Analyzing, Understanding, and Improving Variable Name Prediction in Decompiled Binary Code

by

Ati Priya Bajaj

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2021 by the
Graduate Supervisory Committee:

Ruoyu Wang, Chair
Chitta Baral
Yan Shoshitaishvili

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

Reverse engineers use decompilers to analyze binaries when their source code is unavailable. A binary decompiler attempts to transform binary programs to their corresponding high-level source code by recovering and inferring the information that was lost during the compilation process. One type of information that is lost during compilation is variable names, which are critical for reverse engineers to analyze and understand programs. Traditional binary decompilers generally use automatically generated, placeholder variable names that are meaningless or have little correlation with their intended semantics. Having correct or meaningful variable names in decompiled code, instead of placeholder variable names, greatly increases the readability of decompiled binary code. Decompiled Identifier Renaming Engine (DIRE) is a state-of-the-art, deep-learning-based solution that automatically predicts variable names in decompiled binary code. However, DIRE's prediction result is far from perfect. The first goal of this research project is to take a close look at the current state-of-the-art solution for automated variable name prediction on decompilation output of binary code, assess the prediction quality, and understand how the prediction result can be improved. Then, as the second goal of this research project, I aim to improve the prediction quality of variable names. With a thorough understanding of DIRE's issues, I focus on improving the quality of training data. This thesis proposes a novel approach to improving the quality of the training data by normalizing variable names and converting their abbreviated forms to their full forms. I implemented and evaluated the proposed approach on a data set of over 10k and 20k binaries and showed improvements over DIRE.

ACKNOWLEDGMENTS

Every individual needs a positive environment to grow. This environment was provided to me by Fish, who is an awesome mentor. I am extremely grateful to him for introducing me to the fascinating field of binary analysis, encouraging and supporting me throughout my Masters. From him, I learned what good research looks like. I would like to thank my committee members Chitta and Yan for their invaluable support and enthusiasm for my work. Lastly, I would like to thank my family and friends for always being there for me and keeping up with my research.

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND 3

2.1 Compilation 3

2.2 Disassembly of Binary Code 4

2.3 Decompilation of Binary Code 4

2.3.1 Decompilation Challenges 5

3 EXISTING SOLUTIONS 7

3.1 DEBIN 7

3.2 DIRE 7

3.2.1 DIRE's Dataset 8

4 IMPERFECT DIRE 11

4.1 DIRE's prediction 12

4.2 DIRE is better at recognition than prediction 14

4.3 Incorrect and Unnatural Prediction 14

4.4 Implementation Issues 17

5 IMPROVING TRAINING DATASET 19

5.1 Observation 19

5.2 Insight 19

5.3 Improvements 19

5.3.1 Unifying IDA-generated Variable Names 20

5.3.2 Converting abbreviations to their full forms 20

CHAPTER Page

6 EVALUATION . . . 23
6.1 Experiments . . . 23
6.2 Evaluation Results . . . 24
6.3 Case Study . . . 25
7 LIMITATIONS . . . 26
8 FUTURE WORK . . . 27
9 CONCLUSION . . . 28
REFERENCES . . . 29

LIST OF TABLES

Table | Page

3.1 DIRE versus DEBIN. Data in this table comes from the DIRE paper Lacomis *et al.* (2019a). 8

4.1 Total and unique count of variables and IDA-generated variables in DIRE's Dataset 12

4.2 Top 15 Variables in DIRE's Dataset 14

4.3 Prediction accuracy of all the variables versus IDA-generated variables when the function containing these variables were in training set versus not in training set 14

4.4 DIRE's Incorrectly Predicted Variable Names 15

4.5 DIRE's Invalid Predicted Variable Names 15

4.6 DIRE's Unnatural Predicted Variable Names 16

4.7 DIRE's Incorrectly Predicted Variable Names 16

5.1 Variable Transformation Based on Lookup 21

5.2 Preprocessing Pipeline. First normalize variable names then convert abbreviated variable names to their full form 22

6.1 Comparison of Model's accuracy on $1/16^{th}$ data and $1/8^{th}$ data of original dataset 25

LIST OF FIGURES

Figure Page

3.1 The relation between DIRE's prediction accuracy and the number of variables in a function. Generally, the more variables there are in a function, the lower prediction accuracy there is. 9

3.2 Number of variables in functions that are in DIRE's data set. 9

3.3 Accuracy versus functions versus the number of variables in each function. The variable name prediction accuracy decreases as the number of variables in a function grows. 10

6.1 Accuracy Versus Size of Training Set. Accuracy increases with increase in the size of training set Lacomis *et al.* (2019a) 23

Chapter 1

INTRODUCTION

Reverse engineering is the process of analyzing a program without access to the source code. Reverse engineers greatly benefit from decompilers as they translate binaries to a high-level language. Decompilers like Hex-Rays Cit (2019), and Ghidra NSA (2019) try to use the associated debug symbols (if they exist) for reconstruction. However, they fail to recover most of the rich source-level information, such as variable names, data types, and structural information that is lost during compilation.

When analyzing Commercial off-the-shelf (COTS) binaries, finding vulnerabilities Yakdan *et al.* (2015); Emmerik (2007); Brumley *et al.* (2013), patching bugs in legacy software Emmerik (2007); Brumley *et al.* (2013), and performing malware research, decompilers come in handy. Meaningful variable names would greatly enhance the readability of the decompiled code and make the decompiled code easier to understand for reverse engineers. Because variable names embed semantics inside, with meaningful variable names, it is easy for a human analyst to infer the actual meaning of the function where the variable is. For example, if there is a variable named "password" in decompiled code, a human analyst can naturally make the inference that this variable is used to store passwords or password-related strings. Further, maybe the function where this "password" variable belongs is related to authentication. In conclusion, meaningful variable names in decompiled binary code helps human analysts with reverse engineering tasks by saving the amount of time and effort that is required for understanding decompiled code.

Using deep learning approach, the state-of-the-art solution DIRE Lacomis *et al.* (2019a) has pioneered in direction of predicting variable names in decompiled binary

code. In their approach, they rely on structural and lexical information of decompiled binary code that is recovered from Hex-Rays. For structural information, DIRE uses the internal AST representation of the decompiled binary. In addition, they created a dataset of 164,632 unique x86-64 binaries generated from GitHub's C projects. Their results, however, after careful analysis, show that their dataset and predictions are imperfect.

DIRE is an open-source project, which was really helpful in analysis. Approximately half of their dataset has IDA generated variables, which are meaningless. Moreover, the predicted variable names are not only incorrect but are also invalid such as frac_Q ? or _ ??. Another problem with DIRE is that it performs better on functions that the model was trained on. Meaning, if the model has never seen a function before, the prediction accuracy drops to 17%. Further, it has an implementation issue that inflates the accuracy of the model.

After a thorough understanding of their approach, I try to improve the quality of their dataset with an aim to improve the accuracy of the model. The improvements result from observations as a programmer and the insight that abbreviated variable names are treated as different variables though they may be semantically the same. The first improvement is an attempt to improve IDA-generated variables, which are in large amounts in the dataset. The second improvement aims to have full form of variable names instead of abbreviated variable names and normalize them. In addition to the more meaningful variable names, these improvements will keep the vocabulary small.

The next, step is to evaluate the proposed improvements. To evaluate the model, I train three different models based on the two improvements, as well as a reference model of DIRE. Keeping in mind the author's study of the impact of the training dataset on the accuracy, I train the models on $1/16^{\text{th}}$ of data and $1/8^{\text{th}}$ of data.

Chapter 2

BACKGROUND

Before jumping directly into the variable name prediction, let's start with some basics and understand how variable names are lost and recovered in the first place. Any discussion about decompiled binary code would be incomplete without discussing compilation.

2.1 Compilation

The Journey of C source code to a binary is a multi-step process. First comes the preprocessor, which removes the comments marked by /* */ or // from the source code. In the next step, it expands all header files (example: #include stdio.h) that are specified in the beginning of the C file . The final step in preprocessing is replacing the macros with their values (example: #define BUFFER_SIZE 1024).

Compiling follows preprocessing; the compiler takes the expanded code and generates equivalent platform-specific assembly code. During this step, the compiler performs various code optimizations.

At the third stage, assembly code is translated into relocatable machine code by the Assembler Cit (1997). Assemblers assign memory locations to all instructions and variables using offsets; there is also a list of unresolved references for the linker to resolve. The final step of compilation is linking. Linker links together object files and libraries to form an executable, assigning absolute memory locations and resolving unresolved references.

2.2 Disassembly of Binary Code

The disassembler works in the opposite way to an assembler, converting the binary code to assembly code. IDA and GNU Debugger (GDB) Cit (1986) are two widely used disassemblers.

```
00001189  endbr64
0000118d  push    rbp
0000118e  mov     rbp, rsp
00001191  sub     rsp, 0x20
00001195  mov     rax, qword [fs:0x28]
0000119e  mov     qword [rbp-0x8 ], rax
000011a2  xor     eax, eax
000011a4  lea     rdi, [rel data_2004]
000011ab  mov     eax, 0x0
000011b0  call    printf
```

Listing 1: Disassembled Code

2.3 Decompilation of Binary Code

The decompilation process is complex and challenging, as most of the information from the source code is lost during compilation. A decompiler attempts to reverse the compilation process, i.e. to convert the binary code into C-like code.

First, the binary code is disassembled to assembly code, which is then converted to Intermediate Representation. The decompiler attempts to recover variable names, types, and control flow structure by using program analysis. The decompiler produces an Abstract Syntax Tree (AST), which is converted to a high-level source code.

```
__int64 __fastcall add_new_symbol(__int64 a1, const char
   symname)
{
    int v2; \/\/ ebx
    char **v3; \/\/ rbp
    __int64 result; \/\/ rax
    *(_QWORD *)(a1 + 8)= realloc(*(void **)(a1 + 8),8LL
       *(*(_DWORD *)(a1 +16)+ 1));
    v2 = *(_DWORD *)(a1 + 16);
    v3 = (char **)(*(_QWORD *)(a1 + 8) + 8LL * v2);
    v3 = strdup(symname);
    result = a1;

    *(_DWORD *)(a1 + 16) = v2 + 1;
    return result;
}
```

Listing 2: Decompiled Code

Ghidra and Hex-Rays (often referred to as IDA) are widely used decompilers.

2.3.1 Decompilation Challenges

Compilation strips important information like variable names, data types, and control flow structure so it is difficult for a decompiler to reconstruct this lost information. As seen in the example of decompiled code (Listing 2), decompilers automatically generate placeholder values for variable names such as `v2` and `a1`. These variable names make little sense and have little relationship with their intended mean-

ing. Having a meaningful variable name enhances the readability and usability of the code.

Chapter 3

EXISTING SOLUTIONS

3.1 DEBIN

DEBIN He *et al.* (2018) predicts debug information in Stripped binaries. Using machine learning, they train model on non stripped binaries and predict debugging information like identifiers, types and names. DEBIN works in two stages. In the first step, it predicts variable's memory location, and then it predicts the variable for all recovered memory locations. However, the locations of the predicted variables do not match with their location in the original binary. DEBIN is no longer a state-of-the-art solution.

3.2 DIRE

The current state-of-the-art solution is DIRE: Decompiled Identifier Renaming Engine Lacomis *et al.* (2019a). DIRE uses deep learning, specifically encode-decoder model to predict variable names in the decompiled binary code. DIRE relies on lexical and structural information recovered by the decompiler. Recurrent Neural Networks (RNNs) learn from input sequence, hence they are used by DIRE to extract lexical information from the decompiled binary code in form of source tokens. For structural information DIRE uses Abstract Syntax Trees (AST) from Hex-Rays decompiler. It uses Gated-Graph Neural Networks (GNNs) for learning the structure. In their paper, the authors of DIRE talk about their analysis of DEBIN. Since DIRE is considered state-of-the-art, in my assessment, I will only focus on DIRE.

Accuracy	DIRE (1%)	DEBIN (1%)	DIRE (3%)	DEBIN (3%)
Overall	32.2%	2.4%	38.4%	3.9%
Body in Train	40%	3%	47.2%	4.8%
Body not in Train	5.3%	0.6%	8.6%	0.7%

Table 3.1: DIRE versus DEBIN. Data in this table comes from the DIRE paper Lacomis *et al.* (2019a).

As shown in Table 3.1, DIRE beats DEBIN in overall accuracy by roughly 30% if the size of corpus is 1% and by 34% when the size of corpus was 2%.

3.2.1 DIRE's Dataset

DIRE Lacomis *et al.* (2019a) open-sourced their dataset. For dataset creation, they mined C source code from GitHub. Every function in the dataset has three components. First is the tokenized code, where each variable name has unique `VAR_id`. Second is Abstract Syntax Tree (AST) with modified variable names. Third is a lookup that maps this unique `VAR_id` to IDA-given names and developer given names. To create this mapping, the authors first decompile the binary normally and collect IDA-given names. After the binary is decompiled they traverse AST and replace variable references with unique place holder tokens. Using this modified AST they generate decompiled code.

For developer given names, they decompile the binary once again, but this time the binary has associated debug DWARF Eager (2012) information with it. A decompiler behaves differently when it decompiles a stripped binary versus binary with debug information. First, the code structure may be different. Second, a decompiler generates intermediate variables therefore, there can't be complete mapping in IDA-given names and developer given names. To overcome this challenge the authors,

identify each variable by their instruction offset and operations.

They mined open-source C GitHub repositories and collected 164,632 unique binaries, which have 3,195,962 functions. They also open-sourced their pre-trained model, the training and testing split used at the time of training.

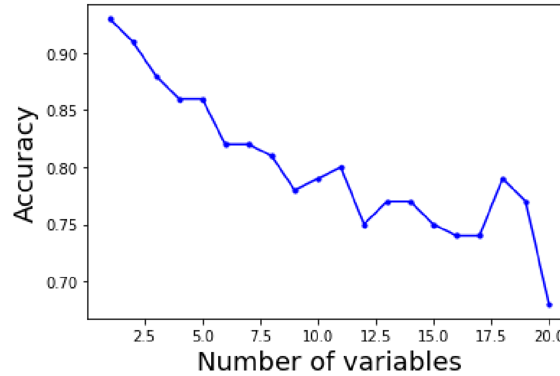

Figure 3.1: The relation between DIRE's prediction accuracy and the number of variables in a function. Generally, the more variables there are in a function, the lower prediction accuracy there is.

As shown in Figure 3.1, the x-axis represents the number of variables, while the y-axis shows the accuracy of all functions with their corresponding numbers of variables. The accuracy of the model decreases as the number of variables in a function increases.

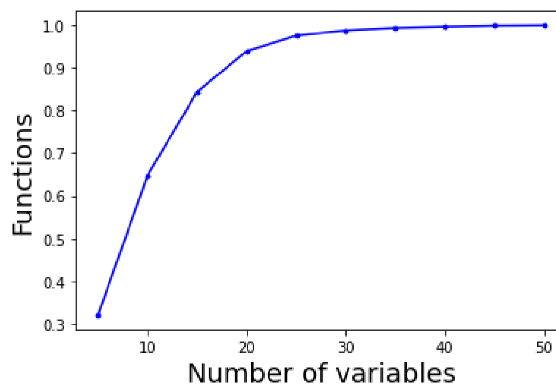

Figure 3.2: Number of variables in functions that are in DIRE's data set.

Y-axis represents the percentage of functions and x-axis represents the number of

variables in Figure 3.2. Functions that have up to 20 variables make 98.38% of the data.

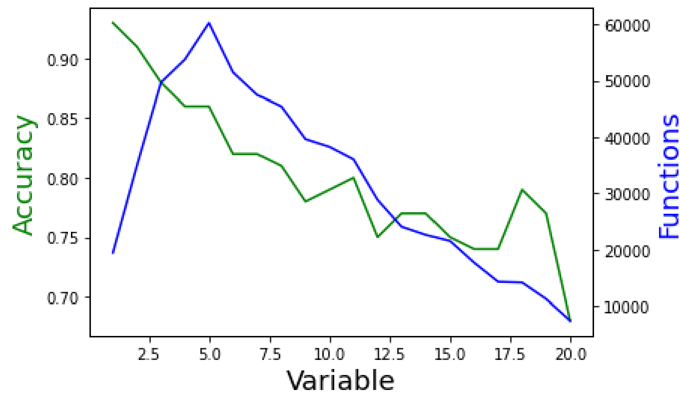

Figure 3.3: Accuracy versus functions versus the number of variables in each function. The variable name prediction accuracy decreases as the number of variables in a function grows.

Figure 3.3 depicts how accuracy, number of variables, and functions are related to each other. It shows a major portion of the dataset has functions with variables up to 20. With an increase in the number of variables the accuracy decreases. Smaller functions contribute significantly to the DIRE's accuracy.

Chapter 4

IMPERFECT DIRE

I performed careful evaluation on DIRE's results, and found four major issues that are impacting DIRE's variable name prediction performance, as listed below:

1. DIRE's variable name predictions are mostly a direct result of how Hex-Rays decompiler names not how humans name it. The predicted variable names are `v1`, `v2` or `a1`. These variables are all IDA-generated placeholders. A programmer would give meaningful variable names like `length` rather than using `v1` and `v2`. Since the model is trained on decompiled code with IDA-generated variables, it learns to predict IDA-generated names.

2. DIRE is better at recognizing functions than predicting variable names. If the model is trained on a particular function, the accuracy of variable prediction in such functions is higher than the accuracy of variable prediction in functions that were not present in the training set.

3. In addition to incorrect variable names, DIRE predicts unnatural variable names. These unnatural variable names are typically characters that are invalid as per the naming convention of a variable.

4. DIRE has implementation issues that lead to inflated prediction accuracy. As seen in decoder code, if the model fails to find a hypothesis or is unable to predict variable names, they directly replace the predicted variable with the original variable. They do not exclude these failed predictions when calculating accuracy. Hence, the accuracy is higher than it is.

4.1 DIRE's prediction

After taking a close look at the DIRE's prediction results, it is clear that DIRE predicts variable names as Hex-Rays decompiler not how humans would name them. Using program analysis and heuristics, Hex-Rays attempts to recover data lost during compilation, such as variable names, data types, and control flow structure. Therefore, the variable names are anything that the decompiler could recover. As for the rest of the variable names, the decompiler automatically generates placeholder names like `v1`, `v2`, `a1`, etc.

Due to the fact that all the functions of the DIRE's dataset are recovered from Hex-Rays, they contain a significant number of variable names such as `v1`, `v2`, and `a1`. A few variable names, such as `result`, `i`, and `s`, can also be recovered by Hex-Rays. The programmer also uses these variable names. Therefore, it is difficult to differentiate between variables generated by IDA and those given by a human. So I only consider the variables like `v1` and `a1` as IDA generated. An example function from the DIRE's dataset is shown above.

The data shows that **53**% of the total variables in DIRE's training set are IDA generated. Thus, when the deep learning model is trained on these variables, it learns how IDA names variable. Hence, most of the variables in their prediction are `v1`, `v2`, `v3` etc.

Variable	Total	Unique
All	88M	172K
IDA-generated	47M	2613

Table 4.1: Total and unique count of variables and IDA-generated variables in DIRE's Dataset

```
__int64 __fastcall add_new_symbol( __int64 @@VAR_73@@a1@@
   a1, const char *@@VAR_76@@a2@@symname)
{
  int @@VAR_74@@v2@@v2;
  char **@@VAR_75@@v3@@v3; \/\/ rbp

  __int64 @@VAR_77@@result@@result; \/\/ rax    *(_QWORD
   *)(@@VAR_73@@a1@@a1 + 8)= realloc(*(void **)(
   @@VAR_73@@a1@@a1 + 8),8LL *(*(_DWORD *)(@@VAR_73@@a1@@
   a1 +16)+ 1));

  @@VAR_74@@v2@@v2 = *(_DWORD *)(@@VAR_73@@a1@@a1 + 16);

  @@VAR_75@@v3@@v3 = (char **)(*(_QWORD *)(@@VAR_73@@a1@@
   a1 + 8) + 8LL * @@VAR_74@@v2@@v2);

  *@@VAR_75@@v3@@v3 = strdup(@@VAR_76@@a2@@symname);
  @@VAR_77@@result@@result = @@VAR_73@@a1@@a1;

  *(_DWORD *)(@@VAR_73@@a1@@a1 + 16) = @@VAR_74@@v2@@v2 +
   1;
  return @@VAR_77@@result@@result;
}
```

Listing 3: DIRE's function

Of the top 15 variables in the training set only `result`, `i` and `s` are the ones that are not IDA generated.

result	a1	v3	v4	v5	i	v6	v2	a2	v7	v8	v9	v1	v10	s

Table 4.2: Top 15 Variables in DIRE's Dataset

4.2 DIRE is better at recognition than prediction

Accuracy	All Variables	Without IDA-generated Variables
Overall	82%	43%
Function in training set	88%	48%
Function not in training set	41%	17%

Table 4.3: Prediction accuracy of all the variables versus IDA-generated variables when the function containing these variables were in training set versus not in training set

As seen in table 4.3, there is a clear disparity between the accuracy of DIRE when the function is in the training set and when is not in the training set.

If I exclude the variable names generated by IDA and calculate DIRE's accuracy then it comes out to be even lower. It drops from **41%** to **17%** for functions that are not in training set. This means, DIRE correctly predicted only 17% of the total variables if it has not encountered that function before.

4.3 Incorrect and Unnatural Prediction

To evaluate the prediction of DIRE, I examined a sample of 1142 functions from 100 files. There were approximately 6k variables in the same.

15% of the total variables are incorrectly predicted. **20**% of 1142 functions have at least 1 incorrectly predicted variable. The variable names in red are incorrect predictions.

Original	Predicted
oldPredCity	t
cityId	px
result	result
preCityId	e

Table 4.4: DIRE's Incorrectly Predicted Variable Names

Unnatural variable names account for **0.5**% of all predicted variable names. They are divided into three categories

- DIRE predicts invalid variable names which are or include ??, _ ??

Original	Predicted
Z	??
Z	?? 1
buff	buff2
a5	buff_len

Original	Predicted
v3	v3
frac_Q7	frac_Q ?
v4	v4
in	in

Table 4.5: DIRE's Invalid Predicted Variable Names

- Sometimes when DIRE predicts variable names, it keeps appending the same word to the same variable name.

Original	Predicted
i	i
task 2	task 2
task6	task_task_task_set
task7	task_task_task_set1
task8	task_task_set2

Table 4.6: DIRE's Unnatural Predicted Variable Names

- In many cases, the prediction results are mismatched among variables of the same function.

  For example, in table 4.7, the variable name src appears twice, and both the times it is incorrectly predicted as ina and in. Variable dest is predicted as src. The human analyst could end up making wrong decisions based solely on the predicted variable.

Original	Predicted
i	cl_0a
ctx	buffer
b	cl_0b
cl_0a	ctx

Original	Predicted
destsize	r
src	ina
src	in
destsize	r
result	dest
dest	src

Table 4.7: DIRE's Incorrectly Predicted Variable Names

```
if not hyps:
    # return identity renamings
    print(f"Failed to found a hypothesis for function {ast
       .compilation_unit}", file=sys.stderr)
    variable_rename_result = dict()
    for old_name in ast.variables:
        variable_rename_result[old_name] = {'new_name':
           old_name,'prob': 0.}

    example_rename_results = [variable_rename_result]
```

Listing 4: Code Snippet from GitHub repository Lacomis *et al.* (2019b)

4.4 Implementation Issues

It is great that DIRE is open-sourced, making it easy for me to replicate their results, verify my findings and evaluate any improvements. When DIRE fails to find a hypothesis for the function and is unable to predict variable names, DIRE's default decoders copy original variable names to predicted variable names. These are likely to be implementation mistakes. But a significant amount of original variable names are IDA-generated so when these are directly used as predicted variables, it inflates the accuracy.

Figures 3.1 and 3.2 shows the code snippets from their decoder where they directly copy original variable (old_name) to the predicted variable (new_name)

```
for var_id, old_name in enumerate(ast.variables):
    var_name_token_ids = hyp.variable_list[var_id]
    if var_name_token_ids == [same_variable_id,
       end_of_variable_id]:
        new_var_name = old_name
    else:
        new_var_name = self.vocab.target.subtoken_model.
           decode_ids(var_name_token_ids)
```

Listing 5: Code Snippet from GitHub repository Lacomis *et al.* (2019c)

Chapter 5

IMPROVING TRAINING DATASET

The goal is to improve the quality of DIRE's training dataset and then evaluate the model's performance. The improvements are based on a few observations and insights of a programmer. About half of the variables in DIRE's dataset are IDA-generated, so the first improvement is unifying IDA-generated names. For the second improvement, I will normalize the variables and convert abbreviated variable names to their full forms.

5.1 Observation

- **i**, **j**, **k** are almost always used as loop counters.
- **pos** and **position** are semantically same but they are treated as two different variables.

5.2 Insight

Different variable names may capture the same semantics. In the eyes of a progam-mer `addr` and `address` or `tot` and `total` mean the same. But `addr` and `address` are treated as two different variables.

5.3 Improvements

Based these observations and insights, I propose, a solution to improve the quality of training data.

1. Unifying IDA-generated variable names
2. Converting abbreviations to their full forms

5.3.1 Unifying IDA-generated Variable Names

DIRE's dataset has **53**% IDA-generated variable names like v1, v2, and a1. For unification, I replace all the aN with uniqueidaarg and vN with uniqueidavar. This improvement reduces the size of the data set and vocabulary. This could help the model to learn better.

5.3.2 Converting abbreviations to their full forms

The second improvement is normalizing variable names and mapping variable name abbreviations to their full forms. To the best of my knowledge, this is the first systematic approach to match variable name's abbreviations with their full forms.

Intuitively when writing the code, we use prefixes for variable names like res for result or tot for total . Alternatively, the variable name is shortened by stripping the vowels, such as rsrcs for resources. Based on these insights I created a lookup that maps abbreviations of variable names to their full forms.

Lookup Creation

1. Collected variable names from source code written by humans.
2. Created a universal lookup of abbreviations with all possible combinations of prefixes and vowels stripped variables.
3. An abbreviation is considered valid only if it appears in the human-authored source.
4. Further mapping of abbreviations and full forms is done by choosing the most likely full form based on their occurrences in human source code.

Abbreviation	Full Form
src	source
len	length
obj	object
addr	address
seq	sequence
res	result

$\xrightarrow{Transform}$

Original	Improved
res	result
src_obj	source_object
seq_len	sequence_length
addr_num	address_number

Table 5.1: Variable Transformation Based on Lookup

Normalization

Normalization of variable names involves the following steps:

1. Converting variable names in different styles to snake case
   - *Example: dstSize → dst_size*
2. Striping numbers at the end of variable names
   - *Example: tmp1 → tmp*
3. Stripping underscore from the variable names
   - *Example: __value → value*
4. Lemmanize variable names
   - *Example: bytes → byte*

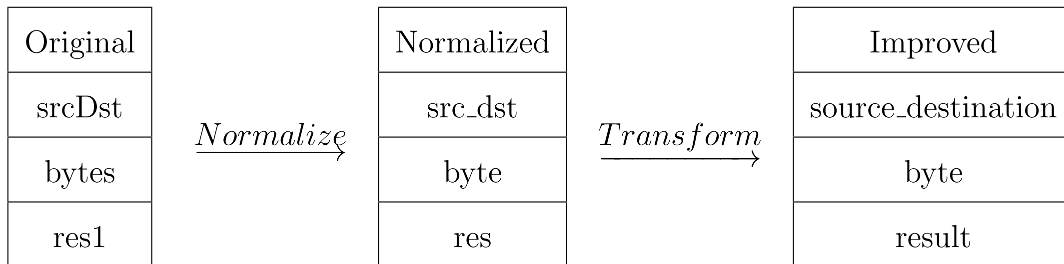

Table 5.2: Preprocessing Pipeline. First normalize variable names then convert abbreviated variable names to their full form

Chapter 6

EVALUATION

In order to evaluate the proposed solution, I preprocessed DIRE's dataset and trained it using DIRE's model.

6.1 Experiments

1. Model A: Normalize, map abbreviations and unify IDA-generated variable names
2. Model B: Normalize variable names and map abbreviations
3. Model C: Unify IDA-generated variable names
4. Reference Model: DIRE's model

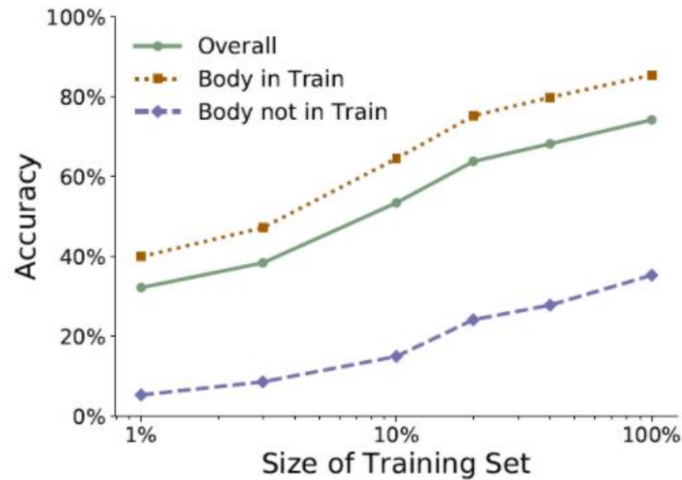

Figure 6.1: Accuracy Versus Size of Training Set. Accuracy increases with increase in the size of training set Lacomis *et al.* (2019a)

The authors of DIRE discuss the impact of the size of the training set on accuracy. As seen in the graph (Figure 6.1) Lacomis *et al.* (2019a), accuracy increases with the increase in the size of the training set. Hence, for my analysis, I am using $1/16^{\text{th}}$ data and $1/8^{\text{th}}$ data of the original dataset which would give me comparable results.

6.2 Evaluation Results

Figure 6.2, shows the accuracy of four models on two different size of dataset i.e. $1/16^{\text{th}}$ data and $1/8^{\text{th}}$ data of the original dataset. This evaluation is done on their respective test splits that were created during training of each model.

For Model A, all the aN, and vN are converted to two high-frequency variable names and the remaining variable names are normalized. Two high-frequency variable names may have introduced a bias into the model, impacting accuracy. So, even after increasing the size of the training set, there was no significant improvement in model's accuracy.

For Model B, I only normalize the variable names and IDA-generated variable names remain unchanged. As shown in the data, normalization of variable names leads to increased accuracy of the model, as it reduces the vocabulary without creating bias. Thus, the model can learn better.

For Model C, I only replace IDA-generated variables aN and vN with uniqueidaarg, and uniqueidavar respectively. This does result in a smaller vocabulary, but at the expense of bias, so this model performs worse. The accuracy for Model C is only **40**%.

The Reference Model is trained on the original dataset of DIRE to evaluate the impact of the proposed improvement.

Model	Accuracy ($1/16^{th}$)	Accuracy ($1/8^{th}$)
Model A	45%	45%
Model B	70%	71.4%
Model C	40%	43%
Reference Model	51%	68%

Table 6.1: Comparison of Model's accuracy on $1/16^{th}$ data and $1/8^{th}$ data of original dataset

6.3 Case Study

Original Variable	DIRE's Prediction	Improved Prediction
mac_callback_ptr	mac_list	mac_callback_pointer
v2	v2	uniqueidavar
signum	n	signal_time
v1	state	uniqueidavar
__pos	val	position
j	v1	j
val	flags	value
offset	mask	offset
offset	a2	offset
st_fs	st	st_fs
__initialize_p	__initialize_p	initialize_p

DIRE's Prediction: Original Dataset versus Improved Dataset

Figure 6.3 has some example predictions from DIRE's reference model and Model A. The variable names are more readable after training the model on improved dataset.

Chapter 7

LIMITATIONS

When building any solution, there are almost always limiting factors that keep us at bay from perfection. DIRE's dataset has a humongous amount of IDA-generated variables, which limits the scope of improvement. Moreover, the lookup is based on certain assumptions that do not always fit different situations. Below are a few known limitations of this work.

- The lookup is based on the insight that programmers use prefixes as variable names and vowel stripped variable names. But the lookup does not include any commonly used abbreviated variable names that do not fall under the above assumptions. For example, `packet` is written as `pkt` or `pointer` as `ptr`.

- The full form of abbreviations are selected based on probability. For example, `strm` can have two full forms `stream` or `storm`. Based on the probability of their occurrences `stream` is a better fit than `storm`. This selection makes sense from the point of view of computer science. But there are some cases that are considered invalid. Based on probability the full form of abbreviation `dst` comes out to be `dust`, but actual the full form is `destination`. Given that this mapping is not under lookup's assumption, it fails in similar situations. So the lookup needs to be manually cleaned to remove invalid or inappropriate abbreviations.

- In the dataset of DIRE, the most important variables are `v1`, `v2`, `v3`, etc. Due to their abundance and lack of meaning, the lookup procedure will be unable to map or improve their meaning to a large extent.

Chapter 8

FUTURE WORK

The proposed improvements on DIRE's dataset are mere syntax level improvements. But the source code or variable names have more semantic meaning associated with them. None of the existing approaches targets variable name recovery at its semantic level. Future directions for this research would be to explore more ways of having semantically meaningful variable names. Here are two possible directions to go in.

- Variable names like `sum` and `addition` are usually used for storing the result of addition of two numbers. But these are considered two different variables, even though their intended meaning is same. There is no existing solution that maps variable names at such semantic level. It will be interesting to study and map variable names that are synonyms.

- As mentioned in limitations, the lookup does not include the frequently used abbreviations like `ptr` and `pkt`. There is no existing approach or a lookup that can directly give you full forms of abbreviated variable names. There are tons of such variable names. Having an auto-generated lookup for similar variables will be interesting future work.

Chapter 9

CONCLUSION

After analyzing, understanding, and trying to improve variable names in the decompiled code, here are a few key takeaways.

- Model C performs the worst of the three Models, so it is not a good idea to unify the IDA-generated variable names on DIRE's original dataset. The two artificial variable names `uniqueidaarg` and `uniqueidavar` are present in high frequency in the preprocessed dataset. This can possibly be a source of bias, resulting in inaccurate and biased predictions.

- Based on Model B's accuracy, I can say that normalizing variable names is a good idea since it has the highest accuracy among all models. However, the effect of normalization may not be very apparent when unified IDA-generated names are present in large quantities as shown by Model A's accuracy.

- IDA-generated variables make up approximately **53%** of the data set, probably leading to a bias in training and making the predicted variable names IDA-like.

REFERENCES

"Gnu debugger", https://www.gnu.org/software/gdb/ (1986).

"Compilers, assemblers, linkers, loaders: A short course", https://courses.cs.washington.edu/courses/cse378/97au/help/compilation.html (1997).

"Hex-rays. the hex-rays decompiler.", https://www.hex-rays.com/products/decompiler (2019).

Brumley, D., J. Lee, E. J. Schwartz and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring", in "22nd USENIX Security Symposium (USENIX Security 13)", pp. 353–368 (USENIX Association, Washington, D.C., 2013), URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz.

Eager, M. J., "Introduction to the dwarf debugging format",

Emmerik, M. J. V., *Static Single Assignment for Decompilation*, Ph.D. thesis, The University of Queensland (2007).

He, J., P. Ivanov, P. Tsankov, V. Raychev and M. T. Vechev, "Debin: Predicting debug information in stripped binaries", in "Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018", edited by D. Lie, M. Mannan, M. Backes and X. Wang, pp. 1667–1680 (ACM, 2018), URL https://doi.org/10.1145/3243734.3243866.

Lacomis, J., P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig and B. Vasilescu, "DIRE: A neural approach to decompiled identifier naming", CoRR **abs/1909.09029**, URL http://arxiv.org/abs/1909.09029 (2019a).

Lacomis, J., P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig and B. Vasilescu, "Dire: Decompiled identifier renaming engine", https://github.com/CMUSTRUDEL/DIRE/blob/master/DIRE/neural_model/model/attentional_recurrent_subtoken_decoder.pyL254 (2019b).

Lacomis, J., P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig and B. Vasilescu, "Dire: Decompiled identifier renaming engine", https://github.com/CMUSTRUDEL/DIRE/blob/master/DIRE/neural_model/model/attentional_recurrent_subtoken_decoder.pyL284 (2019c).

NSA, "Ghidra. the ghidra decompiler.", https://ghidra-sre.org/ (2019).

Yakdan, K., 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 (NDSS), ISOC", (2015), URL http://www.internetsociety.org/sites/default/files/11$_{42}$.*pdf*.