Coda: An End-to-End Neural Program Decompiler

Cheng Fu\textsuperscript{1}, Huili Chen\textsuperscript{1}, Haolan Liu\textsuperscript{1}, Xinyun Chen\textsuperscript{3}, Yuandong Tian\textsuperscript{2}, Farinaz Koushanfar\textsuperscript{1}, Jishen Zhao\textsuperscript{1},

\textsuperscript{1}UC San Diego, \textsuperscript{2}Facebook AI Research, \textsuperscript{3}UC Berkeley

\textit{Coda} is the abbreviation for CodeAttack
Background: Decompileation

- **Goal of Decompileation:**
  - Binary executables to High-level programming language

- **Decompileation for SW defense:**
  - Malware analysis
  - Vulnerability detection and fixing
  - Binary comparison/verification

- **Decompileation 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;
}
```

```
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;
```

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
  - .elf, .bin, .exe …
Intuitively, decompilation is a translation problem and can be solved using an 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

End-to-End Framework

Stage 1
- Code Sketch Generation
- Find good candidates

Stage 2
- Iterative Error Correction
- Iteratively correct the candidates towards perfect match

Low-level code → High level program

- Start with low-level code
- Code Sketch Generation
- Find good candidates
- Iteratively correct the candidates towards perfect match
- End up with a high-level program
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).

---

"Improved semantic representations from tree-structured long short-term memory networks." 
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

\[
e_{20} = \tanh(W_1 c_t + W_2 h_{20}')
\]

Attention Vec

\[
t_{20} = \text{argmax} \ \text{softmax}(W e_{20})
\]

Prediction
Stage 2: Iterative Error Correction

- The sketch generated in **Stage 1** 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;
} 

- Dynamic information that can be leveraged:
  - I/O pair to identify the correctness of the functionality
  - Recompile the program back into low-level code (\( \phi' \))\( \rightarrow \) should match with the golden low-level input (\( \phi \)).
Stage 2: Iterative Error Correction

- Correct the error using an Error Correction machine (EC machine) guided by the Error Predictor (EP).
- Optimization techniques:
  - Prevent the false alarm by recompile the updated sketch code and check its Levenshtein edit loss from the golden input.
  - Ensemble multiple error predictors to cover more potential errors for updates.
Stage 2: Iterative Error Correction

Algorithm 1 Workflow of iterative EC Machine.

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

OUTPUT: Error corrected program $P'_f$.

1: $s_i \leftarrow 0$
2: while $s_i < S_{max}$ do
3:     $Q \leftarrow \emptyset$, $\phi' = \Gamma(P')$, $\Delta' \leftarrow Edit\_loss(\phi', \Gamma(P'))$
4:     if $\Delta' = 0$ then break
5:     $Q \leftarrow EP_i(P')$ for $i = 1, \ldots, N_{EP}$        // Attach all the detected error to queue $Q$
6:     $\tilde{Q} \leftarrow Prob\_sort(Q, c_{max})$   // Rank $Q$ using output probabilities, keep $c_{max}$ results.
7:     while $\tilde{Q}$ is not empty do
8:         err, node $\leftarrow \tilde{Q}.pop()$
9:         $P'_f \leftarrow FSM\_Error\_Correct(P', err, node)$      // correct the error in the program
10:        $\Delta = \Delta' - Edit\_loss(\phi, \Gamma(P'_f))$
11:        if $\Delta \geq 0$ then
12:            $P' \leftarrow P'_f$
13:        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

  - 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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Seq2Seq</th>
<th>Seq2Seq+Attn</th>
<th>Seq2AST+Attn</th>
<th>Inst2seq+Attn</th>
<th>Inst2AST+Attn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Karel_S</td>
<td>51.61</td>
<td>97.13</td>
<td>99.81</td>
<td>98.83</td>
<td><strong>99.89</strong></td>
</tr>
<tr>
<td>Math_S</td>
<td>23.12</td>
<td>94.85</td>
<td>99.12</td>
<td>96.20</td>
<td><strong>99.72</strong></td>
</tr>
<tr>
<td>NE_S</td>
<td>18.72</td>
<td>87.36</td>
<td>90.45</td>
<td>88.48</td>
<td><strong>94.66</strong></td>
</tr>
<tr>
<td>(Math+NE)_S</td>
<td>14.14</td>
<td>87.86</td>
<td>91.98</td>
<td>89.67</td>
<td><strong>97.90</strong></td>
</tr>
<tr>
<td>Karel_L</td>
<td>33.54</td>
<td>94.42</td>
<td>98.02</td>
<td>98.12</td>
<td><strong>98.56</strong></td>
</tr>
<tr>
<td>Math_L</td>
<td>11.32</td>
<td>91.94</td>
<td>96.63</td>
<td>93.16</td>
<td><strong>98.63</strong></td>
</tr>
<tr>
<td>NE_L</td>
<td>11.02</td>
<td>85.92</td>
<td>85.97</td>
<td>85.97</td>
<td><strong>91.92</strong></td>
</tr>
<tr>
<td>(Math+NE)_L</td>
<td>6.09</td>
<td>81.56</td>
<td>85.32</td>
<td>86.16</td>
<td><strong>93.20</strong></td>
</tr>
</tbody>
</table>

X_S short programs, X_L long programs
Examples

Karel.h

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

Normal Expressions

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.

<table>
<thead>
<tr>
<th>BenchMarks</th>
<th>(i) Error Detection</th>
<th>(ii) Before EC</th>
<th>After EC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s2s,10</td>
<td>i2a,10</td>
<td>s2s</td>
</tr>
<tr>
<td>MathS</td>
<td>91.4</td>
<td>94.2</td>
<td>40.1</td>
</tr>
<tr>
<td>NEs</td>
<td>83.5</td>
<td>88.7</td>
<td>6.6</td>
</tr>
<tr>
<td>(Math+NE)S</td>
<td>83.6</td>
<td>90.1</td>
<td>3.5</td>
</tr>
<tr>
<td>MathL</td>
<td>87.5</td>
<td>91.3</td>
<td>21.7</td>
</tr>
<tr>
<td>NE L</td>
<td>78.1</td>
<td>84.5</td>
<td>0.2</td>
</tr>
<tr>
<td>(Math+NE) L</td>
<td>80.2</td>
<td>85.3</td>
<td>0.1</td>
</tr>
</tbody>
</table>

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 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\textsuperscript{1}, Huili Chen\textsuperscript{1}, Haolan Liu\textsuperscript{1}, Xinyun Chen\textsuperscript{3}, Yuandong Tian\textsuperscript{2}, Farinaz Koushanfar\textsuperscript{1}, Jishen Zhao\textsuperscript{1},

\textsuperscript{1}UC San Diego, \textsuperscript{2}Facebook AI Research, \textsuperscript{3}UC Berkeley