

R2M2

RADARE2 + MIASM2 = ♥

@guedou - 09/09/2016



@GUEDOU?

- French
- hobbyist reverser
- network security researcher
 - IPv6, DNS, TLS, BGP, DDoS mitigation, ...
- [Scapy](#) co-maintainer
 - Python-based packet manipulation program & library
- neither a [radare2](#) nor [miasm2](#) power user

I needed to implement a rare CPU architecture **easily**

Back in December 2015, only objdump knew this architecture

```
binutils$ ./objdump -m mep -b binary -D mister.bin
```

```
mister.bin:      file format binary
```

Disassembly of section .data:

```
00000000 <.data>:
```

```
      0:      08 d8 01 00      jmp 0x100
```

```
      4:      18 df 08 00      jmp 0x8e2
```

```
[...]
```

```
67c4a:      b0 6f      add $sp,-20
```

```
67c4c:      1a 70      ldc $0,$lp
```

```
67c4e:      12 48      sw $8,0x10($sp)
```

```
67c50:      0e 47      sw $7,0xc($sp)
```

```
67c52:      0a 46      sw $6,0x8($sp)
```

```
67c54:      06 40      sw $0,0x4($sp)
```

```
67c56:      10 07      mov $7,$1
```

```
67c58:      a3 bf      bsr 0x67bfa
```

```
67c5a:      ff 5c      mov $12,-1
```

```
67c5c:      c1 e0 24 00  beq $0,$12,0x67ca4
```

```
67c60:      86 d1 f5 cc  movu $1,0xccf586
```

R2M2 GOALS?

r2m2 is a radare2 plugin that aims to:

- use [radare2](#) as a frontend to [miasm2](#)
 - tools, GUI, shortcuts, ...
- use [miasm2](#) as a backend to radare2
 - asm/dis engine, symbolic execution, ...
- be architecture independent

MIASM 101

WHAT IS MIASM?

Python-based reverse engineering framework with many features:

- assembling / disassembling x86 / ARM / MIPS / SH4 / MSP430
- representing assembly semantic using intermediate language
- emulating using JIT
- ...

See the [official blog](#) for examples and demos

ASSEMBLING

```
# Create a x86 miasm machine
>>> from miasm2.analysis.machine import Machine
>>> m = Machine("x86_32")

# Get the mnemonic object
>>> mn = m.mn()

# Convert to an internal miasm instruction
>>> instr = mn.fromstring("MOV AX, 1", 32)

# Assemble all variants
>>> mn.asm(instr)
['f\xb8\x01\x00', 'fg\xb8\x01\x00', 'f\xc7\xc0\x01\x00',
'fg\xc7\xc0\x01\x00']
```


DISASSEMBLING

```
# Disassemble all variants
>>> [str(mn.dis(x, 32)) for x in mn.asm(instr)]
['MOV      AX, 0x1',
 'MOV      AX, 0x1',
 'MOV      AX, 0x1',
 'MOV      AX, 0x1']
```

MIASM INTERMEDIATE LANGUAGE

```
# Disassemble a simple ARM instruction
>>> m = Machine("arm1")
>>> instr = m.mn.dis("002088e0".decode("hex"), "1")

# Display internal instruction arguments
>>> instr.name, instr.args
('ADD', [ExprId('R2', 32), ExprId('R8', 32), ExprId('R0', 32)])

# Get the intermediate representation architecture object
>>> ira = m.ira()

# Get the instruction miasm intermediate representation
>>> ira.get_ir(instr)
([ExprAff(ExprId('R2', 32),
            ExprOp('+', ExprId('R8', 32), ExprId('R0', 32))), []])
```

SYMBOLIC EXECUTION

```
# Add the instruction to the current block
>>> ira.add_instr(instr)

# Display the IR block
>>> for label, bloc in ira.blocs.items():
...     print bloc
...
loc_0000000000000000:0x00000000
    R2 = (R8+R0)

    IRDst = loc_0000000000000004:0x00000004
```



```
# Import miasm expression objects
>>> from miasm2.expression.expression import ExprId, ExprInt32

# Affect a value to R0
>>> s.symbols[ExprId("R0", 32)] = ExprInt32(0)
>>> r = s.emul_ir_bloc(ira, 0)
>>> s.dump_id()
R2 R8_init # the expression was simplified
[..]

# Affect a value to R8
>>> s.symbols[ExprId("R8", 32)] = ExprInt32(0x2807)
>>> r = s.emul_ir_bloc(ira, 0)
>>> s.dump_id()
R2 0x2807 # R0 + R8 = 0 + 0x2807
[..]
```

EMULATION / JIT

Let's build a simple binary to emulate

```
$ cat add.c
int add (int a, int b) { return a+b; }
main () { printf ("add (): %d\n", add (1, 2)); }

$ gcc -m32 -o add add.c

$ ./add
add(): 3
```

Then, build a miasm sandbox to emulate add ()

```
$ cat sandbox_r2con.py
from miasm2.analysis.sandbox import Sandbox_Linux_x86_32

# Parse arguments
parser = Sandbox_Linux_x86_32.parser(description="ELF sandboxer")
parser.add_argument("filename", help="ELF Filename")
options = parser.parse_args()

# Create sandbox
sb = Sandbox_Linux_x86_32(options.filename, options, globals())

# Get the address of add()
addr = sb.elf.getsectionbyname(".symtab").symbols["add"].value

# /\ the last part of the code is on the next slide /\ #
```


Finally, emulate add ()

```
$ python sandbox_r2con.py ./add
080483E4 PUSH      EBP
080483E5 MOV       EBP, ESP
080483E7 MOV       EAX, DWORD PTR [EBP+0xC]
080483EA MOV       EDX, DWORD PTR [EBP+0x8]
080483ED ADD       EAX, EDX
080483EF POP       EBP
080483F0 RET

add() : 0x2807
```

GDB SERVER

```
$ python sandbox_r2con.py ./add -g 2807  
Listen on port 2807
```

```
$ gdb  
(gdb) target remote localhost:2807  
Remote debugging using localhost:2807  
0x080483ff in ?? ()  
(gdb) info registers eip eax  
eip                0x080483ff          0x080483ff  
eax                0x0                0  
  
(gdb) c  
Continuing.  
  
Program received signal SIGTRAP, Trace/breakpoint trap.  
0x1337beef in ?? ()  
(gdb) info registers eip eax  
eip                0x1337beef          0x1337beef  
eax                0x3                3
```

ADDING A NEW ARCHITECTURE TO MIASM

HIGH-LEVEL CHECKLIST

1. registers in *miasm2/arch/ARCH/regs.py*
2. opcodes in *miasm2/arch/ARCH/arch.py*
3. semantic in *miasm2/arch/ARCH/sem.py*

ADDING A NEW OPCODE IN ARCH.PY

MIPS ADDIU

Encoding 001001 ss ssst tttt iiiiii iiiiii iiiiii

The opcode is defined as:

```
addop("addiu", [bs("001001"), rs, rt, s16imm], [rt, rs, s16imm])
```

The arguments are defined as:

```
rs = bs(l=5, cls=(mips32_gpreg,))  
rt = bs(l=5, cls=(mips32_gpreg,))  
s16imm = bs(l=16, cls=(mips32_s16imm,))
```

*mips32_** objects implement `encode()` and `decode()` methods that return miasm expressions!

ADDING A NEW OP CODE IN SEM.PY

Solution#1 - Implement the logic with miasm expressions

```
def addiu(ir, instr, reg_dst, reg_src, imm16):  
  
    expr_src = ExprOp("+", reg_src, imm16.zeroExtend(32))  
  
    return [ExprAff(reg_dst, expr_src)], []
```

Solution#2 - Be lazy, and implement using the *sembler*

```
@sbuild.parse
def addiu(reg_dst, reg_src, imm16):
    reg_dst = reg_src + imm16
```


The resulting expression is:

```
>>> ir.get_ir(instr) # instr being the IR of "ADDIU A0, A1, 2"
([ExprAff(ExprId('A0', 32), ExprOp('+', ExprId('A1', 32),
ExprInt(uint32(0x2L))))], [])
```

R2 PLUGINS IN PYTHON

RADARE2-BINDINGS BASED PLUGINS

```
$ cat radare2-bindings_plugin_ad.py

from miasm2.analysis.machine import Machine
import r2lang

def miasm_asm(buf):
    # [...]
    return asm_str

def miasm_dis(buf):
    # [...]
    return [dis_len, dis_str]

# /\ the last part of the code is on the next slide /\ #
```


Quite easy to use

```
$ r2 -i radare2-bindings_plugin_ad.py /bin/ls -qc 'e asm.arch=miasm; pd 5'
;-- entry0:
0x004049de      31ed      XOR      EBP, EBP
0x004049e0      4989d1    MOV      R9, RDX
0x004049e3      5e        POP      RSI
0x004049e4      4889e2    MOV      RDX, RSP
0x004049e7      4883e4f0  AND      RSP, 0xFFFFFFFFFFFFFFFF0
```

As of today, only *assembly* and *disassembly* plugins can be implemented

CFFI BASED PLUGINS

More steps must be taken:

1. call Python from C
2. access r2 structures from Python
3. build a r2 plugin

The **CFFI** Python module produces a .so!

STEP#1 - CALL PYTHON FROM C

Example: convert *argv[1]* in base64 from Python

1 - C side of the world

```
$ cat test_cffi.h
char* base64(char*); // under the hood, a Python function will be called

$ cat test_cffi.c
#include <stdio.h>
#include "test_cffi.h"

int main(int argc, char** argv)
{
    printf("[C] %s\n", base64(argc>1?argv[1]:"r2con"));
}
```

2 - Python side of the world

```
$ cat cffi_test.py

import cffi
ffi = cffi.FFI()

# Declare the function that will be exported
ffi.embedding_api("".join(open("test_cffi.h").readlines()))

# /\ the last part of the code is on the next slide /\ #
```


3 - compile

```
$ python cffi_test.py # build python_embedded.so  
$ gcc -o test_cffi test_cffi.c python_embedded.so
```

4 - enjoy

```
$ LD_LIBRARY_PATH=./ ./test_cffi cffi
[P] cffi
[C] Y2ZmaQ==

$ LD_LIBRARY_PATH=./ ./test_cffi
[P] r2con
[C] cjJjb24=
```

STEP#2 - ACCESS R2 STRUCTURES FROM PYTHON

- can't simply use `set_source()` on all r2 headers
 - CFFI C parser ([pycparser](#)) does not support all C extensions / dialects
- must *prepare* headers with alternative solutions:
 - use a C preprocessor, aka `gcc -E`
 - use [pycparser and fake headers](#)
 - automatically extract r2 plugins structures
 - ↑ r2m2 does that ↑

In a nutshell

```
// c
RAnalOp test;
set_type((RAnalOp_cffi*)&test, 0x2806);
printf("RAnalOp.type: 0x%x\n", test.type);
```

```
# Python
@ffi.def_extern()
def set_type(r2_op, value):
    r2_analop = ffi.cast("RAnalOp_cffi*", r2_op)
    r2_analop.type = value + 1
```

```
shell$ LD_LIBRARY_PATH=./ ./test_r2
RAnalOp.type: 0x2807
```

See r2m2 source code for a whole example

STEP#3 - BUILD A R2 PLUGIN

The r2 Wiki shows how to make a [r_asm](#) plugin

```
#include <r_asm.h>
#include <r_lib.h>
#include "r2_cffi.h"
#include "cffi_ad.h"

static int disassemble(RAsm *u, RAsmOp *o, const ut8 *b, int l) {
    python_dis(b, l, (RAsmOp_cffi*)o);
    return o->size;
}

static int assemble(RAsm *u, RAsmOp *o, const char *b) {
    python_asm(b, (RAsmOp_cffi*)o);
    return p->size;
}

// /\ the following part of the code is on the next slide /\
```

```
// /\ the first part of the code is on the previous slide /\
```

```
RAsmPlugin r_asm_plugin_cffi = {  
    .name = "cffi",  
    .arch = "cffi",  
    .license = "LGPL3",  
    .bits = 32,  
    .desc = "cffi",  
    .disassemble = disassemble,  
    .assemble = assemble  
};
```

```
// /\ the following part of the code is on the next slide /\
```

```
// /\ the other parts of the code are on the previous slides

#ifndef CORELIB
struct r_lib_struct_t radare_plugin = {
    .type = R_LIB_TYPE_ASM,
    .data = &r_asm_plugin_cffi
};
#endif
```


R2M2

(at last!)

WHAT IS R2M2?

- uses everything described so far to bring miasm2 to radare2!
- keeps most of the smart logics in miasm2
 - r2m2 aims to be architecture independent
 - uses the R2M2_ARCH env variable to specify the arch
- provides two r2 plugins:
 - ad: assembly & disassembly
 - Ae: Analysis & esil

```
r2m2$ rasm2 -L |grep r2m2
adAe  32          r2m2          LGPL3    miasm2 backend
```

R2M2_AD - THE EASY PLUGIN

- simple CFFI / C wrapper around a miasm2 Machine()
- provides miasm2 assembly & disassembly features to radare2

MIPS32 assembly/disassembly with rasm2:

```
r2m2$ export R2M2_ARCH=mips32l; rasm2 -a r2m2 'addiu a0, a1, 2' |rasm2 -a r2m2 -d -  
ADDIU      A0, A1, 0x2
```

miasm2 MSP430 in r2 with random instructions:

```
r2m2$ R2M2_ARCH=msp430 r2 -a r2m2 -qc 'wOR; pd 5' -  
0x00000000      07fa      and.w      R10, R7  
0x00000002      47ad      dadd.b      R13, R7  
0x00000004      f05e0778    add.b      @R14+, 0x7807(PC)  
0x00000008      f46d81ed    addc.b     @R13+, 0xED81(R4)  
0x0000000c      3fdc      bis.w      @R12+, R15
```

miasm2 x86-64 on /bin/ls:

```
r2m2$ R2M2_ARCH=x86_64 r2 -a r2m2 /bin/ls -qc 'pd 7 @0x00404a1c'
0x00404a1c      4883f80e      CMP      RAX, 0xE
0x00404a20      4889e5      MOV      RBP, RSP
0x00404a23      761b      JBE      0x1D
0x00404a25      b800000000  MOV      EAX, 0x0
0x00404a2a      4885c0      TEST     RAX, RAX
0x00404a2d      7411      JZ      0x13
0x00404a2f      5d      POP      RBP
```

Where does these jumps go?

R2M2_AE - THE CHALLENGING ONE

Use miasm2 to automatically

- find branches
- find function calls
- split blocks
- emulate instructions
- ...

HOW?

Step#1 - use miasm2 expressions and internal methods

- `breakflow()`, `dstflow()`, `is_subcall()`

```
# r2m2 incomplete example
if instr.is_subcall():
    if isinstance(instr.arg, ExprInt):
        analop.type = R_ANAL_OP_TYPE_CALL
        analop.jump = address + int(instr.arg)
    else:
        analop.type = R_ANAL_OP_TYPE_UCALL
```

A simple MIPS32 output

```
r2m2$ R2M2_ARCH=mips32b rasm2 -a r2m2 'j 0x4; nop' -B > j_nop.bin
```

```
r2m2$ R2M2_ARCH=mips32b r2 -a r2m2 j_nop.bin -qc 'pd 2'
      ,=< 0x00000000      08000001      J      0x4
      `-> 0x00000004      00000000      NOP
```


A more complex output - r2 vs r2m2

```

r2$ r2 /bin/ls -qc 'pd 12 @0x00404a1c'
      0x00404a1c      4883f80e      cmp rax, 0xe
      0x00404a20      4889e5      mov rbp, rsp
      ,=< 0x00404a23      761b      jbe 0x404a40
      | 0x00404a25      b800000000    mov eax, 0
      | 0x00404a2a      4885c0      test rax, rax
      ,==< 0x00404a2d      7411      je 0x404a40
      || 0x00404a2f      5d      pop rbp
      || 0x00404a30      bf60e66100    mov edi, loc.__edata      ; loc.__edata
      || 0x00404a35      ffe0      jmp rax
      || 0x00404a37      660f1f840000.  nop word [rax + rax]
      ``-> 0x00404a40      5d      pop rbp
      0x00404a41      c3      ret

```

```

r2m2$ R2M2_ARCH=x86_64 r2 -a r2m2 /bin/ls -qc 'pd 12 @0x00404a1c'
      0x00404a1c      4883f80e      CMP      RAX, 0xE
      0x00404a20      4889e5      MOV      RBP, RSP
      ,=< 0x00404a23      761b      JBE      0x1D
      | 0x00404a25      b800000000    MOV      EAX, 0x0
      | 0x00404a2a      4885c0      TEST     RAX, RAX
      ,==< 0x00404a2d      7411      JZ       0x13
      || 0x00404a2f      5d      POP      RBP
      || 0x00404a30      bf60e66100    MOV      EDI, loc.__edata
      || 0x00404a35      ffe0      JMP      RAX
      || 0x00404a37      660f1f840000.  NOP      WORD PTR [RAX*0x2]
      ``-> 0x00404a40      5d      POP      RBP
      0x00404a41      c3      RET

```

Step#2 - convert miasm2 expression to radare2 ESIL

- both achieve the same goal: express instructions semantics
- simple automatic conversions are possible

```
m2 expr -> ExprAff(ExprId("R0", 32), ExprInt(0x2807, 32))  
r2 esil -> 0x2807,r0,=
```

- need to dynamically define the radare2 registers profile
 - done thanks to CFFI and miasm2
- some instructions are problematic, as their semantics are complex
 - radare2 limits ESIL to be less than 64 bytes long

What to do with long ESIL expressions?

- drop them
 - weird solution
- truncate them
 - difficult to predict the outcome, but today r2m2 does that
- try to simplify them in r2
 - ↑ r2m2 should do that, sooner or later ↑

A simple MIPS32 output

```
r2m2$ R2M2_ARCH=mips32b rasm2 -a r2m2 'j 0x4; nop' -B > j_nop.bin
```

```
r2m2$ R2M2_ARCH=mips32b r2 -a r2m2 j_nop.bin -qc 'e asm.emu=true; pd 2'
      ,=< 0x00000000      08000001      J      0x4      ; pc=0x4
      `-> 0x00000004      00000000      NOP      n
```

A more complex output

```
R2M2_ARCH=x86_64 r2 -a r2m2 /bin/ls -qc 'e asm.emu=true; pd 12 @0x00404a1c'
      0x00404a1c      4883f80e      CMP      RAX, 0xE      ; zf=0x0
      0x00404a20      4889e5      MOV      RBP, RSP      ; rbp=0x0
,=< 0x00404a23      761b      JBE      0x1D      ; unlikely
| 0x00404a25      b800000000 MOV      EAX, 0x0      ; eax=0x0
| 0x00404a2a      4885c0      TEST     RAX, RAX      ; zf=0x1 -> 0
,==< 0x00404a2d      7411      JZ       0x13      ; unlikely
|| 0x00404a2f      5d      POP      RBP      ; rsp=0x8 ; r
|| 0x00404a30      bf60e66100 MOV      EDI, loc._edata ; edi=0x61e66
|| 0x00404a35      ffe0      JMP      RAX      ; rip=0x0
|| 0x00404a37      660f1f840000. NOP      WORD PTR [RAX*0x2]
``-> 0x00404a40      5d      POP      RBP      ; rsp=0x10 ->
      0x00404a41      c3      RET      ; rsp=0x18 ->
```

CURRENT ISSUES & FUTURE WORK

- truncated ESIL: simplify with `m2.expr_simp()`
- calling conventions: specify them dynamically
- redesign r2m2 as regular Python module
 - ease code reuse (for Python or r2pipe plugins)
 - ease unit & regression tests
- add r2m2 to r2pm

CONCLUDING REMARKS

- miasm2 and radare2 are powerful tools
 - combining them turned out to be efficient
- r2m2 is more than *"PoC that works on my laptop"*

```
$ docker run --rm -it -e 'R2M2_ARCH=mips321' guedou/r2m2  
"rasm2 -a r2m2 'addiu a0, a1, 2'"
```

- too good to be true?
 - could be, yet r2m2 is better than nothing

Today, in September 2016, r2m2 allows me to get call graphs

[0x00067c4a]> VV @ fcn.00067c4a (nodes 12 edges 15 zoom 100%) BB-NORM mouse:canvas-y movements-speed:5

```
[0x67c4a]
(fcn) fcn.00067c4a 4348
; arg int arg_4h @ sp+0x4
; arg int arg_8h @ sp+0x8
; arg int arg_ch @ sp+0xc
; arg int arg_10h @ sp+0x10
; arg int arg_20h @ sp+0x20
; arg int arg_2ch @ sp+0x2c
; arg int arg_30h @ sp+0x30
; arg int arg_34h @ sp+0x34
; arg int arg_38h @ sp+0x38
; arg int arg_3ch @ sp+0x3c
; arg int arg_40h @ sp+0x40
ADD SP, -20; sp=0x3ffffec -> 0xffffffff0
LDC R0, LP; r0=0x0
SW R8, 0x10(SP)
SW R7, 0xc(SP)
SW R6, 0x8(SP)
SW R0, 0x4(SP)
MOV R7, R1; r7=0x0
BSR 0xFA2 ;[a]; lp=0x67c5c -> 0x2400c100; CALL: 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
MOV R12, -1; r12=0xffffffff -> 0xffffffff0
BEQ R0, R12, 0x48 ;[b]; unlikely
```

```
0x67c60
MOVU R1, 0xCCF586; r1=0xccf586 -> 0xffffffff00
BSR 0x17F20 ; fcn.strlen ;[d]; lp=0x67c6c -> 0xb9d84f00; pc=0x7fb88 -> 0x3001900; CALL:
MOV R8, R0; r8=0x0
MOVU R1, 0xCE4FEC; r1=0xce4fec -> 0xffffffff00
BSR 0x17F16 ; fcn.strlen ;[d]; lp=0x67c76 -> 0x69d85000; pc=0x7fb88 -> 0x3001900; CALL:
ADD3 R8, R0, R8; r8=0x0
MOVU R1, 0xCE5002; r1=0xce5002 -> 0xffffffff00
BSR 0x17F0C ; fcn.strlen ;[d]; lp=0x67c80 -> 0x69dd0100; pc=0x7fb88 -> 0x3001900; CALL:
ADD3 R8, R0, R8; r8=0x0
ADD3 R1, R8, 0x1; r1=0x1 -> 0xdf010000
BSR 0xD4AC ;[e]; lp=0x67c8a -> 0x51ce7200; pc=0x75132 -> 0xe471200; CALL: 0x80018df, 0x0, 0x0, 0x0
MOV R6, R0; r6=0x0
BNEZ R6, 0x20 ;[f]; unlikely
```

```
0x67ca8
MOVU R2, 0xCE519E; r2=0xce519e -> 0xffffffff00
MOVU R3, 0xCE4FEC; r3=0xce4fec -> 0xffffffff00
MOV R1, R6; r1=0x0
MOVU R4, 0xCE5002; r4=0xce5002 -> 0xffffffff00
SW R4, (SP)
MOVU R4, 0xCCF586; r4=0xccf586 -> 0xffffffff00
BSR 0xFAAE02 ;[g]; lp=0x67cc4 -> 0x548000; pc=0x1012ac2 -> 0xffffffff00; CALL: 0x80018df, 0x0, 0x0, 0x0
MOV R1, R7; r1=0x0
MOV R2, R6; r2=0x0
MOV R3, R8; r3=0x0
MOV R4, 0; r4=0x0
BSR 0xB9AEB4 ;[h]; lp=0x67cd0 -> 0xd4005900; pc=0xc02b80 -> 0xffffffff00; CALL: 0x80018df, 0x0, 0x0, 0x0
MOV R8, R0; r8=0x0
MOV R1, R6; r1=0x0
BSR 0xD4AA ;[i]; lp=0x67cd8 -> 0x51cea500; pc=0x7517e -> 0xfa00600; CALL: 0x80018df, 0x0, 0x0, 0x0
BGEI R8, 0x0, 0x32 ;[j]; unlikely
```

```
0x67c8a
MOVU R1, 0xCE5172; r1=0xce5172 -> 0xffffffff00
MOVU R2, 0xCE5161; r2=0xce5161 -> 0xffffffff00
MOV R3, 363; r3=0x16b "J"
BSR 0xFAADF8 ;[k]; lp=0x67c9e -> 0x79df5100; pc=0x1012a92 -> 0xffffffff00; CALL: 0x80018df, 0x0, 0x0, 0x0
MOV R2, R8; r2=0x0
MOVU R1, 0xCE5187; r1=0xce5187 -> 0xffffffff00
BSR 0xFAADEE ;[k]; lp=0x67ca8 -> 0x51ce9e00; pc=0x1012a92 -> 0xffffffff00; CALL: 0x80018df, 0x0, 0x0, 0x0
```

```
0x67d06
MOV R0, 0; r0=0x0
```

```
0x67ca4
MOV R0, -1; r0=0xffffffff -> 0xffffffff00
BRA 0x62 ;[c]; pc=0x67d0a -> 0xf470b00
```

```
0x67cd8
MOVU R1, 0xCE51A5; r1=0xce51a5 -> 0xffffffff00
MOVU R2, 0xCE5161; r2=0xce5161 -> 0xffffffff00
MOV R3, 370; r3=0x172 -> 0xc004a00
BSR 0xFAADAA ;[k]; lp=0x67cec -> 0x96d40b00; pc=0x1012a92 -> 0xffffffff00; CALL: 0x
MOV R2, -1; r2=0xffffffff -> 0xffffffff00
BLTI R7, 0x0, 0x16 ;[l]; unlikely
```


Questions? Comments? Issues? Beers?

<https://github.com/guedou/r2m2>