"Len or index or count, anything but v1": Predicting Variable Names in Decompilation Output with Transfer Learning

Kuntal Kumar Pal*, Ati Priya Bajaj*, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, Ruoyu Wang
Arizona State University
{*kkpal, atipriya, pbanerj6, dutcher, mutsumi, zbasque, hgupta35, ssawan13, uananthe, yans, doupe, chitta, fishw*}*@asu.edu*

2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00152

***Abstract*—Binary reverse engineering is an arduous and tedious task performed by skilled and expensive human analysts. Information about the source code is irrevocably lost in the compilation process. While modern decompilers attempt to generate C-style source code from a binary, they cannot recover lost variable names. Prior works have explored machine learning techniques for predicting variable names in decompiled code. However, the state-of-the-art systems, DIRE and DIRTY, generalize poorly to functions in the testing set that are not included in the training set—31.8% for DIRE on DIRTY's data set and 36.9% for DIRTY on DIRTY's data set.**

In this paper, we present VARBERT, a Bidirectional Encoder Representations from Transformers (BERT) to predict meaningful variable names in decompilation output. An advantage of VARBERT is that we can pre-train on human source code and then fine-tune the model to the task of predicting variable names. We also create a new data set VarCorpus, which significantly expands the size and variety of the data set. Our evaluation of VARBERT on VarCorpus, demonstrates a significant improvement in predicting the developer's original variable names for `O2` optimized binaries achieving accuracies of 54.43% for IDA and 54.49% for Ghidra. VARBERT is strictly better than state-of-the-art techniques: On a subset of VarCorpus, VARBERT could predict the developer's original variable names 50.70% of the time, while DIRE and DIRTY predicted original variable names 35.94% and 38.00% of the time, respectively.

1. Introduction

Compilation, which transforms high-level source code into low-level machine code, is fundamentally a *lossy* procedure. Much semantic information, including control flow structures, function names, variable names, and comments, is discarded during compilation because the target machine does not need such information. For example, a CPU does not understand the notions of variables, types, or loops (relying only on registers, memory, and branch statements), so the compiled output does not need this information.

This phenomenon of compilation-induced information loss makes it more difficult for human analysts to understand binary programs ("binaries") than to understand source code [61], despite the fact that a compiler-generated binary encodes the same logic as the corresponding source code. To aid humans in such understanding, and support a number of downstream security tasks, researchers have developed a number of *de*compilation techniques, which take as input binary code, recover the lost semantic information from the binary, and derive roughly equivalent source code (or pseudocode, which generally is in an approximate version of C). State-of-the-art decompilers, e.g., IDA Pro's Hex-Rays decompiler [27], Ghidra [13], and Binary Ninja [43], are widely used in academia and industry. Security applications for decompilation include malware analysis [18, 19, 61], vulnerability discovery in binary code [40], patching software defects [50], protocol reverse engineering [32], and code reuse discovery [42].

However, decompilation is far from perfect, and significant problems continue to be addressed by researchers, including the reconstruction of code structure [62], inferencing of variable types [10, 37, 45, 66], and even the recovery of meaningful variable names [10, 36]. Variable name recovery is important because developers strive to properly name variables to *embed semantic meaning* and to improve readability (and maintainability) of source code [49]. Unfortunately, unless debug information is preserved during compilation, these carefully-chosen variable names are lost, contributing to the difficulty of binary code understanding.

An important insight from the existing variable name recovery work DIRE [36] and DIRTY [10] is that the semantics of variable names are related to their context, i.e., the surrounding code. DIRE uses information from the Abstract Syntax Tree of the decompiled code to feed into a neural network model to predict variable names. DIRECT [44] improved upon DIRE by only relying on text tokens and was evaluated on DIRE's data set. DIRTY uses a Transformer-based neural network model, trained on a large corpus of

* Equal contribution

DOI 10.1109/SP54263.2024.00152

annotated decompilation output, to infer variable types and names.

While these models can correctly predict the developer-given variable name in 57.5% (DIRE on DIRTY's data set) and 66.4% (DIRTY on DIRTY's data set) of cases in DIRTY's variable name prediction evaluation [10, Table 4], when functions that are in the training set are removed from the testing set, the accuracy drops to 31.8% (DIRE on DIRTY's data set) and 36.9% (DIRTY on DIRTY's data set). It is unnecessary for a variable name prediction system to correctly identify variables in functions that are seen during training—users can already create signatures[1] of common libraries and functions to recognize function names and variable names. In fact, the biggest promise of variable name prediction systems is to predict variable names on functions that they have *never* seen before, and to predict semantically valid variable names based on the context of a variable's use.

Therefore, we aim to further the promise of variable name prediction systems by leveraging recent advances in machine-learning (ML) language models. Both DIRE and DIRTY[2] train special-purpose neural network models for the specialized task of variable name predication. To improve prediction performance, modern ML language models that aim to understand human text are no longer specially-trained on a specific task. For instance, the Bidirectional Encoder Representations from Transformers (BERT) [15] model pioneered the concept of first *pre-training* a model on natural language text, then *fine-tuning* the model for a specific task. The pre-training phase allows the model to learn about text and the relationship between the words including the whole context, and pre-training *transfers* the knowledge learned by the model to the fine-tuning task.

As stated in DIRE and DIRTY, code (both source and decompiled) share similarities with natural languages. Therefore, we believe that the variable name prediction task can be improved by using the BERT concept of pre-training and fine-tuning. We created a novel model, called VARBERT, to validate this idea. In VARBERT, the pre-training phase is on real source code, so that the model can learn about intricacies of the source code (including the relationship between variable names and their context). Then, we fine-tune the model on decompiled code for the task of variable name prediction and variable origin (i.e., is the variable an extraneous one generated only during decompilation) prediction. In this way, we significantly augment the learning of VARBERT beyond just a data set of decompilation and ground-truth of variable names.

In the process of replicating the DIRE and DIRTY results and comparing with VARBERT, we identified several latent shortcomings with their evaluation data sets that, when corrected, impact that system's evaluation results. Both the DIRTY data set, which we will call DIRT [2], and the DIRE data set, which we will call DIRE-DataSet [1], use similar data set generation techniques and contain similar issues. Hence, we built a more extensive data set, which is expanded with an eye toward evaluating the realistic application of variable name prediction techniques, for training and testing VARBERT. For pre-training, we used source code from C packages in the Debian APT repository, totaling 5.2M functions. For fine-tuning, we collected C and C++ packages from the Gentoo package repository and compiled them targeting x86-64 with four different compiler optimizations (`O0`, `O1`, `O2`, and `O3`).

After decompiling with both the IDA Hex-Rays decompiler (referred to as IDA hereinafter) and Ghidra, we produced a deduplicated corpus called VarCorpus. VarCorpus is significantly larger than DIRE-DataSet and DIRT, including four different optimization levels and two different decompilers, whereas DIRE-DataSet and DIRT include one optimization level and one decompiler.

Our evaluation shows that after training, VARBERT could predict the same variable names that the original developer chose for `O2` optimized binaries 54.43% of the time for IDA and 54.49% for Ghidra. VARBERT strictly improves the performance over state-of-the-art techniques: On a comparable subset of VarCorpus (IDA-O0), VARBERT achieved 50.70% while DIRE and DIRTY achieved 35.94% and 38.00%, respectively. We also performed a user study on how having meaningful variable names (as prediced by VARBERT) impacts humans' understanding of code. Results from the study show that having variable names that VARBERT predicts not only makes correctly understanding decompiled code easier, but also significantly reduces the time required for humans to understand decompiled code.

Contributions. We summarize our contributions as follows:

- We built a new neural-network model, VARBERT, to predict variable names and variable origins in decompilation output using transfer learning from source code samples.
- We used novel techniques to build a new data set, including decompiled functions from both C and C++ binaries, which is significantly larger and more extensive than existing data sets.
- Using our new data set, we evaluated DIRE, DIRTY, and VARBERT. VARBERT achieved 54.43% and 54.49% prediction accuracy on our data set of `O2` binaries, for IDA and Ghidra respectively. VARBERT (50.70%) outperformed DIRE (35.94%) and DIRTY (38.00%) on `O0` binaries for IDA. Our evaluation shows that VARBERT is applicable to variable name and origin prediction on decompilation output of stripped, real-world software.
- We conducted the user study to evaluate the impact of predicted variable names in decompiled functions on code understanding. The study results provide statistically significant evidence to support the usefulness of VARBERT on the task of understanding decompiled code.

1. For instance, IDA Pro has Fast Library Identification and Recognition Technology (FLIRT) signatures for this purpose.

2. While DIRTY also predicts variable types, in this paper we focus only on predicting variable names.

In the spirit of open science, we release our research artifacts, including all data sets, the source code, and our models[3].

2. Background

Predicting variable names is built atop many layers of foundations. In this section, we provide the necessary background knowledge on these layers.

2.1. Binary Reverse Engineering

Binary reverse engineering usually refers to the process of analyzing binary programs with limited or no access to the original source code. The obscurity of binary programs makes analyzing malware [18, 19, 61], finding software vulnerabilities [40], and mitigating software defects [50] extremely difficult, and the goal of binary reverse engineering is to alleviate this problem.

Human binary reverse engineering. Ethical human analysts use binary reverse engineering techniques in malware analysis [20], deobfuscation [60], binary diffing [9, 17], inferring data structures [37, 45, 55, 66], manually finding vulnerabilities [57], assisting automated vulnerability discovery [7, 31, 52, 54], and patching vulnerabilities in binary code [58]. Reverse engineering binary programs usually requires significant expertise and the use of sophisticated, sometimes very expensive, tools. Popular binary reverse engineering frameworks include IDA Pro [27], Ghidra [13], Binary Ninja [43], Hopper [28], and angr [53].

Binary decompilation. Decompilation is an intuitive approach for making binary code less obscure by recovering high-level source code (such as C code) from binary code [50]. Due to the lossyness inherent in the compilation process, binary decompilation must attempt the undecidable task of recovering variables. Even this simple task can be complex and create decompilation artifacts: With only one variable in the original code, the decompiler may infer many variables. These could be due to operations of the binary code (e.g., storing and retrieving the same variable from the stack), to compilation operations (e.g., creating temporary variables), or to compiler optimizations (e.g., duplicating code or variables to improve performance). Therefore, the variables that exist in decompilation can be either *human-created* (i.e., in the original source code) or *extraneous* (i.e., not in the original source).

By enabling certain compilation flags (e.g., `-g` for GCC and Clang), compilers may preserve *debug information* either in the binary or as a separate file for debugging purposes.

Predicting variable names in decompiled code. The quality of decompilation output will be significantly lower than the original source code due to information discarded during compilation. A critical category of lost information is variable names. While modern decompilers attempt to infer some variable names in decompilation output in a rule-based manner (e.g., arguments passed to known library functions), they still leave a large portion of variables unnamed. Human analysts must understand the decompilation output, which is tedious, and rename unnamed variables one by one. Because variable names are critical in assisting with understanding the source code, the lack of such information in decompilation output severely hampers its readability.

2.2. NLP Fundamentals

We take inspiration for VARBERT from the concepts of *transfer learning* generally and specifically Bidirectional Encoder Representations from Transformers (BERT) [15].

Transfer learning. Training a neural model determines the optimal parameter values or weight of each node. First, initial parameter values are usually randomly selected, then they are updated based on training for a specific task (providing a known input to the model and updating the weights so that the model is more likely to predict the intended output).

Researchers in NLP and computer vision proposed (and demonstrated the success of) the concept of *transfer learning* [12, 14, 41, 65], wherein parameter values originally trained for Task A can be re-used as initial parameter values when training a neural model for Task B if A and B are similar. Creating an initial model for Task A is called *pre-training*, and the second training run on Task B is called *fine-tuning*.

Given the obvious similar nature between source code and decompilation output, transfer learning (pre-training on source and fine-tuning on decompilation output) becomes a natural choice. It also reduces the need for clean and large task-specific data sets. In VARBERT, we use transfer learning to compensate for the difficulty of creating a large corpus of clean decompilation output: We pre-train on easy-to-obtain source code and then fine-tune on the decompilation task.

BERT. Bidirectional Encoder Representations from Transformers (BERT) is a neural model for pre-training deep, bidirectional representations from unlabeled text by jointly conditioning on left and right context in all layers [15]. It learns how words in text are used while considering the entire context wherein the word appears. BERT is pre-trained in an unsupervised way on a huge collection of natural language text with two pre-training tasks: Masked Language Modeling (MLM), which helps the model learn token representations based on relationships among input tokens, and Next Sentence Prediction (NSP), which helps in understanding sentences relations in a passage.

Masked Language Modeling (MLM). BERT proposed a language modeling task, called Masked Language Modeling (MLM), to train Transformer Encoders and learn rich representations for tokens. The idea is to randomly mask (i.e., remove) tokens from a complete token sequence and ask the model to predict the masked tokens.

While there are other details in the MLM training task for BERT, we refer the interested reader to the original BERT paper [15] for those details.

3. https://github.com/sefcom/VarBERT

Vocabulary for neural models. A neural model uses a vocabulary to translate tokens to numeric encodings. A vocabulary is created using a tokenizer on its input (e.g., text). The size of the vocabulary should not be too low or too high for performance concerns. *Byte-Pair Encoding (BPE)* [51] is a hybrid method between the character and word level text representations. BPE can handle large vocabularies, where a word is split into smaller sub-unit words.

Overlap between pre-training and fine-tuning corpora. Overlap between unlabeled or auto-labeled corpora used in pre-training and labeled corpora used in finetuning in modern models is common and generally accepted: For example, BERT, which was pre-trained on books and Wikipedia, is used on tasks such as reading comprehension (SQuAD [48]), questions and answers (Natural-Question-Dataset [35]), and complex question/answer pairs (HotpotQA [64]), all of which use Wikipedia to build task datasets.

3. Overview

VARBERT comprises two main components: The first component is Corpus Generation, which collects and processes source code, builds binary executables or libraries with compiler optimizations (e.g., `O0`, `O1`, `O2`, and `O3`), generates debug symbols as ground truth for training and testing the model, and invokes external decompilers (e.g., IDA and Ghidra) to decompile binaries and create corpora. The second component (shown in Figure 1) is a BERT-based neural network model that takes as input tokenized decompiled code and predicts, for each variable, its *variable origin* and its human-created *variable name*. Variable origin refers to whether a variable is created by the software developer (human-created) or introduced by the decompilation process, and did not exist in original source code (extraneous).

3.1. The Neural Model in VARBERT

Collecting a large and diverse corpus of decompiled source code is a difficult task that prior research acknowledges [10, 25, 36]. However, acquiring a large and diverse body of source code without *requiring the code to be compilable* is much simpler. This insight led us to use *transfer learning* (described further in Section 2.2) in VARBERT for predicting variable names in decompilation output. Building this model contains two major steps:

Step 1: Pre-training. During pre-training, VARBERT takes pre-processed source code functions as input, where each function is considered as an independent instance of input. VARBERT tokenizes each function and generates a token stream. Using the token streams, we learn a vocabulary of most frequent tokens using Byte-Pair Encoding [51]. Finally, we learn the representation of code-tokens and subsequently pre-train a BERT model from scratch using *Constrained Masked Language Modeling*, a modification of Masked Language Modeling that will be described in Section 4.2.

Step 2: Fine-tuning. Next, we fine-tune the BERT model on decompilation corpora for the following tasks:

- *Predicting variable origin.* During decompilation, decompilers may introduce variables that do not exist in the original source. For each variable, VARBERT predicts whether it is created by developers and exists in the original source as human-created, or is introduced by decompilation (and does not exist in the original code) as extraneous. Collapsing extraneous variables into human-created ones improves the readability of decompilation.
- *Predicting variable name.* For each variable, VARBERT predicts its human-created variable name.

We discuss the fine-tuning process in Section 4.3.

3.2. Building A New Data Set

In the process of replicating the DIRE [36] and DIRTY [10] variable name prediction results to compare with VARBERT, we identified several shortcomings that impact both the model training and evaluation. These shortcomings necessitate a new and more extensive data set for the field of variable name prediction in decompiled code. Therefore, we build two types of corpora for VARBERT:

Pre-training corpus. The first type of corpora comprises annotated functions that human developers author and are used for pre-training. We collected C source files from the Debian APT repository, then parsed and pre-processed these files. The goal of pre-processing is to make source code resemble decompilation output, as required by transfer learning. Pre-processing includes comment removal, macro expansion, invalid identifier removal, etc. Finally, we annotated these files to indicate the variables in each function. Before creating the training and testing sets, we deduplicated the functions, resulting in a total of 5,235,792 C functions. For the rest of the paper, we refer to this corpus as the *Human-Source-Code (HSC) corpus*.

Fine-tuning corpora. The second type of corpora consists of decompilation output generated by decompilers and will be used as input for fine-tuning. We collected C and C++ packages from the Gentoo package repository, built them targeting x86-64 for four compiler optimizations: `O0`, `O1`, `O2`, and `O3` with debug symbols preserved (`-g`). We had a total number of 112,488 deduplicated x86-64 binary executables or libraries prior to decompilation. This number includes binary executables for all four compiler optimizations.

Figure 2 illustrates the process. We processed the debug symbols for each binary, stripping their type information, and leaving only human-created variable names. This is important because it eliminates the impact that debug symbols (especially type information) have on decompilation output. Then, we decompiled each binary with two decompilers, IDA and Ghidra, and collected the decompilation output per function. We also removed obviously useless functions (e.g., PLT and glibc stubs) from the collection, then annotated the remaining functions to indicate the locations of variable names in each function. We call this corpus *VarCorpus*. VarCorpus-O0 is a corpus created from binaries compiled with `-O0` compiler optimization and so on.

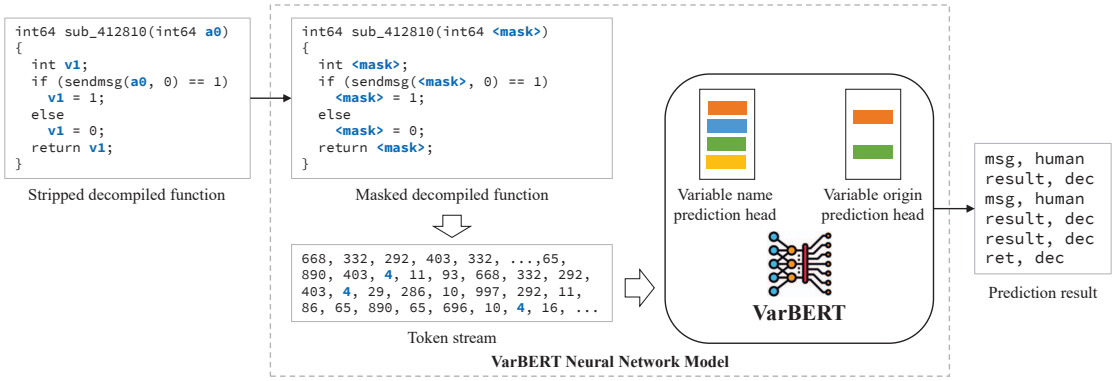

Figure 1: The prediction pipeline of VARBERT. The decompiled code is variable-masked and tokenized into a token stream, which is used as input to the model for prediction. VARBERT predicts both variable name and origin for each masked location.

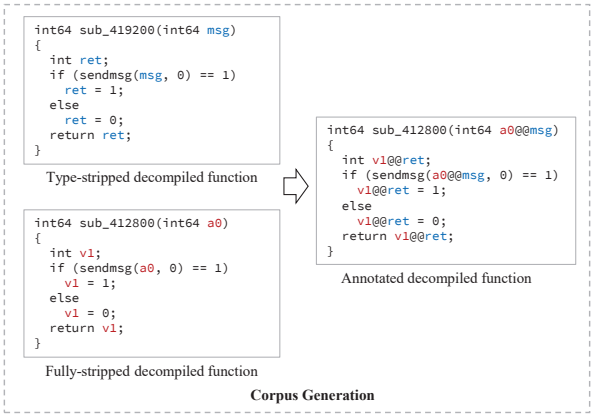

Figure 2: Matching a fully-stripped decompiled function with its corresponding type-stripped version to create an annotation function in a fine-tuning corpus.

4. The VARBERT Model

While the VARBERT model is based on BERT, we customized many aspects of the model design based on our insights on the task of variable name prediction in decompiled code. In the remainder of this section, we first present basic parameters of the VARBERT model (Section 4.1). Then we present the pre-training phase (Section 4.2) and the fine-tuning phase (Section 4.3) of VARBERT, where we discuss critical design choices that we made for our model.

4.1. Basic Parameters

As a Transformer-based model, basic parameters of VARBERT are the number of neural layers L, the number of self-attention heads A, and the hidden dimension H. Training a huge transformer-based model is computationally heavy and time-consuming. Limited by accessible computing resources, VARBERT has fewer layers than the original BERT. We take initial hyper-parameters from RoBERTa [39], which demonstrated an empirically optimal hyper-parameter configuration for BERT on NLP tasks.

We hypothesized that for the task of variable name prediction a model might perform well, and created a neural network with $L = 6$, $A = 8$, and $H = 512$. We chose the maximum number of tokens as 800. Our model has 45 million trainable parameters, which is about 40% of the number of trainable parameters of BERT-Base [39].

4.2. Pre-Training

As discussed in Section 2.2, prior NLP and computer vision research demonstrated the need and impact of pre-training on auxiliary tasks before the final intended task [15, 39, 56, 63].

Tokenization. The vanilla BERT model is familiar with English word vocabulary and uses a word-piece tokenizer. However, English is quite different from source code and decompiled code in terms of structure, syntax, and keywords. Specifically, numbers, parentheses, and square brackets in code all capture meanings based on the context, and a word-piece tokenizer will ignore these important types of tokens because they are not critical in plain English. For the model to better understand code, we learn a new source vocabulary using a Byte-Pair Encoding (BPE) tokenizer instead of a word-piece tokenizer. Following the RoBERTa (an optimized version of the original BERT model) [39] tokenizer, we consider learning a source vocabulary of 50,265 most frequently occurring tokens from the training data set of our human source code (HSC) corpus.

Task formulation. Both DIRE and DIRTY formulate variable name prediction as a *generation* task: They iteratively generate tokens and combine them to form new predicted variable names. However, we observed in both solutions that they frequently predict sub-optimal variable names, such as `fire_fire_fire_fire_fire_fire`. This observation leads us to consider the variable name prediction task as a language model slot-filling task instead. VARBERT

model will predict a variable name based on its left and right contexts.

Tokenization scheme. Vanilla BERT tokenizes all input sentences, which will split most variable names into different tokens (e.g., `word_count` may be split into `word_` and `count`) and does not fit the variable name prediction task. Inspired by the whole-word-masking strategy, which keeps essential English words intact during tokenization and improves performance on many NLP tasks [5], we introduce a new tokenization scheme into the VARBERT model that inserts all frequent variable names into the vocabulary and keeps all variable names intact during tokenization.

Masked Language Modeling. Our motivation is to make VARBERT familiar with the nature of input data. We first tokenize each function of the HSC corpus and generate a token stream using the learned vocabulary. Finally, we learn the representation of the code-tokens using BERT from scratch by Masked Language Modeling approach similar to the approach given in RoBERTa [39]. We randomly mask tokens and ask the model to recover the masked token, and in the process, learn rich representations for the code-tokens.

Constrained Masked Language Modeling. Next, we formulate another pre-training task, Constrained MLM (CMLM, a variation of MLM), which was proposed in prior work [16, 23], where tokens are not randomly masked. The motivation is to teach the model the task of selectively recovering specific code tokens. We define CMLM as follows: Let $W_0, ...W_N$ be a sequence of tokens. Let $C = \{A_0, ..A_c\}$ be a set of tokens which we define as the constrained set of tokens. Then, in CMLM, all tokens in $W_0, ...W_N$ which belong to C are masked, and C is a subset of V (all variable names). We then train the model to predict the masked token with a cross-entropy loss:

$$-\sum_{i=0}^{N}\sum_{c=1}^{M} y_{w_i,v_c}\log(p_{w_i,v_c}) \tag{1}$$

where M is the size of the vocabulary, w_i is the current token, v_c is the target token, y is an indicator variable which is 1 if the target is v_c and 0 otherwise, and p is the probability that w_i is the same as v_c. Here, we select a vocabulary of 50K most frequently occurring human-authored variable names from the HSC corpus. Then, we selectively mask the variable names and train the model to predict them based on the vocabulary.

4.3. Fine-Tuning

After pre-training on general source code, we fine-tuned our pre-trained models for the variable name prediction task and the variable origin prediction task. We again use the CMLM task by masking variables in the training corpus for each decompiler. We first develop a target vocabulary of frequently occurring variable names from the training data along with earlier chosen 50K human-authored variable names. We optimized the model parameters by minimizing the overall joint cross-entropy loss $\mathcal{L}(\mathbf{X}, \mathbf{Y})$ by taking the mean loss of N mini-batches (2), where $\mathbf{X}$ and $\mathbf{Y}$ are the input and labels respectively.

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N}\sum_{n=1}^{N} L_n \tag{2}$$

For the variable name prediction task, each mini-batch loss (L_n) is the sum of all variable-name prediction loss (l_v^t) in that mini-batch. For the joint prediction task, each mini-batch loss is the sum of joint loss of the predicted name (l_v^t) and the predicted origin (l_o^t) for each variable (t) in that mini-batch (n). Equation (3) formalizes this, where T_n is the total number of variables to predict in a particular mini-batch (n).

$$L_n = \sum_{t=1}^{T_n}(l_v^t + l_o^t) \tag{3}$$

We further formalize both cross-entropy loss functions in Equations (4) and (5) where C_1 and C_2 are vocabulary lengths of the variable name prediction task and the variable origin prediction task, respectively.

$$l_v^t(\mathbf{X}, \mathbf{Y}) = \log\frac{exp(x_{ty_t})}{\sum_{c=0}^{C_1-1} exp(x_{tc})} \tag{4}$$

$$l_o^t(\mathbf{X}, \mathbf{Y}) = \log\frac{exp(x_{ty_t})}{\sum_{c=0}^{C_2-1} exp(x_{tc})} \tag{5}$$

Handling long functions. Although we increased the maximum input size of our VARBERT to 800 tokens (from BERT's 512), we still encounter many functions that have more tokens. To predict variable names for functions with more than 800 tokens, VARBERT splits these function bodies into 800-token chunks and considers them as separate samples during prediction.

5. Corpora Generation

Building a good corpus that is diverse and also representative of the target task is critical for training and evaluating any ML model. However, while using the data sets from DIRE [36] and DIRTY [10], we identified several shortcomings, which we briefly describe in Section 5.1. While these shortcomings do not impact the novelty of DIRE or DIRTY, we believe they necessitate the construction of a more extensive, thorough data set for evaluating future research. We describe the construction of a new data set in Section 5.2.

5.1. Shortcomings with Prior Data Sets

We truly appreciate the authors of the pioneer variable name prediction works DIRE [36] and DIRTY [10] for their contributions to open and reproducible science by publicly releasing their data sets (we refer to DIRE's data set as DIRE-DataSet and DIRTY's data set as DIRT) and research artifacts [1, 2]. However, there are several shortcomings in

their data sets[4]. We briefly present these shortcomings next, and provide an in-depth analysis of these shortcomings in Appendix A.1 for interested readers.

Overlap between test and training sets. There is a large overlap in the training and testing set of both DIRE-DataSet (79.9%) and DIRT (65.5%). This may lead to inflated prediction accuracy: It may appear that the model is generalizing when it is memorizing the duplicates [6, 38].

Duplicated functions. Both DIRE-DataSet and DIRT contain duplicated functions in their training sets (683K/1M, 67.6% for DIRE-DataSet and 1M/1.8M, 56.9% for DIRT), which may skew a model's training to over-represent these duplicated functions.

Unmatched variables. In both data sets, many variables in the decompilation output are not correctly matched to their original names *in the ground-truth data sets*, leaving any models trained on this corpus to predict fewer variables than they should.

For completeness, we addressed some of these shortcomings in a best-effort manner and re-evaluated DIRE, DIRTY, and VARBERT on an improved version of existing data sets. Detailed results are in Appendix A.4.

5.2. Building VarCorpus

VARBERT uses two types of corpora: The HSC corpus for pre-training and VarCorpus for fine-tuning. Building the HSC corpus was straightforward, and here we focus on the generation of the fine-tuning corpora.

Compiling packages. We collected C and C++ packages from the Gentoo repository, and compiled them using optimization level `O0` (no optimization), `O1`, `O2`, and `O3` with debug symbols kept (`-g`). Because debug symbols contain all human-created variables and their names, we have a 100% accurate mapping between each variable and its original human-created name. In total, we have 112,488 binary executables or libraries across all optimizations.

Stripping types from debug symbols. Types in debug symbols will influence decompilation output [36]. Because the major use case for VARBERT is predicting variable names on fully stripped binaries (i.e., binaries without any debug symbols, function names, etc.), we must ensure that our debug symbols only contain variable names and do not contain any type information. We developed a novel technique that strips type information from debug symbols, called *type-stripping*, which we discuss in detail in Section 5.3. After type-stripping, we produce binaries with debug symbols that only have variable names preserved.

Decompiling executables and annotating output. We batch decompiled all executables with two decompilers IDA and Ghidra (to ensure diversity in VarCorpus among decompilers), and generated decompiled code with human-created variable names. We chose IDA and Ghidra because, out of all binary decompilers that we tested, only IDA and Ghidra could generate C-style pseudocode with acceptable quality. Other decompilers that we considered either failed to decompile many functions, do not support debug information, or could not generate C-style pseudocode. Then, we batch decompiled the fully stripped version of each executable and generated decompiled code with only decompiler-assigned variable names. By matching each function with only decompiler-assigned variable names against its counterpart with human-created variable names, we annotated each variable with its unique identifier, its original variable name, and whether it was extraneous or human-created. Relying on decompilers' debug symbol loading support enables a more accurate mapping of human-created variables to their names. This way, we eliminate the need for variable matching heuristics used by prior work (which was only about 62% accurate) [30, 36]. Because we fully strip the binary, the functions do not have meaningful function names. While a compiler optimization level of `O2` (increased optimizations) is more commonly used in real-world binaries, both IDA and Ghidra fail in many cases to keep variable names from debug symbols during decompilation[5], leading to lost variable names in the decompilation output when debug symbols are available. We address this issue for IDA while building VarCorpus-O1, VarCorpus-O2, and VarCorpus-O3. However, Ghidra would frequently merge variables in decompilation that correspond to different human-created variables, making these variables extraneous.

Function deduplication. We deduplicated all the functions in VarCorpus before creating training and testing sets. For deduplication we use a hash-based strategy: We first normalize all functions by removing newlines, whitespaces, and decompiler-generated comments, then hash function bodies and only leave one function for each unique hash. This way we remove any duplicate functions (including third-party library functions) that are shared among projects. We found that similarity-based deduplication fails to capture the intricacies of code and ignores minor textual differences that can be critical for differentiating between two functions. We tokenized the code and used near-duplicate-code-detector [6, 29]. This approach removed functions where there were minimal textual differences, but in C/C++ the presence and absence of * and & convey distinct semantics. These nuances can alter the logic and behavior of code. In many cases, near-duplicate-code-detector incorrectly reports function duplicates and removes functions that are otherwise good training candidates. A carefully tuned similarity-based duplication detector may work, which we regard as a research problem that future work may address.

4. We stress that we do not blame the DIRE or DIRTY authors. In fact, we applaud them for releasing data sets and being responsive when we approached them for questions. DIRE and DIRTY pioneered this line of research, and we believe finding and addressing issues in prior research work is how science advances. This work would not be possible without the DIRE and DIRTY authors' dedication to open science, and for that we are very grateful.

5. Specifically, some human-created variable names for register variables are lost during decompilation. This is an implementation issue in both IDA and Ghidra.

Cleaning up. Finally, we cleaned up our corpora by removing irrelevant functions that only exist due to compilation or decompilation. For example, we removed all glibc-initialization functions added by GCC and Clang. We also removed function stubs in Procedure Linkage Tables (PLTs).

5.3. Reliably Matching Variable Names

We rely on a decompiler's output on a binary with debugging information as a proxy for what an "ideal" decompilation on a fully stripped binary should be. The decompiler reads debug symbols associated with the binary and consumes variable names and data types. However, it is important to note that *data types impact the decompilation output*. Therefore, we propose a novel technique, called *type-stripping*, for rewriting the DWARF debugging information of a binary to include *only* name information and no type information. Type-stripping parses DWARF from an ELF binary, manipulates it, and then fully serializes it back to the binary.

While type-stripping is more engineering-intensive than editing the DWARF data stream to remove members we found undesirable, it proved necessary due to quirks in some decompilers' parsing of DWARF data. Both IDA and Ghidra have *brittle* DWARF parsing: They refuse to integrate DWARF data into their analysis unless variables have well-formed names and type information associated. As such, it is necessary to replace all references to a given type with references to a generic scalar type of the same size instead of discarding them. Pointers become pointer-sized words, and aggregate types (i.e., structs, arrays, and unions) become words with the size of the aggregate's first member.

The benefits of type-stripping can be observed in Listing 1. (a) is the decompiled code in the presence of DWARF information, (b) is the decompiled code from the stripped binary, and (c) is the decompiled code from the binary rewritten with type-stripping. The edit distance from (a) to (b) is visibly higher than the edit distance from (c) to (b). This indicates that type-stripping the corpus before training will yield a variable name prediction model which produces more reliable predictions about the missing name information in stripped code.

5.4. Evaluating VarCorpus Quality

Now that we have collected VarCorpus, we must show that this corpus is reasonable and improves over the state-of-the-art corpora.

Binaries. An essential difference between VarCorpus and DIRT (also DIRE-DataSet) is the definition of "binaries." DIRT and DIRE-DataSet use compiled object files whereas VarCorpus only includes binary executables or libraries. Since each executable may be linked from tens, hundreds, or thousands of object files, the numbers of binaries in VarCorpus are not comparable to the ones in DIRE-DataSet and DIRT.

Corpora sizes and duplicated functions. Table 1 shows the summary of all data sets. DIRE-DataSet and DIRT both contain a high number of duplicated functions in their training sets (and test sets). VarCorpus has 0 duplicate functions. Additionally, VarCorpus is more diverse; For `O0` data sets, VarCorpus contains over 1.6x more unique functions than DIRT-Dedup and 4.6x more unique functions than DIRE-DataSet-Dedup. VarCorpus contains functions from both C and C++ binaries, and the percentage of C++ functions

```
void grecs_print_value(grecs_value *val, int flags, FILE
↪  *fp)
{
    grecs_format_closure clos;
    clos.fmtfun = file_fmt;
    clos.data = fp;
    grecs_format_value(val, flags, &clos);
}
```

(a) Decompiled code with debug symbols present.

```
void sub_D5F2(int *a1, unsigned int a2, __int64 a3)
{
    __int64 v3[2];
    v3[0] = sub_D551;
    v3[1] = a3;
    sub_D057(a1, a2, v3);
}
```

(b) Decompiled code of a stripped binary (without debug symbols).

```
void sub_D5F2(__int64 val, unsigned int flags, __int64
↪  fp)
{
    __int64 clos[2];
    clos[0] = file_fmt;
    clos[1] = fp;
    grecs_format_value(val, flags, clos);
}
```

(c) Decompiled code of a type-stripped binary.

Listing 1: Comparison among decompiled code with DWARF information, decompiled code of a stripped binary, and decompiled code of a type-stripped binary. Note that type-stripped decompilation output and stripped decompilation output almost perfectly match.

TABLE 1: Summary of all data sets, including numbers of functions, unique variable names, and numbers of binaries. "C.O." means Compiler Optimization.

Data Set	C.O.	Unique Variables	Functions	Binaries
HSC	N/A	3,561,537	5,235,792	N/A
VarCorpus (IDA)	O0	682,461	2,657,046	26,280
	O1	234,417	722,942	16,815
	O2	201,525	579,606	15,893
	O3	198,297	578,156	15,427
VarCorpus (Ghidra)	O0	521,668	2,066,871	20,433
	O1	179,016	856,608	16,647
	O2	218,633	763,053	17,939
	O3	193,712	628,384	13,770
DIRE-DataSet [36, RQ4]*	O0	92,082	1,259,935	164,632
DIRE-DataSet-Dedup	O0	92,082	463,238	N/A
DIRT [10, Table 11]*	O0	237,928	2,075,762	75,656
DIRT-Dedup	O0	237,928	995,418	N/A

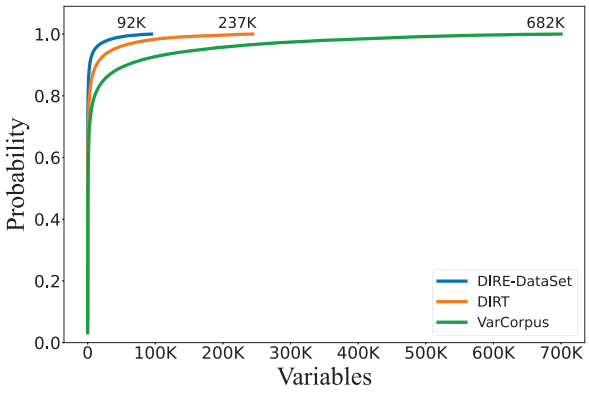

Figure 3: The cumulative distribution of unique variables in each of the three data sets (DIRE-DataSet, DIRT, and VarCorpus).

ranges from 9% to 31% across all data sets. Finally, the number of unique variables in VarCorpus-O0 is 3.57x of the number of unique variables in DIRT.

Overlap between test and training sets. Because there is no function duplication in VarCorpus, our test and training sets also do not overlap.

Overlap between HSC and VarCorpus. The structure of an HSC function is usually very different from its decompiled counterpart. To calculate the overlap between HSC (training set) and VarCorpus (IDA-O0 function test set), we consider an *upper-bound* of potential duplicates by counting duplicate tuples of package-name + function-name. There are 2.54% such duplicate pairs with an average edit distance (AED) of 339.25 characters per function pair. Therefore, we conclude that there is no overlap in HSC and VarCorpus.

Distribution of variable names. We also take a closer look at how many times each variable appears in these data sets. Figure 3 shows the cumulative distribution of each unique variable in DIRE-DataSet, DIRT, and VarCorpus. While all three data sets have similar distributions for top variables, VarCorpus contains a lot more unique variables than the other two data sets.

Variable name matching. Finally, to understand the performance of variable matching in DIRE-DataSet, DIRT, and VarCorpus-O0 (IDA), we randomly sampled 5,000 C functions in each corpus, and measured the ratio between developer-assigned variable names and the total number of variable names. Assuming IDA injects on average similar amount of extraneous variables during decompilation for binaries built with the same optimization level, the ratios on both corpora should be similar. However, through random sampling, we measured a ratio of 56% on DIRE-DataSet, 62% on DIRT, and 75% on VarCorpus (IDA). This shows that we correctly mapped more developer-assigned variable names to variables in decompilation output than DIRE-DataSet and DIRT.

Based on this analysis data, we believe that VarCorpus represents a significant improvement over the prior variable name prediction data sets, and should serve as a benchmark for future research in this area.

6. Evaluation

Throughout this evaluation, we seek to measure the performance of variable name prediction systems on predicting variable names in previously unforeseen functions. Therefore, unlike prior research (DIRE, DIRECT, and DIRTY), we only present the results from data sets where there is no duplication of functions between the training and testing sets.

We attempt to answer the following research questions in our evaluation:

RQ1. How does VARBERT perform on VarCorpus?
RQ2. How do prior work and VARBERT compare on VarCorpus?
RQ3. How do different aspects of VARBERT impact its effectiveness?
RQ4. How does VARBERT help reverse engineers in real-world binary analysis tasks?

6.1. Implementation

We implement VARBERT using Python. For decompilation when building the fine-tuning corpora, we used the Hex-Rays decompiler (in IDA Pro 7.6) and Ghidra 10.4. For building and training the neural model, we used PyTorch [47] 1.9.0, HuggingFace Transformers [59] 4.10.0, and Facebook FairSeq [46] 0.10.0.

6.2. RQ1: How does VARBERT perform on VarCorpus?

Given the high quality of VarCorpus that we evaluated in Section 5.4, we first evaluate the performance of VARBERT on VarCorpus.

We took eight corpora from VarCorpus, four for IDA (the IDA corpus), and another four for Ghidra (the Ghidra corpus) across four compiler optimizations: `O0`, `O1`, `O2`, and `O3`. In a ratio of 80:20, we split each corpus into training and test sets.

We also experimented with two data splitting approaches: (a) randomly splitting data by function (per-function) and (b) randomly splitting data by binary (per-binary). Please note that, as discussed in Section 5.4, VarCorpus per-binary split (where "binaries" are executables) is different from DIRT and DIRE-DataSet (where "binaries" are object files). This split is important because the per-function split might have functions that are from the same source project in both testing and training (although not the same functions in testing and training, as there is no duplication), while with the per-binary split this is less likely (although, it is possible for two binaries to be from different projects and include similar library functions, even if not identical, such as different versions of a library). In this sense, we expect it to be more difficult to predict when using the per-binary split than the per-function split.

TABLE 2: Evaluation of VARBERT's variable-name-prediction and variable-origin-prediction tasks fine-tuned on VarCorpus for different corpus optimization levels (C.O.) and either a function-level split or a binary-level split. Values in Top-N columns are accuracy rates of human-created variable names in percentage: Top-N means the correctly predicted variable is present among the first N predicted variable names.

VarCorpus Variant	C.O.	Split	Variable Name					Variable Origin	
			Top-1	Top-3	Top-5	AED	CER	Accuracy	F1
IDA	O0	Function	54.01	63.25	66.47	2.06	44	88.35	87.51
		Binary	44.80	53.11	56.23	2.33	52	87.69	86.94
	O1	Function	53.51	62.76	65.96	2.24	47	82.86	82.82
		Binary	42.55	49.54	52.25	2.80	62	81.59	81.47
	O2	Function	54.43	63.62	66.78	2.21	46	80.40	80.05
		Binary	42.40	49.67	52.45	2.64	59	79.48	79.04
	O3	Function	56.00	64.81	67.83	2.12	44	80.74	80.49
		Binary	40.70	49.26	52.35	2.71	61	78.28	77.89
Ghidra	O0	Function	60.13	68.60	71.44	1.67	37	88.70	88.69
		Binary	46.1	53.34	56.03	2.15	50	86.90	86.69
	O1	Function	58.47	66.16	68.76	2.30	46	88.63	87.20
		Binary	42.87	49.32	51.82	2.85	63	87.25	85.67
	O2	Function	54.49	61.52	63.93	2.34	46	87.48	85.11
		Binary	40.49	46.64	48.98	2.85	61	86.13	83.63
	O3	Function	54.68	61.71	64.08	2.43	49	88.22	85.13
		Binary	40.09	46.35	48.74	2.89	64	85.89	81.84

As previously discussed in Section 3, we pre-trained VARBERT on human source code and then fine-tuned it separately on each corpus: the IDA corpus and the Ghidra corpus.

Finally, we evaluated VARBERT on two testing sets for two tasks: Predicting the origin of each variable (extraneous versus human-created) and predicting the name for each human-created variable.

Variable origins. The right two columns of Table 2 show the results of the variable-origin-prediction task. VARBERT can predict whether a variable is human-created or extraneous roughly 80% to 90% of the time. We believe that these results can help decompilers to produce *improved* decompilation results that are closer to the source code (an orthogonal benefit to variable name prediction).

Variable names. Table 2 shows the results of the variable name prediction task on the IDA and Ghidra corpora. VARBERT achieved Top-1 accuracy of 54.01% on the IDA-O0 corpus and 60.13% on the Ghidra-O0 corpus, when split on a per-function basis. For reference, these numbers are higher DIRTY's accuracy on DIRT's not-in-train test set for IDA-O0, which was 36.9% [10]. This result shows that VARBERT learns variables' semantics better from their contexts and, because there is no overlap between training and test sets, generalizes better to functions on which the model was not trained.

Variable names on optimized binaries. VarCorpus allows the evaluation of VARBERT's performance on binaries compiled with optimizations enabled. Much to our surprise, we do not observe any significant drop in accuracy when predicting variable names for `O1`, `O2`, or `O3` binaries using per-function splits, and this finding is consistent across both decompilers. This result clearly shows the applicability of VARBERT for predicting variable names in real-world binaries, which are often built with optimizations enabled.

TABLE 3: Comparison of VARBERT, DIRTY, and DIRE on VarCorpus.

Model	Split	Top-1 Accuracy	AED	CER
VARBERT	Function	50.70	2.25	47
	Binary	37.17	2.74	62
DIRTY	Function	38.00[6]	3.13	124
	Binary	32.65[6]	3.11	123
DIRE	Function	35.94	3.34	75

Variable names on per-binary splits. As anticipated, the accuracy of VARBERT when working on per-binary splits is decreased compared to its accuracy on per-function splits. The difference with the same decompiler-optimization pair is usually between 10% and 14%. In per-binary splits, due to the Out-Of-Distribution issue, any variable names that only appear in test set cannot be predicted. We believe the performance of per-binary splits can be improved by increasing the size of VarCorpus, which we leave as future work.

Non-exact matching predictions. While this analysis is on exact-matching variable names, we notice that different variable names may have the same or similar meanings. For example, `ctr`, `count`, and `counter` are sometimes used interchangeably. To measure the performance of VARBERT in non-exact matching cases, we report Average Edit Distance (AED) and Character Error Rate (CER) in Table 2. AED and CER quantify the similarity in predicted variable and ground truth and credits partial mispredictions. For reference, the CER for DIRE on DIRE-DataSet's not-in-train test set was 67.2% [36, Table I], and VARBERT achieves a lower CER of 44%. Later (in Section 7.2) we present cases of mispredicted yet semantically correct variable names.

6.3. RQ2: How do prior work and VARBERT compare on VarCorpus?

In this experiment, we compare VARBERT, DIRE, and DIRTY on VarCorpus. We only evaluate these systems on the IDA-O0 data set because DIRE and DIRTY only support output from IDA with compiler optimization level `O0`. Because DIRE and DIRTY require different input formats, we rebuilt training and test sets for DIRE and DIRTY using binaries in VarCorpus. The training was performed on 1,050 binaries with 400K functions (∼15% of VarCorpus). This is due to the prohibitively long training time that DIRE requires for large datasets. We also attempted to include DIRECT, however we could not train any usable models (using the code released by the authors after fixing several issues) because the training for DIRECT does not scale well.

TABLE 4: The impact of pre-training tasks on variable name and origin prediction accuracy of VARBERT (on VarCorpus-IDA-O2 with function split).

Model Variants	Variable Name			Variable Origin	
	Top-1	Top-3	Top-5	Accuracy	F1
Fine-tuning only	39.83	51.28	55.62	78.68	77.53
MLM + Fine-tuning	50.04	59.16	62.31	79.83	79.40
MLM + CMLM + Fine-tuning	54.43	63.62	66.78	80.40	80.05

Table 3 shows that VARBERT outperforms DIRTY and DIRE by 12.70% and 14.76% respectively on the function split. On the binary split, VARBERT's accuracy is 4.52% higher than DIRTY's. This result shows that VARBERT is better at variable prediction when evaluated on the same data set.

Variances in DIRTY's vocabulary sizes. DIRTY originally used a vocabulary size of 10K to achieve a character coverage of 99.9% on DIRT (2 million functions on 75K binaries), i.e., covering all source code tokens. We used a higher vocabulary size of 16K (for function split) and 14K (for binary split) to achieve the same character coverage on a much smaller data set (400K functions on 10K binaries). This is because binaries in VarCorpus are more diverse and have more unique source code tokens. Appendix A.2 offers insights on this aspect.

6.4. RQ3: How do different aspects of VARBERT impact its effectiveness?

To understand the impact of multiple design choices in VARBERT, we performed an ablation study.

The impact of pre-training. We studied the impact of pre-training tasks by training new models on VarCorpus-IDA-O2. As shown by the "Fine-tuning only" and "MLM + CMLM + Fine-tuning" rows in Table 4, transfer learning, i.e., pre-training on HSC has significant impact (over 14% improvement) on the accuracy of variable name prediction task. This proves the necessity of performing pre-training on HSC.

The impact of CMLM. We further studied the impact of CMLM by pre-training new models with and without CMLM;' the results are in the "MLM + Fine-tuning" and "MLM + CMLM + Fine-tuning" rows in Table 4. When using CMLM, we observe improvements of around 4% in the Top-N accuracy in the variable name prediction task, which shows the usefulness of CMLM. CMLM has minimal impact on the variable origin prediction task.

The impact of corpus sizes. To study the impact of corpus sizes, we trained VARBERT on 20%, 40%, 60%, and 80% of the training set of VarCorpus-IDA-O2. Figure 4 shows that with 20% training data, VARBERT achieved good performance on the variable name prediction task (41.11%) and the variable origin prediction task (77.34%). The accuracy increases with more training data, which shows the importance of building even larger data sets.

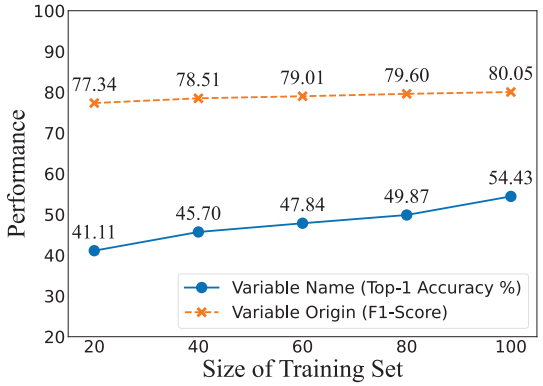

Figure 4: The impact of corpora sizes on VARBERT's performance.

TABLE 5: Performance of VARBERT on about 150K functions collected from only AArch64 binaries.

	Split	Variable Name			Variable Origin	
		Top-1	Top-3	Top-5	Accuracy	F1
IDA	Function	46.89	57.16	60.81	77.72	77.01
	Binary	27.60	36.19	39.86	76.00	75.26
Ghidra	Function	50.40	59.38	62.51	86.32	82.69
	Binary	31.92	38.74	41.65	84.87	81.01

Generalizability to non-x86-64 binaries. The final experiment in the ablation study is understanding the impact of binary architecture choices. We built a new data set using only AArch64 (ARM64) binaries (with optimization level `O2`) from the Gentoo package repository and trained a VARBERT model for it to study the generalizability of VARBERT on non-x86-64 binaries. This new data set (with 150K functions for IDA and 140K functions for Ghidra) is much smaller than VarCorpus because many Gentoo packages failed to build for AArch64. As such, we built another data set by injecting another 150K unique decompiled functions from x86-64 binaries, making a total of around 300K functions for both IDA and Ghidra. The results, in Tables 5 and 6, show comparable performance to VARBERT models that are trained on similar numbers of x86-64 decompiled functions. This shows that the approach that VARBERT represents generalizes to other architectures, and the performance strongly correlates to the number of functions in the training set (regardless of architecture).

6. In DIRTY's evaluations, decompiler-generated variables, such as `v2` are counted as true positives when predicted as `v2`. However, these variables are not developer-assigned; therefore, after postprocessing and eliminating these predictions, the revised accuracies for the Function and Binary splits are 18.55% and 13.15%, respectively. Similarly, the updated accuracy for DIRTY in Table 10 is 37.15%.

TABLE 6: Performance of VARBERT on about 300K functions collected from both AArch64 and x86-64 binaries.

	Split	Variable Name			Variable Origin	
		Top-1	Top-3	Top-5	Accuracy	F1
IDA	Function	52.20	62.03	65.34	79.47	78.83
	Binary	34.89	43.76	47.08	77.73	76.66
Ghidra	Function	56.36	63.60	66.04	87.08	84.25
	Binary	39.60	46.01	48.40	86.81	82.62

6.5. RQ4: How does VARBERT help reverse engineers in real-world binary analysis tasks?

We performed a user study on the task of understanding decompiled functions. Our study was exempted by our Institutional Review Board (IRB) at Arizona State University (ASU). However, we followed the same ethics and privacy requirements that an IRB would normally enforce.

Participants. We invited 34 students who have gone through reverse engineering trainings offered in undergraduate- or graduate-level security courses at ASU to participate in our user study. Out of 34 students, four (4) are expert reverse engineers (who either are frequent Capture-The-Flag players or have over three years of binary reverse engineering experience). 7 students (none of which are experts) registered for the study but did not finish all questions, leaving 27 students who completed the entire study. Appendix A.5.1 provides the invitation email.

Function Understanding Task. We randomly picked 13 decompiled function pairs (with and without predicted variable names) with at least four local variables or function parameters, at least 10 lines of code, at most 100 lines of code, and at least 75% of correctly predicted variable names from the test set of VarCorpus. For each function pair, we provide four choices that describe the intended functionality of each function in English. Each choice contains one or two sentences. During the study, each participant is randomly given one function out of each pair (so no one gets both named and nameless variants of the same function). Appendix A.5.2 provides an example function pair as well as the choices.

During the study, we recorded the score of each participant (one point for every correctly answered question, zero points for incorrect answers) and how long each participant spent to answer each question. At the end of the study, we asked six more questions to collect meta information about the study (e.g., whether the participant used search engines during the study or not) as well as the user perception.

Results. **Correctness.** Table 7 shows average scores for all participants and participants within each skill-class of our user study. On average, participants answered questions more correctly when they faced decompiled functions with predicted variable names (66.67% versus 55.49%, an improvement of 11.18%). In terms of correctness, predicted variable names helped expert, intermediate, and novice reverse engineers at roughly similar levels: The correctness rate improvement was between 10% and 13%. We conclude that having mostly correctly predicted variable names helps users of all skill levels on the function understanding task.

TABLE 7: The average score and time spent on understanding decompiled functions with predicted variable names (Predicted) and without predicted variable names (Raw).

Category	Users	Avg. Score (%)			Avg. Time (sec.)		
		Overall	Raw	Predicted	Overall	Raw	Predicted
All	27	61.34	55.49	66.67	116.87	123.84	110.19
Expert	4	90.70	84.00	100.00	95.11	94.18	96.47
Intermediate	4	71.15	65.22	75.86	118.28	141.35	98.22
Novice	19	54.22	47.41	60.15	120.49	126.92	114.65

Analysis speed. Table 7 also shows the average time that each skill-class participants spent before answering questions (regardless of the correctness of their answers). Overall, participants understand decompiled functions with predicted variable names faster than functions without predicted variable names (110.19 versus 123.84). Interestingly, while predicted variable names showed little help to experts, they helped intermediate-level participants most, reducing the average time by almost 43 seconds (98.22 versus 141.35).

User perception. All but one participant agreed that having predicted and meaningful variable names made understanding the functionality of a decompiled function easier. Additionally, only nine (9) participants copied the code into a local text editor and attempted to rename variables to assist with their understanding (our online interface for the user study did not allow code editing). This shows the usefulness of VARBERT as well as the necessity of predicting variable names for function understanding tasks in general.

7. Case Studies

7.1. DIRTY and VARBERT Comparison on DIRT

We select one example of variable name prediction results to examine here as a case study. We chose the function `qemu_clock_enable` from DIRT and compared the variable name prediction results of DIRTY and VARBERT.

Listing 2a shows the original source code of the `qemu_clock_enable` function (which we were able to find using the decompilation output). Next to Listing 2a shows the decompilation output with the variable name prediction results of DIRTY in Listing 2b and VARBERT in Listing 2c. In the decompilation output, we removed type casts (a keyboard shortcut in IDA) for easier comparison between the outputs. In both decompilation outputs, variables that are predicted correctly are styled ***italic green*** while variables that are predicted incorrectly are styled **red**.

The function in Listing 2a has three developer-intended variables: `clock`, `enabled`, and `old`. DIRTY in Listing 2b correctly predicted *`enabled`*, but failed for the others, predicting `clock` as **`uc`** and `old` as **`status`**. However, VARBERT in Listing 2c correctly predicted *`clock`* and *`old`* but was incorrect in predicting **`enable`** for

```
void qemu_clock_enable(
     QEMUClock *clock, bool enabled) {



  bool old = clock->enabled;
  clock->enabled = enabled;
  if (enabled && !old) {
    qemu_clock_notify(clock);

  }


}
```

(a) Original source code.

```
unsigned __int64 qemu_clock_enable(
     __int64 uc, char enabled) {
  char status;
  unsigned __int64 v4;
  v4 = __readfsqword(Number);

  status = *(uc + Number);
  *(uc + Number) = enabled;
  if (enabled && status != Number) {
   qemu_rearm_alarm_timer(alarm_timer)
     ;
  }

  return __readfsqword(Number) ^ v4;
}
```

(b) DIRTY predicted variable names.

```
unsigned __int64 qemu_clock_enable(
     __int64 clock, char enable) {
  char old;
  unsigned __int64 v4;
  v4 = __readfsqword(Number);

  old = * (clock + Number);
  *(clock + Number) = enable;
  if (enable && old != Number) {
   qemu_rearm_alarm_timer(alarm_timer)
     ;
  }

  return __readfsqword(Number) ^ v4;
}
```

(c) VARBERT predicted variable names.

Listing 2: Example case study on the function `qemu_clock_enable` from the DIRT data set. 2a shows the actual source code for reference. 2b shows DIRTY's predictions and 2c shows VARBERT's predictions on the decompiled code. Note that variables that are predicted correctly are styled ***italic green*** while variables that are predicted incorrectly are styled **red**. Also note that variable `v4` in the decompilation output is an extraneous variable, and is therefore not predicted by either system.

`enabled`. Note that even though `enable` is very semantically similar to `enabled`, the strict design of our evaluation counts this as a failure when considering top-1 accuracy.

7.2. Mispredictions

VARBERT and DIRTY both consider a prediction correct if it is a strict match. But some of these mispredictions are as valuable as predictions. We looked into VARBERT's prediction results when running on VarCorpus-O0 (IDA) to understand its mispredictions. Table 8 shows examples of developer intended variables, the percentage of time that the correct variable was predicted along with the top-3 incorrect predictions. For instance, original variable name `buffer` was predicted correct 59.19% of the time, and the other incorrect predictions were `buf`, `UNK`, and `data`. For a human, the semantic meanings of these predictions are quite similar, however we count these predictions as mispredictions. We see a similar misprediction trend with the variable `position`, where `pos` and `position` are essentially the same word.

We also noticed that the model seemed to pick up on nuances of the English language. For example, the VARBERT prediction from Section 7.1 in Listing 2c: VARBERT predicted `enabled` as `enable`. These two words share the same root, but this is not a correct prediction.

8. Discussion

While VARBERT improves the feasibility of variable name prediction on real-world decompiled code by a sizable margin, we envision that this research challenge will still remain unsolved for some time. We discuss in this section limitations of VARBERT as well as potential future research directions that may lead to the ultimate solution of readable and useful decompilation.

TABLE 8: Common and uncommon variables, their probabilities of getting correctly predicted, and the probabilities of the Top-3 incorrect predictions of VARBERT on VarCorpus-O0 (IDA).

	Correct Predictions	Top-3 Incorrect Predictions		
`buffer`	`buffer` (59.19%)	`buf` (9.19%)	`UNK` (5.41%)	`data` (3.27%)
`len`	`len` (67.52%)	`UNK` (6.86%)	`size` (3.58%)	`n` (2.56%)
`tmp1`	`tmp1` (49.50%)	`tmp0` (14.19%)	`tmp2` (10.91%)	`tmp3` (4.76%)
`position`	`position` (33.19%)	`UNK` (13.30%)	`pos` (4.68%)	`index` (3.95%)
`srcsize`	`srcsize` (40.00%)	`len` (40.00%)	`UNK` (20.00%)	-

8.1. Threats to Validity

Insufficient decompilation quality. In the course of conducting this research we notice that modern binary code decompilers usually fail to generate satisfactory results for C++ binaries, or C binaries that were compiled with optimizations enabled. This is because binary decompilation is its own research area with many unsolved research problems. Variable name prediction can only build upon correct identification of variables in decompilation output. When decompilers fail to yield sufficiently good results, especially when many human-created variables are not identified, VARBERT (and other solutions) cannot predict variable names for variables that do not exist in decompiled code.

Unrepresentative corpora. A key assumption underpinning statistical machine learning is that the training set must be of an independent and identical distribution as real-world samples. We built our corpora by collecting C and C++ packages in package repositories of two major Linux distributions. The corpora may not be representative for executables on other platforms, such as Windows or MacOS.

Decompiler limitations. Our type-stripping technique depends on the decompilers' ability to preserve variables during decompilation. Unfortunately, many decompilers do not even parse DWARF debug symbols, which is why we

use IDA and Ghidra in our evaluation. Using VARBERT on other decompilation output would require resorting to other lower-accuracy variable-matching techniques when working on the output of those decompilers, which may significantly degrade the performance of VARBERT.

Near Duplicates. We use hash-based deduplication for removing duplicates from the data set and remove exact duplicate functions after some preprocessing. Based on our analysis, similarity-based deduplication technique, which aims to eliminate near duplicates, is insensitive to subtleties of the code. This technique might mistakenly identify two functions as near-duplicates solely based on subtle textual variations, despite the two functions exhibiting different functionalities. Therefore, elimination of near-duplicates in code is an interesting research problem that requires more in-depth study.

9. Related Work

Binary decompilation. Decompilation was originally referred to as "reverse compilation" by Cifuentes in a seminal thesis [11]. Critical problems that binary code decompilation must solve are (control-flow) structural analysis and variable type inference. Phoenix first proposed semantics-preserving structural analysis [50]. Dream implemented a goto-free structural recovery algorithm [62]. Gussoni et al. proposed an approach that structures binary-level control flow graphs with zero goto statements [24]. In the most recent work, SAILR [8] identifies the root causes of gotos and inverts goto-inducing compiler optimizations, resulting in decompiled code which is closer to the source code. Regarding variable type inference, TIE [37], retypd [45], and Osprey [66] are recent work for inferencing variable types on binary programs. Advances in binary decompilation benefit VARBERT by providing higher-quality decompilation output that resembles human-developed source code.

Neural models in binary decompilation. Recently, researchers have been applying deep learning in decompiling binary code or improving the decompilation result. Katz et al. used recurrent neural networks to decompile binary code snippets into C code [33]. Coda is a recent end-to-end neural-based approach to decompilation [21]. While they have achieved promising results, it is still too early to decompile reasonably sized binary programs purely with neural models.

More research projects aim to use neural networks to improve the quality of binary reverse engineering results or decompilation output. Debin is the first attempt in using machine learning to predict debug information for stripped binary code [25]. DIRE focuses on predicting variable names in decompilation output [36]. NFRE predicts function names on stripped binaries by learning from a large corpus of stripped binaries [22]. DIRECT improved upon DIRE by not requiring an AST for the decompilation output and only relying on text tokens, and it was evaluated on DIRE's data set [44]. DIRTY advances the field by predicting both variable names and types on decompiled code [10]. VARBERT directly improves on DIRE by first introducing a transfer-learning-based model, which addresses the fundamental challenge in data set building. VARBERT also proposes a new data set VarCorpus that alleviates many key issues in DIRE and DIRTY's original training and test corpus. Finally, VARBERT expands variable name prediction to multiple decompilers (IDA and Ghidra) and optimization levels, while DIRE and DIRTY only support IDA and `-O0` compiler optimization.

10. Conclusion

We propose a new solution VARBERT for predicting meaningful variable names in decompilation output. VARBERT is based on transfer learning, which acquires knowledge from a large corpus of human-developed source code and fine-tunes for the task of variable name and origin prediction in decompilation output from two decompilers (IDA and Ghidra). During our research, we built a new data set that is more extensive than existing data sets. We demonstrate that VARBERT outperforms both DIRE and DIRTY on this new data set. Our research corrects the existing understanding of the research progress on the topic of variable name prediction in decompiled code, establishes new baselines, and, by releasing our research artifacts to the public, will foster new research on this topic.

Acknowledgement

This project has received funding from the following sources: Defense Advanced Research Projects Agency (DARPA) Contracts No. HR001118C0060, FA875019C0003, N6600120C4020, and N6600122C4026; Advanced Research Projects Agency for Health (ARPA-H) Contract No. SP4701-23-C-0074; the Department of the Interior Grant No. D22AP00145-00; the Department of Defense Grant No. H98230-23-C-0270; and National Science Foundation (NSF) Awards No. 2146568 and 2232915.

References

[1] DIRE evaluation dataset. https://zenodo.org/record/3403077.
[2] DIRTY evaluation dataset. cmu-itl.s3.amazonaws.com/dirty/dirt.tar.gz.
[3] huzecong/ghcc: GitHub cloner & compiler. https://github.com/huzecong/ghcc.
[4] The log of a CI run for dirty on GitHub Actions. https://github.com/CMUSTRUDEL/DIRTY/runs/6116940575.
[5] Whole word masking scheme in bert. https://github.com/google-research/bert.
[6] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.
[7] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985, 2019.
[8] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy sailr! there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation. In *Proceedings of the USENIX Security Symposium*, August 2024.
[9] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1–10, 2013.
[10] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting Decompiler Output with Learned

Variable Names and Types. In *Proceedings of the USENIX Security Symposium*, August 2022.
[11] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.
[12] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.
[13] The Ghidra decompiler. The Ghidra decompiler, 2022. https://ghidra-sre.org/.
[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.
[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
[16] Chris Donahue, Mina Lee, and Percy Liang. Enabling language models to fill in the blanks. *arXiv preprint arXiv:2005.05339*, 2020.
[17] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*, 2020.
[18] Lukás Ďurfina, Jakub Křoustek, and Petr Zemek. Psybot malware: A step-by-step decompilation case study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 449–456. IEEE, 2013.
[19] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. In *International Conference on Information Security and Assurance*, pages 72–86. Springer, 2011.
[20] Mobius Strip Reverse Engineering. An exhaustively-analyzed IDB for ComRAT v4, 2021. https://www.msreverseengineering.com/blog/2020/8/31/an-exhaustively-analyzed-idb-for-comrat-v4.
[21] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. A neural-based program decompiler. *arXiv:1906.12029 [cs]*, Jun 2019. arXiv: 1906.12029.
[22] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 607–619, 2021.
[23] Yuxian Gu, Zhengyan Zhang, Xiaozhi Wang, Zhiyuan Liu, and Maosong Sun. Train no evil: Selective masking for task-guided pre-training. *arXiv preprint arXiv:2004.09733*, 2020.
[24] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled c code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, page 637–651. ACM, Oct 2020.
[25] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.
[26] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
[27] SA Hex-Rays. Hex-rays decompiler, 2013.
[28] Hopper. Hopper, 2022. https://www.hopperapp.com/.
[29] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering*, pages 401–412, 2022.
[30] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, pages 20–30, 2018.
[31] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS), Virtual*, 2021.
[32] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. Clik on plcs! attacking control logic with decompilation and virtual plc. In *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*, 2019.
[33] Deborah S. Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, page 346–356. IEEE, Mar 2018.
[34] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[35] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Matthew Kelcey, Jacob Devlin, Kenton Lee, Kristina N. Toutanova, Llion Jones, Ming-Wei Chang, Andrew Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: a benchmark for question answering research. *Transactions of the Association of Computational Linguistics*, 2019.
[36] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.
[37] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, page 18, Feb 2011.
[38] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. *arXiv preprint arXiv:2107.06499*, 2021.
[39] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.
[40] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery – a case study. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS22)*, ASIACCS 22, June 2022.
[41] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. *arXiv preprint arXiv:1708.00107*, 2017.
[42] Omid Mirzaei, Roman Vasilenko, Engin Kirda, Long Lu, and Amin Kharraz. Scrutinizer: Detecting code reuse in malware via decompilation and machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 130–150. Springer, 2021.
[43] Binary Ninja. Binary Ninja, 2022. https://binary.ninja/.
[44] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. DIRECT: A Transformer-based Model for Decompiled Variable Name Recovery. *Workshop on Natural Language Processing for Programming (NLP4Prog)*, 2021.
[45] Matthew Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 27–41, Mar 2016.
[46] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.
[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
[48] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.
[49] Andrea Schankin, Annika Berger, Daniel V Holt, Johannes C Hofmeister, Till Riedel, and Michael Beigl. Descriptive compound identifier names improve source code comprehension. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 31–3109. IEEE, 2018.
[50] Edward J Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, page 17. USENIX Association, 2013.
[51] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.
[52] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.
[53] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.
[54] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 347–362. ACM, Oct 2017.
[55] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, Feb 2011.

[56] Hao Tan and Mohit Bansal. Lxmert: Learning cross-modality encoder representations from transformers. *arXiv preprint arXiv:1908.07490*, 2019.
[57] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th USENIX Security Symposium (USENIX Security 20)*, page 1875–1892, 2020.
[58] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.
[59] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.
[60] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691. IEEE, 2015.
[61] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.
[62] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015.
[63] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5754–5764, 2019.
[64] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.
[65] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *arXiv preprint arXiv:1411.1792*, 2014.
[66] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, page 813–832. IEEE, May 2021.

Appendix A.

A.1. Issues with Prior Data Sets

While investigating the prior data sets DIRE-DataSet and DIRT, we noticed several issues that we detail here.

Significant overlap between test and training sets. In DIRE-DataSet and DIRT, the test and training sets exhibit a significant overlap. Approximately 79.9% of functions in DIRE-DataSet's test set exist in the training set. Therefore, the overall accuracy (74.3%) that DIRE reported does not reflect the true performance of their model as a variable name prediction solution. Instead, it only demonstrates *how well DIRE identifies functions that were known to their model during training*.

DIRE authors were aware of this issue and reported an accuracy of 35.3% for a body-not-in-train test set, where they eliminated from the test set all functions that exist in the training set. Similar is the case for DIRTY: The overlap in their test and train set is 65.5% [10, Table 11] with variable name prediction accuracy of 35.1% for body-not-in-train functions [4]. These numbers reflect the real performance for DIRE and DIRTY to generalize to new decompiled functions.

High number of duplicated functions. Another issue we discovered in DIRE-DataSet and DIRT is the high number of duplicated functions. 683K (67.6%) functions among DIRE's 1M functions that are used for training DIRE are duplicates. Similarly, 1M (56.9%) out of 1.8M are duplicates in training set of DIRTY.

TABLE 9: Numbers of total and duplicated functions in DIRE-DataSet and DIRT. Overall = Train + Test. We do not include the development set of DIRE-DataSet and DIRT (because it did not impact DIRE's and DIRTY's training or testing), which has about 100K functions. Overlap is the duplicate functions between Test set and Train set of DIRE-DataSet and DIRT.

	Functions	DIRE	DIRTY
Overall	Total	1,214,930	1,872,420
	Duplicates	817,298 (67.2%)	1,015,768 (54.2%)
	Deduplicated	397,632	856,652
Train	Total	1,011,054	1,668,544
	Duplicates	683,814 (67.6%)	950,990 (56.9%)
	Deduplicated	327,240	717,554
Test	Total	124,179	203,876
	Duplicates	53,787 (43.3%)	64,778 (31.7%)
	Deduplicated	70,392	139,098
Overlap	Total	99,317	133531
	De-duplicated	46,316	71,967

The nature of neural network models will cause models trained on these data sets to learn more from frequently appearing functions and ignore those less frequently appearing functions.

High failure rate of variable matching. Another issue with DIRE-DataSet and DIRT is the high failure rates of variable matching. We understand that decompilers are unable to recover all of the variable names from source code, however oftentimes the variable matching algorithm of DIRE and DIRT's corpus creation is unable to match the original human-created variable to the decompiler-assigned variable. We performed analysis on a sample of functions from each corpus in Section 5.4. Incorrect matching or missing matches between human-created variable names and decompiler-generated names impacts the ground-truth of the data set.

In conclusion, we believe DIRE-DataSet and DIRT are unsuitable for future research on predicting variable names in decompiled code (but the data sets may still be useful for other purposes).

A.2. Analysis of Quality Issues in DIRE-DataSet and DIRT

For the interested readers, we include some detailed analysis of the quality issues with DIRE-DataSet and DIRT.

Binaries. An essential difference between VarCorpus and DIRT (also DIRE-DataSet) is the definition of "binaries." DIRT and DIRE-DataSet use compiled object files whereas VarCorpus use full binary executables or libraries. The reason being, both DIRT and DIRE-DataSet use ghcc [3] to build packages, and ghcc predominantly outputs and generates object files by compiling every `.c` file instead

of properly building each C project. Thanks to the DIRE authors, we verified our hypothesis with their raw data set of binaries, and we found over 90% of binaries in their data set are object files.

Duplicated functions. Table 9 shows an overview of DIRT and DIRE-DataSet in terms of numbers of functions in each data set split. There are similar percentages of duplicated functions within each split (with the only exception being the test set of DIRT, which may be an outlier).

We believe that the cause of this high duplication in DIRT and DIRE-DataSet is twofold: (1) multiple copies of the same projects with same or different versions or similar projects and (2) duplicate functions across object files. We verified this by examining the GitHub project list used in DIRT. For example, 379 GitHub users had a "linux" repository, which are likely due to the same project pushed multiple times onto GitHub. DIRE attempted to remove duplicate binaries by hashing [36, Section V]. However, because the object file binaries are compiled with debug information, which *includes the source code path in the compiled object file binary*, identical source code projects will likely produce object file binaries with different hashes.

A.3. Hyper Parameters

We optimized our models using BERTAdam [15, 34] with the following parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e{-}6$, and L_2 weight decay of 0.01. We warmed up over the first 10,000 steps to a peak value of $1e{-}4$ and then linearly decayed. We set the dropout to 0.1 on all layers and attention weights. Our activation function was GELU [26]. We trained all our models for 30 epochs, except for experiments about evaluating DIRE on DIRE-DataSet where we trained for 60 epochs (so that our results are comparable to the ones in the DIRE paper, where models were trained for 60 epochs). We performed training and experiments on four and eight 80GB Nvidia A100s, six and eight 49GB Nvidia RTX A6000s and two 24GB NVIDIA GeForce RTX 3090 GPUs.

A.4. Comparison of prior work and VarBERT on DIRE-DataSet and DIRT

Fixing DIRE-DataSet and DIRT. To better understand the impact of duplicated functions on neural models, we attempted to address the duplication problem in the DIRE-DataSet and DIRT corpora by fully deduplicating functions to create fixed data sets that we call DIRE-DataSet-dedup and DIRT-dedup. We deduplicated individually each of the training, validation, and test set of DIRE-DataSet and DIRT. As a result, the number of functions in the training data was reduced from 1M to 327K approximately in DIRE-DataSet-dedup and from 1.6M to 717K approximately in DIRT-dedup. In Section A.4, we will evaluate VARBERT on these two new corpus.

For completeness, we compare the performance of VARBERT, DIRE [36], DIRECT [44], and DIRTY [10] on the original DIRE-DataSet, the fixed DIRE-DataSet-dedup, original DIRT, and the fixed DIRT-dedup data sets. Additionally, we also evaluate VARBERT without pre-training on HSC, indicated as VARBERT (no PT). The goal of this experiment is to demonstrate how much of VARBERT's performance gain is due to the adoption of a new model.

TABLE 10: Results of DIRE, DIRTY, DIRECT, and VARBERT on DIRE-DataSet, the fixed DIRE-DataSet-dedup, DIRT, and the fixed DIRT-dedup. To understand the impact of pre-training, we also ran VARBERT without pre-training on HSC, indicated as VARBERT (no PT). To save computation resources, we did not re-run experimental results for directly comparable results, and results of models marked with an asterisk* are taken from the indicated paper.

Data set	Model	Top-1 Accuracy (%)
DIRE-DataSet	DIRE [36, Table 1]*	35.3
	DIRTY [10, Table 4]*	42.8
	DIRECT [44, Table 1]*	42.8
	VARBERT (no PT)	51.24
	VARBERT	**61.49**
DIRE-DataSet-dedup	DIRE	38.29
	VARBERT	**61.73**
DIRT	DIRE [10, Table 4]*	31.8
	DIRTY [10, Table 4]*	36.9
	VARBERT (no PT)	47.11
	VARBERT	**51.28**
DIRT-dedup	DIRTY	54.06[6]
	VARBERT	**51.02**

Result. Table 10 shows the result of this experiment. Note that to save computational resources in cases where results from papers were directly comparable (on the same dataset) we included the results from prior papers.

The results show that, on the original DIRE-DataSet, VARBERT *without pre-training* outperformed prior work: 35.3% top-1 accuracy for DIRE, 42.8% for DIRTY, 42.8% for DIRECT, and 51.24% for VARBERT without pre-training. However, VARBERT *with* pre-training increased the top-1 accuracy to 61.49%. This same result also held for the original DIRT dataset: 31.8% top-1 accuracy for DIRE, 36.9% for DIRTY, and 47.11% for VARBERT without pre-training. And VARBERT *with* pre-training increased the top-1 accuracy to 51.28%.

To measure the impact of duplication in the DIRE-DataSet and DIRT datasets, we re-ran DIRE and VARBERT on DIRE-DataSet-dedup and DIRTY and VARBERT on DIRT-dedup. The results in Table 10 show that DIRE's accuracy improves on the fixed DIRE-DataSet-dedup from 35.3% to 38.29%, while VARBERT maintains a high accuracy of 61.49%. Likewise, DIRTY's accuracy improves from 36.9% to 54.06% on the fixed DIRT-dedup, while VARBERT's accuracy is 51.02%.

A.5. User Study Details

A.5.1. A Sample Recruitment Email

Dear CSE365/CSE545 hackers,

We are conducting a study on AI-assisted binary decompilation to better understand the effect of meaningful variable names in decompiled functions. The main form of this study is a survey of computer science students who will be asked to read up to 15 decompiled functions and answer multiple choice questions. We invite you to participate in our study.

This study will be conducted remotely on a website. Upon successful completion, you will receive a $50 Amazon gift card as compensation.

We will invite 15 hackers (first reply, first serve!)[7] to lend us up to 60 minutes of their precious time. If you are interested, please reply to this email using either your ASU or your personal email address, wherever you want to receive the compensation. We will then send you a link within the next three days for starting the study.

Please feel free to email us at [REDACTED] or [REDACTED] should you have any questions.

Thank you!

Best, [REDACTED]

A.5.2. An Example Question

Question: Which option best describes the functionality of the the following function (as shown in Listing 3)?

A Calculating the entropy of some data. The address of the data is provided by the first function parameter. The size of the data is provided by the second function parameter.

B Calculating the length of a string. The address of the string is provided by the first function parameter. The length of the string is provided by the second function parameter.

C Calculating the chi-square value of some data. The address of the data is provided by the first function parameter. The size of the data is provided by the second function parameter.

D Calculating the size of a dictionary (or a key-value mapping). The address of the dictionary is provided by the first function parameter. The length of the dictionary is provided by the second function parameter.

7. We intentionally put a lower number here to quickly get students' responses. We invited 34 students in the end.

```
double __fastcall sub_2DC64(double *var_7, int var_4, int
↪  var_1, int var_5, double *var_9)
{
  int var_3;
  double var_10;
  double var_6;
  double var_2;
  double var_8;

  if ( var_1 <= 0 )
    return 0.0;
  var_2 = 0.0;
  for ( var_3 = 0; var_3 < var_4; ++var_3 )
  {
    if ( var_5 >= 0 )
    {
      var_6 = 1.0;
    }
    else if ( *var_7 >= 0.0 )
    {
      if ( *var_7 == 0.0 )
        var_6 = 1.0;
      else
        var_6 = 1.0 / *var_7;
    }
    else
    {
      var_6 = -1.0 / *var_7;
    }
    var_8 = *var_7++ - *var_9++;
    var_2 = var_6 * var_8 * var_8 + var_2;
  }
  var_10 = var_2 / var_1;
  *&var_10 = var_10;
  return var_10;
}
```

(a) The raw decompiled function.

```
double __fastcall sub_2DC64(double *data, int ndim, int
↪  nfree, int mode, double *dfit)
{
  int i;
  double result;
  double weight;
  double chisq;
  double diff;

  if ( nfree <= 0 )
    return 0.0;
  chisq = 0.0;
  for ( i = 0; i < ndim; ++i )
  {
    if ( mode >= 0 )
    {
      weight = 1.0;
    }
    else if ( *data >= 0.0 )
    {
      if ( *data == 0.0 )
        weight = 1.0;
      else
        weight = 1.0 / *data;
    }
    else
    {
      weight = -1.0 / *data;
    }
    diff = *data++ - *dfit++;
    chisq = weight * diff * diff + chisq;
  }
  result = chisq / nfree;
  *&result = result;
  return result;
}
```

(b) The decompiled function augmented with predicted variable names.

Listing 3: An example function pair in the user study.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper provides a new large-scale dataset and proposes to use the BERT model to predict variable names in decompiled code. The proposed method, VarBERT, leverages debug information to generate decompiled code with correct variable names labeled, then randomly masks variable names to train the BERT model. It also identifies defects in the dataset used in previous works and outperforms related works on the new database.

B.2. Scientific Contributions

- Provides a New Data Set For Public Use
- Provides a Valuable Step Forward in an Established Field
- Independent Confirmation of Important Results with Limited Prior Research
- Creates a New Tool to Enable Future Science

B.3. Reasons for Acceptance

1) This paper identifies overlap, duplication, and variable matching issues existing in the dataset used by SOTA works.
2) This work provides a new large-scale, cross-optimization variable names prediction dataset in which variables are more accurately matched.
3) The presented tool largely outperforms existing SOTA methods.
4) The evaluation is comprehensive and accompanied with a user study.
5) Efforts on open science and reproducibility.