Datalog Disassembly

Antonio Flores Montoya and Eric Schulte
GrammaTech, Inc.
August 13, 2020
Motivation

Sometimes the **source code is not available**
Sometimes the **source code is not available**
Disassembly is the first step in:

- Binary analysis
- Binary rewriting/hardening
We want to obtain *reassembleable assembly*
- Assembly code with cross references
- We can modify it without breaking it

This involves two tasks:
- Instruction Boundary Identification
- Symbolization
A binary looks like this:

```
ba 08 3b 44 00 31 c9 0f
1f 80 00 00 00 00 48 39
3a ... 48 83 c2 20 48 83
f9 3a 7e ee
```
A binary looks like this:

\[
\begin{array}{cccccccc}
\text{ba} & 08 & 3b & 44 & 00 & \mid & 31 & c9 & 0f \\
1f & 80 & 00 & 00 & 00 & 00 & \mid & 48 & 39 \\
3a & \mid & \ldots & \mid & 48 & 83 & c2 & 20 & \mid & 48 \\
83 & f9 & 3a & \mid & 7e & ee & \mid \\
\end{array}
\]

Instruction Boundary Identification amounts to finding which addresses correspond to which instructions.
Instruction Boundary Identification

Challenging because:

- X64 instructions have variable sizes
- Data interleaves with instructions
Symbolization: Which numbers are references vs. literals

Symbolization

40c7f2: mov EDX, 443b08
40c7f7: xor ECX,ECX
40c7f9: nop
40c800: cmp [RDX],RDI

40c808: add RDX,32
40c80c: cmp RCX,58
40c810: jle 40c800

mov EDX, Label_1
xor ECX,ECX
nop
Label_2: cmp [RDX],RDI

add RDX,32
cmp RCX,58
jle Label_2

443b40: 04 04 00 00 00 00 00 00 04 04 00 00 00 00 00 00
443b48: d0 50 41 00 00 00 00 00 00 00
443b50: 85 48 44 00 00 00 00 00 00 00

04 04 00 00 00 00 00 00
.quad Label_3
.quad Label_4
Symbolization: Which numbers are references vs. literals

40c7f2: mov EDX, 443b08
40c7f7: xor ECX, ECX
40c7f9: nop
40c800: cmp [RDX], RDI

Symbolic references allow us to move and modify the code without breaking it.

mov EDX, Label_1
xor ECX, ECX
nop

Label_2: cmp [RDX], RDI
add RDX, 32
cmp RCX, 58
jle Label_2

04 04 00 00 00 00 00 00
04 04 00 00 00 00 00 00
.quad Label_3
.quad Label_4
Symbolization: Which numbers are references vs. literals

40c7f2: mov EDX, 443b08
40c7f7: xor ECX,ECX
40c7f9: nop
40c800: cmp [RDX],RDI
40c808: add RDX,32
40c80c: cmp RCX,58
40c810: jle 40c800

443b40: 04 04 00 00 00 00 00 00
443b48: d0 50 41 00 00 00 00 00
443b50: 85 48 44 00 00 00 00 00

Symbolic references allow us to move and modify the code without breaking it.
Relocation information is NOT enough (even for PIE).
Our approach: Use Datalog

Both instruction boundary identification and symbolization are hard problems. We want to:

- Combine different heuristics easily
- Use simple static analyses to inform our decisions
- Run it quickly
Binaries are encoded as **facts** (our initial knowledge of the binary)

- We decode **every possible** offset in code sections (obtaining a **superset** of all possible instructions in the code)

  Instruction:

  4000A0: mov RAX, 420020

  Becomes:

  instruction(4000A0, 5, '', 'MOV', 1, 2, 0, 0)
  op_regdirect(1, 'RAX')
  op_immediate(2, 420020)

  If the decoding fails at address A, we generate invalid(A)
Analyses and heuristics are expressed as Datalog rules:

**Backward traversal that propagates invalid instructions**

```prolog
invalid(From):-
    (must_fallthrough(From,To);
    direct_jump(From,To);
    direct_call(From,To)
    ),(
    invalid(To);
    !instruction(To,_,_,_,_,_,_,_,_)
    ).
```

If there is an instruction at address From that must fallthrough, or jumps, or calls an address To that contains an invalid instruction or no instruction at all, then the instruction at From is also invalid.
1. Backward traversal: propagate invalid instructions
2. Forward traversal: build superset of all possible basic blocks
   ▶ Hybrid between linear sweep and recursive traversal
   ▶ Use potential references in data sections
3. Assign points to candidate blocks using heuristics
   ▶ Entry point: +20
   ▶ Address appears in data section: +1
   ▶ Direct jump: +6
   ▶ …
4. Aggregate points to resolve overlaps (Datalog extension)
Symbolization

Naive approach

- Numbers in the binary address range $\rightarrow$ symbols
- Numbers outside the range $\rightarrow$ literals

- False positives: A literal coincides with the binary address range
- False negatives: (see paper)
Collect **additional evidence** (how the number is used) using supporting analyses and heuristics.

Assign points to candidates:
- Symbol
- Symbol-Symbol
- Strings
- Other (data elements with different size)

Aggregate points to make a decision.
Supporting Analyses: Def-Uses

**Predicate:**  \( \text{def}_{\text{used}}(A_{\text{def}}, R_{\text{reg}}, A_{\text{use}}) \)

Register \( R_{\text{reg}} \) is defined in \( A_{\text{def}} \) and used in \( A_{\text{use}} \)

40c7f2:  \text{mov} EDX, 443b08
40c7f7:  \text{xor} ECX,ECX
40c7f9:  \text{nop}
40c800:  \text{cmp} [RDX],RDI
        :       :
40c808:  \text{add} RDX,32
40c80c:  \text{cmp} RCX,58
40c810:  \text{jle} 40c800
        :       :
443b28:  \ldots
Predicate: \( \text{reg\_val}(A1,\text{Reg1},A2,\text{Reg2},\text{Mult},\text{Disp}) \)

\[ \text{value(Reg1} @ A1) = \text{value(Reg2} @ A2) \times \text{Mult} + \text{Disp} \]

```
40c7f2:  mov  EDX, 443b08
40c7f7:  xor  ECX,ECX
40c7f9:  nop
40c800:  cmp [RDX],RDI
        ... ...
40c808:  add  RDX,32
40c80c:  cmp  RCX,58
40c810:  jle  40c800
        ... ...
```

\[ \text{RDX=443b08} \]
Supporting Analyses: Register value analysis

**Predicate:**  \( \text{reg\_val(A1,Reg1,A2,Reg2,Mult,Disp)} \)

\[ \text{value(Reg1@A1)} = \text{value(Reg2@A2)} \times \text{Mult} + \text{Disp} \]

```
40c7f2:  mov EDX, 443b08
40c7f7:  xor ECX,ECX
40c7f9:  nop
40c800:  cmp [RDX],RDI
        ... ...
40c808:  add RDX,32
40c80c:  cmp RCX,58
40c810:  jle 40c800
        ... ...
443b28:  ... ...
```

\( \text{RDX=}?\times32+443b28 \)
Supporting Analyses: Data access patterns

**Predicate:** `data_access_pattern(Addr, Size, Mult, Addr2)`

`Addr` is accessed with size `Size` and multiplier `Mult` from `Addr2`

```
40c7f2:  mov EDX, 443b08
40c7f7:  xor ECX,ECX
40c7f9:  nop
40c800:  cmp [RDX],RDI
        :  :
40c808:  add RDX,32
40c80c:  cmp RCX,58
40c810:  jle 40c800
        :  :
443b28:  ...
```

RDX = ?*32 + 443b28

access Qword with x32 multiplier
Assigning points

Use supporting analyses to enhance confidence

<table>
<thead>
<tr>
<th>Candidates in data section</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ Pointer to instruction beginning 👍</td>
</tr>
<tr>
<td>➤ Data access match 👍</td>
</tr>
<tr>
<td>➤ Data access conflict 👎</td>
</tr>
<tr>
<td>➤ ...</td>
</tr>
</tbody>
</table>
Our tool Ddisasm supports x64 Linux ELF binaries.

We test our disassembler Ddisasm with:

- 3 benchmark sets
- 7 compiler versions (GCC, Clang and ICC).
- 6 compiler optimization flags

A total of 7658 binaries

Compare to Ramblr state-of-the-art tool in reassembleable disassembly.
Results

% correctly disassembled binaries

<table>
<thead>
<tr>
<th></th>
<th>Ddisasm</th>
<th>Ramblr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolization information</td>
<td>99.72%</td>
<td>79.74%</td>
</tr>
<tr>
<td>Functionality of reassembled binary</td>
<td>99.77%</td>
<td>59.41%</td>
</tr>
</tbody>
</table>
Disassembly Time in Seconds
Conclusion

- Reassemblable disassembly is undecidable
- Practical solutions benefit from analysis and heuristics
- Datalog works for both:
  - Express analysis concisely: less error-prone, fast development
  - Easy to experiment with heuristics expressed as Datalog rules
- Ddisasm is faster and achieves better precision than the state-of-the-art.
Questions?

- Contact: afloresmontoya@grammatech.com
- Tool: https://github.com/GrammaTech/ddisasm
- Experimental evaluation: https://zenodo.org/record/3691736
- Rewriting tutorial using Ddisasm: https://grammatech.github.io/gtirb/md_stack-stamp.html