DRAGON: Predicting Decompiled Variable Data Types with Learned Confidence Estimates

Caleb Stewart, Rhonda K. Gaede, and Jeffrey H. Kulick
The University of Alabama in Huntsville
{cls0027, gaeder, kulickj}@uah.edu

***Abstract*—We present DRAGON, a graph neural network (GNN) that predicts data types for decompiled variables along with a confidence estimate for each prediction. While we only train DRAGON on x64 binaries compiled without optimization, we show that DRAGON generalizes well to all combinations of the x64, x86, ARM64, and ARM architectures compiled across optimization levels O0-O3. We compare DRAGON with two state-of-the-art approaches for binary type inference and demonstrate that DRAGON exhibits a competitive or superior level of accuracy for simple type prediction while also providing useful confidence estimates. We show that the learned confidence estimates produced by DRAGON strongly correlate with accuracy, such that higher confidence predictions generally correspond with a higher level of accuracy than lower confidence predictions.**

I. Introduction

Binary type inference seeks to recover high-level data types from compiled binary code. Since the data types of program variables represent vital information about the code and its functionality, type inference is a critical problem for many kinds of binary analysis such as decompilation. Additionally, performing type recovery as a preliminary step has been shown to enhance other kinds of analysis such as fuzzing [1].

The problem of inferring data types from executable programs has been studied as early as 1999 when COBOL program analysis was explored for its utility in solving the Y2K problem [2]. Since then, the focus has shifted to evaluting data type recovery methods on programs primarily compiled from C or C++ source code using both static and dynamic analysis techniques [3], [4]. More recently, a wide variety of machine learning approaches have been heavily researched and applied with great success.

However, while many machine learning approaches have been proposed, none of them convey information about the level of confidence associated with individual predictions. Since no machine learning model is perfect, all models generate both correct and incorrect predictions which are indistinguishable in the absence of ground truth. Even though highly accurate models tend to produce fewer bad predictions on average, there is no clear way to separate the good from the bad for arbitrary unlabeled samples.

In this paper, we present a technique to recover data types from executable code using a GNN model. We incorporate *learned confidence estimates* [5], an existing uncertainty quantification technique that augments each prediction with a probability representing the faith the model has in its prediction. To our knowledge, we are the first to apply any such technique to a binary type inference machine learning model.

II. Background

While many solutions to binary type inference[1] have been proposed, the application of GNNs remains largely unexplored. Additionally, existing solutions present opportunities for improvement.

Zhu et al. recently proposed TYGR, the first[2] GNN model applied to binary type inference [7]. TYGR performs data type inference using a data flow graph representation of a program. The authors demonstrate that TYGR performs at a level superior to state of the art, and show that GNN models can be effectively applied to the type inference problem. However, TYGR presents only one possible method for mapping relevant program information to a graph representation. We explore decompiled abstract syntax trees (ASTs) as an alternate representation for use with a GNN model.

While the use of GNNs on program representations has been previously studied in contexts where source code is available (both for type inference [8] and AST representations specifically [9], [10]), the problem space for stripped binary code is distinctly different as it lacks much of the information available to source representations such as identifier names and source-level data types. To our knowledge, we are the first to use a GNN model on decompiled ASTs for binary type inference.

A. Syntactic Simple Type Recovery

Type inference can be *semantic*, recovering a meaningful type name in addition to its definition, or simply *syntactic*, recovering only the structural form of the type. Furthermore, we differentiate syntactic types as being either *simple* or *complex*. Simple types only represent the essential form of their data types, including the native type or category (e.g.

Workshop on Binary Analysis Research (BAR) 2025
28 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-4-2
https://dx.doi.org/10.14722/bar.2025.23025
www.ndss-symposium.org

[1] We use the terms *inference*, *prediction*, and *recovery* interchangeably

[2] TIARA [6] notably used a GNN model to classify C++ container types. We consider this to be a related but different problem.

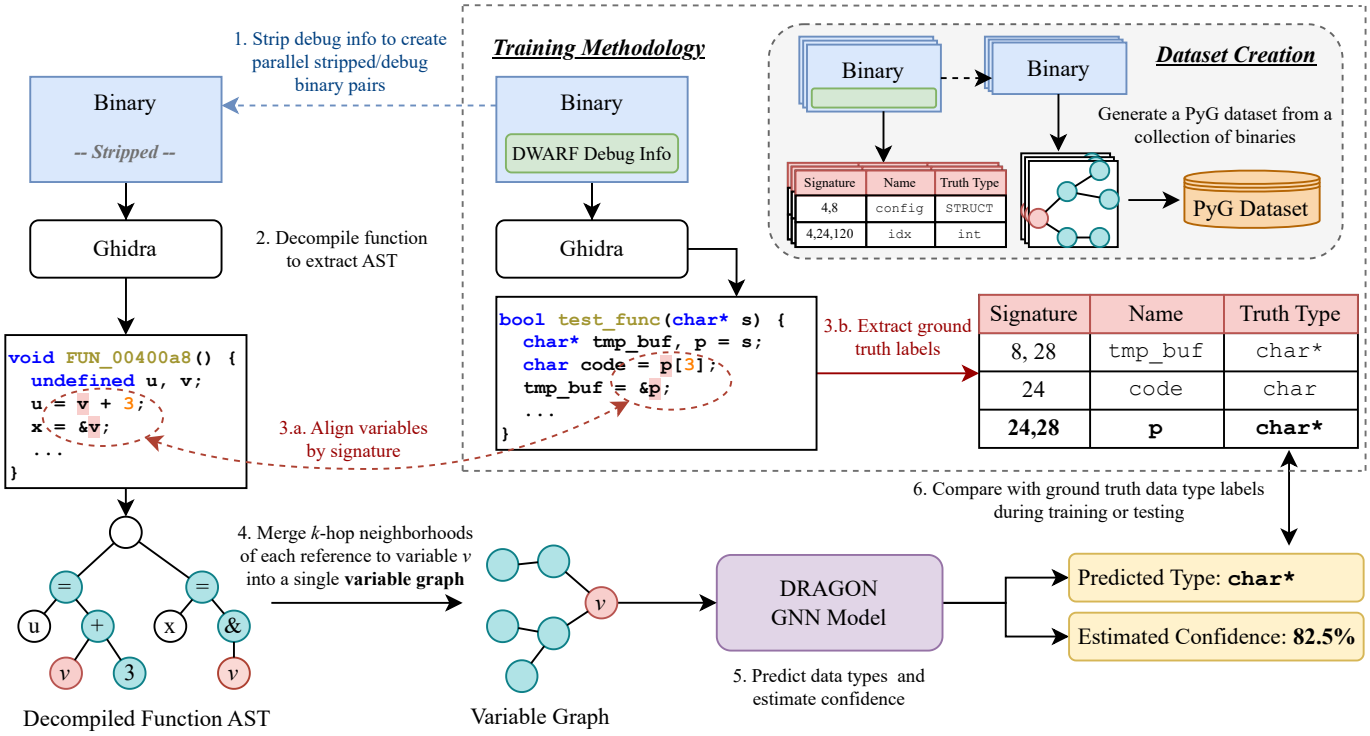


Fig. 1. Overview of DRAGON architecture shows the flow of data as it moves from the stripped binary (top left) through the Ghidra decompiler to the variable graph extracted from the AST and into DRAGON which predicts the data type and confidence estimate. The *Training Methodology* shows the extensions that allow us to build a labeled dataset, including parallel debug and stripped binary pairs and the variable alignment method developed by [11]. Finally, *Dataset Creation* shows that we convert many binaries to a complete Pytorch Geometric [12] dataset for training and testing.

`int`, `double`, `struct`, or `union`) as well as any pointer or array components, but do not include aggregate type information such as structure definitions. Complex types also include simple type information but add structure definitions. In this paper, we study the problem of *syntactic simple type prediction* and do not recover structure definitions or semantic type names.

III. Related Work

Traditional approaches to binary type inference have contrasted algorithmic approaches using static or dynamic analysis [3], [4], [13]. OSPREY recently demonstrated significant improvement in accuracy over Howard by using a probabilistic approach based on a static analysis path-sampling technique called BDA [14].

More recently, the focus of the field has heavily shifted to applying machine learning techniques. TypeMiner [15] uses machine learning classifiers to predict up to 17 simple types, and Debin [16] uses a conditional random field model to both recover variables and predict their name and data type. Other approaches have focused on function prototype recovery [17], [18]. CATI uses convolutional neural network classifiers to predict 19 simple data types [19]. StateFormer recovers 35 simple types using generative state modeling to pretrain a model which they finetune for type inference [20].

DIRTY [21] is in many aspects quite similar to DRAGON. Both DRAGON and DIRTY accept decompiled function ASTs as input, while DIRTY uses a Transformer model architecture as opposed to the GNN of DRAGON. As described in Section V-A1, we employ the variable alignment technique used by both DIRTY and DIRE [11] to align variables from our stripped binaries with their corresponding ground truth labels available in the same binaries with DWARF debug information.

STRIDE [22] is another interesting approach that questions the unchallenged trend of continually increasing model sizes and levels of complexity. By using a simplistic n-gram model for type recovery, STRIDE is able to perform competitively with DIRTY while requiring much less overhead in both model size and execution time. TYGR presents the first application of GNNs to binary type inference, and outperforms both DIRTY and OSPREY [7]. RESYM [23] presents the first application of large language models (LLMs) to binary type inference and also demonstrates superior performance to DIRTY and OSPREY.

Graph neural networks have successfully been applied to source code analysis in a variety of ways. [9] and [10] use GNNs on source code ASTs to address problems such as code clone detection. [8] tackles the type inference needed to support gradual typing using GNNs on a type dependency graph representation of TypeScript code.

While there is synergy between the two areas of study, particularly with respect to effective strategies of mapping AST data to GNN models, the form of an AST decompiled from a binary functions is fundamentally different than that of an AST compiled from source code. For example, since modern decompilers still do not recover structures from stripped binaries (other than as a result of library function recognition techniques), the ASTs in a decompiled function lack structure member expressions almost completely, while these are quite common in source code ASTs.

A. Learned Confidence Estimation

We use learned confidence estimates, a technique developed by DeVries and Taylor [5] which is shown to outperform simple softmax output thresholding as well as an out-of-distribution detection technique called ODIN [24]. While we do not perform a study comparing alternatives for uncertainty quantification, this presents an interesting future area for investigation.

IV. Approach

A. Overview

We develop DRAGON[3], a multi-task [25] GNN model to predict simple syntactic data types for variables from a decompiled AST. As seen in Figure 1, DRAGON accepts a *variable graph* derived from a decompiled AST as input. The variable graph combines the local neighborhoods of each reference to a target variable within a function, and presents it to the model as input. For each such graph, DRAGON uses a GNN model to predict both the variable's data type as well as the level of confidence DRAGON has in the prediction. We recover data types for both local variables and function parameters.

B. Abstract Syntax Tree Representation

As Ghidra does not internally use a traditional AST for decompiled functions (instead, they directly convert the final Pcode into a stream of C code tokens), we modify the Ghidra decompiler to output decompiled ASTs to a JSON format inspired by the Clang AST [26] but simplified to support the subset of C language elements actually produced by Ghidra.

C. Graph Neural Network Architecture

A graph (G) is defined as a set of vertices (V), or *nodes*, and a set of edges (E) that connect them, and is represented as $G = (V, E)$. Graph neural networks are composed of a sequence of layers, with each layer computing a hidden state, or *embedding*, $\overrightarrow{h}_i$ for each node v_i in the graph. Each node embedding $\overrightarrow{h}_i$ aggregates information from its immediate neighborhood $\mathcal{N}_i$ and passes the resulting vector through an activation function. GNNs propagate information throughout the graph by iteratively computing node embeddings for every layer.

[3](**D**ata type **R**ecovery from decompiled **AST**s using **G**NNs with learned c**ON**fidence)

We use graph attention network (GAT) layers for the GNN portion of our model [27]. Equation 1 computes the unnormalized attention score ϵ_{ij} for an edge (v_j, v_i) as described in the original paper. We denote the transpose of $\mathbf{x}$ as $\mathbf{x}^T$ and concatenation as $\|$.

$$\epsilon_{ij} = \text{LeakyReLU}\left(\overrightarrow{\mathbf{a}}^T[\mathbf{W}\overrightarrow{h}_i \| \mathbf{W}\overrightarrow{h}_j]\right) \tag{1}$$

We compute the attention coefficients α_{ij} for edge (v_j, v_i) as shown in Equation 2. Note that $\mathcal{N}_i$ is augmented with i as we use self-loops for all nodes.

$$\alpha_{ij} = \frac{\exp\left(\epsilon_{ij}\right)}{\sum_{k \in \mathcal{N}_i \cup \{i\}} \exp\left(\epsilon_{ik}\right)} \tag{2}$$

Finally, the overarching equation that uses the attention coefficients to compute the embedding $\overrightarrow{h}'_i$ of node i for layer (as a function of terms from the previous layer) is given in Equation 3:

$$\overrightarrow{h}'_i = \Big\Vert_{k=1}^{K} \text{ReLU}\left(\sum_{j \in \mathcal{N}_i \cup \{i\}} \alpha_{ij}^k \mathbf{W}^k \overrightarrow{h}_j\right) \tag{3}$$

where K is the number of attention heads. We use an extended form of Equation 1 to incorporate edge feature information into the embedding, as shown in Equation 4:

$$\epsilon_{ij} = \text{LeakyReLU}\left(\overrightarrow{\mathbf{a}}^T[\mathbf{W}\overrightarrow{h}_i \| \mathbf{W}\overrightarrow{h}_j \| \mathbf{W}\overrightarrow{e}_{ij}]\right) \tag{4}$$

where $\overrightarrow{e}_{ij}$ represents the *edge features* associated with edge (v_j, v_i).

While we compute the forward pass on all nodes in the input graph to allow data to flow freely, we only pass $\overrightarrow{h}'_t$, the final embedding vector for the target node v_t from the GNN layers to the shared layers as shown in Figure 4. As Ghidra recovers an initial data type for the target node, we forward the concatenation of the GNN embedding and this encoded data type to the shared layers. This ensures the shared layers may always observe the original decompiled data type, uninhibited by any potential loss during graph layer propagation.

D. Data Type Encoding

A data type can be thought of as a combination of *pointer levels* and a terminal type, or *leaf type*. The pointer levels describe how many pointer indirections are present, and the leaf type is the pointed-to type that remains after all pointer levels have been dereferenced. We extend the pointer level concept to also include levels of arrays, which may be freely interspersed with pointers in our scheme.

Our data type encoding is inspired by the multi-stage classification designs of TypeMiner [15] and CATI [19]. We encode the pointer hierarchy and leaf type separately, as shown in Figure 2. We select a fixed length of three pointer levels. The full data type vector is the concatenation of the pointer level vectors with the leaf type vector. Additional details of the encoding are described in Appendix A.

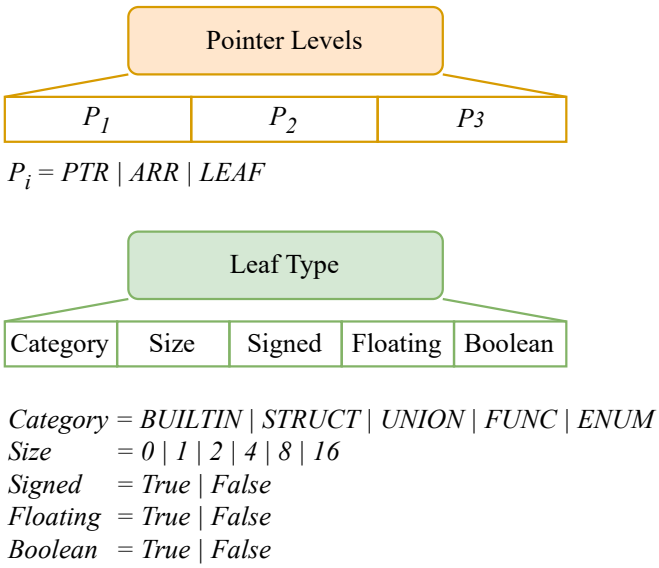


Fig. 2. The DRAGON data type encoding scheme separates the pointer levels of a data type from its leaf type, and defines each element as shown.

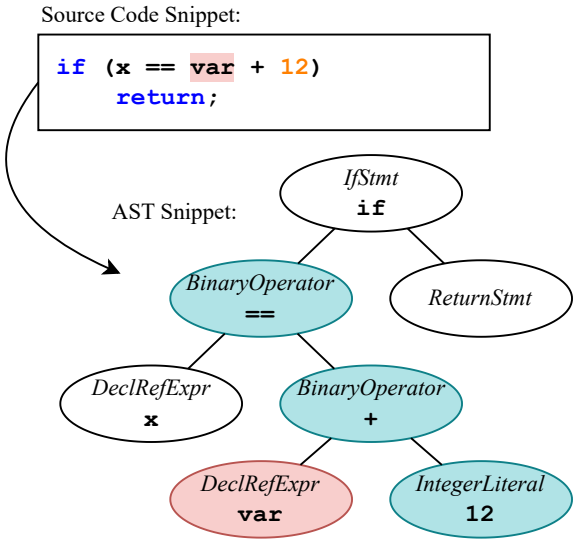


Fig. 3. K-hop neighborhood for node *var* where k=2 (nodes included in this neighborhood are shown in blue)

E. Node Encoding

Each node in the graph is encoded using its *kind*, *data type*, and *operation*. The node kind is its class of AST node, such as *BinaryOperator*, *DeclRefExpr*, or *VarDecl*. The data type is the type associated with this node if one exists, or an empty vector of all zeroes. For example, we encode *IntegerLiteral* nodes with the associated integer data type, and *DeclRefExpr* nodes with the data type of the referenced variable. For *BinaryOperator* and *UnaryOperator* nodes, the operation field encodes the associated C operation such as `+`, `-`, `==`, and `&`. For all other nodes, we encode a special operation value representing an empty operation.

F. Edge Features

We encode edge features to help the model differentiate between situations where the order of child nodes is significant. As pointed out by [10], a homogenous GNN architecture cannot natively distinguish between the expressions `a - b` and `b - a` in an AST graph because all edges appear the same. We one-hot encode an *edge type* for each edge in the graph to help address this issue. Rather than encode every *possible* edge type uniquely, we encode the edges we perceive as requiring distinction with unique edge types, and encode all other edges with a *Default* edge type. We encode each of the following kinds of edges with unique edge types:

- Edges to children of binary operators.
- Edges to children of `if` statements.
- Edges to children of a function call expression, up to the first six arguments
- Edges to children of array subscript expression
- Edges to children of `do`, `while`, `for`, and `switch` statements.

G. Variable Graphs (Inputs)

We generate a *variable graph* for a specific decompiled target variable by collecting the *k-hop neighborhood* of every reference to the target variable within the decompiled AST. In our AST, these appear as *DeclRefExpr* nodes. We merge each of these k-hop neighborhoods into a single input graph by merging each reference node into a single node. This merged node will function as the target node when presented to the GNN, and we compute its embedding to be passed on to the shared layers and through to the output. To create the final graph, we follow the common practice of adding edges bidirectionally to facilitate the flow of data throughout the graph. Figure 3 shows the 2-hop neighborhood of a reference to variable `var` in the AST snippet.

H. Multi-Task Outputs

We use a multi-task architecture [25] to split the embedding output from the shared layers into individual outputs that map to elements of our data type encoding, as shown in Figure 4. Although not explicitly shown in Figure 4, we also forward some of the individual task outputs to the inputs of other task-specific layers by concatenation with the shared embedding. This was chosen ad hoc based on observation during model development, with some higher-performing outputs being fed into other task layers to boost overall performance. Specifically for the *Boolean* layers, we augment the shared embedding with the original encoded Ghidra data type vector as well as the output logits from the *Category*, *Signed*, and *Floating* branches. Similarly, the *Size* layers receive this same augmented input vector plus the output of the *Boolean* branch itself.

We correct invalid pointer level predictions by extending the first predicted *LEAF* element to all subsequent elements. For example, an invalid prediction of *LEAF, PTR, LEAF* would be converted to a data type with no pointers or arrays.

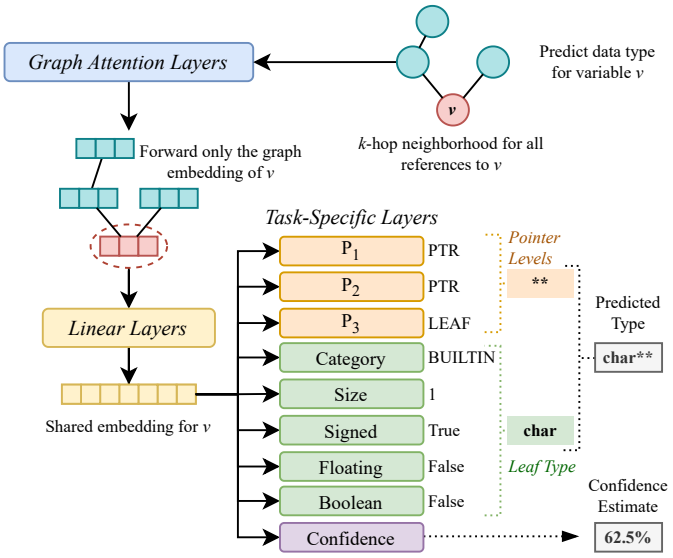


Fig. 4. The DRAGON model architecture 1) computes a graph embedding for variable v, 2) passes this embedding into the shared linear layers, and 3) passes the shared embedding to each task-specific branch as well as to the learned confidence branch to 4) generate a predicted data type and confidence estimate for v.

I. Learned Confidence and Loss Functions

We use the approach described by DeVries and Taylor [5] to incorporate learned confidence estimates into DRAGON. This technique augments each predicted data type with a *confidence* value between zero and one, representing the level of faith the model has in its own prediction. Rather than simply interpreting the softmax outputs of a model as a measure of confidence, this approach specifically modifies the training process such that the model learns to recognize the level of uncertainty associated with a particular input. This is primarily achieved by allowing the model to "ask for hints" during training based on its confidence estimate. As explained in detail in [5], we implement this by interpolating between the prediction and ground truth, with the confidence estimate controlling the weighting between the two. A lower confidence estimate weights the output more heavily towards ground truth, while a high confidence estimate primarily uses the prediction. The interpolated vector is provided as input to the loss function, which also balances the training incentives such that the model does not simply predict confidence estimates of zero to "view" ground truth while learning nothing useful.

We integrate learned confidence into DRAGON by splitting a confidence prediction branch off of the shared layers as seen in Figure 4. This results in a single confidence estimate for the prediction as a whole, and not separate confidence values for each task prediction head.

We modify our loss function as described in [5] to combine the prediction loss with the confidence loss and extend it for multi-task outputs. We refer to the set of tasks in the multi-task network as T. We first define the confidence loss $\mathcal{L}_c$ as shown in Equation 5, where c is the single predicted confidence probability applying to the entire model, as output by the sigmoid function.

$$\mathcal{L}_c = -\log c \tag{5}$$

We interpolate between the predicted class probabilities p_i and ground truth as described in [5] for each task individually, rendering a separate p_i' per task. Each task's interpolated prediction is then passed to the loss function for that task to obtain the task loss, $\mathcal{L}_t$. We define $\mathcal{L}_T$, our overall task loss, as the sum of task-specific losses for all tasks $t \in T$, and combine this with $\mathcal{L}_c$ as shown in Equation 6 to compute our final loss, $\mathcal{L}$:

$$\mathcal{L} = \sum_{t \in T} (\mathcal{L}_t) + \lambda \mathcal{L}_c \tag{6}$$

where λ is a hyperparameter used to balance between the two loss terms. To prevent the confidence estimates from approaching 1 during training, DeVries and Taylor define a budget parameter (β). The budget parameter manages the balance between $\mathcal{L}_t$ and $\mathcal{L}_c$ by controlling λ dynamically: if $\mathcal{L}_c > \beta$ then increase λ, and if $\mathcal{L}_c < \beta$ then reduce λ. In our implementation, we use an initial λ value of 0.1 and a β value of 0.3. For our task-specific loss functions, we use binary cross entropy loss for all binary classification outputs (*Signed*, *Floating*, and *Boolean*) and negative log likelihood for all others.

V. Experimental Setup

A. Training Dataset

Rather than create a brand new dataset of training binaries from scratch, we collect our training binaries from the TyDa dataset, released as part of [7]. TyDa is a large dataset of over 160,000 debug binaries compiled for a variety of architectures and optimization levels. To train and evaluate TyGr, the authors use only a subset of TyDa due to its massive size. They collect a sample of roughly 8% of the TyDa binaries which they refer to as TyDa_{MIN}.

Since the exact composition of TyDa_{MIN} is unknown, we generate a sample from TyDa of similar size called $\text{TyDa}_{\text{MIN-D}}$. We sample only from non-stripped x64 O0 binaries with debug information that were compiled from C code projects.

We use the scripts released by [7] to deduplicate the functions across our entire dataset. We then split the dataset into 80% training, 10% validation, and 10% test splits, with the validation set being used for model tuning and the test set held out completely until we evaluate our final model.

In addition to $\text{TyDa}_{\text{MIN-D}}$, we use ReSym's training dataset to retrain another instance of DRAGON from scratch. This is described further in Section VII-B2.

1) Aligning Variables by Signature: Due to the inherent imprecision of variable recovery, it is not always trivial to align decompiled variables with ground truth in order to create a labeled dataset. To address this issue, we use the variable alignment technique described by DIRE [11] and DIRTY [21].

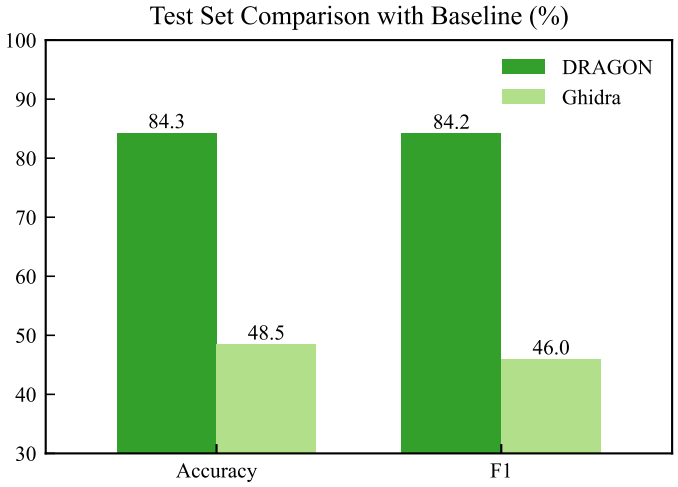


Fig. 5. DRAGON achieves a significant improvement in both Accuracy and F1 as compared with Ghidra on the test set.

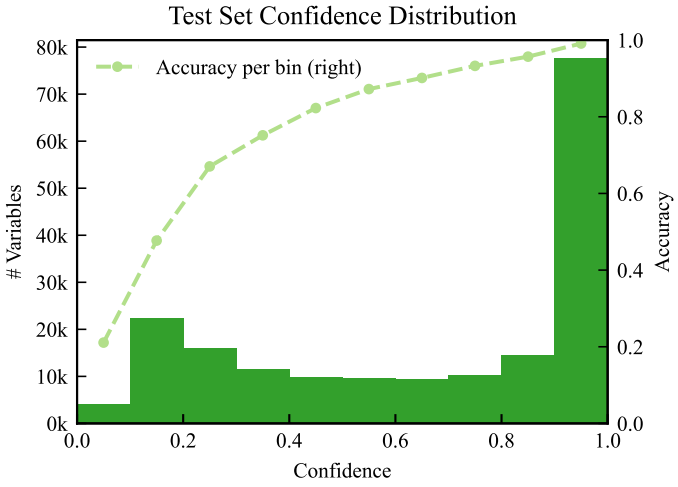


Fig. 6. Histogram of confidence estimates for test set variables. The overlaid line plot shows the average accuracy of variables falling within each bin.

2) Excluded Variables: We discard variables that have duplicate signatures, as we cannot differentiate them by signature only. Additionally, Ghidra recovers variables that exist in a special *Unique* address space which does not correspond actual processor storage locations. The *Unique* address space is defined by Ghidra to handle temporary values when modeling the effects of processor instructions in Pcode. However, these variables will occasionally persist through the decompilation process and be included in the final set of decompiled variables. As these variables do not correspond to any real processor storage location and essentially behave as a function of the other variables in the decompiled code, we exclude *Unique* variables as well.

B. Metrics

We evaluate the performance of DRAGON using the metrics of Accuracy and F1 score. Accuracy is simply the ratio of correct predictions to the total number of predictions. Precision and Recall determine F1 score, and are defined as functions of the number of true positives (TP), false positives (FP), and false negatives (FN) as shown in equations 7 and 8:

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

F1 can be calculated as the harmonic mean of Precision and Recall according to Equation 9:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{9}$$

We aggregate F1 score using weighted averaging.

C. Model Training

We instantiate DRAGON with 5 graph attention layers, 2 shared linear layers, and 2 additional linear layers for each task. Each layer uses ReLU as an activation function. Confidence contains a single linear layer only. Each graph attention layer has 8 heads, and each graph and shared layer have a dimensionality of 256 with the task-specific layers having a dimensionality of 128. We use a dropout of 5% for the GNN and shared linear layers during training. We train with a learning rate of $5e{-}5$ and a batch size of 32 for 28 epochs, choosing the model with the highest validation accuracy at epoch 20. We train DRAGON on a single NVIDIA H100 GPU and 64GB of RAM with a 30-hour time limit.

VI. Implementation

We modify the decompiler from Ghidra version 10.3 to export decompiled function ASTs to JSON. We import binaries into Ghidra with *Decompiler Parameter ID* analysis enabled and the *DWARF Debug Item Limit* set to 1 billion. We use Pytorch Geometric (PyG) 2.5 to implement DRAGON [12]. A Github repository containing directions to run our code will be released here: https://github.com/lasserre/dragon.

VII. Evaluation

A. Test Set Performance

1) Comparison to Baseline: We evaluate the performance of DRAGON on the $\text{TyDa}_{\text{MIN-D}}$ test set using Ghidra as a baseline. The results are shown in Figure 5. DRAGON demonstrates a clear performance improvement over Ghidra in both Accuracy and F1. Because DRAGON accepts decompiled Ghidra variables as input, this comparison is made on the exact same set of variables.

2) Learned Confidence Performance: We also use the test set to evaluate the confidence estimates. Figure 6 shows a histogram of the confidence values associated with each prediction from the test set. We overlay the accuracy of the variables contained in each bin of the histogram.

As seen in Figure 6, the accuracy of each bin steadily increases with bins of increasing confidence values, demonstrating that high confidence predictions tend to outperform low confidence predictions. Additionally, while the distribution of confidence values is not very pronounced for this benchmark,

TABLE I
COMPLEX BENCHMARK BINARIES, INCLUDING PROJECT INFORMATION AND FILE SIZE OF DEBUG BUILDS WITH DWARF.

Project	Version	Binary	ELF File Size (-g)
Apache	2.4.62	httpd	2.0 MB
Nginx	1.26.2	nginx	4.3 MB
OpenSSL	1.1.1k	libssl.so	2.0 MB
Redis	7.2.4	redis-server	16.0 MB
Sqlite	34.7.02	sqlite3	3.8 MB

it appears to resemble a bimodal distribution. This makes sense as the model has been incentivized during training to separate the samples for which it recognizes (in-distribution) from samples for which it is much less familiar (out-of-distribution).

B. Comparison with Prior Work

We compare DRAGON with TYGR [7] and RESYM [23], which are two techniques representing state of the art in binary type inference.

1) TYGR *Comparison:* TYGR is a state-of-the-art GNN type recovery model that predicts data types for nodes from a data flow graph. TYGR has the ability not only to predict simple data types but also to recover user-defined structure definitions. Since DRAGON only predicts simple data types, we only compare with the simple type recovery capability of TYGR.

We perform our evaluation using two different benchmarks. We include Coreutils as it is commonly used for evaluating type inference techniques [3], [7], [14], [21], [23]. Taking inspiration from OSPREY [14] we assemble a second *Complex* benchmark to represent a higher level of difficulty. To form our complex benchmark, we combine the two binaries from OSPREY's complex benchmark (Apache and Nginx) and add three additional binaries as seen in Table I. Since TYGR has a simpler type system than DRAGON, we project both sets of types to a common type system for comparison, as detailed in Appendix B.

Another significant difference is that TYGR makes data type predictions one prediction per data flow graph node, while DRAGON makes a single prediction per variable regardless of the number of references to that variable. Thus the number of TYGR predictions is disproportionately higher than the number of DRAGON predictions on the same data, as shown in the sample data from the Coreutils benchmark in Figure 7. To avoid the sheer number of TYGR samples from dominating the results, we reduce the TYGR predictions down from per-instance predictions to per-variable predictions. For each ground truth variable, we reduce the set of per-node TYGR predictions by taking the *mode* (the most commonly predicted data type). Figure 7 shows that reducing the TYGR predictions results in a set that is of the same order of magnitude as the set of DRAGON variables.

Coreutils Benchmark. We use the pre-trained models provided by TYGR. TYGR provides models trained on each combination of five different architectures and four optimization levels. We compare TYGR and DRAGON across four architectures (x64, x86, ARM, and ARM64) at the O0 optimization level. *However, we only train DRAGON on x64 O0 binaries, and use this single model for all* TYGR *evaluations*.

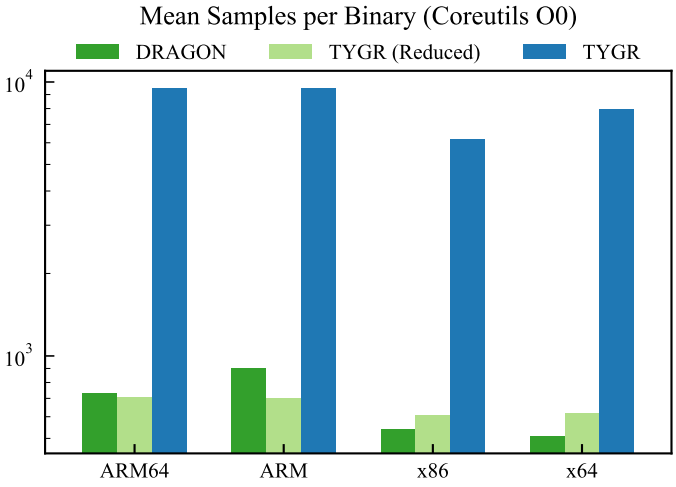
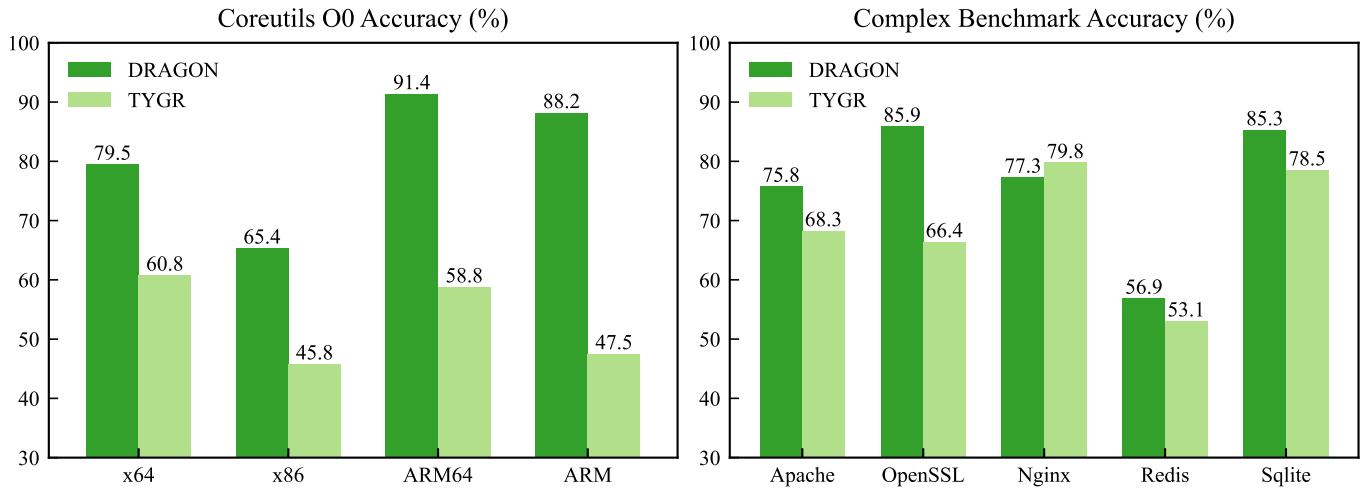


Fig. 7. Reducing the node-level TYGR predictions to variable-level predictions enables a fairer comparison at the same order of magnitude.

Figure 8a presents the comparison results between TYGR and DRAGON on Coreutils. Despite only using a single DRAGON model trained on x64 O0 binaries, DRAGON outperforms TYGR on this dataset. This result makes sense because the inputs presented to DRAGON are directly derived from decompiled ASTs, which lack architecture-specific information. What is interesting is that DRAGON is able to maintain a high level of performance without relying on architecture-specific information such as specific assembly code instructions and their operands. These results suggest that training a single model is sufficient for maintaining a relatively stable level of performance across binaries compiled for different architectures.

We also observe that our experiment results in a lower accuracy from TYGR than that described in their previously published results on the same Coreutils benchmark [7]. We believe this difference is largely a result of the different evaluation techniques used in the two experiments. Specifically, the OSPREY type system used in the TYGR evaluation is much simpler than the type system we use for our evaluation. Additionally, our reduction of TYGR node-level predictions to a single prediction per ground truth variable greatly reduces the size of the TYGR prediction set. Finally, the previously published results exclude functions contained in the TYGR training set, while we perform no such exclusions. We also point out that there may be better alternatives for transforming the node-level predictions into variable-level predictions, which could result in higher accuracy as well.

Complex Benchmark. The results of the complex benchmark evaluation are shown in Figure 8b. As seen in Table I, the Redis binary is significantly larger than the others. This



(a) DRAGON demonstrates a higher level of accuracy than TYGR on Coreutils compiled at O0 across all architectures.

(b) DRAGON achieves superior accuracy to TYGR for every binary except Nginx, on which it remains competitive at only 2.5% less than TYGR.

Fig. 8. TYGR comparison results show that DRAGON maintains competitive or superior performance.

seems to challenge both DRAGON and TYGR equally, as their accuracies collectively plummet below 60%.

DRAGON exhibits competitive performance with TYGR on each of the binaries in the *Complex* benchmark, exceeding the accuracy of TYGR on all but Nginx, for which it only lags by about 2.5%. This demonstrates that DRAGON can predict simple types effectively on binaries of non-trivial complexity as compared with state of the art.

2) RESYM *Comparison:* RESYM [23] is an LLM-based approach that is comprised of multiple components for variable type prediction, structure field recovery, and a reasoning algorithm which post-processes the final results. We compare only with the VarDecoder model, which predicts data types for decompiled variables from Hex Rays just as DRAGON does from Ghidra. We compare DRAGON with the results released by RESYM.

For this comparison, we retrain DRAGON from scratch on the functions from RESYM's training set.[4] We reserve 10% of the data for validation and select the model with the highest validation accuracy for comparison. We train for 35 epochs within a 24-hour time limit. Epoch 32 provides the highest validation accuracy. All other hyperparameters are kept the same as with training on the $\text{TYDA}_{\text{MIN-D}}$ dataset (Section V-C). This retrained DRAGON model is *DRAGON$_R$* below.

Since the outputs of VarDecoder are simple strings, we perform some post-processing to interpret and project the RESYM-predicted data types to our type system. First we remove all variables whose ground truth is labeled with a dash (`"-"`). RESYM uses this label for decompiled variables that do not align with the start of a ground truth variable. Since our alignment technique (Section V-A1) excludes such variables naturally, we exclude ground truth dashes from the RESYM test set.

[4]Ghidra failed to import six of the binaries and their functions were excluded

TABLE II
DRAGON$_R$ DEMONSTRATES COMPETITIVE PERFORMANCE WITH RESYM.

	Accuracy	F1
DRAGON$_R$	**0.8702**	**0.8725**
RESYM	0.8595	0.8613

We begin by processing DWARF debug information from all binaries in both the training and test sets to create a lookup table mapping user-defined type names to their canonical data type definitions. We validate that this covers all ground truth types present in the test set. For each VarDecoder test set prediction, we remove any qualifiers such as `const`, parse the form of the data type, and look up the named portion of the type (such as a typedef or structure name) in our previously generated lookup table. This form of the data type is then recorded in the equivalent format as the usual DRAGON data types. For example, if `T` is a typedef to a structure, we convert the type `T**` to *PTR, PTR, STRUCT*. During the type name lookup step, we ignore any possible ambiguity in type names, since VarDecoder itself does not distinguish beyond the name of the predicted type as the output.

As VarDecoder fine-tunes an existing model, its vocabulary includes type names from data outside the RESYM dataset. While all ground truth types map to a known definition in our lookup table, some VarDecoder predictions do not. While RESYM would have considered these as incorrect predictions due to the type *name* being incorrect, we only consider the syntactic type. Since we have no knowledge of the form of these data types (e.g. enum, structure, typedef, etc.), we exclude these RESYM predictions from the evaluation. These cases account for only 1.35% of the original set of predictions.

Table II presents the results of the evaluation on the RESYM test set binaries. DRAGON$_R$ demonstrates a competitive level

TABLE III
DRAGON GENERALIZES WELL ACROSS ALL COMBINATIONS OF ARCHITECTURE AND OPTIMIZATION LEVEL ON THE COREUTILS BENCHMARK.

	O0		O1		O2		O3	
	Accuracy	F1	Accuracy	F1	Accuracy	F1	Accuracy	F1
ARM	0.870	0.867	0.813	0.807	**0.842**	**0.841**	**0.856**	**0.854**
ARM64	**0.905**	**0.902**	**0.864**	**0.856**	0.839	0.829	0.838	0.829
x64	0.768	0.766	0.721	0.715	0.711	0.698	0.676	0.670
x86	0.624	0.602	0.672	0.656	0.669	0.644	0.637	0.618

of performance, just exceeding the Accuracy and F1 of RESYM. While this VarDecoder model from RESYM was originally trained for both variable name and semantic type prediction, DRAGON_R performs competitively for simple type prediction.

C. Generalizability

We further evaluate the generalizability of DRAGON on the Coreutils benchmark compiled for each combination of architecture (x64, x86, ARM64, and ARM) and optimization level (O0, O1, O2, and O3). Table III shows the Accuracy and F1 score for each combination. Note that while the O0 column is the same configuration as in the TYGR comparison, we do not project DRAGON types to the (simpler) common type system (and thus the accuracy is slightly lower).

DRAGON demonstrates consistent generalizability across all combinations of architecture and optimization level. Interestingly, the ARM and ARM64 configurations outperform the x64 architecture on which DRAGON was trained. As decompilers routinely produce different-looking pseudocode for different architectures, we speculate the higher accuracy exhibited by DRAGON on ARM may be a downstream result of differences in Ghidra's analysis on ARM vs. x64 such as fundamental differences in variable recovery. However, we leave a detailed investigation of these effects for future work.

D. Learned Confidence Evaluation

We evaluate the behavior of the learned confidence estimates by comparing the distribution of confidence values with prediction accuracy for three binaries from the *Complex* benchmark. We select Redis, Apache, and OpenSSL as these three binaries represent low, medium, and high levels of overall accuracy. Figure 9 presents the results of this comparison. Figure 9a shows the confidence distributions of each binary, sorted by the overall prediction accuracy achieved by DRAGON on that binary. The median confidence values shown in the box plot can be seen steadily increasing with increasing accuracy, demonstrating that the learned confidence estimates convey a meaningful sense of the quality of results we can reasonably expect. Figures 9b-9d show a histogram of confidence values produced for each binary, along with the average accuracy of the variables contained within each bin.

VIII. DISCUSSION

Our evaluations demonstrate that DRAGON performs competitively with state-of-the-art approaches, including both another GNN-based model operating at the data flow graph level as well as an LLM-based model. DRAGON's competitiveness with TYGR is interesting because it demonstrates that the information contained in the more abstract AST representation is sufficient for effective syntactic type recovery, rather than the lower level of abstraction used by TYGR.

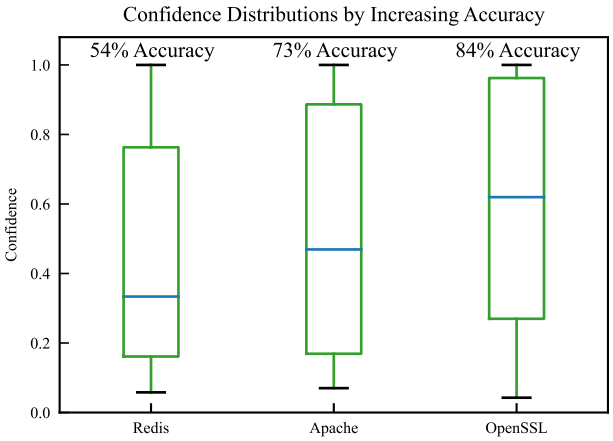
One important point is that, as with any machine learning approach, the quality of the training dataset can have a profound impact on the utility of the resulting model. While our training dataset could definitely be enhanced in the future, we believe our dataset is sufficiently robust both relative to state of the art (we use datasets of nearly identical size and complexity) as well as to real-world programs, on which we demonstrate a high level of accuracy.

We observe that the distribution of confidence values can help provide insight on the relative complexity of an unknown dataset. While the particular confidence values produced by a model are unique to that particular model, its training parameters, and the dataset it was trained on, comparing the confidence distribution of different datasets *using the same model* provides a possible method of comparison.

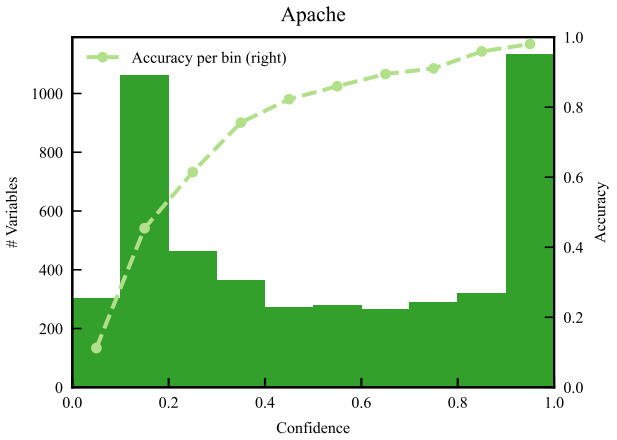
In [5], DeVries and Taylor address challenges to effectively training a model to estimate confidence. One such challenge occurs when there are insufficient misclassified samples during training, which happens when the model is able to correctly classify the vast majority of inputs. To combat this problem, they apply data augmentation techniques to synthetically generate hard to classify inputs. While we do not apply data augmentation to train DRAGON, we believe our training dataset was sufficiently complex, as demonstrated by our evaluation results of the confidence estimates.

IX. CONCLUSION

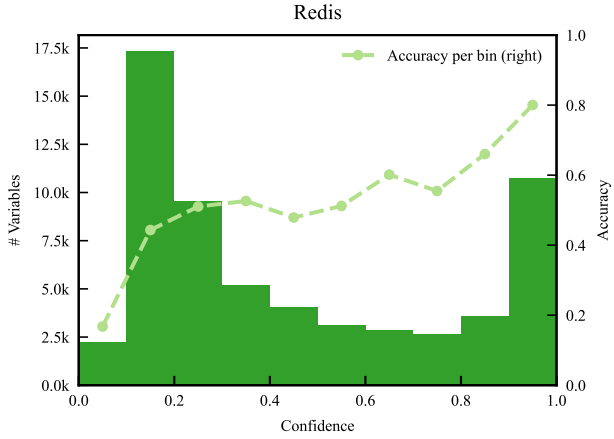
We describe DRAGON, a multi-task GNN model for predicting syntactic simple data types from decompiled ASTs with confidence estimates. We demonstrate that DRAGON performs competitively with TYGR and RESYM, two state-of-the-art binary type inference models. DRAGON is able to generalize well to binary code compiled both for different architectures and optimization levels than the data that was used to train DRAGON. We show that the confidence estimates



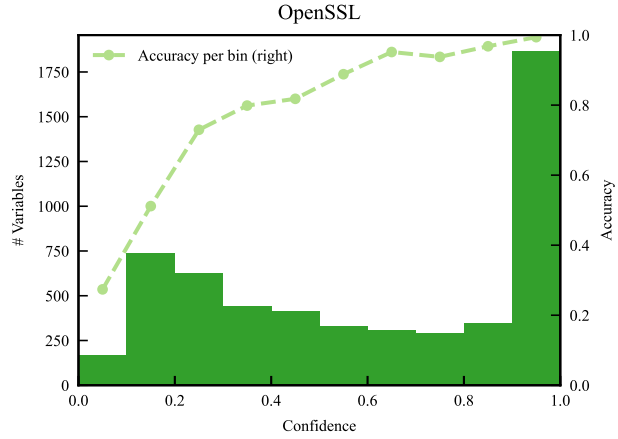
(a) Box plots of confidence estimates sorted by increasing accuracy correlate with corresponding increase in median confidence



(b) Confidence estimate histogram and per-bin accuracy for Apache show a comparable number of predictions in the 10-20% and 90-100% confidence ranges. Per-bin accuracy is correlated with confidence value.



(c) While the dominant percentage of low confidence predictions for Redis is indicative of its low overall accuracy of 54%, the highest confidence bin achieves an accuracy of roughly 80%.



(d) OpenSSL achieves the highest overall accuracy of 84%, consistent with its confidence distribution having the largest proportion of samples in the highest confidence bin.

Fig. 9. Comparison of learned confidence distributions and accuracy on Redis, Apache, and OpenSSL. DRAGON produces a higher ratio of low confidence predictions for benchmarks with lower overall accuracy, yet maintains a strong correlation between predicted confidence value and accuracy in all cases.

returned by DRAGON provide useful information, with higher confidence predictions resulting in an overall higher level of accuracy than those with low confidence.

There are several promising areas for future work. Most obviously, DRAGON could be extended to recover structure definitions. In addition, the confidence estimates offered by DRAGON provide an opportunity for incorporation into other binary analyses that consume type information. This might include things like excluding low confidence type predictions from downstream analysis or using the confidence values as a weighting. We intend to extend DRAGON for structure recovery incorporating the simple type confidence estimates in the near future.

Acknowledgment

This work was made possible in part by a grant of high performance computing resources and technical support from the Alabama Supercomputer Authority. We thank Logan Cannan for his collaboration and feedback from testing and running all the components required to use DRAGON.

We offer our sincere appreciation to the TyGr authors for their assistance and recommendations for effectively running TyGr. Additionally, we express our gratitude for the public availability of TyDa, which we feel is of great benefit to the binary analysis community.

References

[1] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "TIFF: Using Input Type Inference To Improve Fuzzing," in Proceedings of the 34th Annual Computer Security Applications Conference, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, Dec. 2018, pp. 505–517.

[2] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte, "AnnoDomini: From type theory to Year 2000 conversion tool," in Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '99. New York, NY, USA: Association for Computing Machinery, Jan. 1999, pp. 1–14.

[3] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in Network and Distributed System Security Symposium, 2011.
[4] J. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in Proceedings of NDSS'2011, 2011.
[5] T. DeVries and G. W. Taylor, "Learning Confidence for Out-of-Distribution Detection in Neural Networks," Feb. 2018.
[6] X. Wang, X. Xu, Q. Li, M. Yuan, and J. Xue, "Recovering Container Class Types in C++ Binaries," in 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Apr. 2022, pp. 131–143.
[7] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé, M. Naik, Y. Shoshitaishvili, R. Wang, and A. Machiry, "TYGR: Type inference on stripped binaries using graph neural networks," in 33rd USENIX Security Symposium (USENIX Security 24). Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4283–4300.
[8] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "LambdaNet: Probabilistic Type Inference using Graph Neural Networks," Apr. 2020.
[9] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to Represent Programs with Graphs," May 2018.
[10] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to Represent Programs with Heterogeneous Graphs," in 2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC), May 2022, pp. 378–389.
[11] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "DIRE: A Neural Approach to Decompiled Identifier Naming," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), Nov. 2019, pp. 628–639.
[12] "Pytorch Geometric," https://pyg.org//.
[13] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in Proceedings of the 11th Annual Information Security Symposium, ser. Cerias '10. West Lafayette, IN: CERIAS - Purdue University, 2010.
[14] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary," in 2021 IEEE Symposium on Security and Privacy (SP), May 2021, pp. 813–832.
[15] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "TypeMiner: Recovering Types in Binary Programs Using Machine Learning," in Detection of Intrusions and Malware, and Vulnerability Assessment, ser. Lecture Notes in Computer Science, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, pp. 288–308.
[16] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting Debug Information in Stripped Binaries," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 1667–1680.
[17] Z. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in Proceedings of the 26th USENIX Security Symposium, 2017, pp. 99–116.
[18] Y. Lin, D. Gao, and D. Lo, "ReSIL: Revivifying Function Signature Inference using Deep Learning with Domain-Specific Knowledge," in Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy, ser. CODASPY '22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 107–118.
[19] L. Chen, Z. He, and B. Mao, "CATI: Context-Assisted Type Inference from Stripped Binaries," in 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Jun. 2020, pp. 88–98.
[20] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "StateFormer: Fine-grained type recovery from binaries using generative state modeling," in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 690–702.
[21] Q. Chen, J. Lacomis, E. J. Schwartz, C. L. Goues, G. Neubig, and B. Vasilescu, "Augmenting Decompiler Output with Learned Variable Names and Types," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 4327–4343.
[22] H. Green, E. J. Schwartz, C. L. Goues, and B. Vasilescu, "STRIDE: Simple Type Recognition In Decompiled Executables," Jul. 2024.
[23] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries," in Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 4554–4568.
[24] S. Liang, Y. Li, and R. Srikant, "Enhancing the reliability of out-of-distribution image detection in neural networks," arXiv preprint arXiv:1706.02690, 2017.
[25] R. Caruana, "Multitask Learning," Machine Learning, vol. 28, no. 1, pp. 41–75, Jul. 1997.
[26] "Introduction to the Clang AST," https://clang.llvm.org/docs/IntroductionToTheClangAST.html.
[27] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," in International Conference on Learning Representations, Feb. 2018.

APPENDIX

A. Data Type Encoding Implementation

This section provides additional details about the data type encoding, as shown in Figure 2.

The leaf type vector is encoded as a concatenation of all its attributes (e.g. *Category, Size, etc.*) where each attribute is itself one-hot encoded. The proper encoded leaf type form for all non-*BUILTIN* data types is a vector of zeroes.

The pointer levels are interpreted in order, left-to-right. Each level is one of *PTR*, *ARR*, or *LEAF*, indicating whether that level represents a pointer, array, or leaf type respectively. The first occurrence of *LEAF* signals the end of any further *PTR* or *ARR* levels, and we fill the levels (to the right) that follow with *LEAF*. For example, we encode the pointer levels for `FILE*[]` as *ARR, PTR, LEAF*, and `char` as *LEAF, LEAF, LEAF*. We one-hot encode this vector for each of the pointer levels.

Most C data types [5] have an obvious mapping directly to our encoding, like `int32_t` which would be represented as *LEAF, LEAF, LEAF, Category=BUILTIN, Size=4, Signed=True, Floating=False, Boolean=False*. A couple of notable exceptions include `void`, which is indicated by *BUILTIN* type with a size of zero, and function pointers, which do require at least one pointer level (i.e. *PTR, LEAF, LEAF, Category=FUNC, Size=0, Signed=False, Floating=False, Boolean=False*).

In our current implementation that converts between our encoding and a named type, we do not differentiate between 10B and 16B floating point numbers but treat them both as `long double`.

B. Type Projection for TYGR Comparison

This section describes the type projection performed to convert TYGR and DRAGON data types to a common type system. As we do not compare with the structure recovery capability of TYGR, we convert all data types to simple type representations common to both tools.

[5] As we encode type size explicitly, we assume the size of integral types is known or chosen (e.g. `sizeof(int)=8`)

For TYGR, we simply convert all `struct_xyz` type entries into *STRUCT*, since DRAGON does not retype internal structure components.

For DRAGON, we apply the following transformations:

- Convert `uchar` to `char` as TYGR does not represent `uchar`.
- Convert function pointers to `void*`, as TYGR does.
- Convert array element types to `void`, as TYGR does not represent them.
- Convert types with two or more pointer levels to `void**`, as TYGR does.