SmartDec

PUSHING NATIVE CODE DECOMPILATION TO THE NEXT LEVEL

Reverse engineering of binary programs for custom virtual machines

Alexander Chernov, PhD

Katerina Troshina, PhD

Moscow, Russsia

SmartDec.net

About us

- SmartDec decompiler
- Academic background
- Defended Ph.D.

- Industry

Once upon a time…

- In a country far far away…
- Reverse engineering of binary code
  - No datasheet
  - Very poor documentation

High-level representation for engineers

Once upon a time…

High-level representation for engineers

- Call graph
- Flow-chart
- C code

Approaches to the analysis

- First thought: to search the code and data for clues about processor type
- To try different disassemblers and check if the disassembled listing looks like a program
  - IDA Pro supports above 100 “popular” processors (no luck)
  - Download and try disassemblers for other processors (no luck)

Approaches to the analysis

- Fallback: to try to go as far as possible in binary analysis collecting information about processor
- Initial information:
  - Rough separation of code and data (based on pieces of documentation and “look” of data)
  - Enough code for statistics to be meaningful (some 30 KiB)

Search space

- Architecture search space:
  - Word byte order: LE, BE
  - Instruction encoding: byte stream/WORD stream/DWORD stream…
  - Code/data address space: uniform (von Neumann), separate (Harvard)
  - Code addressing: byte, WORD, …
  - Register based, stack based

[WORD – 2 bytes, DWORD – 4 bytes]

RET instruction

- Possible instruction encoding: fixed-width WORD
- Expected “Return from subroutine” (RET) instruction properties:
  - RET instruction has a single fixed opcode and no arguments
  - RET instruction opcode is statistically among the most frequent 16-bit values in code

WORD values frequencies

- 20 most frequent 16-bit values:

0b01	854	5.1
0800	473	2.8
8c0d	432	2.6
2b00	401	2.4
4e1c	365	2.2
0801	277	1.6
890f	261	1.6
8f09	217	1.3
0f00	196	1.2
0b00	195	1.1
0b8f	162	0.97
4a01	163	0.97
0b36	155	0.92
0802	149	0.88
3ddc	145	0.86
0b80	132	0.79
990f	131	0.78
8afa	125	0.75
1aff	119	0.72
0900	117	0.7

- Reference: most frequent instructions of Java bytecode (in the standard libs of jre6)

ALOAD	878565	18
DUP	382278	7.9
INVOKEVIRTUAL	328763	6.8
LDC	231162	4.8
GETFIELD	230158	4.8
ILOAD	214427	4.4
SIPUSH	207427	4.3
AASTORE	168088	3.5
INVOKESPECIAL	140387	2.9
ICONST_0	132669	2.7
ASTORE	128835	2.7
BIPUSH	126262	2.6
ICONST_1	107881	2.2
SASTORE	96677	2.0
PUTFIELD	87405	1.8
NEW	84285	1.7
GOTO	80815	1.7
RETURN	70388	1.5
ISTORE	70064	1.4
ARETURN	68054	1.4
All returns:	176860	3.7

Possible code structure

```
sub1: …
        CALL subn            # absolute address
        …
        RET
sub2: …                      # follows RET
        RET
subn: …                      # follows RET
        RET
```

CALL heuristics

- Assumption 1: there exists a CALL instruction taking the absolute address of a subroutine
- Assumption 2: a considerable number of subroutines start immediately after RET instruction

CALL search

- Search space: for each candidate for RET, try all possible CALL candidates with addresses as 16-bit bitmask in 32-bit words, LE/BE, 16-bit/byte addressing
- Example:
  - 4 bytes: 12 34 56 78
  - Address bitmask: 00 00 ff ff (4057 different)
  - Opcode: 12 34 XX XX
  - Address: 56 78
    - LE: 7856 (byte addr), F0AC (WORD addr)
    - BE: 5678 (byte addr) , ACF0 (WORD addr)

CALL search results

- Only one match!

```
Trying 8c0d as RET
After-ret-addr-set-size: 430
Matching call opcodes for 1, ff00ff00, 1: 000b003d: total: 1275, hits: 843 (66%),
    misses: 432 (33%), coverage: 76%
```

- 430 – number of positions after RET candidate
- 1275 – total DWORDs with mask 00ff00ff
- 843 – calls to addresses right after RET
- 432 – calls to somewhere else
- 76% positions after RET are covered

RET search results

- RET: 8c0d (WORD LE)
- Absolute call to HHLL:

0bHH 3dLL (2 WORD LE)

- Confirmation: code areas end with RET:

```
0000de30 3d88 8c0d 3901 0b24 3daf 2b00 a942 2b00
0000de40 b961 8c0d ???? ???? ???? ???? ???? ????
```

JMP search heuristics

- Assumption 1: absolute unconditional JMP has structure similar to call: XXHH YYLL
- Assumption 2: there must be no JMP’s to outside of code
- Assumption 3: there may be JMP’s to addresses following RETs
- Assumption 4: absolute JMPs are not rare

JMP search results

- Search result:

```
Candidate opcode: 000b000c
  total: 82, hits: 7, misses: 0 adds: 0400 65a8 668e
Candidate opcode: 004e000b
  total: 352, hits: 1, misses: 0 adds: 37c6
Candidate opcode: 004e00bf
  total: 82, hits: 3, misses: 0 adds: 39b4
Candidate opcode: 00d8008c
  total: 29, hits: 5, misses: 0 adds: 311c 5232
```

- Arguments for 0bHH 0cLL:
  - Similarity to CALL encoding
  - Some code blocks end with this instruction

Rel. JMP search heuristics

- Set of target addresses: addresses after RET and JMP and first addresses of code segments
- Assumption 1: offset occupies one byte
- Assumption 2: offset is added to the address of the next instruction
- Assumption 3: code is WORD-addressed
- Assumption 4: relative jumps rarely cross RET instructions
- Assumption 5: no relative jumps to outside of code
- Search produces 32 candidates

Rel. JMP search first results

- Candidate opcode 0cXX - Similar to absolute unconditional JMP 0bHH 0cXX
- Assumption 1: 0bHH – prefix instruction making jump (or call) absolute
- Assumption 2: 0cXX – unconditional relative jump
- Redo relative jump instruction search with two new assumptions

Example

```
1100: ...
1110: 1c27          J?? 1160  # Jump cand. insn
                              # 1112 + 27 * 2 == 1160
1112: ...

115c: 0b0f 0c30     JMP 0f30  # Uncond. abs jump
1160: ...                     # Jump target

1218: 0c12          JMP 123e  # Uncond rel. jump
121a: ...                     # Jump target
123c: 5cee          J?? 121a  # Jump cand. insn
                              # 123e + ffee * 2 == 121a

123e: ...
```

Rel. JMP search results

Search results:

Candidate opcode: 1c00
total: 207, hits: 92, misses: 0, xrets = 12
Candidate opcode: (0b00) 2c00
total: 159, 0b_prefixed: 2, hits: 55, misses: 0, xrets = 19
Candidate opcode: (0b00) 3c00
total: 78, 0b_prefixed: 2, hits: 36, misses: 0, xrets = 20
Candidate opcode: 4c00
total: 81, hits: 40, misses: 0, xrets = 5
Candidate opcode: (0b00) 5c00
total: 93, 0b_prefixed: 2, hits: 43, misses: 0, xrets = 12
Candidate opcode: (0b00) 6c00
total: 182, 0b_prefixed: 2, hits: 72, misses: 0, xrets = 5
Candidate opcode: (0b00) 7c00
total: 147, 0b_prefixed: 1, hits: 81, misses: 0, xrets = 23

Assumption: these are relative conditional jumps

Intermediate results

- Instructions identified:
  - CALL
  - RET
  - JMP
  - Conditional JMPs
- High-byte extenstion prefix is identified
- Control-flow graph can be built and the general structure can be identified

Cond. arithmetics heuristics

- Assumption 1: there are instructions like AND and CMP with immediate arguments preceding conditional jumps
- Assumption 2: the opcodes are XXLL or 0bHH XXLL (byte or WORD values)
- Build the set of opcodes and the corresponding values

Cond. arithm. search results

- Search results:

Jump opcode 1c (207)
  Opcode: 1a
    0001: 1
    0002: 1
    ...
    0100: 1
    0200: 1
    1000: 1
    4000: 1
...

Arith. refinement

- Opcodes 1c, 5c: JZ, JNZ
- Opcodes 3c, 7c: JEQ, JNE
- Opcode 1a: AND with immediate argument
- Opcode 78: CMP with immediate argument
  - Opcode 78 always occurs just before conditional jumps 3c and 7c
- Opcode f8: also always occurs just before conditional jumps

Load-store pattern

- Patterns:

0bHH 3fLL 890f 4a01 8f09

0bHH 3fLL 990f 4a01 8f19

Assumption 1: memory load, +1 (or -1), memory store

Assumption 2: registers 0 and 1 are used

4a01 is never used before condjumps -> ADD

```
MOV DP0, HHLL
MOV R0, @DP0
ADD ACC, 1
MOV @DP0, R0
```

Memory clear pattern

- Often used:

0bHH 3fLL 0f00

- Corresponds to

MOV DP0, HHLL

MOV @DP0, 0

Arithmetics search results

- (0bHH) 2aLL – OR immediate
- (0bHH) 3aLL – XOR immediate
- (0bHH) 4aLL – ADD immediate
- (0bHH) 5aLL – SUB immediate
- 0bHH 3fLL – load memory address
- 890f – load R0 from memory
- 990f – load R1 from memory
- 8f09 – store R0 to memory
- 8f19 – store R1 to memory

Operation encoding

- Known MOVs:

```
890f   MOV   R0, @DP0
990f   MOV   R1, @DP0
8f09   MOV   @DP0, R0
8f19   MOV   @DP0, R1
0f00   MOV   @DP0, 0
->
????   MOV   R0, R1
????   MOV   R1, R0
????   MOV   R0, 0
????   MOV   R1, 1
```

- Known operations:

```
5a01   SUB   ACC, 1
->
????   SUB   ACC, R0
```

Operation encoding

- Known MOVs:

```
890f   MOV   R0, @DP0
990f   MOV   R1, @DP0
8f09   MOV   @DP0, R0
8f19   MOV   @DP0, R1
0f00   MOV   @DP0, 0
->
8919   MOV   R0, R1
9909   MOV   R1, R0
0900   MOV   R0, 0
1901   MOV   R1, 1
```

- Known operations:

```
5a01   SUB   ACC, 1
->
da09   SUB   ACC, R0
```

Register structure

- Instruction 08RR changes the active accumulator (0800, 0801 … 080f)
- Arithmetics: one operand is explicit, another is active accumulator
- The processor has at least 16 arithmetic registers 0 - F

Results

00002a40	0800	0b80	3a00	0802	0b80	3a00	da29	8c0d
00002a50	4e1c	0b01	3f25	890f	8c0d	4e1c	0b01	3f26
00002a60	890f	8c0d	4e2c	bf3f	890f	4e88	8c0d	4e28
00002a70	0b03	3f27	0f0f	4e58	2b51	3f22	898f	0b03
00002a80	3f62	0b0f	3d88	0b03	3f8c	0b0f	3d92	0b03
00002a90	3f16	0b0f	3d92	0b03	3f72	8f09	8958	1aff
00002aa0	d807	4e1c	0b01	3f2e	0b0f	3d22	0b03	3fac
00002ab0	0b0f	3df2	0b02	3f70	0b0d	3d8b	0b01	3f55
00002ac0	0b0e	3d22	0b02	3f70	0b0f	3d14	0b02	3f7b
00002ad0	0b0c	3d8b	0b02	3f50	0b0f	3daf	0b02	3f7b

Total: 16862, Code: 13734 (81%)

Different 16-bit values: 2085, known: 1618 (77%)

Top values

```
0b01    PREFIX
0800    MOV AP, 0     # change the current accumulator
8c0d    RET
2b00    PREFIX
4e1c    ?
0801    MOV AP, 1     # change the current accumulator
890f    MOV R0, @DP0  # load from data memory
8f09    MOV @DP0, R0  # store to data memory
0f00    MOV @DP0, 0   # store 0 to data memory
0b00    PREFIX
0b8f    PREFIX
4a01    ADD ACC, 1
0b36    PREFIX
0802    MOV AP, 1
3ddc    CALL ...
0b80    PREFIX
990f    MOV R1, @DP0
8afa    ?
1aff    AND ACC, ff
0900    MOV R0, 0
```

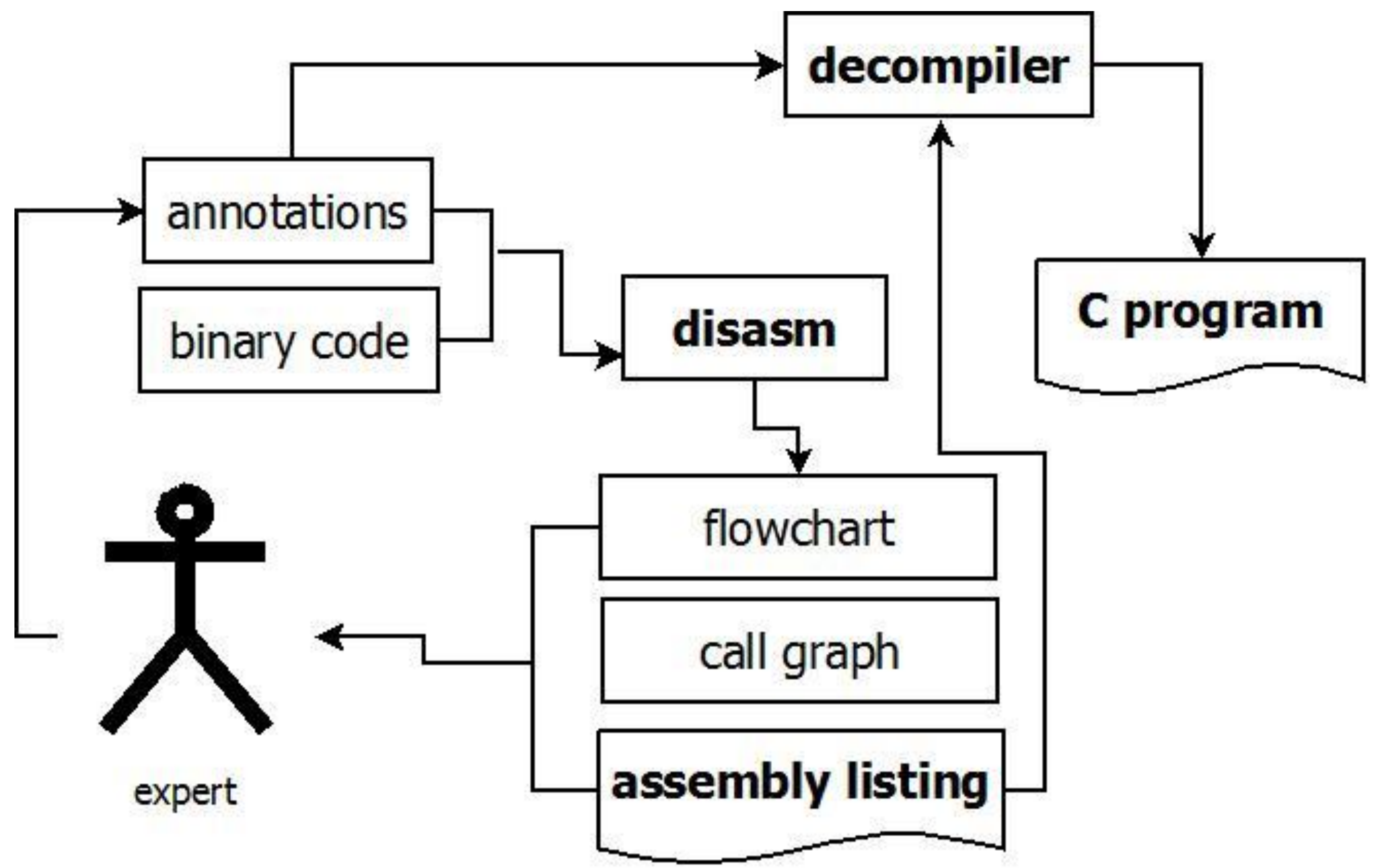
Lessons learned

- It is possible to discover
  - Subroutine structure
  - Unconditional and conditional jumps
  - Some arithmetic instructions
  - Rough register structure
- Only by binary analysis of the code without virtual machine (processor) data sheets

Limitations

- No obfuscation
- Most subroutines follow each other

Tool support



annotations

- Opcode specifications
- Specification of code and data areas
- Entry points
- Symbolic cell names
- Subroutine range and description
- Inline and outline specifications

SmartDec decompiler

- Demo version is free available at decompilation.info
- Current state:
  - Alpha-version
  - Supports limited set of x86/x64 assembly instructions
  - Supports objdump/dumpbin disassembly backbends
  - Initial support for C++ exceptions and class hierarchy recovery

Thank you for your attention

Questions?

info@decompilation.info