CS 250 Software Security

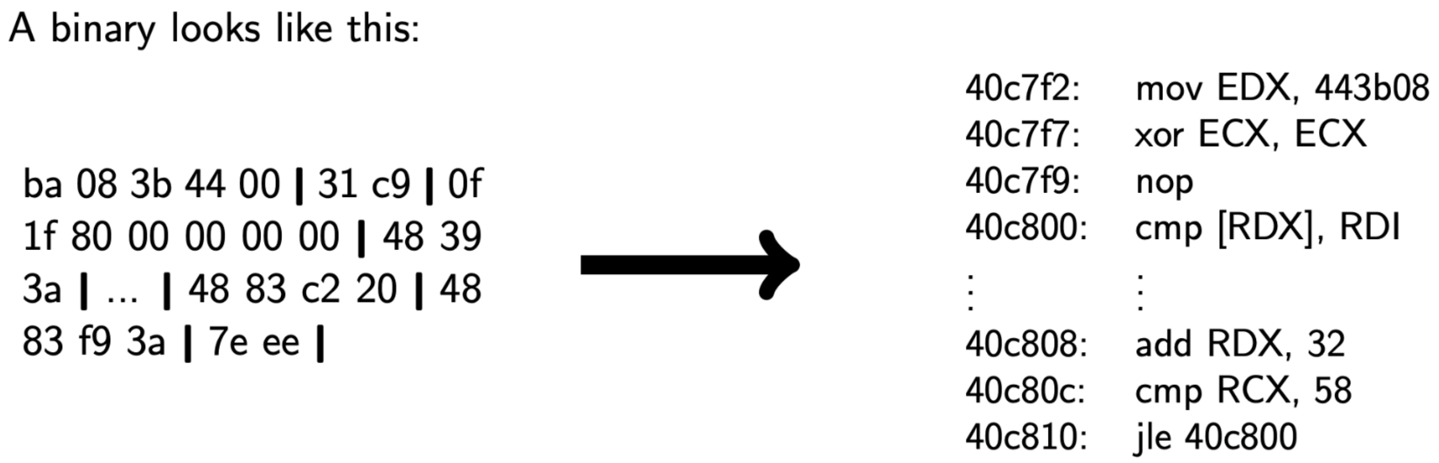
Reassemblable Assembly

Reassemblable Assembly

- Assembly code with cross references

- Two Tasks
  - Instruction Boundary Identification
  - Symbolization

Instruction Boundary Identification



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 |
```

$\longrightarrow$

```
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
```

- Find which addresses correspond to which instructions
- Challenges
  - X86 instructions have variable sizes
  - Data interleaves with instructions

Symbolization

Symbolization: Which numbers are references vs. literals

```
40c7f2:   mov EDX, 443b08                              mov EDX, Label_1
40c7f7:   xor ECX,ECX                                  xor ECX,ECX
40c7f9:   nop                                          nop
40c800:   cmp [RDX],RDI                      Label_2:  cmp [RDX],RDI
  ⋮          ⋮                                            ⋮
40c808:   add RDX,32                                   add RDX,32
40c80c:   cmp RCX,58              ⟶                    cmp RCX,58
40c810:   jle 40c800                                   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                      .quad Label_3
443b50:   85 48 44 00 00 00 00 00                      .quad Label_4
```

What are possible solutions?

- Dynamic Analysis
  - Observe what instructions are executed, and what numbers are used as references
  - Sound, but not complete
- Static Analysis
  - Statically extract some patterns
  - Unsound, potentially more complete than dynamic analysis
- Combining dynamic and static analysis
  - Facts observed in dynamic analysis are trustworthy

Datalog Disassembly

- Published in USENIX Security 2020

- A logic inference approach
  - Collect facts statically
  - Encode expert knowledge as logic rules
  - Perform logic inference

What is logic programming?

- A method to perform optimized search over a finite domain.
- Reduce search space using pre-defined logic rules.

```
father_child(tom, sally).

father_child(tom, erica).

father_child(mike, tom).
```

Facts form the problem domain

```
sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).

parent_child(X, Y) :- mother_child(X, Y).
```

Logic rules describe the search problem

Logic programming - example

```
father_child(tom, sally).

father_child(tom, erica).

father_child(mike, tom).

sibling(X, Y)      :-

     parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).

parent_child(X, Y) :- mother_child(X, Y).
```

```
father_child(mike, tom)?

True
```

```
parent_child(X, Y)

X = tom, Y = sally

X = tom, Y = erica ...
```

Infer Instruction Boundary

- Decode every possible offset in code sections

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)

Infer Instruction Boundary

- Analyses and heuristics are expressed as Datalog rules:

```
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 fall through, 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.

Infer Instruction Boundary

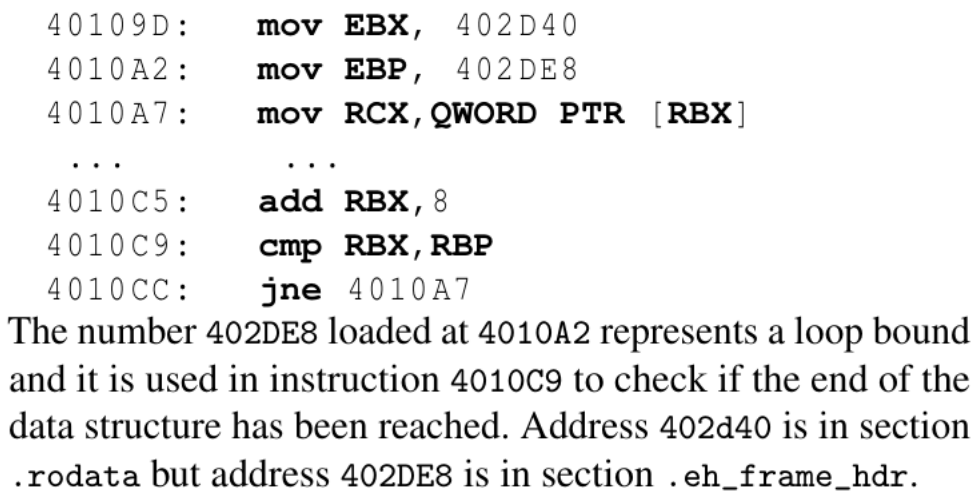
- Backward traversal: propagate invalid instructions
- Forward traversal: build superset of all possible basic blocks

- Assign points to candidate blocks using heuristics
  - Entry point: +20
  - Address appears in data section: +1
  - Direct jump: +6
  - …
- Aggregate points to resolve overlaps (Datalog extension)

Symbolization

- Naïve Approach
  - Numbers in the binary address range →Symbols
  - Numbers outside the range → Literals

  - False positives: A literal coincides with the binary address range
  - False negatives: symbol+constant



```
40109D:    mov EBX, 402D40
4010A2:    mov EBP, 402DE8
4010A7:    mov RCX,QWORD PTR [RBX]
  ...         ...
4010C5:    add RBX,8
4010C9:    cmp RBX,RBP
4010CC:    jne 4010A7
```

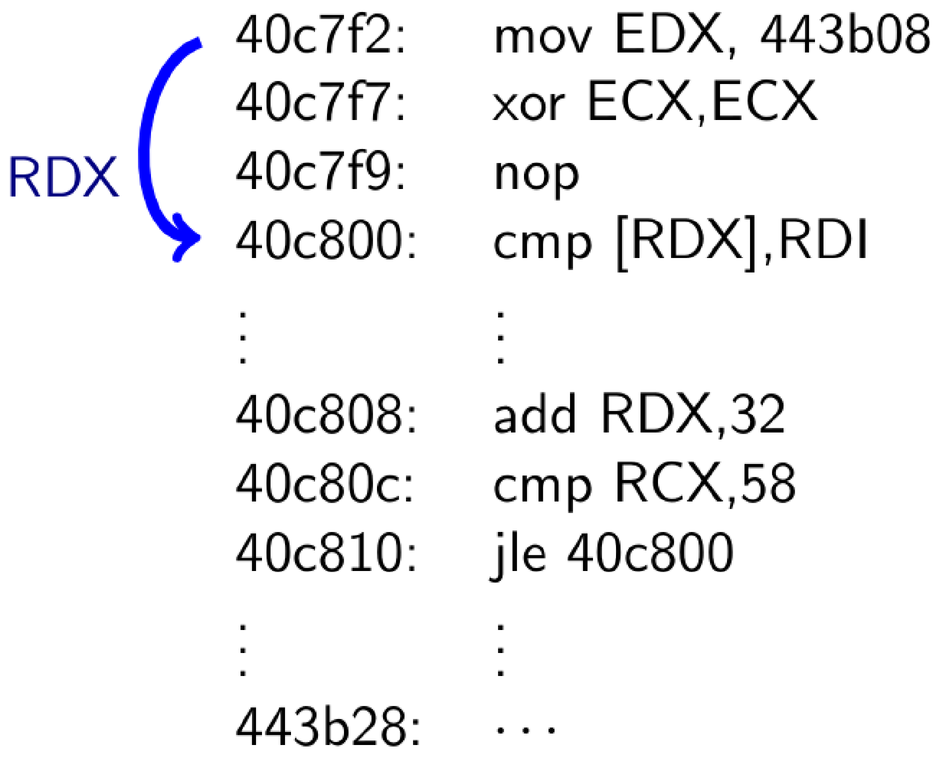
The number 402DE8 loaded at 4010A2 represents a loop bound and it is used in instruction 4010C9 to check if the end of the data structure has been reached. Address 402d40 is in section .rodata but address 402DE8 is in section .eh_frame_hdr.

Symbolization: Reducing False Positives

- Collect additional evidence (how the number is used) using supporting analyses and heuristics
- Assign points to candidates
  - Symbol
  - Symbol-Symbol
  - Strings
  - Other
- Aggregate points to make a decision

Supporting Analysis: Def-Uses

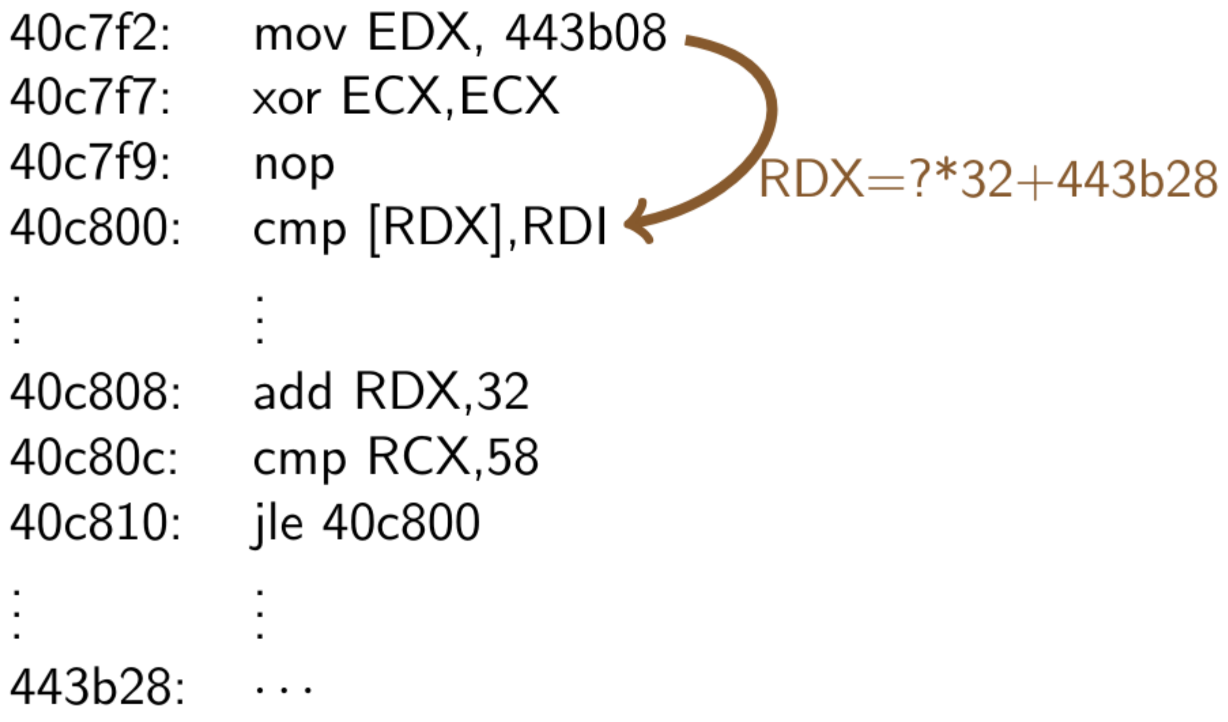
- Predicate: def_used(Adef, Reg, Ause)
  Register Reg is defined in Adef and used in Ause



```
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:   ...
```

Supporting Analysis: Register Value Analysis

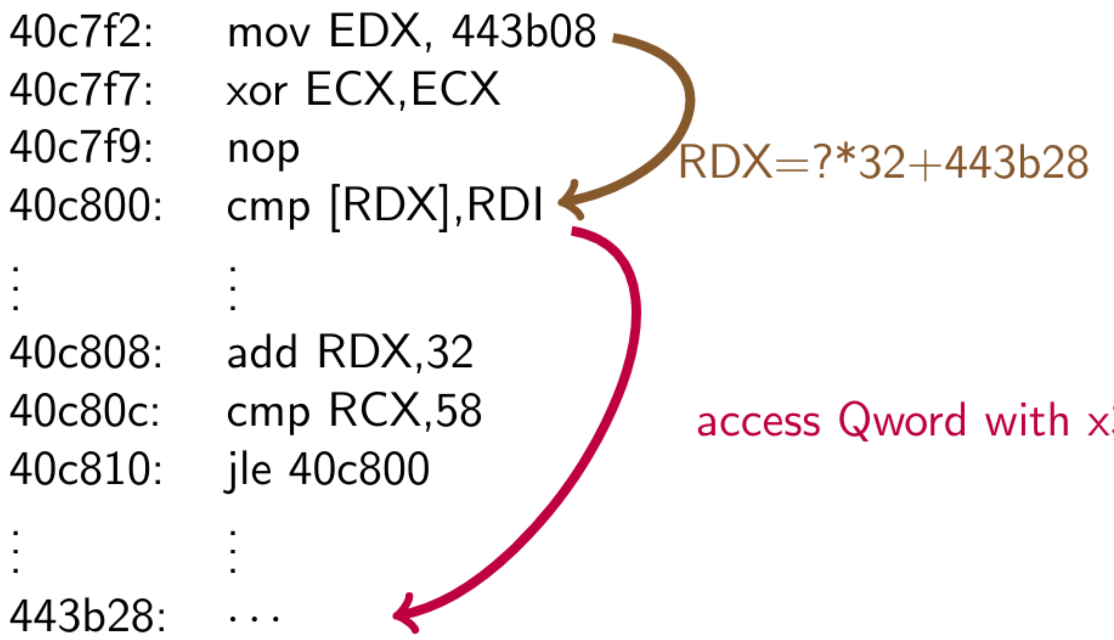
- Predicate: reg_val(A1, Reg1, A2, Reg2, Mult, Disp)
  value(Reg1@A1) = value(Reg2@A2) * Mult + Disp



```
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:   ...
```

Supporting Analysis: 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:    ...
```

Assigning Points

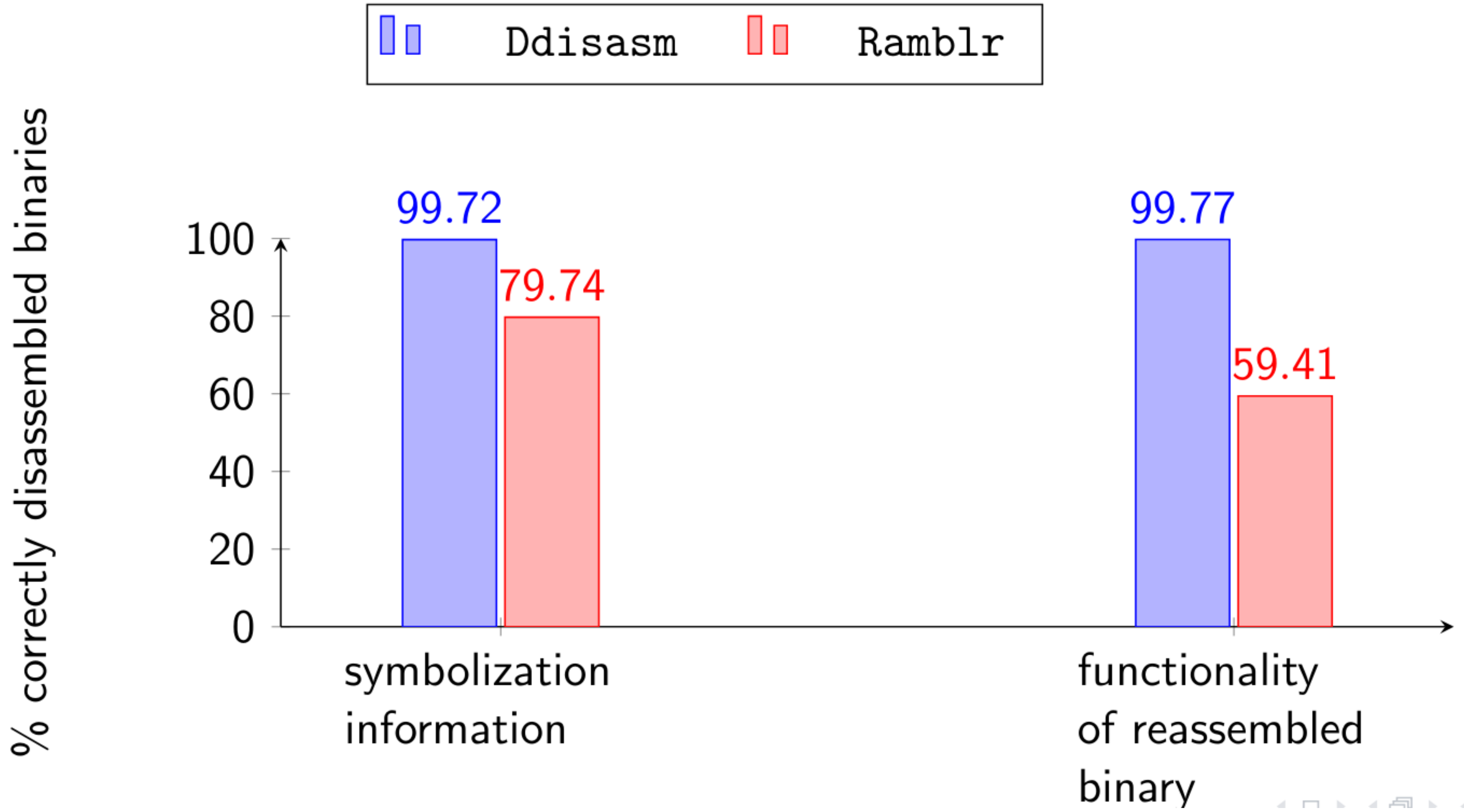
- Use supporting analyses to enhance confidence

- Candidates in data section
  - + Pointer to instruction beginning: A symbol candidate points to the beginning of an instruction.
  - + Data access match: The data object candidate is accessed from the code with the right size.
  - + Symbol arrays: There are several contiguous or evenly spaced symbol candidates.
  - + Pointed by symbol array: Multiple candidates of the same type pointed by a single array
  - - Data access conflict: There is some data access in the middle of a symbol candidate
  - …

Experimental Evaluation

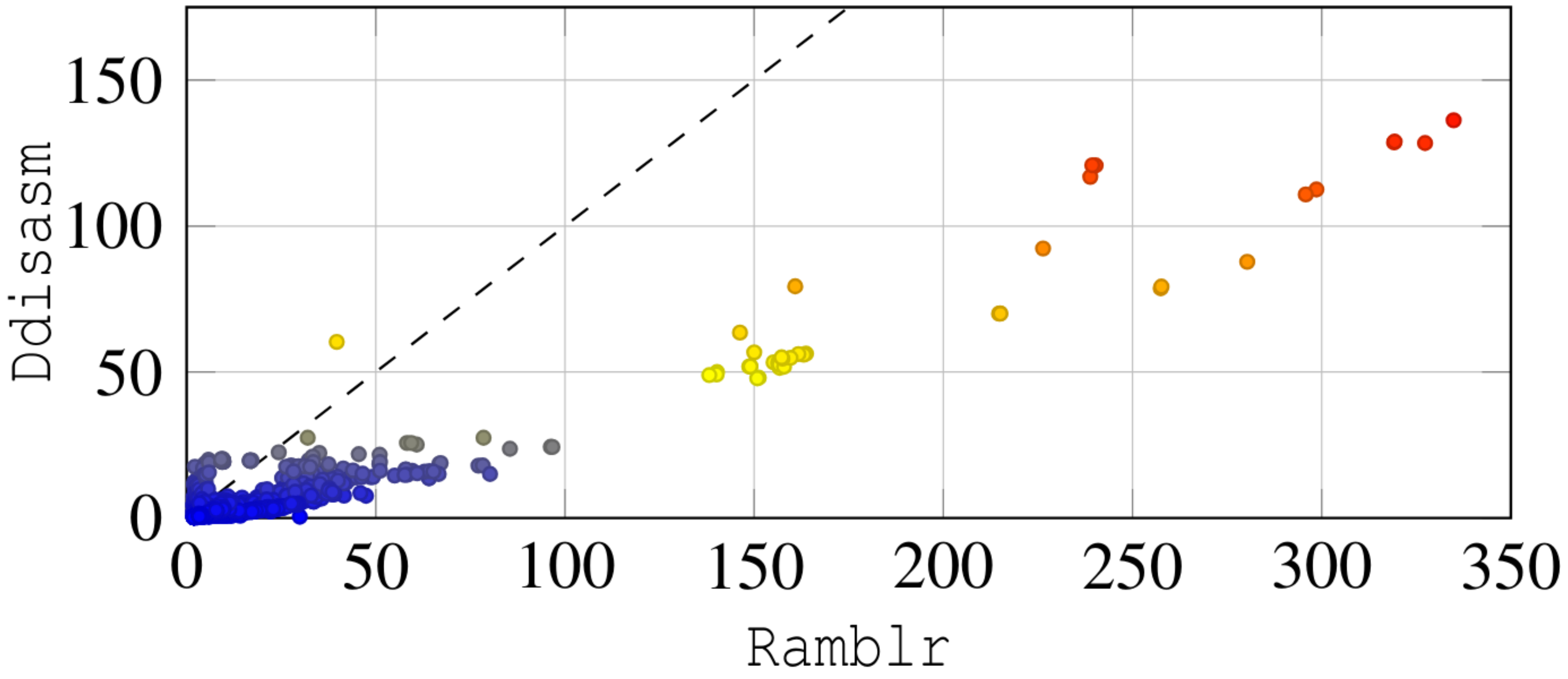
- Ddisasm supports x64 Linux ELF binaries
- Ddisasm is tested with
  - 3 benchmark sets
  - 7 compiler versions (GCC, Clang, and ICC)
  - 6 compiler optimization flags
- A total of 7658 binaries
- Compared to Rambler (another tool based on Angr)

Accuracy



Ddisasm
Ramblr
% correctly disassembled binaries
100
80
60
40
20
0
99.72
79.74
99.77
59.41
symbolization information
functionality of reassembled binary

Disassembly Time in Seconds



Discussion

- Reassembleable disassembly is undecidable in principle
- Practical solutions are still possible
- Other hints to leverage: metadata for relocation, position-independent code, etc.
- https://github.com/SystemSecurityStorm/Awesome-Binary-Rewriting
- Datalog Disassembly demonstrates a **symbolic** approach
- What about neural approach, or neural-symbolic approach?

Lab 3: CFI and Shadow Stack

- Binary Rewriting: use Datalog Disassembly
  - https://github.com/GrammaTech/ddisasm
- Use its docker image
  - docker pull grammatech/ddisasm:latest
  - docker run -v "`pwd`":/shared -it grammatech/ddisasm bash
  - cd /shared
  - ddisasm CADET_00001 --asm CADET.s
  - as CADET.s -o CADET.out
  - ld CADET.out -e _start -o CADET_00001_rewritten

Lab3: CFI Implementation (protecting ret)

```
leaq .L_4c25a8(%rip),%rsi
movl $2,%edi
subq $8,%rsp
xorl %eax,%eax
callq _dl_dprintf
pretetchnta 0x11223344
movl $127,%edi
```

```
40121a:     48 8d 35 87 23 0c 00    lea    0xc2387(%rip),%rsi
401221:     bf 02 00 00 00          mov    $0x2,%edi
401226:     48 83 ec 08             sub    $0x8,%rsp
40122a:     31 c0                   xor    %eax,%eax
40122c:     e8 35 93 08 00          callq  48a566 <_dl_dprintf>
401231:     0f 18 04 25 44 33 22    prefetchnta 0x11223344
401238:     11
401239:     bf 7f 00 00 00          mov    $0x7f,%edi
```

```
cfi_check_ret:
    push %rax
    movq 8(%rsp), %rax
    cmpl $0x11223344, 4(%rax)
    jne .L_failed_ret
    pop %rax
    retq
.L_failed_ret:
    call exit
```

Then I can validate return by replacing retq with jmp_cfi_check_ret

```
            seta %dl
            sbbb $0,%dl
            testb %dl,%dl
            jne .L_400558
.L_400557:
            jmp cfi_check_ret
```

Lab 3: You Job

- Protecting indirect calls with CFI
  - A simple policy is fine: an indirect call can jump to any function entry

- Protecting returns with shadow stack
  - A simple implementation is fine: pre-allocate a large buffer
  - No support for multi-threading is fine
  - No need to protect the shadow stack