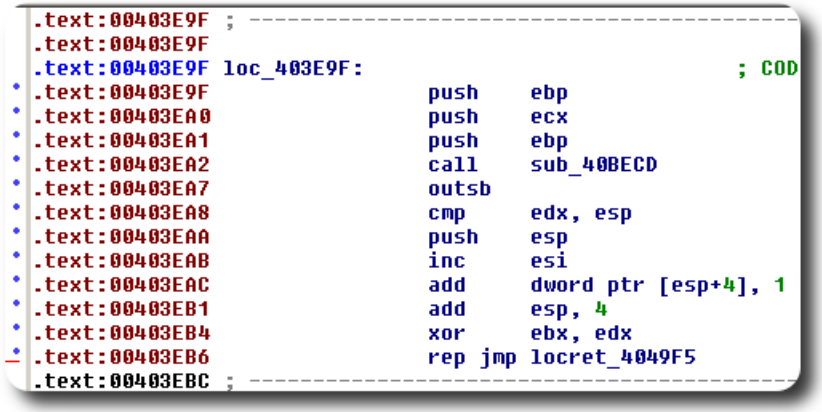
## The reference: **IDA Pro**

- Excellent on unobfuscated binaries
- Not so useful on protected code
  - No code interpretation
  - Strong hypothesis

## Hypothesis

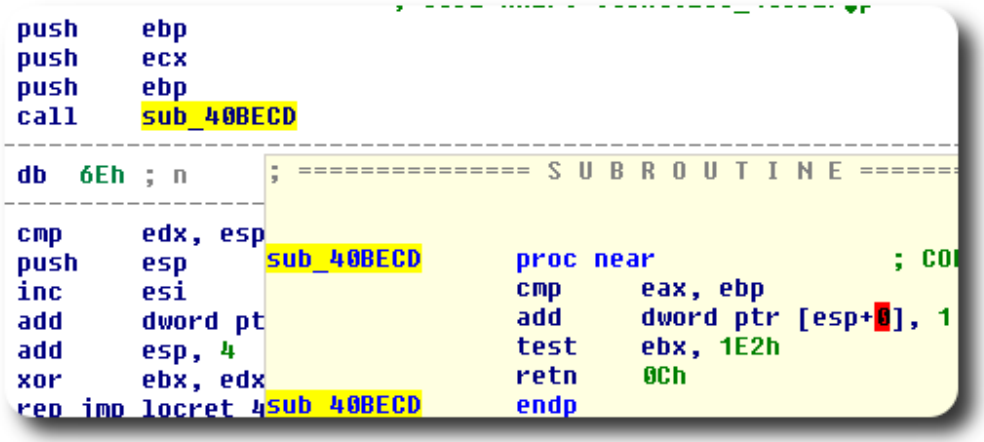
- Both branches are taken on a conditionnal jump
- Two instructions never overlap
- A subfunction call returns

# Hypothesis: all call returns




```
.text:00403E9F loc_403E9F:      ; CODE XREF: .text:loc_40CDEF
.text:00403E9F                  push    ebp
.text:00403EA0                  push    ecx
.text:00403EA1                  push    ebp
.text:00403EA2                  call    sub_40BECD
.text:00403EA7                  outsb
.text:00403EA8                  cmp     edx, esp
.text:00403EAA                  push    esp
.text:00403EAB                  inc     esi
```

# Failure




```
.text:0040BECD sub_40BECD      proc near      ; CODE XREF: .text:00

.text:0040BECD                 cmp     eax, ebp
.text:0040BECF                 add     dword ptr [esp+0], 1
.text:0040BED4                 test    ebx, 1E2h
.text:0040BEDA                 retn    0Ch

.text:0040BEDA sub_40BECD      endp
```

# Binding

**Our solution:**

Express instruction effects through symbolic expressions. This associates semantics to each instruction.

Instruction **ADD**:

```
res = Expression[[a[0], :&, mask], :+, [a[1], :&, mask]]
binding[a[0]] = res
binding[:eflag_z] = Expression[[res, :&, mask], :==, 0]
binding[:eflag_s] = sign[res]
binding[:eflag_c] = Expression[res, :>, mask]
binding[:eflag_o] = Expression[[sign[a[0]], :==, sign[a[1]]],
          :'&&', [sign[a[0]], :'!=', sign[res]]]
```

# Binding

**Our solution:**

Express instruction effects through symbolic expressions. This associates semantics to each instruction.

Instruction **ADD**:

```
res = Expression[[a[0], :&, mask], :+, [a[1], :&, mask]]
binding[a[0]] = res
binding[:eflag_z] = Expression[[res, :&, mask], :==, 0]
binding[:eflag_s] = sign[res]
binding[:eflag_c] = Expression[res, :>, mask]
binding[:eflag_o] = Expression[[sign[a[0]], :==, sign[a[1]]],
            :'&&', [sign[a[0]], :'!=', sign[res]]]
```

# Binding

**Instruction** `CALL`**:**

```
binding[:esp] = Expression[:esp, :-, opsz]
binding[ Indirection[:esp, 4] ] = di.next_addr
```

For exemple:

```
dword ptr [esp] = 0x403EA7
esp = esp-4
```

**Instruction** `RDTSC`**:**

```
binding[:eax] = Expression::Unknown
binding[:edx] = Expression::Unknown
```

# Binding

## Instruction CALL:

```
binding[:esp] = Expression[:esp, :-, opsz]
binding[ Indirection[:esp, 4] ] = di.next_addr
```

## For exemple:

```
dword ptr [esp] = 0x403EA7
esp = esp-4
```

## Instruction RDTSC:

```
binding[:eax] = Expression::Unknown
binding[:edx] = Expression::Unknown
```

# Binding

## Instruction CALL:

```
binding[:esp] = Expression[:esp, :-, opsz]
binding[ Indirection[:esp, 4] ] = di.next_addr
```

## For exemple:

```
dword ptr [esp] = 0x403EA7
esp = esp-4
```

## Instruction RDTSC:

```
binding[:eax] = Expression::Unknown
binding[:edx] = Expression::Unknown
```

# Backtracking, the theory

**Definition**

Symbolic emulation by walking the instruction flow backwards.

# Backtracking, the facts

## Execution flow:

```
    call loc_40becdh                ; @403ea2h   e826800000
[...]
    cmp eax, ebp                    ; @40becdh   39e8
    add dword ptr [esp+0], 1        ; @40becfh   8344240001
    test ebx, 1e2h                  ; @40bed4h   f7c3e2010000
    ret 0ch                         ; @40bedah   c20c00
```

**Backtracing** `x dword ptr [esp] for 40bedah ret 0ch`

1. backtrace `40becfh dword ptr [esp] => dword ptr [esp]+1`
2. backtrace up `40becdh->403ea2h dword ptr [esp]+1`
3. backtrace `403ea2h dword ptr [esp]+1 => 403ea8h`
4. backtrace result: `403ea8h`

# Metasm

## Result:

```
loc_403e9fh:
    push ebp                    ; @403e9fh  55
    push ecx                    ; @403ea0h  51
    push ebp                    ; @403ea1h  55
    call loc_40becdh            ; @403ea2h  e826800000  noreturn
db 6eh                          ; @403ea7h
// Xrefs: 40bedah
loc_403ea8h:
    cmp edx, esp                ; @403ea8h  39e2
    push esp                    ; @403eaah  54
[...]

// Xrefs: 403ea2h
loc_40becdh:
    cmp eax, ebp                ; @40becdh  39e8
    add dword ptr [esp+0], 1    ; @40becfh  8344240001
    test ebx, 1e2h              ; @40bed4h  f7c3e2010000
    ret 0ch                     ; @40bedah  c20c00  x:loc_403ea8h
```

# Plan



# Code protection?

- People want to be able to hide the functionning of their code
  - Using code obfuscation
  - Using code virtualization
  - Both

# Defeating code protection

- Need to understand the virtual machine architecture
- The VM interpreter is obfuscated
- Must defeat code obfuscation
- Generate a translator from virtual bytecode to easy to read code
- We'll introduce a framework to assist is those steps

# Previous work: T2 2007

- On the fly deobfuscation with patterns
- Virtual processor generator
  - Automatic semantics analysis

# The target

- A realworld virtualization based protection:
  - Some protected chunks of code are virtualized
  - Virtualized code (bytecode) is executed using an embedded interpreter
  - Each virtual machine instance is unique (polymorphism)
- Code is massively obfuscated
- List of all the instruction handlers is trivial to get

# Starting point

## Need of automation

- What we already have:
  - Filtering processor
  - Control flow graph (CFG) walking
  - Rewriting rules application
  - On-the-fly CFG modification

## Manual analysis of obfuscated code

- Trying to find patterns manually?
  - Painful
  - Loss of genericity
  - Possibly ineffective: polymorphism

# Plan



# Compiler approach

## Our needs

- Semantics preservation
- Rewrite code into a simpler form
- Dead code elimination
- etc.

- Compilers already do this kind of jobs: **optimization**
- Our optimization criteria: code conciseness

# Using compiler optimisation

## Proposed approach

- Addition of an optimization module to our CFG walkthrough module
- For each handler:
  1. Its code is recovered
  2. Then optimized

# Constant propagation

```
cfh  mov  al ,  12h
67h  mov  cl ,  46h
69h  xor  cl ,  al
```

```
cfh  mov  al ,  12h
67h  mov  cl ,  46h
69h  xor  cl ,  12h
```

Figure: Propagation of 12h through *al*.

# Constant folding

```
cfh  mov  al ,  12h
67h  mov  cl ,  46h
69h  xor  cl ,  12h
```

```
cfh  mov  al ,  12h
67h  mov  cl ,  54h
```

Figure: *cl* value folding.

# *Operation folding*

```
4fh  add  al, -7fh
51h  add  al,  bl
53h  add  al, -70h
```

```
4fh  add  al, 11h
51h  add  al,  bl
```

Figure: *add* operation folding.

# Demonstration

Optimization of a handler

# Next step

- Now we are ready to analyse the virtual machine architecture

# Plan



# At the beginning

## $2^{nd}$ Futamura projection

*Given two languages $L_a$ and $L_b$, it is possible to find a compiler from $L_b$ to $L_a$, if we know an interpreter of $L_b$ written in $L_a$*

- **Once again, compiler approach is the answer!**
- It's only a proof of existence
- How to generate it?

# Semantical analysis of the handlers

- `code_binding`: method from the `Disassembler` object

**Example of an optimized handler**

```
lodsd
xor eax, ebx
add eax, 859fcfaeh
sub ebx, eax
push eax
```

- Semantics (aka *binding*)

```
dword ptr [esp] := (dword ptr [esi]^ebx)+859fcfaeh
eax := (dword ptr [esi]^ebx)+859fcfaeh
ebx := ebx+-(dword ptr [esi]^ebx)-859fcfaeh
esi := esi+4
esp := esp-4
```

# Semantical analysis

- When a handler is encountered for the first time:
  - Its code is optimized
  - Its semantics is computed and stored
- We progressively build **the description of the semantics of the interpreter** !
- Given the current program state, we are able to emulate the next step of bytecode.

# From static to (almost) dynamic

- Handler's binding:

```
dword ptr [esp] := (dword ptr [esi]^ebx)+859fcfaeh
eax := (dword ptr [esi]^ebx)+859fcfaeh
ebx := ebx-(dword ptr [esi]^ebx)-859fcfaeh
esi := esi+4
esp := esp-4
```

- Current context (partial)

```
eax := 93h
ebx := 0fd8dh
esi := 100167beh
[...]
```

- Contextualized binding:

```
dword ptr [esp] := 0c0000001h
eax := 0c0000001h
ebx := 4000fd8ch
esi := 100167c2h
esp := esp-4
```

# From static to (almost) dynamic (2)

Based on contextualized binding:

- Generate corresponding assembly:

```
push 0c0000001h
```

- **Symbolic execution**: compute context after execution of the current handler
- Follow bytecode execution flow
  - Support virtual calls and jumps (conditionnal or not)

⇒ **Recover the whole chunk of code in native `Ia32` assembly**

# Demonstration

Symbolic execution and assembly generation

# Results

- The whole chunk of bytecode is compiled, on-the-fly, into native `Ia32` assembly
- Compiled bytecode itself is still obfuscated
- Still many references to the virtual machine's context
- Looks like a stack automaton

**Next steps**

$\Rightarrow$ **Re-use optimisation module + inject abstraction**

# Abstraction injection

## Processor extension

```
list = Reg.i_to_s[32].concat( %w[virt_eax])
Reg.i_to_s[16].concat( %w[virt_ax])
Reg.i_to_s[8].concat( %w[virt_al])

Reg.s_to_i.clear
Reg.i_to_s.each { |sz, hh|
   hh.each_with_index { |r, i|
      Reg.s_to_i[r] = [i, sz]
   }
}
Reg::Sym.replace list.map { |s| s.to_sym }
```

# Demonstration

Optimized chunk with virtual registers

# Final pass

- Injection of virtual registers
- Applying optimizations $\Rightarrow$
  - Stack automaton aspect totally removed
  - Code is expressed using virtual registers only
- Virtual registers are then mapped back to native registers
- Compilation and links edition

$\Rightarrow$ original, unprotected chunk of code is retrieved

# Plan



# Demonstration

Unvirtualized code, mapped into the original binary

# Plan



# Conclusion(s) 1/2

- **Optimisation** (rewriting rules)
  - Quite effective
  - Our implementation is limited
    - Local optimizations
    - Lake of an intermediate representation
    - Unsuited to control flow obfuscation
- **Partial evaluation or specialization**
  - Pre-computation of all the static elements:
    - Data transfers within the obfuscated code
    - Application of the interpreter to the bytecode
  - Generic Approach
  - Require relatively heavy computation

# Conclusion(s) 2/2

- **Integration and re-usability of the deobfuscator**
  - Actual code is still a "prototype"
  - Being integrated into the framework as a plugin
  - Usable on x86 code, with some cross-platform parts

# Plan



# Advantages

- Arch-specific code reduced to the minimum
- C code is much more expressive than asm
- Standard loops are simpler to handle
- Semantics is often simpler
  - No side-effects on flags

# Limitations

- Some asm constructs are difficult to express in C
  - *rol*, *ror*
  - *jmp eax*
- Needs that the code has certain properties
  - Split in function/subfunctions
  - Follows C ABIs/calling conventions
- Those last things can be worked around
  - Custom __attribute__

# Demo

## Decompilation

# Reminder

- **Metasm**
  `https://metasm.cr0.org/`
- **Blog**
  `http://esec.fr.sogeti.com/blog/`

# Conclusion

**Thanks for listening.**

**Questions ?**