DIRE: A Neural Approach to Decompiled Identifier Naming

Jeremy Lacomis*, Pengcheng Yin*, Edward J. Schwartz†, Miltiadis Allamanis‡,
Claire Le Goues*, Graham Neubig*, Bogdan Vasilescu*

*Carnegie Mellon University. {jlacomis, pcyin, clegoues, gneubig}@cs.cmu.edu; vasilescu@cmu.edu
†Carnegie Mellon University Software Engineering Institute. eschwartz@cert.org
‡Microsoft Research. miallama@microsoft.com

Abstract—The *decompiler* is one of the most common tools for examining binaries without corresponding source code. It transforms binaries into high-level code, reversing the compilation process. Decompilers can reconstruct much of the information that is lost during the compilation process (e.g., structure and type information). Unfortunately, they do not reconstruct semantically meaningful variable names, which are known to increase code understandability. We propose the Decompiled Identifier Renaming Engine (DIRE), a novel probabilistic technique for variable name recovery that uses both lexical and structural information recovered by the decompiler. We also present a technique for generating corpora suitable for training and evaluating models of decompiled code renaming, which we use to create a corpus of 164,632 unique x86-64 binaries generated from C projects mined from GITHUB.[1] Our results show that on this corpus DIRE can predict variable names identical to the names in the original source code up to 74.3% of the time.

I. INTRODUCTION

Software reverse engineering is the problem of understanding the behavior of a program without having access to its source code. Reverse engineering is often used to predict the behavior of malware [1]–[3], discover vulnerabilities [1], [4], [5], and patch bugs in legacy software [4], [5]. For malware and malicious botnets, reverse engineering enables understanding and response, and helps identify and patch infection vectors. For example, by reverse engineering the Torbig botnet (which caused 180K infections and collected 70 GB of credit card/bank account information), responders were able to predict future domain names that bots would contact, and redirect the bots to servers under the responders' control [6]. Reverse engineering can also help identify who created a piece of malware, as was done for the Uroburos rootkit [7] (which captured files and network traffic while propagating over networks of companies and public authorities), and estimate the extent of infection [8].

One of the main tools reverse engineers use to inspect programs is the *disassembler*—a tool that translates a binary to low-level assembly code. Disassemblers range from simple tools like GNU Binutils' objdump [9], to more advanced tools like IDA [10], which can be used interactively and have more sophisticated features. However, reasoning at the assembly level requires considerable cognitive effort even with

these advanced features [2], [4], [5]. More recently, reverse engineers are employing *decompilers* such as Hex-Rays [11] and Ghidra [12], which reverse compilation by translating the output of disassemblers into code that resembles high-level languages such as C, to reduce the cognitive burden of understanding assembly code. These state-of-the-art tools are able to use program analysis and heuristics to reconstruct information about a program's variables, types, functions, and control flow structure.

Even though decompiler output is more understandable than raw assembly, 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. 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 [13], [14], become nothing more than arbitrary placeholders such as `VAR1` and `VAR2`.

In this work, we present DIRE (**D**ecompiled **I**dentifier **R**enaming **E**ngine), a novel neural network approach for assigning meaningful names to variables in decompiled code (Section III). To build DIRE, we rely on two key insights. Our first insight is that software is *natural*, i.e., programmers tend to write similar code and use the same variable names in similar contexts [15], [16]. Therefore, because of this repetitiveness, if given a large enough training corpus one can *learn* appropriate variable names for a particular context.

Prior approaches exist to predict natural variable names from both source code [17]–[20] and compiled executables [21], [22]. However, approaches to predict variable names from executables either operate directly on the binary semantics [22], [23], or on the lexical output of the decompiler [21]. The former ignores the rich abstractions that modern decompilers are able to recover. The latter is an improvement, but a lexical program representation is by its very nature sequential, and lacks rich structural information that could be used to improve predictions. In contrast, DIRE uses the extended context provided by the decompiler's internal abstract syntax tree (AST) representation of the decompiled binary, which encodes additional *structural* information.

To train such models, one needs training data that specifies

---

[1]Data available at https://doi.org/10.5281/zenodo.3403077

IEEE computer society

what names are natural in what contexts. Our second key insight is that unlike other domains, where creating training data often requires manual curation (e.g., machine translation [24]), it is possible to *automatically* generate large amounts of training data for identifier name prediction, To that end, we mine open-source C code from GITHUB, compile it *with debugging information* such that the binaries preserve the original names, and decompile those binaries so that the output contains the original names. We then strip the debug symbols, decompile the binary again, and identify the alignment between the identifiers in the two versions of the decompiler outputs. While this is conceptually straight-forward, the two outputs are not simply α-renamings, making the process of calculating these alignments far from trivial. Prior work identified alignments based entirely on heuristics [21]. In contrast, we observe that the set of instruction addresses that access each variable uniquely identifies that variable, and this can be used to generate accurate alignments (Section IV).

With these insights we train and evaluate DIRE on a large dataset of C code mined from GITHUB, showing that we can predict variable names identical to those chosen by the original developer up to 74.3% of the time. In short, we contribute:

- **D**ecompiled **I**dentifier **R**enaming **E**ngine (DIRE), a technique for assigning meaningful names to decompiled variables that outperforms previous approaches.
- A novel technique for generating corpora suitable for training both lexical and graph-based probabilistic models of variable names in decompiled code.
- A dataset of 3,195,962 decompiled x86-64 functions and parse trees annotated with gold-standard variable names.[1]

II. 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).

A. 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 [25]. 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
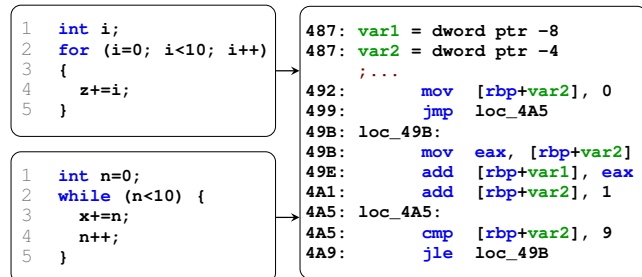

Fig. 1: Two different C loops that compile to the same assembly code. Note the normalized structure and names.

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. 1 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 [26]. 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.

B. Statistical Models of Source Code

A wide variety of statistical models for representing source code have been proposed based on the *naturalness* of software [15], [16]. 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 [27].

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 the variable is used in. These naming models can help to distill coding conventions [28] or analyze obfuscated code [17], [18]. Several classes of statistical models have been used for renaming, including n-grams [18], [28], conditional random fields (CRFs) [17], and deep learning models [29]–[31].

Two recent approaches aim to suggest informative variable names in decompiled code. Our prior work [21] 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.

C. 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).

1) Recurrent Neural Networks: RNNs are networks where connections between nodes form a sequence [32]. They are typically used to process sequences of inputs by reading in one element at a time, making them well-suited to sequences, such as source code tokens. In this work, we use long short-term memory (LSTM) models [33], 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 $\overrightarrow{h_i}$ for each subsequence up to token x_i using the recurrent function $\overrightarrow{h_i} = \overrightarrow{f}_{\text{LSTM}}(\text{emb}(x_i), \overrightarrow{h_{i-1}})$, 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 III, 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 III-B1) 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 III-C).

2) Gated-Graph Neural Networks: While LSTMs are useful for modeling 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 [34], [35]. 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 node based on the initial node 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 $h_{i,t}^g$ 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) [36]. 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 $m_{\mathcal{T}}^{v_j \mapsto v_i} = W_{\mathcal{T}} \cdot h_{j,t-1}^g$, where $W_{\mathcal{T}}$ is a type-specific weight matrix. Then, all $m_*^{v_* \mapsto v_i}$ are aggregated, and summarized into a single vector x_i^g via element-wise mean (pooling):

$$x_i^g = \mathsf{MeanPool}(\{m_{\mathcal{T}}^{v_j \mapsto v_i} : v_j \in \mathcal{N}_{\mathcal{T}}(v_i), \forall \mathcal{T}\}).$$
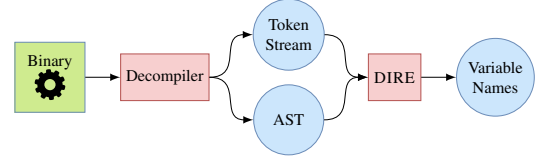

Fig. 2: High-level overview of our approach.

Finally, the state of every node v_i is updated using a nonlinear activation function f: $h_{i,t}^g = f(x_i^g, h_{i,t-1}^g)$. GGNNs use a Gated Recurrent Unit (GRU) update function, $f_{\text{GRU}}(\cdot)$, introduced by Cho et al. [37]. 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 of $f_{\text{GRU}}(\cdot)$ and the $W_{\mathcal{T}}$s) are optimized along with the rest of the model.

III. THE DIRE ARCHITECTURE

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

A. 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. 2 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 each placeholder in the input. The decompiler output can then be rewritten to include the suggested variable names.

Fig. 3 gives an overview of the neural architecture. DIRE follows an encoder-decoder architecture: An *encoder* neural network (Section III-B) 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 III-C) 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 III-B1) and a *structural encoder* (Section III-B2)—to separately capture the lexical and structural signals in the decompiled code. As we will show, this combination of

630

lexical and structural information allows DIRE to outperform techniques that rely on lexical information alone [21].

B. The Encoder Network

Each encoder network in DIRE outputs two sets of 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. 3). 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 III-C.

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`) is automatically broken down into sub-pieces (e.g., `my`, `str`, and `copy`) using SentencePiece [38], 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 II-C1). 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 [27]. 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: $\boldsymbol{v} = \text{MeanPool}(\mathcal{H}_v)$.

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 II-C2). This requires a mechanism to compute initial node states, as well as design choices of which AST edges to consider in the node encoding:

a) Initial Node States: The initial state of a node v_i, $\boldsymbol{h}_{i,t=0}^g$ 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 note in the AST in Fig. 3 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. 3 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. 3 has a name `mystrcopy`), computed by averaging the embeddings of its content subtokens. The initial state $\boldsymbol{h}_{v,t=0}^g$ 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.

b) 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. 3) in the original AST, we also augment it with additional edges [30]:

- 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. 3) 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") [36].
- Finally, we add a reverse edge for all edge types defined above, modeling bidirectional information flow.

c) Representations: For the *element* representation, we use the final state of the GGNN for node n_i, $\boldsymbol{h}_{i,T}^g$, as its representation: $\boldsymbol{n}_i = \boldsymbol{h}_{i,T}^g$ (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
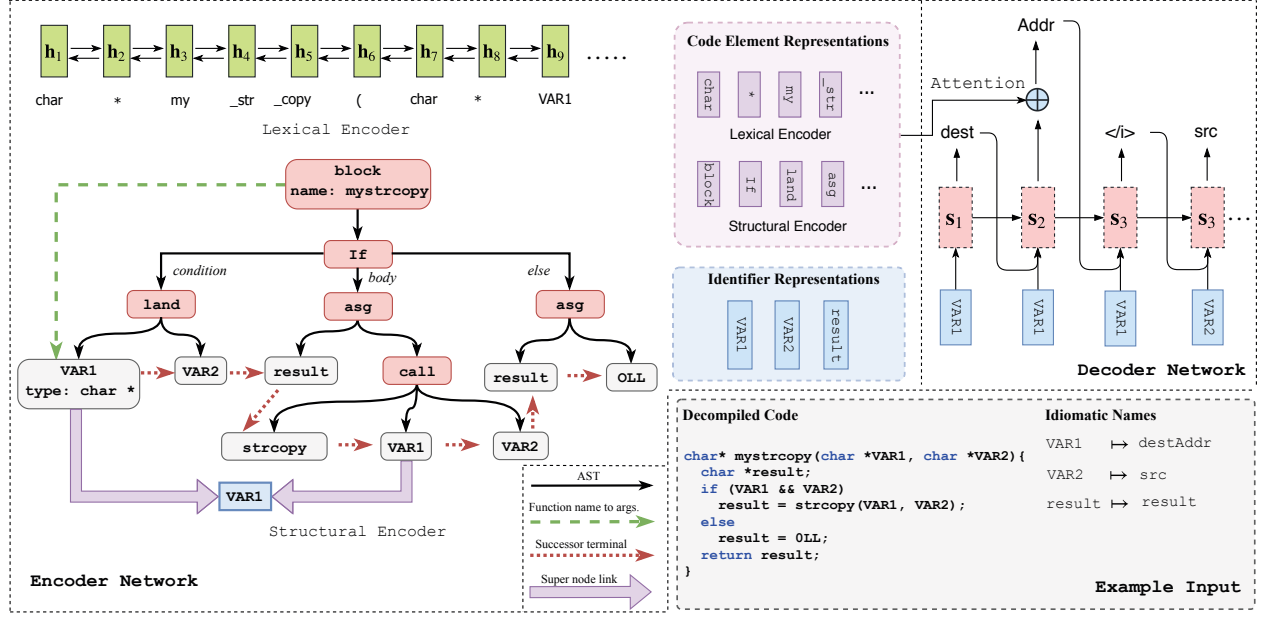
Fig. 3: Overview of DIRE's neural architecture. For clarity, we omit the data-flow links in the AST in the structural encoder.

all its mentions. Therefore, $\boldsymbol{v}_i$ captures information about the usage of v_i in different occurrences.

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 elements are combined 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; identifiers are combined by merging the lexical and structural representations of each identifier v using a linear transformation as its representation.

C. The Decoder Network

The decoder network predicts names for identifiers using the representations given by the encoder. As shown in Fig. 3, 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 III-B1).

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 IV, 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 in [39]. 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* [40] 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:

$$p(y_t|\cdot) = \frac{\exp\left(\boldsymbol{y}_t^\mathsf{T} \tilde{\boldsymbol{s}}_t\right)}{\sum_{y'} \exp\left(\boldsymbol{y}'^\mathsf{T} \tilde{\boldsymbol{s}}_t\right)}$$

D. 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 III-C, 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).

IV. GENERATION OF TRAINING DATA

Training DIRE requires a large corpus of annotated data. Fortunately, it is possible to create this corpus automatically, starting from a large repository of existing C source 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. 4. 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 II-A) 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 variable 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.

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 [21], we leverage the decompiler's ability to incorporate developer-chosen identifier


(a) Tokenized decompiled code with variable placeholders.

(b) AST with placeholders.

(c) Variable lookup table.
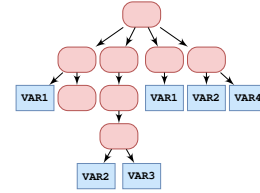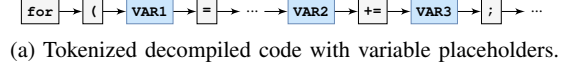
ID	Decompiler	Developer
1	v1	ans
2	v2	size
3	i	i
4	ptr	head

Fig. 4: 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.
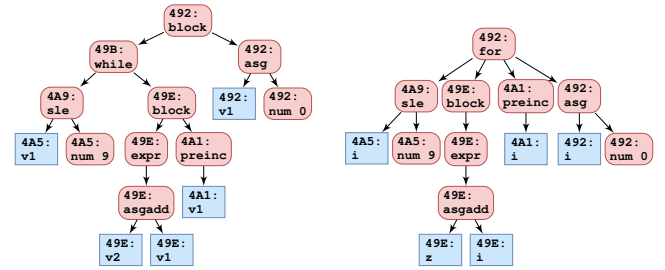
(a) AST without DWARF.
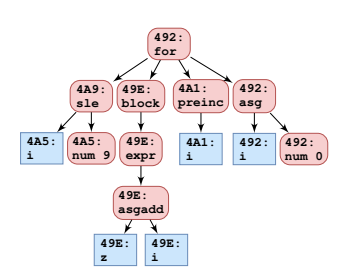
(b) AST with DWARF.

Fig. 5: Decompiler ASTs for the code in Fig. 1. 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`).

names into decompiled code when DWARF debugging symbols [26] 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 II 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. 5. 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 [21]. 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. 5, 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.[2]

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.

V. Evaluation

We ask the following research questions:
- RQ1: How effective is DIRE at assigning names to variables in decompiled code?
- RQ2: How does each component of DIRE contribute to its efficacy?

TABLE I: 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

- RQ3: How does provenance and quantity of data influence the efficacy of DIRE?
- RQ4: Is DIRE more effective than prior approaches?

a) 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.

b) 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 exactly matching 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 [41], assigning partial credit to near misses.

Recall from Section IV 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

```
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. 6: Decompiled function (simplified for presentation), DIRE variable names, and developer-assigned names.

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 likely be misleading.

c) Neural Network Configuration: For our experiments we replicate the neural network configuration of Allamanis et al. [30]. 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 testing time, we use beam search to predict the sequence of sub-tokenized names for each identifier (Section III-C), with a beam size of 5.

A. RQ1: Overall Effectiveness

Experimental results are summarized in Table I. 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 6 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 [42]. 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 functions in different binaries is the use of libraries. We argue that it is reasonable to allow

TABLE II: 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%)

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 sub-categories: **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 I 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.

Table II 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 complex identifiers with compositional names. However, DIRE is still often able to suggest semantically relevant identifiers (e.g., `node`, `child`).

> RQ1 Answer: We find that DIRE is able to suggest variable names identical to those chosen by the original developer 74.3% of the time.

B. RQ2: Component Contributions

Table I 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 7 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

```
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. 7: 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.

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.

C. RQ3: Effect of Data

To answer RQ3, 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%. The results of these experiments are shown in Fig. 8.

Figures 8a and 8b 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. 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 8c 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.

> RQ3 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.

D. RQ4: Comparison to Prior Work

To answer RQ4, we compare to our prior work [21] 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 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. 9, which contains a C function for converting between a number **a1** and its Gray code representation in **a2** bits [43]. Figure 9a 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 9b shows the output of Hex-Rays using debugging information generated using DEBIN's bundled model.[3] 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.

---

[3]https://files.sri.inf.ethz.ch/debin_models.tar.gz, accessed April 10, 2019

(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.
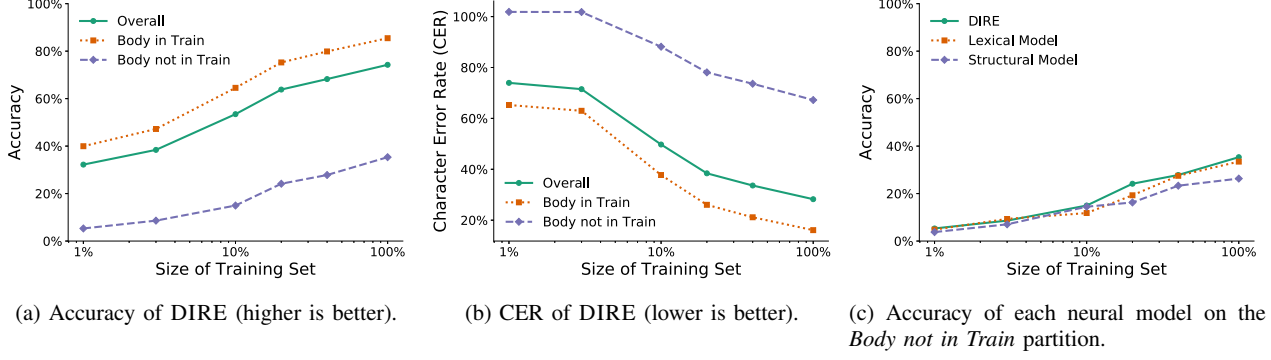
Fig. 8: The impact of training corpus size on the performance of DIRE. Figures (a) and (b) show how increasing the amount of training data improves the performance of DIRE; (c) shows the performance of each of the submodel as training size changes.

```
 1  long gray(unsigned a1,        1  void gray() {
 2            int a2) {            2    unsigned v0;
 3    unsigned v3, v4;             3    int v1;
 4    int v5;                      4    unsigned i, v3;
 5    if (a2 >= 0)                 5    int x;
 6      return a1 ^ (a1 >> 1);     6    if (v1 < 0) {
 7    v5 = 1;                      7      x = 1;
 8    v4 = a1;                     8      v3 = v0;
 9    while (1) {                  9      while (1) {
10      v3 = v4 >> v5;            10        i = v3 >> x;
11      v4 ^= v4 >> v5;           11        v3 ^= v3 >> x;
12      if (v3 <= 1 ||            12        if (i <= 1 ||
13          v5 == 16)             13            x == 16)
14        break;                  14          break;
15      v5 *= 2;                  15        x *= 2;
16    }                           16      }
17    return v4;                  17    }
18  }                             18  }
```

(a) Hex-Rays.   (b) Hex-Rays w/ DEBIN.

Fig. 9: Effects of incorrect debugging information on decompiler output. The `gray` function computes the Gray code of `a1` in `a2` bytes [43]. 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.

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. 9a 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

TABLE III: 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

sets.[4] 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 III.

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 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.

> RQ4 Answer: DIRE is a more accurate and more scalable technique for variable name selection than other state-of-the-art approaches.

VI. THREATS TO VALIDITY

When collecting code and binaries to generate our corpus, we did no filtering of the repositories beyond ensuring that

---

[4]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].

they were written in C and built. 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 [12]. One of the primary causes is the way that Hex-Rays and Ghidra utilize 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.

VII. Conclusion

Semantically meaningful variable names are known to increase code understandability, but they generally cannot be recovered by decompilers. In this paper, we proposed the **D**ecompiled **I**dentifier **R**enaming **E**ngine (DIRE), a novel, 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. Our experiments show that DIRE is able to predict variable names identical to the names used in the original source code up to 74.3% of the time.

VIII. Acknowledgments

References

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

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

[3] L. Durfina, J. Kroustek, and P. Zemek, "PsybOt malware: A step-by-step decompilation case study," in *Working Conference on Reverse Engineering*, ser. WCRE '13, 2013, pp. 449–456.

[4] M. J. van Emmerik, "Static single assignment for decompilation," Ph.D. dissertation, University of Queensland, 2007.

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

[6] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: Analysis of a botnet takeover," in *ACM Conference on Computer and Communications Security*, ser. CCS '09, November 2009.

[7] "Uroburos: Highly complex espionage software with Russian roots," G Data SecurityLabs, Tech. Rep., 2014.

[8] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos, "SoK: P2PWNED — Modeling and evaluating the resilience of peer-to-peer botnets," in *Symposium on Security and Privacy*, ser. SOSP '13, 2013, pp. 97–111.

[9] Binutils. (2019) objdump. [Online]. Available: https://www.gnu.org/software/binutils/

[10] IDA. (2019) Ida. [Online]. Available: https://www.hex-rays.com/products/ida/

[11] Hex-Rays. (2019) The hex-rays decompiler. [Online]. Available: https://www.hex-rays.com/products/decompiler/

[12] Ghidra. (2019) The ghidra decompiler. [Online]. Available: https://ghidra-sre.org/

[13] E. M. Gellenbeck and C. R. Cook, "An investigation of procedure and variable names as beacons during program comprehension," Oregon State University, Tech. Rep., 1991.

[14] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," in *International Conference on Program Comprehension*, ser. ICPC '06, 2006, pp. 3–12.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[16] P. Devanbu, "New initiative: The naturalness of software," in *International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 543–546.

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

[18] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "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*, ser. ESEC/FSE '17, 2017, pp. 683–693.

[19] R. Bavishi, M. Pradel, and K. Sen, "Context2Name: A deep learning-based approach to infer natural variable names from usage contexts," TU Darmstadt, Department of Computer Science, Tech. Rep., November 2017.

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Programming Language Design and Implementation*, ser. PLDI '18, 2018, pp. 404–419.

[21] A. Jaffe, J. Lacomis, E. J. Schwartz, C. Le Goues, and B. Vasilescu, "Meaningful variable names for decompiled code: A machine translation approach," in *International Conference on Program Comprehension*, ser. ICPC '18, May 2018, pp. 20–30.

[22] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "DEBIN: Predicting debug information in stripped binaries," in *Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[23] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries," 2019.

[24] P. Koehn, *Statistical machine translation*. Cambridge University Press, 2009.

[25] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER '18, 2018, pp. 346–356.

[26] M. J. Eager, "Introduction to the DWARF debugging format," April 2012. [Online]. Available: http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf

[27] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[28] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Symposium on the Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 281–293.

[29] ——, "Suggesting accurate method and class names," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, pp. 38–49.

[30] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, ser. ICLR '18, 2018.

[31] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *ICLR*, 2019.

[32] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning representations by back-propagating errors," *Cognitive Modeling*, vol. 5, no. 3, p. 1, 1988.

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

[34] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[35] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[36] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1263–1272.

[37] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP '14, 2014.

[38] T. Kudo and J. Richardson, "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*, 2018, pp. 66–71.

[39] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *EMNLP*, 2015.

[40] K. Cho, A. Courville, and Y. Bengio, "Describing multimedia content using attention-based encoder-decoder networks," *IEEE Transactions on Multimedia*, vol. 17, no. 11, pp. 1875–1886, 2015.

[41] W. Wang, J.-T. Peter, H. Rosendahl, and H. Ney, "CharacTer: Translation edit rate on character level," in *WMT '16*, 2016, pp. 505–510.

[42] M. Allamanis, "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, 2019.

[43] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1992, ch. 20.2 Gray Codes, p. 896.