N-version Disassembly: Differential Testing of x86 Disassemblers

Roberto Paleari\textsuperscript{1}  Lorenzo Martignoni\textsuperscript{2}  Giampaolo Fresi Roglia\textsuperscript{1}  Danilo Bruschi\textsuperscript{1}

\textsuperscript{1}Università degli Studi di Milano  \textsuperscript{2}Università degli Studi di Udine

International Conference on Software Testing and Analysis (ISSTA 2010)
Disassemblers

- Translate machine code into assembly instructions
- Possible uses:
  - Debuggers
  - Binary analysis tools
  - CPU emulators
  - Sandboxes (e.g., Google Native Client)
  - . . .
Implications of incorrect disassembly

- Disassembly is the front end of many analyses that deal with machine code
- An error in the disassembler has \textit{cascade effects} on all the subsequent analysis modules!
Developing disassemblers

It sounds like a trivial task but . . .
Developing disassemblers

It sounds like a trivial task but . . .

A glimpse at Intel x86

- CISC architecture
- $700^+$ possible opcodes
- Instructions have variable length, may have prefixes, support multiple addressing modes
- Several instruction set extensions
  (MMX, SSE, SSE2, SSE3, SSSE3, SSE4, VMX, . . .)
Developing disassemblers

It sounds like a trivial task but . . .

A glimpse at Intel x86

- CISC architecture
- $700^+$ possible opcodes
- Instructions have variable length, may have prefixes, support multiple addressing modes
- Several instruction set extensions (MMX, SSE, SSE2, SSE3, SSSE3, SSE4, VMX, . . .)

Intel x86 disassemblers include about 9000 lines of code!
How disassemblers work?

Our goal is to test the instruction decoder component of Intel x86 disassemblers.

-$c$ $08$ $6b$ $01$ $00$

-$b$ $93$ $08$ $00$ $00$ $00$

-$d$ $2$

81  c3  08  6b  01  00
8b  93  08  00  00  00
85  d2

R. Paleari, L. Martignoni, G. Fresi Roglia, D. Bruschi

N-version Disassembly: Differential Testing of x86 Disassemblers
How disassemblers work?

Our goal is to test the instruction decoder component of Intel x86 disassemblers.
How disassemblers work?

Select next instruction

Decode instruction

add ebx,0x16b08
8b 93 08 00 00 00
85 d2
How disassemblers work?

Select next instruction

Decode instruction

add ebx, 0x16b08

8b 93 08 00 00 00

85 d2
How disassemblers work?

Add ebx, 0x16b08
mov edx, [ebx+0x8]
85 d2
How disassemblers work?

Our goal is to test the instruction decoder component of Intel x86 disassemblers.

R. Paleari, L. Martignoni, G. Fresi Roglia, D. Bruschi
N-version Disassembly: Differential Testing of x86 Disassemblers

```
add ebx, 0x16b08
mov edx, [ebx+0x8]
85 d2
```
How disassemblers work?

Our goal is to test the instruction decoder component of Intel x86 disassemblers.

```
add ebx,0x16b08
mov edx,[ebx+0x8]
test edx,edx
```
How disassemblers work?

Our goal is to test the **instruction decoder** component of Intel x86 disassemblers.
N-version disassembly

Idea

- Differential testing of $n - 1$ disassemblers, with an oracle (the $n^{th}$ disassembler)
- Disassemblers that disagree with the oracle are wrong
- The higher the number of agreeing disassemblers, the higher the confidence in their result

Challenges

- How to develop the oracle?
- How to compare the output of different disassemblers?
- How to generate test cases?
CPU-assisted instruction decoding

- The CPU is the perfect decoder
- Our oracle is an instruction decoder that **leverages the physical CPU**
- The oracle can detect:
  1. If a sequence of bytes encodes a valid instruction
  2. Length of the instruction
  3. Format of non-implicit operands
CPU-assisted decoding: Instruction length

- **Idea**: exploit the fact that the CPU fetches instruction bytes incrementally
- Position an instruction across two memory pages with different permission, and observe the behavior of the CPU
**CPU-assisted decoding: Instruction length**

- **Idea**: exploit the fact that the CPU fetches instruction bytes incrementally

- Position an instruction across two memory pages with different permission, and observe the behavior of the CPU

\[
B = 88 \ b7 \ 53 \ 10 \ \text{fa} \ \text{ca} \ldots
\]

\[
\text{mov} \ [\text{edi+0xcafa1053}],\text{dh}
\]

(valid instruction, six bytes long)

---

In this diagram, the memory address range is shown from 0x1f000 to 0x20fff. The instruction
\[
B = 88 \ b7 \ 53 \ 10 \ \text{fa} \ \text{ca} \ldots
\]

is placed across two memory pages. The left page from 0x1f000 to 0x1fff is readable and executable, while the right page from 0x20000 to 0x20fff is any access is forbidden.
CPU-assisted decoding: Instruction length

- **Idea**: exploit the fact that the CPU fetches instruction bytes incrementally
- Position an instruction across two memory pages with different permission, and observe the behavior of the CPU

\[
B = 88 \ b7 \ 53 \ 10 \ fa \ ca \ldots \\
mov [edi+0xcafa1053], dh \\
(\text{valid instruction, six bytes long})
\]

<table>
<thead>
<tr>
<th></th>
<th>0x1f000</th>
<th>.....</th>
<th>0x1ffff</th>
<th>0x20000</th>
<th>.....</th>
<th>0x20fff</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_1 )</td>
<td></td>
<td></td>
<td>88 b7 53 10 fa ca ...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page fault (on execution) at address 0x20000

\[ \text{Longer} \]
CPU-assisted decoding: Instruction length

- **Idea**: exploit the fact that the CPU fetches instruction bytes incrementally
- Position an instruction across two memory pages with different permission, and observe the behavior of the CPU

\[ B = \text{88 b7 53 10 fa ca ...} \]
\[ \text{mov [edi+Oxcafa1053],dh} \]
(Valid instruction, six bytes long)

Page fault (**on execution**) at address **0x20000**

\[ B_2 \]

0x1f000 \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots 0x1ffff \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots 0x20000 \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots 0x20fff

\text{Longer}
CPU-assisted decoding: Instruction length

- **Idea**: exploit the fact that the CPU fetches instruction bytes incrementally
- Position an instruction across two memory pages with different permission, and observe the behavior of the CPU

\[
B = 88 \ b7 \ 53 \ 10 \ fa \ ca \ \ldots \\
\text{mov} \ [\text{edi}+0xcafa1053],\text{dh} \\
(\text{valid instruction, six bytes long})
\]

\[
\begin{array}{cccccccc}
0x1f000 & \ldots & 0x1ffff & 0x20000 & \ldots & 0x20fff \\
B_6 \ & \boxed{88 \ b7 \ 53 \ 10 \ fa \ ca} & \ldots & \\
\end{array}
\]

Page fault (**on write**) at address 0x78378943

\[\checkmark\]

Valid
**CPU-assisted decoding: Instruction length**

- **Idea:** exploit the fact that the CPU fetches instruction bytes incrementally
- Position an instruction across two memory pages with different permission, and observe the behavior of the CPU

\[ B = \text{f0 00 c0} \ldots \]

(invalid)

<table>
<thead>
<tr>
<th>0x1f000</th>
<th>…………</th>
<th>0x1fff</th>
<th>0x20000</th>
<th>…………</th>
<th>0x20fff</th>
</tr>
</thead>
</table>

---
CPU-assisted decoding: Instruction length

**Idea**: exploit the fact that the CPU fetches instruction bytes incrementally

- Position an instruction across two memory pages with different permission, and observe the behavior of the CPU

\[
\begin{align*}
B &= \text{f0 00 c0 } \ldots \\
\text{(invalid)}
\end{align*}
\]

\[
\begin{array}{cccc}
0x1000 & \cdots & 0x1fff & 0x20000 & \cdots & 0x20fff \\
\hline
B_1 & \text{f0 00 c0 } \ldots & \text{Longer}
\end{array}
\]

Page fault *(on execution)* at address 0x20000
CPU-assisted decoding: Instruction length

- Idea: exploit the fact that the CPU fetches instruction bytes incrementally
- Position an instruction across two memory pages with different permission, and observe the behavior of the CPU

\[ B = \text{invalid} \]

\[ B_2 = \text{f0 00 c0 ...} \]

Page fault (on execution) at address 0x20000

Longer
CPU-assisted decoding: Instruction length

**Idea:** exploit the fact that the CPU fetches instruction bytes incrementally

Position an instruction across two memory pages with different permission, and observe the behavior of the CPU

\[ B = \text{f0 00 c0 } \ldots \]

(invalid)

\[ 0x1f000 \ldots 0x1ffff \quad 0x20000 \ldots 0x20fff \]

\[ B_3 = \text{f0 00 c0 } \cdots \]

Invalid instruction at address 0x1fff

\[
\text{Invalid}
\]
CPU-assisted decoding: Non-implicit operands

- **Idea**: change the bytes that follow the opcode, and observe how the CPU behaves
- The instruction will be invalid if we replace an operand with another one of a different type
CPU-assisted decoding: Non-implicit operands

- **Idea**: change the bytes that follow the opcode, and observe how the CPU behaves
- The instruction will be invalid if we replace an operand with another one of a different type

\[
B = \text{88 b7 53 10 fa ca}
\]

\[
\text{mov [edi+0xcafa1053],dh}
\]

\[
0x1f000 \ldots \ldots \ldots \ldots 0x1ffff \ldots \ldots \ldots \ldots 0x20fff
\]
CPU-assisted decoding: Non-implicit operands

**Idea:** change the bytes that follow the opcode, and observe how the CPU behaves

The instruction will be invalid if we replace an operand with another one of a different type

\[ B = 88 \ b7 \ 53 \ 10 \ \text{fa} \ \text{ca} \]

\[
\text{mov} \ [\text{edi+0xcafa1053}],\text{dh}
\]

\[
\begin{array}{cccccccc}
0x1f000 & \cdots & 0x1ffff & 0x20000 & \cdots & 0x20fff \\
\hline
88 \ 00 & 53 \ 10 & \text{fa} \ \text{ca}
\end{array}
\]

\[
\text{mov} \ [\text{eax}], \text{al}
\]

Page fault (*on write*) at address 0x00 → Valid
CPU-assisted decoding: Non-implicit operands

- **Idea**: change the bytes that follow the opcode, and observe how the CPU behaves
- The instruction will be invalid if we replace an operand with another one of a different type

\[ B = 88 \ b7 \ 53 \ 10 \ fa \ ca \]

\[
\text{mov } [\text{edi+0xcafa1053}], \text{dh}
\]

\[
\begin{array}{cccc}
0x1f000 & \cdots & 0x1ffff & \cdots & 0x20000 & \cdots & 0x20fff \\
B_3 & 88 & 40 & 00 & 10 & \text{fa} & \text{ca} \\
\text{mov } [\text{eax+0x0}], \text{al} \\
\text{Page fault (on write) at address } 0x00 \rightarrow \text{Valid}
\end{array}
\]
CPU-assisted decoding: Non-implicit operands

- **Idea**: change the bytes that follow the opcode, and observe how the CPU behaves
- The instruction will be invalid if we replace an operand with another one of a different type

\[ B = 88 \hspace{1em} b7 \hspace{1em} 53 \hspace{1em} 10 \hspace{1em} fa \hspace{1em} ca \]

\[
\text{mov } [\text{edi}+0\text{xcafa1053}], \text{dh} \\
\]

- \( B_4 \)

\[
0x1f000 \ldots \ldots \ldots \ldots \ldots 0x1fff \ldots \ldots \ldots \ldots \ldots 0x20000 \ldots \ldots \ldots \ldots \ldots 0x20fff \\
\]

\[
\begin{array}{c}
\begin{array}{c}
88 \hspace{1em} 44 \hspace{1em} 25 \hspace{1em} 00 \\
\text{fa} \hspace{1em} \text{ca} \\
\end{array}
\end{array}
\]

\[
\text{mov } [\text{ebp}+0\text{x}0], \text{al} \\
\text{Page fault (on write) at address } 0x00 \rightarrow \text{Valid}
\]
CPU-assisted decoding: Non-implicit operands

★ *Idea:* change the bytes that follow the opcode, and observe how the CPU behaves

★ The instruction will be invalid if we replace an operand with another one of a different type

\[ B = 88 \text{ b7 53 10 fa ca} \]

```assembly
mov [edi+0xcafa1053],dh
```

\[ B_7 = 88 \text{ 04 25 00 00 00 00} \]

```assembly
mov [0x0], al
```

Page fault (*on write*) at address 0x00 → *Valid*

Test passed
Operand is an addressing-form specifier
**CPU-assisted decoding: Non-implicit operands**

- **Idea**: change the bytes that follow the opcode, and observe how the CPU behaves.
- The instruction will be invalid if we replace an operand with another one of a different type.

\[ B = 05\ 12\ 34\ 56\ 78 \]
\[ \text{add eax,0x78563412} \]

\[
\begin{array}{cccc}
0x1f000 & \cdots & 0x1ffff & 0x20000 & \cdots & 0x20fff \\
\end{array}
\]
CPU-assisted decoding: Non-implicit operands

- **Idea**: change the bytes that follow the opcode, and observe how the CPU behaves.
- The instruction will be invalid if we replace an operand with another one of a different type.

\[
B = 05 \ 12 \ 34 \ 56 \ 78 \\
\text{add eax, 0x78563412}
\]

\[
\begin{array}{cccc}
0x1f000 & \cdots & 0x1ffff & \cdots \\
0x20000 & \cdots & 0x20fff \\
\end{array}
\]

\[
B_2 = \begin{array}{c|c|c|c}
05 & 00 & 34 & 56 & 78 \\
\end{array}
\]

Page fault (on execution) at address 0x20000 → Longer

Test failed
Operand is **not** an addressing-form specifier
CPU-assisted decoding: Non-implicit operands

**Idea:** change the bytes that follow the opcode, and observe how the CPU behaves

- The instruction will be invalid if we replace an operand with another one of a different type

\[ B = 05 \ 12 \ 34 \ 56 \ 78 \]

\[ \text{add eax,0x78563412} \]

\[
\begin{array}{cccc}
0x1f000 & \cdots & 0x1ffff & 0x20000 & \cdots & 0x20fff \\
B_5 & 05 \ 00 \ 00 \ 00 \ 01 & \\
\end{array}
\]

\[ \text{add eax, 0x1000000} \]

No exception \(\rightarrow\) Valid
CPU-assisted decoding: Non-implicit operands

★ Idea: change the bytes that follow the opcode, and observe how the CPU behaves

★ The instruction will be invalid if we replace an operand with another one of a different type

\[ B = 05\ 12\ 34\ 56\ 78 \]
add eax, 0x78563412

\[ B'_5 \]
0x1f000 \ldots \ldots 0x1ffff \ldots \ldots 0x20000 \ldots \ldots 0x20fff

05 00 00 00 02

add eax, 0x2000000
No exception \(\rightarrow\) Valid
CPU-assisted decoding: Non-implicit operands

- **Idea:** change the bytes that follow the opcode, and observe how the CPU behaves
- The instruction will be invalid if we replace an operand with another one of a different type

\[ B = 05 \ 12 \ 34 \ 56 \ 78 \]
\[ \text{add eax}, \text{0x78563412} \]

\[
\begin{array}{cccccc}
0x1f000 & \cdots & 0x1ffff & 0x20000 & \cdots & 0x20fff \\
05 \ 00 \ 00 \ 00 \ \text{ff} & \text{Valid} & \\
\text{add eax, 0xff000000} & \\
\text{No exception} & \rightarrow \text{Valid} & \\
\end{array}
\]

Test passed
Operand is a 32-bit immediate
Comparing the output of disassemblers

- The outputs of disassemblers differ for many subtle details

```
88 b7 53 10 fa ca
77 92 a4 9c 4a
```

```
D_1 -> 88 b7 53 10 fa ca
D_2 -> mov 0xcafa1053[edi],esi
D_3 -> mov [edi-0x3f5e9ad],dh
D_4 -> MOV [EDI+0xCAFA1053],DH
```

```
D_1 -> 0
D_3, D_4 -> 0.66
(Disagrees with the oracle)
```
Comparing the output of disassemblers

- The outputs of disassemblers differ for many subtle details
- We **normalize** the outputs through a set of hand-written rules

```
88 b7 53 10 fa ca 77 92 a4 9c 4a
```

Instruction normalizer

```
D_1: mov 0xcafa1053[edi],esi
D_2: mov [edi-0x3505efad],dh
D_3: MOV [EDI+0xCAFA1053],DH → invalid
D_4: mov [edi+0xcafa1053],esi
D_4: mov [edi+0xcafa1053],dh
D_4: mov [edi+0xcafa1053],dh → invalid
```
Comparing the output of disassemblers

- The outputs of disassemblers differ for many subtle details
- We **normalize** the outputs through a set of hand-written rules
- Normalized outputs are then grouped into equivalence classes

```
88 b7 53 10 fa ca 77 92 a4 9c 4a
```

```
D_1
mov 0xcafa1053[edi],esi
D_2
mov [edi−0x3505efad],dh
D_3
MOV [EDI+0xCAFA1053],DH
D_4
invalid
```

```
Instruction normalizer
```

```
mov [edi+0xcafa1053],esi
mov [edi+0xcafa1053],dh
mov [edi+0xcafa1053],dh
invalid
```

```
Output evaluator
```

```
\{D_1\} \rightarrow 0.33
\{D_2, D_3\} \rightarrow 0.66
\{D_4\} \rightarrow 0
```

(Disagrees with the oracle)
Input generation

Random input generation

- Intel x86 instruction set is very dense
- \( \sim 75\% \) of randomly generated strings represent valid instructions
- Can produce invalid or very “exotic” instructions
Input generation

Random input generation

🌟 Intel x86 instruction set is very dense
🌟 \(\sim 75\%\) of randomly generated strings represent valid instructions
🌟 Can produce invalid or very “exotic” instructions

CPU-assisted input generation

🌟 More exhaustive exploration of the instruction set, with low redundancy
🌟 Leverage the oracle to generate only valid instructions
🌟 Iterate over all opcodes up to three bytes, and combine them with different operands
Evaluation of the CPU-assisted decoder

- < 500 lines of C code
- Extensive manual evaluation of the source
- If two CPUs support the same features, the oracle produces the same output

Experiments

- 40k randomly-generated test-cases (16-byte strings)
- We decoded the strings on 4 CPUs and compared the outputs
- The only differences were due to different CPU features

<table>
<thead>
<tr>
<th>CPU</th>
<th>Supported features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MMX</td>
</tr>
<tr>
<td>Intel P3 (1.2GHz)</td>
<td>✓</td>
</tr>
<tr>
<td>Intel P4 (3.0GHz)</td>
<td>✓</td>
</tr>
<tr>
<td>Intel Core2 (2.0GHz)</td>
<td>✓</td>
</tr>
<tr>
<td>Intel Xeon (2.8GHz)</td>
<td>✓</td>
</tr>
</tbody>
</table>
Evaluation of off-the-shelf disassemblers

Setup

- 8 off-the-shelf disassemblers & binary analysis tools
- CPU-assisted decoder executed on a Intel Xeon (2.8GHz)

Test-cases

- About 60k test-cases
- $\frac{2}{3}$ generated randomly, $\frac{1}{3}$ with the CPU-assisted strategy
- Testing took $\sim$ 15 hours
Evaluation of off-the-shelf disassemblers

- diStorm64
- Ida Pro
- libopcode
- Native Client
- ndisasm
- OllyDBG
- Udis86
- XED2

Defects: O N I

R. Paleari, L. Martignoni, G. Fresi Roglia, D. Bruschi
N-version Disassembly: Differential Testing of x86 Disassemblers
Evaluation of off-the-shelf disassemblers

- **diStorm64**
- **Ida Pro**
- **libopcode**
- **Native Client**
- **ndisasm**
- **OllyDBG**
- **Udis86**
- **XED2**

Defects:
- **O**: Invalid instructions that are considered valid by the disassembler
- **N**: Instructions not recognized by the disassembler
- **I**: Invalid instructions that are not considered valid by the disassembler

R. Paleari, L. Martignoni, G. Fresi Roglia, D. Bruschi

N-version Disassembly: Differential Testing of x86 Disassemblers
Evaluation of off-the-shelf disassemblers

Valid instructions that are not recognized by the disassembler
Valid instructions for which there exists another output with an higher coefficient of agreement
Some of the defects we found

<table>
<thead>
<tr>
<th>Disass.</th>
<th>Input</th>
<th>Decoded instruction</th>
<th>Correct result</th>
</tr>
</thead>
<tbody>
<tr>
<td>diStorm64</td>
<td>26 59</td>
<td>invalid</td>
<td>es pop ecx</td>
</tr>
<tr>
<td>Ida Pro</td>
<td>f6 5c 34 ae</td>
<td>neg [esp+esi+0x52]</td>
<td>neg [esp+esi-0x52]</td>
</tr>
<tr>
<td>libopcode</td>
<td>d4 cd</td>
<td>aam 0xffffffffcd</td>
<td>aam 0xcd</td>
</tr>
<tr>
<td>NaCl</td>
<td>0f 21 83</td>
<td>mov dr0,ebx (7 bytes)</td>
<td>mov ebx,dr0</td>
</tr>
<tr>
<td>ndisasm</td>
<td>82 76 e5 dc</td>
<td>invalid</td>
<td>xor byte [esi-0x1b],0xdc</td>
</tr>
<tr>
<td>OllyDBG</td>
<td>d9 7f d2</td>
<td>fstcw [edi-0x2e]</td>
<td>fnstcw [edi-0x2e]</td>
</tr>
<tr>
<td>Udis86</td>
<td>db e0</td>
<td>invalid</td>
<td>fneni</td>
</tr>
<tr>
<td>XED2</td>
<td>8e 0b</td>
<td>mov cs, word [ebx]</td>
<td>invalid</td>
</tr>
</tbody>
</table>
Conclusions

- Disassemblers play an important role in tools that deal with machine code
- Fully automated testing methodology for x86 disassemblers
- Experimental evaluation over 8 off-the-shelf disassemblers

Limitations

- Normalization rules are hand-written
- The oracle cannot be easily adapted to other architectures
Thank you!
Any questions?

Roberto Paleari
roberto@security.dico.unimi.it