TYGR: Type Inference on Stripped Binaries using Graph Neural Networks

Chang Zhu1*, Ziyang Li2*, Anton Xue2, Ati Priya Bajaj1, William Gibbs1, Yibo Liu1, Rajeev Alur1, Tiffany Bao1, Hanjun Dai1, Adam Doupé1, Mayur Naik2, Yan Shoshitaishvili1, Ruoyu Wang1, Aravind Machiry4

1{czhu62, atipriya, wfgibbs, yliu1049, tbao, doupe, yans, fishw}@asu.edu
2{liby99, antonxue, alur, mhnaik}@seas.upenn.edu
3hadai@google.com 4amachiry@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 re-hosting, 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
We consider the problem of mapping binary-level variables to source-level variables, which is 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 reverse engineering tasks. TYGr supports predicting struct shapes and member types, which essentially makes TYGr a type inference technique.

### 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.

#### 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 -0x33(%rbp),%eax
56: movlq %eax, %rdx
59: mov -0x2c(%rbp),%eax
5c: cltq -0x20 (%rbp): char*
60: sub %rax, %rdx -0x28 (%rbp): char*
61: mov -0x30(%rbp),%rax -0x2c (%rbp): int32
65: add %rdx,%rax -0x30 (%rbp): int32
68: mov %rax,-0x20(%rbp) -0x38 (%rbp): char*
6c: mov -0x28(%rbp),%rax -0x40 (%rbp): char*
70: mov %rax,-0x18(%rbp)
74: nop ...

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 P1 or P2 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
The sequence of nodes that \( T \) follows a topological order starting from the entry point. During the exploration, \( T \) extracts variable-level data-flow graphs that capture 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 \( T \) that executes the blocks in a function following a topological order starting from the entry point. The sequence of nodes that \( T \) visits induces a set of simple paths. \( T \) collects how each data location is accessed along each simple path.

### 4.2.2 Data-Flow Graphs

We derive data-flow graphs from the information that \( T \) 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 \( T \) 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). \( T \) 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
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 in-
formation 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., \( f(G) : G \rightarrow \mathbb{R}^{|V| \times d} \). Here \( G \) 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 \text{is}
  \text{Register}, \text{is_arg}
  \text{Register}, and \text{is_ret}
  \text{Register}.

- 11 value features related to the concrete node value,
e.g., \text{is_bool}, \text{is_float}, close_to_stack_pixeler, \text{is_zero},
\text{is_negative}, and \text{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
\( h_v^{(0)} = W_0x_v + b_0 \) 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
edge type \( r(v) \in \mathcal{R} \) representing its backward edge type. As the total number of possible types \(|\mathcal{R}|\) is known beforehand, we can design the message passing operation 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:

\[
h_v^{(l)} = \sigma \left( \text{AGGREGATE}(\{g(h_u^{(l-1)}, r(v), h_v^{(l-1)}) \}_{e=(u, r, v) \in \mathcal{N}_v}) \right)
\]

Here \( \sigma \) 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:

\[
h_v^{(l)} = \text{ReLU} \left( \sum_{r \in \mathcal{R}} \text{MEAN}(\{W_r^{(l)} h_u^{(l-1)} + W_0^{(l)} h_v^{(l-1)} \}_{e=(u, r, v) \in \mathcal{N}_v}) \right)
\]

where \( W_r^{(l)} \in \mathbb{R}^{d \times d} \) are weights that depend on layer index \( l \) and edge type \( r \), and \( W_0^{(l)} \in \mathbb{R}^{d \times d} \). \( \mathcal{N}_v \subseteq \mathcal{N} \) 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 \( \hat{t}_v \) is then compared with the ground truth type vector \( t_v \in \mathbb{R}^{|T|} \), the one-hot encoding of the ground truth type under the set of types \( T \). In this work, we apply cross entropy loss function

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

to compute the loss \( l = L(t_v, \hat{t}_v) \). 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-
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 address 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.

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

<table>
<thead>
<tr>
<th>Arch</th>
<th>Opt. Level</th>
<th># Unique Functions</th>
<th># Functions</th>
<th>#Dup Rate(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x64</td>
<td>O0</td>
<td>43,116</td>
<td>337,608</td>
<td>88.7</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>37,507</td>
<td>330,926</td>
<td>89.8</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>35,987</td>
<td>335,056</td>
<td>90.3</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>34,099</td>
<td>333,076</td>
<td>90.7</td>
</tr>
</tbody>
</table>

Table 3: Numbers of functions for both TYDAMIN and TYDA.

<table>
<thead>
<tr>
<th>Arch</th>
<th>Opt. Level</th>
<th># Functions</th>
<th>#Dup Rate(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x64</td>
<td>O0</td>
<td>543,101</td>
<td>53.6%</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>534,181</td>
<td>54.9%</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>540,132</td>
<td>54.3%</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>524,611</td>
<td>52.0%</td>
</tr>
<tr>
<td>x86</td>
<td>O0</td>
<td>332,644</td>
<td>13.8%</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>379,106</td>
<td>22.4%</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>392,195</td>
<td>22.9%</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>371,174</td>
<td>25.0%</td>
</tr>
<tr>
<td>AArch64</td>
<td>O0</td>
<td>131,984</td>
<td>23.6%</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>133,100</td>
<td>22.1%</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>140,214</td>
<td>22.9%</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>127,610</td>
<td>22.6%</td>
</tr>
<tr>
<td>Arm32</td>
<td>O0</td>
<td>112,714</td>
<td>42.6%</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>116,506</td>
<td>43.9%</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>103,791</td>
<td>45.3%</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>106,085</td>
<td>41.8%</td>
</tr>
<tr>
<td>Mips</td>
<td>O0</td>
<td>-</td>
<td>28.3%</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>-</td>
<td>32.0%</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>-</td>
<td>30.7%</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>-</td>
<td>32.2%</td>
</tr>
</tbody>
</table>

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*.

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?
We first evaluate the performance of TyGr 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x64</td>
<td>O0</td>
<td>81.8</td>
<td>46.7</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>76.0</td>
<td>42.9</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>75.7</td>
<td>50.4</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>72.8</td>
<td>41.0</td>
</tr>
</tbody>
</table>

Table 4: Overall accuracy and struct accuracy of TyGr on the x64 TyDA\textsubscript{MIN} dataset.

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

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\textsubscript{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 lead 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\textsubscript{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\textsubscript{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\textsubscript{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.
We compare T whose types present more than 0.1% of the x64-O0 T. To ensure a fair evaluation, we only compare against the configurations for which each tool was originally designed and inferred results of O. “Pointer Types” refers to the collection of all pointer types.

Table 6: Distribution and inference precision for variables. A further analysis was performed for the O1 dataset, causing the low accuracy. While O outperforms both. SPREY is not publicly available, its authors provide results of SPREY on x64-O0 binaries of GNU Coreutils. For a fair comparison, we remove the Coreutils functions that are within TyGR’s training set. Because the authors compared SPREY against DIRTY in their paper, we also evaluate DIRTY on the same set of binaries. SPREY 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 SPREY supports. For example, we convert both bool and char to Primitive_1, and const char * to Pointer.

Figure 10 shows the prediction accuracy of both overall types and only struct types for DIRTY, SPREY, and TyGR. TyGR outperforms the other tools. Specifically, TyGR is 2.7% more accurate than SPREY 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.

### 7.2 RQ2: Comparison Against Baselines

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

#### 7.2.1 Comparison against DIRTY and SPREY

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

While SPREY is not publicly available, its authors provide results of SPREY on x64-O0 binaries of GNU Coreutils. For a fair comparison, we remove the Coreutils functions that are within TyGR’s training set. Because the authors compared SPREY against DIRTY in their paper, we also evaluate DIRTY on the same set of binaries. SPREY 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 SPREY supports. For example, we convert both bool and char to Primitive_1, and const char * to Pointer.

Figure 10 shows the prediction accuracy of both overall types and only struct types for DIRTY, SPREY, and TyGR. TyGR outperforms the other tools. Specifically, TyGR is 2.7% more accurate than SPREY 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.
Whereas T\textsuperscript{ATE} achieves a higher F1 score of 79.9% and outperforms S\textsuperscript{ORMER} by 40.8%. We believe that this difference is because S\textsuperscript{ORMER} is evaluated on TY\textsubscript{DA}\textsubscript{MIN} (Table 3) which contains significantly more unique functions compared to their dataset (Table 2). In addition to a lower F1 score, S\textsuperscript{ORMER}'s method of identifying variables (\textit{i.e.}, type inference candidates) results in significant redundancy \cite[Table 1]{45}, further raising concerns about its effectiveness. Specifically, S\textsuperscript{ORMER} considers every token in a stream of assembly as a candidate for type inference. This results in a lot of entities, such as \texttt{nop}, having the sentinel type of \texttt{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, \textit{e.g.}, decompiler.

We have also evaluated on the obfuscation binaries provided by S\textsuperscript{ORMER}. Shown in Table 9, S\textsuperscript{ORMER} achieves an average F1 score of 72.1% while T\textsuperscript{ATE} achieves an average F1 of 79.9% and outperforms S\textsuperscript{ORMER} by 7.8%.

Our results show that T\textsuperscript{ATE} outperforms the state-of-the-art machine learning-based type inference techniques by a considerable margin.

### 7.3 RQ3: Efficiency of T\textsuperscript{ATE}

In this section, we measure the inference performance of T\textsuperscript{ATE}. Specifically, we measure each function’s inference time and memory consumption, and the average numbers for each architecture.

#### 7.3.1 Inference Time

The per function inference time for T\textsuperscript{ATE} ranges from 1.5 to 4.5 seconds, which is reasonable. The inference time per function for T\textsuperscript{ATE} 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 T\textsuperscript{ATE} ranges from 0.7 to 2.3 MB per function. As expected, T\textsuperscript{ATE} uses more RAM during type inference for functions in O0 binaries. Interestingly, the memory consumption is higher for RISC architectures. On average, T\textsuperscript{ATE} uses 2.1 MB for AArch64. In comparison, T\textsuperscript{ATE} 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 S\textsuperscript{ORMER} and T\textsuperscript{ATE}. Shown in Table 8, S\textsuperscript{ORMER} achieves an average F1 score of 72.2% for obfuscation binaries.
Debin terminated abruptly after running on one of the binaries for 138 minutes. While other variables like int32 can also be utilized for branch predication, bool nodes are found to be connected to a greater number of edges with labels related to comparison operations such as 

\[ \texttt{eq} \].

We also measure the time it takes to build data flow graphs 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 targets of jumps that involve dynamic computation is much harder. As a result, we may end up never exploring certain

\[ \texttt{eq} \].

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 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.

<table>
<thead>
<tr>
<th>Project</th>
<th># Variables</th>
<th>Runtime (CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageMagick</td>
<td>23,727</td>
<td>208</td>
</tr>
<tr>
<td>PuTTY</td>
<td>22,429</td>
<td>143</td>
</tr>
<tr>
<td>Findutils</td>
<td>6,534</td>
<td>43</td>
</tr>
<tr>
<td>zlib</td>
<td>730</td>
<td>6</td>
</tr>
</tbody>
</table>

*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.
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, 46–64]. 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 TyDAmin, 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


Some variable types are too rare. Certain variable types is only 0.14% of the data set while enum* is only 0.08%. The prediction

<table>
<thead>
<tr>
<th>Arch</th>
<th>Opt. Level</th>
<th>MIN</th>
<th>AVG</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>x64</td>
<td>O0</td>
<td>4</td>
<td>1.787</td>
<td>254,465</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>4</td>
<td>1.302</td>
<td>467,033</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>4</td>
<td>1.323</td>
<td>517,754</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>4</td>
<td>1.273</td>
<td>1,155,116</td>
</tr>
</tbody>
</table>

Table 11: Function sizes in terms of instruction counts on x64
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.

### 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.

<table>
<thead>
<tr>
<th>Arch.</th>
<th>Opt. Level</th>
<th>Avg #Nodes</th>
<th>Avg #Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>x64</td>
<td>O0</td>
<td>288</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>29</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>33</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>33</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>O4</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>x86</td>
<td>O0</td>
<td>99</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>147</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>138</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>175</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>O4</td>
<td>202</td>
<td>130</td>
</tr>
<tr>
<td>AArch64</td>
<td>O0</td>
<td>295</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>101</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>122</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>137</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>O4</td>
<td>199</td>
<td>124</td>
</tr>
<tr>
<td>ARM</td>
<td>O0</td>
<td>137</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>83</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>98</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>102</td>
<td>67</td>
</tr>
<tr>
<td>MIPS</td>
<td>O0</td>
<td>238</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>83</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>98</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>102</td>
<td>67</td>
</tr>
</tbody>
</table>

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