TyGr: Type Inference on Stripped Binaries using Graph Neural Networks

Chang Zhu[1*], Ziyang Li[2*], Anton Xue[2], Ati Priya Bajaj[1], William Gibbs[1], Yibo Liu[1], Rajeev Alur[2], Tiffany Bao[1], Hanjun Dai[3], Adam Doupé[1], Mayur Naik[2], Yan Shoshitaishvili[1], Ruoyu Wang[1], Aravind Machiry[4]

[1]{czhu62, atipriya, wfgibbs, yliu1049, tbao, doupe, yans, fishw}@asu.edu
[2]{liby99, antonxue, alur, mhnaik}@seas.upenn.edu
[3]hadai@google.com [4]amachiry@purdue.edu

Abstract

Binary type inference is a core research challenge in binary program analysis and reverse engineering. It concerns identifying the data types of registers and memory values in a stripped executable (or object file), whose type information is discarded during compilation. Current methods rely on either manually crafted inference rules, which are brittle and demand significant effort to update, or machine learning-based approaches that suffer from low accuracy.

In this paper we propose TyGr, a graph neural network based solution that encodes data-flow information for inferring both basic and struct variable types in stripped binary programs. To support different architectures and compiler optimizations, TyGr was implemented on top of the ANGR binary analysis platform and uses an architecture-agnostic data-flow analysis to extract a graph-based intra-procedural representation of data-flow information.

We noticed a severe lack of diversity in existing binary executables datasets and created TyDa, a large dataset of diverse binary executables. The sole publicly available dataset, provided by StateFormer, contains only 1% of the total number of functions in TyDa. TyGr is trained and evaluated on a subset of TyDa and generalizes to the rest of the dataset. TyGr demonstrates an overall accuracy of 76.6% and a struct type accuracy of 45.2% on the x64 dataset across four optimization levels (O0-O3). TyGr outperforms existing works by a minimum of 26.1% in overall accuracy and 10.2% in struct accuracy.

1 Introduction

Decompilation, the process of transforming a compiled program to a higher-level language such as C, plays a crucial role in the analysis of computer security threats, as it provides a deep understanding of the behavior of compiled programs. However, decompiled code tends to offer less information compared to the original source code. Many abstractions and constructs such as variable types, comments, and control-flow structures are discarded during compilation, posing a challenge for reverse engineers engaged in program analysis.

Type inference is a major opportunity to significantly improve the output of decompilation [20]. The precise inference of types in binary code has many applications, including binary reverse engineering, malware analysis, vulnerability discovery on binary code [11], software patching, binary rehosting, and other security-critical applications [8, 19, 47]. However, automated and precise binary type inference is challenging [31, 68] due to the lack of high-level abstractions and the rich variety and sophistication of compiler optimizations and hardware architectures [55].

Existing binary type inference solutions can be broadly classified into three categories: (1) Rule- and heuristic-based solutions (e.g., type inference in IDA and Ghidra). (2) constraint-solving-based solutions (e.g., TIE [35], Retypd [42], and OSPREY [70]). (3) machine-learning-based solutions (e.g., Debin [32], StateFormer [45], TypeMiner [38], and DIRTY [10]). Our work is motivated by three key challenges that the state-of-the-art techniques face: (1) *Low accuracy in inferred types on stripped binaries*: Even the best solution only achieves an accuracy of 55.7% during evaluation. (2) *Composite struct type prediction*: Most of the existing solutions either do not predict struct member types or partially support member prediction. (3) *Limited architectural and optimization level support*: Many solutions, especially heuristic-based and constraint-based ones, only support binaries on one or a limited number of architectures and compiler optimization levels. This is because generalizing rules, heuristics, and constraint-solving methods to a diverse set of binaries is difficult.

In this paper, we present TyGr, a machine-learning-based binary type inference technique that assists binary reverse engineers by inferring types for registers and memory locations with a high accuracy. TyGr lifts binary code into VEX IR [41], an intermediate representation (IR) with a wide architecture support, runs a light-weight data-flow analysis on each function to collect information about how each variable

* The first two authors contributed equally to this paper.

is accessed, generates a graph-based representation of data-flow information, and finally trains a model based on Graph Neural Networks (GNN) [50] for type inference.

To strike a balance between scalability and accuracy, TYGR employs a novel graph-based intra-procedural representation of the data-flow information. Because the representation is constructed on a per-function basis, TYGR scales to large, real-world binaries. This representation is acceptable to GNNs, a deep neural network model that is well-suited for predicting rich properties of graph-structured data [61]. To the best of our knowledge, TYGR is the first to demonstrate the effective application of GNNs to the problem of binary type inference. Particularly, we demonstrate that they can adequately tolerate missing inter-procedural information, thereby allowing data-flow analysis to scale.

Our evaluation revealed that existing datasets contain many duplicates. Prior works [44, 56] also made this observation. For instance, recent work by Pal *et al.,* [44] found that 52% of functions in the DIRT dataset are duplicates. Similarly, we found that STATEFORMER dataset has an average of 90% duplicate functions. Such high percentages of duplicates make these datasets unsuitable for properly evaluating learning techniques. To handle this, we created TYDA, a deduplicated binaries dataset built from Gentoo and Debian packages for five architectures (x64, x86, AArch64, Arm32, and Mips) across four optimization levels (O0, O1, O2, and O3).

We implement TYGR on top of the ANGR binary analysis framework [55]. Our evaluation on TYDA dataset shows that TYGR has an overall type prediction accuracy of 76.6% and struct type prediction accuracy of 45.2%—outperforming existing state-of-the-art techniques by 10.2% to 26.1%.

Contributions. This paper makes the following contributions:

- We propose a novel graph-based representation of data-flow information that allows a synergistic combination of a data-flow analysis and a graph neural network model to balance scalability and accuracy.
- We employed innovative techniques to construct a new dataset, incorporating binaries from x64, x86, AArch64, Arm32 and Mips architectures. This dataset is notably larger and more expansive compared to existing datasets.
- We implement TYGR, a system that uses the GNN model that is trained on a large dataset of binary functions to infer types of program variables in unseen functions from stripped binaries.
- We demonstrate the effectiveness of TYGR by extensively evaluating it on a subset of TYDA. TYGR demonstrates an overall accuracy of 76.6 % and a struct type accuracy of 45.2% on the x64 dataset across four optimization levels (O0-O3). TYGR outperforms existing works by a minimum of 26.1% in overall accuracy and 10.2% in struct accuracy.

In the spirit of open science, we release our research artifacts at https://github.com/sefcom/TYGR.

2 Background

Before diving into the technical details of TYGR, we will first present necessary background knowledge for readers on binary reverse engineering, binary type inference, and GNNs.

2.1 Binary Reverse Engineering

Binary reverse engineering is the process of understanding a program without access to, or only having limited access to its source code. Reverse engineers analyze binaries to understand the behaviors or provenance of malware [21, 65], discover vulnerabilities in binaries [39], and mitigate defects in legacy software [53]. In most cases, debug symbols are not available to reverse engineers; They are forced to manually recover lost semantic information, such as variable locations, names, and types, during reverse engineering.

2.2 Type Inference on Binaries

Binary type inference is the automated process of reconstructing source-level type information, e.g., types of local variables and function arguments, from untyped byte-addressed memory locations and registers. It is challenging because most information is discarded during compilation unless debug symbols are preserved. As shown in Table 1, existing binary type inference solutions can be broadly classified into three categories based on their core techniques: (1) Rule- and heuristic-based type inference solutions, (2) constraint-solving-based solutions, (3) machine-learning-based solutions.

A key difference between these solutions is if they support type inference of structs and struct members (or struct layouts). Inferring struct members and their types requires complex and accurate reasoning and fine-grained flow information [8], which is hard to gain during static analysis. Most non-constraint-based inference techniques (i.e., top and bottom row groups) do not predict struct member types. REWARDS [37] and HOWARD [57], which do predict struct member types, use dynamic traces to get precise offset information. However, as with any dynamic techniques, they suffer from low completeness: they only support assembly code that is reachable during execution. Therefore, we make a design decision for TYGR to not use dynamic traces and not infer struct members.

2.3 Graph Neural Networks

Graph Neural Network (GNN) is a deep neural network architecture that is well-suited for predicting rich properties of

Table 1: A qualitative comparison among existing binary type inference techniques. All techniques support inferencing primitive types, which are omitted in this table. "Struct," "Struct Ptrs," and "Struct Members" refer to whether each technique can *automatically* infer such information. IDA and Ghidra only support manually specifying struct types to variables and perform extremely limited automated type inference of structs. TypeMiner does not attempt to recover the complete struct layout or types of all struct members. DIRTY only predicts types in its vocabulary and does not support predicting structs that did not appear in its training set.

Category	Technique	Input	Completeness	Struct Types Inference Support			Multi-arch. Support
				Struct	Struct Ptrs	Struct Members	
Rules and Heuristics	IDA [33]	Binary	High	✗	✗	✗	✓
	GHIDRA [1]	Binary	High	✗	✗	✗	✓
	REWARDS [37]	Dyn. Traces	Low	✓	✓	✓	✗
	HOWARD [57]	Dyn. Traces	Low	✓	✓	✓	✗
Type Constraint Solving	TIE [35]	Binary	High	✓	✓	✓	✗
	RETYPD [42]	Binary	High	✓	✓	✓	✗
	OSPREY [70]	Binary	High	✓	✓	✓	✗
Machine Learning	DEBIN [32]	Disassembly	High	✓	✗	✗	✗
	TYPEMINER [38]	Dyn. Traces	Low	✓	✓	✗	✗
	STATEFORMER [45]	Runtime Values	High	✓	✓	✗	✓
	DIRTY [10]	Decompilation	High	✓	✓	✗	✗
	TYGR	Binary	High	✓	✓	✓	✓

graph-structured data [61], through a procedure called *message passing*. GNNs have been used for many program understanding and analysis tasks, such as identifying variable misuses in C# programs [4], localizing and repairing bugs in JavaScript code [18], predicting types in Python programs [3], and detecting code clones [62]. To the best of our knowledge, TYGR is the first to demonstrate the effective application of GNNs to the problem of binary type inference.

3 Overview

In this section, we first define the binary type inference problem. We then provide an overview of TYGR's architecture, highlighting key design choices that enable it to achieve precise and scalable type inference on binary code.

3.1 Binary Type Inference

We consider the problem of mapping binary-level variables to source-level types. Specifically, we focus on function parameters and local variables, which are crucial for understanding the behavior and intent of the function—and are thus of interest to reverse engineering. We illustrate our objective with an example in Figure 1, which shows a C function and its assembly code extracted from an x64 binary compiled with GCC using optimization level O0. In practice, only the binary is available, but inferring types for data that directly correspond to source-level variables is helpful for understanding the intent of the function, and perhaps even extracting a faithful decompilation. At the binary level, local variables are typically stored at stack offsets. For instance, the stack offset `-0x30(%rbp)` corresponds to `name_len` and `-0x38(%rbp)` corresponds to `ext_len`.

Our goal is to predict fine-grained type information in the form of C types, such as `int32`, `uint64`, and `struct*`. These types are familiar to reverse engineers with experience in popular reverse engineering tools (e.g., IDA). We treat type inference as a classification problem and use a fixed subset of primitive types. While C types may be arbitrarily complex, our finite subset is expressive enough to cover over 97.1% types that arise in our large dataset.

It is worth noting that existing machine learning-based type inference techniques are all type *prediction* techniques. The crucial difference between type inference and type prediction is that type prediction techniques only predict types that are in the vocabulary while type inference may output new types that are not in the vocabulary. This difference matters most in inferring struct types, where existing machine learning-based solutions fail to predict struct shapes or types for struct members (DEBIN and STATEFORMER), or can only predict known struct types (DIRTY), which severely limits their use in revese engineering tasks. TYGR supports predicting struct shapes and member types, which essentially makes TYGR a type inference technique.

3.2 TYGR Architecture

Figure 2 shows an overview of TYGR. A key goal of TYGR is architecture independence over the input binary. Therefore, TYGR uses VEX IR [41], an architecture-agnostic IR for binary code in many different architectures.

Importance of Data-flow Information. Data-flow information is highly relevant for type inference. This is evident in traditional constraint-based type inference techniques wherein the typing constraints encode such information [8, 35, 42]. However, these methods are often limited by the constraint-solving step, which prevents them from effectively scaling to

```
int file_has_ext(char* file_name, char* file_ext) {
  char* ext = file_ext;
  if (*file_name) {
    while (*ext) {
      int name_len = strlen(file_name);
      int ext_len = strlen(ext);
      if (name_len >= ext_len) {
        char* a = file_name + name_len - ext_len;
        char* b = ext;
        while (*a && toupper(*a++) == toupper(*b++));
        if (!*a) return 1;
      }
      ext += ext_len + 1;
    }
  }
  return 0;
}
```

```
...
53:  mov    -0x30(%rbp),%eax
56:  movslq %eax,%rdx
59:  mov    -0x2c(%rbp),%eax
5c:  cltq
5e:  sub    %rax,%rdx
61:  mov    -0x38(%rbp),%rax
65:  add    %rdx,%rax
68:  mov    %rax,-0x20(%rbp)
6c:  mov    -0x28(%rbp),%rax
70:  mov    %rax,-0x18(%rbp)
74:  nop
...
```

```
-0x18 (%rbp): char*
-0x20 (%rbp): char*
-0x28 (%rbp): char*
-0x2c (%rbp): int32
-0x30 (%rbp): int32
-0x38 (%rbp): char*
-0x40 (%rbp): char*
```

Figure 1: Top: A C function that checks file extensions. Bottom left: The disassembly abstract of the function in compiled x64 binary. Bottom right: Type predictions for variables at their corresponding stack offsets.

large binary applications. An attractive work-around is to employ machine learning. TYGR thereby uses a model to learn the data-flow patterns in binaries and outputs predicted types. To integrate classic data-flow analysis and modern machine learning, we must design a representation for typing information that is both easy to extract and suitable for machine learning.

Representing Data-flow Information. Our key insight is to design a graph-based intra-procedural representation of data-flow information. First, constraint-encoded data-flow information is also naturally modeled through graphs, and in fact light-weight data-flow graphs are easy and efficient to acquire using ANGR. Moreover, modern graph neural networks (GNNs) are remarkably well-suited to learning and approximating the latent semantics of graph-structured data. This motivates the central data structure of TYGR, which is an efficiently constructed and information-rich graph that explicitly marks the derivation, usage, and location of data-flow throughout program execution. In short, we use ANGR to generate function-level data-flow graphs that are fed to a graph neural network. The graph neural network then generates a continuous embedding of the data-flow graphs that approximate the underlying typing semantics.

Inference as Classification. We model type inference as a classification problem [48], in which we classify an entity as one of the finite C-level types. Although the possible types are, in principle, arbitrarily many, we observe that selecting a much smaller range of commonly seen types already encompasses a large portion of those that exist in the wild. Therefore, rather than incorporating the full complexity of structured prediction, the formulation of type inference as a classification problem suffices for binaries.

TYGR's output is a mapping of binary-level variables to their respective C-level types. It is easily interpretable as this closely matches the type systems of popular tools like IDA and GHIDRA, and is therefore also in a format that is easy to integrate with existing analysis loops.

4 Methodology

In this section we present TYGR's approach to binary type inference as a machine learning problem.

4.1 The VEX IR

TYGR first lifts the input binary function into VEX IR using the ANGR binary analysis framework [55]. Figure 3 shows how TYGR converts a few lines of x64 binary assembly into their corresponding VEX statements. The core semantics of VEX IR center around accesses (reads and writes) to registers, memory locations, and temporary variables (e.g., `t2` and `t6`). Temporary variables are a VEX-specific convention to enforce static-single assignment form [15].

4.2 Data-Flow Analysis

We next discuss how TYGR uses data-flow analysis to generate graphs that capture the relevant information for type inference. Figure 4 shows the function `foo` and its control-flow graph (CFG). Depending on the parameter `a`, either path P_1 or P_2 will be taken, which will appropriately modify the values of the two local stack variables `b` and `p`.

Our goal is to infer the types of `a`, `b`, `p`. To do this we aim to generate information-rich data-flow graphs as shown in Figure 5. These graphs convey how variables derive and use data during execution, and capture information for a GNN to infer types. As a high-level overview, our strategy for data-flow analysis comprises two steps:

Step 1. Perform program execution along different non-cyclic paths in the CFG of a function to generate a data-flow graph for each variable along each path. Each path is then associated with a collection of variable-level data-flow graphs (Figure 5, left).

Step 2. Aggregate the variable-level data-flow graphs of both paths into a function-level data-flow graph (Figure 5, right). This in turn is passed to the training stage of TYGR's pipeline.

4.2.1 Exploring the Control-Flow Graph

TYGR explores all nodes and edges in the CFG of a function and inspects the read-from and written-to locations in each IR

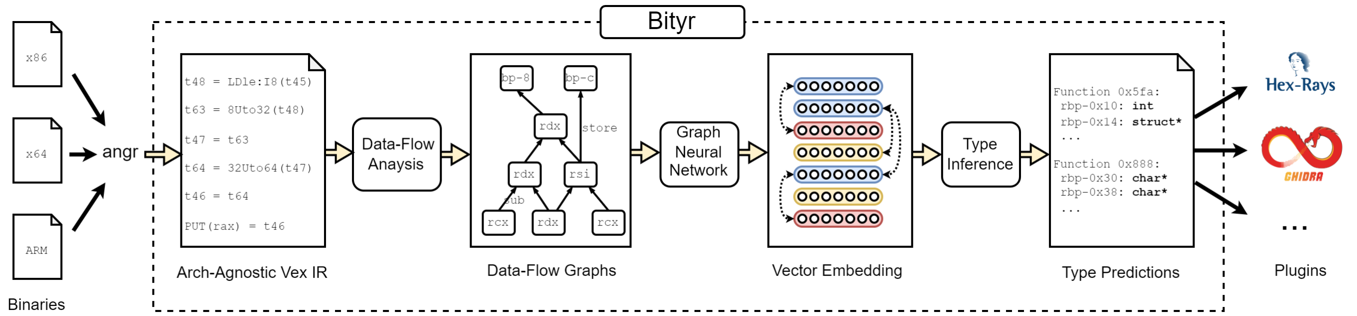

Figure 2: The pipeline of TYGR. Binaries are first converted to VEX IR followed by data-flow analysis to yield a data-flow graph for each function. Each such graph is then passed to a graph neural network which yields a continuous representation that aims to capture the semantic information of the function. This representation is then used to predict a type for each variable present.

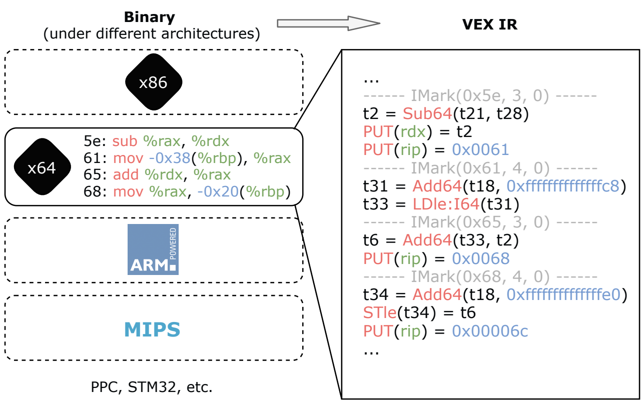

Figure 3: Binary to VEX IR conversion for offset 0x5e–0x68 of the function in Figure 1. Note that all architecture-specific side effects (e.g., changing `rip` in x64) are explicitly encoded in VEX IR.

statement. During the exploration, TYGR extracts *variable-level data-flow graphs* that captures the value at each location.

Our key insight is that it suffices to evaluate each path *once*. This is because how the binary code in a block uses data does not change when it runs for more than once. Another insight is that we can completely disregard the feasibility of each branch and forcibly explore both branches of each branch condition. We are only interested in *how* the binary code uses data at each location and not under what condition each block is reached. Infeasible paths still contain value information regarding how binary code accesses data locations.

Using our insights, we design a function exploration algorithm for TYGR that executes the blocks in a function following a topological order starting from the entry point. The sequence of nodes that TYGR visits induces a set of simple paths. TYGR collects how each data location is accessed along each simple path.

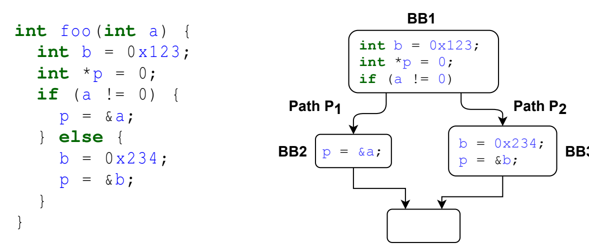

Figure 4: An example illustrating the data-flow analysis in TYGR. (Left) A simple function with two possible paths. (Right) The control-flow graph of the function.

4.2.2 Data-Flow Graphs

We derive data-flow graphs from the information that TYGR collects during function CFG exploration. By examining the read-from or written-to locations along each simple path, we obtain a set of symbolic expressions. Then TYGR uses these expressions to derive the variable-level data-flow graphs as shown in Figure 5 (left). The nodes in these data-flow graphs are constant values (constant bitvector expressions). They correspond to either *immediate operands* (e.g., constants and register offsets) in VEX expressions (e.g., arguments to some VEX operations) or *computed values* (e.g., to-be-written values) that result from some VEX operation. VEX operations include arithmetic operations (e.g., addition and multiplication) and data-access operations (e.g., register-reads and memory-writes). TYGR uses *edge labels* to mark how each node is used: `Addr` means a node is used as the address of a data-access operation; `Value` means a node is used as the value of a data-access operation; `Op1` and `Op2` mean a node is used as the first and second operand of an arithmetic operation, respectively; and `RegID` means a node is used as the "register offset" (which corresponds to a register name) of a register-access operation.

Variable-level data-flow graphs describe how a particular expression at a particular location along a particular path is derived. They do not convey how the value of a variable

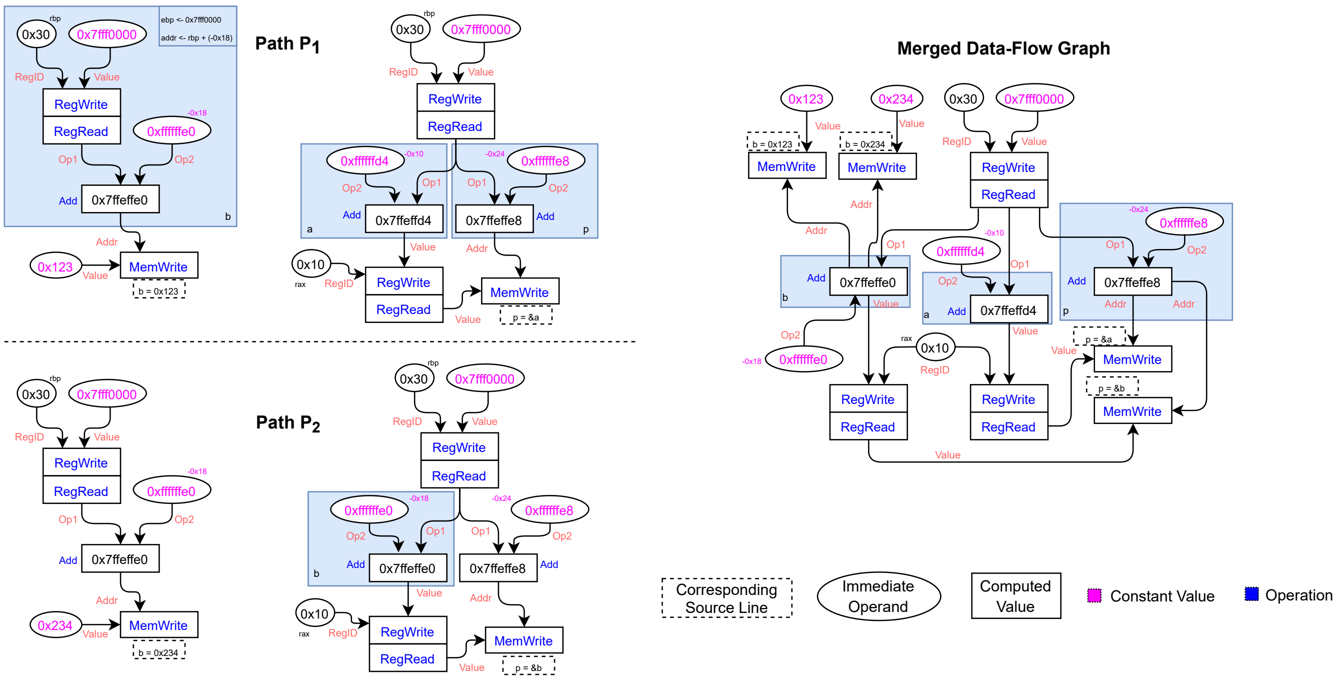

Figure 5: Data-flow graphs for the function `foo` in Figure 4. (Left-Top) variable-level data-flow graphs for `b` and `p` along P_1. (Left-Bottom) variable-level data-flow graphs for `b` and `p` along P_2. Note that because `b` was over-written, it has the value `0x234` rather than `0x123`. (Right) aggregation of all data-flow graphs. Above is a simplified view; data-flow graphs track the operands, operators, bitsizes, and locations of data derivation and usage.

is used by other variables throughout the function. Indeed, different variable-level data-flow graphs may share identical sub-graphs, and inter-variable data-flows give additional information about the type of an expression. Therefore, TYGR aggregates all variable-level data-flow graphs into a single function-level data-flow graph using a graph-union operator, as shown in Figure 5 (right).

4.3 Type Inference with GNNs

We deliberately chose to use a GNN to infer variable types because a GNN explicitly captures the access patterns of variables through edges representing operations and nodes representing variable locations. Such access patterns are only implicitly captured in textual NN structures that other ML-based solutions employ.

Given a graph $G = (V, E)$ that contains a set of nodes V and edges E, a GNN f would embed the graph into a set of vectors (or embeddings), i.e., $f(G) : \mathcal{G} \mapsto \mathbb{R}^{|V| \times d}$. Here $\mathcal{G}$ represents the space of the graphs, while d specifies the dimensionality of the embedding per each node. As shown in Figure 6, the GNN encodes the graph in an iterative fashion, where each iteration or layer of GNN propagates the information from nodes to their direct neighbors. We next elaborate the specific design choices of f.

Node embedding initialization. The first layer of the GNN starts with the initial embedding representation of each node $h_v^{(0)}, \forall v \in V$. In our setting, we represent the node with the following simple features (Figure 6):

- Bitvector expression sizes: one-hot encoding of the size of the node value, from the set of possible sizes $\{1, 8, 16, 32, 64, 128, \text{others}\}$.
- Five register related features, including *is_register*, *is_arg_register*, and *is_ret_register*.
- 11 value features related to the concrete node value, e.g., *is_bool*, *is_float*, *close_to_stack_pointer*, *is_zero*, *is_negative*, and *is_one*.

We denote the above features as $x_v \in \mathbb{R}^D$ where D is the dimension of the features. Then, the initial embedding is $h_v^{(0)} = W_0 x_v + b_0$ where $W_0 \in \mathbb{R}^{d \times D}$ and $b_0 \in \mathbb{R}^d$ are learnable parameters.

Edge type. In our setting, each edge $e \in E$ is a triplet $e = (u, r, v)$ that represents a directional edge of type r from node u to v. An edge type represents the data-flow, control-flow, and other operational meanings in pre-defined types $\mathcal{R}$. For a GNN to function properly, one would need to create a backward edge for any forward edge in the original data-flow graph. The backward edge type must be different from the forward edge type. Therefore for any edge type r, we have an

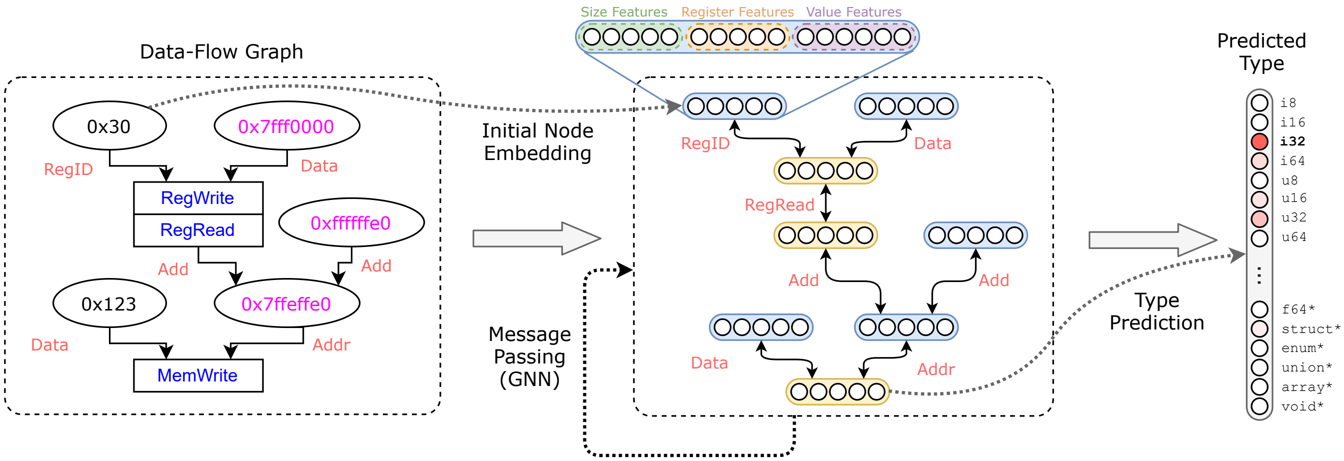

Figure 6: The architecture of the graph neural network in TYGR. The data-flow graph (left) is converted into the vector representation (middle). The nodes in the data-flow graph is transformed into node embedding using Node Embedding Initialization. The directed edges are augmented to allow message passing on both forward and backward directions. After message passing, the node embedding is passed to the type prediction layer to produce the type vector (right).

edge type $\text{rev}(r) \in \mathcal{R}$ representing its backward edge type. As the total number of possible types $|\mathcal{R}|$ is known beforehand, we can design the message passing operator based on the edge types, as described next.

Message passing layer. Each layer of GNN f performs a "message passing" operation that propagates the information from the nodes to their direct neighbors. We denote the embedding of node v at layer l as $h_v^{(l)}$, with the boundary case of $h_v^{(0)}$ defined above, and the update formula defined recursively as follows:

$$h_v^{(l)} = \sigma\Big(\text{AGGREGATE}(\{g(h_u^{(l-1)}, r, h_v^{(l-1)})\}_{e=(u,r,v)\in\mathcal{N}_v})\Big) \tag{1}$$

Here σ is an activation function such as ReLU or Sigmoid. AGGREGATE is a pooling function that aggregates the set of embeddings into a single vector. $\mathcal{N}_v$ denotes all incoming edges to node v, while the function $g(u,r,v)$ is the message function that produces an embedding. We adopt the design choice from RGCN [51], and realize Equation 1 as follows:

$$h_v^{(l)} = \text{ReLU}\Big(\sum_{r\in\mathcal{R}} \text{MEAN}(\{W_r^{(l)} h_u^{(l-1)} + W_0^{(l)} h_v^{(l-1)}\}_{e\in\mathcal{N}_v^r})\Big) \tag{2}$$

where $W_r^{(l)} \in \mathbb{R}^{d\times d}$ are weights that depend on layer index l and edge type r, and $W_0^{(l)} \in \mathbb{R}^{d\times d}$. $\mathcal{N}_v^r \subseteq \mathcal{N}_v$ denotes the incoming edges to node v with edge type r.

After L layers, we use the output of the last layer as the vector representation for each node, $h_v = h_v^L$, and this vector is used for label prediction, as described next.

Type prediction. After obtaining the embedding h_v for a particular node v, we use a multi-layer perceptron (MLP) to classify h_v into the node label, which is the type corresponding to the node. Given a set of types T, our type prediction layer produces a vector $t_v \in \mathbb{R}^{|T|}$ for the node v, as shown as the right most vector in Figure 6. During training, our predicted type vector t_v is then compared with the ground truth type vector $\hat{t}_v \in \mathbb{R}^{|T|}$, the one-hot encoding of the ground truth type under the set of types T. In this work, we apply cross entropy loss function

$$\mathcal{L}(y,\hat{y}) = -\sum_i \hat{y}_i \log(y_i) + (1-\hat{y}_i)\log(1-y_i)$$

to compute the loss $l = \mathcal{L}(t_v, \hat{t}_v)$. The loss l is then back-propagated to update the learnable parameters. During testing and prediction phases, we apply argmax on t_v to obtain the type that is predicted to have the highest probability. While we predict types for all nodes in the graph, during both training and testing, because we know the mapping from source-level variables to graph nodes, we only compare to ground truth the predicted types of the nodes that correspond to source-level variables.

4.4 Type Inference for Structs

Inferring the shape and member types for structs is challenging. Existing approaches, such as OSPREY and DIRTY, either fail to infer types for struct members or use a common type (`struct`) for all struct members. Figure 7 shows a simple C function that involves several operations for one struct variable. OSPREY infers the type of `quoting_options` as `struct<4, 1, 8>` but does not predict struct member types (inferring member sizes only). DIRTY cannot predict types that are not part of the training set. For this function, DIRTY predicts the same incor-

```c
struct options{
    int flag;
    char buffer_type;
    char* name;
};

int set_buffer(struct options default_options, int
     s_flag, char s_type)
{
    struct options quoting_options;
    quoting_options = default_option;
    quoting_options.flag = s_flag;
    quoting_options.buffer_type = s_type;

    int qsize = buffer_restyled(quoting_options);
    return qsize;
}
```

Figure 7: A simple C function that sets a buffer.

rect type `struct{int, char, char*}` for all members of `quoting_options`.

The unique GNN design allows TYGR to tackle the aforementioned limitations. We divide the approach of inferring struct types into two steps. First, we infer struct shapes: For each location, TYGR predicts whether it stores a struct. Second, we infer types for members inside each struct: After knowing a location belongs to a struct type in the first step, TYGR further predicts its type.

During training, suppose TYGR takes the function in Figure 7 as input, every location related to `quoting_options` (e.g., `quoting_options.flag` and `quoting_options.buffer_type`) are first marked as `struct` along with all other basic types for training to obtain $\text{MODEL}_{\text{Base}}$. Then for the second step, we train a new model $\text{MODEL}_{\text{Struct}}$ for predicting struct member types, where we mask all variables of primitive types (base types and pointer types in Figure 9) and only keep struct members in training data. Ideally, this new model will predict `quoting_options.flag` as `int`.

During type inference, for each location $\text{MODEL}_{\text{Base}}$ inferred as `struct`, we further use $\text{MODEL}_{\text{Struct}}$ to infer their struct-member-specific types. If correctly inferred, the model output for `quoting_options.flag` will be `int_S` suggesting it is an `int` type and also a struct member. We derive the final struct accuracy by multiplying the struct accuracy of $\text{MODEL}_{\text{Base}}$ with the overall accuracy of $\text{MODEL}_{\text{Struct}}$.

5 Building the Data Set

When training TYGR on existing data sets that the state-of-the-art solution uses, we found issues that would impact the reliability of type inference. In this section, we first briefly discuss these issues, and then detail how we build our data set, TYDA, for training and evaluating TYGR.

5.1 Shortcomings with Prior Data Sets

An essential component for training and evaluating any machine learning model is a high-quality data set that is both diverse and accurately reflects the task at hand. Unfortunately, the binary data sets from previous studies are plagued by various significant limitations. They are either inaccessible to the public or contain excessive duplicates. The sole publicly available data set, provided by STATEFORMER, unfortunately contains a substantial amount of duplicated functions and only 1% of the total number of functions in TYDA. Table 2 shows detailed statistics of duplicates on the STATEFORMER x64 data set. Figure 8 shows the number of occurrences for each unique functions on a logarithmic scale. On taking a closer look at STATEFORMER binaries, we found that many are built from different versions of the same source package, *e.g.,* `coreutils1.0` and `coreutils2.0`. Furthermore, several source packages produce multiple binaries with only a minor difference. For instance, `binutils` produces 25 `addr2line` binaries. Each of these binaries is slightly customized to handle ELF files from different architectures. These contribute to a high duplication rate in the training set, which may skew the model's learning and bias the outcome [2]. Additionally, significant duplicates may result in substantial overlap in the training and testing sets; while it may appear that the model is generalizing, it might actually be memorizing [36].

To determine the function duplication rate, we hashed the disassembly of every function in the data set after unifying instruction pointer-relative offsets and immediates that fell within the boundary of each binary's addres space. This is an over-approximation as some non-address referencing immediates will be sanitized as well as instruction pointer relative offsets that point to differing locations. On average 89.9% of functions consist of duplicates, potentially biasing a model's training by overemphasizing these repeated functions. Even after deduplicating these functions, the remaining unique functions may be insufficient to adequately represent real-world binaries. Recent work [44] also shows that the data set used by DIRTY (another state-of-the-art) inference work has 56.9% duplicate functions and 65.5% overlap in their training and test sets. Such high rates of duplicate functions will lead to inflated prediction accuracy. We argue that there is an imperative necessity for the construction of a more extensive, thorough data set to facilitate a faithful evaluation of type inference techniques.

5.2 Building The Data Set

We collected C packages from Gentoo and Debian repositories. For compiling Gentoo packages, we utilized the tool used in VarBERT [44] and extended the tool for multi-arch support. We compiled packages from Gentoo for x86 and x64 for four compiler optimization levels, *i.e.,* O0 (no optimization), O1,

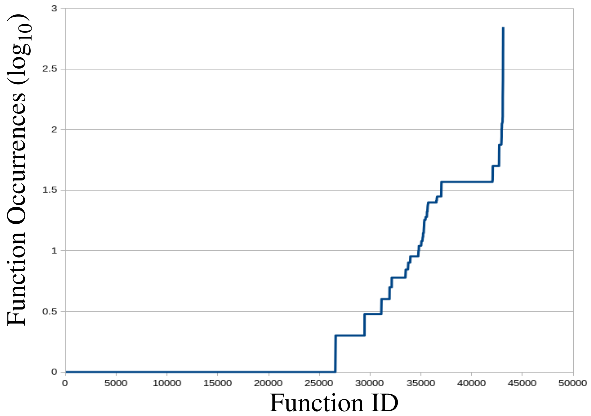

Figure 8: A logarithmic graph of the number of occurrences for each unique function in the STATEFORMER x64 O0 dataset.

Arch.	Opt. Level	# Unique Functions	# Functions	#Dup Rate(%)
x64	O0	43,116	337,608	88.7
	O1	37,507	330,926	89.8
	O2	35,987	335,056	90.3
	O3	34,009	333,076	90.7

Table 2: Statistics of unique functions in the STATEFORMER data set.

O2, and O3, with debug symbols (-g) preserved. Similarly, we compiled C Debian packages for AArch64, Arm32, Mips for four optimizations using QEMU. In total, we built 163,643 binaries, with 83,020 x86, 47,106 x64, 14,885 AArch64, 14,524 Arm32, and 4,107 Mips binaries. We randomly selected approximately 8% binaries from the TYDA data set to train, develop, and evaluate TYGR due to constraints in the available computing resources that we use for this research project. For ease of reference, we refer to this subset of TYDA as TYDA_{MIN}. Due to the smaller size of the Mips data set, we use all Mips binaries. Note that TYDA_{MIN} still contains significantly more binaries and unique functions than the data set that STATEFORMER uses. Table 3 shows the statistics of TYDA_{MIN} and TYDA. We provide function sizes, in terms of instruction counts, in Appendix A.1.

We deduplicated all the functions in TYDA_{MIN} before creating train, test, and validation splits. We disassemble all function bodies, unify all instruction pointer-relevant offsets and intermediates that fall within the address space of the corresponding binary, and get sanitized function body disassembly. We then hashed the disassembly using SHA256 and removed functions with duplicate hashes. Lastly, we introduce these deduplicated functions in TYDA_{MIN} binaries into the TYGR pipeline.

Arch.	Opt. Level	# Functions		Dup. Rate
		TYDA_{MIN}	TYDA	
x64	O0	543,101	12,639,052	53.6%
	O1	534,181	14,876,300	54.9%
	O2	540,132	13,865,615	54.3%
	O3	524,611	19,601,677	52.0%
x86	O0	332,644	11,079,340	13.8%
	O1	379,106	9,834,631	22.4%
	O2	392,195	11,291,892	22.9%
	O3	371,174	11,627,507	25.0%
AArch64	O0	131,984	958,931	23.6%
	O1	133,100	1,936,300	22.1%
	O2	140,214	2,041,829	24.3%
	O3	127,610	526,949	22.6%
Arm32	O0	112,714	711,992	42.6%
	O1	116,506	9,008,051	43.9%
	O2	103,791	3,534,106	45.3%
	O3	106,085	5,594,100	41.8%
Mips	O0	-	158,897	28.3%
	O1	-	178,279	32.0%
	O2	-	184,929	30.7%
	O3	-	164,777	32.2%

Table 3: Numbers of functions for both TYDA_{MIN} and TYDA.

base type ::= `i8` | `i16` | `i32` | `i64` | `i128` |
`u8` | `u16` | `u32` | `u64` | `u128` |
`bool` | `char` | `union` | `enum` | `array`

pointer type ::= *base type** | `void*`

struct member type ::= *base type*_`S` | *pointer type*_`S`

Figure 9: All types that TYGR can predict.

6 Implementation

TYGR comprises 7k lines of Python code. The data-flow analysis module is based on the ANGR framework [55]. The learning module is written using PyTorch Geometric library of PyTorch 1.8.1. Figure 9 shows all output types that TYGR supports, which covers 97.1% of all observed types in the data set. We convert types that TYGR does not support to the closest type. For example, `struct***` is cast to `void*`.

7 Evaluation

Our evaluation aims to answer the following questions:

RQ1 *(Effectiveness)* How accurate is TYGR's type inference on real-world binaries?

RQ2 *(Comparative Evaluation)* How does TYGR compare to existing binary type inference techniques?

RQ3 *(Efficiency)* How efficient is TYGR's type inference engine?

Arch	Opt. Level	Overall Acc.	Struct Acc.
x64	O0	81.8	46.7
	O1	76.0	42.9
	O2	75.7	50.4
	O3	72.8	41.0

Table 4: Overall accuracy and struct accuracy of TYGR on the x64 TYDA$_{\text{MIN}}$ dataset.

Arch	Opt. Level	Precision %	Recall %	F1
x64	O0	82.8	82.2	82.5
	O1	79.2	77.3	78.2
	O2	78.4	76.9	77.7
	O3	76.0	73.7	74.8
x86	O0	76.9	75.5	76.2
	O1	61.6	60.4	61.0
	O2	58.8	57.5	58.1
	O3	61.4	60.1	60.8
AArch64	O0	82.0	81.3	81.7
	O1	77.7	77.0	77.3
	O2	66.3	65.1	65.7
	O3	74.2	73.1	73.7
Arm32	O0	76.7	76.0	76.3
	O1	60.6	58.9	59.7
	O2	57.1	56.5	56.8
	O3	59.3	58.0	58.6
Mips	O0	57.1	56.5	56.8
	O1	47.1	46.0	46.5
	O2	43.7	43.4	43.6
	O3	45.0	44.2	44.6

Table 5: The precision, recall and F1 scores of TYGR on different architectures and optimization levels.

Training Setup. We use the Adam optimizer with an initial learning rate of 10^{-3} and a batch size of 32. We train our model end-to-end using 35 epochs and pick the model with the lowest validation loss. The expected training time is about 50 hours on average for each architecture. We use ReLU as the activation function during message passing and the type prediction. Finally, our GNN is configured to have eight ($L = 8$) message passing layers, with latent dimension $d = 64$. For every architecture-optimization combination, we adhere to common practices by employing an 8:1:1 split ratio for training, validation, and testing.

Machine Setup. All the experiments are run on a Linux server with Ubuntu 20.04, Intel Xeon Gold 5218 at 2.30GHz with 64 cores, 251GB of RAM, and two NVIDIA GeForce RTX 3090-Ti GPUs.

7.1 RQ1: Type Inference Performance

We first evaluate the performance of TYGR on TYDA$_{\text{MIN}}$. We train and test TYGR for each combination of architecture and optimization level (e.g. x64-O0).

Overall performance. Existing work uses different performance metrics. For example, DIRTY and OSPREY use accuracy while STATEFORMER uses precision, recall, and F1 scores. Therefore, we show the performance of TYGR using both metrics.

Table 4 shows the accuracy of TYGR on x64. TYGR achieves an average overall accuracy of 76.6% and an average overall struct accuracy of 45.2%. Table 5 shows the precision, recall, and F1 scores. TYGR achieves an average F1 score of 65.5%. Compared to other optimization levels, TYGR performs best on O0. We believe that as optimization levels increase, more variables are eliminated during compilation, leading to reduced information that can be encoded into data flow graphs. For example, in the O0 dataset, the average number of edges and nodes per graph nearly triples that of the O2 dataset, resulting in comparatively inferior type inference performance.

However, it seems that increasing optimization levels does not always led to less performance. Upon closer examination of the O2 and O3 datasets, we discovered that the average number of edges and nodes generated per graph for O3 *increased* compared to O2. We believe that this is because functions are inlined (and thus optimized away) when compiling in O3, and this is what improves performance (for O3 compared to O2). We observed the same trend for STATEFORMER on TYDA$_{\text{MIN}}$ (and their datasets). To contextualize this, we provide the average number of edges and nodes generated per graph in Appendix A.3.

Inference accuracy per type. Table 6 illustrates how inference accuracy varies across different types. In general, the inference accuracy of each type is relatively high (between 74.7% to 91.1%) until the very bottom of the table, where the inference accuracy for `i16` is only 55.0%. This shows (a) with a sufficient number of samples of a specific type, TYGR can easily achieve a high accuracy in inferring that type, and (b) TYGR needs more samples to make precise inference for types that do not appear frequently enough in the dataset.

Types of return variables. Although TYGR does not infer function prototypes, knowing the types of variables that a function returns can act as a secondary source of information when recovering function prototypes. We take a deeper look into the prediction performance of TYGR on return variables of all functions in the x64-O0 split of TYDA$_{\text{MIN}}$. TYGR achieves a prediction accuracy of 81.3%, which conforms with the overall accuracy of x64-O0 in Table 4.

Generalizability. To evaluate the generalizability of TYGR, we test TYGR on randomly selected functions that are in TYDA but not in TYDA$_{\text{MIN}}$. For each optimization level, we randomly select 40k functions for x64 and x86 and 13k functions for AArch64 and Arm32 to test. There is no test on Mips as we are using all binaries from TYDA Mips. Table 7 shows the results. TYGR demonstrates consistent performance for unseen data.

Type	Percentage (%)	Accuracy (%)
`Pointer Types`	52.8	80.0
`struct*`	25.6	91.1
`i32`	23.1	90.9
`char*`	16.3	74.7
`u64`	7.0	84.7
`u32`	5.9	80.0
`bool`	1.7	83.9
`char`	1.4	92.0
Omitted	...	...
`f64*`	0.3	48.5
`i16`	0.1	55.0

Table 6: Distribution and inference precision for variables whose types present more than 0.1% of the x64-O0 TyDA$_{\text{MIN}}$ dataset. "Pointer Types" refers to the collection of all pointer types.

Arch	Opt. Level	Precision	Recall	F1
x64	O0	81.1	80.2	80.6
	O1	77.6	76.0	76.8
	O2	77.4	76.0	76.7
	O3	74.7	73.1	73.9
x86	O0	77.4	75.7	76.5
	O1	59.1	60.4	61.0
	O2	58.3	57.1	57.7
	O3	61.0	60.0	60.5
AArch64	O0	81.2	80.6	80.9
	O1	74.2	73.6	73.9
	O2	63.5	62.8	63.1
	O3	73.7	72.6	73.1
Arm32	O0	75.4	74.7	75.1
	O1	59.2	57.9	58.5
	O2	58.1	57.6	57.9
	O3	58.5	57.9	58.2

Table 7: Test of TyGr on the functions that are not from TyDA$_{\text{MIN}}$.

7.2 RQ2: Comparison Against Baselines

We compare TyGr against state-of-the-art binary type inference techniques that are publicly available: Dirty [10], Osprey [70], and StateFormer [45]. We omit comparisons against commercial tools (Ghidra and IDA Pro) because Osprey outperforms both.

7.2.1 Comparison against Dirty and Osprey

To ensure a fair evaluation, we only compare against the configurations for which each tool was originally designed and evaluated (e.g., Dirty and Osprey only support x64 binaries).

While Osprey is not publicly available, its authors provide results of Osprey on x64-O0 binaries of GNU Coreutils. For a fair comparison, we remove the Coreutils functions that

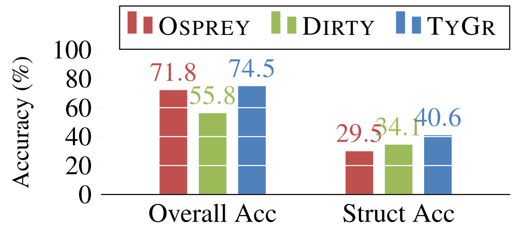

Figure 10: Accuracy results on GNU coreutils O0 executables.

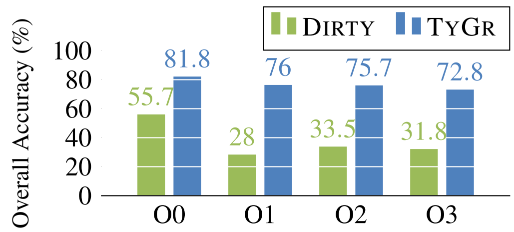

Figure 11: Overall accuracy of TyGr and Dirty on the TyDA$_{\text{MIN}}$ x64 dataset.

are within TyGr's training set. Because the authors compared Osprey against Dirty in their paper, we also evaluate Dirty on the same set of binaries. Osprey predicts only a few primitive and complex types (e.g., `Primitive_1`, which represents a primitive type that takes one byte in memory), so we post-process the prediction results of Dirty and TyGr into the types that Osprey supports. For example, we convert both `bool` and `char` to `Primitive_1`, and `const char *` and `char *` to `Pointer`.

Figure 10 shows the prediction accuracy of both overall types and only struct types for Dirty, Osprey, and TyGr. TyGr outperforms the other tools. Specifically, TyGr is 2.7% more accurate than Osprey in terms of overall type prediction, and more than 11.1% more accurate when predicting struct types.

Dirty was trained and evaluated only on x64 O0, but should support predicting types on x64 O1-O3 binaries (as stated in their paper). Therefore, we also train and test Dirty on x64 O1-O3 binaries. Because the authors of Dirty only report prediction accuracy, we compare the accuracy of Dirty against the accuracy of TyGr.

Figure 11 shows the overall type prediction accuracy of Dirty and TyGr, where TyGr outperforms Dirty by at least 26.1%. Figure 12 shows prediction accuracy for struct types. TyGr outperforms Dirty by at least 10%. Both struct accuracy and overall accuracy for Dirty on O1 are low due to the aforementioned shortcoming that is related to unseen variables. A further analysis was performed for the O1 dataset and we found 20.8% of variable types do not existing in the ground truth of Dirty. Dirty is unable to infer the types for them, causing the low accuracy.

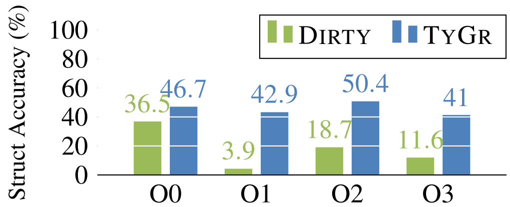

Figure 12: Accuracy of predicting `struct` types for TYGR and DIRTY on TYDA$_{\text{MIN}}$ x64 dataset.

7.2.2 Comparison against STATEFORMER

STATEFORMER does not support AArch64, so we only train and test it on x64, x86, Arm32, and Mips. STATEFORMER does not support struct member prediction; It only predicts every struct member variable as `struct`. Therefore, we post-process the prediction results of TYGR and unify struct member types that TYGR predicts into `struct` for a fair comparison.

As Table 8 shows, STATEFORMER achieves an average F1 score of 22.5% on x64, x86, Arm32, and Mips datasets. Whereas TYGR achieves an average F1 score of 63.3% and outperforms STATEFORMER by 40.8%. We believe that this difference is because STATEFORMER is evaluated on TYDA$_{\text{MIN}}$ (Table 3) which contains significantly more unique functions compared to their dataset (Table 2). In addition to a lower F1 score, STATEFORMER's method of identifying variables (*i.e.,* type inference candidates) results in significant redundancy [45, Table 1], further raising concerns about its effectiveness. Specifically, STATEFORMER considers *every* token in a stream of assembly as a candidate for type inference. This results in a lot of entities, such as `nop`, having the sentinel type of *no-access*. This inflates the successful predictions count, where most predictions are made on entities that are not directly usable by the end-user or any downstream analysis task, *e.g.,* decompiler.

We have also evaluated on the obfuscation binaries provided by STATEFORMER. Shown in Table 9, STATEFORMER achieves an average F1 score of 72.1% while TYGR achieves an average F1 of 79.9% and outperforms STATEFORMER by 7.8%.

Our results show that TYGR outperforms the state-of-the-art machine learning-based type inference techniques by a considerable margin.

7.3 RQ3: Efficiency of TYGR

In this section, we measure the inference performance of TYGR. Specifically, we measure each function's inference time and memory consumption, and the average numbers for each architecture.

Arch.	Opt.	TYGR	STATEFORMER
x64	O0	82.5	52.6
	O1	78.2	38.7
	O2	77.7	40.8
	O3	74.8	22.8
x86	O0	76.2	38.9
	O1	61.0	39.7
	O2	58.1	20.7
	O3	60.8	25.4
Arm32	O0	76.3	12.8
	O1	59.7	11.6
	O2	56.8	4.1
	O3	58.6	7.7
Mips	O0	56.8	12.1
	O1	46.5	5.7
	O2	43.6	15.5
	O3	44.6	10.3

Table 8: Comparison on F1 scores between STATEFORMER and TYGR.

Arch.	Opt.	TYGR	STATEFORMER
x64	bcf	80.2	72.0
	cff	81.0	72.1
	sub	78.5	72.2

Table 9: Comparison on F1 scores between STATEFORMER and TYGR for obfuscation binaries.

7.3.1 Inference Time

The per function inference time for TYGR ranges from 1.5 to 4.5 seconds, which is reasonable. The inference time per function for TYGR is slightly higher for O0 binaries across all architectures, ranging from 1.8 to 4.5 seconds. The inference time per function for other optimization levels range from 1.7 to 3.1 seconds This is because O0 binaries (without compiler optimizations) have more variables and more instructions (thus more VEX expressions and statements) than binaries that are compiled under higher compiler optimization levels.

7.3.2 Memory Consumption during Type Inference

The average memory consumption of TYGR ranges from 0.7 to 2.3 MB per function. As expected, TYGR uses more RAM during type inference for functions in O0 binaries. Interestingly, the memory consumption is higher for RISC architectures. On average, TYGR uses 2.1 MB for AArch64. In comparison, TYGR uses an average of 0.8 MB of RAM for x86 and 0.9 MB of RAM for x64. This is because RISC architectures have higher numbers of load and store instructions than on x86 and x64, leading to more nodes in the data-flow graph and, consequently, higher memory consumption.

We present the inference time in the same environment for four software projects in comparison with STATE-

Project	# Variables	Runtime (CPU)			
		TyGr	Stateformer	Debin	Ghidra
ImageMagic	23,727	208	148	N/A*	597
PuTTY	22,429	143	124	3,042	359
Findutils	6,534	43	18	675	80
zlib	730	6	3	41	9

*Debin terminated abruptly after running on one of the binaries for 138 minutes.

Table 10: Prediction time on CPU (in seconds) of TyGr, StateFormer, Debin, and Ghidra on four software projects with varying number of variables.

Former, Debin and Ghidra in Table 10. We chose these projects as they were used to measure the inference time in StateFormer. TyGr performs on par with StateFormer with slight difference. This shows that TyGr achieves higher precision with the same inference time.

7.3.3 Graph Building (or Training) Performance

We also measure the time it takes to build data flow graphs necessary for training on x64 Coreutils O0 binaries. It takes less than ten seconds for 80.42% of functions and 93.37% of functions complete within 30 seconds, and 97.35% of functions complete the process within 60 seconds.

For most cases, as expected, the graph build time is proportional to the number of variables. However, there are some cases where this is not the case. For instance, it took 400 seconds to build the data flow graph for a complex function with few variables. This is because the graph-building time also depends on the complexity of the data flows (e.g., nested loops and conditionals).

7.4 Feature Analysis

Table 6 provides inference results for certain types. An intriguing observation is that although bool type occupies a very small portion of the dataset, 1.7% in this case, it achieves comparative high accuracy. To understand what features contribute to this unique high precision, we conducted a case study on bool type. Bool type is typically used as a flag for branches and determines the following program path, and thereby the same bool variable may occur multiple times for different program paths when performing dynamic analysis. While other variables like i32 can also be utilized for branching purposes, bool variables are predominantly used for this task. This feature for bool variables exhibit a distinct characteristic. Specifically, bool nodes display a structure pattern that involves more than twice the number of location nodes and edges, such as `RegRead` node and `Value` edge in Figure 5, with respect to other variable nodes. Furthermore, these bool nodes are found to be connected to a greater number of edges with labels related to comparison operations such as `__eq__`.

8 Discussion

Our experiments demonstrate that using graph neural networks to learn and apply data-flow patterns for type inference is competitive with other machine learning approaches as well as industry-standard tools such as IDA. We now discuss the main limitations of TyGr and how to overcome or mitigate them.

Choice of program variables. The memory space is partitioned to three distinct regions: global, stack, and heap. TyGr focuses on predicting the types of stack and heap variables . It is possible to extend our implementation to handle global variables in a similar manner. In particular, it necessitates combining data-flow graphs from different functions that use the global variable.

Training set sizes and model performance. We varied the size of training set and retrained our models, and we observed that the prediction accuracy peaks at around 80%. We believe that the main reasons are (a) Certain variable types (e.g., `enum*` and `union*`) are too rare in the training set for training, and (b) Some variable types cannot be effectively differentiated by only observing how the variables are used. Interested readers can refer to Appendix A.2 for an in-depth analysis of these reasons. A critical improvement for TyGr will be incorporating callee- and caller-access patterns when building the data-flow graphs, which we leave as future work.

Predicting boundaries between two adjacent struct. For all struct variables, TyGr infer their types as `struct` before further inferring types for their members. However, if there are two adjacent struct variables on the stack, TyGr will predict them as a large consecutive struct variable and cannot infer the boundaries between them. Luckily, this scenario is rare in TyDa: Out of all functions with struct variables on their stack frames, only 0.1% of these functions have two or more struct variables, and even fewer of these struct variables are adjacent on the stack. We leave it as future work.

Scope of data-Flow analysis. Our data-flow analysis only examines intra-procedural data-flow. We expect that an inter-procedural analysis will yield richer input data for the learning model, and thus better performance. However, inter-procedural analysis is potentially expensive, and excessive analysis might offset the scalability benefit of using a machine learning approach. Nevertheless, we believe this to be a fruitful direction of investigation.

Indirect jump target resolution. Our analysis relies on angr to construct control-flow graphs. While angr can accurately compute the targets of direct jumps, estimating the targets of jumps that involve dynamic computation is much harder. As a result, we may end up never exploring certain

Because heap variables must be indirectly accessed by dereferencing pointers, TyGr predicts the type of a heap variable by predicting the shape of pointers that point to the variable.

parts of a function. Fortunately, the learning model can cope with such missing information.

9 Related Work

This work primarily focuses on type inference applied to binaries. Specifically, we focus on data type inference, i.e., recovering simple types for the identified variables.

Static analysis, specifically constraint solving, is one of the commonly used approaches. These techniques work by first introducing types based on specific rules and then propagate this seed information to different entities (variables/registers or memory objects) based on the program's data-flow [40]. One one hand, some prior works focus on inferring a limited set of types such as signed/unsigned integers [66], strings [12], `struct` types [58]. On the other hand, works such as TIE [35] and Retypd [42] attempt to infer a more comprehensive set of types specified using a type-lattice. These techniques seed their algorithms by assigning types based on certain base rules. For instance, an operand for `load` or `store` instruction should be of pointer type. They then use propagation techniques either based on Value Set Analysis [5] or constraint solving to propagate these seed types to all other entities.

Best-effort techniques [22, 29] that are based on heuristics suffer from precision. Furthermore, most of these techniques are specific to each architecture, such as x64, x86, etc. Although TIE [35] and Retypd [42] try to be architecture-agnostic by using an IR such as BIL [7], not all architectures (e.g., MIPS) are supported by BIL. Finally, none of these techniques are available as open-source [8], which makes it hard to evaluate or extend them. There are other techniques specific to C++ [26], where the main goal is to determine the classes and layout of objects. These techniques are not directly applicable as they mainly focus on recovering object-oriented features [67] such as class hierarchy [27, 52] and virtual table layout [17].

Some techniques use dynamic analysis [14, 30, 34, 69], wherein type propagation is usually done by taint tracking, and finally combine results from different executions to determine the type of a variable. However, the effectiveness of these techniques depends on the feasibility of executing the program and the availability of high coverage test cases, which is not easy, especially for libraries, embedded programs, and network-based programs.

Machine learning (ML) techniques have been explored in the context of binary analysis, popular applications being vulnerability detection [24, 24, 43, 60], function identification [6, 49, 54, 59], and code clone detection [16, 23, 25, 46, 62–64]. Most of these use traditional ML models such as SVMs. However, recent work [54, 62] have started using Neural networks, especially Recurrent Neural Networks (RNNs). ML techniques are also used for semantic problems such as type inference. CATI [9] uses word2vec to predict types based on usage contexts. Similarly, EKLAVYA [13] uses RNNs to predict function signatures, including types of the arguments. DEBIN [32] uses probabilistic models to predict debug information (types and names of variables) in stripped binaries. They use a dependency graph to encode uses of identified variables and then convert them into feature vectors and then train a model based on Extremely Randomized Trees [28]. STATEFORMER [45] sidesteps the problem of feature selection by using transformers on micro execution traces to learn the instruction semantics as pre-trained models. These pre-trained models are further used to perform type prediction. Similarly, DIRTY [10] also uses a transformer model for type prediction. The most recent technique, OSPREY [70] tries to combine both constraints solving and machine learning. Unfortunately, as shown in Table 1, none of these techniques have multi-architecture support and require considerable effort to extend to a new architecture. Finally, our comparative evaluation in Section 7.2 shows that TYGR outperforms all these techniques.

In contrast, TYGR uses data-flow analysis to precisely capture intra-procedural data-flow graphs and encodes them using graph neural networks, which in turn perform type inference via classification.

10 Conclusion

We present TYGR, a new technique for binary type inference. TYGR uses data-flow analysis to precisely track data flows for variables in an architecture-agnostic manner. The data-flow information is encoded using GNN, which then performs type inference as a classification task. We evaluate TYGR on TYDA_{MIN}, and demonstrate that it predicts types for variables with a high accuracy.

Acknowledgment

We thank the anonymous shepherd and reviewers for their valuable feedback. The following sources has partially supported the work upon which this material is based: Defense Advanced Research Projects Agency (DARPA) Contracts No. HR001118C0060, N6600120C4020, N6600120C4031, N6600122C4026, and N660012224037; Department of the Interior Grant No. D22AP00145-00; Department of Defense Grant No. H98230-23-C-0270; Office of Naval Research (ONR) Award N00014-23-1-2563; Army Research Office (ARO) Grant No. W911NF2310063; Advanced Research Projects Agency for Health (ARPA-H) Grant No. SP4701-23-C-0074; and National Science Foundation (NSF) Awards No. 1836936, 2107429, 2146568, 2232915, 2247686, 2247954, and 2313010. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, ONR, ARO, ARPA-H, or the U.S. Government.

References

[1] National Security Agency. Ghidra. https://ghidra-sre.org/.

[2] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

[3] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105, 2020.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. 2018.

[5] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, 2014.

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[8] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys (CSUR)*, 48(4):1–35, 2016.

[9] Ligeng Chen, Zhongling He, and Bing Mao. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98. IEEE, 2020.

[10] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium*, Boston, MA, August 2022.

[11] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[12] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. String analysis for x86 binaries. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 88–95, 2005.

[13] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, 2017.

[14] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, 2008.

[15] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[16] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *ACM SIGPLAN Notices*, 51(6):266–280, 2016.

[17] David Dewey and Jonathon T Giffin. Static detection of c++ vtable escape vulnerabilities in binary code. In *NDSS*, 2012.

[18] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

[19] EN Dolgova and AV Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35(2):105–119, 2009.

[20] Luke Dramko, Jeremy Lacomis, Edward J Schwartz, Bogdan Vasilescu, and Claire Le Goues. A taxonomy of c decompiler fidelity issues. In *33th USENIX Security Symposium (USENIX Security 24)*, 2024.

[21] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. In *International Conference on Information Security and Assurance*, pages 72–86. Springer, 2011.

[22] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 51–60, 2013.

[23] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, volume 52, pages 58–79, 2016.

[24] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[25] Qian Feng, Rundong Zhou, Yanhui Zhao, Jia Ma, Yifei Wang, Na Yu, Xudong Jin, Jian Wang, Ahmed Azab, and Peng Ning. Learning binary representation for automatic patch detection. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2019.

[26] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: approaching c++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356. IEEE, 2011.

[27] Alexander Fokin, Katerina Troshina, and Alexander Chernov. Reconstruction of class hierarchies for decompilation of c++ programs. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 240–243. IEEE, 2010.

[28] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[29] I Guilfanov. Simple type system for program reengineering. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 357–361. IEEE, 2001.

[30] Philip J Guo, Jeff H Perkins, Stephen McCamant, and Michael D Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 255–265, 2006.

[31] Christophe Hauser, Yan Shoshitaishvili, and Ruoyu Wang. Poster: Challenges and next steps in binary program analysis with angr.

[32] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[33] Hex-Rays. IDA Pro. https://hex-rays.com/ida-pro/.

[34] Changhee Jung and Nathan Clark. Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–66, 2009.

[35] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

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

[37] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, page 18, San Diego, CA, Feb 2010.

[38] 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*, pages 288–308. Springer, 2019.

[39] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery – a case study. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS22)*, ASIACCS 22, June 2022.

[40] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer, 1999.

[41] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[42] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 27–41, 2016.

[43] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 450–459. IEEE, 2015.

[44] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. "len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 152–152. IEEE Computer Society, 2024.

[45] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman 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*, pages 690–702, 2021.

[46] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[47] Aravind Prakash, Heng Yin, and Zhenkai Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 311–322, 2013.

[48] Sakthi Kumar Arul Prakash and Conrad S Tucker. Node classification using kernel propagation in graph neural networks. *Expert Systems with Applications*, 174:114655, 2021.

[49] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Machine learning-assisted binary code analysis. In *NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security, Whistler, British Columbia, Canada, December*. Citeseer, 2007.

[50] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[51] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer, 2018.

[52] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 426–441, 2018.

[53] Edward J Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, page 17. USENIX Association, 2013.

[54] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626, 2015.

[55] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[56] Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. Cornucopia: A framework for feedback guided generation of binaries. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[57] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, Feb 2011.

[58] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. Reconstruction of composite types for decompilation. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 179–188. IEEE, 2010.

[59] Shuai Wang, Pei Wang, and Dinghao Wu. Semantics-aware machine learning for function recognition in binary code. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 388–398. IEEE, 2017.

[60] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, 7:36340–36352, 2019.

[61] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.

[62] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[63] Hongfa Xue, Guru Venkataramani, and Tian Lan. Clone-hunter: accelerated bound checks elimination via binary code clone detection. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 11–19, 2018.

[64] Hongfa Xue, Guru Venkataramani, and Tian Lan. Clone-slicer: Detecting domain specific binary code clones through program slicing. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, pages 27–33, 2018.

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

Arch.	Opt. Level	MIN	AVG	MAX
x64	O0	4	1,787	254,465
	O1	4	1,302	467,033
	O2	4	1,323	517,754
	O3	4	1,273	1,155,116

Table 11: Function sizes in terms of instruction counts on x64 TYDA$_{\text{MIN}}$

[66] Qiuchen Yan and Stephen McCamant. Conservative signed/unsigned type inference for binaries using minimum cut. 2014.

[67] Kyungjin Yoo and Rajeev Barua. Recovery of object oriented features from c++ binaries. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 231–238. IEEE, 2014.

[68] Bin Zeng. Static analysis on binary code. Technical report, Tech-report, 2012.

[69] Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. 2012.

[70] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-Chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. OSPREY: recovery of variable and data structure via probabilistic analysis for stripped binary. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 813–832. IEEE, 2021.

A Appendix

A.1 Function Sizes

Detailed function size data, based on instruction counts, are shown in Table 11. We only show x64 here because the other architecture results are very similar.

A.2 Impact of Data on Prediction Accuracy

To analyze the effect of data volumes on the prediction performance of TYGR, we varied the size of the training data for x64 O0 and measured the change in the performance of our models. Training data was subsampled from TYDA$_{\text{MIN}}$ at rates of 30%, 40%, 60%, and increased by 20% for the rest. We also oversampled from TYDA to show the effect on training beyond TYDA$_{\text{MIN}}$ (this is represented as the size of training data between 100% and 140%). Figure 13 shows the results of our experiments, where the prediction performance of TYGR remains stable at around 80% beyond using 100% of TYDA$_{\text{MIN}}$.

What may cause the plateau of TYGR's prediction performance? We believe there are two main reasons:

(1) Some variable types are too rare. Certain variable types are too rare in the training set. For example, `union*` is only 0.14% of the data set while `enum*` is 0.08%. The prediction

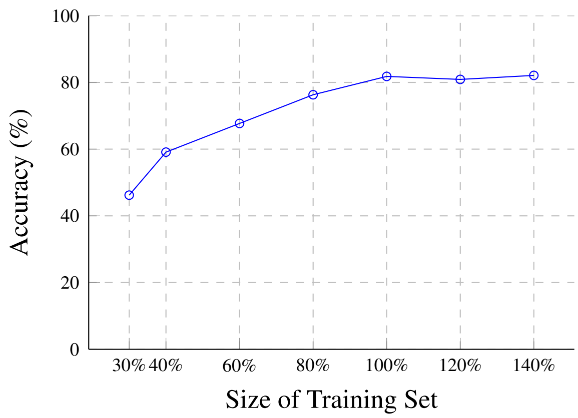

Figure 13: The accuracy of TYGR on x64-O0. 100% usage represents TYDA_{MIN}. The extra 20% and 40% are randomly selected from TYDA.

accuracy of these two types are 65% and 31%, respectively. We measured a total of 4.4% variables whose types occupy less than 1% of the training set, and the prediction accuracy for them are all around 50%, which are significantly lower than the overall accuracy of 80%.

(2) Similar access patterns for different variable types. We also noticed that some variable types cannot always be effectively differentiated only by their access patterns, such as `char*` versus `struct*` (consider a struct with only one-byte member fields), and `i32` versus `u32`.

We took a deeper look into `i32` variables. Out of all mis-prediction cases for `i32`, 36.35% are mis-predicted as `u32`, which is the most mis-predicted type. We randomly selected from our test set 40 C functions (15% out of 269) where `i32` variables are mis-predicted as `u32`. Then we compiled two versions of such functions: original (without change the type of the mis-predicted variable) and type-updated (where we update the type of the mis-predicted variable from `i32` to `u32`). Finally we compare the assembly after compiling both versions of functions. Not surprisingly, the assembly of type-updated function is the same as the assembly of the original function in all 40 pairs, which means that their access patterns will be the same, making it impossible for TYGR (or any other type inference tools that only rely on variable access patterns) to differentiate. We also manually inspected these 40 functions to understand how they use the mis-predicted `i32` variables. In most cases, these variables are used as flags, file descriptors, and other variables that only hold small integers; Updating their types to `u32` does not change their access patterns.

We observed a similar situation for `enum`, which is the second most mis-prediction cases (33.04% for `i32` mis-predictions). In most cases, `enum` variables are used like `i32` variables (e.g., both `int SANE_STATUS_GOOD = 0;` and `typedef enum{SANE_STATUS_GOOD = 0, ... }` used in the same code: `if (SANE_STATUS_GOOD) {...}`), resulting in the exact same assembly code after compiling.

Arch.	Opt. Level	Avg #Nodes	Avg #Edges
x64	O0	181	288
	O1	24	29
	O2	27	33
	O3	26	33
x86	O0	65	99
	O1	27	37
	O2	37	53
	O3	39	56
AArch64	O0	196	309
	O1	99	147
	O2	95	138
	O3	118	175
ARM	O0	180	295
	O1	67	101
	O2	81	122
	O3	89	137
MIPS	O0	142	238
	O1	55	83
	O2	64	98
	O3	67	102

Table 12: Average numbers of nodes and edges per graph generated by TyGr

A.3 Graph Statistics

Detailed statistics about the generated graphs (average number of nodes and edges) are shown in Table 12. O0 binaries contain the most variables and contribute to the highest average number of nodes and edges. The increase for the average numbers from O1 to O3 could be caused by the decrease of functions that contain variables.