N-version Disassembly: Differential Testing of x86 Disassemblers

Roberto Paleari[1] Lorenzo Martignoni[2]
Giampaolo Fresi Roglia[1] Danilo Bruschi[1]

[1]Università degli Studi di Milano [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 **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

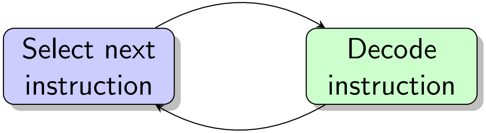
It sounds like a trivial task **but** . . .

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, . . .)

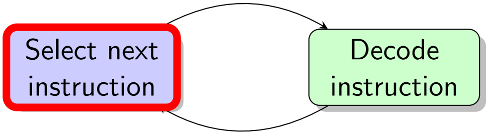
Intel x86 disassemblers include about **9000 lines of code**!

How disassemblers work?



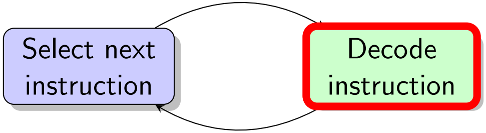
```
81 c3 08 6b 01 00
8b 93 08 00 00 00
85 d2
```

How disassemblers work?



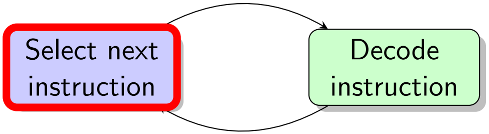
```
➔ 81 c3 08 6b 01 00
  8b 93 08 00 00 00
  85 d2
```

How disassemblers work?



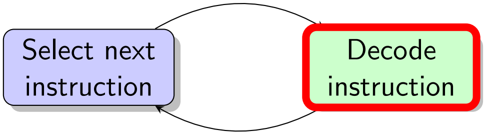
```
add ebx,0x16b08
8b 93 08 00 00 00
85 d2
```

How disassemblers work?



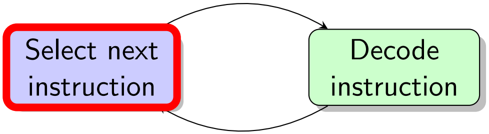
```
  add ebx,0x16b08
➔ 8b 93 08 00 00 00
  85 d2
```

How disassemblers work?



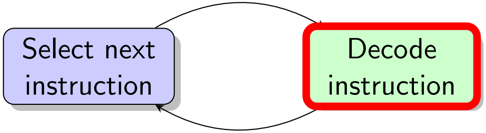
```
add ebx,0x16b08
mov edx,[ebx+0x8]
85 d2
```

How disassemblers work?



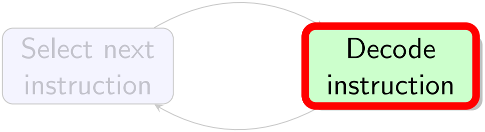
```
  add ebx,0x16b08
  mov edx,[ebx+0x8]
➔ 85 d2
```

How disassemblers work?



```
add ebx,0x16b08
mov edx,[ebx+0x8]
test edx,edx
```

How disassemblers work?



```
add ebx,0x16b08
mov edx,[ebx+0x8]
test edx,edx
```

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

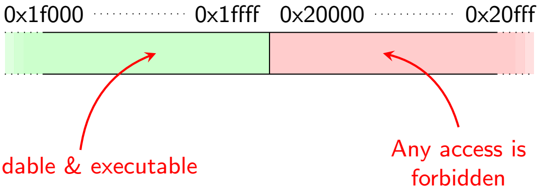
CPU-assisted decoding: Instruction length

- *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 fa ca ...`

`mov [edi+0xcafa1053],dh`

(valid instruction, six bytes long)



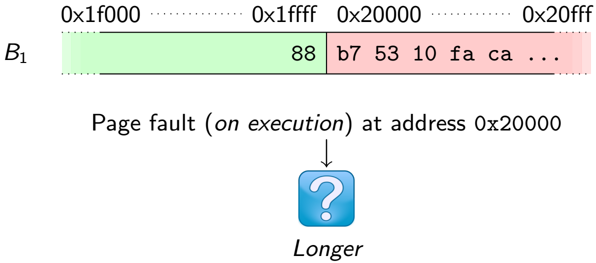
CPU-assisted decoding: Instruction length

- *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 ...`

`mov [edi+0xcafa1053],dh`

(valid instruction, six bytes long)



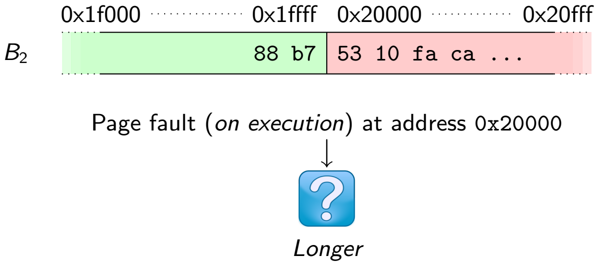
CPU-assisted decoding: Instruction length

- *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 ...`

`mov [edi+0xcafa1053],dh`

(valid instruction, six bytes long)



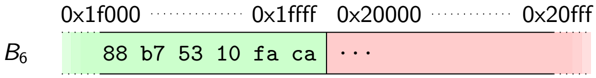
CPU-assisted decoding: Instruction length

- *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 ...`

`mov [edi+0xcafa1053],dh`

(valid instruction, six bytes long)



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

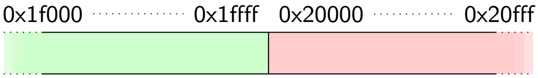
$\downarrow$

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

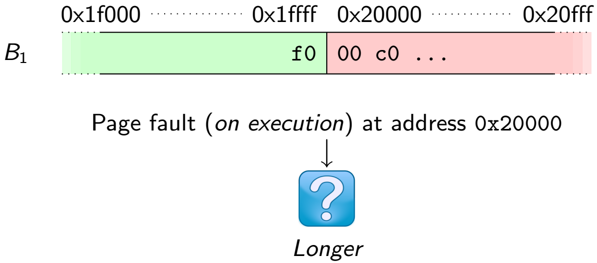
$B =$ `f0 00 c0 ...`
(invalid)



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

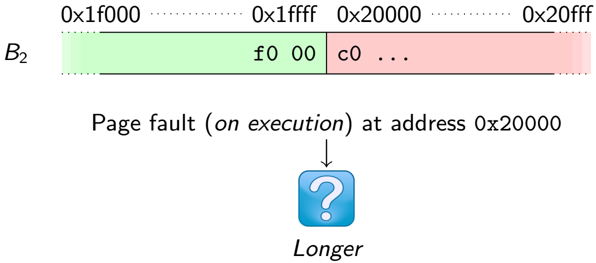
$B =$ `f0 00 c0 ...`
(invalid)



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

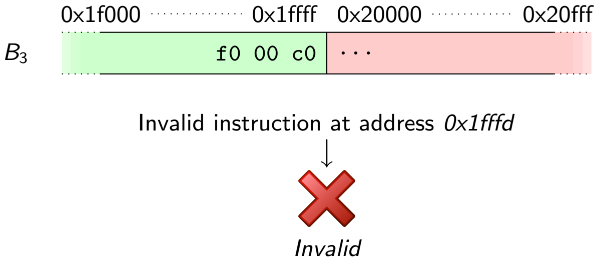
$B =$ `f0 00 c0 ...`
(invalid)



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

$B =$ `f0 00 c0 ...`
(invalid)



CPU-assisted decoding: Non-implicit operands

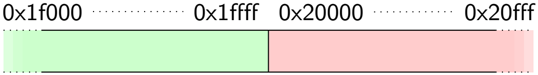
- *Idea*: change the bytes that follow the opcode, and observe how the CPU behaves
- The instruction will be invalid if we replace an operand with another one of a different type

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

$B =$ `88 b7 53 10 fa ca`

`mov [edi+0xcafa1053],dh`

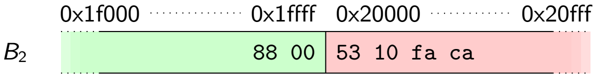


CPU-assisted decoding: Non-implicit operands

- *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`

`mov [edi+0xcafa1053],dh`



`mov [eax], al`

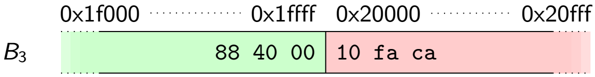
Page fault (*on write*) at address `0x00` → *Valid*

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`

`mov [edi+0xcafa1053],dh`



`mov [eax+0x0], al`

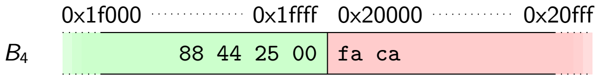
Page fault (*on write*) at address `0x00` → *Valid*

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`

`mov [edi+0xcafa1053],dh`



`mov [ebp+0x0], al`

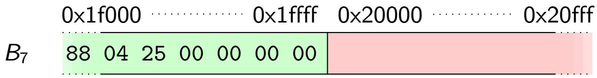
Page fault (*on write*) at address `0x00` $\rightarrow$ *Valid*

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

$B =$ `88 b7 53 10 fa ca`

`mov [edi+0xcafa1053],dh`



`mov [0x0], al`

Page fault (*on write*) at address `0x00` $\rightarrow$ *Valid*

Test passed

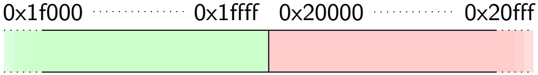
Operand is an addressing-form specifier

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`

```
add eax,0x78563412
```

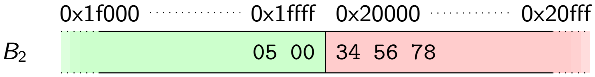


CPU-assisted decoding: Non-implicit operands

- *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`



Page fault (*on execution*) at address `0x20000` $\rightarrow$ *Longer*

Test failed

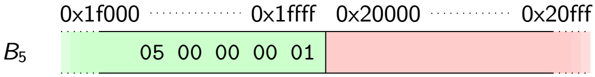
Operand is **not** an addressing-form specifier

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`

`add eax,0x78563412`



`add eax, 0x1000000`

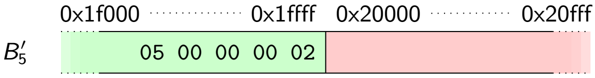
No exception → *Valid*

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`



`add eax, 0x2000000`

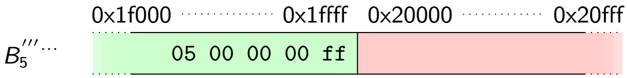
No exception → *Valid*

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`



`add eax, 0xff000000`

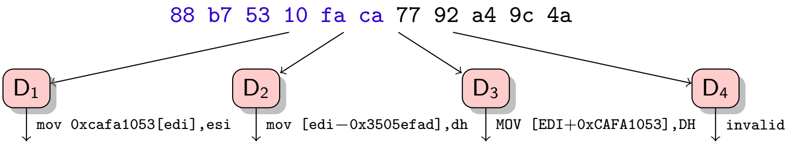
No exception → *Valid*

Test passed

Operand is a 32-bit immediate

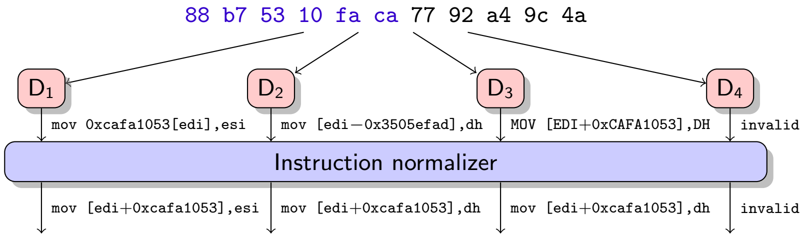
Comparing the output of disassemblers

- The outputs of disassemblers differ for many subtle details



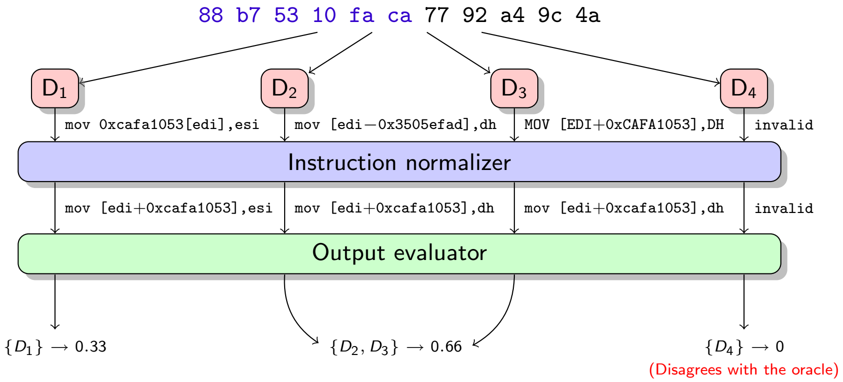
Comparing the output of disassemblers

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



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



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

CPU	**Supported features**				
	MMX	**SSE**	**SSE2**	**SSE3**	**SSE4**
Intel P3 (1.2GHz)	✓	✓			
Intel P4 (3.0GHz)	✓	✓	✓		
Intel Core2 (2.0GHz)	✓	✓	✓	✓	
Intel Xeon (2.8GHz)	✓	✓	✓	✓	

Evaluation of off-the-shelf disassemblers

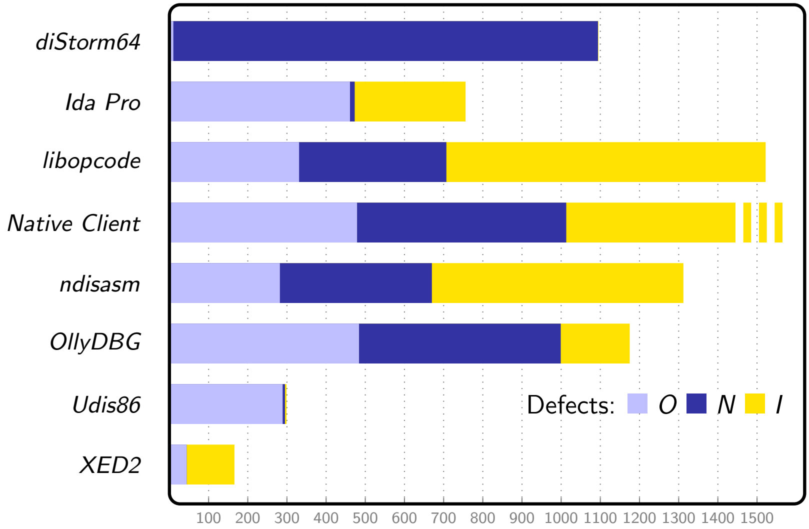
Setup

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

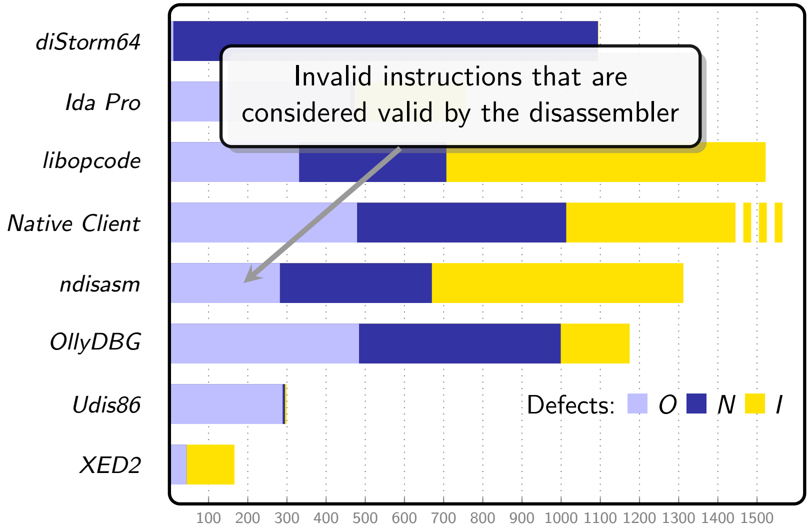
Test-cases

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

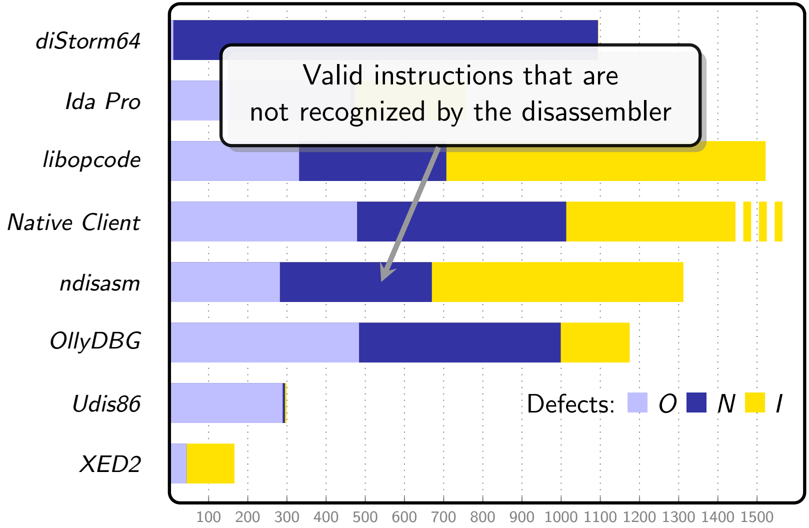
Evaluation of off-the-shelf disassemblers



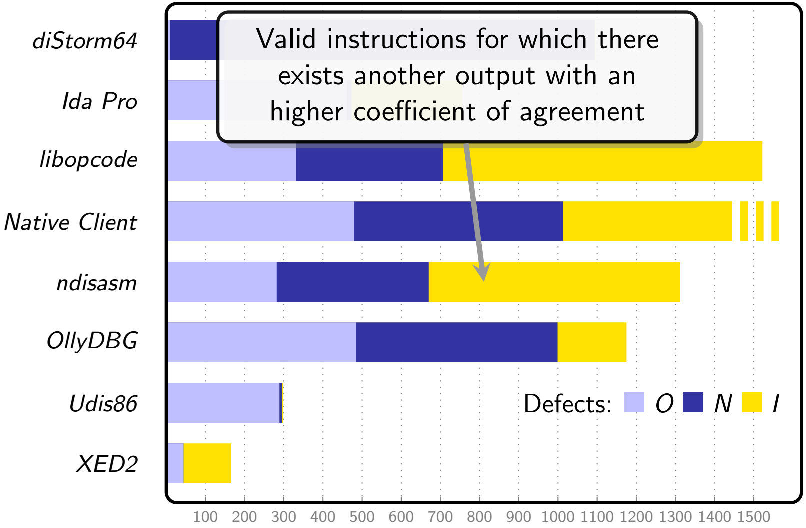
Evaluation of off-the-shelf disassemblers



Evaluation of off-the-shelf disassemblers



Evaluation of off-the-shelf disassemblers



Some of the defects we found

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

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`