DIRE and its Data: Neural Decompiled Variable Renamings with Respect to Software Class

LUKE DRAMKO, JEREMY LACOMIS, and PENGCHENG YIN, Carnegie Mellon University, USA
EDWARD J. SCHWARTZ, Carnegie Mellon University Software Engineering Institute, USA
MILTIADIS ALLAMANIS, Microsoft Research, UK
GRAHAM NEUBIG, BOGDAN VASILESCU, and CLAIRE LE GOUES, Carnegie Mellon University, USA

The *decompiler* is one of the most common tools for examining executable binaries without the corresponding source code. It transforms binaries into high-level code, reversing the compilation process. Unfortunately, decompiler output is far from readable because the decompilation process is often incomplete. State-of-the-art techniques use machine learning to predict missing information like variable names. While these approaches are often able to suggest good variable names in context, no existing work examines how the selection of training data influences these machine learning models. We investigate how data provenance and the quality of training data affect performance, and how well, if at all, trained models generalize across software domains. We focus on the variable renaming problem using one such machine learning model, DIRE. We first describe DIRE in detail and the accompanying technique used to generate training data from raw code. We also evaluate DIRE's overall performance without respect to data quality. Next, we show how training on more popular, possibly higher quality code (measured using GitHub stars) leads to a more generalizable model because popular code tends to have more diverse variable names. Finally, we evaluate how well DIRE predicts domain-specific identifiers, propose a modification to incorporate domain information, and show that it can predict identifiers in domain-specific scenarios 23% more frequently than the original DIRE model.

Additional Key Words and Phrases: Machine Learning, Decompilation, Data Provenance

1 INTRODUCTION

Decompilers, i.e., tools such as Hex-Rays [23] and Ghidra [18], which help translate binaries into code that resembles high-level languages such as C, are essential for software reverse engineers looking to predict the behavior of malware [15, 57, 58], discover vulnerabilities [49, 53, 58], and patch bugs in legacy software [49, 53]. Decompilers use sophisticated program analysis and heuristics to reconstruct information about a program's variables, types, functions, and control flow structure, effectively increasing program comprehension for reverse engineers who would otherwise work directly with binaries or with assembly code.

Still, the output of all existing decompilers is far from readable, as decompilation is often incomplete. Compilers discard source-level information and lower its level of abstraction in the interest of binary size, execution time, and even obfuscation. As a result, comments, variable names, user-defined types, and idiomatic structure are all lost at compile time, and are typically unavailable in decompiler output. In particular, variable names, which are highly important for code comprehension and readability [17, 32], become nothing more than arbitrary placeholders such as `VAR1` and `VAR2`. While many compilers offer the option to include debugging information that preserves variable names in the resulting executable, malware authors and commercial vendors typically set compiler flags to prevent this in an effort to frustrate security researchers or protect corporate intellectual property.

Authors' addresses: Luke Dramko, lukedram@andrew.cmu.edu; Jeremy Lacomis, jlacomis@cs.cmu.edu; Pengcheng Yin, pcyin@cs.cmu.edu, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, Pennsylvania, USA, 15213; Edward J. Schwartz, eschwartz@cert.org, Carnegie Mellon University Software Engineering Institute, Pittsburgh, Pennsylvania, USA; Miltiadis Allamanis, miallama@microsoft.com, Microsoft Research, UK, Cambridge; Graham Neubig, gneubig@cs.cmu.edu; Bogdan Vasilescu, vasilescu@cmu.edu; Claire Le Goues, clegoues@cs.cmu.edu, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, Pennsylvania, USA, 15213.

To improve on existing decompilers, researchers have been developing a suite of deep-learning based techniques to automatically infer informative variable names (and in some cases also user-defined types) in a given context, e.g., DIRE [31], DIRECT [40], and DIRTY [10]. Since programmers tend to write similar code in similar contexts [14, 24], such techniques can learn to infer *natural* names, even if not necessarily the original ones pre-compilation. These techniques can be applied as a post-processing step to decompilation, taking the output of decompilers as input and automatically refactoring it. While the specifics of the learning approaches vary, the key idea that enables learning such models is that one can generate arbitrary amounts of *parallel* training data given access to open-source software repositories and standard compilers and decompilers; these data consist of pairs of original source, with presumably human-written variable names, and corresponding decompiler output, with placeholder variable names.

However, while results from prior work [10, 31, 40] show that, on average, the performance of existing techniques on decompiled open-source binaries is high, i.e., it is often possible to recover the exact variable names chosen by the authors of that code pre compilation/decompilation, open questions remain about how the provenance and quality of the training data affect performance, and how well, if at all, trained models generalize across software domains. We argue that answers to these questions are direly needed. Indeed, data quality is a universal issue that affects machine learning models in all domains. In addition, closer to our problem, researchers have already raised concerns about neural models of code, e.g., that they are negatively affected by code duplication in the training data [1, 34], do not scale well on code completion tasks because of large vocabularies and out-of-vocabulary issues [28], are not robust to semantic-preserving program transformations [43], and do not readily generalize to other downstream tasks [27]. In essence, while much work, including the original ASE '19 paper [31], has focused on increasing the power of *models* of operations on code, here we focus on the orthogonal, underexplored issue of better selecting and harnessing the *data* used to train these models to increase performance.

Specifically, in this paper we investigate how data quantity, quality, and software domain provenance affect the performance of the neural identifier renaming technique DIRE [31]. Issues of data quality and model robustness have, thus far, been underexplored by researchers interested in the decompiled identifier renaming problem. Our work is an extension of an ASE 2019 paper [31] where DIRE was originally presented, the technical details of which we reproduce verbatim here, for completeness. These include the technical description of DIRE and the approach used to generate its parallel training data, plus a data-provenance-agnostic evaluation of DIRE, all originally reported in the ASE 2019 paper [31]. In addition, relative to the conference paper, in this paper we make two major new contributions. First, we study how the performance of DIRE varies when trained on decompiled binaries from more versus less popular repositories (using GitHub repository stars, the measure of popularity, also as a loose proxy for code quality), and contribute empirical results showing that training on highly-starred code leads to a more generalizable model because such code tends to be more diverse, i.e., it has a higher-entropy distribution of variable names. Second, we evaluate how well DIRE learns to predict domain-specific identifiers, propose a modification to incorporate domain information, and show empirically that the modified version can predict identifiers in domain-specific scenarios 23% more frequently than the original DIRE model.

Note that contemporaneously with our current work, two competing decompiler identifier renaming techniques have been proposed, DIRECT [40] and DIRTY [10]. The two use different learning approaches than DIRE (both use a transformer-based architecture [55]) and report higher accuracy than DIRE when compared directly; however, neither addresses the data provenance and model robustness research questions of interest here, although the approaches to train all three systems are fundamentally similar. Therefore, our current work (i.e., the new research questions and experiments relative to the ASE 2019 paper [31]) can be seen as orthogonal.

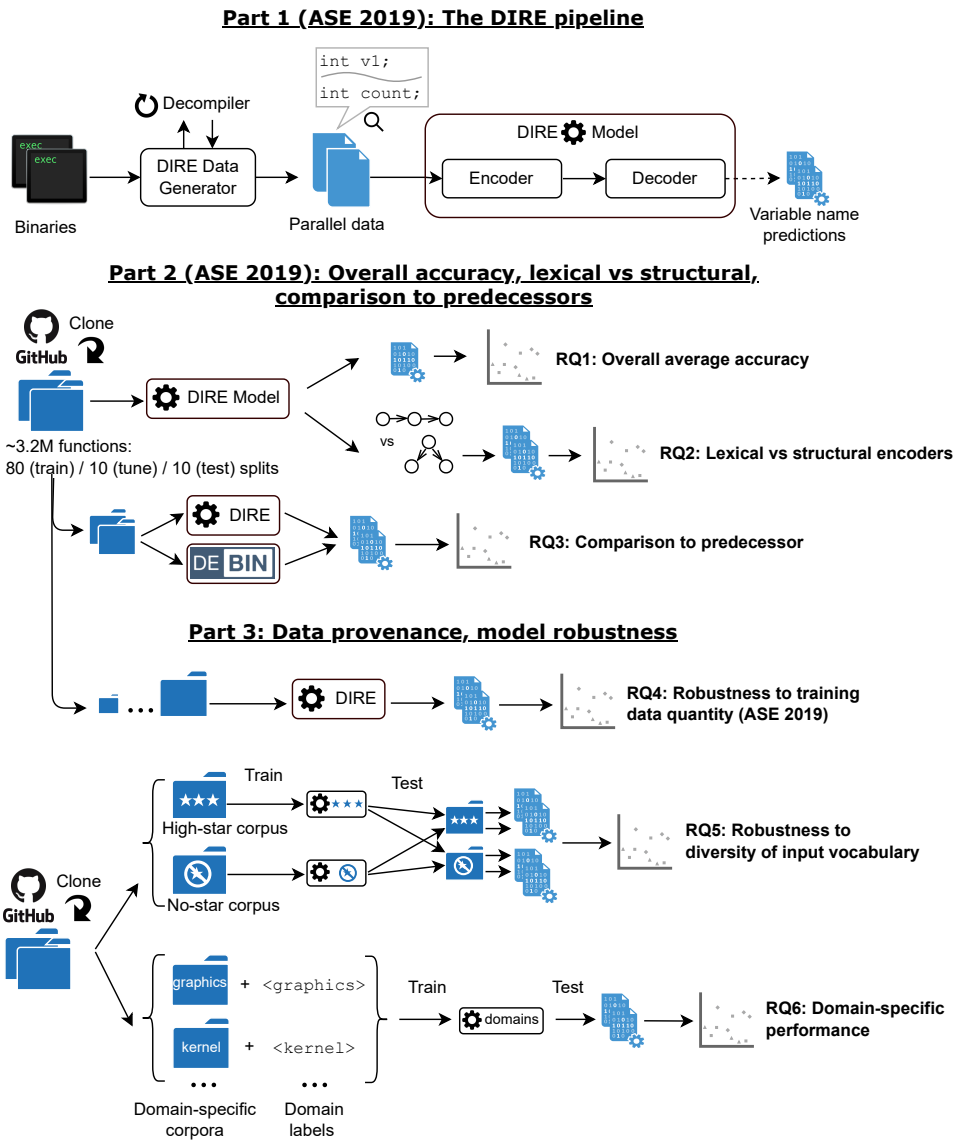


Fig. 1. Overview of our paper.

2 PAPER OVERVIEW

This paper is organized in three main parts, summarized in Fig. 1.

In the **first part**, we cover the technical details of the DIRE approach. Specifically, in Section 3 we provide background on the decompilation process, the challenges it raises, and related work to help address these challenges, as well as background on statistical models of source code. Next, we present DIRE (the **D**ecompiled **I**dentifier **R**enaming **E**ngine) in Section 4, our technique for assigning meaningful names to decompiled variables. DIRE is an ensemble of two different

machine learning models, one which captures sequential lexical information about the source code and another which captures graphical structural information. Finally, in Section 5, we address the challenges with obtaining and cleaning DIRE's training data, a nontrivial task.

In the **second part** (Section 6), we answer our first set of research questions about the overall efficacy of the DIRE technique, a breakdown by each of the two model components, and a comparison to its closest predecessor; these results were all part of the original ASE 2019 conference paper [31].

Specifically, in RQ1 we establish a baseline for DIRE's performance on our corpus, without focusing on any dataset characteristics.

- RQ1: How effective is DIRE at assigning names to variables in decompiled code?

DIRE incorporates multiple streams of information when making predictions. Some approaches [26], inspired by natural language processing techniques, treat source code as a sequence of tokens, as might be produced from a lexer. However, source code has a formally defined structure. In designing DIRE, we incorporated this structural information as well, using enhanced abstract syntax tree representations in addition to a lexical representation. In RQ2, we investigate how each of these two components influences and contributes to DIRE's overall performance.

- RQ2: How does each component of DIRE contribute to its efficacy?

In RQ3, we compare DIRE against its closest predecessor.

- RQ3: Is DIRE more effective than prior approaches?

Finally, in the **third part** (Section 7), which is almost entirely new relative to the conference paper, we answer our second set of research questions about the impact of data provenance on the performance of DIRE. Prior work has generally not focused on the curation of training data for such models, instead implicitly treating all such data as equivalent. As discussed above, we hypothesize that even if all training data for such models comes from one open source super-repository like GitHub, because of the inherent diversity of projects hosted there, there are systematic differences between data sampled in different ways that will significantly affect the behavior and performance of decompiled variable renaming models.

To test this general hypothesis, we address research questions about how the characteristics of the training set influence the performance of DIRE. Specifically, in RQ4, we investigate how DIRE performs in data-constrained environments, given that training deep neural networks can be computationally expensive, and it may not always be feasible to train DIRE on a large data set.

- RQ4: How does the quantity of data influence the efficacy of DIRE?

In RQ5, we study to what extent there are noticeable differences in the performance and behavior of DIRE when trained on data coming from popular, "high-star" repositories compared to data from unpopular, "no-star" repositories. On transparent, social coding platforms like GitHub, project popularity as indicated by the number of repository stars indicates more community interest, attention, and ultimately pressure on the project owners to follow software engineering best practices, and to provide and maintain high-quality code [9].

- RQ5: How does the provenance of training data influence the efficacy of DIRE?

Some identifiers occur more frequently in context of other identifiers. For example, a variable named `url` is more likely to occur near a variable named `http_request` than in code in general. We define a software domain as a collection of contexts like this that share a similar purpose. Examples of software domains include networking and graphics. Software domains, however, can be very unevenly distributed on GitHub, leading to potential bias — the model may

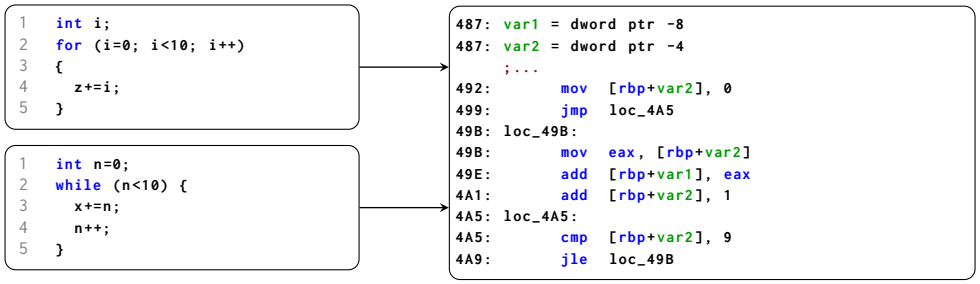


Fig. 2. Two different C loops that compile to the same assembly code. Note the normalized structure and names.

learn to disproportionately suggest names from overrepresented domains. We account for this effect by modifying DIRE to make it domain-aware and evaluate this modification.

- RQ6: How well can DIRE perform on diverse code from various domains if it is made aware of its software domain?

The paper ends with a discussion of threats to validity in Section 8 and conclusions in Section 9.

3 BACKGROUND

Before diving into the technical details of our approach, we start with some background on decompilation, statistical models of source code, and the two particular classes of deep learning models we rely on, recurrent neural networks (RNNs) and gated-graph neural networks (GGNNs).

3.1 Decompilation

At a high level, a compiler generates binaries from source using a pipeline of processing stages, and decompilers try to reverse this pipeline using various techniques [29]. Typically, a binary is first passed through a platform-specific *disassembler*. Next, assembly code is typically *lifted* to a platform-independent intermediate representation (IR) using a binary-to-IR lifter. The next stage is the heart of the decompiler, and is where a number of program analyses are used to recover variables, types, functions and control flow abstractions, which are ultimately combined to reconstruct an abstract syntax tree (AST) corresponding to an idiomatic program. Finally, a code generator converts the AST to the decompiled output.

Decompilation is more difficult than compilation, because each stage of a compiler loses information about the original program. For example, the lexing/parsing stage of the compiler does not propagate code comments to the AST. Similarly, converting from the AST to IR can lose additional information. This loss of information allows multiple distinct source code programs to compile to the same assembly code. For example, the two loops in Fig. 2 are reduced to the same assembly instructions. The decompiler cannot know which source code was the original, but it does try to generate code that is *idiomatic*, using heuristics to increase code readability. For example, high-level control flow structures such as `while` loops are preferred over `goto` statements.

The choice of which code to generate is largely heuristic, but can be informed by the inclusion of DWARF debugging information [16]. This debugging information, which can optionally be generated at compile-time, greatly assists the decompiler by identifying function offsets, types of variables, identifier names, and user-defined structures and unions.

3.2 Statistical Models of Source Code

Identifiers play an important role in program comprehension [17, 32]. Researchers have employed the renaming of identifiers as a way to ensure that they are consistently named within the program [36] or are consistent with program semantics. However, the relationship between identifiers and program semantics is complex and difficult to describe explicitly. Thus, researchers have recently begun turning to statistical models of source code to capture these relationships, e.g., determining if a method's name is consistent with its body [37] or determining why an identifier was changed when a given piece of code was refactored [8].

A wide variety of statistical models for representing source code have been proposed based on the *naturalness* of software [14, 24]. This key property states that source code is highly repetitive given context, and is therefore predictable. Statistical models capture the implicit knowledge hidden within code, and apply it to build new software development tools and program analyses, e.g., for code completion, documentation generation, and automated type annotation [5].

Predicting variable names is no exception. Work has shown that statistical models trained on source code corpora can predict descriptive names for variables in a previously-unseen program, given the contextual features of the code in which the variable is used. These naming models can help to distill coding conventions [3] or analyze obfuscated code [44, 54]. Several classes of statistical models have been used for renaming, including n-grams [3, 54], conditional random fields (CRFs) [44], and deep learning models [4, 6, 7].

Two recent approaches aim to suggest informative variable names in decompiled code. Our prior work [26] proposed a lexical n-gram-based machine translation model that operates on decompiler output. That approach used heuristics to align variables in the decompiler output and original source, which are needed for training, and is able to exactly recover 12.7% of the original names in the test set.

Contemporaneously, He et al. [22] proposed a two-step approach that operates on a stripped binary rather than the decompiler output. First, the authors predict whether a low-level register or a memory offset maps to a variable at the source-level. Then, using structured prediction with CRFs, they predict names and types for the mapped variables. 63.5% of the variables in the test set for which the first step succeeded could be recovered exactly.

3.3 Neural Network Models

Our approach builds on two advances in statistical models for representing source code: recurrent neural networks (RNNs) and gated-graph neural networks (GGNNs).

3.3.1 Recurrent Neural Networks. RNNs are networks where connections between nodes form a sequence [45]. They are typically used to process sequences of inputs by reading in one element at a time, making them well-suited to modeling source code tokens. In this work, we use long short-term memory (LSTM) models [25], a variant of RNNs widely used in text processing. Formally, an LSTM has the following structure: given a sequence of tokens $\{x_i\}_{i=1}^{n}$, an LSTM $\overrightarrow{f}_{\text{LSTM}}$ processes them in order, maintaining a hidden state $\overrightarrow{\boldsymbol{h_i}}$ for each subsequence up to token x_i using the recurrent function $\overrightarrow{\boldsymbol{h_i}} = \overrightarrow{f}_{\text{LSTM}}(\text{emb}(x_i), \overrightarrow{\boldsymbol{h_{i-1}}})$, where $\text{emb}(\cdot)$ is an embedding function mapping x_i into a learnable vector of real numbers. As we will elaborate later in Section 4, we use two types of LSTMs in DIRE: encoding LSTMs and decoding LSTMs. An encoder LSTM reads the input sequence (e.g., a sequence of source code tokens, as in Section 4.2.1) and encodes it into continuous vectors, while a decoder LSTM takes these vectors and generates the output sequence (e.g., the sequence of predicted names for all identifiers, as in Section 4.3).

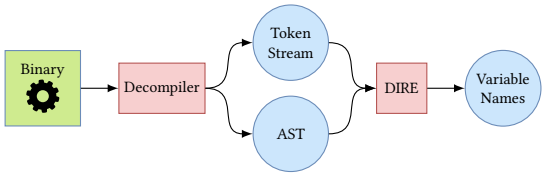


Fig. 3. High-level overview of our approach.

3.3.2 Gated-Graph Neural Networks. While LSTMs are useful for representing and generating sequences, they do not capture any additional structural information. Within the decompilation task, structured information provided by the AST is a natural information source about choice of variable names. For this purpose, we also employ structural encoding of the code using GGNNs, a class of neural models that map *graphs* to outputs [35, 48].

At a high level, GGNNs are neural networks over directed graphs. Initially, we associate each vertex with a learned or computed hidden state containing information about the vertex. GGNNs compute representations for each vertex based on the initial vertex information and the graph structure.

Formally, let $\mathcal{G} = \langle V, E \rangle$ be a directed graph describing our problem, where $V = \{v_i\}$ is the set of vertices and $E = \{(v_i \mapsto v_j, \mathcal{T})\}$ is the set of typed edges. Let $\mathcal{N}_\mathcal{T}(v_i)$ denote the set of vertices adjacent to v_i with edge type $\mathcal{T}$. In a GGNN, each vertex v_i is associated with a state $\boldsymbol{h}^g_{i,t}$ indexed by a time step t. At each time step t, the GGNN updates the state of all nodes in V via neural message passing (NMP) [19]. Concurrently for each node v_i at time t, NMP is performed as follows: First, for each $v_j \in \mathcal{N}_\mathcal{T}(v_i)$ we compute a message vector $\boldsymbol{m}^{v_j \mapsto v_i}_\mathcal{T} = \boldsymbol{W}_\mathcal{T} \cdot \boldsymbol{h}^g_{j,t-1}$, where $\boldsymbol{W}_\mathcal{T}$ is a type-specific weight matrix. Then, all $\boldsymbol{m}^{v_* \mapsto v_i}_*$ are aggregated, and summarized into a single vector $\boldsymbol{x}^g_i$ via element-wise mean (pooling):

$$\boldsymbol{x}^g_i = \mathsf{MeanPool}(\{\boldsymbol{m}^{v_j \mapsto v_i}_\mathcal{T} : v_j \in \mathcal{N}_\mathcal{T}(v_i), \forall \mathcal{T}\}).$$

Finally, the state of every node v_i is updated using a nonlinear activation function f: $\boldsymbol{h}^g_{i,t} = f(\boldsymbol{x}^g_i, \boldsymbol{h}^g_{i,t-1})$. GGNNs use a Gated Recurrent Unit (GRU) update function, $f_{\text{GRU}}(\cdot)$, introduced by Cho et al. [13]. By repeatedly applying NMP for T steps, each node's state gradually represents information about that node and its *context* within the graph. The computed states can then be used by a decoder, similarly to the LSTM-based decoder architectures. As in LSTMs, all GGNN parameters (parameters $f_{\text{GRU}}(\cdot)$ and the $\boldsymbol{W}_\mathcal{T}$s) are optimized along with the rest of the model.

4 THE DIRE ARCHITECTURE

We start with an overview of our approach, then dive into the technical details of each component.

4.1 Overview

We designed DIRE to work on top of a decompiler as a plugin that can automatically suggest more informative variable names. We use Hex-Rays, a state-of-the-art industry decompiler, though our approach is not fundamentally coupled to Hex-Rays and can be adapted to other decompilers.

Fig. 3 gives a high-level overview of our workflow. First, a binary is passed to a decompiler, which decompiles each function in the binary. For each function, our plugin traverses the AST, inserting placeholders at variable nodes. This produces two outputs: the AST and the tokenized code. These outputs are provided as input to our neural network model, DIRE, which generates unique variable names for each placeholder in the input. The decompiler output can then be rewritten to include the suggested variable names.

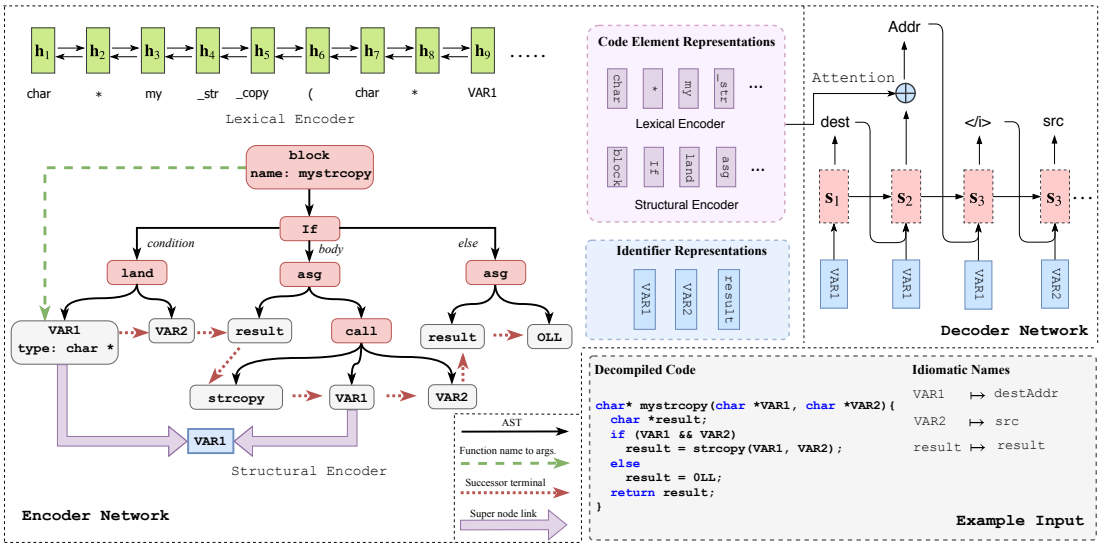


Fig. 4. Overview of DIRE's neural architecture. For clarity, we omit the data-flow links in the AST in the structural encoder.

Fig. 4 gives an overview of the neural architecture. DIRE follows an encoder-decoder architecture: An *encoder* neural network (Section 4.2) first encodes the decompiler's output—both the sequence of decompiled code tokens and its internal AST—and computes distributed representations (i.e., real-valued vectors, or *embeddings*) for each identifier and code element. These encoded representations are then consumed by a *decoder* neural network (Section 4.3) that predicts meaningful names for each identifier based on the contexts in which it is used.

The key takeaway is that DIRE uses both lexical information obtained from the tokenized code as well as structural information obtained from the corresponding ASTs. This is achieved by using two encoders—a *lexical encoder* (Section 4.2.1) and a *structural encoder* (Section 4.2.2)—to separately capture the lexical and structural signals in the decompiled code. As we will show, this combination of lexical and structural information allows DIRE to outperform techniques that rely on lexical information alone [26].

4.2 The Encoder Network

Each encoder network in DIRE outputs two sets of real-valued vector representations:

- A *code element representation* for each element in the decompiler's output. Depending on the type of the encoder, a code element will either be a token in the surface code (for the lexical encoder), or a node in the decompiler's internal AST (for the structural encoder).
- An *identifier representation* for each unique identifier defined in the input binary, which is a real-valued vector that represents the identifier in the neural network.

The lexical and structural representations are then merged to generate a unified encoding of the input binary (dashed boxes in Fig. 4). By computing separate representations for code elements and identifiers, the DIRE decoder can better incorporate the contextual information in the encodings of individual code elements to improve name predictions for the different identifiers; see Section 4.3.

4.2.1 Lexical Code Encoder. The lexical encoder sequentially encodes the tokenized decompiled code, projecting each token x_i into a fixed-length vector encoding $\boldsymbol{x_i}$. Specifically, the lexical encoder uses the sub-tokenized code as the input, where a complex code token (e.g., the function name `mystrcopy`) in Fig. 4) is automatically broken down into sub-pieces (e.g., `my`, `str`, and `copy`) using SentencePiece [30], based on sub-token frequency statistics. Sub-tokenization reduces the size of the encoder's vocabulary (and thus its training time), while also mitigating the problem of rare or unknown tokens by decomposing them into more common subtokens. We treat the placeholder and reserved variable names (e.g., `VAR1`, `VAR2`, and the decompiler-inferred name `result`) in the decompiler's output as special tokens that should not be sub-tokenized.

DIRE implements the lexical encoder using LSTMs (described in Section 3.3.1). We use a bidirectional LSTM: The forward network $\overrightarrow{f}_{\text{LSTM}}$ processes the tokenized code $\{x_i\}_{i=1}^{n}$ sequentially. The backward LSTM processes the input tokenized code in backward order, producing a backward hidden state $\overleftarrow{\boldsymbol{h_i}}$ for each token x_i. Intuitively, a bidirectional LSTM captures informative context around a particular variable both before and after its sequential location. *Element Representations* We encode a token x_i by concatenating its asssociated state vectors, i.e., $\boldsymbol{x_i} = [\overrightarrow{\boldsymbol{h_i}} : \overleftarrow{\boldsymbol{h_i}}]$, a common strategy in source code representations using LSTMs [5]. For a particular token x_i we compute the forward (resp. backward) representation using both its embedding and the hidden states of its preceding (resp. succeeding) tokens. This is important because the resulting encoding $\boldsymbol{x_i}$ captures both the local and contextual information of the current token and its surrounding code. To compute the *identifier* representation $\boldsymbol{v}$ for each unique identifier v, we collect the set of subtoken representations $\mathcal{H}_v$ of v, and perform an element-wise mean over $\mathcal{H}_v$ to get a fixed-length representation: $\boldsymbol{v} = \text{MeanPool}(\mathcal{H}_v)$.

4.2.2 Structural Code Encoder. The lexical encoder only captures sequential information in code tokens. To also learn from the rich structural information available in the decompiler AST, DIRE employs a gated-graph neural network (GGNN) structural encoder over the AST (Section 3.3.2). This requires a mechanism to compute initial node states, as well as design choices of which AST edges to consider in the node encoding:

Initial Node States. The initial state of a node v_i, $\boldsymbol{h}^{g}_{i,t=0}$ is computed from three separate embedding vectors, each capturing different types of information of v_i: 1) An embedding of the node's syntactic type (e.g., the root node in the AST in Fig. 4 has the syntactic type `block`). 2) For a node that represents data (e.g., variables, constants) or an operation on data (e.g., mathematical operators, type casts, function calls), an embedding of its data type, computed by averaging the embeddings of its subtokenized type. For instance, the variable node `VAR1` in Fig. 4 has the data type `char *`; its embedding is computed by averaging the embeddings of the type subtokens `char` and `*`. 3) For named nodes, an embedding of the node's name (e.g., the root node in Fig. 4 has a name `mystrcopy`), computed by averaging the embeddings of its content subtokens. The initial state $\boldsymbol{h}^{g}_{v,t=0}$ is then derived from a linear projection of the concatenation of the three separate embedding vectors. For nodes without a data type or name, we use a zero-valued vector as the respective embedding.

Graph Edges. Our structural encoder uses different types of edges to capture different types of information in the AST. Besides the simple *parent-child* edges (solid arrows in the AST in Fig. 4) in the original AST, we also augment it with additional edges [6]:

- We add an edge from the root `block` node containing the function name to each identifier node. The function name can inform names of identifiers in its body. In our running example the two arguments `VAR1` and `VAR2` defined in the `mystrcopy` function might indicate the source and destination of the copy. This type of link ("Function name to args." in Fig. 4) captures these naming dependencies.

- To capture the dependency between neighboring code, we add an edge from each terminal node to its lexical successor ("Successor terminal").
- To propagate information among all mentions of an identifier, we add a virtual "supernode" (rectangular node labeled `VAR1`) for each unique identifier v_i, and edges from mentions of v_i to the supernode ("Super node link") [19].
- Finally, we add a reverse edge for all edge types defined above, modeling bidirectional information flow.

Representations. For the *element* representation, we use the final state of the GGNN for node n_i, $\boldsymbol{h}^g_{i,T}$, as its representation: $\boldsymbol{n}_i = \boldsymbol{h}^g_{i,T}$ (the recurrent process unrolls T times; $T = 8$ for all our experiments). For the *identifier* representation for each unique identifier v_i, its representation $\boldsymbol{v}_i$ is defined as the final state of its supernode as the encoding of v_i. Since the supernode has bidirectional connections to all the mentions of v_i, its state is computed using the states of all its mentions. Therefore, $\boldsymbol{v}_i$ captures information about the usage of v_i in different occurrences.

4.2.3 Combining Outputs of Lexical and Structural Encoders. The lexical and the structural encoders output a set of representations for each identifier and code element. In the final phase of encoding, we combine the two sets of outputs. Code element representations are computed by unioning the lexical set (of code tokens) and structural set (of AST nodes) of element representations as the final encoding of each input code element; identifier representations are computed by merging the lexical and structural representations of each identifier v using a linear transformation as its representation.

4.3 The Decoder Network

The decoder network predicts names for identifiers using the representations given by the encoder. As shown in Fig. 4, the decoder predicts names based on both the representations of identifiers, and contextual information in the encodings of code elements. Specifically, as with the encoder, we assume an identifier name is composed of a sequence of sub-tokens (e.g., `destAddr` ↦ `dest`, `Addr`; see Section 4.2.1). The decoder factorizes the task of predicting idiomatic names to a sequence of time-indexed decisions, where at each time step, it predicts a sub-token in the idiomatic name of an identifier. For instance, the idiomatic name for `VAR1`, `destAddr`, is predicted in three time steps (s_1 through s_3) using sub-tokens `dest`, `Addr`, and `</i>`, (the special token `</i>` denoting the end of the token prediction process). Once a full identifier name is generated, the decoder continues to predict other names following a pre-order traversal of the AST. As we will elaborate in Section 5, not all identifiers in the decompiled code will be labeled with corresponding "ground-truth" idiomatic names, since the decompiler often generates variables not present in the original code. DIRE therefore allows an identifier's decompiler-assigned name to be preserved by predicting a special `</identity>` token.

The probability of generating a name is therefore factorized as the product of probabilities of each local decision while generating a sub-token y_t:

$$p(Y|X) = \prod_{t=1}^{T} p(y_t|y_{<t}, X),$$

where X denotes the input code, and Y is the full sequence of sub-tokens for all identifiers, and $y_{<t}$ denotes the sequence of sub-tokens before time step t.

We model $p(y_t|y_{<t}, X)$ using an LSTM decoder, following the parameterization proposed by Luong et al. [38]. Specifically, to predict each sub-token y_t, at each time step t, the decoder LSTM maintains an internal state $\boldsymbol{s}_t$ defined by

$$\boldsymbol{s}_t = f_{\text{LSTM}}([\boldsymbol{y}_{t-1} : \boldsymbol{v}_t : \boldsymbol{c}_t], \boldsymbol{s}_{t-1}),$$

where [:] denotes vector concatenation. The input to the decoder consists of two representations: the embedding vector of the previously predicted name, $\boldsymbol{y}_{t-1}$; and the encoder's representation of the current identifier to be predicted, $\boldsymbol{v}_t$. Our decoder also uses *attention* [12] to compute a context vector $\boldsymbol{c}_t$, generated by aggregating contextual information from representations of relevant code elements. $\boldsymbol{c}_t$ is computed by taking the weighted average over encodings of AST nodes and surface code tokens, for each current sub-tokenized name y_t. The decoder's hidden state is then updated using the context vector, incorporating the contextual information into the decoder's state $\tilde{\boldsymbol{s}}_t = \boldsymbol{W} \cdot [\boldsymbol{s}_t : \boldsymbol{c}_t]$, where $\boldsymbol{W}$ is a weight matrix. Then, the probability of generating a sub-token (y_t) is:

$$p(y_t|\cdot) = \frac{\exp\left(\boldsymbol{y}_t^\top \tilde{\boldsymbol{s}}_t\right)}{\sum_{y'} \exp\left(\boldsymbol{y'}^\top \tilde{\boldsymbol{s}}_t\right)}$$

4.4 Training the Neural Network

Since DIRE is constructed from neural networks, training data is required to learn the weights for each neural component. Our training corpus is a set $\mathcal{D} = \{\langle X_i, Y_i \rangle\}$, consisting of pairs of code X and sub-token sequences Y, denoting the decoder-predicted sequence of identifier names. DIRE is optimized by maximizing the log-likelihood of predicting the gold sub-token sequence Y_i for each training example X_i:

$$\sum_{\langle X_i, Y_i \rangle} \log p(Y_i|X_i) = \sum_{\langle X_i, Y_i \rangle} \sum_{t=1}^{|Y_i|} w_t \cdot \log p(y_{i,t}|X_i),$$

where $Y_{i,t}$ denotes the t-th sub-token in the decoder's prediction sequence Y_i. As discussed in Section 4.3, there are intermediate variables in the decompiled code. To ensure the decoder network will not be biased towards predicting `</identity>` for other identifiers, we use a tuning weight w_i and set it to 0.1 for sub-tokens that correspond to intermediate variables (and 1.0 otherwise).

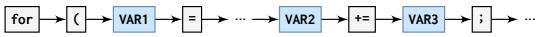
5 DATA PREPARATION

In this section, we address the challenges surrounding the data: training DIRE requires a large corpus of structure-rich, annotated data. We describe our technique to automatically generate such data from C code at scale. With the ability to automatically generate training data this way, we must choose appropriate projects from the large number of C-language repositories on social coding platforms.

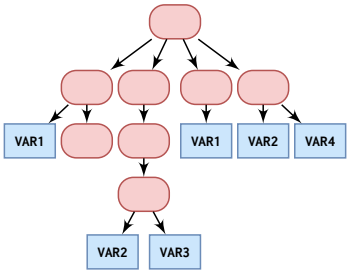
5.1 Generating a Parallel Corpus

The scale of the structure-rich, annotated corpus needed to train DIRE practically necessitates a technique to generate this corpus automatically. Fortunately, it is possible to do this starting from a large repository of existing C code.

At a high level, each entry in our corpus corresponds to a source code function, and consists of the information necessary to train our model. An entry in the training corpus is illustrated in Fig. 5. Each entry contains three elements: (a) the tokenized code, with variables replaced by an ID that uniquely identifies the variable in the function; (b) the decompiler's AST (Section 3.1) modified to contain the same unique variable IDs; and (c) a lookup table mapping variable IDs to both the decompiler- and developer-assigned names. It is important to assign a unique name to each variable to disambiguate any shadowed variable definitions. The tokenized code and AST representations are used in both the model's input and output. The input representation uses the decompiler-assigned names, while the output uses the developer-assigned names.



(a) Tokenized decompiled code with variable placeholders.



(b) AST with placeholders.

ID	Decompiler	Developer
1	v1	ans
2	v2	size
3	i	i
4	ptr	head

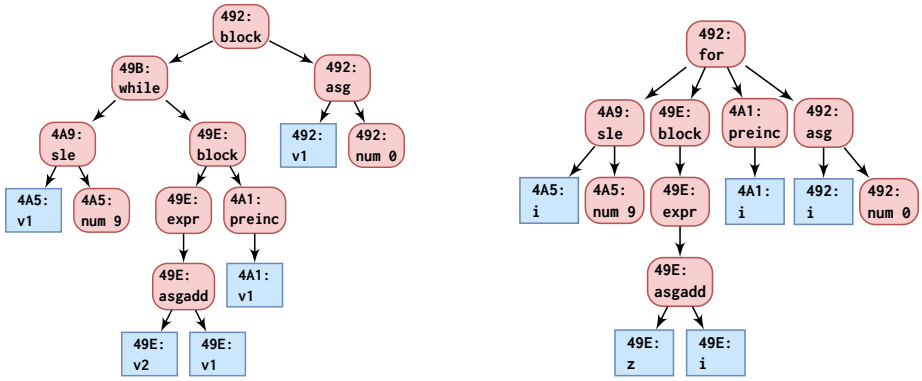
(c) Variable lookup table.

Fig. 5. Entry in the training corpus. Each corresponds to a function and contains (a) tokenized code (b) the AST, both with variables replaced with unique IDs, and (c) a lookup table containing decompiler- and developer-assigned names.

Generating the placeholders and decompiler-chosen names is relatively straightforward. First, a binary is compiled normally and passed to the decompiler. Next, for each function, we traverse its AST and replace each variable reference with a unique placeholder token. Finally, we instruct the decompiler to generate decompiled C code from the modified AST, tokenizing the output. Thus, we have tokenized code, an AST, and a table mapping variable IDs to decompiler-chosen names. The remaining step, mapping developer-chosen names to variable IDs, is the core challenge in automatic corpus generation. Following our previous approach [26], we leverage the decompiler's ability to incorporate developer-chosen identifier names into decompiled code when DWARF debugging symbols [16] are present in the binary. However, this alone is not sufficient to identify which developer-chosen name maps to a particular variable ID generated in the first step.

Specifically, challenges arise because decompilers use debugging information to enrich the decompiler output in a variety of ways, such as improving type information. Recall from Section 3 that decompilers often make choices between semantically-identical structures: the addition of debugging information can change which structure is used. Unfortunately, this means that the difference between code generated with and without debugging symbols is not always an α-renaming. In practice, the format and structure of the code can greatly differ between the two cases. An example is illustrated in Fig. 6. In this example, the first pass of the decompiler is run without debugging information, and the decompiler generates an AST for a `while` loop with two automatically-generated variables named `v1` and `v2`. Next, the decompiler is passed DWARF debugging symbols and run a second time, generating the AST on the right. While the decompiler is able to use the developer-selected variable names `i` and `z`, it generates a very different AST corresponding to a `for` loop.

An additional challenge is that there is not always a complete mapping between the variables in code generated with and without debugging information. Decompilers often generate more variables than were used in the original code. For example, `return (x + 5);` is commonly decompiled to `int v1; v1 = x + 5; return v1;`. The decompiled code introduces a temporary variable `v1` that does not correspond to any variable in the original source code. In this case, there is no developer-assigned name for `v1`, since it does not exist in the original code. The use of debugging information can change how many of these additional variables are generated. One solution to these problems proposed by prior work is to post-process the decompiler output using heuristics to *align* decompiler-assigned and developer-assigned names [26].



(a) AST without DWARF. (b) AST with DWARF.

Fig. 6. Decompiler ASTs for the code in Fig. 2. Hexadecimal numbers indicate the location of the disassembled instruction used to generate the node. While the ASTs are different, operations on variables and their offsets are the same, enabling mapping between variables (i.e., **v1**↦**i** and **v2**↦**z**).

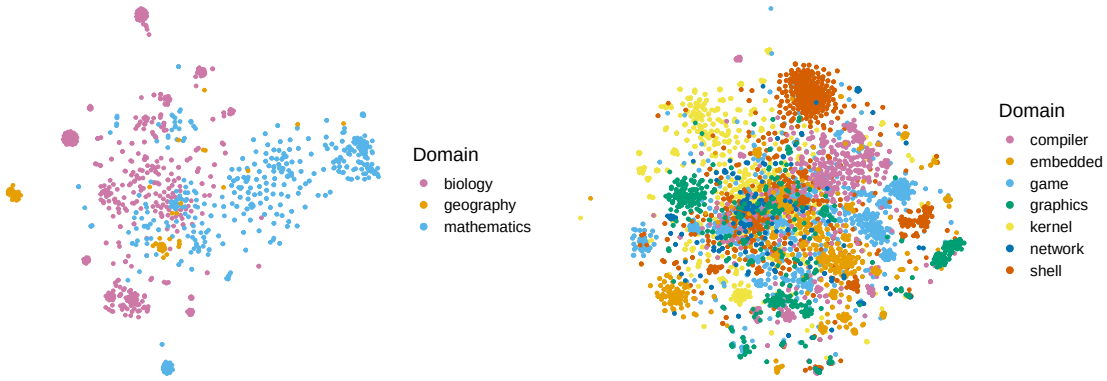
However, this technique can only correctly align 72.8% of variable names, therefore limiting the overall accuracy of any subsequent model trained on this data. Instead, we developed a technique that directly integrates with the decompiler to generate an accurate alignment *without using heuristics*. Our key insight is that while the AST and code generated by the decompiler may change when debugging information is used, *instruction offsets and operations on variables do not change*. As a result, each variable can be uniquely identified by the set of instruction offsets that access that variable. For example, in Fig. 6, although there is not an obvious mapping between the nodes in the trees, the addresses of the variable nodes in the trees have not changed. This enables us to uniquely identify each variable by creating a signature consisting of the set of all offsets where it occurs. The variables **v1** and **i** have the signature `{492,49E,4A1,4A5}`, while **v2** and **z** have the signature `{49E}`. Note that some uses of variables overlap, e.g., **v1** (**i**) is summed with **v2** (**z**) in the instruction at offset `49E`. This necessitates collecting the full set of variable uses to disambiguate these instances.[1]

In summary, to generate our corpus we: 1) Decompile binaries containing debugging information. 2) Collect signatures and corresponding developer-assigned names for each variable in each function. 3) Strip debugging information and decompile the stripped binaries. 4) Identify variables by their signature, and rename them in the AST, encoding both the decompiler- and developer-assigned names. 5) Generate decompiled code from the updated AST. 6) Post-process the updated AST and generated code to create a corpus entry. The final output is a per-binary file containing each function's AST and decompiled code with corresponding variable renamings.

5.2 Selecting Data

As with any machine learning model, the characteristics of the data used to train DIRE can have a large impact on its performance. We consider multiple different classes of data and how each affects DIRE. We sort data into classes by mining metadata from GitHub and selecting repositories according to some signal for the desired characteristic. In particular, we look at two ways of partitioning GitHub repositories into classes:

[1]While it is possible for two variable signatures to be identical, we found these collisions to occur very rarely in practice. In these cases we do not attempt to assign names to variables.



(a) Debian Data; Classifier Accuracy: 96.53%

(b) GitHub Data; Classifier Accuracy: 83.54%

Fig. 7. Dimensionality-reduced visualizations of the embeddings generated by our model. Note how the different domains tend to form separable clusters.

- by GitHub stars, an indicator of popularity.
- by use case that the software is intended for (i.e., the software's domain).

5.2.1 Stars. We use GitHub stars as a signal for level of popularity. On the GitHub platform, users can "star" a repository that they appreciate, find useful, or find interesting (similar to a "like" on social media platforms). The star count is thus a crowd-sourced measure for popularity, but it also tends to correlate with improved maintenance activities and other software engineering best practices [9, 52].

We partition code into classes based on stars because of what popularity and maintenance might mean for identifiers. Source code which is popular or well-maintained is more frequently read and understood, sometimes for the purpose of correcting errors or otherwise making modifications. Descriptive identifier names are critical to these processes [47], so out of necessity, the identifiers from highly-starred projects are likely to be useful to developers. In contrast, identifiers from poorly maintained projects with few stars may not be useful. Note that for our purposes, the project need not be actively maintained nor currently popular; rather, it needs only to have been actively maintained at some point, so that it is more likely that the variables names have been subjected to scrutiny.

In our evaluation, we use two star-related classes of software on either end of the star count spectrum. We created the *high-star* dataset by sampling repositories sorted by the number of stars in descending order. The minimum number of stars of any selected high-star repository was 50 stars, the maximum, 76301 (from `torvalds/linux`), the median, 155, and the mean, 589.2.

The *no-star* dataset consists of all repositories which have no stars at all. We sample repositories randomly with uniform probability when we select from this dataset.

We ascertain star information from the GHTorrent database [20] which contains snapshots of metadata for all publicly-available GitHub projects at certain intervals; we used the version from June, 2019.

5.2.2 Software Domain. DIRE works best when predicting general, common variable names like `len`, `value`, and `buf` as discussed in Section 6. However, DIRE can struggle on identifiers that are more rare and unique to a particular

software domain. For example, an identifier named `lexeme` likely occurs most frequently in compilers, and less often in unrelated domains like graphics. This motivates the construction of a corpus of domain classes.

We can mine software domains from repository metadata, though not all repositories have the requisite metadata, nor do binaries found in the wild without context. We built a simple classifier to assign domain class to binaries for which we have no explicit domain classification and to check our expectation that binaries contain domain information. The classifier is an amalgamation of two models in sequence: `doc2vec` [33], which converts a piece of text of arbitrary size into an embedding (with 64 dimensions in our case), and a support vector machine (SVM) with a radial basis function (RBF) kernel.

To develop our classifier, we iterated on a smaller and easier to use set of data drawn from the Debian ALLSTAR dataset [51] with domain class assignments drawn from the Ultimate Debian Database [41]. In this dataset, domain classes are scientific and technical disciplines (e.g., biology and geography). To localize which parts of the code contained significant amounts information, we experimented with removing different code features. We found that the collective string literals from a given binary are often sufficient to correctly identify that binary's domain using our classifier.

We applied the classifier to our GitHub data, excluding some very small object files containing only a few functions functions and no string literals (artifacts of the automated compilation process). Here, we use the repository's topic labels to assign repositories to domains. Topic labels are tags voluntarily assigned by project maintainers to give viewers of the project a rough idea of the project's characteristics at a glance. Not all repository maintainers elect to supply topic labels, and not all topic labels are software domains (perhaps unsurprisingly, `c` is the most common topic label in our dataset). For the purposes of demonstration, we hand-selected seven of the most common topic labels that corresponded to software domains: `kernel`, `game`, `shell`, `compiler`, `network`, `embedded`, `graphics`.

Our domain classifier is able to correctly classify binaries 96.53% of the time on the Debian dataset and 83.54% of the time on the GitHub dataset. Fig. 7 shows a plot of embeddings generated by our model after we dimensionality-reduced each via the t-distributed stochastic neighbor embedding (t-SNE) method into two dimensions for viewing. While it is impossible to perfectly represent a 64 dimensional space in just two dimensions, it is still possible to see how different domains form distinct, separable clusters.

Embeddings from a structural model may perform better on the source code itself, but we leave this to future work.

5.2.3 Data Cleaning and Deduplication. We performed several types of data cleaning on our large GitHub stars corpus. We excluded all binaries written in languages other than C (our decompiler's target language); it is common to find C++ and other languages in majority-C repositories. We also removed automatically generated code, the repetitive structure and machine-generated names of which undermines the naturalness hypothesis. We used a simple heuristic to exclude these repositories: any repository for which any of the strings `"generated code"`, `"autogenerated"`, `"auto-generated"`, `"automatically generated"`, or `"automatically-generated"` appeared was excluded. This is a rather harsh criterion, but due to the abundance of data available on GitHub, we opted to potentially exclude some acceptable training data to prevent unnatural code from infiltrating our corpus.

We also performed deduplication on our corpus. Duplicate projects abound on GitHub, and can be detrimental to the model's ability to generalize [2]. We used a multi-layer deduplication strategy. The first layer applies the `gh-dedup` dataset [50], which lists duplicate repositories along with their parent repositories, excluding any duplicate repositories whose parents were also in the data set. While this dataset is able to identify some duplicates, it does not identify all of them. We hashed each binary, and ensured only one copy of each binary occurred in our dataset. Unfortunately, this

```
Input: List of clusters for each repository clusters.
Input: Merge threshold threshold.
Output: Merged list of clusters.
numSinceLastMerge ← 0
while numSinceLastMerge < length(clusters) do
  current ← remove and return head of clusters
  mergeOccured ← false
  for c in clusters do
    if size(intersection(c, current)) ≥ threshold then
      remove c from clusters
      current ← merge(c, current)
      mergeOccured ← true
    end if
  end for
  if mergeOccured then
    numSinceLastMerge ← 0
  else
    numSinceLastMerge++
  end if
  append current to clusters
end while
return clusters
```

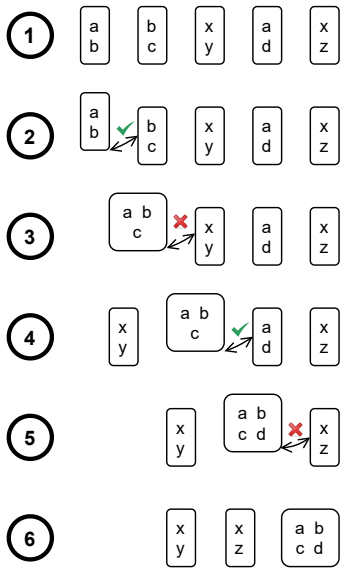


Fig. 8. Clustering algorithm for deduplication. Shown right is one pass through the while loop with a threshold of 0.5. The repositories that are merged to form each cluster are also tracked; as the members of each cluster constitute duplicates, only one member repository is sampled from each cluster when selecting data for a given model.

approach is very sensitive to any changes in the binary. It does not detect common situations which result in large numbers of duplicate functions, such as when slightly different versions of a project appear in our dataset.

To help address these concerns, we performed a clustering-based deduplication step. See Fig. 8 for an illustration of and pseudocode for our clustering algorithm. We initialize the algorithm with one cluster for each repository; the elements of each cluster are the hashes for each binary from that repository. The goal of performing clustering is to form groups of repositories that differ slightly, perhaps by a few versions; that is, most binaries in the project are the same and only a few differ. Clusters are stored sequentially in an arbitrary order. We attempt to merge the first cluster with each other in the sequence; when a merge occurs, we continue attempting to merge until we have attempted (successfully or not) to merge against all other clusters in the sequence. This cluster is then placed at the end of the sequence. A merge occurs when the size of the intersection between two clusters exceeds some threshold percentage of either of the two clusters involved. We do this repeatedly until no more merges can occur. The result is a set of repository clusters which represent groups of near-identical repositories.

Repositories can share binaries if they are duplicates, but also if they use the same libraries. Thus, any choice of threshold incurs a trade-off. Too low a threshold produces clusters or repositories that happen to use the same library. Too high a threshold clusters fewer true duplicates.

To choose the threshold, we manually evaluated a sample of the dataset. We consider two projects duplicates if they meet one of the following criteria:

- the names of the repositories are the same.

Table 1. Evaluation of DIRE. Values are percentages, higher accuracy and lower character error rate (CER) are better.

	DIRE		Lexical Enc.		Structural Enc.	
	Acc.	CER	Acc.	CER	Acc.	CER
Overall	**74.3**	**28.3**	72.9	28.5	64.6	37.5
Body in Train	**85.5**	**16.1**	84.3	16.3	75.7	25.5
Body not in Train	**35.3**	**67.2**	33.5	67.7	26.3	76.1

- the project's README indicates the projects are the same or based on the same source.
- the intersection of the sets of folder names make up at least 90% of the folder names of at least one of the two repositories under consideration. This method is especially useful for identifying duplicates that are missing READMEs or projects for which one is heavily based on another but without acknowledgement in the README.

We sampled ten clusters repositories from each threshold level (every 5%) and chose two repositories from those clusters evaluating how many were duplicates using the above criteria. We chose the lowest threshold level for which 90% or more or these were duplicates on each of the high-star and no-star partitions of our corpus. This method resulted in a threshold of 50%.

When we sample from these data sets, we sample one repository from each cluster.

6 EFFECTIVENESS

In this section, we establish the efficacy of DIRE as a technique and compare it to prior work. These results are reproduced from the original presentation of DIRE [31].

- RQ1: How effective is DIRE at assigning names to variables in decompiled code?
- RQ2: How does each component of DIRE contribute to its efficacy?
- RQ3: Is DIRE more effective than prior approaches?

Data Preprocessing. To answer our first two research questions, we trained DIRE on 3,195,962 decompiled functions extracted from 164,632 binaries mined from GitHub. First, we automatically scraped GitHub for projects written in C. Next, we modified project build scripts to include debug information when compiling the project, and collected all successfully generated 64-bit x86 binary files. We then hashed each binary to remove any duplicates. We then passed these binaries through our automated corpus generation system. Finally, we filtered out any functions that did not have any renamed variables and, for practical reasons, any functions with more than 300 AST nodes. After filtering, 1,259,935 functions with an average AST size of 77 nodes remained. These functions were randomly split per-binary into training, development and testing sets with a ratio of 80:10:10. Splitting the sets per-binary ensures that binary-specific identifiers are not included in both the training and test sets.

Evaluation Methodology. After training, we ran DIRE to generate name suggestions on the test data. We evaluate the accuracy of these predictions, comparing the predicted variable names to names used in the original code (i.e., names contained in the debugging information) counting a successful prediction as one that exactly matches the original name. However, there can be multiple, equally acceptable names (e.g., `file_name`, `fname`, `filename`) for a given identifier. An accuracy metric based on exact match cannot detect these cases. We therefore use character error rate (CER), a metric that calculates the edit distance between the original and predicted names, then normalizes by the length of the original name [56], assigning partial credit to near misses.

```
1  void *file_mmap(int V1, int V2)
2  {
3    void *V3;
4    V3 = mmap(0, V2, 1, 2, V1, 0);
5    if (V3 == (void *) -1) {
6      perror("mmap");
7      exit(1);
8    }
9    return V3;
10 }
```

ID	DIRE	Dev.
1	`fd`	`fd`
2	`size`	`size`
3	`buf`	`ret`

Fig. 9. Decompiled function (simplified for presentation), DIRE variable names, and developer-assigned names.

Recall from Section 5.1 that there are often many more variables in the decompiled code than in the original source; these variables will not have a corresponding original name. In our corpora, the median number of variables in each function is 5, with 3 having a corresponding original name. Although DIRE generates predictions for *all* variables, we do not evaluate predictions on variables that do not have a developer assigned name. We do this because it is not necessarily incorrect for a renaming system to assign names to variables not present in the original source code. Recall the example where `return (x + 5);` is decompiled to `int v1; v1 = x + 5; return v1;`. The name `sum` is likely more informative than `v1`, and it would be unhelpful to penalize a system that suggests this renaming. However, although renaming in these cases could be helpful, we do not want to overapproximate the effectiveness of our system by claiming any renaming of these variables as correct: it is also possible to assign variables a misleading name that *decreases* the readability of code by obfuscating the purpose of a variable. For example, suggesting the name `filename` to replace `v1` in the above code would be misleading.

Neural Network Configuration. For our experiments we replicate the neural network configuration of Allamanis et al. [6]. We set the size of word embedding layers to be 128. The dimensionality of the hidden states for the recurrent neural networks used in the encoders is 128, while the hidden size for the decoder LSTM is 256. For both the sequential and structural encoders, we use two layers of recurrent computation, adding another identical recurrent network to process the decompiled code using the output hidden states of the first layer. For both DIRE and the baseline neural systems, we train each model for 60 epochs. At test time, we use beam search to predict the sequence of sub-tokenized names for each identifier (Section 4.3), with a beam size of 5.

6.1 RQ1: Overall Effectiveness

Experimental results are summarized in Table 1. The "Overall" row shows the performance of our technique on the full test set and the leftmost column shows the accuracy of DIRE. From this, we can see that DIRE can recover 74.3% of the original variable names in decompiled code, demonstrating that it is effective in assigning contextually meaningful names to identifiers in decompiled code.

Figure 9 shows an example renaming generated by DIRE. Here, DIRE generates the variable names shown in the "DIRE" column of the table. The developer-chosen names are shown in the "Dev." column. Two of three names suggested by DIRE exactly match those chosen by the developer. Though DIRE suggests `buf` instead of `ret` as the replacement for `V3`, the name is not entirely misleading: `mmap` returns a pointer to a mapped area of memory that can be written to or read from. Work has shown that large code corpora may contain near-duplicate code across training and testing sets, which can cause evaluation metrics to be artificially inflated [2]. Though our corpus contains no duplicate binaries, splitting test and training sets per-binary still results in functions appearing in both. A common cause of duplicate

Table 2. Example identifiers from the *Body not in Train* testing partition and DIRE's top-5 most frequent predictions.

`len`	`value`	`new_node`	`bytes_read`
`len` (60%)	**`value`** (28%)	**`node`** (48%)	**`size`** (38%)
`n` (6%)	**`data`** (7%)	**`child`** (31%)	**`bytes_read`** (13%)
`size` (5%)	**`val`** (3%)	**`treea`** (0.3%)	**`len`** (13%)
`length` (1%)	**`name`** (3%)	**`tree`** (0.3%)	**`cmd_code`** (13%)
`l` (1%)	**`key`** (2%)	**`root`** (0.3%)	**`read`** (13%)

functions in different binaries is the use of libraries. We argue that it is reasonable to allow such duplication since reverse-engineering binaries that link against known (e.g., open source) libraries is a realistic use case.

Nevertheless, to better understand the performance of our system, we partition the test examples into two subcategories: ***Body in Train*** and ***Body not in Train***. The *Body in Train* partition includes all functions whose entire body matches at least one function in the training set; similarly, the *Body not in Train* set includes only functions whose body does not appear in the training set. The last two rows in Table 1 show the performance on these partitions. DIRE performs well on the *Body in Train* test partition (85.5%). This indicates that DIRE is particularly accurate at name prediction when code has appeared in its training set (e.g., libraries, or code copied from another project). DIRE is still able to exactly match 35.3% of variable names in the *Body not in Train* set, indicating that it still generalizes to unseen functions.

We note that it is possible to perform better on the body-in-train portion by storing the training data in a dictionary indexed by the canonicalized decompiled code and checking the dictionary before using the model to make variable name predictions. The pseudocode for this experiment is as follows:

```
if decompiled_code is in dictionary:
    variable_names = get_names_from_dictionary(decompiled_code)
else:
    variable_names = DIRE(decompiled_code)
```

Because the test set is 77.6% body-in-train, this apporach achieves an overall accuracy of 85.5%. The cost of this improvement is storing and loading this large dictionary in addition to the model at test time. A more nuanced approach might involve curating a set of library functions that occur frequently in decompiled code for use in the dictionary instead. We leave this to future work.

Table 2 contains example identifiers from the *Body not in Train* test set, along with DIRE's most frequent predictions. We observe that inexact suggested names are often semantically similar to the original names. DIRE also performs best on simple identifiers such as **`len`** and **`value`**. This is because it is difficult to predict the exact name for long, complex identifiers with compositional names. Each subtoken in the variable name is predicted with a probability of correctness less than one. Thus, probability of predicting an entirely correct name generally decreases with length. In addition, any given long, complex identifier will likely occur relatively infrequently in the dataset. This means the model has less chance to adjust to the context around these variable names. However, DIRE is still often able to suggest semantically relevant identifiers (e.g., **`node`**, **`child`**).

RQ1 Answer: We find that on average DIRE is able to suggest variable names identical to those chosen by the original developers 74.3% of the time, in a best case scenario that does not explicitly deduplicate functions between the training and test sets.

```
1  file *f_open(char **V1, char *V2, int V3) {
2    int fd;
3    if (!V3)
4      return fopen(*V1, V2);
5    if (*V2 != 119)
6      assert_fail("fopen");
7    fd = open(*V1, 577, 384);
8    if (fd >= 0)
9      return fdopen(fd, V2);
10   else
11     return 0;
12 }
```

ID	Lexical	Structural	DIRE	Developer
1	`file`	`fname`	`filename`	`filename`
2	`name`	`oname`	`mode`	`mode`
3	`mode`	`flags`	`create`	`is_private`

Fig. 10. Decompiled function (simplified for presentation), suggested names, and developer-assigned names. The lexical and structural models are unable to correctly predict the name `filename` for variable 1, but DIRE can by combining them.

6.2 RQ2: Component Contributions

Table 1 also shows the results for models using only our lexical or structural encoders. We find that the lexical encoder is able to correctly predict 72.9% of the original variable names, while a model using the structural encoder is able to correctly predict 64.6% of the original variable names. These simpler models still perform well, but by combining them in DIRE we are able to achieve even better performance.

Figure 10 illustrates how DIRE can effectively combine these models to improve suggestions. Here, the placeholders `V1`, `V2`, and `V3` are variables which should be assigned names. The "Lexical", "Structural", and "DIRE" columns show the predictions from each model, and the "Developer" column shows the name originally assigned by the developer. In this example, the lexical and the structural models are unable to predict any of the original variable names, while DIRE is able to correctly predict two of the three names. This example also shows the contributions from each of the submodels. For example, for `V1`, the lexical model predicts `file` while the structural model predicts `fname`. Combining the predicted subtokens generates `filename`, the same name chosen by the developer. For `V2`, the lexical and structural models both fail to predict `mode`, but note that the lexical model *does* predict `mode` for `V3`. By combining the models, DIRE instead correctly predicts `mode` for `V2`.

> RQ2 Answer: Each component of DIRE contributes uniquely to its overall accuracy.

6.3 RQ3: Comparison to Prior Work

To answer RQ3, we compare to our prior work [26] and to Debin [22], the state-of-the-art technique for predicting debug information directly from binaries. In our earlier work, which used a purely-lexical model based on statistical machine translation (SMT), we were able to exactly recover 12.7% of the original variable names chosen by developers. In contrast, DIRE is able to suggest identical variable names 74.3% of the time. We attribute this improvement to two factors: 1) the improved accuracy of our corpus generation technique, and 2) the use of a model that incorporates both lexical and structural information. To better understand the performance of DIRE, we also compare to Debin, a different approach to generating more understandable decompiler output. Debin uses CRFs to learn models of binaries and directy generate DWARF debugging information for a binary, which can be used by a decompiler such as Hex-Rays. The debugging information generated by Debin contains predicted identifiers, types, and names. To choose a variable name, Debin proceeds in two stages: it predicts which memory locations correspond to function-local arguments and variables, and then predicts names for the variables it identified. In contrast, DIRE leverages the decompiler to

```
long gray(unsigned a1,
          int a2) {
  unsigned v3, v4;
  int v5;
  if (a2 >= 0)
    return a1 ^ (a1 >> 1);
  v5 = 1;
  v4 = a1;
  while (1) {
    v3 = v4 >> v5;
    v4 ^= v4 >> v5;
    if (v3 <= 1 ||
        v5 == 16)
      break;
    v5 *= 2;
  }
  return v4;
}
```

(a) Hex-Rays.

```
void gray() {
  unsigned v0;
  int v1;
  unsigned i, v3;
  int x;
  if (v1 < 0) {
    x = 1;
    v3 = v0;
    while (1) {
      i = v3 >> x;
      v3 ^= v3 >> x;
      if (i <= 1 ||
          x == 16)
        break;
      x *= 2;
    }
  }
}
```

(b) Hex-Rays w/ Debin.

Fig. 11. Effects of incorrect debugging information on decompiler output. The `gray` function computes the Gray code of `a1` in `a2` bytes [42]. On the left, (a) is the output of Hex-Rays without debugging symbols; it is able to correctly identify the arguments and return type. On the right, (b) is the output with incorrect DWARF information generated by Debin: note missing arguments, `return` statements, and incorrect type.

identify function offsets and local variables. Building on top of the decompiler helps DIRE maintain the quality of pseudocode output. An example is shown in Fig. 11, which contains a C function for converting between a number `a1` and its Gray code representation in `a2` bits [42]. Figure 11 (a) shows the output of Hex-Rays when passed a binary with no debug information. Although these variables do not have meaningful names, it is clear that `gray` is a function that takes two arguments and returns a `long`. Figure 11 (b) information was generated using Debin's bundled model.[2] We observe that Debin does not accurately recover variable names in this case, perhaps since its model was trained on a different set of code. However, this example also surfaces a fundamental limitation of the Debin approach: both the inferred structure and the types of the variables in the program have changed. This occurs because Hex-Rays prioritizes debugging information over its own analyses and heuristics. In this case, the debugging information generated by Debin does not indicate a return value of the `gray` function nor any arguments, misleading the decompiler. By starting at the point shown in Fig. 11 DIRE maintains structure and typing even in the presence of incorrect predictions. To evaluate our performance compared to Debin, we trained it on binaries in our dataset. Due to time restrictions, we found it impractical to train Debin on the full dataset. For a fair comparison, we instead subsampled our training set at 1% and 3% and trained both Debin and DIRE on these sets.[3] After training, we ran Debin on binaries in our test set, extracted names using our corpus generation pipeline, and measured the accuracy of predictions. Our results are shown in Table 3. We find that DIRE is able to outperform Debin at all sampling sizes. When trained on 1% of the corpus DIRE is able to exactly recover 32.2% of all identifiers, while Debin recovers 2.4%. On the 3% partition, DIRE is able to recover 38.4% of names, while Debin is able to recover 3.9%. The lower performance of Debin we observed could be attributed to compound error: in addition to variable names themselves, Debin must predict what memory locations correspond to variables. If a memory location is not predicted to be a variable, Debin cannot assign it a name. We also note that

[2]https://files.sri.inf.ethz.ch/debin_models.tar.gz, accessed April 10, 2019
[3]The 3% subsampling we used is a slightly larger training set than the 3,000 binaries used to train Debin in the original paper [22].

Table 3. Comparison of DIRE and Debin trained on 1% and 3% of our full corpus of 164,632 binaries. All accuracy values are percentages, higher accuracy is better.

	1% of corpus		3% of corpus	
	DIRE	Debin	DIRE	Debin
Training Time (h)	1.8	13.3	6.1	17.2
Accuracy – Overall	32.2	2.4	38.4	3.9
Accuracy – Body in Train	40.0	3.0	47.2	4.8
Accuracy – Body not in Train	5.3	0.6	8.6	0.7

we were able to train DIRE much faster than Debin, although DIRE is GPU-accelerated, while Debin as distributed is limited to execution on the CPU.

RQ3 Answer: DIRE is a more accurate and more scalable technique for variable name selection than other state-of-the-art approaches.

7 THE EFFECT OF DATA

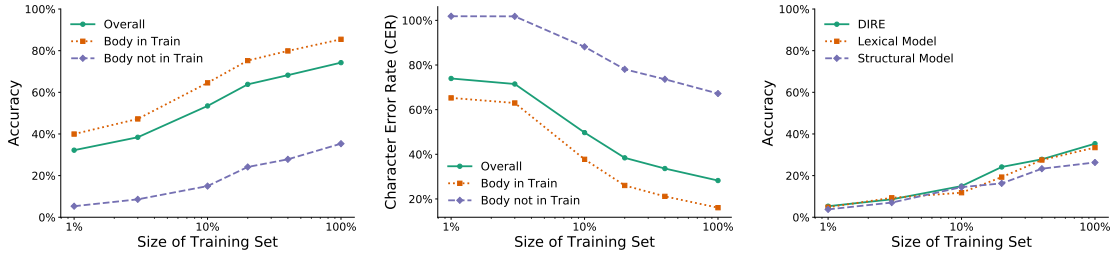
Any machine learning system's performance is related to both the model's structure and the data used to train it. Here, we investigate the impact that data has on DIRE. Training a model is computationally expensive, and it may not always be feasible to train DIRE on a large data set. This motivates investigating the performance of DIRE on data quantity. Data can also encode biases, which in turn become encoded in machine learning models [39]. This motivates our investigation of the impact of data provenance on DIRE. We show that models of source code tend to strengthen biases towards common, generic variable names. This motivates us to modify DIRE to better predict uncommon domain-specific variable names which nonetheless provide important context.

We ask the following research questions. Only RQ4 is reproduced from the original presentation of DIRE [31].

- RQ4: How does the quantity of data influence the efficacy of DIRE?
- RQ5: How does the provenance of training data influence the efficacy of DIRE?
- RQ6: How well can DIRE perform on diverse code from various domains if it is made aware of its software domain?

7.1 RQ4: Effect of Data Quantity

To answer RQ4, we varied the size of the training data and measured the change in performance of our models. Training data was subsampled at rates of 1%, 3%, 10%, 20%, and 40% (equivalent to 31,960, 95,879, 319,596, 639,192, and 1,278,385 functions, respectively). The results of these experiments are shown in Fig. 12. Figures 12a and 12b show the change in accuracy and CER of DIRE respectively. The size of the training data is plotted on the x-axes, while accuracy and CER are plotted on the y-axes. While DIRE has low accuracy on the *Body not in Train* set at the lowest sampling rates, at a 1% sampling rate it is still able to correctly select names over 40% of the time for the *Body in Train* test set, suggesting that it is possible to use much less data to train a model if the target application is reverse engineering of libraries rather than binaries in general. Note however that the CER of DIRE is still high at low sampling rates. This implies that in the cases where DIRE selects an incorrect variable name the chosen name is quite different from the correct name.



(a) Accuracy of DIRE (higher is better).

(b) CER of DIRE (lower is better).

(c) Accuracy of each neural model on the *Body not in Train* partition.

Fig. 12. The impact of training corpus size on the performance of DIRE. Figures (a) and (b) show how increasing the amount of training data improves the performance of DIRE; (c) shows the performance of each of the submodel as training size changes.

Table 4. Examples of the variable name quality characterizations on variable names drawn from our corpus. For all characterizations, higher is better.

		Fluency		Recognizability	
Name	Length	Tokenization	Value	English Words	Value
`nofile_limit`	12	`no file _ limit`	5	`no file _ limit`	1.000
`phmmap_map`	10	`ph mmap _ map`	4	`hmm a _ map`	0.800
`wpid`	4	`w pid`	3	`id`	0.500
`gstring`	7	`gstring`	7	`string`	0.857
`tdbRef`	6	`tdb Ref`	3	`db Ref`	0.833
x	1	x	1		0.000

Sampling at a higher rate dramatically decreases the CER, allowing for namings that are closer the developers' choices. At a sampling rate of 40%, DIRE comes quite close to the performance of the model trained on the full training set, with an overall accuracy of 68.2% (vs. 74.2%) and a CER of 33.6% (vs. 28.2%). Figure 12c shows the effect of training set size on the performance of DIRE and its component neural models on the *Body not in Train* test set. Note how at sampling rates at or below 10% the models have similar performance. In cases where there is little training data, training time can be further reduced by using only one of the two submodels.

RQ4 Answer: DIRE is data-efficient, performing competitively using only 40% of the training data. DIRE is also robust, outperforming the lexical and structural models in most sub-sampling cases.

7.2 RQ5: Effect of Data Provenance

Biased selection of training data may impact the performance of the model trained on that data. Given that unpopular projects far outnumber popular ones, a random sample of data from GitHub is likely to contain mostly unpopular projects. However, as popular projects are better maintained and subjected more scrutiny, variable names from popular projects may be systematically different from unpopular projects' variable names.

We analyze the impact of project popularity on DIRE's predictions.

7.2.1 Methodology. We train two versions of the DIRE model, one on our high-star corpus and another on our low-star corpus.

We prepare the datasets for training and testing each model by sampling roughly the same number of functions from each of the high and no-star partitionsportion of the corpus (2,739,098 and 2,737,104, respectively), following the same data preparation steps described in Section 6. Thus, we have two training sets, two development sets, and two test sets, one for the high-star portion and one for the no-star portion, respectively. We train two models: one on the high-star training set, and another on the no-star training set. We evaluate each model on each test set, resulting in four evaluations: the model trained on the high-star training set (henceforth the high-star model) on the high-star test set, the high-star model on the no-star test set, the no-star model on the high-star test set, and the no-star model on the no-star test set.

As in RQ4, we report accuracy and character error rate on predictions. These metrics have some weaknesses: accuracy is sensitive to semantic-perserving differences (for example, predicting `length` instead of `len`) and character-error rate will grade misses of different lengths differently. We use the VarCLR technique [11] as a third metric. VarCLR uses contrastive learning to put similar variable names, that is, those that can be substituted for one another, close to each other in an embedding space, while pushing names that are opposite farther away. A VarCLR score is generated by computing the cosine similarity between embeddings for two variable names. A score around 50% indicates a lack of relationship between the two variables, higher indicates increasing degrees of similarity, and lower indicates increasing anti-similarity: the names have opposite meanings.

VarCLR scores provide a notion of similarity, but the score is symmetric and thus does not indicate which variable name is more descriptive, if either. Thus, we present three additional ways to characterize the quality of DIRE's predictions:

- Length: more information can be contained in a longer message, meaning the variable is potentially more useful.
- Fluency: Given a byte-pair-encoding (BPE) as discussed in Section 4.2.1 based on the *developer* variable names in the corpus, the fluency score is the length of the longest subword when the name is broken up into subwords. This metric rewards longer names and those that are common among developers. A higher score indicates that the model's predictions are more "fluent" in the developers' vernacular. Because the high-star and no-star corpora do not have identical vocabularies, we report the average fluency value for each prediction.
- Recognizability: The fraction of the variable name that consists of words from an English dictionary (GNU `aspell`), along with a short list of 127 common abbreviations in computing (such as `str` for `string` and `lib` for `library`). With the exception of `a` and `i`, we remove single letters from the dictionary. Single letter "words" communicate little information in the context of identifiers; rather, acronyms and full words are most useful to developers [32].

Examples of all three variable name quality characterizations are shown in Table 4. We expect that high-star code will score higher on these metrics.

In addition, we reason that better maintained code may be more likely to contain more descriptive variables customized for specific use cases; that is, we expect high-star code and thus the predictions of models trained on it to be more diverse. To quantify variable diversity, we calculate Shannon's Entropy on the variable name frequency distribution. Entropy is measured in bits and quantifies how skewed a distribution is towards common names. Entropy captures both the number of variable names that occur and the frequency at which each variable name occurs. Low entropy indicates low variable diversity. At the lowest possible entropy, zero bits, one variable name has probability

Table 5. DIRE's performance when trained and tested on each permutation of high or low star training data and tested on high or low star test data. Values are percentages. Higher accuracy, lower character error rate (CER), and Higher VarCLR are better.

Training Data	Test Set	Overall			Body in Train			Body not in Train		
		Accuracy	CER	VarCLR	Accuracy	CER	VarCLR	Accuracy	CER	VarCLR
High Stars	High Stars	62.8	40.0	82.2	74.5	27.1	89.2	33.6	72.3	70.8
High Stars	No Stars	50.9	53.2	76.4	68.7	36.1	86.4	36.6	66.5	71.9
No Stars	High Stars	31.8	71.8	66.2	69.6	32.9	86.9	15.6	87.1	62.2
No Stars	No Stars	73.5	28.5	87.5	81.9	19.3	92.2	39.6	62.9	73.4

Table 6. The percentage of all test-set variables that are in body-in-train functions.

Training Data	Test Set	Percent Body-in-Train
High Stars	High Stars	71.4
High Stars	No Stars	44.3
No Stars	High Stars	30.0
No Stars	No Stars	80.3

one and all others have probability zero; that is, every variable name is the same. At the other extreme, the maximum possible entropy occurs when every variable name that occurs does so with equal probability; that is, every variable name occurs the same number of times.

7.2.2 Results. We show results in Table 5. It is difficult to compare overall accuracy directly because the fraction of variables that are body-in-train varies considerably among the four training data/test set combinations, as shown in Table 6. Instead, we examine both the body-in-train and body-not-in-train portions separately.

In all four cases, the model performs well on the body-in-train portion, with accuracies of nearly 70% or above. The "cross testing" scenarios—where models trained on the high-star data is evaluated on the low-star test set and vice versa—receive lower overall accuracies and slightly lower VarCLR scores than the other two trials. This suggests that there are systematic differences in variables between high-and-low star code, though in all cases the model is often able to select a variable name that is appropriately similar. One cause for the existence of a body-in-train portion for the high-star model's predictions on the low star test set and vice versa is libraries. For example, the open-source `libavcodec` library, which helps encode and decode video content, is used in multiple projects across the high-star and low-star data sets related to multimedia.

The body-not-in-train, however, is different. The model trained on the high-star data yields fairly even results across all three metrics. However, while the model trained on low-star data performs very well on the low-star dataset, it performs especially poorly on the high-star dataset, with an accuracy of just 15.6%, under half any other body-not-in-train accuracy. Similarly, the CER is much higher and VarCLR score is much lower. This suggests that high-star data leads to models which are more generalizable.

The differences in performance can be attributed to a distribution mismatch between high-star and no-star datasets. We show the entropies of the high and no-star test sets with respect to the original developer names as well as with respect to the predicted names in Table 7. The high-star test set has significantly higher entropy than the low-star test set with respect to the developer-given variable names, indicating that high-star code uses a greater variety of variable names more often. Both overall and on the body-not-in-train portion, the entropies of the predictions by both models

Table 7. The entropies of the high-star and no-star data sets with respect to the original variable names as well as with respect to the predicted names. Entropy values are calculated based on a normalized frequency distribution of variable names in each test set. Lower entropy values indicate that the distribution is more skewed towards a few frequently used variable names. Entropies are measured in bits.

Training Data	Test Set	Entropy: Developer Names	Entropy: Predicted Names (Overall)	Entropy: Predicted Names (Body not in Train)
High-Star	High-Star	11.487	10.470	9.966
High-Star	No-Star	10.784	9.695	9.248
No-Star	High-Star	11.487	9.629	9.246
No-Star	No-Star	10.784	10.046	9.663

Model	Test Set	Entropy Loss
High-Star	High-Star	1.017
High-Star	No-Star	1.090
No-Star	High-Star	1.857
No-Star	No-Star	0.738

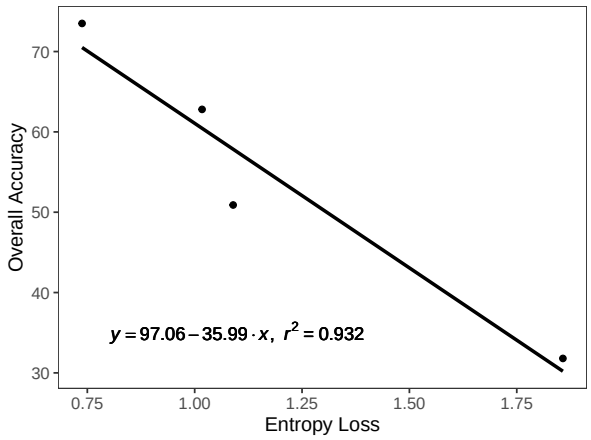
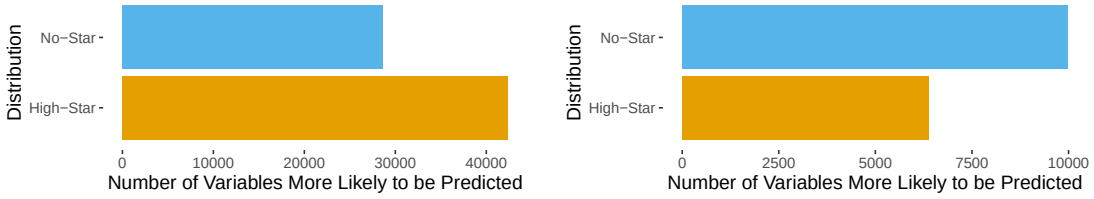


Fig. 13. The table on the left shows the entropy loss. Entropy loss is calculated by subtracting the entropy of the distribution of developers' names from the entropy of the distribution of the predictions. Entropy loss is highly correlated with accuracy as shown right. While overall accuracy is shown, this holds true for other types of accuracy as well.

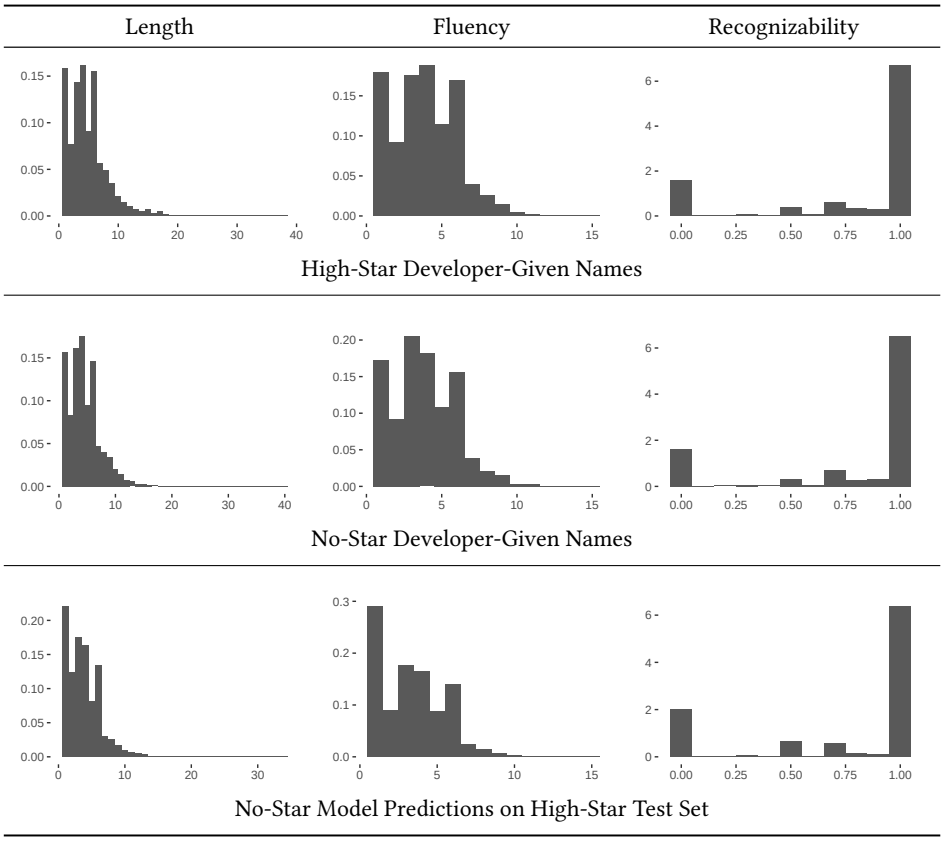


(a) On the union of the supports

(b) On the intersection of the supports

Fig. 14. Each bar in the bar graphs represents the number of variables for which that distribution has a higher probability value than the other distribution. The high-star distribution is more diverse; density of the high-star distribution is better spread over a wider variety of variable names, while much of the density of the no-star distribution is concentrated in the most common types of variable names.

Table 8. Values of the three quality metrics both on the original developer-given variable names as well as on the predictions of the No-Star model on the High-Star training set. None of the distributions of any quality characterization for any predictions are meaningfully different from one another, except the predictions of the No-Star distribution on the High-Star test set, shown in the last row.



are lower than the original entropy for that test set; DIRE tends to predict common variables disproportionally often. In effect, the training process results in a narrowing of the distribution, skewing it towards common variable names at the expense of less common ones. Because the high-star distribution starts from a position of higher entropy, it is in a better position after the narrowing effect of training occurs.

We can measure the narrowing of the distribution by calculating the amount of entropy lost as shown in Fig. 13. No-star data has has a lower entropy (and thus is less diverse) than high-star data. After training's narrowing effect, the no-star model's predictions are even less diverse, leading to especially poor performance. Figure 14 shows how the high-star model is more likely to predict a wider variety of variable names. Ultimately, such a higher-entropy, more diverse distribution helps the high-star model generalize better than its no-star counterpart.

However, we find that while high-star code is more diverse, the differences in the quality of variable names from high-star and no-star code is much less pronounced. We consider six collections of variables:

- The high-star developer-given names (High/Dev)
- The no-star developer-given names (No/Dev)

Table 9. *Cohen's d* differences in the means of each distribution of each set of variables for each quality metric. In the tables below, High/Dev represents the high-star code's developer names, High/No represents the high star model's predictions on the no-star test set, and so on. All differences are negligible (< 0.20) except when comparing to the no star model's predictions on the high-star test set, for which the difference is small (<0.50) when compared with the original developers' names of both high-star and no-star code in terms of both length and fluency. The difference of the no-star model's predictions on high-star test set compared to the high-star model's predictions on its own test set is also small in terms of fluency.

(a) Length

	High/Dev	No/Dev	High/High	No/High	High/No	No/No
High/Dev	0.00	0.10	0.05	**0.28**	0.16	0.20
No/Dev		0.00	0.00	**0.21**	0.10	0.11
High/High			0.00	0.12	0.06	0.05
No/High				0.00	0.07	0.12
High/No					0.00	0.02
No/No						0.00

(b) Fluency

	High/Dev	No/Dev	High/High	No/High	High/No	No/No
High/Dev	0.00	0.02	0.09	**0.32**	0.15	0.10
No/Dev		0.00	0.07	**0.31**	0.14	0.08
High/High			0.00	**0.23**	0.06	0.01
No/High				0.00	0.16	**0.22**
High/No					0.00	0.05
No/No						0.00

(c) Recognizability

	High/Dev	No/Dev	High/High	No/High	High/No	No/No
High/Dev	0.00	0.00	0.05	0.16	0.09	0.08
No/Dev		0.00	0.05	0.16	0.09	0.08
High/High			0.00	0.12	0.05	0.04
No/High				0.00	0.07	0.08
High/No					0.00	0.01
No/No						0.00

- The high-star model's predictions on the high-star test set (High/High)
- The no-star model's predictions on the high-star test set (No/High)
- The high-star model's predictions on the low star test set (High/No)
- The no-star model's predictions on the no star test set (No/No)

Distributions in variable quality on each metric are shown in Table 8. The distributions are largely identical for five of the six collections of variables. The outlier is the no-star model's predictions on the high-star test set (the No/High set). Table 9 quantifies these differences with the Cohen's d scaled differences between the means. All differences are negligible (< 0.20) except when the No/High set (the lowest entropy of the sets of variables) is compared with the highest-entropy sets of variables on length and fluency (bolded in Table 9). The no-star model performs poorly on variable names from the more diverse high-star test set and often chooses shorter, more generic variable names for

many of the names it fails to predict correctly. Indeed, the predictions of the no-star model on variable names it predicts incorrectly are shorter on average (3.50 vs. 4.10) and less fluent (2.98 vs. 3.62) than those it predicts correctly (Cohen's d 0.125 and 0.327, respectively). Thus, although high-star and no-star code don't have meaningfully different variable name quality, models trained on low-star code may have somewhat lower quality names.

RQ5 Answer: Training DIRE on data with higher variable-name distribution entropies leads to a more generalizable model. High star code has a higher entropy distribution of variable names than no-star code.

7.3 RQ6: Effects of Software Domain

In Section 7.2 we show that training causes the model to "narrow" its distribution, making it less likely to predict variable names which are already relatively unlikely. Unfortunately, sometimes these rare variable names play important roles. This is the case for domain-specific identifiers, which can provide crucial context for some functions. To help DIRE account for this effect and predict domain-specific identifiers more accurately, we modify DIRE to make it *domain-aware*.

7.3.1 Methodology. We performed several interventions on DIRE to incorporate domain information:

- Borrowing a technique from natural language processing, we added *domain labels* as appropriate to each member of each class. Labels consisted of the domain name surrounded by angle brackets, as in `<kernel>` or `<shell>`. For consistency, all of the additional data with indeterminate domain were given the label `<unknown>`. Domain labels were added to the vocabulary as special tokens. To incorporate domain labels into the sequence model, we prepended them to the input token sequence; to incorporate them into the graph model, we added an additional "label" node with edges to every AST node. The goal of this process is to create an association between a particular domain and identifiers common in that domain.
- Recall from Section 4.2.2 that the names of named AST nodes are incorporated into the model through an embedding computed by averaging the subtokens of that name. By default, DIRE considers the name of a constant to be a generic placeholder value based on that constant's type, meaning all constants of the same type have identical nodes in the AST. Similarly, all constants are replaced with generic placeholders based on their types in the lexical encoder as well. Due to our earlier finding that string literals are indicative of domain, we opted to not replace string literals with these generic placeholders and instead leave them in. This necessitated increasing the model's vocabulary size to include words that appear in string literals but not identifiers.

Unfortunately, domain-specific data is relatively scarce because many GitHub project maintainers do not elect to tag their repositories. To address this challenge, we sampled additional data from a pool of data with indeterminate domain. This "unknown" data allows DIRE to learn about common patterns in code, while the tagged domain-specific data help DIRE adjust to the idiosyncrasies of particular domains.

We created train/development/test splits for each domain and the unknown data separately. We trained the model on a pool of all training data from all domains and the unknown data, but kept each test set separate. To evaluate our model, we report accuracy numbers for each test set separately—one for each of the seven domains, and one for the unknown data—for a total of eight.

In addition, we trained two control models following the same process (and using the same data with the same train/development/test splits), but without any of the interventions. Thus, the control trials had no domain labels and had placeholders instead of string literals. The two control models varied in vocabulary size. Vocabulary size incurs a

Table 10. A comparison of the original DIRE model vs. the domain-aware version. Results for both DIRE and its lexical component are shown. Accuracy information for each domain and for performance on the "unknown" data set sampled from the data used in RQ1 are shown. Domain-Aware DIRE requires a larger vocabulary size than the original version of DIRE because it must handle tokens in string literals. Increasing the vocabulary size improves the performance of the lexical component of DIRE; therefore, results for the original version of DIRE are shown at both the original (10000 subtokens) and expanded (30000 subtokens) vocabulary sizes. Across all domains, the lexical component of DIRE with interventions applied outperforms the model as a whole.

(a) Overall

	Lexical Model			DIRE		
Domain	Original (10K Vocab)	Original (30K Vocab)	Domain-Aware (30K Vocab)	Original (10K Vocab)	Original (30K Vocab)	Domain-Aware (30K Vocab)
unknown	73.92%	76.85%	80.16%	72.38%	74.22%	75.90%
kernel	69.73%	72.28%	78.64%	67.84%	68.87%	71.70%
game	34.92%	42.48%	49.46%	34.77%	38.30%	40.77%
shell	33.69%	37.22%	43.25%	32.71%	33.85%	35.24%
compiler	43.70%	46.90%	51.35%	42.89%	43.49%	47.38%
network	57.77%	61.30%	64.00%	56.45%	58.99%	57.72%
embedded	76.33%	75.84%	84.84%	72.29%	72.57%	75.86%
graphics	22.97%	23.18%	31.80%	21.12%	18.50%	23.74%
Mean (Domains)	48.44%	51.31%	57.62%	46.87%	47.80%	50.34%
Performance Rank	4	2	1	6	5	3

(b) Body not in Train

	Lexical Model			DIRE		
Domain	Original (10K Vocab)	Original (30K Vocab)	Domain-Aware (30K Vocab)	Original (10K Vocab)	Original (30K Vocab)	Domain-Aware (30K Vocab)
unknown	35.89%	38.55%	43.85%	32.88%	35.51%	37.09%
kernel	39.99%	39.94%	46.07%	36.21%	38.65%	38.86%
game	22.40%	26.84%	32.51%	21.46%	23.23%	24.34%
shell	19.63%	20.90%	24.49%	19.10%	17.73%	19.20%
compiler	22.97%	24.93%	27.42%	21.97%	21.51%	24.68%
network	20.13%	23.03%	25.60%	20.77%	17.55%	18.84%
embedded	46.19%	48.13%	54.33%	43.87%	43.83%	46.26%
graphics	18.58%	18.28%	26.67%	16.57%	13.14%	18.40%
Mean (Domains)	27.13%	28.86%	33.87%	25.71%	25.09%	27.23%
Performance Rank	4	2	1	5	6	3

trade off: a larger vocabulary increases memory consumption and can become a computational bottleneck [21], but a larger vocabulary can boost accuracy as the model needs to predict fewer subwords per variable name on average [46]. We trained one model with the same vocabulary size as used in the evaluation of RQ1 (10,000) and another with the same vocabulary size as used in the domain-aware version of DIRE (30,000). All other hyperparameters were held constant, though embedding sizes were increased by 50% (on all models) to give the model additional capacity to store domain information. We report results for both DIRE and its lexical component of DIRE alone, which performed almost as well as DIRE—within 2%—in the evaluation of RQ1 (see Table 1).

7.3.2 Results. Table 10 shows the results. With the interventions applied, the lexical component of DIRE alone performed the best with an average accuracy score of 57.62% across all seven domain-specific data sets. Then next-best performing model, the lexical model with expanded vocabulary size, had an accuracy of 51.31%. Increasing the vocabulary size alone impacted the lexical model's performance, but not the performance of DIRE. Although the difficulty of different domains varied significantly, gains across domains were fairly consistent. While the lexical model performed better than DIRE, in both cases, applying the interventions lead to increased performance over the controls. The performance of the lexical model with interventions applied improves the accuracy on domain-specific test sets to an average of 57.62% from an average of 46.87% on the original base DIRE model, a 22.94% increase.

RQ6 Answer: By applying domain-labelling techniques and harnessing the domain-rich information in string literals, DIRE can be made domain-aware and its best performing submodel alone can predict identifiers correctly in domain-specific scenarios 22.94% more frequently than the base DIRE model.

8 THREATS TO VALIDITY

In answering research questions 1-4, when collecting code and binaries to generate our corpus, we did no filtering of the repositories beyond ensuring that they were written in C and built. In answering all of our research questions, it is possible that the code we collected does not accurately represent the types of binaries that are typically targets of reverse-engineering effort. Additionally, we did not experiment with binaries compiled with optimization enabled, nor did we experiment with intentionally obfuscated code. It is possible that DIRE does not perform as well on these binaries. However, reverse engineering of these binaries is a general challenge for decompilers, and we do not believe that our technique applies exclusively to the test code we experimented with. Although we have found that it is possible to uniquely identify variables in Hex-Rays based on the code offsets where it is accessed, we have found that other decompilers do not have this property. In particular, our approach did not work well with the newly released Ghidra decompiler [18]. One of the primary causes is the way that Hex-Rays and Ghidra use debug symbols to name variables. Hex-Rays uses debug symbols in a very straight-forward manner, and generally does not propagate local names outside of their function. Ghidra, however, will actually propagate variable names at some function calls. For example, if an unnamed variable is passed as an argument to a function whose parameter has a name, in some cases Ghidra will rename the variable to match the parameter's name. This behavior is problematic for corpus generation because it does not reflect the developer's intended names. A new approach for corpus generation would be required for compatibility with Ghidra, but Ghidra's open-source nature (as opposed to Hex-Rays' closed model) allows potential modification of the decompiler, including disabling the problematic propagation of names at function calls. We leave Ghidra integration to future work.

In answering research question 5, we selected a set of metrics to characterize variable names in various corpora. Determining what counts as a "good" variable name is a difficult problem and is often both subjective and context dependent. We acknowledge that there may be other ways to characterize variable name quality that we did not consider.

Ultimately, all of the metrics used to evaluate DIRE are imperfect proxies for the quality that is most important: how useful the variable names generated by DIRE are to reverse engineers. We leave a study on how reverse engineers interact with DIRE to future work.

Finally, while we evaluate DIRE against prior state-of-the-art approaches in Section 6.3, we only do so with a random sample from the entire dataset. It is possible that some techniques perform best in specific circumstances. We leave this to future work.

9 CONCLUSION

Semantically meaningful variable names are known to increase code understandability, but they generally cannot be recovered by decompilers. In this paper, we presented the **D**ecompiled **I**dentifier **R**enaming **E**ngine (DIRE), a probabilistic technique for variable name recovery which uses both lexical and structural information. We also presented a technique for generating corpora suitable for training DIRE, which we used to generate a corpus from 164,632 unique x86-64 binaries.

In addition, we extended the work originally published at ASE 2019 [31] to demonstrate that the choice of data used to train DIRE can have a profound impact on the resulting model. In particular, we find that models trained on data with more diverse, higher-entropy variable-name distributions tend to generalize better, and that the training process concentrates probability towards variable names which were already more common in the original distribution.

As another novel contribution in this extended paper, we modify DIRE to incorporate software domain information to help alleviate the impact of the latter effect on rare but important domain-specific variable names. In other words, we find that we can improve generalizability and performance by better selecting and augmenting the training data. While we performed our experiments on DIRE, the principle is not fundamentally tied to DIRE and we expect it to be generally applicable to machine learning models on code.

Our findings suggest that it may be possible to predict the performance of a model trained on a specific class of data on a specific test set without training. Assuming most of the probability in each variable name probability distribution falls on the intersection of the supports, it may be possible to use regression as in Fig. 13 to predict accuracy if the "narrowing" effect of training is taken into account. We leave this to future work.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the Software Engineering Institute (LINE project 6-18-001) and National Science Foundation (awards 1815287 and 1910067). We also gratefully acknowledge hardware support (Quadro P6000 GPU) from the NVIDIA Corporation. Many thanks to both Prem Devanbu and members of the CERT Division at the Software Engineering Institute for helpful feedback on earlier drafts.

REFERENCES

[1] Miltiadis Allamanis. 2019. 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*. 143–153.

[2] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM.

[3] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Symposium on the Foundations of Software Engineering (FSE '14)*. 281–293.

[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE '15)*. 38–49.

[5] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations (ICLR '18)*.

[7] Uri Alon, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *ICLR*.

[8] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2014. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering* 40, 5 (2014), 502–532.

[9] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016

[10] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting Decompiler Output with Learned Variable Names and Types. In *USENIX Security Symposium (Oakland '22)*.

[11] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2022. VarCLR: Variable Semantic Representation Pre-training via Contrastive Learning. In *International Conference on Software Engineering (ICSE '22)*.

[12] Kyunghyun Cho, Aaron Courville, and Yoshua Bengio. 2015. Describing multimedia content using attention-based encoder-decoder networks. *IEEE Transactions on Multimedia* 17, 11 (2015), 1875–1886.

[13] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP '14)*.

[14] Premkumar Devanbu. 2015. New initiative: The naturalness of software. In *International Conference on Software Engineering (ICSE '15)*. 543–546.

[15] Lukas Durfina, Jakub Kroustek, and Petr Zemek. 2013. PsybOt malware: A step-by-step decompilation case study. In *Working Conference on Reverse Engineering (WCRE '13)*. 449–456.

[16] Michael J. Eager. 2012. Introduction to the DWARF Debugging Format. http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf

[17] Edward M. Gellenbeck and Curtis R. Cook. 1991. *An Investigation of Procedure and Variable Names as Beacons During Program Comprehension.* Technical Report. Oregon State University.

[18] Ghidra. 2019. *The Ghidra decompiler.* https://ghidra-sre.org/

[19] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 1263–1272.

[20] Georgios Gousios. 2013. The GHTorrent Dataset and Tool Suite. In *Working Conference on Mining Software Repositories (MSR '13)*. 233–236.

[21] Thamme Gowda and Jonathan May. 2020. Finding the Optimal Vocabulary Size for Neural Machine Translation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*. 3955–3964.

[22] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *Conference on Computer and Communications Security (CCS '18)*.

[23] Hex-Rays. 2019. *The Hex-Rays decompiler.* https://www.hex-rays.com/products/decompiler/

[24] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[25] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[26] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful Variable Names for Decompiled Code: A Machine Translation Approach. In *International Conference on Program Comprehension (ICPC '18)*. 20–30.

[27] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. 2019. Assessing the generalizability of code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–12.

[28] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1073–1085.

[29] Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using Recurrent Neural Networks for Decompilation. In *International Conference on Software Analysis, Evolution and Reegnineering (SANER '18)*. 346–356.

[30] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 66–71.

[31] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Renaming. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. San Diego, California.

[32] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *International Conference on Program Comprehension (ICPC '06)*. 3–12.

[33] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[34] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2021. Deduplicating training data makes language models better. *arXiv preprint arXiv:2107.06499* (2021).

[35] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[36] Hui Liu, Qiurong Liu, Yang Liu, and Zhouding Wang. 2015. Identifying renaming opportunities by expanding conducted rename refactorings. *IEEE Transactions on Software Engineering* 41, 9 (2015), 887–900.

[37] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1–12.

[38] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *EMNLP*.

[39] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. 2021. A survey on bias and fairness in machine learning. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–35.

[40] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. DIRECT: A Transformer-based Model for Decompiled Variable Name Recovery. *NLP4Prog 2021* (2021), 48.

[41] Lucas Nussbaum and Stefano Zacchiroli. 2010. The Ultimate Debian Database: Consolidating bazaar metadata for Quality Assurance and data mining. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 52–61. https://doi.org/10.1109/MSR.2010.5463277

[42] Wiliiam H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. *Numerical Recipes in C: The Art of Scientific Computing* (2nd ed.). Cambridge University Press, Chapter 20.2 Gray Codes, 896.

[43] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.

[44] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Symposium on Principles of Programming Languages (POPL '15)*. 111–124.

[45] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. 1988. Learning representations by back-propagating errors. *Cognitive Modeling* 5, 3 (1988), 1.

[46] Elizabeth Salesky, Andrew Runge, Alex Coda, Jan Niehues, and Graham Neubig. 2020. Optimizing segmentation granularity for neural machine translation. *Machine Translation* 34, 1 (2020), 41–59.

[47] Felice Salviulo and Giuseppe Scanniello. 2014. Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance: Results from an Ethnographically-Informed Study with Students and Professionals. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. Association for Computing Machinery, New York, NY, USA, Article 48, 10 pages. https://doi.org/10.1145/2601248.2601251

[48] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80.

[49] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. 2013. Native x86 Decompilation Using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring. In *USENIX Security Symposium (USENIXSEC '13)*. 353–368.

[50] Diomidis Spinellis, Zoe Kotti, and Audris Mockus. 2020. A Dataset for GitHub Repository Deduplication. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 523–527. https://doi.org/10.1145/3379597.3387496

[51] JHU/APL Staff. 2019. Assembled Labeled Library for Static Analysis Research (ALLSTAR) Dataset. http://allstar.jhuapl.edu/

[52] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 2018. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *International Conference on Software Engineering (ICSE)*. 511–522.

[53] Michael James van Emmerik. 2007. *Static Single Assignment for Decompilation*. Ph.D. Dissertation. University of Queensland.

[54] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering Clear, Natural Identifiers from Obfuscated JavaScript Names. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE '17)*. 683–693.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[56] Weiyue Wang, Jan-Thorsten Peter, Hendrik Rosendahl, and Hermann Ney. 2016. CharacTer: Translation Edit Rate on Character Level. In *WMT '16*. 505–510.

[57] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *IEEE Symposium on Security and Privacy (SP '16)*. 158–177.

[58] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Network and Distributed System Security Symposium (NDSS '15)*.