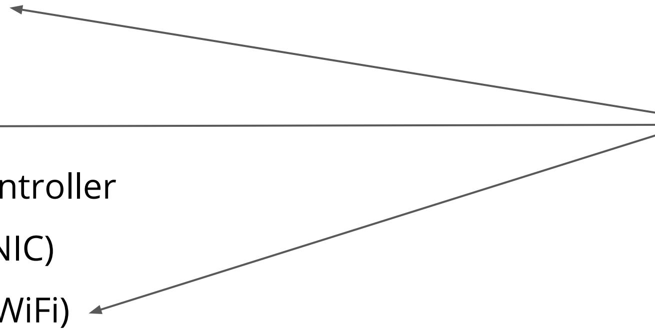
Analyzing ARCompact firmware with Ghidra

Nicolas IOOSS

SSTIC 2021

Combien de processeurs y a-t-il dans un ordinateur portable Lenovo Thinkpad ?

- Central Processing Unit (CPU)
- Graphics Processing Unit (GPU), Graphics Microcontroller (GuC), HEVC/H.265 microcontroller (HuC), display microcontroller (DMC)
- Intel Management Engine (IME, CSME)
- Trusted Platform Module (TPM)
- Lenovo Embedded Controller (EC)
- Hard Disk Controller / Flash Memory Controller
- Ethernet Network Interface Controller (NIC)
- Intel Connectivity Integration (CNVi, for WiFi)
- ...



ARCompact

Ledger

Argonaut RISC Core, https://www.synopsys.com/designware-ip/processor-solutions.html

(1995 : Argonaut Technologies Limited, 1997 : ARC International, 2009 : Virage Logic, 2010 : Synopsys)

ARCompact Instruction Set Architecture

ARCv2 ISA

ARCv3 ISA

ARCTangent-A4

ARCtangent-A5

ARC600 family
- ARC 601
- ARC 605
- ARC 610D
- ARC 625D
- ARC 630D

ARC700 family
- ARC 710D
- ARC 725D
- ARC 750D
- ARC 770D

2013
- ARC EM4
- ARC EM6
- ARC HS34
- ARC HS36

2020
- ARC HS5x
- ARC HS6x

t

(EM = Embedded, HS = High Speed)

Outillage pour analyser des firmware ARCompact

Open Source Software for Synopsys's DesignWare ARC Processors (https://github.com/foss-for-synopsys-dwc-arc-processors)

- gcc
- objdump (dans binutils-gdb)
- gdb
- qemu-arc

Hex-Rays IDA-pro:

- désassembleurs pour ARCTangent-A4, ARCompact et ARCv2

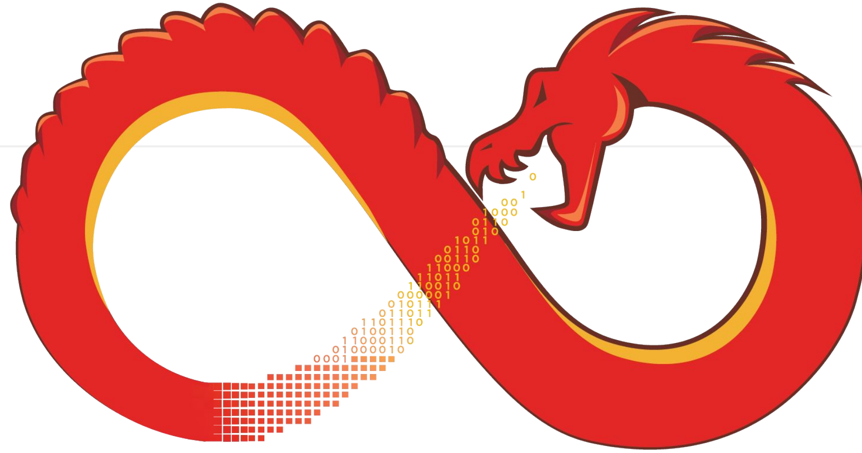
Rizin (Radare2):

- Support de “Argonaut RISC Core”

Pas de décompilateur.

Ghidra

Ledger



- Publié en 2019, https://ghidra-sre.org/
- Désassembleur et décompilateur pour de nombreuses architectures.
- Comment ajouter le support d'ARCompact ?

Plan

Ledger

1. Ajout d’un jeu d’instructions
2. Multiplication
3. Instruction LP

1. Ajout d’un jeu d’instructions

Ledger

Comment un jeu d’instruction est défini ?

- Dossier Ghidra/Processors
  (https://github.com/NationalSecurityAgency/ghidra/tree/Ghidra_9.2.4_build/Ghidra/Processors)
- Fichiers .ldefs, .cspec et .pspec : caractéristiques du processeur (32 bits, Little Endian, conventions d’appel, etc.)
- Fichier .slaspec : définition des instructions, en SLEIGH (langage spécifique de Ghidra)

```
:mov_s s_rh, s_rb is s_major_opcode=0x0e & s_sub_opcode_b3b4=3 & s_rb & s_rh  { s_rh = s_rb;
}
```

Documentation officielle : https://ghidra.re/courses/languages/

1. Ajout d'un jeu d'instructions

Ledger

```
:mov_s s_rh, s_rb is s_major_opcode=0x0e & s_sub_opcode_b3b4=3 & s_rb & s_rh { s_rh = s_rb; }
```

Assembleur

Décodage des instructions

p-code (décompilateur)

Une telle ligne utilise des *tokens* (extraction des bits)

et des registres

```
define space register type=register_space size=2;
define register offset=0x00 size=4 [
    r0 r1 r2 r3 …
];
```

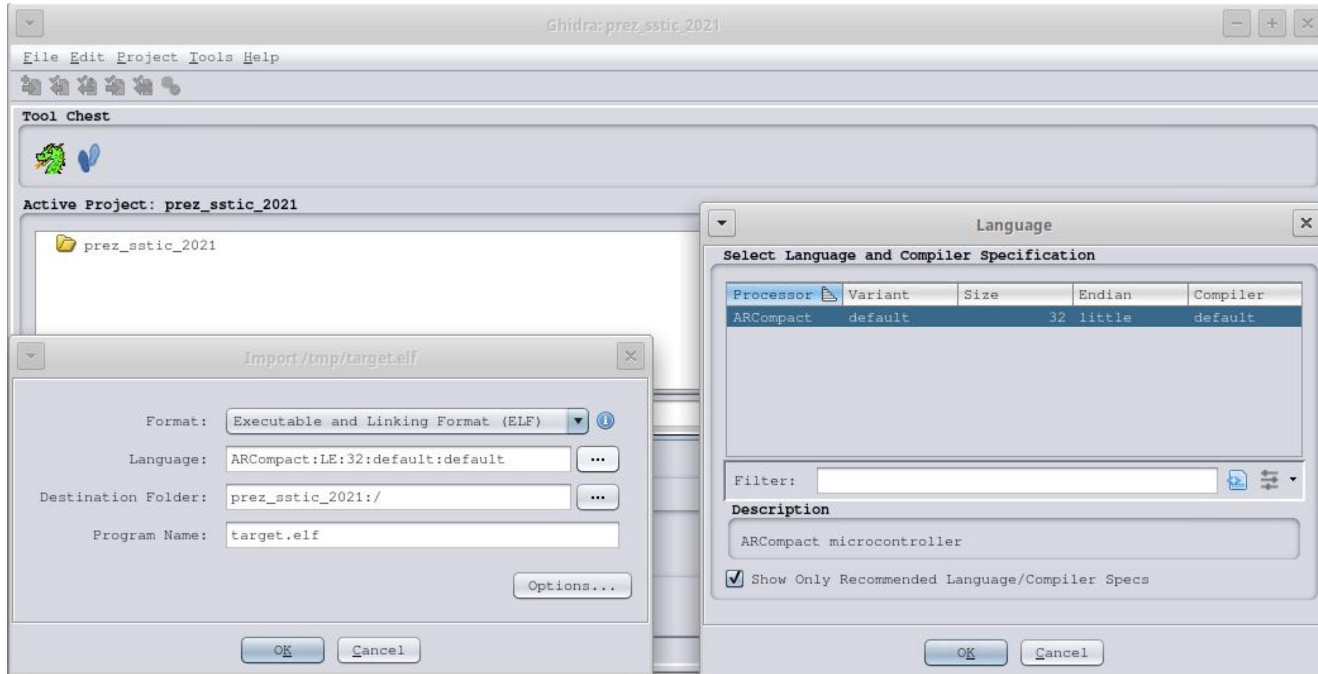
```
define token instr16 (16)
    s_major_opcode = (11, 15)
    s_rb = (8, 10)
    s_rh_low = (5, 7)
    s_sub_opcode_b3b4 = (3, 4)
    s_rh_high = (0, 2)
;
```

1. Ajout d’un jeu d’instructions

Ledger

Compilation du fichier .slaspec :

```
support/sleigh -xulntecf -a Ghidra/Processors/ARCompact/data/languages
```



(Avec un conteneur : `podman run --rm -v “$(pwd):/ghidra” -ti docker.io/library/openjdk:16-slim /ghidra/support/sleigh …`)

Plan

Ledger

1. Ajout d’un jeu d’instructions
2. Multiplication
3. Instruction LP

2. Multiplication

Ledger

Que fait cette fonction ?

```
                     FUN_ram_c0085164
ram:c0085164 08 74          mov_s      r12,r0          : r12 = r0
ram:c0085166 e0 78          nop_s
ram:c0085168 1d 22 41 00    mpyu       r1,r2,r1        : ???
ram:c008516c 1d 22 00 03    mpyu       r0,r2,r12
ram:c0085170 1c 22 0b 03    mpyhu      r11,r2,r12
ram:c0085174 1d 23 0c 03    mpyu       r12,r3,r12
ram:c0085178 61 71          add_s      r1,r1,r11       : r1 = r1 + r11
ram:c008517a 99 61          add_s      r1,r1,r12       : r1 = r1 + r12
ram:c008517c e0 7e          j_s        blink      : return
```

2. Multiplication

Ledger

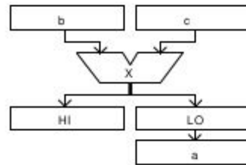
```
ram:c0085168 1d 22 41 00     mpyu  r1,r2,r1   : r1 = low 32 bits of r2*r1
ram:c008516c 1d 22 00 03     mpyu  r0,r2,r12 : r0 = low 32 bits of r2*r12
ram:c0085170 1c 22 0b 03     mpyhu r11,r2,r12: r11 = high 32 bits of r2*r12
```

MPYU

32 x 32 Unsigned Multiply Low

Extension Option

Operation:
dest ← (src1 X src2).low



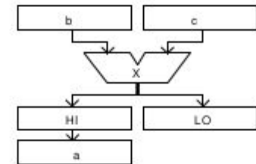
Format:
inst dest, src1, src2

MPYHU

32 x 32 Unsigned Multiply High

Extension Option

Operation:
dest ← (src1 X src2).high



Format:
inst dest, src1, src2

2. Multiplication

Ledger

ARCompact.slaspec :

```
:mpyhu^op4_dotcond op4_a, op4_b_src, op4_c_src is
  (l_major_opcode=0x04 & l_sub_opcode6=0x1c & l_flag=0 &
  op4_dotcond & op4_a) ... & op4_b_src & op4_c_src
{
  # extend source values to 64 bits
  local val_b:8 = zext(op4_b_src);
  local val_c:8 = zext(op4_c_src);
  # compute the product
  local result:8 = val_b * val_c;
  # extract high 32 bits
  op4_a = result(4);
}
```

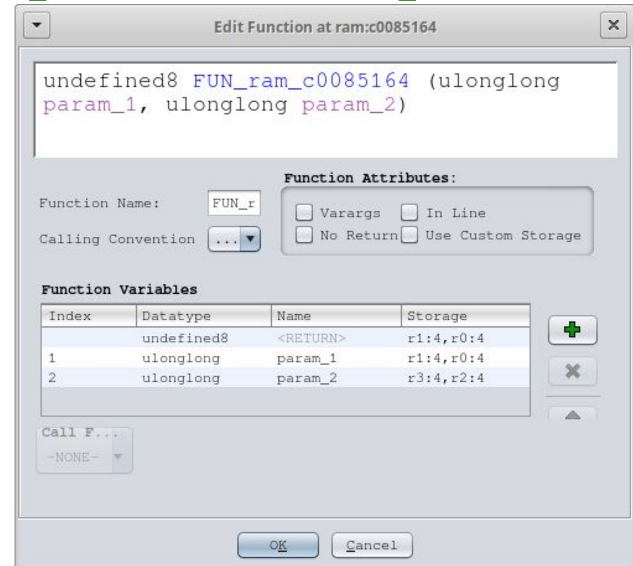
2. Multiplication

Résultat de la décompilation :

```
undefined8 FUN_ram_c0085164(uint param_1,int param_2,uint param_3,int param_4)
{
  return CONCAT44(param_3 * param_2 + (int)((ulonglong)param_3 * (ulonglong)param_1 >> 0x20)
+
                  param_4 * param_1,param_3 * param_1);
}
```

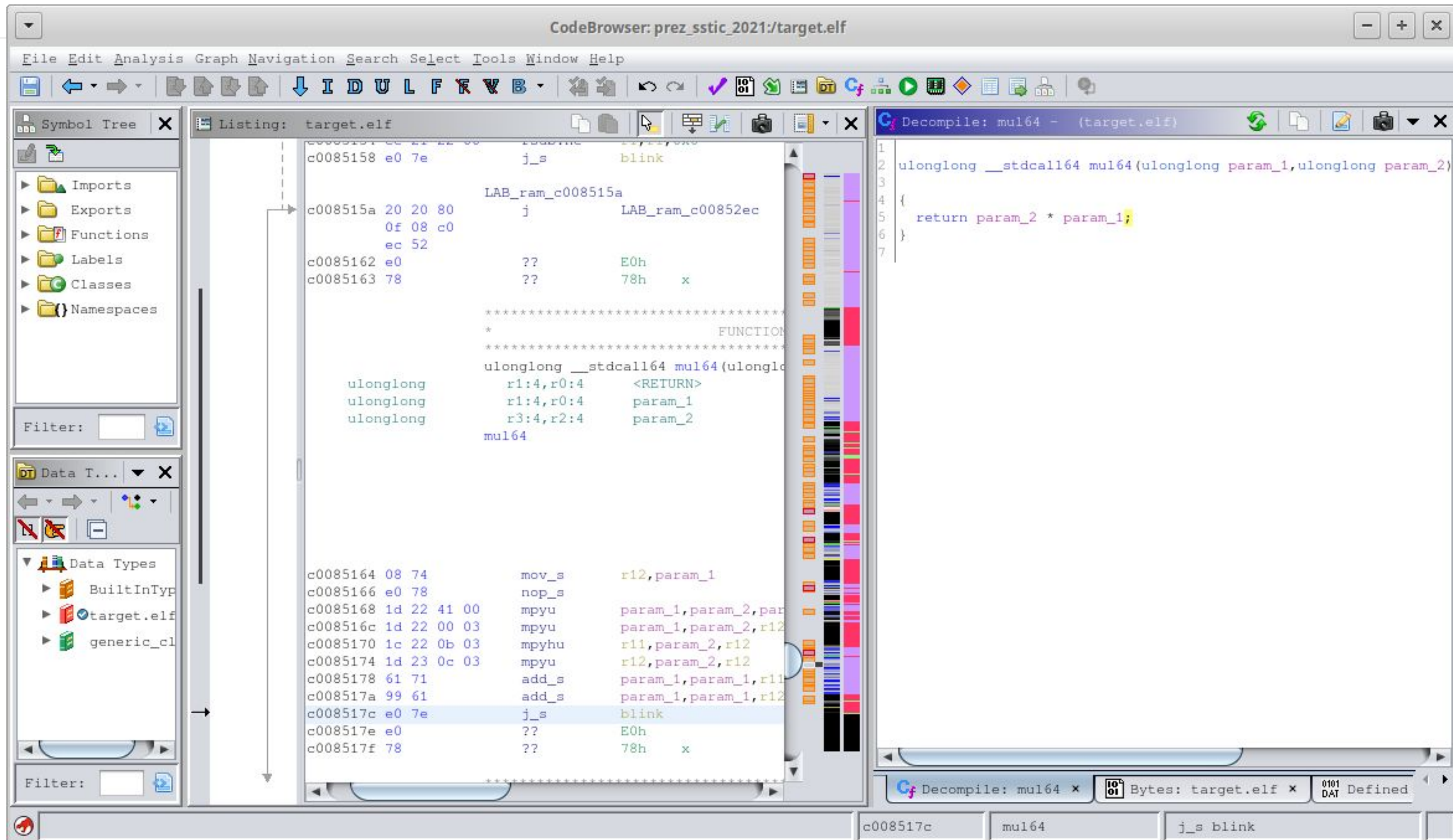
Après un retypage, tout devient plus clair :

```
longlong __stdcall64 FUN_ram_c0085164(ulonglong param_1,
     ulonglong param_2)
{
  return param_2 * param_1;
}
```



2. Multiplication

Ledger



Plan

Ledger

1. Ajout d’un jeu d’instructions
2. Multiplication
3. Instruction LP

3. Instruction LP

Ledger

Que fait l’instruction LP ?

```
c0085230 0a 24 80 70     mov        lp_count,r2    : lp_count = r2;
c0085234 42 21 41 00     sub        r1,r1,0x1      : r1 = r1 - 1;
c0085238 42 20 43 00     sub        r3,r0,0x1      : r3 = r0 - 1;
c008523c a8 20 80 01     lp         LAB_c0085248   : ???

c0085240 01 11 84 02     ldb.a      r4,[r1,0x1]    : r4 = *(++r1);
c0085244 01 1b 0a 01     stb.a      r4,[r3,0x1]    : *(++r3) = r4;
                       LAB_c0085248
c0085248 20 20 c0 07     j          blink          : return;
```

3. Instruction LP

Ledger

Que fait l’instruction LP ?

```
c0085230 0a 24 80 70      mov      lp_count,r2     : lp_count = r2;
c0085234 42 21 41 00      sub      r1,r1,0x1       : r1 = r1 - 1;
c0085238 42 20 43 00      sub      r3,r0,0x1       : r3 = r0 - 1;
c008523c a8 20 80 01      lp       LAB_c0085248    : ???

c0085240 01
c0085244 01

c0085248 20
```

LPcc

Loop Set Up

Branch Operation

Operation:

if (cc=false) then cPC ← (cPCL+rd) else (LP_END ← cPCL+rd) & (LP_START ← nPC)

Format:

inst rel_addr

3. Instruction LP

Ledger

Que fait l’instruction LP ? Comme do{}while ! (ou le préfixe REP en x86)

```
c0085230 0a 24 80 70      mov          lp_count,r2    : lp_count = r2;
c0085234 42 21 41 00      sub          r1,r1,0x1      : r1 = r1 - 1;
c0085238 42 20 43 00      sub          r3,r0,0x1      : r3 = r0 - 1;
c008523c a8 20 80 01      lp           LAB_c0085248   : do {

c0085240 01 11 84 02      ldb.a        r4,[r1,0x1]    :   r4 = *(++r1);
c0085244 01 1b 0a 01      stb.a        r4,[r3,0x1]    :   *(++r3) = r4;
                       LAB_c0085248                   : } while (--lp_count != 0);
c0085248 20 20 c0 07      j            blink          : return;
```

3. Instruction LP

Ledger

Comment implémenter la sémantique de LP en SLEIGH ?

En lisant https://ghidra.re/courses/languages/html/sleigh_context.html !

> *However, for certain processors or software, the need to distinguish between different interpretations of the same instruction encoding, based on context, may be a crucial part of the disassembly and analysis process. There are two typical situations where this becomes necessary.*
>
> - *[...]*
> - *The processor supports instructions that temporarily affect the execution of the immediately following instruction(s). For example,* ***many processors support hardware loop instructions*** *that automatically cause the following instructions to repeat without an explicit instruction causing the branching and loop counting.*

3. Instruction LP

Ledger

```
define register offset=0x140 size=4 [ lp_start ];
define register offset=0x148 size=1 [ is_in_loop ];
define register offset=0x200 size=4 [ contextreg ];

define context contextreg
  phase = (0,0)
  loopEnd = (1,1) noflow
;

op4_lp_loop_end: target is
  l_op_format=2 & l_u6 & l_s12_high
  [ target = (inst_start & ~3) +
      (l_s12_high << 7) + (l_u6 << 1);
  ] { export *[ram]:4 target;  }
```

```
:lp op4_lp_loop_end  is
  l_major_opcode=0x04 & l_sub_opcode6=0x28 & l_flag=0
&
  l_op_format=2 & op4_lp_loop_end
  [ loopEnd = 1; globalset(op4_lp_loop_end, loopEnd); ]
{
  lp_start = inst_next;
  is_in_loop = 1;
}
```

3. Instruction LP

Ledger

```
:^instruction is phase=0 & instruction [ phase = 1; ]  { build instruction; }
:^instruction is phase=0 & loopEnd=1 & instruction [ phase = 1; ]
{
  if (is_in_loop == 0) goto <end_loop>;
  lp_count = lp_count - 1;
  if (lp_count == 0) goto <end_loop>;
  pc = lp_start;
  goto [pc];
<end_loop>
  is_in_loop = 0;
  build instruction;
}
with: phase = 1 {
      # ... all instructions are decoded here
}
```

Fin du « do {} while (--lp_count != 0); »

Exécution normale de l’instruction ciblée par LP

3. Instruction LP

Ledger

Résultat de la décompilation :

```
puVar3 = (undefined *)((int)src + -1);
puVar4 = (undefined *)((int)dst + -1);
do {
  puVar3 = puVar3 + 1;
  puVar4 = puVar4 + 1;
  *puVar4 = *puVar3;
  size = size - 1;
} while (size != 0);
```

On voit bien le « do { … } while (…); » et il s’agit bien de la fonction `memcpy`

3. Instruction LP

Ledger

CodeBrowser: prez_sstic_2021:/target.elf

File Edit Analysis Graph Navigation Search Select Tools Window Help

Symbol Tree: Imports, Exports, Functions, Labels, Classes, Namespaces

Data Types: BuiltInTyp, target.elf, generic_cl

Listing: target.elf

```
c00851fc 04 11 05 04    ld.ab     r5,[src,0x4]
c0085200 04 1b 10 01    st.ab     r4,[r3,0x4]
c0085204 04 1b 50 01    st.ab     r5,[r3,0x4]

                LAB_ram_c0085208
c0085208 0d 0a 9e 00    bbit0     size,0x2,LAB_ram_c0
c008520c 04 11 04 04    ld.ab     r4,[src,0x4]
c0085210 04 1b 10 01    st.ab     r4,[r3,0x4]

                LAB_ram_c0085214
c0085214 0d 0a 5e 00    bbit0     size,0x1,LAB_ram_c0
c0085218 02 11 04 05    ldw.ab    r4,[src,0x2]
c008521c 02 1b 14 01    stw.ab    r4,[r3,0x2]

                LAB_ram_c0085220
c0085220 0d 0a 1e 00    bbit0     size,0x0,LAB_ram_c0
c0085224 01 11 84 04    ldb.ab    r4,[src,0x1]
c0085228 01 1b 12 01    stb.ab    r4,[r3,0x1]

                LAB_ram_c008522c
c008522c 20 20 c0 07    j         blink

                LAB_ram_c0085230
c0085230 0a 24 80 70    mov       lp_count,size
c0085234 42 21 41 00    sub       src,src,0x1
c0085238 42 20 43 00    sub       r3,dst,0x1
c008523c a8 20 80 01    lp        LAB_ram_c0085248

                LAB_ram_c0085240
c0085240 01 11 84 02    ldb.a     r4,[src,0x1]
c0085244 01 1b 0a 01    stb.a     r4,[r3,0x1]

                LAB_ram_c0085248
c0085248 20 20 c0 07    j         blink=>LP_START
```

Decompile: memcpy - (target.elf)

```
4  void * memcpy(void *dst,void *src,size_t size)
5
6  {
7    undefined2 uVar1;
8    bool bVar2;
9    undefined *puVar3;
10   undefined4 *puVar4;
11   undefined *puVar5;
12   undefined4 uVar6;
13   undefined4 uVar7;
14   undefined4 uVar8;
15   undefined4 uVar9;
16   uint uVar10;
17
18   if (size != 0) {
19     if ((((uint)dst | (uint)src) & 3) != 0) {
20       puVar3 = (undefined *)((int)src + -1);
21       puVar5 = (undefined *)((int)dst + -1);
22       do {
23         puVar3 = puVar3 + 1;
24         puVar5 = puVar5 + 1;
25         *puVar5 = *puVar3;
26         size = size - 1;
27       } while (size != 0);
28       LP_START = 0xc0085240;
29       LP_END = 0xc0085248;
30       return dst;
31     }
32     uVar10 = size >> 4;
33     bVar2 = false;
34     puVar4 = (undefined4 *)dst;
35     if (uVar10 == 0) goto LAB_ram_c00851ec;
36     LP_START = 0xc00851cc;
37     LP_END = 0xc00851ec;
38     bVar2 = true;
```

Decompile: memcpy × | Bytes: target.elf × | Defined

c008523c | memcpy | lp 0xc0085248

Conclusion

Ledger

Ghidra peut maintenant décompiler des firmware ARCompact !

Code : https://github.com/niooss-ledger/ghidra/tree/arcompact/Ghidra/Processors/ARCompact

Pull Request : https://github.com/NationalSecurityAgency/ghidra/pull/3006

Questions ?