DIRECT : A Transformer-based Model for Decompiled Variable Name Recovery

Vikram Nitin * **Anthony Saieva** * **Baishakhi Ray** **Gail Kaiser**

Department of Computer Science, Columbia University

vikram.nitin@columbia.edu, {ant,rayb,kaiser}@cs.columbia.edu

Abstract

Decompiling binary executables to high-level code is an important step in reverse engineering scenarios, such as malware analysis and legacy code maintenance. However, the generated high-level code is difficult to understand since the original variable names are lost. In this paper, we leverage transformer models to reconstruct the original variable names from decompiled code. Inherent differences between code and natural language present certain challenges in applying conventional transformer-based architectures to variable name recovery. We propose DIRECT, a novel transformer-based architecture customized specifically for the task at hand. We evaluate our model on a dataset of decompiled functions and find that DIRECT outperforms the previous state-of-the-art model by up to 20%. We also present ablation studies evaluating the impact of each of our modifications. We make the source code of DIRECT available to encourage reproducible research.

1 Introduction

Proprietary software often comes in binary form, making it difficult to comprehend its functionality, as many high-level code abstractions (e.g., meaningful variable names, code structures, etc.) are lost when source code is compiled to binaries. To extract meaningful information from binaries, software analysts typically use reverse engineering that converts binary executables into another form that can be more easily comprehended (Ďurfina et al., 2013). Reverse engineering is often applied in binary code inspection, legacy software maintenance, malware analysis, and cyber forensics. For example, reverse engineering uncovered the rebirth of ZeUS malware variants during the coronavirus pandemic of 2020 (Osborne, 2020).

* Equal contribution

```
void __fastcall add_match(char *a1) {
  // ... var declarations omitted ...
  v1 = (int)(a1 - 1);
  while ( 1 ) {
    v3 = *(unsigned __int8 *)(v1++ + 1);
    v2 = v3;
    if ( !v3 ) break;
    v4 = v2 > 0x7F;
    if ( v2 != 127 )
      v4 = v2 > 0x1F;
    if ( !v4 ) {
      free(a1);
      return;
    }
  }
  // ... some code omitted ...
}
```

Figure 1: Real world Hex-Rays decompilation. Reconstructed source differs significantly from original, and it is hard to deduce original developers' intentions.

Traditionally, the primary reverse engineering tools are *disassemblers*, which extract assembly instructions from a binary executable. However, in recent years, *decompilers* like Ghidra (Ghidra) and Hex Rays (Hex-Rays) have become practical and popular. They produce a source code-like approximation of the binary code as shown in Figure 1. While these tools can retrieve the approximate code structure, they introduce variable names that have no semantic meaning, drastically reducing code readability and comprehensibility (Katz et al., 2018; Hu et al., 2018; Hayati et al., 2018).

In recent years, Machine Learning-based models have shown promise in recovering lost variable names from decompiled code using a frequency-based model (He et al., 2018) or LSTMs (Lacomis et al., 2019). However, variables in source code are not independent of each other and often have hidden long-range dependencies. LSTMs and frequency-based models are not well-suited to capture such dependencies. Since transformer-based models can more effectively capture long-range dependencies (Vaswani et al., 2017), in this work we explore transformer-based models to recover variable names from decompiled code.

Transformers are popular in natural language processing (Vaswani et al., 2017; Devlin et al., 2019; Yang et al., 2019). However, code differs from natural language in many significant ways (Allamanis et al., 2018; Ding et al., 2020), hence vanilla transformer architectures need modifications for practical application to the task of variable recovery. Consider the following problems:

Unknown number of tokens to be predicted: Transformers that capture bidirectional context usually predict a known number of tokens, but to make the vocabulary size manageable, identifiers must be split into subtokens. For example, an identifier like `my_var` could be split up as three subtokens - "`my`", "`_`", and "`var`". Each identifier can be comprised of an arbitrary number of subtokens, and the model needs to access the information contained in the entire sequence while predicting the name for an identifier. To deal with this problem, we use an encoder-decoder transformer architecture as in (Ahmad et al., 2021).

Syntactic constraints: Unlike natural language, code's strict syntax requires that a variable assigned a name at one occurrence in the prediction must be the same at all other occurrences. For example, if a decompiled identifier name `v1` appears on line 3 and line 100, the predicted name must be the same on both lines. We propose a novel algorithm that uses the joint probability over sequences to predict variable name identifiers while still obeying constraints imposed by the code syntax.

Token Non-uniformity: While training a model for natural language, all tokens are usually given equal importance (Vaswani et al., 2017). However, for semantic understanding of code the identifier tokens are more important than those tokens that are built into the language syntax. For example, a variable name like "`click_count`" provides much more sematic information than a keyword like "`while`". We propose a token weighting scheme specially crafted for the variable name recovery problem.

Code sequences are long: Adaptations of NLP techniques to code often consider functions analogous to sentences. Traditional transformers limit the maximum sequence size to a few hundred tokens. While this restriction rarely presents a problem dealing with sentences, many functions are much longer. For example, the longest function in our benchmark dataset (Section 4.1) is over 4000 tokens long. To handle longer functions we propose a mechanism to break long sequences into smaller pieces and recombine their individual predictions while still obeying code's syntactic constraints.

Putting all these together, we propose DIRECT (**D**ecompiled **I**dentifier **R**enaming **E**ngine using **C**ontextual **T**ransformers), the first transformer-based model built specially for variable recovery from decompiled binaries. We compare DIRECT to DIRE (Lacomis et al., 2019), the state of the art in variable name recovery on a benchmark dataset and show that DIRECT improves on the baseline by 20%. We also evaluate the individual impact of each of our specific adaptations by performing a series of ablation studies. We provide the source code for DIRECT [1] in the hope that it will prove to be a useful tool for other researchers.

2 Related Work

Variable Name Recovery: DIRE (Lacomis et al., 2019), compared to in the evaluation, performs the same task as DIRECT but uses traditional LSTMs combined with GGNNs. DIRECT uses DIRE's tokenizer as is, our innovations replace DIRE's bidirectional LSTM with our task-specific transformer architecture. Prior to DIRE, Debin (He et al., 2018) represented the prior state of the art using decision tree-based modeling.

Type Inference: Debin also attempted to recover type information – which is a different problem. Typilus (Allamanis et al., 2020) is a new GGNN-based approach for type inference.

Function Name Recovery: An orthogonal decompilation problem is function name recovery. Function names are usually left in executables' metadata, by default, but in malware these symbols are probably stripped. Recent work by Artuso et al. (Artuso et al., 2020) has shown transformers are highly applicable to this task and the pre-training/fine-tuning paradigm has a place in code analysis, but they limit their experiments to function names. Other work like David et al. (David et al., 2020) uses LSTM architectures to encode API call sequences as function profiles and learned the function names commonly associated with those call sequences.

Transformers for Filling-in Blanks: Filling in blanks in an input sequence necessitates a model that can capture bidirectional context. BERT's pre-training objective (Devlin et al., 2019) solves

[1] https://github.com/DIRECT-team/DIRECT-nlp4prog

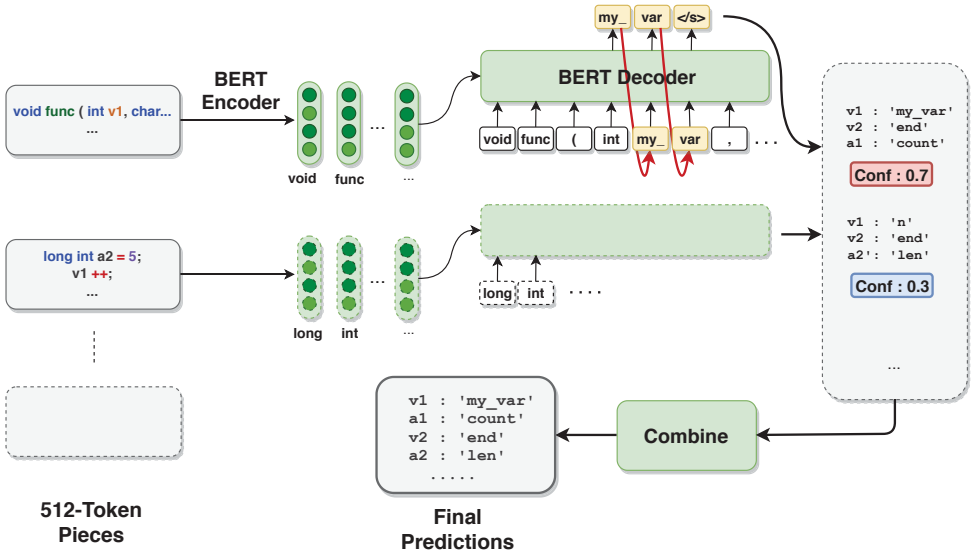

Figure 2: Our state-of-the-art variable renaming model, DIRECT. DIRECT breaks the function into pieces, passes each piece through a BERT encoder and decoder, and combines the predictions of all the pieces. For simplicity, we have omitted the advanced prediction algorithm (Algorithm 1; Figure 3) in this diagram. For more details, refer to Section 3.2.

this problem by reconstructing random masked tokens. SpanBERT (Joshi et al., 2020) focuses on contiguous spans of masked tokens with a modified pre-training objective. These methods require the location and length of each blank to be known in advance, but Insertion Transformers (Stern and Uszkoreit, 2019) solve for variable-length blanks without explicitly controlling insertion.

Blank Language Models (Shen et al., 2020) solve for fixed blanks with variable length with a special blank character that the model can predict and feed back in a loop. Another architecture that solves the same problem is BART (Lewis et al., 2020). Similar to us, BART uses a BERT encoder and a left-right decoder to perform arbitrary transformations on the input. However both these approaches cannot be directly applied to variable renaming without modification to guarantee that multiple blanks have the same prediction.

Decompilers: There are two decompilers used in practice. One is Hex-Rays (Hex-Rays), from which the training set was built, and the other is the open-source Ghidra platform (Ghidra), which both fail to make meaningful efforts at reconstructing variable names without debugging information. A research compiler DREAM++ (Yakdan et al., 2016) function signature heuristics to generate meaningful variable names, but does not apply ML models.

Adapting ML to SE tasks: Recent works like (Rahman et al., 2019) and (Ding et al., 2020) have also investigated the difficulties of applying ML models from other disciplines directly to software engineering tasks.

3 Design

Figure 2 provides an overview of DIRECT. In this section we detail each of the problems we encountered and the design decision solutions.

3.1 Encoding/Decoding

Transformers are traditionally used to predict entire sequences; however in our problem setting most tokens are fixed. Therefore we need to adapt transformers from predicting entire sequences to predicting individual tokens based on the fixed tokens.

While making a prediction on an occurrence of a particular variable, the model should ideally have access to the information contained in the entire input sequence. The naive solution is to use a bidirectional transformer that with a Masked Language Model (MLM) training scheme, such as BERT. However by design, an MLM is designed to predict the same number of tokens as in the input sequence. In our case, because of subtokenization, the predicted subsequences can be of unknown length. Adapting an MLM transformer to solve this problem is non-trivial.

The next option is to use a transformer as a

sequence-to-sequence language model to predict the immediate next token given all the preceding tokens. One could feed the entire sequence until a variable is reached, start generating tokens one at a time in an autoregressive manner, and stop when a special end token is predicted. However such a model cannot use bidirectional context while making a prediction, and can only leverage the part of the sequence that precedes each variable.

We propose to use an encoder-decoder setup, as in (Vaswani et al., 2017). The transformer encoder embeds each input token, and the sequential decoder attends over these encoder embeddings while making predictions one token at a time. So although we give the decoder only the portion of the input sequence that precedes the variable of interest, it also has access to the entire input sequence through the encoder embeddings.

Of course, this still leaves open the question of how to constrain multiple instances of the same variable to have the same prediction. While we will present a better solution to this problem in Section 3.2, a good first approximation is to simply use the prediction at the first occurrence of the variable we are interested in. We hypothesize that since the encoder-decoder model has access to the entire sentence while making a prediction for each occurrence, one cannot do drastically better than this simple approximation.

3.2 Advanced Prediction Algorithm

Effective sequence modeling requires not only making predictions, but also predictions that fit the problem setting (Ding et al., 2020). Semantic preserving identifier renaming mandates that once a variable has been renamed it must have the same value at each occurrence. This additional constraint poses a challenge for vanilla transformers since they predict each token independently in traditional language modeling. Exhaustively searching the target vocabulary space is computationally intractable, so we narrow the search space with a specialized prediction algorithm that fits the problem setting.

At the variable's first occurrence, we make m predictions for its name, each of which leads to a different sequence of variable name assignments. Throughout our algorithm, we maintain the top k sequences only. Thus at the first occurrence of a variable, we generate $m \times k$ possible sequences, and pick the top k. In practice, we use $m = k$.

At later occurrences of a variable, we update the scores of the existing predictions, thus maintaining the list of k sequences. This is where our algorithm differs from standard beam search. Note that the predictions made at the first occurrence of a variable constrain its predictions at further occurrences, but choosing a large k mitigates this problem.

This procedure, "Advanced prediction", is shown in Figure 3 for the case when $k = 2$. Algorithm 1 describes it in detail. In our experiments, we observed that choosing $k = 5$ was optimal.

Algorithm 1 Advanced Prediction

Input : A sequence of decompiler output tokens S, and a model M

Output : S with predicted names

$gen \leftarrow [[\,]]$, $probs \leftarrow [1]$

for $tok \in S$ **do**
  if tok is not a variable **then**
    for $seq \in gen$ **do**
      seq.append(tok)
    continue
  if tok has been seen before **then**
    for $j \in 1...\text{len}(gen)$ **do**
      $n \leftarrow$ current pred of tok in $gen[j]$
      $p \leftarrow$ prob assigned to n by M at the current position
      $gen[j] \leftarrow gen[j] + p$
      $probs[j] \leftarrow probs[j] \times p$
  else
    for $j \in 1...\text{len}(gen)$ **do**
      Using beam search over subtokens with M, find the top k possibilities for the name of tok
      Let the names be $n_1, ..., n_k$ and their probabilities be $p_1, ..., p_k$
      Replace $gen[j]$ with $(gen[j] + n_1), ..., (gen[j] + n_k)$
      Replace $probs[j]$ with $(probs[j] \cdot p_1), ..., (probs[j] \cdot p_k)$
  Sort gen and $probs$ in desc. order of $probs$
  $gen \leftarrow gen[1:k]$
  $probs \leftarrow probs[1:k]$
return $gen[1]$

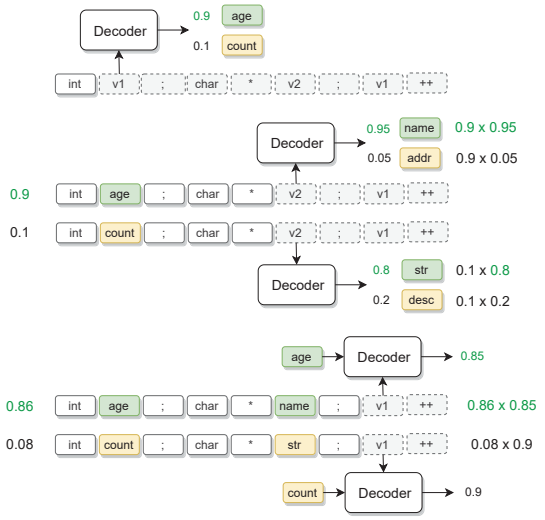

Figure 3: Advanced Prediction with $k = 2$. The decoder takes as input the portion of the sequence that *precedes* the variable being predicted. Our algorithm differs from standard beam search in the prediction of the second occurrence of `v1`. Rather than generate multiple predictions for `v1`, the algorithm simply updates the scores of the existing predictions in order to obey the syntactic constraints of code.

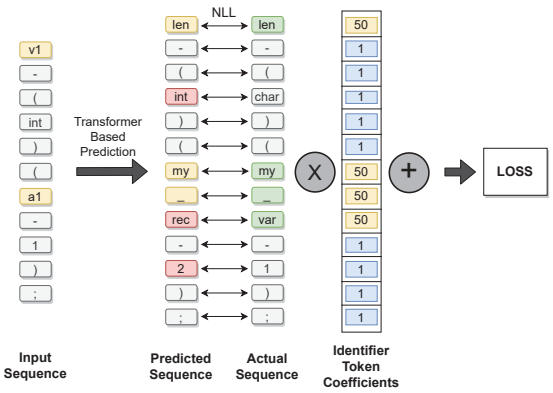

Figure 4: Identifier Token Coefficient loss function. The Negative Log Likelihood (NLL) loss is computed for each token, and a weighted sum taken to compute the loss.

3.3 Identifier Token Coefficient

A typical transformer treats all tokens identically when computing the loss function during pre-training and fine-tuning. Code differs from natural language in the grammar requires the majority of the tokens. The only opportunity for the programmer to inject semantic meaning into the source code text is through identifiers, which makes this problem compelling in the first place. The model should therefore treat identifier tokens differently.

We implement this concept by training with a custom loss function as shown in Figure 4. Traditional NLP architectures predict the entire sequence, and then train on a loss function by averaging the error uniformly across all tokens. Our custom weighting scheme places increased significance on prediction of identifiers, using a mask which increases the loss 50-fold for identifiers as compared to all other tokens. We expect that this *identifier token coefficient* (ITC) hyper parameter could be tuned in the future for better performance.

Predicting the identifiers and ignoring the rest of the characters in the sequence would result in a model that doesn't learn the context surrounding the identifier which informs the prediction.

3.4 Splitting and Merging Mechanism

Another inherent difference between code and natural language when considering sequence to sequence modeling is the length of the sequence. Discussion of natural language modeling overlooks this aspect since sentences rarely exceed 200 tokens. In code however functions are significantly longer so the ML models must support sequences of arbitrary length. In fact our benchmark dataset contains a small number of sequences with length greater than 2000. With respect to identifier recovery, longer sequences mean more variables to recover, multiple usages per variable, and more opportunity for errors. This poses a problem for transformers as traditional transformer based architectures, like BERT, require a maximum sequence length set in advance. Furthermore since attention must be trained across all tokens, the memory usage increases quadratically with sequence length.

In order to use our model for arbitrary sequence lengths, we developed a novel splitting and joint prediction mechanism. As described in Figure 2 we divide the sequence into multiple chunks of 512 tokens upon which the model predicts. A single variable can have a different prediction in each chunk we combine these predictions using the prediction at the first chunk in which a variable occurs.

We also tried using the chunk with the highest confidence, but we found that this did not perform as well. We suspect this is because the probabilities are less than one, and multiplications with each successive variable only decrease the probability of the entire sequence. Hence smaller pieces with perhaps just one or two occurrences of a variable will

be more confident in their predictions despite having less information. One could impose a penalty for pieces with fewer variables, but we defer this analysis to future work.

Other transformer variants can handle sequences arbitrary lengths like XLNet (Yang et al., 2019), and we expect these advanced models will handle this issue as well as present new challenges. We again leave these endeavors to future work.

3.5 DIRECT

Using the techniques from the previous sections, we put it all together to get DIRECT, a state-of-the-art variable renaming system. Given an input sequence, DIRECT splits it into pieces of length at most 512 each (default BERT architecture), and puts each piece through a BERT encoder and a BERT decoder with advanced prediction (Algorithm 1). Different predictions across pieces for the same variable are combined by taking the prediction of the first piece in the sequence that contains the variable. Figure 2 depicts the entire model.

4 Experimental Setup

4.1 Data

We use the dataset provided by DIRE (Lacomis et al., 2019). It was generated using C binaries from Github, which were then decompiled using Ida's Hex-Rays decompilation plugin (Hex-Rays). The training data set consists of 1,011,049 functions, with a median of 16 variables per function, a median of 4 *unique* variables per function, and a median sequence length of 150 subtokens. We follow DIRE and use Sentencepiece (Kudo and Richardson, 2018) to split the functions into subtokens.

We use only the "Body-not-in-train" subset for the validation and test data. They consist of 23662 and 24862 examples, respectively.

4.2 Metrics

We define *accuracy* as an exact match between the original variable name as determined by the debug information mapping, and the name predicted by DIRECT. We also examine the *edit distance* between predicted names and true names, and use the edit distance per number of characters (the character error rate) as our metric as in DIRE (Lacomis et al., 2019) to capture success of partial matches. We also measure the *Jaccard similarity* which is the ratio of the number of overlapping n-grams between two sequences to the total number of n-grams contained in them. We use n=1, so that each word is treated as a set of its constituent characters. There are some instances when decompiler variables have no corresponding true name. These are ignored from all metrics.

4.3 Pre-training Procedure

We pre-train one BERT model using the standard MLM task on source sequences directly from the decompiler output (with the dummy variable names from the decompiler). We call this the BERT encoder. Similarly we pre-train another BERT model using MLM on target sentences (with the true variable names), and call this the BERT decoder. Both models used 4 attention heads, 6 hidden layers, and a hidden embedding size of 256. We trained the encoder and decoder for 220k and 140k steps, respectively, using a batch size of 128 sequences. While masking tokens, we do not differentiate between variable and non-variable tokens since we want the model to learn the complete structure of the code sequences. We also used the standard optimization techniques employed by BERT (Devlin et al., 2019), wherein an Adam optimizer is used with a variable learning rate. The learning rate increases linearly from 0 to 10^{-4} over the "warm-up" period of 40k iterations, and then decreases linearly from 10^{-4} to 0 at the end of pre-training.

4.4 Fine-tuning Procedure

After reviewing our proof of concept experiments we trained our best configuration for 85 epochs to produce the DIRECT prototype. We follow the same convention as DIRE (Lacomis et al., 2019), whereby the number of sequences per batch is variable, but the total number of tokens in the batch is fixed to define the size of the batch. We used a batch size of 4096 tokens per batch. We used a learning rate of 1e-4 for the first 10 epochs, 0.3e-4 for the next 10, and 1e-5 thereafter.

5 Results

5.1 DIRECT Evaluation

In order to evaluate the effectiveness of DIRECT, we compare its performance against that of DIRE on our test dataset. The results are shown in Table 1. We observe that DIRECT achieves an increase of 7.1 percentage points in accuracy over DIRE, which is a relative increase of 19.9%. We obtained all DIRE results by re-running the authors' code on

Model	Accuracy (%) ↑	Top-5 Accuracy (%) ↑	CER ↓	Jaccard Dist ↓
DIRE	35.8	41.5	.664	.537
DIRECT	**42.8**	**49.3**	**.663**	**.501**
Improvement	20%	19%	.2%	6.5%

Table 1: Test Accuracy, Top-5 Accuracy (computed by taking the top 5 predictions for each *sequence* and using the predictions of variables contained in these sequences), Character Error Rate and Jaccard distance of DIRE vs DIRECT. DIRECT outperforms DIRE on all four metrics. DIRE results are reproduced by re-running the authors' code on our dataset.

```
signed __int64 __fastcall prussdrv_open ( unsigned int
host_interrupt ) {
    char name ;
    unsigned __int64 @@v3@@ ;
    @@v3@@ = __readfsqword ( Number ) ;
    if ( dword_15A4 [ @@v1@@ ] ) return Number ;
    sprintf ( & name , String , @@v1@@ ) ;
    dword_15A4 [ @@v1@@ ] = open ( & name , Number ) ;
    return _prussdrv_memmap_init ( ) ;
    }

signed __int64 __fastcall prussdrv_open ( unsigned int
host_interrupt ) {
    char name ;
    unsigned __int64 @@v3@@ ;
    @@v3@@ = __readfsqword ( Number ) ;
    if ( dword_15A4 [ host_interrupt ] ) return Number ;
    sprintf ( & name , String , host_interrupt ) ;
    dword_15A4 [ host_interrupt ] = open ( & name ,
Number ) ;
    return _prussdrv_memmap_init ( ) ;
```

Figure 5: A visualization of the attention weights of the trained decoder while predicting variables. Darker represents larger weights. The variable subtokens that are being predicted are boxed. For more details, refer to Section 5.

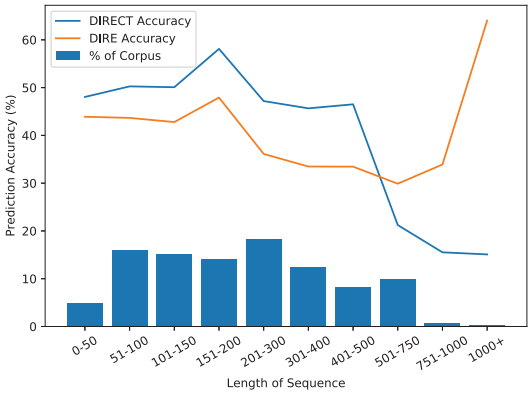

Figure 6: Variation of Accuracy of DIRECT and DIRE with length. The spike in DIRE's performance for the last two categories with very few examples is likely to be an anomaly and not representative of its true performance on sequences of those lengths. Note that this is on the validation set.

the dataset, rather than simply using the numbers from their paper.

5.2 Qualitative Analysis

We also perform some qualitative inspection of the attention weights of the trained model to understand what information it is using to make its inferences. An example of this is shown in Figure 5 where the predicted identifier is outlined in black. The attentions shown are the weights used while predicting a name for the variable shown in a box, averaged over all attention heads at the last layer of the decoder.

We observe that when making a prediction on the first occurrence of a variable, the decoder model pays attention mainly to the function header, more specifically the return type and function name. However for later occurrences of the same variable, although it does look at the function header to some extent, it relies chiefly on its predictions for earlier instances of the same variable.

5.3 Performance on Long Sequences

The graph in Figure 6 shows the accuracy of DIRECT on sequences of various lengths. As we cross the 500 token mark, and the splitting technique takes over, there is a steep drop in accuracy. This problem is mirrored in DIRE's accuracy although not quite as steeply. Still for sequences of length less than 512 tokens DIRECT has a improvement of 10 percentage points over DIRE (48.9% vs. 38.8%). DIRE has high accuracy in the longest two sets of sequences, but this is likely an anomaly caused by insufficient samples sizes.

Other transformer based variants address this sequence issue such as XLNet (Yang et al., 2019), and we expect these advanced models will handle this issue as well as present new challenges. We again leave these endeavors to future work.

Model	Accuracy (%) ↑	CER ↓
Uniform token weighting	30.0	.80
Weighting identifiers only	33.7	.76
ITC weighting scheme	**34.4**	**.75**

Table 2: Validation accuracy and Character Error Rate for various token weighting schemes. Prediction is done using the "first prediction" strategy. All the models are trained for 15 epochs. Refer to Section 3.3 for more details.

Model	Accuracy (%) ↑	CER ↓
First pred	34.4	**.75**
Advanced pred	**34.6**	**.75**

Table 3: Validation accuracy and Character Error Rate for advanced prediction versus first prediction. Both models are trained for 15 epochs. Refer to Section 3.2 for more details.

Model	Accuracy (%)	CER
Decoder Only	19.6	.97
Encoder-Decoder	**34.4**	**.75**

Table 4: Validation accuracy and Character Error Rate for our encoder-decoder model versus a decoder-only model. Both models are trained for 15 epochs. Refer to Section 3.1 for more details.

5.4 Ablation Studies

In this section, we evaluate the impact of each of our design choices. We train all the models for 15 epochs and evaluate them on the *validation* set.

5.4.1 Encoder-Decoder Architecture

Table 4 shows the performance of our encoder-decoder model vs a decoder-only model (a single transformer, operating as an autoregressive language model) using the prediction at the *first* occurrence of each variable. As we can see, the decoder-only model does significantly worse, which is expected since it has access only to a part of the function while making a prediction at the first instance of a variable.

5.4.2 Advanced Prediction Algorithm

Table 3 compares the results of advanced prediction with "first prediction", i.e., taking the prediction at the first occurrence of a variable. We observe that advanced prediction improves the performance of our encoder-decoder model by a small amount.

This could be explained by our observation in Section 5.2 that the model seems to rely its earlier predictions while predicting the name of a particular variable. Later predictions of a variable refer to the value assigned at the first prediction, and so the prediction of a variable seldom changes from what was predicted at the first instance.

5.4.3 Identifier Token Coefficient

We compare the performance of three different token weighting schemes in the loss function - weighting all tokens uniformly, weighting according to ITC (as described in Section 3.3), and weighting the identifiers only while ignoring the rest of the tokens.

As seen in Table 2, ITC shows a 4.4% increase in accuracy relative to the uniform weighting scheme, *without* hyperparameter tuning of the coefficient. As expected the model that ignores the surrounding tokens in the loss function performs worse. This is because the model doesn't effectively learn the context surrounding the identifiers, resulting in a decrease in accuracy by 0.7 percentage points.

6 Conclusion and Future Work

The problem of variable name reconstruction poses certain challenges for traditional transformer-based models. Specifically, the variable length of the prediction target, the constraints imposed by code syntax, architecture limitations that make long sequences difficult, and the task specific non-uniformity of token significance. In this work, we developed a series of solutions to address these issues, namely 1) an encoding/decoding scheme to handle arbitrary sub-token length prediction, 2) a specialized prediction algorithm, 3) a customized identifier token coefficient weighting scheme, and 4) a splitting and combining algorithm for standard transformers to handle sequences of arbitrary length. In addition to empirical studies evaluating the effectiveness of each of these techniques, we also combined them to create DIRECT, a practical open-sourced identifier renaming engine. We evaluated DIRECT using a standard benchmark dataset against the state of the art, DIRE (Lacomis et al., 2019), and found that DIRECT provides a 20% improvement. We hope that in addition to an open-sourced tool, this work functions as a roadmap for

other researchers trying to solve the types of problems we encountered when adapting transformer-based models to code analysis tasks. Future work could leverage the Abstract Syntax Tree (AST) of each function, and employ new transformer architectures like XLNet (Yang et al., 2019) to avoid splitting up the input while handling longer sequences. Our approach might also improve the results of other code analysis tasks like type inference, function re-naming, docstring prediction, and function boundary identification.

7 Acknowledgements

This work was supported in part by DARPA N6600121C4018, NSF CCF-1815494, NSF CNS-1563555, NSF CCF-1845893, NSF CCF-1822965, NSF CNS-1842456. We thank the anonymous reviewers for their helpful feedback. We would also like to thank Suman Jana for his generous provision of computing resources.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 91–105.

Fiorella Artuso, Giuseppe Antonio Di Luna, Luca Massarelli, and Leonardo Querzoni. 2020. In Nomine Function: Naming Functions in Stripped Binaries with Neural Networks. *arXiv preprint arXiv:1912.07946*.

Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural Reverse Engineering of Stripped Binaries. *arXiv preprint arXiv:1902.09122*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *17th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 4171–4186.

Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. 2020. Patching as translation: the data and the metaphor. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 275–286. IEEE.

Ghidra. 2021. GHIDRA, A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission. https://ghidra-sre.org/.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680.

Hex-Rays. 2021. Hex-Rays Decompiler. https://hex-rays.com/decompiler/.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. 2020. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77.

Deborah S Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356. IEEE.

Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.

Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628—-639.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

Charlie Osborne. 2020. Zeus sphinx revamped as coronavirus relief payment attack wave continues. https://tinyurl.com/ka2t6k2r.

Md Masudur Rahman, Saikat Chakraborty, Gail Kaiser, and Baishakhi Ray. 2019. Toward Optimal Selection of Information Retrieval Models for Software Engineering Tasks. In *19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 127–138.

Tianxiao Shen, Victor Quach, Regina Barzilay, and Tommi Jaakkola. 2020. Blank Language Models. *arXiv preprint arXiv:2002.03079*.

Kiros Stern, Chan and Uszkoreit. 2019. Insertion transformer: Flexible sequence generation via insertion operations. https://arxiv.org/abs/1902.03249.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *31st Conference on Neural Information Processing Systems (NIPS)*.

Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *IEEE Symposium on Security and Privacy (S&P)*, pages 158–177.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *33rd Conference on Neural Information Processing Systems (NIPS)*.

Lukás Ďurfina, Jakub Křoustek, and Petr Zemek. 2013. Psybot malware: A step-by-step decompilation case study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 449–456.