Augmenting Decompiler Output with Learned Variable Names and Types

Qibin Chen[*], Jeremy Lacomis[*], Edward J. Schwartz[†],
Claire Le Goues[*], Graham Neubig[*], Bogdan Vasilescu[*]

[*]Carnegie Mellon University. {qibinc, jlacomis, clegoues, gneubig, bogdanv}@cs.cmu.edu
[†]Carnegie Mellon University Software Engineering Institute. eschwartz@cert.org

Abstract

A common tool used by security professionals for reverse-engineering binaries found in the wild is the *decompiler*. A decompiler attempts to reverse compilation, transforming a binary to a higher-level language such as C. High-level languages ease reasoning about programs by providing useful abstractions such as loops, typed variables, and comments, but these abstractions are lost during compilation. Decompilers are able to deterministically reconstruct structural properties of code, but comments, variable names, and custom variable types are technically impossible to recover.

In this paper we present DIRTY (DecompIled variable ReTYper), a novel technique for improving the quality of decompiler output that automatically generates meaningful variable names and types. DIRTY is built on a Transformer-based neural network model and is trained on code automatically scraped from repositories on GitHub. DIRTY uses this model to postprocesses decompiled files, recommending variable types and names given their context. Empirical evaluation on a novel dataset of C code mined from GitHub shows that DIRTY outperforms prior work approaches by a sizable margin, recovering the original names written by developers 66.4% of the time and the original types 75.8% of the time.

1 Introduction

Reverse engineering is an important problem in the context of software. For example, security professionals use reverse engineering to understand the behavior or provenance of malware [12, 54, 55], discover vulnerabilities in libraries [49, 55], or patch bugs in legacy software [49, 55]. However, since it is rare to have access to source code, analysis is often performed at the binary level. This can be challenging: compilers optimize for execution speed or binary size, not readability.

A number of tools attempt to make the process of examining binary programs easier. One is the *disassembler*, which converts raw binary code to a sequence of instructions executed by the compiler. Although this produces human readable code, reasoning about assembly code can still be difficult.

Operations that are simple to specify at a high-level are often translated into a long sequence of assembly instructions (e.g., looping over the elements of an array requires instructions that maintain an index variable, increment it each iteration of a loop, and conditionally jump on its value). Another, more sophisticated tool is a *decompiler*, which transforms code from binary to a high-level language such as C.

Although decompilers generate abstractions that improve code readability and are widely used by reverse engineers in practice, they never fully reconstruct the original developer-written code [43], since the process of compilation irrevocably destroys some information. This means that useful pieces of information, such as comments, identifier names, and types, all of which are known to meaningfully contribute to program comprehension [17, 30], are typically absent from decompiler output. Nonetheless, recent work has shown that it *is* possible to reconstruct some useful information about the original code during decompilation, namely identifier [25, 29] and procedural names [8], *even when this information is not part of the binary*. The key insight is that human-written code is often repetitive in the same context [2, 9, 23]. Therefore, given large corpora of human-written code, one can *learn* highly probable names for identifiers in similar contexts, even if not always the exact names the authors of the code chose originally. This is an important improvement on the facilities of modern decompilers, which almost completely ignore names beyond simple heuristics (e.g., `i` and `j` for loop guards).

In this paper we focus on the closely related problem of recovering meaningful variable *types*, an important additional layer of code documentation that can help improve readability and understandability [14, 43, 48]. Figure 1 shows an example of a simple function and its decompilation. The author of the original code in Figure 1a has defined a `pnt` structure that contains two `float` members used to refer to the X and Y coordinates of a point. This makes it possible to define a new point and refer to its members by name (e.g., `p1.x` and `p1.y`). Because the decompiler does not know about the `pnt` structure, it creates two `float` arrays instead of generating a `struct` (Figure 1b). This can harm understandability. First, it

```
typedef struct point {
  float x;
  float y;
} pnt;

void fun() {                    void fun() {
  pnt p1, p2;                     float v1[2], v2[2];
  p1.x = 1.5;                     v1[0] = 1.5;
  p1.y = 2.3;                     v1[1] = 2.3;
  // ...                          // ...
  use_pts(&p1, &p2);              use_pts(v1, v2);
}                               }
```

     (a) Original code   (b) Decompiled `fun`

Figure 1: A function with a `struct` and its decompilation.

```
void fun() {                    void fun() {
  // stack layout:                // stack layout:
  // [xxx][p][yyyy]                // [xxxx][yyyy]
  char x[3];                      char x[4];
  int y;                          int y;
  // ...                          // ...
}                               }
```

     (a) Original code   (b) Decompiled `fun`

Figure 2: A function illustrating the data layout problem in decompilation. In the stack layout the characters `x`, `y`, and `p` represent a single byte assigned to the variables `x` and `y`, or padding data respectively. The decompiler cannot recognize that the inserted padding data does not belong to the `x` array.

is not clear that `v1` and `v2` represent points at all. Second, even if better names were chosen, such as `point1` and `point2`, and a reverse engineer concluded that they represent 2D points, it is not clear which array index refers to which coordinate, or even that the coordinates are Cartesian (instead of e.g., polar).

Unlike names, types are constrained by memory layouts, and thus theoretically should be easier to recover (only types that fit that memory layout should be considered as candidates). In fact, decompilers already narrow down possible type choices using the fact that base types targeting a specific platform can only be assigned to variables with a specific memory layout (e.g., on most platforms an `int` variable can never be retyped to a `char` because they require different amounts of memory). This already makes it possible for decompilers to infer base types and a small set of commonly-used `typedefs`.

On the other hand, despite performing a battery of complex binary analyses, the data layout inferred by the decompiler is often incorrect, which makes the problem harder. For example, consider the program shown in Figure 2. Two top-level variables are declared, `x`: a three-byte `char` array, and `y`: a four-byte `int`. During compilation, the compiler inserts a single byte of padding after the `x` array for alignment. When this function is decompiled, the decompiler can tell where `x` and `y` begin, but it cannot tell if `x` is a three-byte array followed by a single byte of padding, or a four-byte array whose last element is never used.

Prior work on reconstructing types falls into two groups. The first, such as TIE [31], attempt to recover *syntactic* types,

e.g., `struct {float; float}`, but not the names of the structure or its fields. The second, such as REWARDS [33], attempt to also recover the type name (referred to as *semantic* types). However, these systems typically only support a small set of manually-defined types and well-known library calls. Neither the first nor the second deal with the padding issue above.

In contrast, our system DIRTY (DecompIled variable ReTYper) recovers both semantic and syntactic types, handles padding, and is not limited to a small set of manually-defined types. Instead, DIRTY supports 48,888 possible types encountered "in the wild" in open-source C code (compared to the 150 different type names in 84 standard library calls supported by REWARDS). At a high level, DIRTY is a *Transformer-based [50] neural network model to recommend types* in a particular context, which operates as a postprocessing step to decompilation. DIRTY takes a decompiled function as input, and outputs probable names and types for all of its variables.

To build DIRTY, we start by mining open-source C code from GitHub, and then use a decompiler's typical ability to import variable names and types from DWARF debugging information to create a *parallel corpus of decompiled functions with and without their corresponding original names and types*. As a side effect of this large-scale mining effort, we also automatically compile a *library of types* encountered across our open-source corpus. We then train DIRTY on this data, introducing two task-specific innovations. First, we use a Data Layout Encoder to incorporate memory layout information into DIRTY's predictions and simultaneously address a fundamental limitation of decompilers caused by padding. Second, we address both the variable renaming and retyping tasks simultaneously with a joint Multi-Task architecture, enabling them to benefit from each other.

We show that DIRTY can assign variable *types* that agree with those written by developers up to 75.8% of the time, and DIRTY also outperforms prior work on variable *names*.

Note that even though we implement DIRTY on top of the Hex-Rays[1] decompiler because of its positive reputation and its programmatic access to decompiler internals, our approach is not fundamentally specific to Hex-Rays, and should conceptually work with any decompiler that names variables using DWARF debug symbols.

In summary, we contribute:

- DIRT—the Dataset for Idiomatic ReTyping—a large-scale public dataset of C code for training models to retype or rename decompiled code, consisting of nearly 1 million unique functions and 368 million code tokens.

- DIRTY—the DecompIler variable ReTYper—an open-source Transformer-based neural network model to recover syntactic and semantic types in decompiled variables. DIRTY uses the *data layout* of variables to improve retyping accuracy, and is able to *simultaneously retype and rename* variables in decompiled code.

---

[1] https://www.hex-rays.com/products/decompiler/

Example output from DIRTY is available online at https://dirtdirty.github.io/explorer.html.

2 Model Design

In this section, we describe our machine learning model and decisions that influenced its design, starting with some relevant background. Our model is a *neural network* with an *encoder-decoder* architecture.

2.1 The Encoder-Decoder Architecture

Our task consists of generating variable types (and names) as output given individual functions in decompiled code as input. This means that unlike a traditional classification problem with a fixed number of classes, both our input and output are sequences of variable length: input functions (e.g., fed into the network as a sequence of tokens) can have arbitrarily many variables, each requiring a type (and name) prediction.

Therefore, we adopt an encoder-decoder architecture [7], commonly used for sequence-to-sequence transformations, as opposed to the traditional feed-forward neural network architecture used in classification problems with a fixed-length input vector and prediction target. More specifically, the *encoder* takes the variable-length input and encodes it as a fixed-length vector. Then, this fixed-length encoding is passed to the *decoder*, which converts the fixed-length vector into a variable-length output sequence. This architecture, further enhanced through the *attention* mechanism [3], has been shown to be effective in many tasks such as machine translation, text summarization [36], and image captioning [53].

2.2 Transformers

There are several ways to implement an encoder-decoder. Until recently, the standard implementation used a particular type of recurrent neural network (RNN) with specialized neurons called long short-term memory units; these neurons and networks constructed from them are commonly referred to as LSTMs [24]. More recently, Transformer-based models [4, 15, 42, 57], building on the original Transformer architecture [50], have been shown to outperform LSTMs and are considered to be the state-of-the-art for a wide range of natural language processing tasks, including machine translation [4], question answering and abstractive summarization [10, 32], and dialog systems [1]. Transformer-based models have also been shown to outperform convolutional neural networks such as ResNet [19] on image recognition tasks [11].

Transformers have several properties that make them a particularly good fit for our type prediction task. First, they capture long-range dependencies, which commonly occur in program code, more effectively than RNNs. For example, a variable declared at the beginning of a function may not be used until much later; an ideal model captures information about all uses of a variable. Second, transformers can perform more computations in parallel on typical GPUs than LSTMs. As a result, training is faster, and a Transformer can train on more data in the same amount of time. In our case, this enables us to train on our large-scale, real-world dataset, which consists of 368 *million* decompiled code tokens.

Although there have been a number of advances in neural machine translation since the original Transformer model [50], most recent advances focus on improvements on other factors, such as training data and objectives [4, 10, 32, 42], dealing with longer sequences [57], efficiency [6], and scaling [15], rather than changing the fundamental architecture. Moreover, most of these improvements are tailored for the natural language domain, making them less generalizable than the original model and inapplicable to our task. Instead, we keep our model simple, which allows different, better architectures or implementations to be used out-of-the-box in the future. For example, the recent Vision Transformer (ViT) [11], which also intentionally follows the original Transformer architecture "as closely as possible" when adapting Transformers to computer vision tasks.

We omit the technical details of Transformers, including multi-headed self-attention, positional encoding, and the specifics of training as they are beyond the scope of this paper.

2.3 DIRTY's Architecture

In DIRTY, we cast the retyping problem as a transformation from a sequence of tokens representing the decompiled code to a sequence of types, one for each variable in the original source code. This section describes DIRTY's architecture in detail. Figure 3 shows an overview of the architecture.

Code Encoder. The encoder converts the sequence of code tokens of the decompiled function (lower-left of Figure 3), $x = (x_1, x_2, \ldots, x_n)$, into a sequence of representations,

$$\mathbf{H} = (\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_n}), \tag{1}$$

where each continuous vector $\mathbf{h_i} \in \mathbb{R}^{d\_model}$ is the *contextualized representation* for the i-th token x_i. During training, the encoder learns to encode the information in the decompiled function x relevant to solving the task into $\mathbf{H}$. For example, for a code token $x_i =$v1, useful information about v1 in the context of x (e.g., operations performed on v1) is automatically learned and stored in $\mathbf{h_i}$.

Specifically, we denote the encoding procedure as

$$\mathbf{H} = f_{en}(x; \theta_{en}), \tag{2}$$

where the input $x = (x_1, x_2, \ldots, x_n)$ is the code token sequence of the decompiled function and the output $\mathbf{H} = (\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_n})$ is the sequence of deep contextualized representations. f_{en} denotes the encoder, implemented with neural networks, and θ_{en} denotes its learnable parameters.

The ultimate goal of DIRTY is to make type predictions about each *variable* that appears in the decompiled function. However, the encoder produces hidden representations for
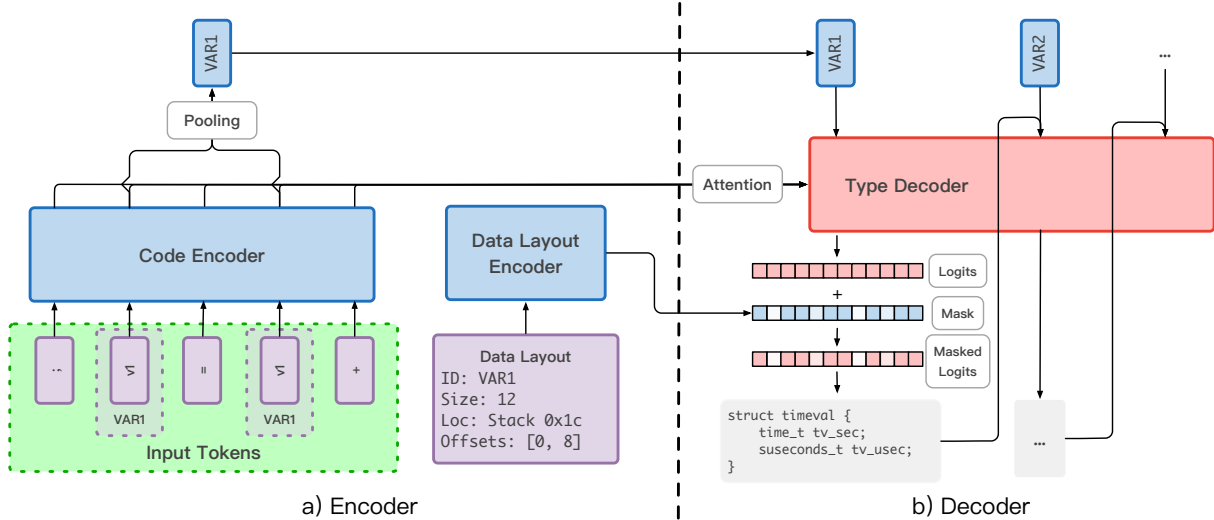
Figure 3: Overview of DIRTY's neural model architecture for predicting types. Decompiled code is sequentially fed into the Code Encoder. When the input of the code encoder corresponds to a specific variable (e.g., VAR1), it is pooled with other instances of the same variable to generate a single encoding for that variable. Each pooled encoding is then passed into the Type Decoder, which outputs a vector of the log-odds (logits) for predicted types. This vector is masked with a vector generated by the Data Layout encoder and the most probable type is chosen from the masked logits.

every *code token* (e.g., "v1", ":", "=", "v1", "+", "1" are all tokens). Because a variable can appear multiple times in the code tokens of a function, we need a way to summarize all appearances of a variable. We achieve this through *pooling*, where the representation for the t-th variable[2] is computed based on all of its appearances in the code tokens, A_t, using an average pooling operation [29]

$$\mathbf{v_t} = \text{AveragePool}_{x_i \in A_t} \mathbf{h_i}, \ t = 1, \dots, m \quad (3)$$

where m is the number of variables in the function. This solution removes the burden of gathering all information about a variable throughout the function into a single token representation from the model. The pooled representation for the first variable, VAR1, is shown in the upper-left of Figure 3.

Type Decoder. Given the encoding of the decompiled tokens, the decoder predicts the most probable (i.e., idiomatic) types for all variables in the function. The decoder takes the encoded representations of the code tokens ($\mathbf{H}$) and identifiers ($\mathbf{v_t}$) as input and predicts the original types $\hat{y} = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_m)$ for all m variables in the function. Unlike the encoder, the decoder predicts the output step-by-step using former predictions as input for later ones.[3]

At each time step t, the decoder tries to predict the type for the t-th variable as follows:

1. The decoder takes the code representations $\mathbf{H}$ and variable representation $\mathbf{v_t}$ from the encoder, and also previous predictions $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_{t-1}$ from itself, to compute a

hidden representation $\mathbf{z_t} \in \mathbb{R}^{d\_model}$

$$\mathbf{z_t} = f_{de}(\hat{y}_1, \hat{y}_2, \dots, \hat{y}_{t-1}, \mathbf{v_t}, \mathbf{H}; \theta_{de}) \quad (4)$$

where f_{de}, θ_{de} denotes the decoder and its parameters. The hidden representation $\mathbf{z_t}$ is then used for prediction.

2. The output layer of the decoder then uses its learnable weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ to transform the hidden representation $\mathbf{z_t}$ to the *logits* for prediction

$$\mathbf{s_t} = \mathbf{W}\mathbf{z_t} + \mathbf{b}, \quad (5)$$

where $\mathbf{s_t} \in \mathbb{R}^{|\mathcal{T}|}$, $\mathbf{W} \in \mathbb{R}^{|\mathcal{T}| \times d\_model}$, $\mathbf{b} \in \mathbb{R}^{|\mathcal{T}|}$, and $|\mathcal{T}|$ is the number of types in the type library. The logits $\mathbf{s_t}$ is the unnormalized probability predicted by the model, or the model's *scores* on all types

3. Finally, the softmax function computes a probability distribution over all possible types from $\mathbf{s_t}$

$$\Pr(\hat{y}_t | \hat{y}_1, \hat{y}_2, \dots, \hat{y}_{t-1}, x) = \text{softmax}\, \mathbf{s_t} \quad (6)$$

Note that the type library $\mathcal{T}$ is fixed, meaning DIRTY can only predict types that it has seen during training. We discuss this limitation, its implications, and potential mitigations in Section 5. However, DIRTY can recover structure types as well as normal types, as both are simply entries in $\mathcal{T}$.

The goal of the decoder is to find the *optimal* set of type predictions for all variables in a given function (i.e., the predictions with the highest combined probability): $\text{argmax}_{\hat{y}} \Pr(\hat{y}|x)$. This probability can be factorized as the product of probabilities at each step:

$$\Pr(\hat{y} \mid x) = \prod_{t=1}^{m} \Pr(\hat{y}_t \mid \hat{y}_1, \hat{y}_2, \dots \hat{y}_{t-1}, x). \quad (7)$$

---

[2]t is commonly used in RNN literature because it refers to a "timestep".
[3]This is known as an *autoregressive* model.

We've shown how to compute $\Pr\left(\hat{y}_t \mid \hat{y}_1, \hat{y}_2, \ldots \hat{y}_{t-1}, x\right)$ with the decoder, but finding the optimal $\hat{y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$ is not an easy task, because each variable can have $|\mathcal{T}|$ possible predictions, and each prediction affects subsequent predictions. The time complexity of exhaustive search is $O(|\mathcal{T}|^m)$. Therefore, finding the optimal prediction is often computationally infeasible for large functions. A simple approach is *greedy decoding*, selecting the most promising prediction at every step based on the previously selected predictions, i.e., taking the max $\hat{y}_t = \text{argmax}_{y_t} \Pr\left(\hat{y}_t = i \mid \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}, x\right)$. Greedy decoding is fast, but it often finds subpar predictions.

In DIRTY, we use beam search [37], a compromise between greedy decoding and an exhaustive search. Rather than only taking the most promising prediction (greedy), beam search considers a configurable number of most promising predictions at each step. In practice, it is usually able to find good (but not optimal) predictions, but is significantly faster than an exhaustive search.

2.4 Data Layout Encoder

The model described so far only uses information encoded into the code tokens of the decompiled representation. But to actually *create* such an output, decompilers typically perform a battery of complex binary analyses. Some decompilers allow the user to programmatically access the interim results from some of these analyses. In particular, Hex-Rays provides information about the storage location (e.g., register or stack offset), size, nested data types (e.g., if the variable is a `struct`), and offsets of its members, if any (e.g., offsets in an array or of fields in a `struct`), for each variable in a function. Intuitively, this information can help DIRTY rule out bad predictions. For instance, a variable that is 4 bytes long could not be a `char` type because it would not fit.

One inefficient approach could use this information as a hard constraint on the decoder's predictions, i.e., a *mask* which sets the probability of any "incompatible" types to 0. However, this runs into a problem when the decompiler incorrectly reconstructs the data layout (see Figure 2). To mitigate this, DIRTY learns a *soft* mask, reducing probabilities without setting them to 0. For example, DIRTY can learn based on many observations that a decompiled `char[4]` should be typed as a `char[3]` 5% of the time and `char[4]` 80%, and adjust the predictions of the type decoder accordingly. This allows the model to learn how best to incorporate the data layout information from the decompiler, including when the information is likely to be incorrect. Figure 3 illustrates where the data layout encoder fits into the overall architecture.

To implement the soft mask encoder, we jointly train another Transformer encoder to use data layout information to generate a mask. Figure 4 shows the internals of the data layout encoder. First, variable data layout is passed to the encoder. There are three parts to the data layout for a specific variable, each of which is simply converted to a token:
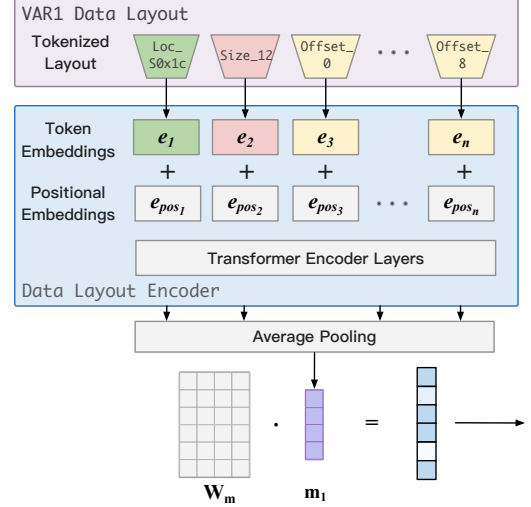

Figure 4: The Data Layout Encoder of DIRTY. The data layout for a specific variable, including its location, size, and offsets of its members is passed into the layout encoder (top), generating a mask (bottom).

Location: A variable can be located either in registers (tokenized as [Loc_<Register Name>]) or on the stack ([Loc_S<Offset>]). E.g., a variable stored 28 bytes below the stack pointer is tokenized as [Loc_S0x1c].

Size: Measured in bytes and tokenized as [Size_<Size>].

Internal Offsets: The offsets of members of the type (either array elements or struct fields), in bytes. E.g., the type `int[2]` would have the offsets $\{0, 4\}$, while a `struct` with two `char` fields would have the offsets $\{0, 1\}$. These are tokenized as a sequence of [Offset_<Offset>]. For consistency, we also use [Offset_0] for types without substructure (i.e., scalar types like `int`).

The tokenized data layout information is concatenated into a sequence denoted M_t and then encoded as

$$\mathbf{m}_t = f_{layout}\left(M_t; \theta_{layout}\right), \tag{8}$$

where $\mathbf{m}_t$ is the hidden representation of data layout information. Inspired by Michel and Neubig [35], we adjust the output type distribution with data layout information. Formally, we modify Equation (5) to fuse the data layout representation $\mathbf{m}_t$ into the final output layer:

$$\tilde{\mathbf{s}}_t = \mathbf{s}_t + \mathbf{W}_m\mathbf{m}_t = \mathbf{W}\mathbf{z}_t + \mathbf{W}_m\mathbf{m}_t + \mathbf{b}, \tag{9}$$

where $\mathbf{s}_t$ is the logits predicted by the Type Decoder, $\mathbf{W}_m\mathbf{m}_t$ is the "soft mask" produced by the Data Layout Encoder, and $\tilde{\mathbf{s}}_t$ is the new masked logits. $\mathbf{W}_m \in \mathbb{R}^{|\mathcal{T}| \times d\_model}$ denotes the learnable weight matrix in the final layer of Data Layout Encoder for transforming the data layout representation $\mathbf{m}_t \in \mathbb{R}^{d\_model}$ to the mask $\in \mathbb{R}^{|\mathcal{T}|}$. This implements a soft filter for type prediction using data layout information.
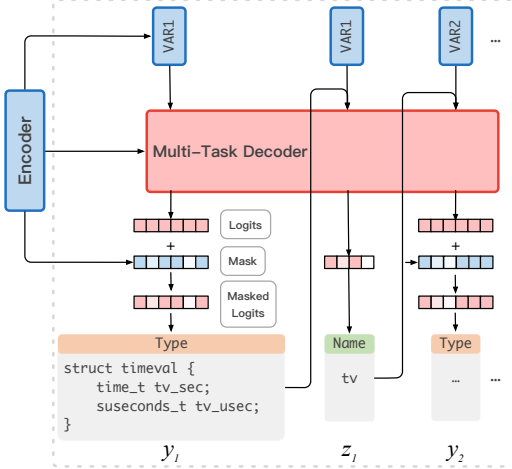
Figure 5: The multi-task decoder for DIRTY, which predicts both variable types and names. The encoder architecture is the same as in Figure 3. Each variable is passed to the decoder twice, the first time a type is predicted (y_i), and the second time a name is predicted (z_i). Note that the data layout encoding of a variable is only used to weight type predictions.

2.5 Multi-Task

Many variable names are indicative of their underlying type. For example, `i` and `j` are often used to represent integers, `s` and `str` are often used to represent strings, etc. Thus, intuitively, there is some connection between a variable's *name* and its *type*. Indeed, measuring the adjusted mutual information [51] between variable names and types in our dataset, we find a moderate association (0.41 on the scale $[0, 1]$). Since variable *names* can often be recovered from decompiled code using neural models [29], this may help us learn to predict variable types as well (and vice versa).

To test this, we extend DIRTY to also predict names with a single, integrated multi-task model. That is, we also predict a variable name for each variable in the function

$$\hat{z} = (\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_m) \tag{10}$$

where $\hat{z}_t$ denotes the predicted name for the t-th variable.

DIRTY's decoder outputs are interleaved to predict names and types in parallel (Figure 5). The first time the decoder is invoked on the t-th variable, it outputs the predicted *type* ($\hat{y}_t$) and the second time it outputs a predicted *name* ($\hat{z}_t$).

The training and prediction procedures remain almost the same, with two notable exceptions. First, to improve performance, the Data Layout encoder is *not* activated when the decoder is predicting a variable's name. This is unnecessary because name prediction depends on the predicted type, which has already incorporated the data layout information. Preliminary experiments confirmed no improvement in accuracy when using the Data Layout encoder for name prediction.

Second, there are two ways to interleave the predictions of types and names: types first or names first. In theory, this does

not matter because they are equivalent if the learned model and the decoding algorithm are ideal. In practice, we chose to predict types first because we believe the type prediction task should be easier (since there is more information) and it better reflects how developers define variables.

3 Evaluation

We conducted experiments to evaluate DIRTY, answering the following research questions:

RQ1: How effective is DIRTY at idiomatic retyping?

RQ2: How well does DIRTY perform on other decompilation benchmarks compared to prior work?

RQ3: How does each component of DIRTY contribute to the retyping and renaming performance?

RQ4: How does compiler optimization affect DIRTY's prediction accuracy?

3.1 Experimental Setup

First, we introduce the DIRT dataset we used for training DIRTY, and experimental setup details. The detailed hyperparameters for our deep learning model and environment configuration are described Appendix A.

Dataset for Idiomatic ReTyping (DIRT). To create DIRT, we queried a 2017 version of the GHTORRENT[4] database, compiling a list of public GITHUB repositories predominantly written in C. We then cloned these repositories locally using an open-source tool, GHCC,[5] to automatically build them. GHCC identifies build instructions (e.g., Makefiles) in repositories, creates a Docker container with the requisite libraries, and attempts to build the project. We used GCC version 9.2.0. For most experiments, we explicitly disable optimizations using the `-O0` compiler flag. We also evaluated DIRTY at higher optimization levels in Section 3.5. This process resulted in 4,346,134 automatically compiled 64-bit x86 binaries. After compilation, we then decompiled each binary using Hex-Rays and filtered out any functions that did not have variables requiring renaming or retyping. Following DIRE [29], we compiled each binary again with debugging information to align decompiler-assigned variable names (e.g., `v1`) and developer-assigned variable names (e.g., `picture`) to form training examples.

Since DIRE was only concerned with renaming, its dataset did not include variables which did not correspond to a named variable in the original source code. Many such variables are actually caused by mistakes in the decompiler during type recovery, for instance decompiling a structure to multiple scalar variables instead. Since the goal of DIRT is to enable type recovery and fix such mistakes, we label these instances

---
[4] https://ghtorrent.org
[5] https://github.com/huzecong/ghcc

as `<Component>` to denote that they are *components* of a variable in the source code. This allows the model to combine them with other variables into an array or a struct.

The final DIRT dataset consists of 75,656 binaries randomly sampled from the full set of 4,346,134 binaries to yield a dataset that we could fully process based on the computational resources we had available. We split the dataset per-binary as opposed to per-function, which ensures that different functions from the same binary cannot be in both the test and training sets. The training dataset consists of 997,632 decompiled functions, and a total number of 48,888 different types. We also preprocess the decompiled code with byte-pair encoding (BPE) [45], a widely adopted technique in NLP tasks to represent rare words with limited vocabulary by tokenizing them into subword units. After this step, the DIRT dataset consists of 368 *million* decompiled code tokens, and an average of 220.3 tokens per function. Detailed statistics about the DIRT dataset and the train/valid/test split can be found in Table 11 in Appendix A.

Metrics. We evaluate DIRTY using two metrics:

Name Match: Following DIRE [29], we consider a variable name prediction correct if it exactly string matches the name assigned by the original developer. We compute the prediction accuracy as the average percentage of correct predictions across all functions in the test set.

Type Match: We consider a type prediction to be correct only if the predicted type fully matches the ground truth type, including data layout, and the type and name of any fields if applicable. We serialize types to strings and use string matching to determine type matching.

Note that both metrics are conservative. Predictions may still be meaningful, even if not identical to the original names. A human study evaluating the quality of predicted types and names is beyond the scope of the current paper.

Meaningful Subsets of the Test Data. We introduce several subsets of the DIRT test set to better interpret the results:

Function in training **vs** *Function not in training.*
Similarly to Lacomis et al. [29], *Function in training* consists of the functions in the test set that also appear in the training set, which are mainly library functions. Allowing this duplication simulates the realistic use case of analyzing a new binary that uses well-known libraries. We also separately measure the cases where the function is not known during training (i.e., *Function not in training*) to measure the model's generalizability.

Structure types. Only 1.8% of variables in DIRT have structure types. Because of this low percentage, examining overall accuracy may not reflect DIRTY's accuracy when predicting structure types, which we have found anecdotally to be more challenging. To mitigate this, we separately measure DIRTY's accuracy on structures in addition to its overall accuracy.

Method	Overall		In Train		Not in Train	
	All	Struct	All	Struct	All	Struct
F_{Size}	23.6	9.7	23.5	9.1	23.8	10.4
HR	37.9	28.7	39.0	28.7	36.4	28.7
DIRTY	**75.8**	**68.6**	**89.9**	**79.2**	**56.4**	**54.6**

Table 1: DIRTY has higher retyping accuracy than Frequency By Size (F_{Size}) and Hex-Rays (HR) on the DIRT dataset, both for all types (All) and on structural types alone (Struct).

3.2 RQ1: Overall Effectiveness

We evaluate DIRTY on the idiomatic retyping task and report its accuracy compared to several baselines.

Baselines. We measure our accuracy with respect to two baseline methods for predicting variable types:

Frequency by Size The number of bytes a variable occupies is the most basic information for a type. For this technique, we predict the most common developer-assigned type for a given size (as reported by the decompiler). E.g., `int` is the most common 4-byte type, and `__int64` is the most common 8-byte type; this baseline simply assigns these types to variables of the respective size.

Hex-Rays [22] During decompilation, Hex-Rays already predicts a type for each variable, so we can use these predictions as a baseline. However, Hex-Rays cannot predict developer-generated types without prior knowledge of them, e.g., Hex-Rays assigns `unsigned __int16` instead of the more common `uint16_t`, which puts it at an unfair disadvantage. For this baseline, we reassign the type chosen by Hex-Rays to the most common developer-chosen name associated with it (e.g., we replace every `unsigned __int16` with `uint16_t`.

Results. As shown in Table 1, DIRTY can correctly recover 75.8% of the original (developer-written) types from the decompiled code. In contrast, Hex-Rays, the highest scoring baseline, can only recover 37.9% of the original types.

As expected, DIRTY performs even better when it has seen a particular function before (In Train), generating the same type as the developer 89.9% of the time. This indicates that DIRTY works particularly well on common code such as libraries. Even when a function has never been seen (Not in Train), DIRTY predicts the correct type 56.4% of the time.

Table 1 also shows the performance of DIRTY on structure types alone. Correctly predicting structure types is more difficult than predicting scalar types, and all models show a drop in performance. Despite this drop, DIRTY still achieves 68.6% accuracy overall, and 54.6% accuracy on the *Function not in training* category. Frequency By Size struggles on structures with only 9.7% accuracy; this is expected since structures of a given size can have many possible types. Hex-Rays is slightly

more accurate at 28.7%, as the decompiler is able to analyze the layout of structures.

Table 2 shows several examples of retyping predictions from the *Function not in training* partition. These examples show that accuracy is not the full story; even when DIRTY is unable to predict the correct type, the differences are often minor (e.g., `unsigned int` v. `int`, and `const char *` v. `char *`). The bottom half of Table 2 shows prediction examples of structure types.[6] DIRTY is able to recover the actual structure much of the time. At other times, DIRTY also produces some semantically reasonable but syntactically unacceptable predictions, like `char[32]` for `class std::string`.

3.3 RQ2: Comparison with Prior Work

We further compare DIRTY with recent work on type recovery [58] and variable name recovery [29].

Type Recovery. While there is prior work on type recovery (see also Section 4), none of the existing approaches, TIE [31], Howard [47], Retypd [39], TypeMiner [34] and OSPREY [58], are publicly available. We are grateful to Zhang et al. [58], the authors of OSPREY, for kindly sharing their evaluation material so we could compare results.

OSPREY is a recently proposed probabilistic technique for variable and structure recovery that outperforms existing work including Howard [47], Angr [46], Hex-Rays [22] and Ghidra [58]. The OSPREY authors provided us with the GNU coreutils[7] executables they used in their evaluation, which were compiled with `-O0` to disable optimization. We ran DIRTY on these executables, but only evaluated on stack and heap variables, since OSPREY does not recover register variables. This benchmark consists of 101 binaries and 17,089 variables. We also define two subsets of the dataset:

Visited A subset of 13,020 variables that are covered by BDA [59], a binary abstract interpretation tool that OSPREY relies on. OSPREY is expected to perform better on these covered functions than uncovered functions, which we also report as *Non-Visited*.[8] However, DIRTY is not subject to this limitation.

Struct A subset of 3,061 variables related to structure types. Following OSPREY, we include structs allocated on the stack, pointers to structs on the heap, and arrays of structs. These variables do *not* have to be in the Visited subset.

Because DIRTY can predict up to 48,888 different types, each including the full syntactic and semantic information, we convert its predictions in a post-hoc manner to make it comparable with OSPREY.[9]

Table 3 compares the accuracies of both systems. On the overall coreutils benchmark, DIRTY slightly outperforms OSPREY (76.8% vs 71.6%). OSPREY outperforms DIRTY on the Visited subset, but as expected, performs worse on the Non-Visited functions. Meanwhile, DIRTY is more consistent on Visited and Non-Visited. When only looking at structure types, OSPREY outperforms DIRTY (26.6% vs 15.7%).

However, this comparison puts DIRTY at a disadvantage, since OSPREY was designed for this task of recovering syntactic types, while DIRTY was trained to recover variable and type/field names, and much of this information is thrown out for this evaluation. To address this, we trained a new model, DIRTY$_{Light}$, on DIRT, but tailored the training to OSPREY's simplified task. The accuracy of this model is also reported in Table 3. As expected, the DIRTY$_{Light}$ model outperforms the off-the-shelf DIRTY model, since it is trained specifically for this task. DIRTY$_{Light}$ greatly improves prediction accuracy on the Struct subset, and even outperforms OSPREY.

To further get a fine-grained comparison with OSPREY, we calculate accuracy on 101 coreutils binaries individually, and show the prediction accuracies of DIRTY and OSPREY with respect to the number of variables in the programs in Figure 6.

We observe that DIRTY is competitive compared with OSPREY. Interestingly, while the results on large binaries are close, DIRTY performs better on small binaries. This suggests our learning-based method trained on GitHub data might generalize better on rare patterns compared to empirical methods that might have been developed based on observations on a limited number of common and relatively larger programs.

In addition, DIRTY is also much faster and scalable. On average, OSPREY takes around 10 minutes to analyze one binary in coreutils, while it takes 75 seconds for DIRTY$_{Light}$ to finish inference on the whole coreutils benchmark.

Overall, we believe both methods are valuable. Since at this point DIRTY is using Hex-Rays recovered data layout as input to its Data Layout Encoder, we believe a promising future direction is to combine these two methods—using OSPREY's results as the input to DIRTY's, and the combined approach can potentially achieve even better results.

Name Recovery. The Decompiled Identifier Renaming Engine (DIRE) is a state-of-the-art neural approach for decompiled variable name recovery [29]. The DIRE model consists of both a lexical encoder and a structural encoder, utilizing both tokenized decompiled code and the reconstructed abstract syntax tree (AST). In contrast, DIRTY's simpler encoder only uses the tokenized decompiled code.

The DIRE authors provide a public dataset for decompiled variable renaming compiled with `-O0`. To compare with DIRE, we train DIRTY on the DIRE dataset and also train DIRE on the DIRT dataset. Since DIRE is focused on variable renaming, and there is no type information collected in their

---

[6]We omit the full predicted contents of structs here for conciseness.

[7]https://www.gnu.org/software/coreutils/

[8]A majority of uncovered functions are unreachable from the entry point of the binary, and others are indirect call targets which BDA fails to analyze.

[9]Specifically, we discard type names and field names. For example, `bool` and `char` are both converted to `Primitive_1`, which stands for a primitive type occupying 1 byte of memory, `const char *` and `char *` are converted

to `Pointer<Primitive_1>`, and `struct ImVec2 {float x; float y ;}` converted to `Struct<Primitive_4, Primitive_4>`.

int		char *		class std::string	
`int`	88.8%	`char *`	60.3%	`class std::string`	47.5%
`unsigned int`	4.3%	`const char *`	11.4%	`char[32]`	24.2%
`<Component>`	2.7%	`<Component>`	4.4%	`char[47]`	14.6%
`uint32_t`	0.8%	`__int64`	4.1%	`class std::__cxx11::basic_string`	6.1%
`u_int32_t`	0.3%	`size_t`	1.8%	`char[40]`	3.5%

Table 2: Example variable types from the *Function not in training* testing partition. The top rows are the developer-assigned types and the columns show DIRTY's top-5 most frequent predictions. `<Component>` represents a prediction that the variable in the decompiled code does not correspond to a variable in the source code (e.g., because it corresponds to a member of a struct).

	Coreutils			
Model	All	Visited	Non-Visited	Struct
OSPREY	71.6	**83.8**	32.4	26.6
DIRTY	76.8	79.1	69.6	15.7
DIRTY$_{Light}$	**80.1**	80.1	**80.1**	**27.7**

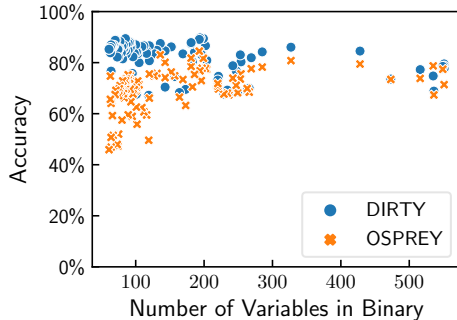Table 3: Accuracy comparison on the Coreutils benchmark.

Figure 6: Accuracy of DIRTY and OSPREY on 101 individual programs in the coreutils benchmark with different number of variables. The two methods are competitive on large binaries, while DIRTY performs much better on small binaries.

	DIRE Dataset			DIRT Dataset		
Model	All	FIT	FNIT	All	FIT	FNIT
DIRE	72.8	84.1	33.5	57.5	75.6	31.8
DIRTY	**81.4**	**92.6**	**42.8**	**66.4**	**87.1**	**36.9**

Table 4: Accuracy comparison of DIRE and DIRTY on the DIRE and DIRT datasets. Accuracy is reported overall (All), when functions are in the training set (FIT), and when functions are not in the training set (FNIT).

	Accuracy	
Model	Overall	Struct
DIRTY$_S$	74.5	65.4
DIRTY	**75.8**	**68.6**

Table 5: Effect of model size. The accuracy columns show the overall accuracy and the accuracy on struct types.

dataset, we cannot use the Data Layout Encoder for these experiments. Instead, we only use our Code Encoder and Renaming Decoder. We report the accuracy of both systems in Table 4. DIRTY significantly outperforms DIRE in terms of overall accuracy on both the DIRE dataset (81.4% vs. 72.8%), and on the DIRT dataset (66.4% vs. 57.5%). DIRTY also generalizes better than DIRE: when functions are not in the training set, DIRTY outperforms DIRE on both the DIRE (42.8% vs. 33.5%) and the DIRT datasets (36.9% vs. 31.8%).

DIRTY outperforms DIRE in spite of the fact that it only leverages the decompiled code, whereas DIRE leverages both the decompiled code *and* the reconstructed AST from Hex-Rays. Since the primary difference between DIRTY without type prediction and DIRE is that it uses Transformer as its encoder and decoder network, we attribute this improvement to the power of Transformers, which allow modeling interactions between any pair of tokens, unrestricted to a sequential or tree structure as in DIRE.

Also notable is how DIRTY trains faster than DIRE. We found that DIRTY surpassed DIRE in accuracy after training for 30 GPU hours, compared to the 200 GPU hours required to train DIRE on the full DIRT dataset, which we again attribute to the efficiency of the Transformer architecture.

3.4 RQ3: Ablation Study

To understand how each component of DIRTY contributes to its overall performance, we perform an ablation study.

Model Size. Transformers have the merit of scaling easily to larger representational power by stacking more layers, increasing the number of hidden units and attention heads per layer [10, 50]. We compare DIRTY to a modified, smaller
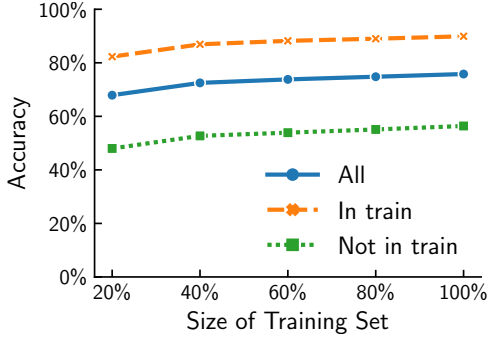
Figure 7: Effect of training data size. With 100% of the data, the accuracies of *All*, *In train*, and *Not in train* are 75.8%, 89.9%, and 56.4% respectively. With 20%, these drop to 67.9%, 82.3%, and 48.0% respectively.

Model	Overall	In train	Not in train
DIRTY$_{NDL}$	72.2	88.4	49.9
DIRTY	**75.8**	**89.9**	**56.4**

Table 6: Effect of the Data Layout encoder on the accuracy of DIRTY. Accuracy is reported for the model with (DIRTY) and without (DIRTY$_{NDL}$) the encoder.

version DIRTY$_S$. DIRTY contains 167M parameters, while DIRTY$_S$ only 40M. Table 10 contains details of the hyperparameter differences between the two models.

Table 5 shows overall DIRTY is 75.8% accurate vs. 74.5% for DIRTY$_S$'s. This indicates increasing the model size has a positive effect on retyping performance. The gain from increased model capacity is notably larger when comparing performance on structures. This improvement suggests that complex types are more challenging and require a model with larger representational capacity. We are not able to train a larger model due to limits on computation power.

Dataset Size. We examine the impact of training data size on prediction accuracy. As a data-driven approach, DIRTY relies on a large-scale code dataset; studying the impact of data size gives us insight into the amount of data to collect. We trained DIRTY on 20%, 40%, 60%, 80% and 100% portion of the full training partitionand report the results in Figure 7.

Figure 7 shows the change in accuracy with respect to the percentage of training data. Increasing the size of training data has a significant positive effect on the accuracy. Between 20% and 100% of the full size the accuracy increases from 67.9% to 75.8%, a relative gain of 11.6%.

Notably, accuracy on *Function not in training* has a relative gain of 17.5% much larger than on the *Function in training* partition. This is likely because the *Function in training* partition contains common library functions shared by programs both in the training and test set, and even a smaller dataset will have programs that use these functions. In contrast, the *Function not in training* part is open-ended and diverse.

It is also worth noting that the accuracy drops sharply when the training set size is decreased from 40% to 20%, justifying the necessity for using a large-scale dataset.

Data Layout Encoder. We explore the impact of the Data Layout encoder on DIRTY's performance. We experiment with a new model with no Data Layout encoder, DIRTY$_{NDL}$.

Table 6 shows the accuracy results overall and on the *Func-*

tion in training and *Function not in training* partitions. The inclusion of the Data Layout encoder improves overall accuracy from 72.2% to 75.8%, indicating that the Data Layout encoder is effective. The results are even more interesting when the results are broken into the two partitions. The relative gain on the *Function in not training* partition is 13% (49.9% to 56.4%), compared to 1.7% on the *Function in training* partition (88.8% to 89.9%). This suggests the Data Layout encoder greatly improves DIRTY's generalization ability.

Table 7 compares example predictions from DIRTY and DIRTY$_{NDL}$ on the same types from the *Function not in training* partition. For the `__int64` example, the type predictions from DIRTY mostly have the correct size of 8 bytes. DIRTY$_{NDL}$, however, often incorrectly predicts `int` and `unsigned int`. This is understandable because in situations where the value doesn't exceed the 32-bit integer, `__int64` can be safely interchanged with `int`, these situations can be identified in some decompiled code. However, apart from the correctness of the retyped program, accuracy to the original binary, (i.e., allocating 8 bytes instead of 4), is also important. DIRTY achieves this better than DIRTY$_{NDL}$.

In the second example, the `struct __m128d` type occupies 16 bytes, and has two members at offset 0 and 8. DIRTY$_{NDL}$ mainly mistakes this structure as a `double`, which might make sense semantically but is unacceptable syntactically. With the Data Layout encoder, DIRTY effectively reduces these errors. This demonstrates this component achieves the soft masking effect on type prediction as intended in Section 2.4.

Multi-Task Decoder. In this section we study the effectiveness of the Multi-Task decoder when compared to decoders designed for only retyping or only renaming. Inspecting the accuracy numbers reported in Table 8, the Multi-Task decoder has similar, but slightly lower overall accuracy on both tasks as the two specialized models (-0.8% for retyping and -1.3% for renaming). One possible reason is that the Multi-Task model has twice the length of decoding lengths than a specialized model, which makes greedy decoding harder.

Despite the small decrease in performance, the unified model has advantages. These are illustrated in the ✓Name and ✓Type columns of Table 8. ✓Name and ✓Type stand for the subsets of the full dataset where the Multi-Task decoder makes correct renaming predictions and correct retyping predictions, and we evaluate the retyping and renaming performance on

DIRTY				DIRTY$_{NDL}$			
__int64		struct __m128d		__int64		struct __m128d	
__int64	74.3%	struct __m128d	78.7%	__int64	67.0%	double	33.1%
<Component>	5.7%	<Component>	15.4%	int	6.3%	<Component>	27.2%
void *	1.7%	void	2.9%	<Component>	6.0%	__int64	10.3%
char *	1.7%	__int128	2.2%	unsigned int	1.5%	struct __m128d	5.9%
const char *	1.6%	double	0.7%	char *	1.2%	int	3.7%

Table 7: Comparative examples from DIRTY with and without Data Layout encoder from the *Function not in training* partition. Predictions inside a gray box have a different data layout than the ground truth type. DIRTY effectively suppresses these, which helps guide the model to a correct prediction. The structure's full type is `struct __m128d {double[2] m128d_f64;}`.

	Retyping		Renaming	
Model	Overall	✓Name	Overall	✓Type
Retyping	**75.8**	90.6	-	-
Renaming	-	-	**66.4**	82.6
Multi-Task	74.9	**92.3**	65.1	**84.6**

Table 8: Performance comparison of the Retyping-only, Renaming-only, and Multi-Task decoders. Overall performance is shown, in addition to performance on retyping when the name is correct (✓Name) and performance on renaming when the type is correct (✓Type).

	GNU coreutils			
Model	-O0	-O1	-O2	-O3
DIRTY	48.20	46.01	46.04	46.00

Table 9: Accuracy comparison of DIRTY on the GNU coreutils benchmark compiled with -O0, -O1, -O2, and -O3 optimization levels.

them, respectively.[10] The Multi-Task decoder outperforms the specialized models by 1.9% and 2.4% relatively on these metrics, in spite of the longer decoding length. This means the type and name predictions from the Multi-Task decoder are more consistent with each other than from specialized models. In other words, making a correct prediction on one task increases the probability of success on the other task.

In practice, this offers additional flexibility and opens the opportunity for more applications. For example, consider a cooperative setting where a human decompilation expert uses DIRTY as an analysis tool. The human expert may be unsatisfied with the model's top prediction and want to switch to another one in the top-k candidates list. With a Multi-Task decoder, the model adjusts the name prediction for that variable, which is impossible with the specialized decoders.

---

[10]The probability of success on the other task also increases by chance, because success on one task implies it is easier than average. We have eliminated this influence by, e.g., comparing 92.3 to 90.6, instead of 74.9.

3.5 RQ4: Compiler Optimization Levels

We study the impact of compiler optimizations on DIRTY's accuracy. In keeping with the spirit of the OSPREY evaluation on coreutils compiled with -O3, we choose coreutils as our evaluation dataset. However, since we did not have access to the original dataset used by OSPREY except -O0, we recompiled GNU coreutils 3.2 ourselves using optimization levels -O0, -O1, -O2, and -O3. Table 9 shows how accurately DIRTY is able to recover the full type (including type and field names) informaition at each optimization level. As expected, DIRTY does best at -O0, since DIRTY is trained on -O0 code and we believe -O0 code to be simpler. Going from -O0 to -O1, DIRTY's accuracy drops from 48.2% to 46.0%. However, there is little difference in performance between -O1, -O2, and -O3. This suggests that DIRTY does slightly better on the optimization level of code it was trained on, but that the effect of optimizations is small. We believe this is because Hex-Rays recognizes and will "undo" some optimizations so that the decompiled code will be very similar. For example, unoptimized code will often reference stack variables using a frame pointer, but optimized code will reference such variables using the stack pointer, or even maintain them in a register. But both implementations will look similar in the decompiled code, since the mechanism used to reference the variable is not important at the C level. Since DIRTY operates on the decompiled code, the decompiler effectively insulates DIRTY from these optimizations.

3.6 Illustration

To gain more qualitative insights into DIRTY's predictions, consider the example in Figure 8. The code shown is the Hex-Rays output, cleaned for presentation. Here, we would like to rename and retype the arguments a1, a2, and a3, in addition to the variable v1. The table in Figure 8 shows the developer's chosen types and names together with DIRTY's suggestions. DIRTY suggests the same types and names as the developer for a3 and v1, and the same type but a different name for a2. Although the names disagree for a2, we note that pic is an abbreviation for picture, so the disagreement is minor. We

```c
int find_unused_picture(int a1, int a2, int a3) {
    int i, j, v1;
    if (a3) {
        for (i = <Num>;; ++i) {
        if (i > <Num>)
            goto LABEL_13;
        if (!*(*(<Num> * i + a2) + <Num>))
            break;
        }
        v1 = i;
    } else {
        for (j = <Num>;; ++j) {
        if (j > <Num>) {
        LABEL_13:
            av_log(a1, <Num>, <Str>);
            abort();
        }
        if (pic_is_unused(<Num> * j + a2))
            break;
        }
        v1 = j;
    }
    return v1;
}
```

ID	Developer	DIRTY
a1	AVCodecContext_0 *avctx	MpegEncContext_0 *s
a2	Picture_0 *picture	Picture_0 *pic
a3	int shared	int shared
v1	int result	int result

Figure 8: Simplified Hex-Rays output. `<Num>` and `<Str>` are placeholder tokens for constant numbers and strings respectively. The table summarizes the original developer names and types along with the names and types predicted by DIRTY.

also observe that `Picture_0 *`, the type of `a2` itself carries a lot of semantic information; even if DIRTY was unable to suggest a meaningful name, `Picture_0 *a2` is still helpful for reverse engineering.

The developer and DIRTY disagree on both the name and the type of `a1`. In this case, the name chosen by DIRTY (`s`) would probably not be considered a very useful improvement over `a1`. However, the type suggested by DIRTY (`MpegEncContext_0 *`) could still be quite useful to a reverse engineer, even if it is inaccurate. It suggests that this argument is a "context", and hints that this function is used for video.

4   Related Work

Other projects related to type recovery for decompilation are REWARDS [33], TIE [31], Retypd [39], and OSPREY [58]. Unlike our approach, they use program analyses to compute constraints on types. Additionally, they are either limited to only predicting the syntactic type (TIE, Retypd, OSPREY), or only predicting one of a small set of hand-written types (150 for REWARDS). In comparison, DIRTY automatically generates a database of types by observing real-world code.

Other projects use machine learning to predict types, but tar-

get different languages than DIRTY. DEEPTYPER [20] learns type inference for JavaScript and OPTTYPER [40], LAMB-DANET [52], R-GNN_{NS-CTX} [56] target TypeScript. Training a machine learning algorithm for the task of typing dynamic languages like these is a slightly easier task: generating a parallel corpus is simple, since the types can simply be removed without changing the semantics. The DIRT dataset is fundamentally different: including debug information often changes the layout of the code as the decompiler adds structures and syntax for accessing them.

To the best of our knowledge, the most directly-related work to DIRTY is TypeMiner [34]. TypeMiner is a pioneering work, providing the proof-of-concept for recovering types from C binaries. However, they use much simpler machine learning algorithms and their dataset only consists of 23,482 variables and 17 primitive types. Escalada et al. [14] has provided similar insights. They adopt simple classification algorithms to predict function return types in C, but they only consider from only 10 different (syntactic) types and their dataset is limited to 2,339 functions from real programs and 18,000 synthetic functions.

Two other projects targeting the improvement of decompiler output using neural models are DIRE [29], which predicts variable names, DIRECT [38], which extends DIRE using transformer-based models, and Nero [8], which generates procedure names. Other approaches work directly on assembly [16, 26, 27], and learn code structure generation instead of aiming to recover developer-specified variable types or names. Similarly, DEBIN [18] and CATI [5] use machine learning to respectively predict debug information and types directly from stripped binaries without a decompiler.

5   Discussion

In this paper we presented DIRTY, a novel deep learning-based technique for predicting variable types and names in decompiled code. Still, DIRTY is limited in several ways that provide key opportunities for future improvements.

Alternative Decompilers to Hex-Rays. We implement DIRTY on top of the Hex-Rays decompiler because of its positive reputation and the programmatic access it affords to decompiler internals. However, DIRTY is not fundamentally specific to Hex-Rays, and the technique should conceptually work with any decompiler that names variables using DWARF debug symbols. Note that, due to its recent popularity and promise, we attempted to evaluate our techniques using the newer, open-source Ghidra decompiler. Unfortunately, it is currently infeasible, because Ghidra routinely failed to accurately name stack variables based on DWARF. This appears to be a combination of specific issues[11] and the general design of the decompiler. Ghidra's decompiler consists of many *passes* which modify and augment the current decompilation.

---

[11] https://github.com/NationalSecurityAgency/ghidra/issues/2322

Some of these passes combine variables, but in doing so may combine a DWARF-named variable with others. Since the combined variable no longer corresponds directly with the DWARF variable information, Ghidra discards the name. We are optimistic, however, that when the above-mentioned issues are addressed, Ghidra may again be a reasonable target for our approach.

Generalizing to Unseen Types. A limitation of DIRTY's current decoder is that it can only predict types seen during training. Fortunately, there appears, empirically, to be sufficient redundancy across large corpora that DIRTY is still frequently able to successfully recover structural types. This lends credence to the hypothesis that code is *natural*, an observation that has been explored in several domains [9, 23]. It moreover appears that data layout is of particular importance here: layout information recovered from the decompiler impose key constraints on the overall prediction problem. Indeed, our results in Section 3.4 corroborate the intuition that the Data Layout Encoder is especially important for succeeding on previously unseen code.

We envision meaningful future opportunities to more directly expand DIRTY's capabilities to predict unseen structures. This problem is analogous machine translation models that must deal with rare or compound words (e.g., xenophobia) that are not present in their dictionary. Byte Pair Encoding [45] (BPE) is the most frequently used technique to tackle this problem in the natural language domain. It automatically splits words into multiple tokens that are present in the dictionary (e.g., xeno and ##phobia). (The ## indicates the word is still part of the current word, instead of a new word next to it.) This technique greatly increases the number of words a model can handle despite a limited dictionary size, and enables the composition of new words that were not seen during training. This suggests that we can similarly extend DIRTY's decoder to predict previously unseen types by decomposing structure types into multiple pieces with BPE. For example, a structure type `struct timeval {time_t tv_sec; suseconds_t tv_usec;}` is split into four separate tokens, which are 1) `struct timeval`, 2) `time_t tv_vec;`, 3) `suseconds_t tv_usec;`, and 4) `<end_of_struct>`.

However, unfortunately, our preliminary experiments suggested that this hurts overall prediction accuracy. It also significantly slows down prediction, since it drastically increases the number of decoding steps. It moreover requires finer-grained accuracy metrics, like tree distance, to allow us to measure and credit partially correct predictions. Based on these observations, we believe unseen structure types should be handled specially with a tailored model, a problem we leave to future work.

Supporting Non-C Languages. A benefit of decompiling to C is that as a relatively low-level language, it can express the behavior of executables beyond those written in C. Although we designed DIRTY to be used with C programs and types, DIRTY can run on non-C programs, and will try to identify the C type that best captures the way in which that variable is being used. Thus, DIRTY provides value to analysts seeking to understand non-C programs, similar to how C decompilers such as Hex-Rays help analysts to understand C++ programs.

However, many compiled programming languages have type systems far richer than C's, and expressing these types in terms of C types may be confusing. For example, in C++, virtual functions are often implemented by reading an address out of a *virtual function table* [13, 44]. Although techniques like DIRTY can recognize such tables as structs or arrays of code pointers, it does not reveal the connection to the higher-level C++ behavior of virtual functions.

Extending DIRTY to support higher-level languages such as C++ is an interesting open problem. To some degree, as long as the decompiler is able to import the higher-level type information from debug symbols into the decompiler output, it should be possible to train DIRTY to recognize non-C types. For instance, 6% of the programs in DIRT are written in C++, and our evaluation measures DIRTY's ability to predict common C++ class types such as `std::string`. But recovering higher level properties of these types, especially for those never seen during training, is a challenging problem and is likely to require language-specific adaptations [13, 44].

Limited Input Length. As common with Transformers, we truncate the decompiled function if the length n exceeds some upper limit *max_seq_length*, which makes training more efficient. In our experiments we set *max_seq_length* to 512 for two reasons. First, 512 is the default value for *max_seq_length* in many Transformer models [10, 50]. Second, in DIRT, the average number of tokens in a function is 220.3, and only 8.8% of the functions have more than 512 tokens, i.e., we exclude relatively few of the possible inputs encountered in the wild. Still, if enough computational resources are available, we recommend using efficient Transformer implementations such as Big Bird [57] instead. These can deal with much larger *max_seq_length* and can be used out-of-the-box to replace our implementation.

6 Conclusion

The decompiler is an important tool used for reverse engineering. While decompilers attempt to reverse compilation by transforming binaries into high-level languages, generating the same code originally written by the developer is impossible. Many of the useful abstractions provided by high-level languages such as loops, typed variables, and comments, are irreversibly destroyed by compilation. Decompilers are able to deterministically reconstruct some structural properties of code, but comments, variable names, and custom variable types are technically impossible to recover.

In this paper we address the problem of assigning decompiled variables meaningful names and types by statistically

modeling how developers write code. We present DIRTY (DecompIled variable ReTYper), a novel technique for improving the quality of decompiler output that automatically generates meaningful variable names and types. Empirical evaluation of DIRTY on a novel dataset of C code mined from GitHub shows that DIRTY outperforms prior work approaches by a sizable margin, recovering the original names written by developers 66.4% of the time and the original types 75.8% of the time.

Acknowledgments

References

[1] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020.

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

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, ICLR, 2015.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[5] Ligeng Chen, Zhongling He, and Bing Mao. CATI: Context-assisted type inference from stripped binaries. In *International Conference on Dependable Systems and Networks*, DSN, 2020.

[6] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[7] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2014.

[8] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[9] Premkumar Devanbu. New initiative: The naturalness of software. In *International Conference on Software Engineering*, ICSE, pages 543–546, 2015.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL-HLT, pages 4171–4186, 2019.

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

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

[13] Rukayat Ayomide Erinfolami and Aravind Prakash. Devil is virtual: Reversing virtual inheritance in C++ binaries. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, pages 133–148, 2020.

[14] Javier Escalada, Ted Scully, and Francisco Ortin. Improving type information inferred by decompilers with supervised machine learning. *arXiv preprint arXiv:2101.08116*, 2021.

[15] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[16] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Conference on Neural Information Processing Systems*, NeurIPS, 2019.

[17] Edward M. Gellenbeck and Curtis R. Cook. An investigation of procedure and variable names as beacons during program comprehension. Technical report, Oregon State University, 1991.

[18] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. DEBIN: Predicting debug information in stripped binaries. In *Conference on Computer and Communications Security*, CCS, 2018.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 770–778, 2016.

[20] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2018.

[21] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (GELUs). *arXiv preprint arXiv:1606.08415*, 2016.

[22] Hex-Rays. The hex-rays decompiler, 2019. URL https://www.hex-rays.com/products/decompiler/.

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

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

[25] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *International Conference on Program Comprehension*, ICPC, pages 20–30, May 2018.

[26] Deborah S. Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *International Conference on Software Analysis, Evolution and Reengineering*, SANER, pages 346–356, 2018.

[27] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[28] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, ICLR, 2015.

[29] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *International Conference on Automated Software Engineering*, ASE, 2019.

[30] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? A study of identifiers. In *International Conference on Program Comprehension*, ICPC, pages 3–12, 2006.

[31] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium*, NDSS, 2011.

[32] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Annual Meeting of the Association for Computational Linguistics*, ACL, pages 7871–7880, 2020.

[33] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *CERIAS Annual Security Symposium*, CERIAS, 2010.

[34] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. TypeMiner: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA, 2019.

[35] Paul Michel and Graham Neubig. Extreme adaptation for personalized neural machine translation. In *Annual Meeting of the Association for Computational Linguistics (Short Papers)*, ACL, pages 312–318, 2018.

[36] Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, and Bing Xiang. Abstractive text summarization using sequence-to-sequence RNNs and beyond. In *SIGNLL Conference on Computational Natural Language Learning*, CoNLL, pages 280–290, 2016.

[37] Hermann Ney, Dieter Mergel, Andreas Noll, and Annedore Paeseler. A data-driven organization of the dynamic programming beam search for continuous speech recognition. In *International Conference on Acoustics, Speech, and Signal Processing*, ICASSP, 1987.

[38] Vikram Nitkin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. DIRECT: A transformer-based model for decompiled identifier renaming. In *Workshop on Natural Language Processing for Programming*, 2021.

[39] Matthew Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Conference on Programming Language Design and Implementation*, PLDI, pages 27–41, 2016.

[40] Irene Vlassi Pandi, Earl T Barr, Andrew D Gordon, and Charles Sutton. OptTyper: Probabilistic type inference by optimising logical and natural constraints. *arXiv preprint arXiv:2004.00348*, 2020.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems*, NeurIPS, pages 8024–8035. 2019.

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67, 2020.

[43] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research*, BAR, 2018.

[44] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover C++ classes and methods from compiled executables. In *Conference on Computer and Communications Security*, CCS, 2018.

[45] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The art of war: Offensive techniques in binary analysis. In *Symposium on Security and Privacy*, SP, pages 138–157, 2016.

[47] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium*, NDSS, 2011.

[48] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. Reconstruction of composite types for decompilation. In *Working Conference on Source Code Analysis and Manipulation*, SCAM, pages 179–188, 2010.

[49] Michael James van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems*, NeurIPS, pages 6000–6010, 2017.

[51] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research*, 11:2837–2854, 2010.

[52] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*, ICLR, 2020.

[53] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, ICML, pages 2048–2057, 2015.

[54] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Network and Distributed System Security Symposium*, NDSS, 2015.

[55] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Symposium on Security and Privacy*, SP, pages 158–177, 2016.

[56] Fangke Ye, Jisheng Zhao, and Vivek Sarkar. Advanced graph-based deep learning for probabilistic type inference. *arXiv preprint arXiv:2009.05949*, 2020.

[57] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big Bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*, 2020.

[58] Z. Zhang, Y. Ye, W. You, G. Tao, W. Lee, Y. Kwon, Y. Aafer, and X. Zhang. OSPREY: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *Symposium on Security and Privacy*, SP, pages 872–891, 2021.

[59] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. BDA: Practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–31, 2019.

A   Experimental Setup

Hyperparameter Configurations   Our detailed hyperparameters are shown in Table 10. We use a six-layer Transformer Encoder for the code encoder, a three-layer Transformer Encoder for the data layout encoder, and a six-layer Transformer Decoder for the type decoder. We set the number of attention heads to 8. Input embedding dimensions and hidden sizes d_{model} are set to 512 for the code encoder, and 256 for the data layout encoder. Following prior work, we empirically set the size of the inner-layer of the position-wise feed-forward inner representation size d_{ff} to four times the hidden size d_{model} [50]. We use the *gelu* activation function [21] rather than the standard *relu*, following BERT [10]. During training, we set the batch size to 64 and the learning rate to 1×10^{-4}. We use the Adam optimizer [28] and set $\beta_1 = 0.9, \beta_2 = 0.999$ and $\varepsilon = 1 \times 10^{-8}$. We apply gradient clipping by value within the range $[-1, 1]$. We also apply a dropout rate of 0.1 as regularization. We train the model for 15 epochs. At the inference time, we use beam search to predict the types for each function with a beam size of 5.

Hardware Configuration   We conducted all experiments on Linux servers equipped with two Intel Xeon Gold 6148 processors, 192GB RAM and 8 NVIDIA Volta V100 GPUs. We expect that a similar machine could reproduce the full training and testing stage of our main experiments in 120 GPU hours.

Software   We implemented our models with PyTorch [41] version 1.5.1 and Python 3.6. We plan to release our dataset, code and pre-trained models at publication time.

Hyperparameter	DIRTY	DIRTY$_S$
Max Sequence Length	512	512
Encoder/Decoder layers	6/6	3/3
Hidden units per layer	512	256
Attention heads	8	4
Layout encoder layers	3	3
Layout encoder hidden units	256	128
Batch size	64	64
Training epochs	15	30
Learning rate	10^{-4}	10^{-4}
Adam ε	10^{-8}	10^{-8}
Adam β_1	0.9	0.9
Adam β_2	0.999	0.999
Gradient clipping	1.0	1.0
Dropout rate	0.1	0.1
Number of parameters	167M	40M

Table 10: Summary of the hyperparameters of DIRTY and the smaller DIRTY$_S$.

Dataset	DIRTY
#Binaries	75,656
Unique #functions (train)	718,765
Unique #functions (valid)	139,473
Unique #functions (test)	139,394
% func body in train (valid)	64.6%
% func body in train (test)	65.5%
Avg. #code tokens	220.3
Median #code tokens	86
Avg. #identifiers per function	5.1
Median #identifiers per function	3

Table 11: Statistics of the DIRT datasets.