Using Recurrent Neural Networks for Decompilation

Deborah S. Katz
Carnegie Mellon University

Jason Ruchti and Eric Schulte
Grammatech, Inc.

SANER 2018

We want better decompilation.

Approach:
We use a model based on recurrent neural networks to translate from binary machine code to source code.

Decompilation is Translating Binary Code to Source Code

```
00 00 09 00 d3 12 00 00 70 33 00 00
00 00 00 00 00 00 09 00 d3 12 00 00
78 33 00 00 00 00 00 00 00 00 09 00
d3 12 00 00
```

```
00000000 00000000 00001001 00000000
00010010 00000000 00000000 01110000
00110011 00000000 00000000 00000000
...
```

Decompilation is Translating Binary Code to Source Code

```
00 00 09 00 d3 12 00 00 70 33 00 00
00 00 00 00 00 00 09 00 d3 12 00 00
78 33 00 00 00 00 00 00 00 00 09 00
d3 12 00 00
```

```
g_return_if_fail(screen_info != NULL);
```

Source Code is More Useful to Humans than Binary

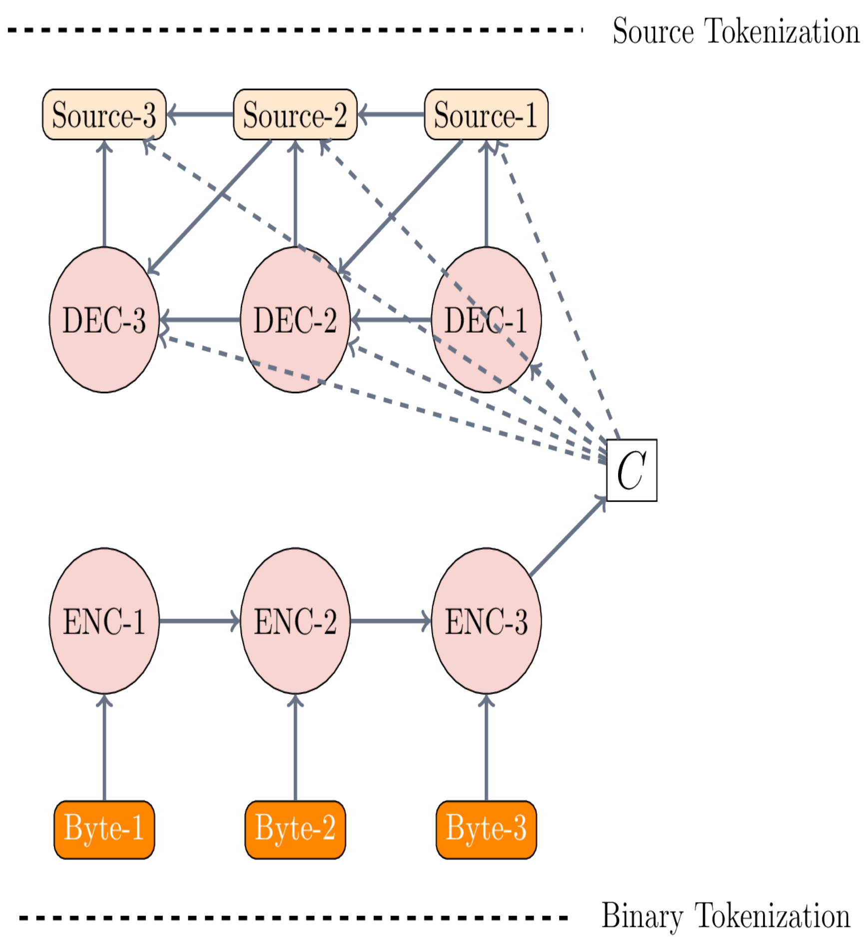
- Human-Readable
- More analysis tools available for source
- Decompilation does not always produce the most useful output
  - Can leave in compiler artifacts, such as:
    - GOTOs
    - Stack pushes for function calls
- Newer techniques rely on compiler details
  - Very specific to individual compilers/languages
- Existing tools are expensive and often unavailable

Decompilation is a Translation Problem

- On some level decompilation is translating:
  - Machine-level binary code to higher-level source or intermediate code
- Look to the techniques for translating other equivalent sequences

Key insight:
To we can translate from **binary** to **source** in the same way we can **translate natural languages**.

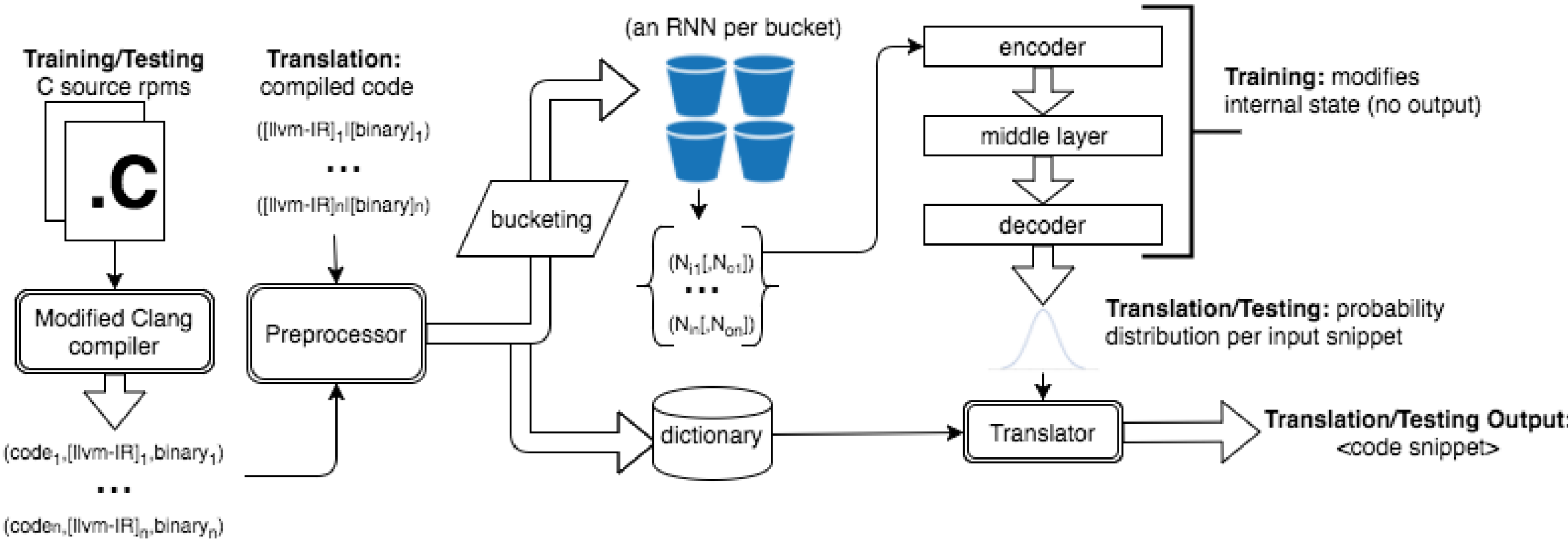
What You Need to Know About Neural Networks (Not Much)



- Encoder-Decoder Model
  - Available off-the shelf with TensorFlow*
  - Designed for translating sequences
- Adapt to translate compiled machine code to higher-level source
- Train model, then use for decompilation

* https://www.tensorflow.org/

Overview



Training/Testing
C source rpms
.C
Modified Clang compiler
(code1,[llvm-IR]1,binary1)
...
(coden,[llvm-IR]n,binaryn)
Translation:
compiled code
([llvm-IR]1|[binary]1)
...
([llvm-IR]n|[binary]n)
Preprocessor
bucketing
(an RNN per bucket)
(Ni1[,No1])
...
(Nin[,Non])
dictionary
encoder
middle layer
decoder
Training: modifies internal state (no output)
Translation/Testing: probability distribution per input snippet
Translator
Translation/Testing Output:
<code snippet>

Creation of Parallel Corpuses

- Used a customized version of Clang to obtain a database of snippets of source code and the equivalent machine/binary code
  - Under certain compiler settings
- We obtained the corpus by compiling many open-source RPM packages
  - 1,151,013 paired snippets of source and binary

The Encoder-Decoder Model Operates on Sequences of Integers

- We train the model on the paired snippets:
  - Machine code and the equivalent source code
  - Each snippet is represented as a sequence of integers
- For example:

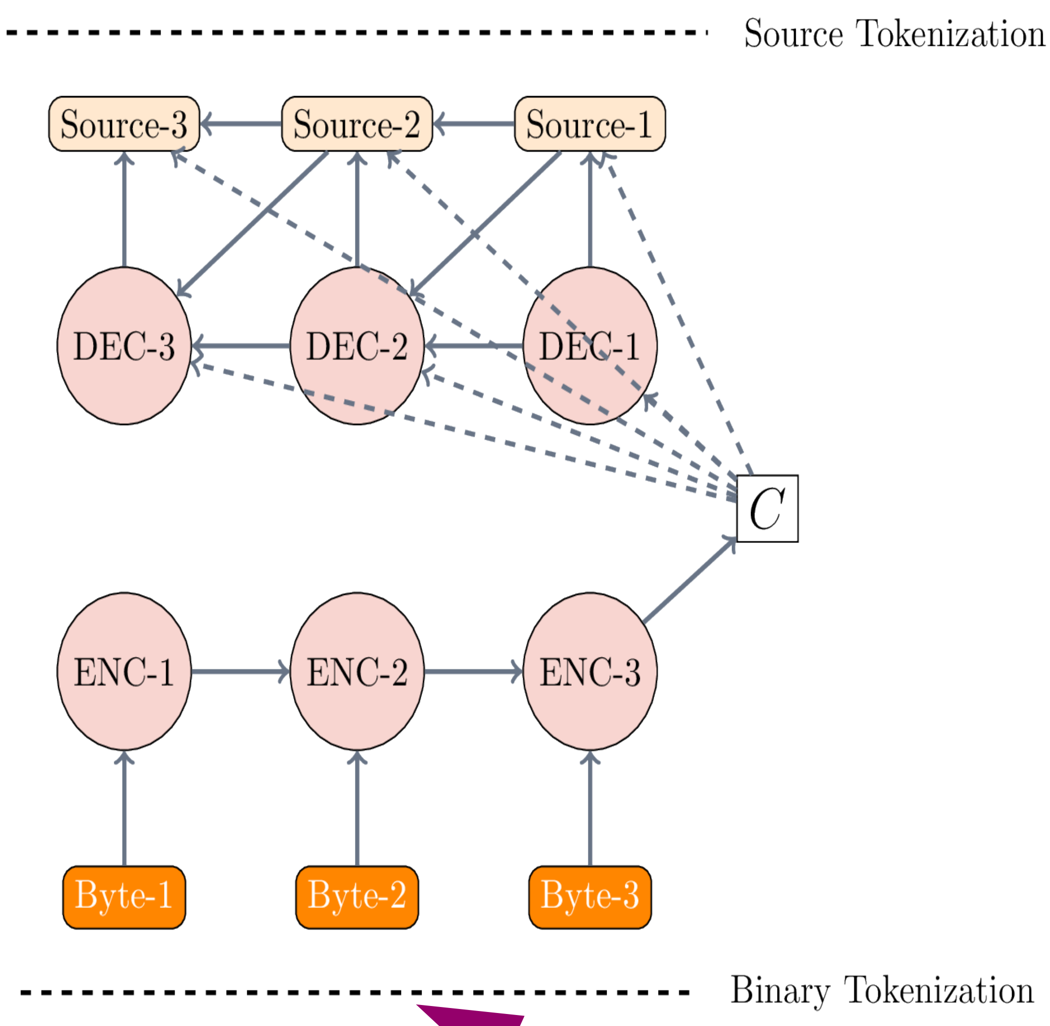
```
Binary: 00 00 09 00 d3 12 00 00 70 33 00 00 00 00 00 00 00 00
09 00 d3 12 00 00 78 33 00 00 00 00 00 00 00 00 09 00 d3 12 00
00

Binary tokenization: 4 4 80 4 198 136 4 4 118 173 4 4 4 4 4 4
4 4 80 4 198 136 4 4 78 173 4 4 4 4 4 4 4 4 80 4 198 136 4 4
```

Tokenize Binary and Source Into Useful Units

- Lex source into language-appropriate tokens
  - Keep most popular variable names
    - Normalize others
- Assign integers based on frequency in the corpus

Integer	Source Token	Integer	Source Token
4	(	15	}
5	)	16	*
6	;	17	if
7	,	18	var_3
8	var_0	19	0
9	function	20	“string”
10	=	21	]
11	var_1	22	[
12	->	23	var_4
13	var_2	24	.
14	{	25	1

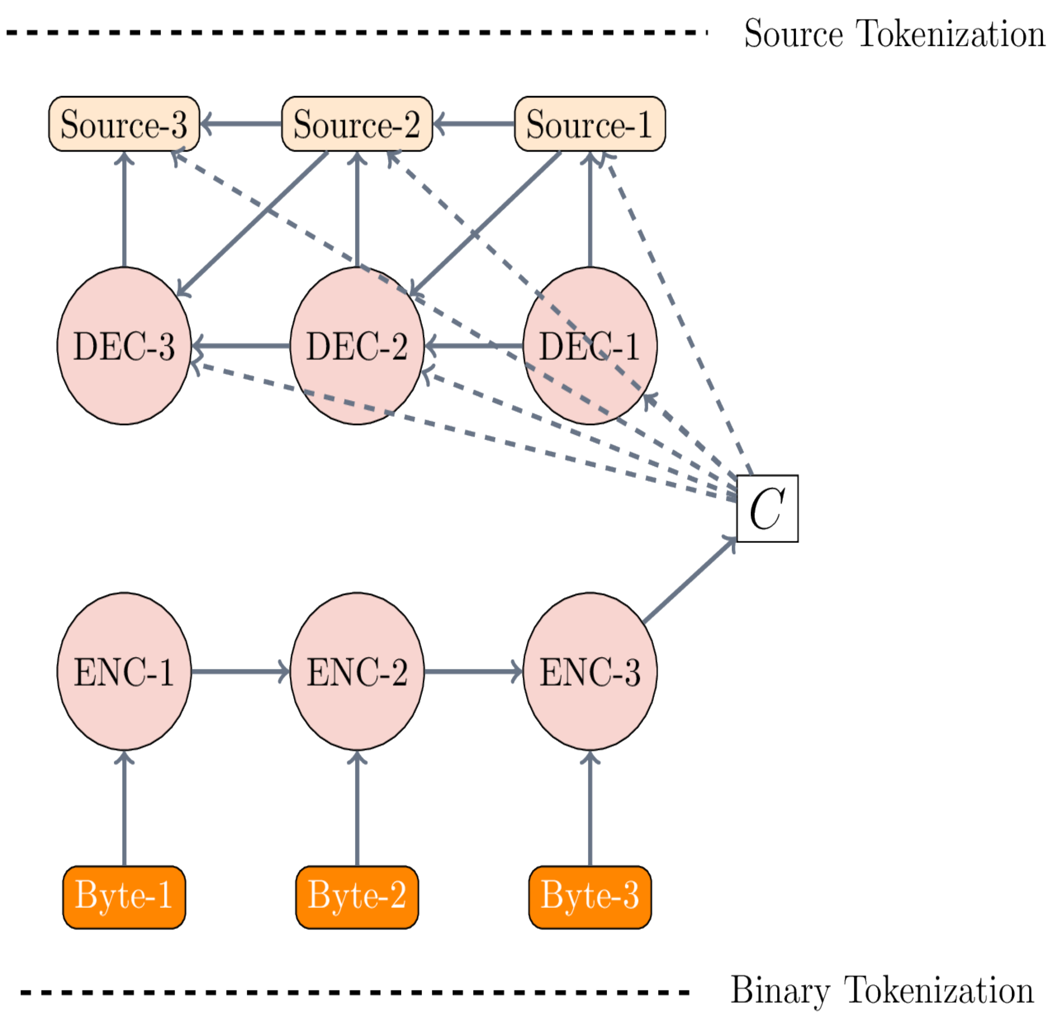
Our Trained Model Takes Binary Machine Code as Input



INPUT:

```
00 00 09 00 d3 12 00 00
70 33 00 00 00 00 00 00
00 00 09 00 d3 12 00 00
78 33 00 00 00 00 00 00
00 00 09 00 d3 12 00 00
```

Our Trained Model Turns Binary Machine Code Into Tokens



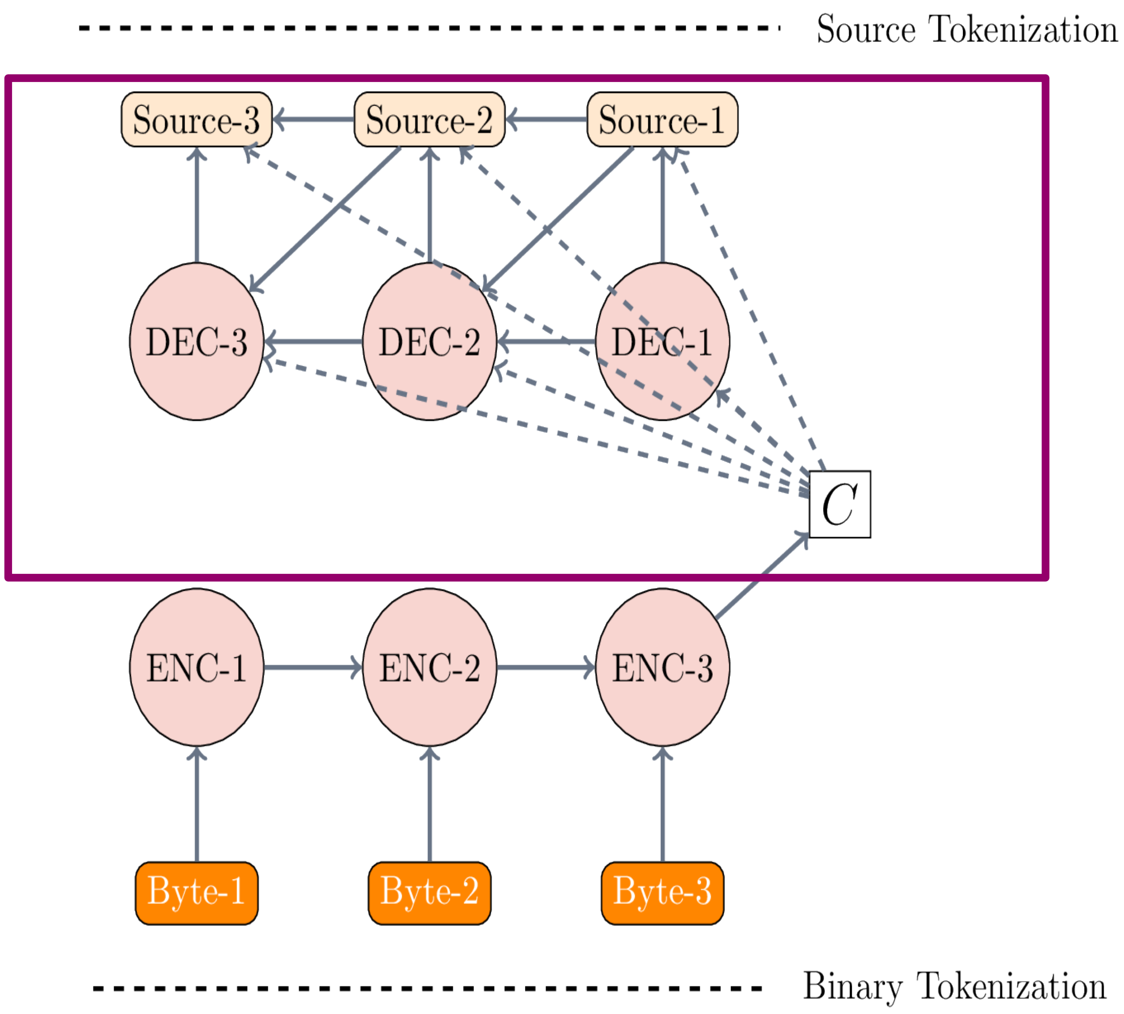
Binary Tokenization:

4 4 80 4 198 136 4 4
118 173 4 4 4 4 4 4 4 4
80 4 198 136 4 4 78 173
4 4 4 4 4 4 4 4 80 4
198 136 4 4

INPUT:

00 00 09 00 d3 12 00 00
70 33 00 00 00 00 00 00
00 00 09 00 d3 12 00 00
78 33 00 00 00 00 00 00
00 00 09 00 d3 12 00 00

Our Trained Model Translates Binary Tokens to Source Tokens

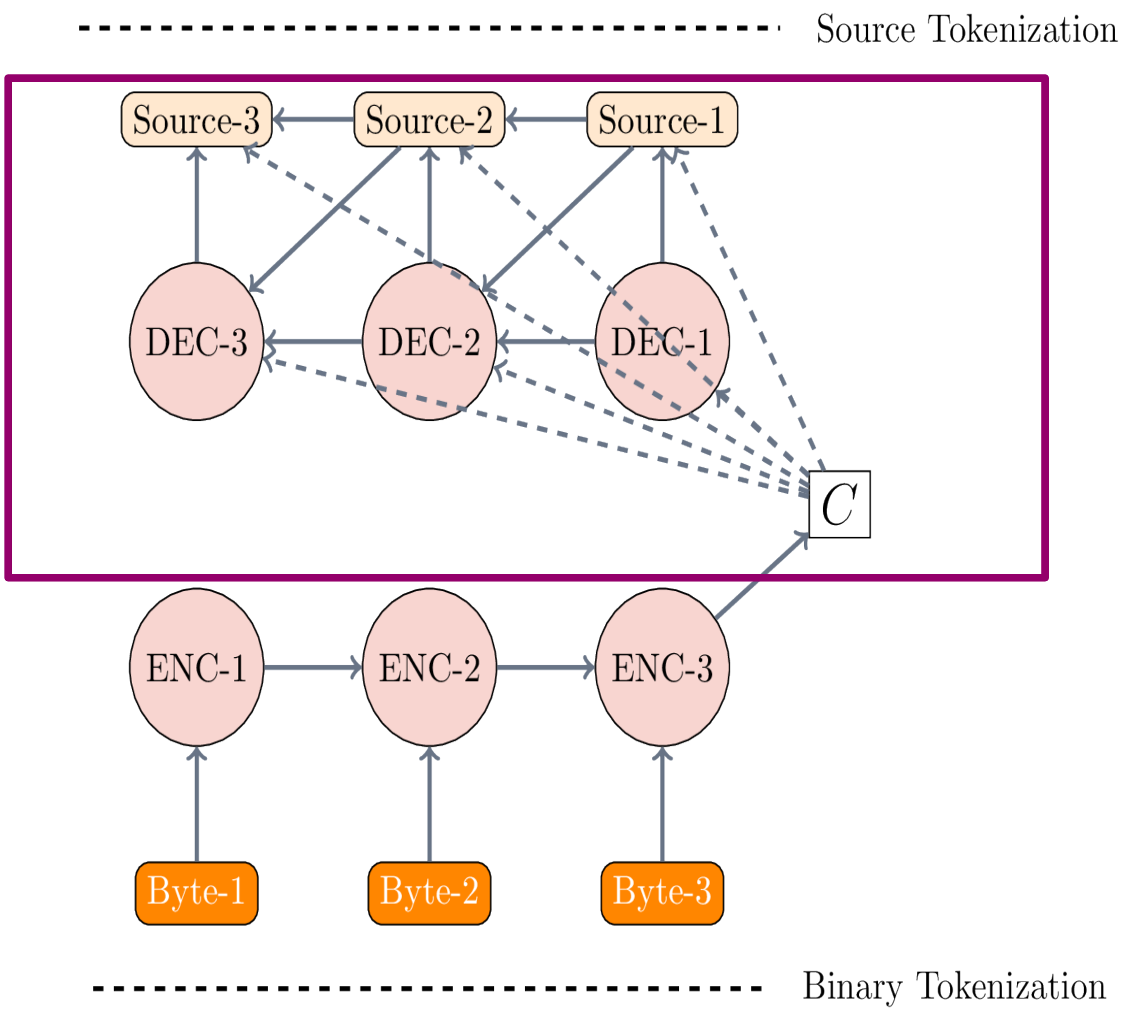


Source Tokenization:
111 4 8 42 31 5 6

Predictions

Binary Tokenization:
4 4 80 4 198 136 4 4
118 173 4 4 4 4 4 4 4 4
80 4 198 136 4 4 78 173
4 4 4 4 4 4 4 4 80 4
198 136 4 4

Our Trained Model Translates Binary Tokens to Source Tokens



(Actual tokenizations are reversed to allow the model to build context)

Source Tokenization:

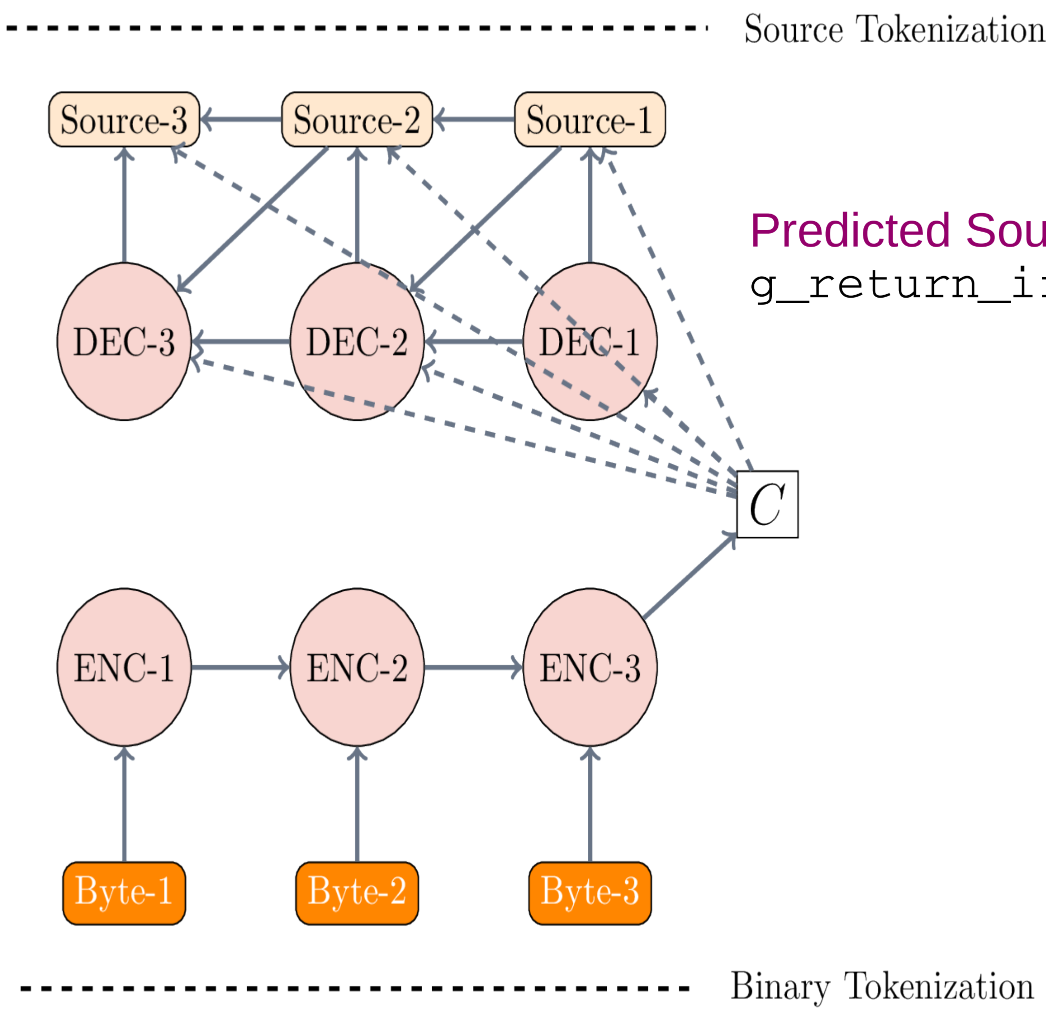
111 4 8 42 31 5 6

Predictions

Binary Tokenization:

4 4 80 4 198 136 4 4 118 173 4 4 4 4 4 4 4 4 80 4 198 136 4 4 78 173 4 4 4 4 4 4 4 4 80 4 198 136 4 4

Our Trained Model Turns Source Token Sequences Into Source Code



Predicted Source Code:

```
g_return_if_fail( var_0 != var_NULL );
```

Source Tokenization:

```
111 4 8 42 31 5 6
```

Evaluating Accuracy and Usefulness

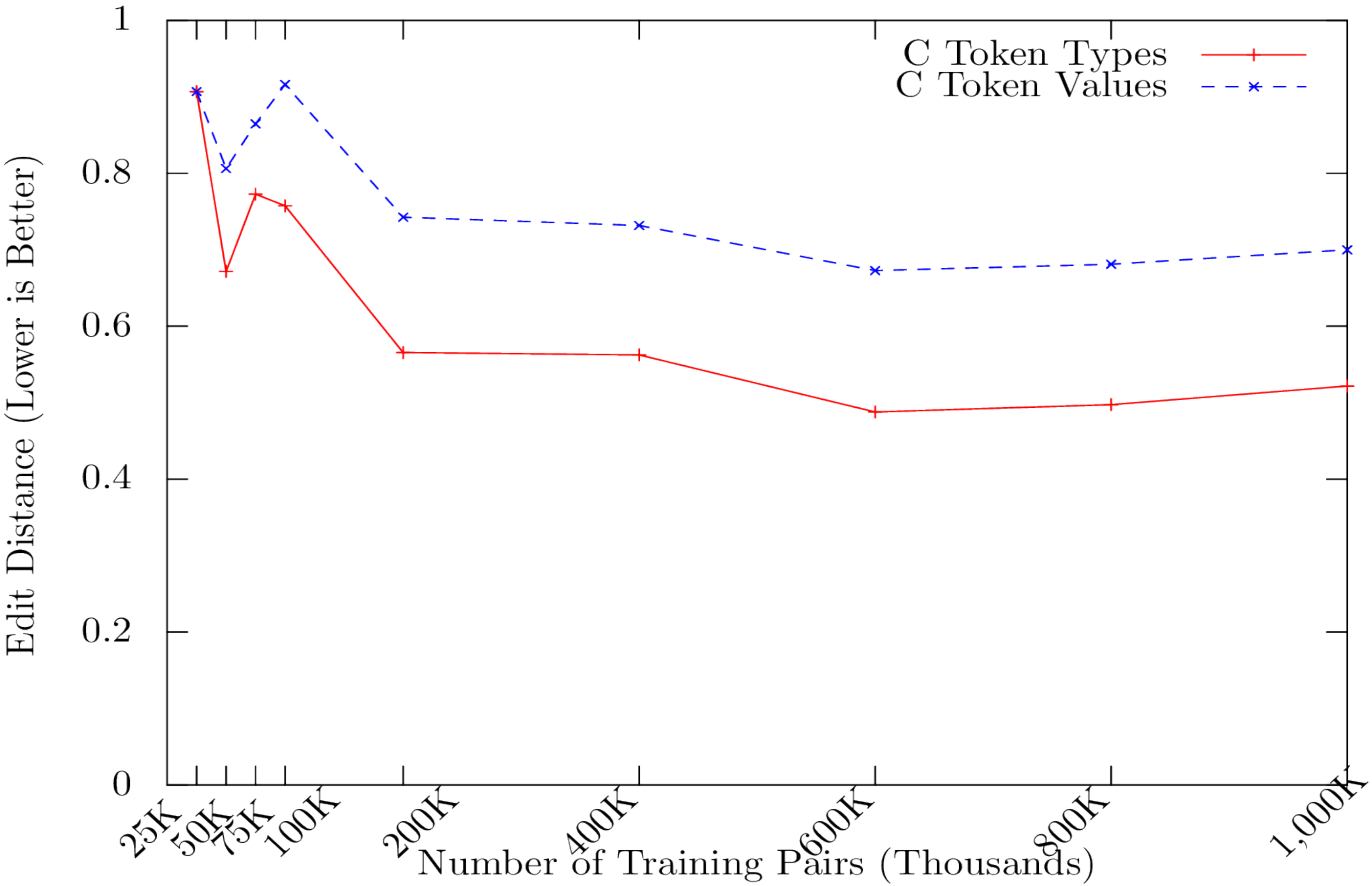
- We evaluate on a metric based on edit distance (lower is better)
  - Evaluation on recovery of exact token sequences
  - Evaluation on recovery of the correct types of tokens
- Ideally, we would like to do a user study to evaluate the usefulness of the translations

Research Questions:

RQ1: How long do we have to train for useful translations?

RQ2: How effective is our technique at translating machine code binary to C source code?

RQ1: The Effect of Additional Training Levels Out



RQ2: Usefulness by Edit Distance

	Maximum Number of C Tokens Per Snippet	Mean Edit Distance	Mean Edit Distance on Token Types
All C Source		0.70	0.52
Small snippets	5	0.65	0.56
Small-medium	9	0.67	0.45
Medium	17	0.72	0.52
Large	88	0.75	0.55

Example: Recovery of Function Call, Function Name, and Variable Name

- Ground Truth:

```
g_return_if_fail(screen_info != NULL);
```

- Translation:

```
g_return_if_fail( var_0 != var_NULL );
```

- Edit distance: 0.29

Example: Recovery of Function Call, Function Name, and Variable Name

- Ground Truth:

```
g_return_if_fail(screen_info != NULL);
```

- Translation:

```
g_return_if_fail( var_0 != var_NULL );
```

- Edit distance: 0.29

Example: Recovery of Function Call, Function Name, and Variable Name

- Ground Truth:

```
g_return_if_fail(screen_info != NULL);
```

- Translation:

```
g_return_if_fail( var_0 != var_NULL );
```

- Edit distance: 0.29

Example: Recovery of the General Structure of a Statement

- Ground Truth:

```
itr->e = h->table[i];
```

- Translation:

```
var_0->var_1 = var_2->var_3;
```

- Edit distance: 0.64
  - Misses variable names and array index

Example: Recovery of an `if` statement

- Ground Truth:

```
if (ts) {
    adjusted_timespec[0] = timespec[0];
    adjusted_timespec[1] = timespec[1];
    adjustment_needed = validate_timespec(ts);
}
```

- Translation:

```
if ( var_0 ) {
    function( var_1 , var_0->var_2 );
}
```

- Edit distance: 0.79

Example: Recovery of a `for` loop

- Ground Truth:

```
for (node = tree->head; node; node = next) {
      next = node->next;
      avl_free_node(tree, node);
   }
```

- Translation:

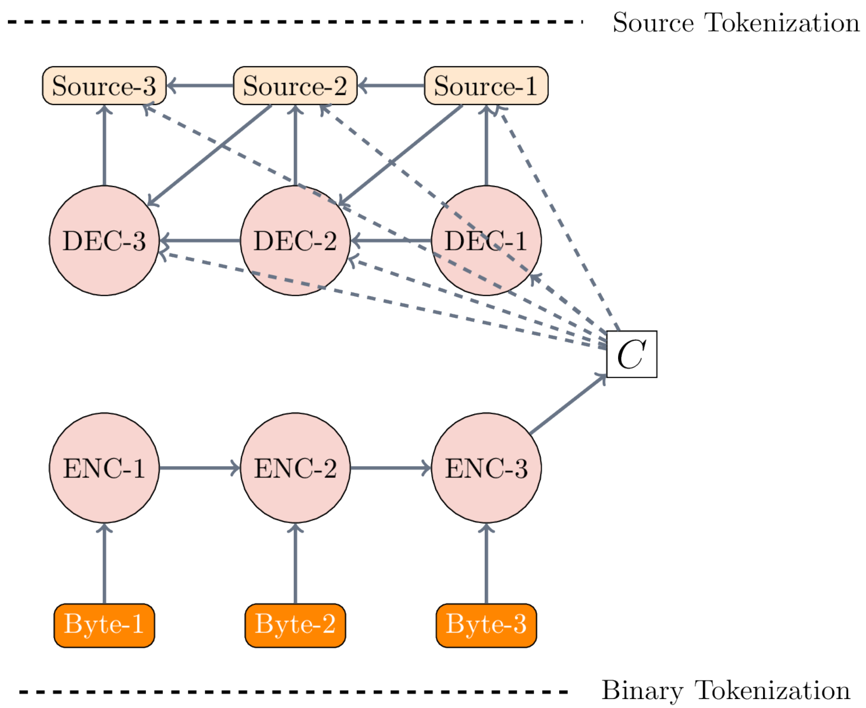
```
for ( var_0 = var_1 ) var_0 != var_NULL ; var_0 =
var_0->var_2 {
     function(var_0->var_3);}
```

- Edit distance: 0.66

Technique Advantages

- Language-independence
- Recovers semantic knowledge about programs

Summary




```
itr->e =
h->table[i];
```

```
var_0->var_1 =
var_2->var_3;
```