Using Recurrent Neural Networks for Decompilation

Deborah S. Katz
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
dskatz@cs.cmu.edu

Jason Ruchti and Eric Schulte
Grammatech, Inc.
Ithaca, NY, USA
{jruchti, eschulte}@grammatech.com

***Abstract*—Decompilation, recovering source code from binary, is useful in many situations where it is necessary to analyze or understand software for which source code is not available. Source code is much easier for humans to read than binary code, and there are many tools available to analyze source code. Existing decompilation techniques often generate source code that is difficult for humans to understand because the generated code often does not use the coding idioms that programmers use. Differences from human-written code also reduce the effectiveness of analysis tools on the decompiled source code.**

To address the problem of differences between decompiled code and human-written code, we present a novel technique for decompiling binary code snippets using a model based on Recurrent Neural Networks. The model learns properties and patterns that occur in source code and uses them to produce decompilation output. We train and evaluate our technique on snippets of binary machine code compiled from C source code. The general approach we outline in this paper is not language-specific and requires little or no domain knowledge of a language and its properties or how a compiler operates, making the approach easily extensible to new languages and constructs. Furthermore, the technique can be extended and applied in situations to which traditional decompilers are not targeted, such as for decompilation of isolated binary snippets; fast, on-demand decompilation; domain-specific learned decompilation; optimizing for readability of decompilation; and recovering control flow constructs, comments, and variable or function names. We show that the translations produced by this technique are often accurate or close and can provide a useful picture of the snippet's behavior.

***Index Terms*—decompilation; recurrent neural networks; translation; deep learning;**

I. Introduction

Decompilation is the process of translating binary machine code into code at a higher level of abstraction, such as C source code or LLVM intermediate representation (IR). Decompilation is useful for analyzing or understanding a program in many situations in which source code is not available. Many developers and system administrators rely on third-party libraries and commercial off-the-shelf binaries with undistributed source code. In systems with otherwise-stringent security requirements, it is necessary to perform security audits of these binaries. Similarly, malware is distributed without source code, but it is vital for security researchers to understand its effects [1], [2]. While new techniques for direct analysis of binary executables have shown promise [3], [4], analysis is generally easier on source code than on binary machine code. It is even often useful for a human to see a small portion of the binary of a program translated to a form that is easier for a human to understand, such as in tandem with an analysis tool such as IDA[1] or GrammaTech's CodeSurfer for binaries.[2]

Unfortunately, useful and natural decompilation is hard. Existing decompilers often produce code that does not conform to standard idioms or even parse, leading to confusion for both human and automated analysis. For example, existing decompilers can produce source code that contains elements that are uncommon or particularly difficult-to-understand, such as `GOTO` statements or explicit, high-level representations of what are usually under-the-hood memory loads and stores [5]. These elements reduce the usefulness of the decompiled source code because they do not line up with how a developer would reasonably structure or think about a program.

Recurrent neural networks (RNNs) [6] are a useful tool for various deep learning applications, and they are particularly useful in translating sequences. That is, they have been employed in various situations where a sequence of tokens (e.g., words, characters, or symbols) in one vocabulary (e.g., English or French) can be translated to another [7]. They are useful in translating between natural (human) languages, such as in translating from English to French, and in translating from a natural language to other forms of sequences [8]. In recent years, the technology around RNNs and using them for language analysis has advanced greatly, both on the hardware and software side. Increasingly powerful GPUs [9] and even purpose-built hardware [10] have enabled practical use of meaningfully large RNNs [9]. Various generally-available frameworks such as TensorFlow,[3] Keras,[4] and Caffe[5] have enabled researchers to apply deep-learning technologies (such as RNNs) to various problem domains.

Decompilation can be seen as a translation problem. The decompiler needs to translate a sequence of tokens in binary machine code into a sequence of tokens in a higher

[1] https://www.hex-rays.com/products/ida/overview.shtml
[2] https://www.grammatech.com/products/codesurfer
[3] https://www.tensorflow.org/
[4] https://keras.io/
[5] http://caffe.berkeleyvision.org/

level language, such as C source code. Previous work has investigated decompilation using other techniques that were originally used for translating natural languages, such as statistical machine translation [11]. A translation system based on RNNs needs a corpus of training data. The very large amount of publicly-available, open-source code provides a good basis for building a corpus of parallel binary machine code and higher-level source code for training and testing an RNN-based decompiler. Furthermore, additional work using various forms of deep learning, including RNNs, to analyze various aspects of code and programming languages shows that this area has promise [12]–[15].

We propose a novel technique using a model based on RNNs for decompilation. We create a corpus pairing short snippets of higher-level code (C source code) with corresponding bytes of binary code in a compiled version of the program. We train an RNN-based encoder-decoder translation system on a portion of the corpus. The trained model can then accept previously-unseen pieces of binary code and output a predicted decompilation of the corresponding higher-level code. We then test the trained model on snippets of binary that had not been seen in training, evaluating the predicted decompilations based on their fidelity to the original source code.

A major advantage of our approach is that it is general and easily retargeted to different languages. While most existing tools that turn binary machine code into a more human-understandable form are designed specifically for a particular language, our approach is designed to be general and extend to any language for which there is a sufficient corpus of training data. While some domain knowledge of the specific target language is useful in determining the best approach to preprocessing the input data and postprocessing the output, there is nothing in the technique that inherently relies on knowledge of how either the language or the compiler operates.

Another advantage of our approach is that it is well-suited to decompiling small snippets of binary machine code. As mentioned above, a user of a reverse engineering tool such as IDA or GrammaTech's CodeSurfer may want to see a local translation of a series of bytes into a source language, to give the reverse engineer a more complete understanding of the code. Our approach is ideally suited to working with analysis tools in this way.

The main contributions of this work are as follows:

- Adapt encoder-decoder RNN models originally designed for natural language translation to translation between programming language representations, including compiled machine code.
- An evaluation of the utility and speed of RNN-based models for decompilation. The evaluation shows that our technique often provides reasonably quick and understandable snippet decompilation.
- Post processing techniques that improve the effectiveness of RNNs for reverse engineering.
- Identification of useful tokenization schemes to support reverse engineering.

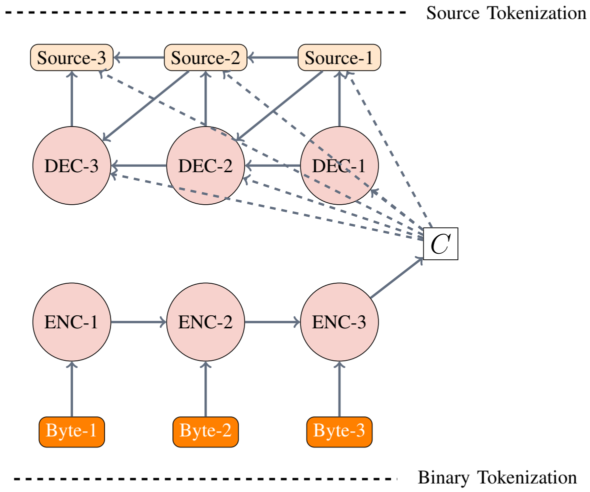

Fig. 1. Encoder Decoder Architecture

This paper proceeds as follows. Section II explains the general architecture of our system and our procedure for using it. Section III discusses our experimental setup, focusing on parts of the general approach that are specific to our experiments. Section IV presents our results. In Section V, we present several threats to validity. Section VI discusses some of the other work that uses techniques similar to ours or has similar goals. In Section VII we conclude.

II. Approach

The following sections introduce our approach to using an RNN-based technique for decompilation and IR lifting:

- An overview of the general procedure (Section II-A)
- Creating a corpus of data (Section II-B)
- Preprocessing the data for use in the RNN-based model (Section II-C)
- Training the model using a subset of the preprocessed data (Section II-D)
- Testing the accuracy of the trained model, using a different subset of data (Section II-E)

These sections set out the general approach, while Section III sets out details of modifications specific to our experiments.

A. Overview

We have developed a technique for using a model based on a recurrent neural network (RNN) as a decompiler. Here we provide a brief primer on recurrent neural networks and design choices specific to the problem of decompilation. In general, we use an existing library to train and validate our models and provide this overview as background.

Unlike traditional neural networks, recurrent neural networks (RNNs) have feedback loops, allowing information to persist, which in turn permits reasoning about previous inputs [6]. An unrolled recurrent neural network has a natural sequence-like structure which makes it well suited for translation tasks. There are several variations on RNNs; of

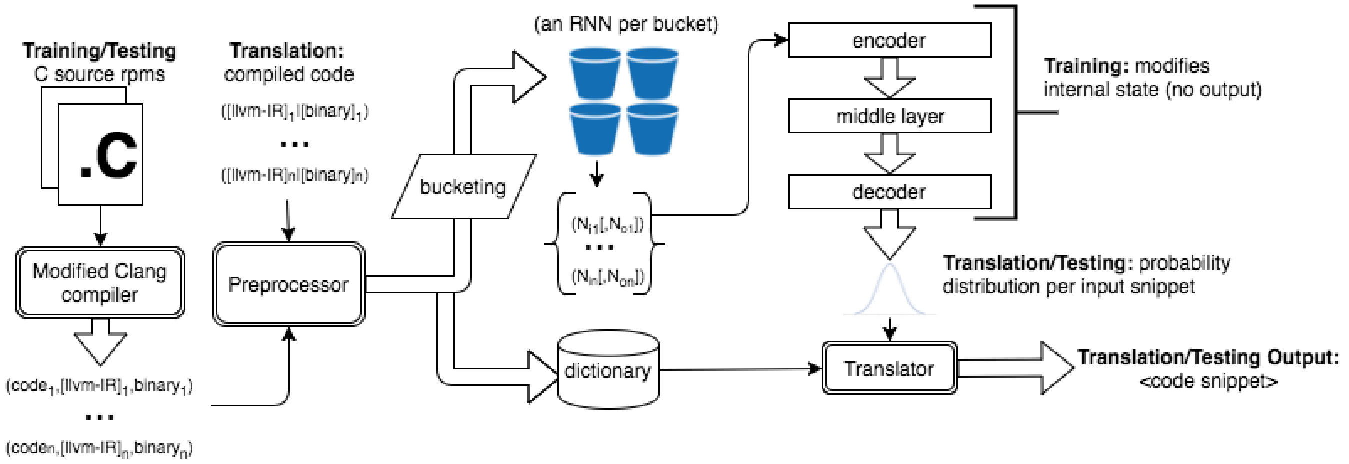

Fig. 2. Overview of Architecture of RNN-Decompilation System

relevance is the Long Short-Term Memory (LSTM) model, which consists of a standard recurrent neural network with memory cells [16] which are well-suited for learning long term dependencies. Prior applications of LSTM-based models have included learning natural language grammars and extracting patterns from noisy data [16].

Our models are based on existing sequence-to-sequence models that make use of RNNs. These and related models are well-suited to tasks that have sequences as inputs and outputs, such as translations, and have been used for many such tasks, from translating natural languages to developing automated question-answering systems [17]. A sequence-to-sequence model is composed of two recurrent neural networks: an encoder, which processes the input sequence, and a decoder, which creates outputs [7]. Further, the sequence-to-sequence model contains a hidden state, C, which summarizes the entire input sequence. As shown in Figure 1, to generate the first source token, the decoder cell relies on the summary, C, of the entire input sequence. Subsequent states are generated based on the previously-generated output tokens, the input sequence summary C, the previous decoder state, and an activation function yielding valid probabilities (in our case, softmax).

Overall, using RNNs for a task like decompilation involves *training* the model using input data with associated known-good answers, and then *validating* its correctness. Both tasks involve design decisions related to the structure and handling of the data in question. We address some of these decisions in Sections II-B through II-E and Section III.

Figure 2 provides a simplified overview of the architecture of our RNN-based decompiler. We begin with training input that is drawn from open-source software packages in the RPM Package Manager (RPM) format that contain C source code (Section II-B). We compile those RPMs using a modified compiler based on Clang,[6] which decomposes the source into small snippets that correspond to subtrees of an abstract syntax tree (AST) and then produces a pair for each produced snippet. Each pair consists of strings representing the snippet of source code and the corresponding binary output.

We preprocess the snippets by tokenizing each string (Section II-C) and translating the resulting token list to a list of integers. Preprocessing produces two outputs: (1) a list of integers for each string in the set of pairs and (2) a dictionary of which integers correspond to which tokens in both the higher-level and binary snippets.

We place each pair of lists of integers into one of four buckets, chosen based on the lengths of the lists. Bucketing is an approach for efficiently handling token sequences of different lengths. The reasoning behind using bucketing in this type of model is explained in other sources [18]. For training, we provide the pairs of lists for each bucket to its corresponding encoder-decoder (Section II-D). During training, the encoder-decoder modifies its internal state and does not provide additional output, although we can access some internal state and evaluation metrics used for training.

When we have trained models, we can then use those models to translate binary snippets to the relevant higher-level language (*i.e.*, C source code). See Section II-E. Using the the same process of tokenization used in preprocessing and the dictionary generated in the preprocessing step, we turn the binary snippets to be translated into lists of integers. We then provide each list of integers (representing each binary snippet) to the trained encoder-decoder model. The decoder layer outputs a probability distribution for each of the lists of integers (representing binary snippets). We then turn the probability distribution into a list of integers that represents the predicted corresponding higher-level code. We use the dictionary for the higher-level language, generated in the preprocessing step, to turn the list of integers into a predicted higher-level translation.

B. Corpus Creation

Using an encoder-decoder RNN for translation requires a set of examples of correct translations. In a natural language translation application, the pairs might be pairs of sentences

[6] https://clang.llvm.org/

that have the same meaning – one in the original language, such as English, and one in the destination language, such as French. For our application, translating binary data to a higher level representation, our pairs are snippets of compiled binary code paired with the corresponding higher level code, such as C source code.

To obtain these pairs of snippets, we create a database by compiling programs using a customized compiler based on Clang and LLVM; this compiler pairs AST trees and subtrees of source code with the binary machine code into which those trees and subtrees compile. From the database of snippet pairs, we used pairs that had 112 or fewer binary tokens and 88 or fewer source code tokens. An explanation of the tokenization of these snippets can be found in Section II-C. The programs are drawn from open source RPMs for Fedora containing C source code. Details of the corpus can be found in Section III-A.

C. Preprocessing

The RNN infrastructure we use is designed to operate on pairs of lists of integers, so we preprocess our snippets to turn them from strings to meaningful lists of integers that the RNN can understand.

Tokenization: The first step, tokenization, turns the snippets, which are represented as strings, into lists of smaller units. We use some domain knowledge in choosing a tokenization scheme, as we must choose one that makes sense for the language. A part of this choice is choosing an appropriate level of granularity. For example, in tokenizing binary code, we have several options. A naive choice would be to tokenize the binary using only two tokens, as a sequence of ones and zeros. However, this choice would not be the most efficient – it would require keeping large amounts of data; it would require large RNN structures; and it would not take advantage of the system's ability to deal with a large vocabulary of tokens. As another choice, we could tokenize several bytes into each token. However, given the variety of sequences of bytes that appear in binary code, this tokenization scheme would make it difficult for the RNN to learn which patterns occur frequently at a lower-level.

We tried several tokenization schemes depending on what kind of code we were tokenizing. For source code, we tested tokenizing the text character-by-character, on whitespace, and using a lexer for C source code. Finally, for binary, we tried tokenizing bit-by-bit and byte-by-byte. As a result of an informal parameter sweep, we found the best results tokenizing binary byte-by-byte, tokenizing C source code using a Python-based C lexer.[7] For each token, the lexer provides a string that categorizes the token as, for example, an identifier, a language keyword, a string literal, or a float constant. We use the category strings provided to identify tokens that represent variable identifiers, function identifiers, and string literals. For string literals, we replace the contents of the string with the word, `STRING`, followed by any format specifiers that appear in the original string. We do this because we anticipate that string literals would be more difficult for the model to get right because they do not follow a regularized pattern; in addition, they are often more easily recoverable by other means. We add the format specifiers because having the correct number and types of format specifiers is essential for compilation. For function identifiers, we keep the 20 most frequently-used function identifiers and replace the others with the word, `function`. For variables, we keep the 100 most frequently-used variable identifiers, and for the others, we replace each unique identifier used within a snippet with `var_XXX`, where `XXX` is an integer counter. For other tokens, we take the tokens as they are. By replacing infrequently used variables, functions, and strings with a canonical placeholder value, we minimize the vocabulary size and allow the RNN to fit more closely to AST sequences instead of overfitting to variable names.

From Sequences of Tokens to Sequences of Integers: We use a standard methodology for turning the training snippet pairs into an input vocabulary the RNN can accept. Looking separately at each set of tokens (*i.e.*, repeating the process separately for the binary tokens and the C source tokens), we rank each unique token according to how frequently it occurs in the training data. For each snippet, we replace each token with an integer corresponding to the popularity rank, up to a ceiling. For less popular tokens, we replace the token with an integer representing "unknown".

Three additional integer tokens represent unique values in the RNN infrastructure: `_PAD`, `_GO`, and `_EOS`. The integer representing the `_GO` symbol is prepended to each integer list representing a target sequence (here, the C code snippets), while `_EOS` is appended to the end.

The pairs of integer lists are then categorized into buckets based on length. Buckets allow pairs of similar lengths to be grouped together, for efficiency of the RNN structure. For each bucket, there is a maximum length for binary sequences and a maximum length for target higher-level sequences. Each pair of lists is placed in the smallest bucket for which both binary and target token lists fit. The integer representing the `_PAD`, symbol is appended to the end of the lists of integers so that all binary snippets in a bucket are the same length and all higher-level snippets in a bucket are the same length. We choose bucket sizes based on the lengths of the pairs of snippets in our training set, allowing for four buckets that contain roughly equal numbers of snippets.

Readying Sequences for the Model: Because of technical limitations relating to the size of memory and our RNN design, we discard some snippets that are too long. The threshold is determined dynamically, based on the composition of the training set and the bucket sizes.

In a final preprocessing step, we reverse the sequence of tokens corresponding to the binary snippet, a technique used by Sutskever et al. in natural language translation [17].

The pairs of lists of integers representing pairs of snippets are then used to train and evaluate an RNN model. Our training subset consists of a pre-defined number of pairs of binary and

[7] https://github.com/eliben/pycparser/tree/master/pycparser

source tokens corresponding to snippets from the corpus. This subset is selected pseudorandomly.

D. Training

Here, we present a simplified overview of training our system. Conceptually, the encoder-decoder RNN has three parts: an encoder layer, a decoder layer, and a layer or layers in between.

To train the RNN model, we draw a batch of training examples from a given bucket, including both the tokens corresponding to the binary (input) and the tokens corresponding to the higher-level code (C source code). We feed the batch of binary input tokens to the encoder layer of the RNN corresponding to that bucket, while we feed the corresponding batch of higher-level tokens to the decoder layer. The encoder layer embeds the inputs into fixed-size vectors to provide to the decoder layer. The system uses the inputs and outputs to alter the internal state and parameters for all layers. It is this internal state and parameters that the model later uses to perform translations.

A subset of the training examples is held out during training for the model to use for self-evaluation. At intervals during training, the model evaluates itself on these held-out examples and adjusts its parameters and internal state based on performance.

E. Evaluation

Using a separate data set, we evaluate the performance of the trained RNN by feeding the model sequences of binary code. The RNN determines to which bucket the binary input belongs and provides the sequence of tokens as input to the encoder layer of the corresponding trained RNN. When testing the trained RNN, we do not provide inputs to the decoder layer, instead setting a parameter so that the RNN generates outputs from the decoder layer instead. Conceptually and simplifying the details, the encoder layer, using its saved parameters and internal state, turns the binary input tokens into a fixed-length embedding, which it provides to the layer or layers in between the encoder and decoder. The layer or layers in between translates the vector to a different vector to provide to the decoder layer. The decoder layer then generates a series of tokens, corresponding to predicted output in the destination language, C source code.

We compare the tokenized output C source code predicted by the RNN against tokenized known, ground truth values, taking the Levenschtein distance between the two.

III. Experimental Setup

In this section we provide details relating to how we implement each step described in Section II.

A. Corpus Creation

We leverage the Clang compiler toolchain and debug information to identify binary code associated with individual abstract syntax tree (AST) subtrees at the operator, expression, statement, and function levels. The code that is represented in snippets at a smaller AST granularity also appears in the appropriate snippets at larger granularity. Using this tool, we create a database of pairs of C source code snippets and corresponding binary snippets. Note that, although we attempt to limit the corpus to C code, there may be a small subset of C++ snippets.

The full corpus contains 1,151,013 pairs, of which 73 are literals, 249,331 are operators, 203,305 are expressions, 649,842 are statements, 47,585, are functions, and 972 belong to other categories.

The corpus of binary and C source code snippet pairs is drawn from the Fedora selections for the MUSE project, a DARPA project to develop software engineering techniques that draw on large amounts of available code.[8] From those RPMs, we select programs that use C source code and compile with minimal modification. We use our custom compiler to associate the AST-appropriate binary snippets to corresponding C source code snippets.

Although larger snippets are in our corpus, because of technical limitations, we limit the length of the snippets we use in training and testing to a maximum number of tokens, the number chosen dynamically. Further details are in the discussion of bucketing in Section II-C.

We compile the corpus on a Fedora 19 virtual machine with Clang 3.7.1 to x86.

B. Training the RNN

We train and test the RNN model on two machines running Ubuntu 14.04.5 and TensorFlow 0.8, each with an Nvidia GeForce GTX 980 Ti GPU. Additionally, both have 12 virtual (6 physical) Intel Xeon CPUs E5-2620 0 @ 2.00GHz. The machine used for the C corpus has 32 GB RAM.

We modify an existing encoder-decoder model, based on the translate.py file, that makes use of the seq2seq model, distributed with TensorFlow version 0.8.[9]

Any RNN-based system has many parameters and hyperparameters that can be adjusted. We conduct an informal parameter sweep and find best performance with the parameters shown in Table I. Note that for some parameters, performance was nearly equivalent for different values of the parameter, in which case we choose for convenience.

We train our models on 758,706 training pairs. We pick bucket sizes dynamically, based on the technical restrictions of the system and the lengths of the tokenized snippets in the training corpus. The bucket sizes and number of snippets per bucket is shown in Table II.

C. Evaluation

We use two evaluation metrics to assess the quality of the C source code generated by our technique. The first metric, perplexity, is based on the RNN's internal sampled softmax loss function, as described in Jean *et al.* [19], and is used

[8] http://corpus.museprogram.org

[9] TensorFlow can be found at https://www.tensorflow.org/. The files on which we based our model can be found at https://github.com/tensorflow/tensorflow/tree/v0.8.0rc0/tensorflow/models/rnn/

TABLE I
PARAMETER AND HYPERPARAMETER VALUES.

Parameter	Value
learning rate	0.8
learning rate decay	0.9
number of layers	4
size	768
max gradient norm	5
batch size	64
higher level vocab size limit	40000
binary vocab size limit	40000

TABLE II
BUCKET SIZES AND NUMBERS OF SNIPPETS PER BUCKET FOR THE LARGEST TRAINING CORPUS. "MAX. BIN. LEN" IS THE MAXIMUM NUMBER OF TOKENS IN A BINARY SNIPPET IN THE BUCKET. "MAX. C SOURCE LEN." IS THE MAXIMUM NUMBER OF TOKENS IN A C SOURCE CODE SNIPPET IN THAT BUCKET.

Max. Bin. Len.	Max. C Source Len.	# Training Pairs
11	5	79,747
22	9	240,490
47	17	274,412
112	88	311,016

during RNN training. Perplexity is a common measurement in work relating to RNNs, especially in work relating to models of languages. It is a measure of the accuracy of a probabilistic model in predicting a sample [20]. We use the perplexity measurement as a proxy for RNN performance to determine when to end training.

The second metric, an evaluation of the usefulness of the translations external to the measures used by the RNN, is based on the Levenshtein edit distance between the sequence of tokens in the ground truth source code associated with a given binary sequence and the sequence of tokens output by the RNN. We recognize that this evaluation metric is an imperfect proxy for the usefulness of the decompilation output; we leave a user study to future work.

We have also developed several 'compilation fixer' transformations that we run as a post-processing step on the predicted source snippets. While the RNN often gives a prediction close to the structure of the original source code, the prediction does not always observe syntax rules. Our transformations balance brackets, parenthesis, and braces; add missing commas; and delete extra semicolons. These simple transformations can be incorporated into any implementation of an RNN decompiler. However, although these transformations can improve human readability, we have not found them to have a large effect on edit distances. We believe that more sophisticated compilation fixing techniques may have the potential to have a greater effect, but we leave those to future work.

We use two variations on edit distance to evaluate our trained model. First, we look at straight edit distance between the tokens in the prediction and those in the ground truth snippet. In addition, to ensure differences in identifier names and constants do not influence our results, we lex both the output translations and ground truth snippets to obtain sequences of token types. We perform the edit distance calculation between corresponding sequences of token types.

We report the average edit distance for the exact tokens predicted; the average edit distance for the types of tokens predicted; and the percent of predicted token sequences that match the ground truth token sequences perfectly, without post-processing. We report these numbers separately for each bucket and in aggregate for all buckets.

We are aware that other decompilation work uses different metrics, such as BLEU, edit distance ratio, and syntactic correctness ratio [11]. However, we believe that our metrics are more appropriate to our work.

IV. RESULTS

RQ1 How does the duration of training and number of examples trained on influence the effectiveness of the RNN for decompilation?

RQ2 How accurate is the RNN for translating binary data to C code?

A. Effects of Different Amounts of Training

RQ1: How does the duration of training and number of examples trained on influence the effectiveness of the RNN for decompilation?

Recall that our overall goal is to determine whether an RNN-based setup is useful for decompilation. This question evaluates the subgoal of determining how much training is needed or useful to train an RNN-based model to be effective for decompilation. To answer this question, we take the encoder-decoder-based system described in Section III and run it for binary-to-C-source translation. We train the model with the model with a given number of pairs, after which we pause the training and evaluate the effectiveness of the trained model. Then we continue the training with additional sets of pairs, evaluating after each set.

For this evaluation, we measure accuracy by edit distance between the post-processed predictions and the original ground-truth higher-level snippets, as described in Section III-C. Lower is better, and means that the predicted snippet is closer to the original source code snippet.

For this experiment, we train 9 models, each on pairs of snippets drawn from pools of 25,000, 50,000, 75,000, 100,000, 200,000, 400,000, 600,000, 800,000, and 1,000,000. We evaluate each trained model on a pre-defined set of test snippet pairs, using the same set for each. We feed each binary snippet to the RNN and record the RNN's prediction for the C source code.

Figure 3 shows results for translating binary code to C source code. The X axis represents the number of pairs of snippets given to the model for training. The Y axis represents the average token edit distance between the C snippets output by the RNN, and the ground-truth high-level snippets that correspond to the binary fed to the RNN. Results are shown both with and without post-processing and for token values as well as token types.

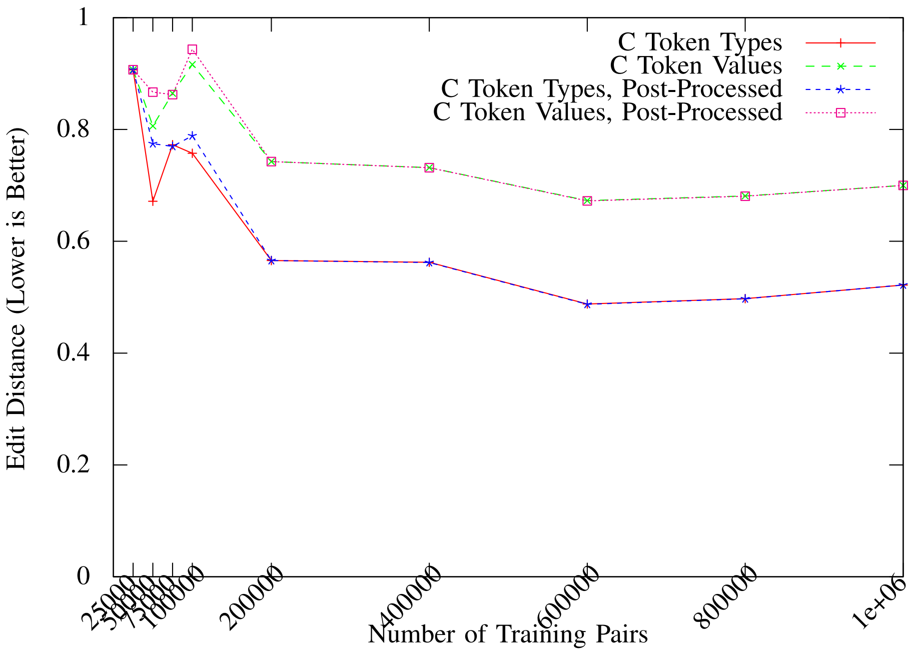

Fig. 3. Accuracy Of Translations, for C Source Given Different Numbers of Training Pairs

Note that the graphs show diminishing returns after 200,000 snippet pairs, and even worsening accuracy, after 600,000 snippet pairs. The results produced after this smaller amount of training on 200,000 snippet pairs, taking approximately 65 minutes may be good enough to use in many applications, avoiding the necessity of training the model through a much larger corpus, as suggested by the literature [7], [17].

Also note that the graph shows that the model is significantly more likely to produce output sequences of the correct token types than the exact correct tokens.

B. Translating Binary to C

RQ2: How accurate is the RNN for translating binary data to C code?

Recall that our overall goal is to determine whether an RNN-based setup is useful for decompilation. This question evaluates the effectiveness of the RNN-based setup for decompiling binary to C source code, specifically. To answer this question, we take the encoder-decoder-based system described in Sections II and III and run it on the corpus of preprocessed binary and C source code snippet pairs. We train on 758,706 snippet pairs, of which 152,016 are held out to be used to improve training, taking 243.33 minutes (14599.67 seconds) and test on a pre-defined set of 400 snippet pairs, not contained in the training set.

In this evaluation, we measure accuracy based on the Levenshtein edit distance between the token sequences of the post-processed predictions that our system produces and the tokens in the original ground-truth C source code snippets, as explained in Section III-C. Recall that lower results are better and correspond to translations that more closely resemble the ground truth.

We also measure the percent of snippets that are predicted to have the exact same token sequences as the ground truth. In addition, we report the time taken to train the model and translate the snippets.

Table III summarizes the results. It shows that the edit distance between predicted token sequence and ground truth token sequence, in general, averages 0.7000. Our model performs somewhat better for shorter token sequences than longer token sequences, as shown by the per-bucket performance. The snippets in Bucket 0 are shorter than those in Bucket 1, and so on, as shown in Table II.

The table also shows edit distances for sequences of token types, rather than the tokens themselves. That is, we lex the ground truth token sequence and the translated output token sequence and create a corresponding sequence of token types for each. The table shows significantly improved performance when we use token types. This result suggests that our technique is better at recovering syntactic structure than variable and function names. Although we perform some post-processing on the translated output, we do not present edit distances for those sequences here, because they are not significantly different from the non-post-processed edit distances.

Edit distance can be a non-intuitive measurement, and it does not precisely capture the usefulness of predicted translation outputs. We did not conduct a user study to quantify the usefulness of the results. For these reasons, we present Figure IV, which shows several examples of ground-truth snippets and their corresponding post-processed outputs from the trained model, along with the evaluation metrics for each translation. We picked these examples because they illustrate

TABLE III
SUMMARY OF STATISTICS ON ONE TRAINED MODEL. A LOWER AVERAGE EDIT DISTANCE IS BETTER. FOR EXAMPLES TO ILLUSTRATE WHAT THESE NUMBERS REPRESENT, SEE FIGURE IV.

	avg. edit dist.	avg. edit dist. token types	perc. perfect	training time (secs)	translation time (secs)	translation time per snippet (secs)
C Source	0.70	0.52	3.8%	14599.67	202.55	0.51
Bucket 0	0.65	0.56	11.0%			
Bucket 1	0.67	0.45	3.0%			
Bucket 2	0.72	0.52	0.0%			
Bucket 3	0.75	0.55	1.0%			

some of the strengths of the technique that are not necessarily captured in the edit distance metric.

Note that our technique can successfully recover various aspects of the original source code, as shown in the examples in Figure IV:

- In Example 1, the technique correctly predicts that the binary represents a function call, the name of the function called, the structure inside the parentheses that a variable is not equal to another variable, and the variable name, `NULL`.
- Example 2 recovers the general structure of the statement.
- Example 3 shows the recovery of the assignment of the result of a function call to a variable, although it does not capture that the two variables should be the same. In this example, the technique recovers the sequence of token types exactly (*i.e.*, variable identifier, equals sign, function identifier, open parenthesis, variable identifier, closed parenthesis, semicolon).
- In Example 4, the technique reproduces an `if` statement conditioned on a single variable with a function call in its body, although the details of the body are not precisely correct.
- Example 5, shows the recovery of a `for` loop. Although the syntax of the `for` loop is not exactly correct, the model does pick up on some of the semantics. If you consider `var_0` to represent the variable `node`, the translation correctly initializes `node` to another variable, tests whether `node` is null, and sets the increment to `node->next`, even though a portion of the increment happens in the body of the `for` loop in the ground truth.

These examples show that our evaluation metrics do not fully capture the amount of information conveyed in snippets decompiled using our technique and that our technique shows promise in human-facing scenarios.

V. THREATS TO VALIDITY

A. The Length of the Snippets

Because of technical limitations, our training and testing snippets are limited in length. Therefore, while our technique performs well at recovering source code from small binary snippets, we cannot handle longer ones. We believe, however, that decompiling short snippets is useful in itself and combined with other techniques to decompile larger portions of the program.

B. Training with only AST-Appropriate Snippets

Our training corpus and testing examples consist of snippets that are entire subtrees of an abstract syntax tree (AST). While we can control the makeup of our training corpus, if we would like to use our technique on binary that we know nothing about, we cannot necessarily determine where the boundaries of subtrees are in the binary. We do not know how well our technique will perform on snippets that represent incomplete subtrees or portions of more than one incomplete subtree.

C. Redundancy in Snippets

Because we use snippets at multiple levels of an AST, there is a chance that a snippet in the training set may be a subset of a snippet in the test set, and vice versa. This threat is mitigated by the use of bucketing, which means that snippets that are significantly different in length are unlikely to be in the same bucket. Because separate RNN models are used for each bucket, it is less likely that the inclusion of one of the snippets in the training set will affect the evaluation of the corresponding snippet.

There is also a risk from redundancy in that we did not ensure that the corpus was free from identical snippets, which means that a snippet that has been a part of the training set may be identical to a snippet included in the test set. However, since our corpus was drawn from real programs, any advantage conveyed by identical snippets may also exist in a real situation in which this approach may be used.

D. Compiler, Machine, and Options Limitations

We obtain our corpus using one compiler on one virtual machine, using one set of compiler options. Specifically, we compile at optimization level zero (`-O0`). We do not know whether our work will extend to other settings and optimization levels. We also do not know how a model trained on a corpus generated on one machine or with one set of options will perform on binary generated differently.

E. Limits of Experimental Investigation

We make claims that the source code that our technique produces is useful to humans. Although we believe this to be true from our own inspection, we do not conduct a user study to verify these claims, as others have recently done [2].

Studies of the applicability of a technique are naturally limited by the ingenuity and level-of-effort of the performer. The authors are confident that most decisions made in this study are

TABLE IV
EXAMPLE TRANSLATIONS FROM BINARY MACHINE CODE TO C SOURCE CODE GENERATED BY AN RNN MODEL. WE SELECTED THESE EXAMPLES TO SHOW INTERESTING PROPERTIES.

		Original Source and Decompiler Output	# Binary Tokens	Edit Dist.
Ex. 1	Ground Truth:	`g_return_if_fail(screen_info != NULL);`	176	0.29
	Translation:	`g_return_if_fail( var_0 != var_NULL );`		
Ex. 2	Ground Truth:	`itr->e = h->table[i];`	53	0.64
	Translation:	`var_0->var_1 = var_2->var_3;`		
Ex. 3	Ground Truth:	`aucBuffer = xfree(aucBuffer);`	41	0.43
	Translation:	`var_0 = function( var_1 );`		

Ex. 4	Ground Truth:	`if (ts) {`
		`adjusted_timespec[0] = timespec[0];`
		`adjusted_timespec[1] = timespec[1];`
		`adjustment_needed = validate_timespec(ts);`
		`}`
			158	0.79

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

Ex. 5	Ground Truth:	`for (node = tree->head; node; node = next) {`
		`next = node->next;`
		`avl_free_node(tree, node);`
		`}`
			164	0.66

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

reasonable and that obvious optimizations and extensions were tried. However, there remain additional untested techniques which may significantly improve the applicability of RNNs to decompilation, specifically the use of structured generation of source code and context.

Constraining the generation of output from the RNN such that only structurally valid parseable source may be emitted has been shown to significantly improve the ability of RNNs to generate source code [13]. We do not evaluate this technique.

Similarly, the use of context in natural language process has been shown to significantly improve RNN translation performance [21]. In many decompilation tasks snippets may appear within a wide context of surrounding machine code. Intuitively this context provides a great deal of information relevant to the decompilation of a machine code snippet including types of variables held in registers and surrounding control-flow constructs. The integration of context into RNN decompilation may also provide for better results than those reported in this work.

Choosing an evaluation metric not directly comparable to those used for other decompilation work prevents us from claiming that our decompilation technique outperforms others.

VI. RELATED WORK

A. Traditional Decompilation

Traditionally, decompilers operate in several stages: (1) parse and disassemble the binary into a machine-neutral intermediate representation, attempting to replace compiler idioms with higher-level operations; (2) use data-flow and type analysis to transform intermediate representation into control-flow structures; (3) translate the control-flow structures into source code. Ďurfina *et al.* provides an outline of the history of decompilation [22].

In recent years, research has focused on the control-flow structuring stage of decompilation [1], [5], [22]–[25]. Yakdan *et al.* [5] introduce pattern-independent control-flow structuring, a technique capable of performing semantics-preserving transformations which produce structured source code with reduced numbers of GOTOs, which can appear in traditionally-decompiled code.

Because our technique leverages human-written source code for training, we are able to generate code which is often less idiosyncratic than code created by traditional decompilers. In addition, by focusing at a lower-level, our technique may be used in conjunction with other decompilation techniques that generate structure; the other techniques can suggest a high-level structure, while our technique can fill in lower-level pieces with more useful code.

Other recent decompilation work has focused on modeling complex instruction set architecture semantics by leveraging compiler knowledge. Hasabnis and Sekar's approach uses machine learning and achieves great success at lifting binary code to their intermediate representation [26].

B. Related Techniques for Source Code

A wide variety of software engineering problems are being attacked with "Big Code" and machine learning. JSNice and JSNaughty use a large corpus of publicly-available Javascript code to predict identifier names and type annotations in obfuscated Javascript programs with high accuracy [27], [28].

The research of Caliskan-Islam *et al.*, into identifying authorship of compiled binaries based on coding style indicates that there is a strong relationship between human-written source code style and even highly-optimized compiled code. This work indicates that using machine learning on the binary machine code in conjunction with its corresponding source code may provide machine-useable insights on how to write code like a human [29].

Recent work by Levy and Wolf focuses on using neural networks to align snippets of source code to the corresponding compiled code, which can also be used in conjunction with our technique [30].

Madison and Tarlow focus on generative models of natural source code. Their work has applications for source code autocompletion, mining and understanding API usage patterns, and enforcement of coding conventions. Their trained model has demonstrated success in completing partial programs and observing proper syntax rules [13]. By leveraging their result, we could constrain the RNN to only generate parseable sequences of tokens; this should have a very significant impact on the quality of the generated output.

Mou *et al.* propose a method for building vector representations of programs, as a basis for using deep learning for program analysis [31]. Van Nguyen *et al.* propose using vector representations of programs for retrieval of API examples [15]. Some of their techniques may be applicable in conjunction with our work.

Others have used statistical machine translation, another technique used for translation of human languages, for decompilation [11], [32], [33].

C. Related Techniques in Natural Language Translation

While neural machine translation demonstrates promise, Bahdanau *et al.* note that, because a traditional encoder-decoder neural network must compress all the necessary information for the source sentence into a fixed-length vector, performance may suffer, especially for longer sentences [34]. Bahdanau *et al.* propose a extension which learns to align and translate jointly. We would like to leverage this technique for decompilation as, especially with compiler transformations enabled, source code may be elided or relocated in the compiled binary.

VII. Discussion and Conclusions

We have demonstrated success in decompilation through use of an RNN-based technique. In our experiments, we have shown that an encoder-decoder RNN structure can be useful in recovering the structure of source code snippets from binary machine code snippets.

We believe that this approach is useful both in itself and in combination with other decompilation techniques.

In addition, our model can be extended to any other language or platform for which there exists a sufficient corpus of paired binary machine code snippets and higher-level language snippets. We believe expansions to other languages such as LLVM intermediate representation, and other types of binary code such as MIPS binaries and other types of firmware binaries, would be successful.

We performed some initial experiments translating binary machine code snippets to LLVM intermediate representation (IR). Although our initial LLVM IR results were not as good as the results for translation to C source, the translations showed promise. We believe there may be several reasons for the difference in performance. We used a very simple tokenization scheme for the LLVM IR data, while we used a much more sophisticated tokenization scheme for the C data. For example, we did not do the equivalent in LLVM IR of normalizing variable and function names in the C data. Anecdotally, we observed that the normalization improves results in the C data.

We would like to see further work in using an RNN-based technique for decompilation at a larger level. Our work is limited to small snippets of code because of the technical limitations of our setup and hardware, but we believe that similar techniques can be applied to larger snippets. Furthermore, we believe it is worth investigating whether a modified version of the approach can lead to recovery of higher-level program structure from binary. This higher-level approach, used together with the low-level approach discussed in this paper, may allow decompilation of much larger program units.

Acknowledgments

This material is based upon work supported by the SPAWAR Systems Center Pacific Office under Contract No. N66001-13-C-4046. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this article are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] L. Ďurfina, J. Křoustek, and P. Zemek, "Psybot malware: A step-by-step decompilation case study," in *Working Conference on Reverse Engineering*, ser. WCRE '13. IEEE Computer Society, 2013, pp. 449–456.

[2] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *Security and Privacy*, ser. SP '16, 2016, pp. 158–177.

[3] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," *International Conference on Information Systems Security*, pp. 1–25, 2008.

[4] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, 2016.

[5] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Network and Distributed System Security*, ser. NDSS '15, 2015.

[6] H. E. T. Siegelmann, "Foundations of recurrent neural networks," Ph.D. dissertation, Rutgers University, New Brunswick, NJ, USA, 1993, uMI Order No. GAX94-12680.

[7] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *Computing Research Repository*, 2014. [Online]. Available: http://arxiv.org/abs/1406.1078

[8] O. Vinyals, L. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. E. Hinton, "Grammar as a foreign language," *Computing Research Repository*, 2014. [Online]. Available: http://arxiv.org/abs/1412.7449

[9] J. Schmidhuber, "Deep learning in neural networks: An overview," *CoRR*, vol. abs/1404.7828, 2014. [Online]. Available: http://arxiv.org/abs/1404.7828

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *Operating Systems Design and Implementation*, ser. OSDI '16, 2016, pp. 265–283.

[11] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '13, 2013, pp. 651–654.

[12] W. Ling, E. Grefenstette, K. M. Hermann, T. Kociský, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *Computing Research Repository*, 2016. [Online]. Available: http://arxiv.org/abs/1603.06744

[13] C. J. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*, ser. ICML '14, 2014, pp. II–649–II–657.

[14] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," *Computing Research Repository*, 2016. [Online]. Available: http://arxiv.org/abs/1608.02715

[15] T. V. Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen, and T. N. Nguyen, "Combining word2vec with revised vector space model for better code retrieval," in *International Conference on Software Engineering Companion*, ser. ICSE-C '17, 2017, pp. 183–185.

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, ser. NIPS '14, 2014, pp. 3104–3112.

[18] "Tutorial: Sequence-to-sequence models." [Online]. Available: https://www.tensorflow.org/tutorials/seq2seq

[19] S. Jean, K. Cho, R. Memisevic, and Y. Bengio, "On using very large target vocabulary for neural machine translation," *Computing Research Repository*, 2014. [Online]. Available: http://arxiv.org/abs/1412.2007

[20] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[21] T. Mikolov and G. Zweig, "Context dependent recurrent neural network language model." in *SLT*, 2012, pp. 234–239.

[22] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna, "Design of a retargetable decompiler for a static platform-independent malware analysis," in *Information Security and Assurance*, ser. ISA '11, 2011, pp. 72–86.

[23] ——, "Design of an automatically generated retargetable decompiler," in *Circuits, Systems, Communications & Computers*, ser. CSCC '11, 2011, pp. 199–204.

[24] G. Chen, Z. Qi, S. Huang, K. Ni, Y. Zhen g, W. Binder, and H. Guan, "A refined decompiler to generate C code with high readability," *Software: Practice and Experience*, vol. 43, no. 11, pp. 1337–1358, 2013.

[25] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *USENIX Conference on Security*, ser. SEC '13, 2013, pp. 353–368.

[26] N. Hasabnis and R. Sekar, "Lifting assembly to intermediate representation: A novel approach leveraging compilers," in *Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, 2016, pp. 311–324.

[27] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "Big Code"," in *Principles of Programming Languages*, ser. POPL '15, 2015, pp. 111–124.

[28] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated JS names," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '17, 2017, pp. 683–693.

[29] A. Caliskan-Islam, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, "When coding style survives compilation: De-anonymizing programmers from executable binaries," *Computing Research Repository*, 2015. [Online]. Available: http://arxiv.org/abs/1512.08546

[30] D. Levy and L. Wolf, "Learning to align the source code to the compiled object code," in *International Conference on Machine Learning*, ser. ICML '17, 2017, pp. 2043–2051.

[31] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Knowledge Science, Engineering and Management*, ser. KSEM '15, 2015, pp. 547–553.

[32] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (T)," in *Automated Software Engineering*, ser. ASE '15, 2015, pp. 574–584.

[33] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014, 2014, pp. 173–184.

[34] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *Computing Research Repository*, 2014. [Online]. Available: http://arxiv.org/abs/1409.0473