NeurIPS 2019

Coda: An End-to-End Neural Program Decompiler

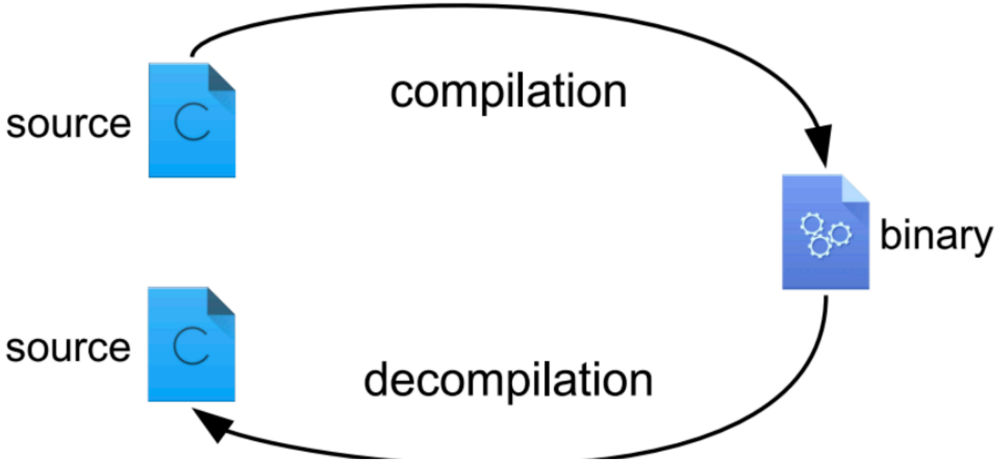
Cheng Fu[1], Huili Chen[1], Haolan Liu[1], Xinyun Chen[3], Yuandong Tian[2], Farinaz Koushanfar[1], Jishen Zhao[1],

[1]*UC San Diego*, [2]*Facebook AI Research*, [3]*UC Berkeley*

Coda is the abbreviation for CodeAttack

Background: Decompilation

- ***Goal of Decompilation:***
  - Binary executables to High-level programming language
- ***Decompilation for SW defense***:
  - Malware analysis
  - Vulnerability detection and fixing
  - Binary comparison/verification
- ***Decompilation for SW attacking***:
  - Reverse engineering software binary with copyright protection for illegal usage.



Challenges

- Prior decompilers (e.g. Hex-rey [1], RetDec [2]…) focus on reverse
  engineering the ***functionality*** of binary executables:
  - Semantics not guaranteed

Source Code

```
int a = atoi(argv[1]);;
int b = atoi(argv[2]);;
int c = atoi(argv[3]);;
a = b * c - 1 ;
if( a > 1 ){
    a = b + c;
    c = a * c - b;
```

RetDec →

Decompiled Code

```
int32_t v1 = (int32_t)argv;
atoi((char*)*(int32_t*)(v1 + 4));
int32_t v2 = *(int32_t*)(v1 + 8);
int32_t v3 = *(int32_t*)(v1 +12);
int32_t result;
if (v3 * v2 >= 3) {
  result = (v3 + v2) * v3 - v2;
} else {
  result = v3 * v2 < 3;
}
return result;
```

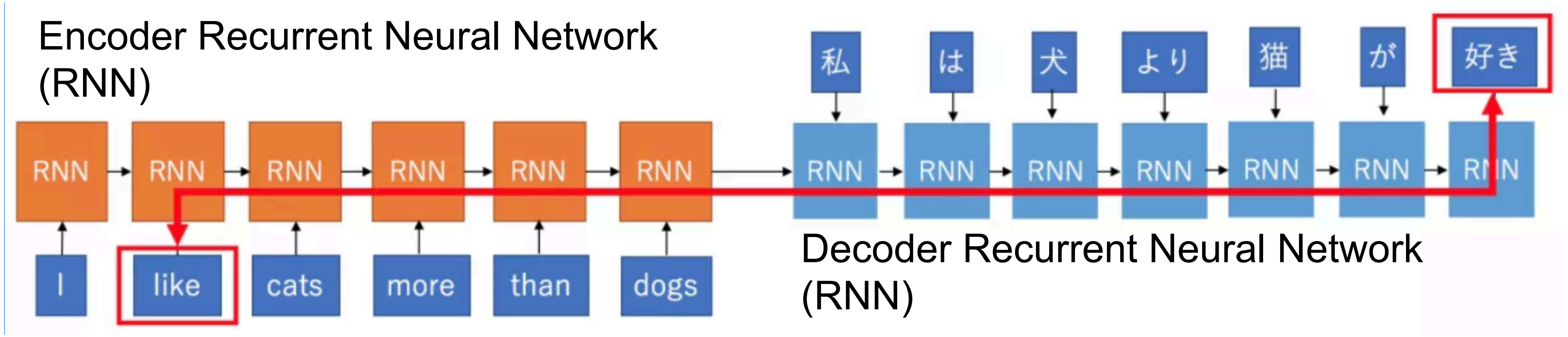
[1] HexRey, 2018, https://www.hex-rays.com/products/decompiler/
[2] RetDec, 2019, https://retdec.com/

Challenges

- Many hardware architectures (ISA): x86, MIPS, ARM
- Many Programming Languages (PL)
  - Extra Human effort to extend to the new version of the hardware architectures or programming languages
- Many formats of binary files
  - .efl, .bin, .exe …

Decompilation as Machine translation

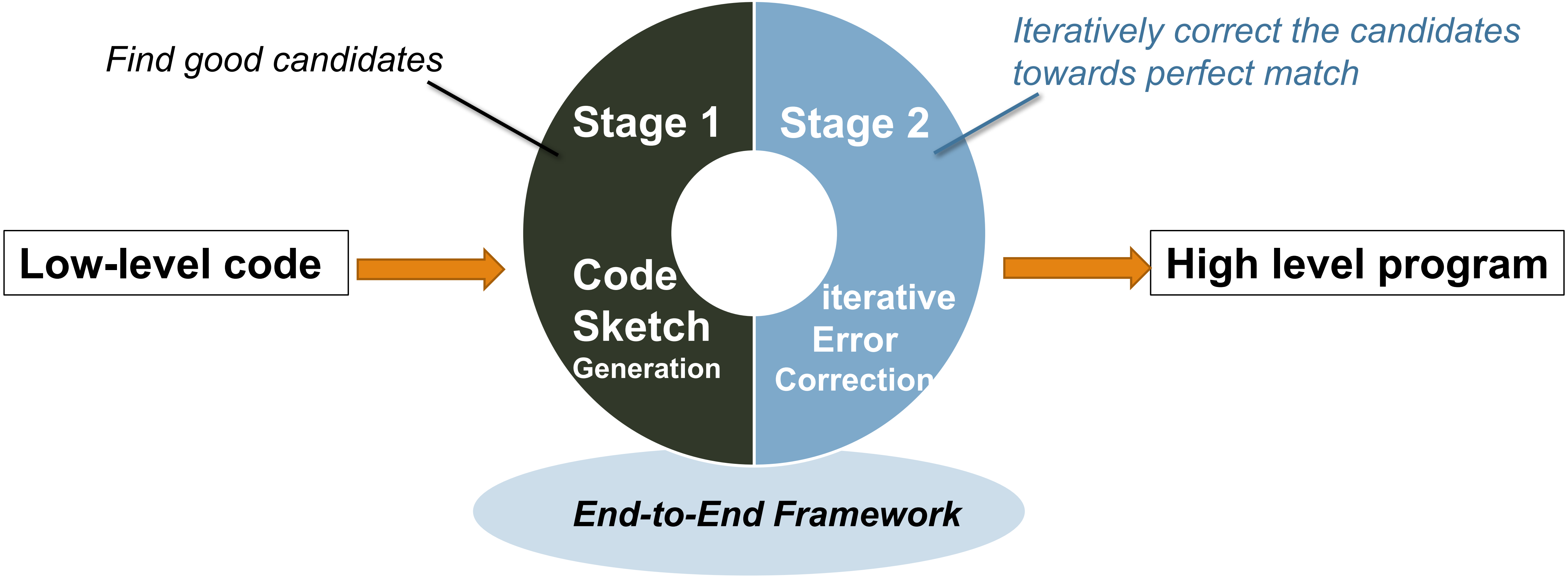
- Intuitively, decompilation is translation problem and can be solved using auto-encoder for machine translation:



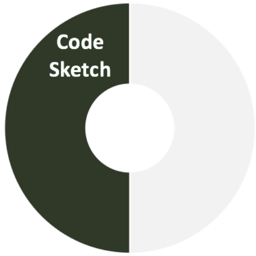
- However, a naïve ***sequence-to-sequence*** model is hard to capture the meaning of low-level code and learn the grammar of high-level PL.

Coda Design

Leverage both syntax and dynamic information



Stage 1: Code Sketch Generation



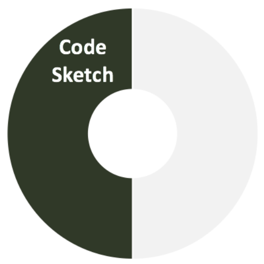
- What should the encoder captures?
  - inter and intra instruction dependencies
- Instruction-aware encoder :
  - Coda leverages **N-ary Tree Encoder [3]** to capture inter and intra dependencies of the low-level code.
    - Opcode and its operands are encoded together.
    - Preserve the order of operands
  - Different encoders are used for encoding different instruction types, namely, **memory** (mem), **branch** (br) and **arithmetic** (art).

```
# source C program
 a = b * c ;
 if( a > c ){
   c = a * c - b;
 }
```

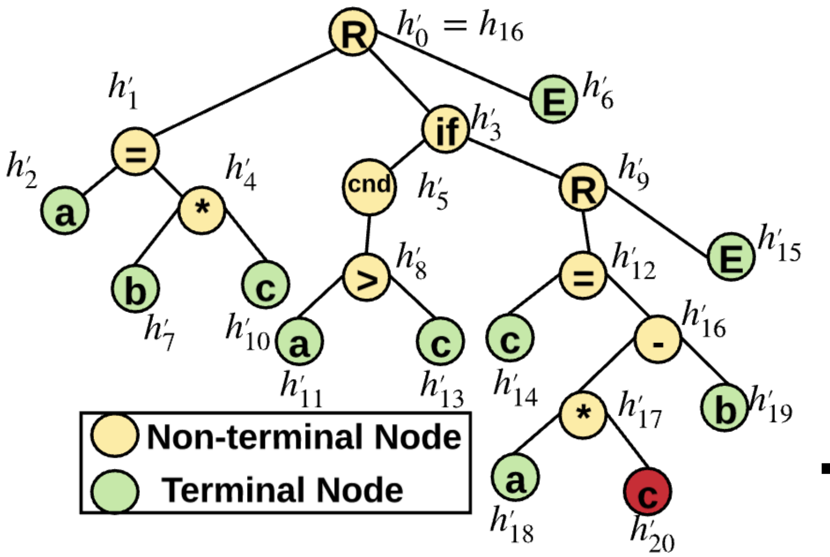
0	lw	$1, 24($fp)	mem h0
1	lw	$2, 20($fp)	mem h1
2	mul	$1, $1, $2	art h2
3	sw	$1, 28($fp)	mem h3
4	lw	$1, 28($fp)	mem h4
5	lw	$2, 20($fp)	mem h5
6	slt	$1, $2, $1	art h6
7	beqz	$1, $BB0_3	br h7
8	j	$B2	br h8
9	$B2:		br h9
10	lw	$1, 28($fp)	mem h10
11	lw	$2, 20($fp)	mem h11
12	mul	$1, $1, $2	art h12
13	lw	$2, 24($fp)	mem h13
14	subu	$1, $1, $2	art h14
15	j	$B3	br h15
16	sw	$1, 20($fp)	mem h16

[3] Tai, Kai Sheng, Richard Socher, and Christopher D. Manning. "Improved semantic representations from tree-structured long short-term memory networks." *arXiv preprint arXiv:1503.00075* (2015).

Stage 1: Code Sketch Generation



- Tree decoder for Abstract Syntax Tree (AST) generation:
  - AST can be equivalently translated into its corresponding high level Program
  - Advantages:
    - Prevent error propagation/ Preserve node dependency / easy to capture PL grammar
    - Boundaries are more explicit (terminal nodes)
- Parent and input attention feeding mechanism



$$s_k^{20} = \frac{\exp\{h_k^T \cdot h'_{20}\}}{\sum_{j=0}^{16} \exp\{h_j^T \cdot h'_{20}\}}$$

Attention Probability

$$c_{20} = \sum_{k=0}^{16} h_k \cdot s_k^{20}$$

Attention expectation

$$e_{20} = \mathbf{tanh}(W_1 c_t + W_2 h'_{20})$$

Attention Vec

$$t_{20} = \mathbf{argmax}\ softmax(We_{20})$$

Prediction

Stage 2: Iterative Error Correction

- The sketch generated in **<u>Stage 1</u>** may contain errors:
  - Mispredicted tokens, missing lines, redundant lines

Golden program
```
If( a > c ) {
        a = b + c * a;
        b = a – c ;
}
```

Mispredicted
```
If( a > b ) {
        a = b + c * a;
        b = a - b;
}
```

Missing lines
```
If( a > c ) {
        a = b + c * a;
}
```

Redundant lines
```
If( a > c ) {
        a = b + c * a;
        b = a;
        b = a;
}
```

- Dynamic information that can be leveraged:
  - I/O pair to identify the correctness of the functionality
  - **Recompile the program back into low-level code (ϕ')---> should match with the golden low-level input (ϕ).**

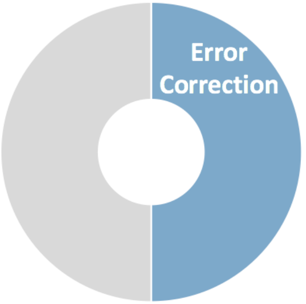
Stage 2: Iterative Error Correction

$(\phi, \phi') \rightarrow$ **Error Predictor** $\rightarrow (Err\ Type\ , Err\ location)$

- Correct the error using an ***<u>Error Correction machine</u>*** (EC machine) guided by ***<u>the Error Predictor (EP)</u>***.
- Optimization techniques:
  - Prevent the false alarm by recompile the updated sketch code and check its ***<u>Levenshtein edit loss</u>*** from the golden input.
  - ***<u>Ensemble</u>*** multiple error predictors to cover more potential errors for updates.

	G A C G	G A C G	G A C G
	G C G	G G C G	G A C G
ED	1	1	0

Stage 2: Iterative Error Correction



Algorithm 1 Workflow of iterative EC Machine.

INPUT: N_{EP} **Ensembled Error Predictors** EP**; Source assembly** ϕ**; Decompiled Sketch program** P'**; Compiler** Γ**; Maximum iterations** S_{max} **and steps in each iteration** c_{max}**;**

OUTPUT: Error corrected program P'_f**.**

$s_i \leftarrow 0$
while $s_i < S_{max}$ *do*
    $Q \leftarrow [],\ \phi' = \Gamma(P'),\ \ \Delta' \leftarrow Edit\_loss(\phi, \Gamma(P'))$
    if $\Delta' = 0$ **then break**
    $Q \leftarrow EP_i(P')$ for $i = 1, \ldots, N_{EP}$ // Attach all the detected error to queue Q
    $\widetilde{Q} \leftarrow Prob\_sort(Q, c_{max})$ // Rank Q using output probabilities, keep c_{max} results.
    while $\widetilde{Q}$ is not empty **do**
        $err, node \leftarrow \widetilde{Q}.pop()$
        $P'_t \leftarrow FSM\_Error\_Correct(P', err, node)$ // correct the error in the program
        $\Delta = \Delta' - Edit\_loss(\phi, \Gamma(P'_t))$
        if $\Delta \geq 0$ **then**
            $P' \leftarrow P'_t$
Return: $P'_f \leftarrow P'$

- Iterative updating workflow

Experimental Setup

- Compiler configuration : ***clang –O0*** (disabled optimization)
- Benchmarks:
  - ***Synthetic programs***:
    - ***Karel library (Karel)*** – only function calls (control graph)
    - ***Math library (Math)*** – function calls with arguments (Data dependency + control graph)
    - ***Normal expressions (NE)*** – (^,&,*,-,<<,>>,|,% ….) (Data dependency + control graph)
    - ***Math library + Normal expressions (Math + NE)*** – replaces the variables in NE with a return value of math function. (Data dependency + control graph)
  - ***Metrics:***
    - ***Token Accuracy : The percentage of predicted tokens that match with the ground-truth ones.***
    - ***Program Accuracy: The percentage of programs that yields 100% Token accuracy.***

Results – Stage 1 Performance

- ***Token accuracy*** across benchmarks

Benchmarks	Seq2Seq	Seq2Seq+Attn	Seq2AST+Attn	Inst2seq+Attn	Inst2AST+Attn
Karel_S	51.61	97.13	99.81	98.83	**99.89**
Math_S	23.12	94.85	99.12	96.20	**99.72**
NE_S	18.72	87.36	90.45	88.48	**94.66**
$(\text{Math+NE})_S$	14.14	87.86	91.98	89.67	**97.90**
Karel_L	33.54	94.42	98.02	98.12	**98.56**
Math_L	11.32	91.94	96.63	93.16	**98.63**
NE_L	11.02	81.80	85.92	85.97	**91.92**
$(\text{Math+NE})_L$	6.09	81.56	85.32	86.16	**93.20**

Baseline

Ours

X_S short programs, X_L long programs

- Coda yields the highest token accuracy across all benchmarks (**96.8%** on average) compared to all the other methods.
- Coda engenders **10.1%** and **80.9%** margin over a naive **Seq2Seq** model **with** and **without** attention.
- More tolerant to the growth of program length.

Examples

```
#Golden:
TurnOn();
TurnOff();
while(leftIsClear){
PutBeeper();
TurnLeft();
if(notFacingNorth){
PickBeeper();
PickBeeper();
PickBeeper();
continue;}
}
PickBeeper();
}
```

```
#Decompiled
TurnOn();
TurnOff();
while(leftIsClear){
PutBeeper();
TurnLeft();
if(notFacingNorth){
PickBeeper();
PickBeeper();
PickBeeper();
continue;}
}
PickBeeper();
}
```

Karel.h

Math.h

```
TARGET CODE
48 46
d=isgreaterequal(a,d);
g=sin(f);
while(islessgreater(c,f)){
b=sin(b);
e=atan2(a,e);
e=fmax(e,c);
}
b=acos(f);
e=ceil(a);
```

```
TRANSLATE CODE
d=isgreaterequal(a,d);
g=sin(f);
if(islessgreater(c,f)){
b=sin(b);
e=atan2(a,e);
e=fmax(e,e);
}
b=acos(f);
e=ceil(a);
```

err

Examples

```
#golden
if(d<g*f){
a=a*a-d;
b=a/a+f;
e=a/f;
a=a-a-g;
}
else{
d=c*f-g;
d=a/f;
a=c+f+c;
}
a=d*f;
a=a+e;
```

```
#decompile
if(d<g*f){
a=a*a-d;
b=a/a+f;
e=a-f;
a=a-a;
}
else{
d=c*f-g;
d=a/f;
a=c+f+c;
f=g;
}
a=d*f;
a=a+e;
a=a+e;
```

Normal Expressions

```
#golden
b=fmax(l,c)*asin(j);
f=fmax(k,h);
if(isless(k,c)||islessequal(f,g)){
b=tan(f);
a=isless(b,a)*islessgreater(c,f);
a=fdim(i,l)/islessequal(l,k);
}
else{
j=exp(j)+pow(j,h);
a=pow(d,a)/sin(g);
h=islessequal(i,b)/tan(j);
}
```

```
#decompiled
b=fmax(l,c)*asin(j);
f=fmax(k,h);
if(isless(k,c)||islessequal(f,g)){
b=tan(f);
a=atan(b,a)*islessgreater(c,f);
a=fdim(i,l);
}
else{
j=exp(j)+isgreaterequal(j,g);
a=pow(d,a)/sin(g);
h=islessequal(i,b)/tan(j);
}
```

Math.h + Normal Expressions

Results – Stage 2 Performance

- **Part (i):** By ensemble 10 EP, Coda achieves **90.8%** error detection rate.
- **Part (ii):** Coda's EC machine increases the ***program accuracy*** from ***30% to 82%*** on average for Inst2AST-based code sketch generation, respectively.

BenchMarks	(i) Error Detection		(ii) Befor EC		After EC	
	$s2s$,10	$i2a$,10	$s2s$	$i2a$	$s2s$	$i2a$
Math_S	91.4	94.2	40.1	64.8	91.2	**100.0**
NE_S	83.5	88.7	6.6	12.2	53.0	**78.6**
$(\text{Math+NE})_S$	83.6	90.1	3.5	43.2	63.6	**89.2**
Math_L	87.5	91.3	21.7	51.8	83.9	**99.5**
NE_L	78.1	84.5	0.2	2.6	33.1	**56.4**
$(\text{Math+NE})_L$	80.2	85.3	0.1	4.9	38.3	**67.2**

Baseline

Coda

s2s = sequence-to-sequence with attention

I2a = instruction encoder to AST decoder with attention

Results -- Overall

- **Coda vs. traditional decompiler (RetDec)**
  - Lines of code: ~10K vs. ~500K -- **50x** reduction
  - Toolkit size: ~10MB Neural network size vs. ~5GB toolkit size -- **500x** reduction
  - Program accuracy: **82%** vs. no semantics guarantee

Discussion

- **Extremely long programs**
  - LSMT is not good at remembering long sequence.
  - Unlike nature language, low-level PL does not have explicit breakup position.
- **Sensitive to ISA**
- **Complicated data type / structure / class ….**
- **Compiler Optimizations / Obfuscation …**

Summary of Coda

- The first neural-based decompilation framework, which preserves both the ***semantics*** and the ***functionality*** of the high-level program
- Decomposes the decompilation task into of two key phases -- ***code sketch generation*** and ***iterative errors correction***
- Significantly outperforms the Seq2Seq model and traditional decompilers

Coda: An End-to-End Neural Program Decompiler

Cheng Fu[1], Huili Chen[1], Haolan Liu[1], Xinyun Chen[3], Yuandong Tian[2], Farinaz Koushanfar[1], Jishen Zhao[1],

[1]UC San Diego, [2]Facebook AI Research, [3]UC Berkeley