AN OVERCOMPLICATED WAY TO DISASSEMBLE BYTES

ARMV7 DISASSEMBLING

HAIL CORPORATE!

ABOUT US

- Agustín Gianni
  - Security researcher at Avast Software
  - agustin.gianni@gmail.com
  - https://twitter.com/agustingianni
- Pablo Sole
  - VP of research at Avast Software
  - pablo.sole@gmail.com
  - https://twitter.com/_pablosole_

WHY?

- We needed information about specific parts of an instruction.
- The ARM platform is widely used and we needed to get used to its assembly language.
- Other disassemblers are inaccurate, incomplete and/or plainly unusable.
  - We are looking at you libopcodes.
- At the time, capstone did not exist.

MAIN IDEA

- Extract all the decoding specifications from the architecture manual.

- Parse the specifications (pseudocode) and obtain an AST.

- Transcode the pseudocode into compilable C++ code.

- Build a disassembler using the automatically generated decoder engine.

THE SPECS

- ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition.
- The format is pretty consistent:
  - Name of the instruction.
  - Description.
  - Different encodings.
  - Versions.
  - Binary representation.
  - Decoding pseudocode.

EXAMPLE SPEC

ADC (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
ADC{S}<c> <Rd>, <Rn>, #<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	1	0	S	Rn				0	imm3			Rd				imm8							

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADC{S}<c> <Rd>, <Rn>, #<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	1	0	1	S	Rn				Rd				imm12												

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

EXTRACTION PROCEDURE

- The pdf can be parsed somewhat successfully.
  - Any pdf library will do it, heck, even pdf2text helps.
- Some of the entries had a different format.
  - Moderate manual intervention was needed.
- You don’t need to do it again because we are sharing it!
- It took us one work day to get the extraction right.

DECODING SPECIFICATION

- The decoding specification is a dictionary containing the following keys:
  - Name.
  - Encoding.
  - Version.
  - String representation.
  - Pattern.
  - Decoding specification.

SPECIFICATION EXAMPLE

```
instructions = [
{
    "name" : "ADC Immediate",
    "encoding" : "T1",
    "version" : "ARMv6T2, ARMv7",
    "format" : "ADC{S}<c> <Rd>, <Rn>, #<imm32>",
    "pattern" : "11110 i#1 01010 S#1 Rn#4 0 imm3#3 Rd#4 imm8#8",
    "decoder" : """d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = ThumbExpandImm(i:imm3:imm8);
    if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;"""
} , {
    "name" : "ADC Immediate",
    "encoding" : "A1",
    "version" : "ARMv4All, ARMv5TAll, ARMv6All, ARMv7",
    "format" : "ADC{S}<c> <Rd>, <Rn>, #<imm32>",
    "pattern" : "cond#4 0010101 S#1 Rn#4 Rd#4 imm12#12",
    "decoder" : """if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
    d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = ARMExpandImm(imm12);"""
} , {
    "name" : "ADC Register",
    "encoding" : "T1",
    "version" : "ARMv4T, ARMv5TAll, ARMv6All, ARMv7",
    "format" : "ADCS <Rdn>, <Rm>:ADC<c> <Rdn>, <Rm>",
    "pattern" : "0100000101 Rm#3 Rdn#3",
    "decoder" : """d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
    (shift_t, shift_n) = (SRType_LSL, 0);"""
```

INSTRUCTION VERSION & ENCODING

- The **version** specifies in which processor version the instruction is available.
  - Mandatory for a good representation of embedded devices.
  - If the version of the processor is wrong then most likely some strings of bytes will be misinterpreted.
- The **encoding** specifies the mode where the instruction can appear.
  - That is ARM or THUMB modes.

INSTRUCTION FORMAT

- String with a very simple template format.
  - Optional value — {name}
  - Mandatory value — <name>
- Used to automatically generate the ‘toString’ method of an instruction.
- Saves us a lot of monkey coding.
- Example: **ADC{S}<c> <Rd>, <Rn>, #<imm32>**
  - **{S}**: Only present if the instruction modifies the CPU flags.
  - **<c>**: Condition.
  - **<Rd>, <Rn>**: Registers.
  - **<imm32>**: Immediate value of size 32 (bits).

INSTRUCTION FORMAT

- From the instruction format specification we can actually generate the correct string representation of an instruction.
- For each instruction in each of its mode we generate a ‘toString’ method that will give us the ‘pretty printing’ of the decoded instruction.

GENERATED CODE

```
std::string decode_adc_immediate_t1_to_string(const ARMInstruction *ins) {
    char op_name[128], op_args[128];
    // DEBUG: ADC{S}<c> <Rd>, <Rn>, #<imm32>
    int ret = snprintf(op_name, sizeof(op_name),
            "ADC%s%s",
            S_str(ins).c_str(),
            c_str(ins).c_str()
    );

    assert(ret >= 0);
    ret = snprintf(op_args, sizeof(op_args),
            " %s, %s, #%s",
            regular_reg_str(ins->d).c_str(),
            regular_reg_str(ins->n).c_str(),
            integer_to_string(ins->imm32).c_str()
    );

    assert(ret >= 0);
    return std::string(op_name) + std::string(op_args);
}
```

INSTRUCTION PATTERN

- ARMv7 instructions are either 16 bits or 32 bits.
- Each instruction is comprised of a set of fixed bits and a set of fields that represents the variable parts of the instruction.
    - Registers, constants, modes, etc.
- Each variable has a name and a length.
    - The name can specify registers, flags, or immediate values.
    - The length of the variable specifies the number of bits in said variable.
- All the information we need to identify an instruction is specified here.
    - With this information we build our decoding table.

INSTRUCTION PATTERN EXAMPLE

- 11110 i#1 01010 S#1 Rn#4 0 imm3#3 Rd#4 imm8#8
  - out_consts = {11110, 01010, 0}
  - out_consts_sizes = {5, 5, 1}
  - out_vars = {i, S, Rn, imm3, Rd, imm8}
  - out_vars_sizes = {1, 1, 4, 3, 4, 8}

DECODING TABLE

- From the instruction bit pattern we can obtain a pair (value, mask) that when applied to a set of bytes will only match the correct instruction.

- if (ins & mask) == value then decode_ins_xxx(...)

DECODING TABLE

```
// Format: (mask, value, version, encoding, decoder_function, name)
ARMOpcode arm_opcodes[] = {
    { 0x0fe00000, 0x02a00000, ... , &ARMDecoder::adc_immediate_a1, "ADC Immediate"},
    { 0x0fe00010, 0x00a00000, ... , &ARMDecoder::adc_register_a1, "ADC Register"},
    { 0x0fe00090, 0x00a00010, ... , &ARMDecoder::adc_register_shifted_register_a1, "ADC (register-shifted register)"},
    { 0x0fe00000, 0x02800000, ... , &ARMDecoder::add_immediate_arm_a1, "ADD (immediate, ARM)"},
    { 0x0fe00010, 0x00800000, ... , &ARMDecoder::add_register_arm_a1, "ADD (register, ARM)"},
    { 0x0fe00090, 0x00800010, ... , &ARMDecoder::add_register_shifted_register_a1, "ADD (register-shifted register)"},
    { 0x0fef0000, 0x028d0000, ... , &ARMDecoder::add_sp_plus_immediate_a1, "ADD (SP plus immediate)"},
    { 0x0fef0010, 0x008d0000, ... , &ARMDecoder::add_sp_plus_register_arm_a1, "ADD (SP plus register, ARM)"},
    { 0x0fff0000, 0x028f0000, ... , &ARMDecoder::adr_a1, "ADR"},
    { 0x0fff0000, 0x024f0000, ... , &ARMDecoder::adr_a2, "ADR"},
    { 0x0fe00000, 0x02000000, ... , &ARMDecoder::and_immediate_a1, "AND (immediate)"},
    { 0x0fe00010, 0x00000000, ... , &ARMDecoder::and_register_a1, "AND (register)"},
    { 0x0fe00090, 0x00000010, ... , &ARMDecoder::and_register_shifted_register_a1, "AND (register-shifted register)"},
    { 0x0fef0070, 0x01a00040, ... , &ARMDecoder::asr_immediate_a1, "ASR (immediate)"},
    { 0x0fef00f0, 0x01a00050, ... , &ARMDecoder::asr_register_a1, "ASR (register)"},
```

DECODING SPECIFICATION

- The manual specifies the only way an instruction can be decoded.
- The way it does it is by writing the decoding algorithm in a fairly simple pseudocode.

DECODING SPECIFICATION

- Comprised of a set of simple statements.
- Colons are the concatenation of bits.
- Other cool operators like ‘IN’, etc.
- Corner case statements like ‘UNPREDICTABLE’, etc.
- Loosely described in the architecture manual.

```
d = UInt(Rd);
n = UInt(Rn);
setflags = (S == '1');
imm32 = ThumbExpandImm(i:imm3:imm8);

if d IN {13,15} || n IN {13,15} then
    UNPREDICTABLE;
```

DECODING SPECIFICATION: GENERATED CODE

```
ARMInstruction *ARMDecoder::decode_adc_immediate_t1(uint32_t opcode, ARMInstrSize ins_size, ARMEncoding e) {
    int     i = get_bit(opcode, 26);
    int     S = get_bit(opcode, 20);
    int    Rn = get_bits(opcode, 19, 16);
    int  imm3 = get_bits(opcode, 14, 12);
    int    Rd = get_bits(opcode, 11,  8);
    int  imm8 = get_bits(opcode,  7,  0);

    int setflags = (S == 1);
    int imm32    = ThumbExpandImm(Concatenate(Concatenate(i, imm3, 3), imm8, 8));
    int d        = UInt(Rd);
    int n        = UInt(Rn);

    if (unlikely(((d == 13 || d == 15) || (n == 13 || n == 15)))) {
        return new UnpredictableInstruction();
    }

    ARMInstruction *ins = ARMInstruction::create();
    ins->opcode = opcode;
    ins->ins_size = ins_size;
    ins->id = adc_immediate;
    ins->m_to_string = decode_adc_immediate_t1_to_string;
    ins->m_decoded_by = "ARMDecoder::decode_adc_immediate_t1";
    ins->setflags = setflags;
    ins->imm32 = imm32;
    ins->d = d;
    ins->n = n;
    ins->encoding = e;
    ins->imm3 = imm3;

    return ins;
}
```

TESTING

- Decoding pattern
  - Count the number of bits decoded: it should be 16 or 32.
- Decoding pseudocode
  - The parser did a good job identifying mistakes in the pseudocode.
  - The mistakes were fixed on the specification.
  - In the end we need to trust the specification.
- Transcoded code
  - The compiler warns you about invalid translations.
  - We need to manually verify each of the translation routines.

TESTING

- In order to test the accuracy of the disassembler we need a way to compare the results with a “trusted” source.
- In order to be more accurate we’ve chosen to compare our results to more than one source.
  - Capstone
  - DARM
  - libopcodes

TESTING HARNESS

- A C++ program traverses the decoding table.
- For each pair (value, mask) it generates N “random” instructions that fit that pair.
- Disassemble each of the N instructions using M different disassemblers.
- Fix the differences in instruction representation.
  - Normalize aliases for registers: r13=SP — r14=LR — r15=PC
  - Normalize instruction names: PUSH=STMDB
- Compare the results of each of the M disassemblers.

EXAMPLE OUTPUT


```
{
  "opcode"  : 3802909214,
  "decoder" : "ARMDecoder::decode_adc_immediate_a1",
  "reto"    : "ADC r12, r11, #0x1e000",
  "caps"    : "adc ip, fp, #0x1e000",
  "darm"    : "ADC r12, r11, #0x1e000"
},
{
  "opcode"  : 1118082781,
  "decoder" : "ARMDecoder::decode_adc_immediate_a1",
  "reto"    : "ADCmi r9, r4, #0xdd00000",
  "caps"    : "adcmi sb, r4, #0xdd00000",
  "darm"    : "ADCMI r9, r4, #0xdd00000"
},
```

FUTURE IMPROVEMENTS

- Code generation needs to be improved.
- Represent instructions in a better, more compact, and easy to use way.
- Finish the emulator side.

BIBLIOGRAPHY

- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html
- https://github.com/jbremer/darm
- https://github.com/aquynh/capstone

THANKS

- Ekoparty