Analyzing ARCompact Firmware with Ghidra

Nicolas Iooss
nicolas.iooss@ledger.fr


Ledger Donjon

Abstract. Some microcontroller units use the ARCompact instruction set. When analyzing their firmware, several tools exist to recover the code in assembly instructions. Before this publication, no tool existed which enabled to recover the code in a language which is easier to understand, such as C language.

Ghidra is a powerful reverse-engineering project for which it is possible to add support for many instruction sets. This article presents how ARCompact support was added to Ghidra and some challenges which were encountered in this journey. This support enabled using Ghidra's decompiler to recover the semantics of the code of studied firmware in a language close to C.

1 Introduction

A modern computer embeds many microcontroller units (MCU). They are used to implement complex features in the Network Interface Cards (NIC), the hard disks, the flash memory devices, etc. These MCUs run code in a similar way to usual processors: they use some *firmware* which contains instructions for a specific architecture.

For example:

— Some NICs implement complex features using MIPS instruction set [7].
— On some HP servers, the iLO (integrated Lights-Out) is implemented using ARM instruction set [8].
— On some Dell servers, the iDRAC (integrated Dell Remote Access Controller) is implemented using Renesas SuperH instruction set [6].
— Some Hardware Security Modules (HSM) are implemented using PowerPC instruction set [4].
— On some Intel computers, the ME (Management Engine) is implemented using ARCompact instruction set [11].
— On some of Lenovo's Thinkpad computers, the EC (Embedded Controller) is implemented using ARCompact instruction set [3, 5].
— On some computers, the WiFi chipset runs code implemented using ARCompact instruction set (cf. page 5 of [5]).

Many of the used instruction sets have been implemented in reverse-engineering tools such as Binary Ninja, Ghidra, IDA, metasm, miasm, objdump and radare2. However these tools usually only implement a *disassembler* for instructions sets which are not widely used. The static analysis of firmware is much easier when the code can be actually *decompiled*, for example in C language or in a pseudo-code which is easier to read than raw assembly instructions.

Ghidra (`https://ghidra-sre.org/`) is a powerful tool which enables implementing a decompiler for any instruction set quite easily. This relies on a domain-specific language called SLEIGH [1].

ARCompact is the name of an instruction set used by some ARC processors (Argonaut RISC Core). It is still widely used in several MCUs embedded in computers. This is why implementing support for this instruction set in reverse-engineering tools can be very useful.

This article presents how the support for ARCompact was added to Ghidra in order to analyze the firmware of an MCU studied by Ledger Donjon. This support enabled using Ghidra's disassembler and decompiler in the analysis. It was submitted as a Pull Request in May 2021, `https://github.com/NationalSecurityAgency/ghidra/pull/3006`. This article highlights the main challenges which were encountered and how they were solved.

2 ARCompact instruction set discovered through Ghidra

When studying an instruction set, some characteristics need to be determined. Is the length of instructions fixed? How many core registers are available? Are there several address spaces for code and data? How are functions called?

For ARCompact, the Programmer's Reference [2] provides answers to all these questions. ARCompact is an instruction set which operates on 32-bit values using variable-length instructions. There are sixty-four 32-bit core registers. Some instructions can be conditionally executed, with a condition which depends on four condition flags (`Z`, `N`, `C` and `V`) like ARM instruction set.[1] When calling functions, the instruction *Branch and Link* (`bl`) puts the return address in the register named `blink`, like ARM's link register.

These characteristics enabled to bootstrap ARCompact support in Ghidra. For this, several files were created in a new directory named

1. `Z` is the *Zero* flag, `N` is the *Negative* flag, `C` is the *Carry* flag and `V` is the *Overflow* flag.

`Ghidra/Processors/ARCompact` in Ghidra's directory. These files were inspired from the support of other instruction sets, including previous works about supporting MeP [12–14] and Xtensa instruction sets [9, 10].

The file which described how instructions are decoded, `Ghidra/Processors/ARCompact/data/languages/ARCompact.slaspec` was initialized with the definition of some registers (listing 1).

```
define register offset=0x00 size=4 [
  r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15
  r16 r17 r18 r19 r20 r21 r22 r23 r24 r25 gp fp sp ilink1 ilink2
      blink
  r32 r33 r34 r35 r36 r37 r38 r39 r40 r41 r42 r43 r44 r45 r46 r47
  r48 r49 r50 r51 r52 r53 r54 r55 r56 mlo mmid mhi lp_count
      r61reserved r62limm pcl
];
define register offset=0x130 size=1 [ Z N C V ];
```

Listing 1. SLEIGH specification of ARCompact core registers

Implementing the decoding of each instruction is then a matter of defining *tokens* to extract bits and defining the associated semantics in pseudo-code. This process was described in length in previous presentations [12] and in Ghidra's documentation [1].

There were several challenges in the implementation of ARCompact instruction set. One of them was that instructions using 32-bits constants encode them in a mix of Little Endian and Big Endian: the value `0xAABBCCDD` is encoded as bytes `BB AA DD CC`. This issue was solved by defining a constructor `limm` (for *long immediate*) using specific tokens (listing 2).

```
define token limm_low_token (16) limm_low = (0, 15);
define token limm_high_token (16) limm_high = (0, 15);
limm: limm is limm_high ; limm_low [ limm = (limm_high << 16) +
    limm_low; ] { export *[const]:4 limm; }
```

Listing 2. SLEIGH specification of the decoding of 32-bit immediate values

Some other challenges are described in the following sections.

3 64-bit multiplication

The analyzed firmware contains the code in listing 3.

```
Address  Bytes        Instruction         Description
c0085164 08 74        mov_s r12,r0      ; move the value in r0 to r12
c0085166 e0 78        nop_s             ; no operation
c0085168 1d 22 41 00 mpyu  r1,r2,r1     ; multiply r2 and r1 into r1
```

```
c008516c 1d 22 00 03 mpyu  r0,r2,r12  ; multiply r2 and r12 into r0
c0085170 1c 22 0b 03 mpyhu r11,r2,r12 ; multiply r2 and r12 and
    store the high 32 bits in r11
c0085174 1d 23 0c 03 mpyu  r12,r3,r12 ; multiply r3 and r12 into r12
c0085178 61 71       add_s r1,r1,r11  ; add r1 and r11 into r1
c008517a 99 61       add_s r1,r1,r12  ; add r1 and r12 into r1
c008517c e0 7e       j_s   blink      ; jump back to the caller
```

Listing 3. Assembly code containing multiplication instructions

`mpyu` and `mpyhu` compute the product of two 32-bit registers as a 64-bit value and store in the destination register either the low 32 bits or the high 32 bits of the result. Using both instruction could mean that the code implements a 64-bit multiplication. When doing some maths, it appears that the code indeed computes the 64-bit product of two 64-bit numbers. With Ghidra, it is possible to accelerate this analysis by implementing the semantics of the instructions.

The SLEIGH specification of these instructions was implemented as shown in listing 4.

```
: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);
}

:mpyu^op4_dotcond op4_a, op4_b_src, op4_c_src is
  (l_major_opcode=0x04 & l_sub_opcode6=0x1d & l_flag=0 &
  op4_dotcond & op4_a) ... & op4_b_src & op4_c_src
{
  local val_b:8 = zext(op4_b_src);
  local val_c:8 = zext(op4_c_src);
  local result:8 = val_b * val_c;
  # extract low 32 bits
  op4_a = result:4;
}
```

Listing 4. SLEIGH specification of instructions `mpyu` and `mpyhu`

This enabled Ghidra to directly understand the function as the implementation of a 64-bit multiplication between values stored in registers `r1:r0` and `r3:r2` (figure 1 and listing 5).

```
uint64_t mul64(uint64_t param_1,uint64_t param_2)
```

```
{
  return param_2 * param_1;
}
```

Listing 5. Decompiled output of the function given in listing 3

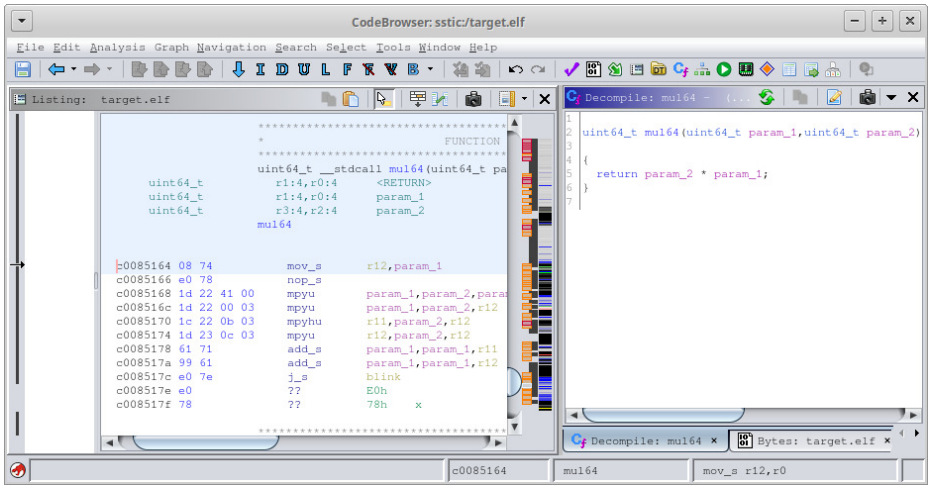

Fig. 1. Implementation of a 64-bit multiplication

This example shows how a decompiler can speed-up the time spent at reverse-engineering a firmware. Instead of trying to understand how `mpyu` and `mpyhu` are combined together, it is possible to rely on the code produced by the decompiler, which is much simpler.

4 Loop instruction

ARCompact instruction set provides an instruction named *Loop Set Up Branch Operation* and written `lp` in assembly code. This instruction could be misleading. To understand it, listing 6 presents a piece of code which uses this instruction in the analyzed firmware.

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

c0085240 01 11 84 02      ldb.a       r4,[r1,0x1]
c0085244 01 1b 0a 01      stb.a       r4,[r3,0x1]
                        LAB_c0085248
```

```
9 | c0085248 20 20 c0 07     j           blink
```

Listing 6. Assembly code containing a loop

Contrary to usual branching instruction, `lp LAB_c0085248` does not mean: branch to address `c0085248` if some condition is met. Instead, it means:

— Execute instructions until address `c0085248` is reached.
— When reaching `c0085248`, decrement register `lp_count`.
— If `lp_count` is not zero, branch back to the instruction right after `lp` (at address `c0085240`).

This makes the code repeat the instructions between `lp` and the address given as parameter (`c0085248`) exactly `lp_count` times. In this example, the instructions copy a byte from the memory referenced by `r1` into the one referenced by `r3`, incrementing the pointers at each iteration.

The problem caused by instruction `lp` is that the semantic of the instruction located at the address given as parameter changes. In order to decompile the example code correctly, the semantic of the loop needs to be added to the instruction at address `c0085248`.

In a real ARCompact MCU, `lp` is implemented by using two auxiliary registers, `lp_start` and `lp_end`:

— `lp LAB_c0085248` puts the address of the next instruction (`c0085240`) into `lp_start` and the given address `c0085248` into `lp_end`.
— When the MCU reaches address `c0085248`, as it matches the content of `lp_end`, it decrements `lp_count` and branches to `lp_start` if it is not zero.

How such a semantic can be implemented in Ghidra? The answer is surprisingly simple, thanks to Ghidra's documentation which already contains an example of such a problem in `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.* [...] *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.*

The SLEIGH processor specification language supports a feature called *context variables*. Here is how the `lp` instruction was implemented with this feature.

First, a context was defined as well as a register storing `lp_start` (listing 7). Another register was defined, `is_in_loop`, which defines whether the `lp` instruction was executed (which is important to implement conditional `lp` instruction).

```
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
;
```

Listing 7. SLEIGH specification of the context used to implement instruction `lp`

Then, the decoding of `lp` sets the `loopEnd` bit of the context to `1` for the address given to `lp` (listing 8). This is done using a built-in function named `globalset`.

```
: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;
}
```

Listing 8. SLEIGH specification of instruction `lp`

Finally, to change the semantic of the instruction which ends the loop, a two-phase instruction decoding was implemented (listing 9).

```
:^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 {
```

```
20
21  # ... all instructions are decoded here
22
23  }
```

Listing 9. SLEIGH specification of a two-phase instruction decoding pipeline

With these changes, the instructions of the example are decompiled as something which seems to be a correct implementation of a `memcpy` function (figure 2 and listing 10).

```
1  puVar3 = (undefined *)((int)src + -1);
2  puVar4 = (undefined *)((int)dst + -1);
3  do {
4    puVar3 = puVar3 + 1;
5    puVar4 = puVar4 + 1;
6    *puVar4 = *puVar3;
7    size = size - 1;
8  } while (size != 0);
9  return;
```

Listing 10. Decompiled output of the instructions given in listing 6

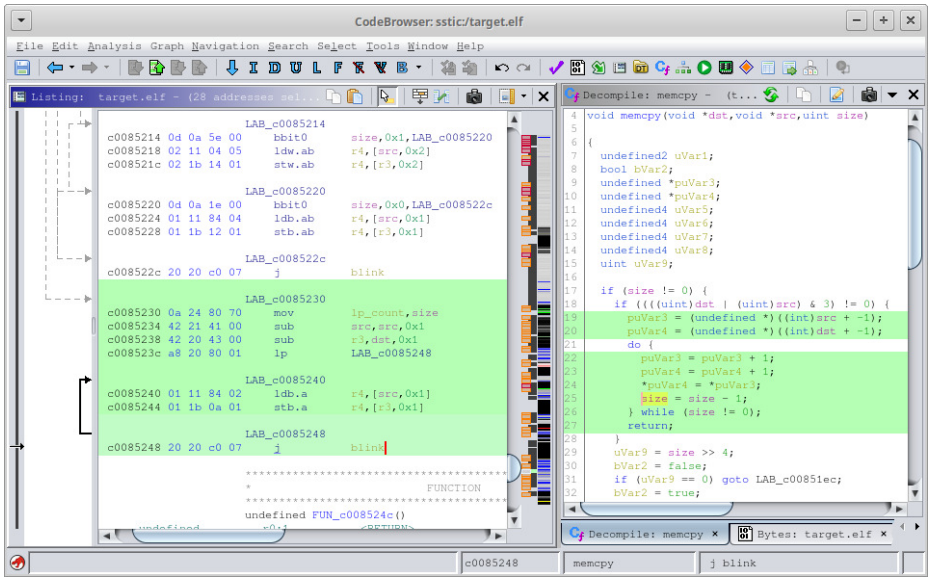

Fig. 2. Implementation of `memcpy` in the studied firmware

5 Conclusion

Thanks to this work, it is possible to perform static analysis on firmware of some MCUs using ARCompact. This work enabled Ledger's security team to bypass the secure boot feature of a MCU. This result will hopefully be presented in the future. This will also help finding issues in code running on MCUs, for example by plugging a fuzzer to Ghidra's machine emulator.

References

1. Ghidra language specification. https://ghidra.re/courses/languages/index.html.
2. Arcompact instruction set architecture, programmer's reference, 2008. http://me.bios.io/images/d/dd/ARCompactISA_ProgrammersReference.pdf.
3. Embedded controllers used in lenovo thinkpad, 2016. https://github.com/hamishcoleman/thinkpad-ec/blob/v1/docs/chips.txt.
4. Jean-Baptiste Bédrune and Gabriel Campana. Everybody be cool, this is a robbery! SSTIC, June 2019. https://www.sstic.org/2019/presentation/hsm/.
5. Alexandre Gazet. Sticky fingers & kbc custom shop. SSTIC, June 2011. https://www.sstic.org/2011/presentation/sticky_fingers_and_kbc_custom_shop/.
6. Nicolas Iooss. idrackar, integrated dell remote access controller's kind approach to the ram. SSTIC, June 2019. https://www.sstic.org/2019/presentation/iDRACKAR/.
7. Yves-Alexis Perez, Loïc Duflot, Olivier Levillain, and Guillaume Valadon. Quelques éléments en matière de sécurité des cartes réseau. SSTIC, June 2010. https://www.sstic.org/2010/presentation/Peut_on_faire_confiance_aux_cartes_reseau/.
8. Fabien Périgaud, Alexandre Gazet, and Joffrey Czarny. Backdooring your server through its bmc: the hpe ilo4 case. SSTIC, June 2018. https://www.sstic.org/2018/presentation/backdooring_your_server_through_its_bmc_the_hpe_ilo4_case/.
9. Sebastian Schmidt. Tensilica xtensa module for ghidra, 2019. https://github.com/yath/ghidra-xtensa.
10. Sebastian Schmidt. Ghidra pull request #1407: Add tensilica xtensa processor support, 2020. https://github.com/NationalSecurityAgency/ghidra/pull/1407.
11. Igor Skochinsky. Intel me secrets, hidden code in your chipset and how to discover what exactly it does. Recon, June 2014. https://recon.cx/2014/slides/Recon%202014%20Skochinsky.pdf.
12. Guillaume Valadon. Implementing a new cpu architecture for ghidra. BeeRump, 2019. https://guedou.github.io/talks/2019_BeeRump/slides.pdf.
13. Guillaume Valadon. Toshiba mep-c4 for ghidra, 2019. https://github.com/guedou/ghidra-processor-mep.
14. xyzz. Toshiba mep processor module for ghidra, 2019. https://github.com/xyzz/ghidra-mep.