Toward Inferring Structural Semantics from Binary Code Using Graph Neural Networks

Noriki Sakamoto
Graduate School of Engineering
Osaka Electro-Communication University
Neyagawa, Osaka, Japan
mi24a001@oecu.jp

Kazuhiro Takeuchi
Faculty of Information and Communication Engineering
Osaka Electro-Communication University
Neyagawa, Osaka, Japan
takeuchi@osakac.ac.jp

Abstract

Recovering semantic information from binary code is a fundamental challenge in reverse engineering, especially when source-level information is unavailable. We aim to analyze the types and roles of structural elements from the binary observed in the compiled program, focusing on their contextual usage patterns and associations to other members.We refer to such semantic aspects as *structural semantics* , meaning that cooccurring patterns of jointly updated structure members reveal the functional roles that can be inferred from their coupling, throughout this paper. Recent approaches have applied graph neural networks (GNNs) to data-flow graphs (DFGs) for variable type inference, but most rely on a single model architecture, such as the relational graph convolutional network (R-GCN). While effective, such models may overlook alternative patterns of structure member behavior. In this paper, we investigate the effectiveness of three alternative GNN architectures gated graph neural networks (GGNN), graph attention networks (GAT), and standard graph convolutional networks (GCN) in capturing structural semantics from binary-level data-flow graphs. We evaluate these models on real-world binaries compiled at multiple optimization levels, measuring their ability to infer semantic properties of structure members. Our results show that these architectures capture complementary aspects of structural semantics. GGNN is effective at modeling long-range dependencies, GAT suppresses irrelevant connections, and GCN offers computational simplicity. Different model architectures emphasize distinct aspects of structural semantics, capturing complementary patterns of how structure members are accessed together in memory. This demonstrates that architectural diversity provides richer perspectives for semantic inference in binary analysis.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

Keywords

Decompiler, Reverse Engineering

ACM Reference Format:
Noriki Sakamoto and Kazuhiro Takeuchi. 2025. Toward Inferring Structural Semantics from Binary Code Using Graph Neural Networks. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3733822.3764673

1 Introduction

Most software is developed using high-level programming languages and then compiled into binary format for deployment and execution on target platforms. As a result, what remains is a sequence of binary instructions that operate directly on hardware resources such as CPU registers, memory addresses, and stack frames. Given this loss of semantic information, it is extremely difficult to recover the original purpose or logical structure of a program solely by observing low-level memory state changes. Furthermore, this challenge is exacerbated by factors such as compiler optimizations and the diversity of hardware architectures [20]. Although a program's execution can, in principle, be interpreted as a time-series of state transitions across registers, caches, and memory cells, such information alone is insufficient to reconstruct high-level semantics accurately. This limitation becomes especially apparent when attempting to understand how different parts of a program interact, or what roles particular data structures and variables play in a larger computational context.

Simply tracing low-level state transitions poses specific problems. In particular, it is inadequate for recovering high-level meanings related to interprocedural interactions and variable roles. The need to recover such structural semantics arises in a wide range of domains, including proprietary software auditing, firmware verification, and incident response. Specifically, in the fields of malware analysis [4] and vulnerability discovery [13], analysts are tasked with analyzing binary executables to understand malicious behavior, identify security flaws, and reconstruct the functionality of stripped or obfuscated code. In such contexts, source code is often unavailable by design - either because the software is distributed only in compiled form, or because the original code has been intentionally obfuscated.

To address this challenge, reverse-engineering tools such as decompilers aim to reconstruct approximate source-level representations from binaries. However, before high-level code can be recovered, critical intermediate tasks must be performed, such as identifying variable types, detecting struct layouts, and grouping related memory elements. These low-level reconstructions serve as foundational indicators for subsequent analyses. As a result, recent research has increasingly focused on techniques for recovering lost semantic information—not only variable types and structure layouts, but also variable names and usage roles. For example, structure layout recovery methods include rule-based and heuristic approaches [14], constraint-solving techniques [10] [21], and machine learning–based methods [2] [19] [12] [16].

Recent tools and studies, including Ghidra and TYGR (TYpe inference on stripped binaries using GRaph neural networks) [22], employ DFG as an intermediate representation for variable recovery. TYGR, for instance, applies a GNN over a directed DFG that is augmented with separate reverse-edge relations (i.e., backward edges are treated as distinct edge types) to assign compact embeddings to each node, allowing type prediction without manual analysis. Although originally designed for type classification, these embeddings implicitly capture broader semantic cues, such as pointer-usage patterns, aliasing, and latent state transitions.

This work focuses on the impact of model choice when applying GNNs to DFGs for data structure recovery. While we do not address explicit state-machine inference in this paper, we interpret information propagation on the DFG as a process that maps localized updates of program state. In this sense, a GNN can be viewed as approximating latent state transitions through local dependencies, as conceptually illustrated in Figure 1. The empirical validation of this perspective is left for future work; here, we focus on coefficient analysis and cross-model comparison. To this end, our method applies multiple GNN models, including RGCN [17], GGNN [11], GCN [9], and GAT [18], to DFGs to infer struct members based on instruction dependencies and memory access behavior. We perform comparative evaluations across these architectures to assess their impact on inference accuracy and reconstruction performance. Experiments are conducted on real-world binaries stripped of debug information, with evaluation focusing on member prediction accuracy, correctness of inferred structural transitions, and analysis of misprediction patterns. Our findings contribute to advancing binary analysis through internal state modeling and support the development of high-precision tools for decompilation, protocol reverse engineering, security analysis, and firmware inspection.

2 Background and Related Work

2.1 Recent Software Development Status

Program execution can be understood as a time-series of state changes occurring in various forms of memory and storage systems, including CPU registers, caches, and main memory (RAM). At a concrete level, program behavior progresses through the sequential execution of instructions, each of which updates the machine state by writing to registers or modifying memory contents. Consequently, capturing and analyzing the temporal evolution of memory values has emerged as a promising direction for inferring the underlying computations and logical behavior of binary programs. Recent studies have investigated fine-grained execution traces at the hardware level to recover control flow and memory access patterns, as well as speculative or transient states in memory to infer execution semantics. These memory-centric approaches have shown applicability in reverse engineering, program decompilation, and even detecting malicious behaviors in stripped binaries using large language models (LLMs) [19].

Beyond mere memory access tracking, the interpretation of variable grouping plays a critical role in understanding program semantics. For instance, when a structure contains an `int`-type member variable, it may often serve as a counter or state holder. If it contains only an array of strings, its design as a struct may be unnecessary. However, when a string represents meaningful information—such as a file name—and is linked with other state information (e.g., file status: *open*, *modified*), grouping them as a struct becomes a natural design choice. By co-locating related variables as struct members, it becomes possible to centrally manage the operational state of an object and explicitly represent state transitions. This pattern is also observed in counters. For example, a monotonically increasing `int` member may track retry counts in communication processes. When exceeding a certain threshold, this value may trigger a transition from *connecting* to *failed* In such designs, counters are rarely handled in isolation; instead, they are integrated into data structures alongside relevant identifiers, such as socket descriptors. Thus, in binary-level analysis, understanding how variables are grouped—and not just their individual types—is essential for recovering structural program semantics.

In recent years, while AI-driven code generation has seen remarkable progress, our capacity—whether manual or automated—to comprehend generated code remains limited. This limitation is especially pronounced at the binary level. During the compilation process, much of the source-level abstraction is irreversibly lost. High-level programming languages are designed to express complex logic, abstract control structures, and programmer intent in a semantically rich and human-readable manner. However, during compilation to machine code, essential contextual elements such as variable names, data types, structural relationships, and control abstractions are frequently discarded or flattened [1].

To address these challenges, the field of reverse engineering has gained importance. It aims to reconstruct meaningful representations—such as control-flow graphs, variable structures, and semantic groupings—from binary artifacts. This often requires a combination of static and dynamic analysis, and more recently, machine learning-based techniques have been explored to bridge the semantic gap between low-level binary instructions and high-level meaning. The ultimate goal is to support tasks such as vulnerability triage, binary lifting, automated decompilation, and program similarity analysis by recovering sufficient semantic insight from compiled binaries.

Binary type analysis has traditionally relied on heuristic and constraint-based reconstruction techniques. Tools such as *Ghidra* [14] employ heuristic-based decompilation methods, utilizing empirically derived patterns and rule-based matching to infer data types from compiled code. While partially effective, these methods often operate independently of formal instruction set architectures and instead depend on manually curated patterns derived from human expertise. As a result, such approaches may produce inaccurate or inconsistent outcomes, especially when analyzing highly optimized or obfuscated binaries.

2.2 Binary analysis-based recovery methods

Lee et al.[10] proposed TIE, a binary type inference method based on constraint solving derived from instruction set specifications.

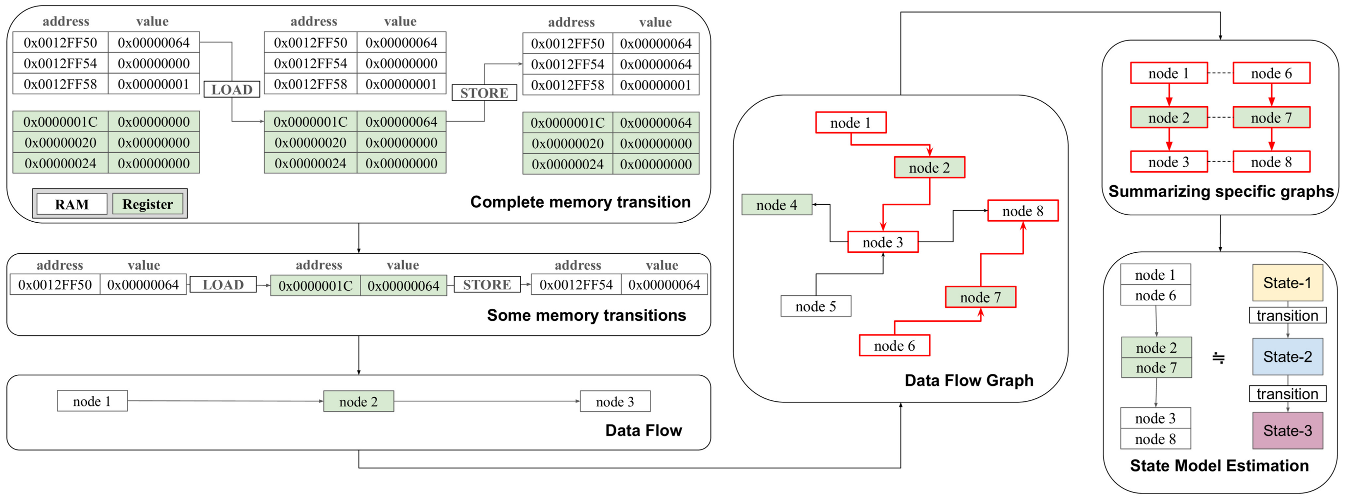


Figure 1: Modeling state transitions of disjoint subgraphs on a DFG

Technique	Input	Struct	Struct Ptrs	Struct Members	Structure grouping	Multi-arch
IDA [7]	Binary	✗	✗	✗	✗	✓
Ghidra [14]	Binary	✗	✗	✗	✗	✓
TIE [10]	Binary	✓	✓	✓	✓	✗
RETYPD [15]	Binary	✓	✓	✓	✓	✗
OSPREY [21]	Binary	✓	✓	✓	✓	✗
DEBIN [6]	Disassembly	✓	✗	✗	✗	✗
STATEFORMER [16]	Runtime Values	✓	✓	✗	✗	✓
DIRTY [2]	Decompilation	✓	✓	✗	✗	✗
TYGR [22]	Binary	✓	✓	✓	✗	✓

Table 1: Comparison of existing type-recovery techniques (dynamic-trace–based approaches excluded)

This approach analyzes how register and memory operations in binary code relate to variables and expressions used in the original source code, and generates type constraints based on usage patterns. For example, if a register is used in a signed division, a constraint is derived indicating that the register must at least be of a signed type. By collecting such constraints from across the entire code and solving them through a constraint resolution algorithm, TIE enables a more principled and flexible form of type recovery without relying on empirical heuristics. However, constraint-based methods like TIE face limitations in narrowing down candidate types when the inclusion relationships among types are insufficiently captured, making it difficult to deduce a single concrete type.

To address the reliance on manually crafted rules in traditional binary type inference methods, He et al.[6] proposed DEBIN, a probabilistic model that captures the relationship between binary code and the original source code. DEBIN represents the correlation between instruction patterns in binary code and their corresponding variables or expressions using a conditional random field, thereby enabling type inference. This approach eliminates the need for manually defined rules and instead infers types based on learned statistical relationships.

Furthermore, Chen et al.[2] highlighted two key limitations: constraint-based methods such as TIE are incapable of inferring variable names and type names, while machine learning approaches like DEBIN, which rely on instruction sequences, often lack syntactic consistency. To overcome both issues, they proposed DIRTY, a method designed to recover types and variable names simultaneously. DIRTY is structured to ensure consistency in both syntax and semantics, enabling more practical and accurate reconstruction of program information.

The studies discussed so far still suffer from several limitations, including incomplete recovery of fine-grained structure types and a lack of support for a wide range of architectures. In the context of variable-type recovery and the fine-grained reconstruction of structure types, the data-flow graph (DFG) has drawn particular attention. Because a DFG records where each value is produced, propagated, and consumed, it retains information that hints at data types in the most unadulterated form.

This section clarifies the mutual relationship between the data-flow graph (DFG) and the control-flow graph (CFG) that is traversed while the DFG is being generated.

A CFG partitions binary code into basic blocks and represents control-flow transfers between them as directed edges. It captures inter-function calls and depicts the program's possible execution paths, making it well suited for reasoning about call relationships, branch reachability, and loop behavior. Because the CFG concentrates on control aspects, however, it is not specialized for detailing how data are exchanged between registers and memory or how values are transformed and propagated—that role is filled by the DFG.

A DFG is defined as a graph structure in which nodes represent data stored in registers or at memory offsets, and directed edges denote operations—such as arithmetic instructions—performed on those nodes [8]. This representation models how the state of each node (corresponding to a finite storage location) evolves over successive operations, and it is commonly expressed in Static Single Assignment (SSA) form in many analysis tools. By composing these node–operation relationships across multiple layers, one can reconstruct the sequence of transformations that the data undergoes—that is, its overall computational behavior. Furthermore, by examining properties of a node after its state has been updated—such as bit width or signedness—it becomes possible to infer higher-level information like the variable's type. Many decompilers, including Ghidra, thus adopt the DFG as a foundational representation for variable-type recovery.

Performing static analysis directly on a DFG is challenging: complex branches, interprocedural dependencies, and the computational cost of large code bases make the task difficult for both humans and machines. To mitigate these issues, we focus on TYGR [22], which embed the structural dependencies of a DFG with a GNN, allowing variable types to be recovered more easily.

Zhu et al.[22] proposed TYGR, a GNN-based type inference method, to address two shortcomings of many prior works: the lack of support for fine-grained type recovery of struct members, and the limited architecture compatibility of existing tools and methods. TYGR infers variable types in the original source code by analyzing register value operations and dependencies in the Data Flow Graph (DFG). It embeds these register relationships into feature vectors, allowing the model to predict types based on the resulting embedded representations.

Although recent work has begun to address these shortcomings, challenges specific to structures remain unresolved. The following discussion therefore concentrates on the relationship between structure reconstruction and multi-architecture support.

Table 1 summarizes representative type-recovery techniques from five perspectives: direct reconstruction of structures, analysis of structure pointers (i.e., mutual references), inference of member types, higher-level grouping of multiple structures, and support for multiple instruction-set architectures. The table reveals stark disparities among tools. IDA Pro and Ghidra, while covering a wide range of architectures, offer only limited capabilities for rebuilding structures themselves and provide insufficient support for pointer analysis or member-type inference. By contrast, TIE, Retypd, and Osprey can, in principle, infer the structure body, its pointers, and the member types and can even group several structures under a higher-level abstraction; however, none of these approaches generalizes across multiple architectures. Our goal, following the direction suggested in Figure 2, is to predict fine-grained variable types while simultaneously grouping structure members.

The TYGR paper makes it clear that its goal is only to infer each variable node's type independently with an R-GCN, and it does not consider consistency across structure fields.

We nonetheless believe that the embeddings TYGR learns encode cues beyond mere type labels—such as shared access patterns and dependencies among members of the same structure. When the entire DFG is viewed, nodes belonging to the same structure might still cluster together in vector space even if they reside in distant subgraphs. Leveraging this property could pave the way for field-level clustering and, ultimately, automated grouping at the structure level.

2.3 Graph Neural Networks for Type Inference

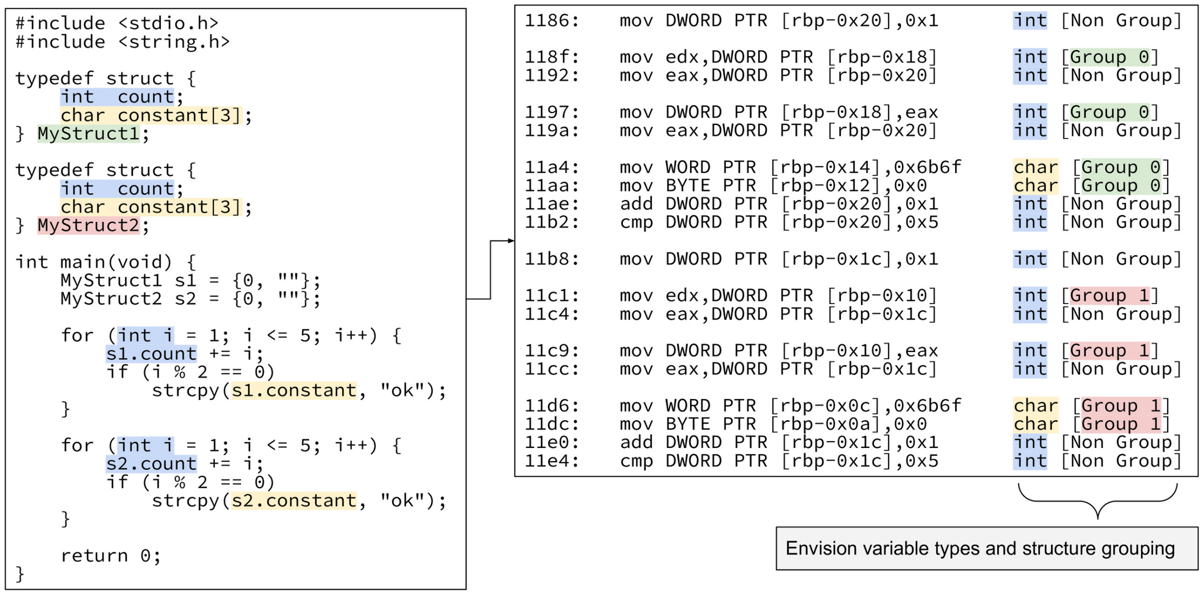
This section provides an overview of the processing pipeline proposed in the original TYGR paper and organizes the components necessary for the reproduction experiments. TYGR consists of four major stages: (i) binary preprocessing, (ii) DFG construction, (iii) initial node embedding based on node features, and (iv) embedding and type classification using a GNN.

In the first stage, binary preprocessing is performed using angr[5]. The goal here is to construct a CFG composed of the intermediate representation VEX IR derived from the instruction set architecture (ISA). A key advantage of using angr is that it mitigates a common limitation in many existing studies: their dependency on specific architectures. Because angr can lift binaries into the architecture-independent VEX IR, it supports a wide range of ISAs including x86, x64, ARM, MIPS, and RISC-V. As a result, the downstream learning modules can focus purely on data dependencies without being architecture-aware. VEX IR captures register and memory accesses in an ISA-formatted manner[3], and angr also supports transformation into a CFG at the basic-block level.

Using the generated VEX IR and its associated CFG, a DFG is constructed, where nodes represent registers or memory offsets, and directed edges correspond to operations acting on these nodes. Multiple paths may emerge in the graph due to constructs like conditional branches (e.g., if-statements).

To prepare the DFG for processing by a GNN, each node is assigned an initial embedding based on its structural and semantic characteristics. TYGR embeds each DFG node into a 31-dimensional one-hot vector. These features include bit width, immediate value patterns, and other low-level attributes. Edges are labeled with one of 44 operation types, each represented using a one-hot vector as well.

Subsequently, the embedded DFG is passed through an 8-layer RGCN (Relational Graph Convolutional Network), where message passing is performed at each layer to iteratively update the node representations. The parameters are not shared across layers, the activation function used is GELU, and the aggregation function is mean pooling.



```
#include <stdio.h>
#include <string.h>

typedef struct {
    int  count;
    char constant[3];
} MyStruct1;

typedef struct {
    int  count;
    char constant[3];
} MyStruct2;

int main(void) {
    MyStruct1 s1 = {0, ""};
    MyStruct2 s2 = {0, ""};

    for (int i = 1; i <= 5; i++) {
        s1.count += i;
        if (i % 2 == 0)
            strcpy(s1.constant, "ok");
    }

    for (int i = 1; i <= 5; i++) {
        s2.count += i;
        if (i % 2 == 0)
            strcpy(s2.constant, "ok");
    }

    return 0;
}
```

```
1186:   mov DWORD PTR [rbp-0x20],0x1      int [Non Group]

118f:   mov edx,DWORD PTR [rbp-0x18]      int [Group 0]
1192:   mov eax,DWORD PTR [rbp-0x20]      int [Non Group]

1197:   mov DWORD PTR [rbp-0x18],eax      int [Group 0]
119a:   mov eax,DWORD PTR [rbp-0x20]      int [Non Group]

11a4:   mov WORD PTR [rbp-0x14],0x6b6f    char [Group 0]
11aa:   mov BYTE PTR [rbp-0x12],0x0       char [Group 0]
11ae:   add DWORD PTR [rbp-0x20],0x1      int [Non Group]
11b2:   cmp DWORD PTR [rbp-0x20],0x5      int [Non Group]

11b8:   mov DWORD PTR [rbp-0x1c],0x1      int [Non Group]

11c1:   mov edx,DWORD PTR [rbp-0x10]      int [Group 1]
11c4:   mov eax,DWORD PTR [rbp-0x1c]      int [Non Group]

11c9:   mov DWORD PTR [rbp-0x10],eax      int [Group 1]
11cc:   mov eax,DWORD PTR [rbp-0x1c]      int [Non Group]

11d6:   mov WORD PTR [rbp-0x0c],0x6b6f    char [Group 1]
11dc:   mov BYTE PTR [rbp-0x0a],0x0       char [Group 1]
11e0:   add DWORD PTR [rbp-0x1c],0x1      int [Non Group]
11e4:   cmp DWORD PTR [rbp-0x1c],0x5      int [Non Group]
```

Figure 2: Left: A C function with two simple structs. Right: Variable type prediction and struct grouping at corresponding stack offsets.

3 Learning GNN model for Structural Semantics

3.1 Research challenges in applying DFGs to GNNs

TYGR reports its results with a single architecture that directly adopts the message-passing design of R-GCN [17], yet it never explains why R-GCN was chosen or how it fits a data-flow graph (DFG) better than other approaches.

The four architectures in Table 2 share the message-passing paradigm, but they aggregate information and treat edges in fundamentally different ways. GCN averages the features of adjacent nodes uniformly, offering low computational cost and a simple implementation, although it tends to dilute important dependencies when edge semantics vary. GAT introduces an attention mechanism that learns individual importance weights for each neighbor, allowing it to highlight essential dependencies while down-weighting noise. R-GCN goes a step further by assigning a separate weight matrix to every relation type, making it highly compatible with DFGs that carry dozens of distinct operation labels. Finally, GGNN employs GRU-based gated recurrent propagation, enabling it to incorporate long-range dependencies over multiple hops.

Because a DFG captures where values are produced, propagated, and consumed inside a binary, its nodes and edges exhibit a wide diversity of relationships. Consequently, the most suitable GNN depends on how many heterogeneous edges the graph contains and how prominently long-distance dependencies appear.

For this reason, it is essential to evaluate more than a single model, comparing multiple GNNs and examining their respective strengths and weaknesses. Doing so clarifies the rationale behind model selection and supports evidence-based architecture choices for downstream tasks such as type inference and program comprehension.

3.2 Proposed Method

This study aims to systematically evaluate how different graph neural network (GNN) architectures capture variable-access patterns in binary code and how much they improve type-inference accuracy. We compare four architectures: the relational graph convolutional network (R-GCN) employed in TYGR [22], the vanilla graph convolutional network (GCN) [9], the graph attention network (GAT) [18], and the gated graph neural network (GGNN) [11]. GNNs learn node representations by mapping sets of nodes into a latent space according to the local and global relational structure encoded by nodes and edges; however, because each architecture weights edges and aggregates messages differently, the resulting embeddings possess distinct characteristics. We therefore quantify how these differences emphasize or abstract variable memory-access patterns and how they ultimately affect type-inference performance.

Following the notation of TYGR, let $G = (V, E)$ be the directed data-flow graph of a function, and let each edge $e = (u, r, v) \in E$ denote a relation of type r from node u to node v. TYGR augments the graph by adding a backward edge for every forward edge that has a well-defined inverse relation, thereby enabling bidirectional message passing. Each node $v \in V$ is associated with an initial feature vector $\mathbf{x}_v \in \mathbb{R}^D$, which is transformed into the initial node embedding $\mathbf{h}_v^{(0)} = \mathbf{W}_0\mathbf{x}_v + \mathbf{b}_0$, where $\mathbf{W}_0 \in \mathbb{R}^{d\times D}$ and $\mathbf{b}_0 \in \mathbb{R}^d$ are trainable parameters, D denotes the feature dimension, and d is the embedding size. We denote by $\mathbf{h}_v^{(l)}$ the embedding of node v at layer l and use $\sigma(\cdot)$, such as ReLU, for nonlinear activation. Let $\mathcal{N}_v$ be the set of incoming neighbors of v, including those connected via the added reverse edges. In what follows, we present the layer-wise node-embedding update rules for each GNN model in our approach, employing the same graph and feature settings as in TYGR.

Model	Key Characteristics
R–GCN (Relational GCN)	Utilizes separate weight matrices for each relation type, enabling multi-relational message passing; effective when a graph contains many distinct edge labels, such as operation kinds in a DFG.
GCN (Graph Convolutional Network)	Applies shared weights uniformly to all neighboring nodes, capturing local structure efficiently; may underperform on graphs with heterogeneous edge semantics.
GAT (Graph Attention Network)	Learns attention coefficients for each neighbor, emphasizing important dependencies while down-weighting noisy edges; robust to heterogeneous or noisy graphs.
GGNN (Gated Graph Neural Network)	Employs GRU-based gated recurrent message passing, making it well-suited for capturing long-range dependencies and iterative patterns in control or data flows.

Table 2: Models and Key Characteristics

In Graph Convolutional Networks (GCNs), the embedding of a node v at layer l, denoted $h_v^{(l)}$, is obtained by first aggregating the previous-layer embeddings $h_u^{(l-1)}$ of the node itself and of its neighbors $u \in \mathcal{N}(v)$. This sum is normalized by the node degrees, multiplied by the layer weight matrix $W^{(l)}$, and then passed through a non-linear activation function.

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\tilde{d}_i\, \tilde{d}_j}}\, h_j^{(l)} W^{(l)}\right) \tag{1}$$

Here, d_v is the degree of node v (including the self-loop), and σ is an activation function such as ReLU. The formula represents a graph convolution that averages feature vectors from adjacent nodes, scaling each contribution by the symmetric normalization factor $1/\sqrt{d_v d_u}$. In other words, a GCN computes the next-layer embedding by averaging and weighting the features of node v and its neighborhood $\mathcal{N}(v)$.

Relational GCNs (RGCNs) extend GCNs to multi-relational graphs by using distinct weight matrices W_r for messages arriving along each edge type r.

$$h_v^{(l+1)} \;=\; \sigma\!\left(W_0^{(l)}\, h_v^{(l)} \;+\; \sum_{r \in \mathcal{R}} \sum_{u \in \mathcal{N}_v^r} \frac{1}{|\mathcal{N}_v^r|}\, W_r^{(l)}\, h_u^{(l)}\right). \tag{2}$$

The next-layer embedding of node v, $h_v^{(l+1)}$, is obtained by applying the weight $W_r^{(l)}$ to the previous-layer embeddings $h_u^{(l)}$ of neighbors u connected via relation r, averaging these messages within each relation, and then summing over all relations. In addition, a self-loop term applies $W_0^{(l)}$ to the node's own representation $h_v^{(l)}$. Finally, a non-linear activation function σ (e.g., ReLU) produces the updated embedding $h_v^{(l+1)}$.

In Graph Attention Networks (GATs), an attention coefficient $\alpha_{vu}^{(l)}$ is introduced for the message that node v receives from each neighbor $u \in \mathcal{N}(v)$.

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{ij}^{(l)}\, h_j^{(l)} W^{(l)}\right) \tag{3}$$

The coefficient $\alpha_{vu}^{(l)}$ is an attention weight learned from the feature vectors of nodes v and u and is normalized by a softmax so that the weights over each neighborhood sum to 1. Consequently, node v assigns larger weights to informative neighbors and suppresses irrelevant information. In the equation, $W^{(l)}$ is the weight matrix for layer l, and σ is a non-linear activation. In short, a GAT aggregates messages from neighbors with attention-based weights, emphasizing more important neighborhood information when updating embeddings.

In Gated Graph Neural Networks (GGNNs), gated recurrent units (GRUs) are used to iteratively update node representations. At each step (layer) l, node v first forms the message $m_v^{(l)}$ by linearly transforming and summing the embeddings of its neighbors, and then combines this message with the previous hidden state in a GRU to produce the new embedding $h_v^{(l)}$.

$$h_i^{(t)} = \mathrm{GRU}\Big(h_i^{(t-1)},\ \sum_{j \in \mathcal{N}(i)} h_j^{(t-1)} W + b\Big) \tag{4}$$

Concretely, the neighbor message is aggregated as $m_v^{(l)} = \sum_{u \in \mathcal{N}(v)} W^{(l)} h_u^{(l-1)}$, which, together with the previous state $h_v^{(l-1)}$, is fed into a GRU cell to yield the updated embedding $h_v^{(l)}$. Inside the GRU, reset and update gates adjust the relative importance of the old state and the new message, mitigating vanishing gradients and supporting long-range information propagation. As a result, in a GGNN, neighborhood information is propagated through gated mechanisms over multiple steps and gradually reflected in the embedding of node v.

4 Experiments and Evaluation

4.1 Dataset for Experiments

In this paper, we employ the TYDA dataset[22], which comprises binaries compiled from a variety of software repositories maintained by the Gentoo and Debian projects. Each binary is built with debug-symbol inclusion enabled (using the *-g* option), thereby embedding DWARF-format debug information. This DWARF metadata contains type definitions and structure layouts, and it serves as the basis for our data-preprocessing pipeline. We utilize the preprocessing scripts distributed alongside the dataset and, taking computational resource constraints into account, process as much of the available data as is feasible. Table 3 reports the total number of functions and variables that were successfully processed.

Opt. Level	Functions	Variables
O0	338,520	7,924,530
O1	69,989	1,938,719
O2	61,418	2,079,354
O3	66,466	428,856

Table 3: Number of functions and variables per optimization level (x64 binaries)

4.2 Comparison with TYGR

In our experiments we randomly sampled the x64 binaries in the TYDA data set and split the samples into training, validation and test partitions in an 8 : 1 : 1 ratio. We implemented R-GCN, GAT, GGNN and GCN, but here we concentrate on the R-GCN configuration that mirrors the one used in the TYGR paper. The hyper-parameters followed that reference: a learning rate of 10¯3, batch size 32, the Adam optimiser, eight message-passing layers and a hidden dimension of 64. Training stopped as soon as the validation loss reached its minimum. All runs were carried out on an Ubuntu 20.04 server equipped with two Intel Xeon E5-2687W v4 processors, 400 GiB of RAM and four Tesla K40 GPUs. Precision, Recall and F1-score served as the evaluation metrics; we computed them separately for base and struct variables and reported their mean as F1 average.

Table 4 summarises the resulting type-inference accuracy. At optimisation level O2 the F1 average climbed to 0.798, whereas at O0 it stayed at 0.701. For base variables the F1-score rose from 0.748 (O0) to 0.777 (O2); for struct variables it increased from 0.654 (O0) to 0.819 (O2). Thus the inlining-heavy O2 setting achieved the highest accuracy, while the greater amount of noisy information present in O0 tended to depress performance.

The TYGR paper reports overall F1-scores on x64 binaries of 0.825 at O0, 0.782 at O1, 0.777 at O2 and 0.748 at O3. Our O2 result exceeds TYGR's by 2.1 percentage points, whereas our O0 result falls short by 12.4 points. We believe this discrepancy stems from differences in the sample mix: our data set contains many binaries that make intensive use of system calls, which diversify pointer and structure access patterns. At O2, however, the increased graph density produced by inlining allowed R-GCN to exploit structural cues more effectively, leading to a higher score than TYGR's at that level.

Opt. Level	base			struct			F1 average
	Precision	Recall	F1-score	Precision	Recall	F1-score	(base, struct)
O0	0.8315	0.6936	0.7484	0.7212	0.6328	0.6544	0.7014
O1	0.7651	0.5233	0.5875	0.7737	0.7059	0.7173	0.6524
O2	0.8248	0.7464	0.7772	0.8950	0.7926	0.8190	0.7981
O3	0.8192	0.5939	0.6642	0.5971	0.5389	0.5352	0.5997

Table 4: Test results for base and struct variables

4.3 Performance comparison of type inference

Opt	RGCN	GGNN	GAT	GCN
O0	74.84	**77.42**	70.66	60.51
O1	58.75	**67.73**	60.59	55.0
O2	**77.72**	72.63	68.16	50.41
O3	**66.42**	46.14	47.97	37.70

Table 5: F1-score (%) for base variables at each optimisation level

Opt	RGCN	GGNN	GAT	GCN
O0	63.28	45.71	**69.46**	21.25
O1	**71.73**	50.4	58.88	37.41
O2	**81.90**	55.87	67.02	38.62
O3	**53.51**	48.42	36.19	8.98

Table 6: F1-score (%) for struct variables at each optimisation level

Table 5 summarises the F1-scores (%) for base variables at optimisation levels O0–O3. At O0, GGNN achieves the highest score (77.42 %), with R-GCN (74.84 %) and GAT (70.66 %) following, while GCN lags behind at 60.51 %. GGNN continues to lead at O1 (67.73 %), but its margin narrows as GAT records 60.59%and R-GCN 58.75 %. When the optimiser is tightened to O2, R-GCN surges to 77.72 %, overtaking GGNN (72.63 %) and GAT (68.16 %). Under the most aggressive setting, O3, all models incur losses, yet R-GCN remains on top at 66.42 %, whereas GAT and GGNN decline to 47.97%and 46.14%respectively; GCN stays lowest throughout, dropping to 37.70%at O3.

Table 6 reports the F1-scores (%) for struct variables. Here GAT leads at O0 with 69.46 %, edging out R-GCN's 63.28 %, while GGNN and GCN trail at 45.71%and 21.25 %. From O1 onward, however, R-GCN becomes dominant, rising to 71.73%at O1 and peaking at 81.90%at O2. Even though its score falls to 53.51%at O3, R-GCN still keeps a clear advantage over GGNN (48.42 %), GAT (36.19 %) and GCN (8.98 %). GGNN's performance on struct variables remains modest across all levels (45.71–55.87 %), and the edge-agnostic GCN consistently stays below 40 %, confirming its weakness in capturing structure-specific semantics.

Table 7 reports the prediction accuracy for base-type variables in x64 binaries compiled at optimization level O0. Although RGCN delivers the best accuracy for most types, three exceptions—bool, i64*, and u16 show a slight edge for other architectures. For bool, GCN surpasses RGCN by 3.9 percentage points, and for i64* GCN also posts a marginally higher value. In the u16 type, GAT exceeds RGCN by about 3.93 %, and GCN outperforms RGCN by roughly

Type	RGCN Acc(%)	GGNN Acc(%)	GAT Acc(%)	GCN Acc(%)
type	**57.07**	23.79	20.00	17.16
void*	**44.48**	26.72	23.80	19.24
void**	**22.34**	14.74	4.71	3.50
struct	**57.11**	37.05	35.05	24.19
union	**22.02**	0.40	0.40	0.20
enum	**45.14**	26.04	23.85	22.73
bool	65.69	56.45	60.99	**69.59**
char	**73.35**	35.32	31.17	12.37
i16	**38.62**	0.00	0.00	0.00
i32	**82.55**	73.96	80.89	74.05
i64	**65.30**	52.65	46.39	44.85
u16	67.90	56.69	71.83	**76.17**
u32	**75.18**	47.80	43.19	50.70
u64	**69.35**	56.87	55.29	50.09
u128	**87.29**	0.00	0.00	0.00
f32	**45.08**	23.02	34.53	37.17
f64	**87.43**	65.60	80.82	71.26
f128	0.00	0.00	0.00	0.00
struct*	**89.17**	84.94	84.46	83.42
union*	**41.56**	33.53	30.27	33.57
enum*	**44.35**	5.24	0.75	1.09
char*	**59.03**	42.37	40.99	36.94
i16*	**39.90**	0.00	0.00	0.00
i32*	**53.05**	41.42	36.27	32.59
i64*	7.44	4.70	1.37	**7.83**
u16*	**50.89**	2.38	20.09	9.28
u32*	**34.43**	16.23	11.35	12.56
u64*	**52.55**	36.68	34.64	28.00
f32*	**18.72**	0.00	0.00	0.00
f64*	**32.85**	0.00	0.00	0.00

Table 7: Inference accuracy for Base variable types across models

Type	RGCN Acc(%)	GGNN Acc(%)	GAT Acc(%)	GCN Acc(%)
type	**19.78**	3.54	3.30	0.00
void*	**65.88**	51.53	53.18	39.53
void**	**72.00**	0.00	0.00	0.00
struct	**44.48**	7.59	9.83	0.00
union	**70.62**	8.53	20.85	0.00
enum	**49.37**	7.39	6.67	0.00
bool	**66.36**	43.06	55.26	42.72
char	**50.00**	0.00	1.76	10.59
i16	58.62	0.00	**62.07**	0.00
i32	**60.90**	15.45	42.10	0.00
i64	**69.16**	9.09	23.38	7.47
u16	**79.69**	67.19	45.31	64.06
u32	49.36	68.79	57.64	**86.46**
u64	**75.48**	13.89	51.72	11.97
f64	**82.91**	12.82	34.19	21.37
struct*	**84.21**	71.33	77.16	78.89
union*	0.00	0.00	0.00	0.00
enum*	0.00	0.00	0.00	0.00
char*	**59.89**	23.57	14.61	1.72
i32*	8.70	46.74	**82.61**	0.00
i64*	**7.69**	0.00	0.00	0.00
u16*	0.00	0.00	0.00	0.00
u32*	**5.34**	0.00	0.00	0.00
u64*	**31.54**	0.00	0.00	0.00
f64*	0.00	0.00	0.00	0.00

Table 8: Inference accuracy for Structure variable types across models

8.27 %. Apart from these exceptions, the overall trend still favors RGCN. Nevertheless, minor signs of shrinking performance gaps in the signed/unsigned 16-bit family and several pointer types suggest that biases in data distribution or instruction patterns may warrant closer examination.

Table 8 lists the prediction accuracy for struct-type variables across the four architectures. RGCN continues to achieve the highest accuracy for the majority of types in structure-member inference, yet other models prevail for several cases. For instance, in the i16 type GAT exceeds RGCN by about 3.45 %. In the u32 type, GAT, GGNN, and GCN outperform RGCN by 8.28 %, 19.43 %, and 37.1 %, respectively. The most striking gap appears in the i32* type: while RGCN attains only 8.70 %, GGNN reaches 46.74%(a 4.4-fold relative improvement) and GAT achieves 82.61%(an 8.5-fold relative improvement), highlighting RGCN's weakness on pointer types. Excluding these outliers, RGCN generally retains a 10–30%margin and remains the top performer overall.

5 Further Analysis

5.1 Feature Analysis

Table 9 enumerates the most frequent mispredicted labels for each base-type variable. Across all four architectures—including RGCN—many types most often receive struct* as the incorrect label; for example, void* nodes are misclassified as struct* 33.00%of the time with RGCN and 47.26%with GAT, and a similar tendency appears for struct and i64. Conversely, for the u32 type, misclassification into the signed integer i32 reaches 10.21%with RGCN, 30.51%with GAT, and 26.11%with GCN, indicating that sign recognition remains challenging. Although the current node embeddings encode simple traits such as "whether the high bit is 1 or 0," they may still miss the broader context of arithmetic sign or zero extension. Augmenting the feature set with cast instructions or mask operations that follow arithmetic results could help strengthen the signed/unsigned distinction. Additional patterns also emerge—for instance, char is often confused with bool (61.51%in GCN), and the floating-point pointer f32* is sometimes replaced by i32* or struct*—highlighting further avenues for reducing misclassification.

Table 10 lists, for structure-related variables, the most frequently assigned incorrect label. As with base-type variables, the overwhelmingly most common incorrect label across all GNN architectures is struct*. For instance, the unknown label type is misclassified as struct* in 52.99% of RGCN cases, 70.33% for GGNN, 79.12% for GAT, and 83.52% for GCN.

Confusion between signed and unsigned integers is also pronounced. i32 is mistaken for u32 in 9.44%of RGCN predictions, 19.33%for GAT, 49.69%for GGNN, and 65.49%for GCN, so GCN's error rate exceeds RGCN's by more than a factor of seven. The reverse direction (u32 → i32 or enum) occurs in roughly 15–22%of cases, implying that the embeddings do not adequately encode integer signedness. Bit-width proximity further amplifies errors: for

	RGCN	GGNN	GAT	GCN
Type	Wrong(%)	Wrong(%)	Wrong(%)	Wrong(%)
type	struct* (29.14%)	struct* (48.93%)	struct* (55.08%)	struct* (46.99%)
void*	struct* (33.00%)	struct* (43.93%)	struct* (47.26%)	struct* (55.99%)
void**	struct* (47.11%)	struct* (47.26%)	struct* (61.63%)	struct* (61.93%)
struct	struct* (30.11%)	struct* (37.59%)	struct* (42.54%)	struct* (54.03%)
union	char (33.63%)	bool (32.23%)	bool (31.13%)	bool (36.94%)
enum	struct* (19.38%)	i32 (35.40%)	i32 (42.60%)	i32 (33.63%)
bool	struct* (19.67%)	struct* (28.87%)	struct* (23.51%)	struct* (19.45%)
char	struct* (12.40%)	bool (32.13%)	bool (40.14%)	bool (61.51%)
i16	struct* (24.55%)	u16 (31.70%)	u16 (43.97%)	u16 (61.16%)
i32	struct* (9.31%)	struct* (12.91%)	struct* (9.83%)	struct* (11.53%)
i64	struct* (12.67%)	struct* (16.45%)	struct* (18.75%)	struct* (23.13%)
u16	struct* (19.52%)	struct* (13.15%)	struct* (11.70%)	struct* (11.06%)
u32	i32 (10.21%)	i32 (21.89%)	i32 (30.51%)	i32 (26.11%)
u64	struct* (20.67%)	struct* (26.58%)	struct* (28.87%)	struct* (31.64%)
u128	u64 (6.78%)	u64 (88.14%)	u64 (94.07%)	u64 (82.20%)
f32	struct* (32.13%)	i32 (25.66%)	i32 (20.62%)	struct* (17.03%)
f64	struct* (7.13%)	struct* (9.98%)	struct* (10.32%)	struct* (8.87%)
f128	i64 (63.41%)	struct (68.29%)	type (63.41%)	i64 (63.41%)
struct*	char* (2.86%)	char* (4.64%)	char* (3.84%)	char* (4.73%)
union*	struct* (49.90%)	struct* (49.71%)	struct* (48.98%)	struct* (52.98%)
enum*	struct* (30.79%)	struct* (43.64%)	struct* (48.35%)	struct* (51.84%)
char*	struct* (29.51%)	struct* (38.59%)	struct* (40.59%)	struct* (44.52%)
i16*	u16* (20.73%)	struct* (44.56%)	struct* (31.09%)	struct* (57.51%)
i32*	struct* (21.20%)	struct* (29.82%)	struct* (40.01%)	struct* (44.41%)
i64*	struct* (36.59%)	struct* (38.94%)	struct* (47.75%)	struct* (49.71%)
u16*	struct* (27.78%)	struct* (37.45%)	struct* (41.62%)	struct* (49.65%)
u32*	struct* (33.09%)	struct* (38.53%)	struct* (46.05%)	struct* (51.65%)
u64*	struct* (26.50%)	struct* (32.23%)	struct* (39.23%)	struct* (44.95%)
f32*	i32* (57.64%)	i32* (57.14%)	struct* (71.43%)	struct* (38.92%)
f64*	struct* (31.18%)	struct* (32.61%)	struct* (40.77%)	struct* (40.53%)

Table 9: Most Frequently Misclassified Labels for Base Variables by Architecture

	RGCN	GGNN	GAT	GCN
Type	Wrong(%)	Wrong(%)	Wrong(%)	Wrong(%)
type	struct* (52.99%)	struct* (70.33%)	struct* (79.12%)	struct* (83.52%)
void*	struct* (14.12%)	struct* (35.29%)	struct* (29.18%)	struct* (48.94%)
void**	struct* (18.00%)	struct* (42.00%)	struct* (42.00%)	void* (36.00%)
struct	struct* (20.34%)	struct* (26.03%)	struct* (42.59%)	struct* (64.48%)
union	void* (5.69%)	u32 (45.50%)	u32 (27.01%)	u32 (59.24%)
enum	i32 (12.97%)	u32 (43.78%)	i32 (32.61%)	u32 (43.42%)
bool	char (20.77%)	u32 (26.32%)	i32 (27.42%)	char (38.60%)
char	enum (21.76%)	u32 (58.82%)	i32 (55.29%)	struct* (36.47%)
i16	u16 (34.48%)	u16 (72.41%)	u32 (17.24%)	u16 (68.97%)
i32	u32 (9.44%)	u32 (49.69%)	u32 (19.33%)	u32 (65.49%)
i64	struct (8.77%)	struct* (67.86%)	struct* (34.42%)	struct* (77.60%)
u16	u64 (4.69%)	u32 (12.50%)	i16 (31.25%)	u32 (12.50%)
u32	enum (21.50%)	i32 (15.45%)	i32 (18.15%)	struct* (4.94%)
u64	char* (6.44%)	u32 (26.61%)	struct* (13.22%)	u32 (40.50%)
f64	i32 (10.26%)	u32 (48.72%)	struct* (39.32%)	u32 (39.32%)
struct*	char* (8.48%)	struct (8.44%)	u32 (10.72%)	u64 (13.42%)
union*	u32 (70.00%)	u32 (40.00%)	i32 (40.00%)	u32 (70.00%)
enum*	struct* (59.57%)	struct* (85.11%)	struct* (91.49%)	struct* (97.87%)
char*	struct* (30.09%)	struct* (61.67%)	struct* (70.05%)	struct* (86.63%)
i32*	struct* (60.87%)	struct* (41.30%)	struct* (8.70%)	struct* (47.83%)
i64*	struct* (61.54%)	struct* (92.31%)	struct* (88.46%)	struct* (88.46%)
u16*	struct (38.10%)	char* (58.73%)	struct* (93.65%)	struct* (95.24%)
u32*	struct* (45.99%)	struct* (67.95%)	struct* (85.16%)	struct* (91.10%)
u64*	struct* (36.10%)	struct* (50.62%)	struct* (81.33%)	struct* (89.21%)
f64*	struct* (66.67%)	struct* (83.33%)	struct* (100.00%)	struct* (100.00%)

Table 10: Most Frequently Misclassified Labels for Struct Variables by Architecture

16-bit integers, i16 is mislabeled as u16 in 34.48%of RGCN outputs, 72.41%of GGNN outputs, and 68.97%of GCN outputs.

5.2 Impact of DFG-Derived Properties on GNN Type Inference

To elucidate the relationship between DFG-derived structural properties and the correctness of GNN-based type inference, we analyzed both base and struct variables by computing, for each node, its in-degree, out-degree, degree centrality, betweenness centrality, and eigenvector centrality on the DFG. We focused on the GCN and GAT architectures, which demonstrated superior accuracy for certain types compared to RGCN, based on our earlier error analysis. This approach aims to reveal how differences in the way each model leverages graph-structural features contribute to success or failure in type prediction. For these experiments, we employed a logistic regression with an L1 penalty, using the liblinear solver and a maximum of 2000 iterations to ensure convergence stability. To address class imbalance, class_weight was set to "balanced," and random_state was fixed at 0 for reproducibility.

Furthermore, we exclude from the analysis any classes for which all predictions are either entirely correct or entirely incorrect, as well as classes with fewer than ten evaluation samples.

Model	Data set	ROC-AUC	eigen_centrality	deg_centrality	betweenness	in_deg	out_deg
RGCN	Base	0.601	0.3479	-0.0770	-0.0317	0.0199	0.0000
RGCN	Struct	0.758	-1.3435	4.2249	-1.7277	-0.3547	-0.0006
GAT	Base	0.581	0.3652	-0.2504	-0.0572	0.0499	0.0000
GAT	Struct	0.734	-0.9867	4.1682	-2.2132	-0.2213	-0.0000
GCN	Base	0.574	0.3484	-0.2484	-0.0659	0.0495	0.0000
GCN	Struct	0.835	-0.8968	5.1661	-2.8307	-0.4445	-0.0002
GGNN	Base	0.574	0.3285	0.1855	-0.0771	0.0507	-0.0002
GGNN	Struct	0.818	-0.9985	5.0008	-2.6030	-0.4637	-0.0000

Table 11: Logistic-regression coefficients (L1, class_balanced) for each GNN and data set

Table 11 summarizes the results of logistic-regression analyses applied to the Base and Struct variables for four GNN architectures—RGCN, GAT, GCN, and GGNN. With respect to ROC-AUC, the Base setting yields closely aligned scores (+0.574 to +0.601), while in the Struct setting all scores rise (+0.734 to +0.835). For eigen-centrality, all models are positive in Base (+0.328 to +0.365) but negative in Struct (−0.896 to −1.343). For degree centrality, all models are slightly negative to slightly positive in Base (−0.077 to +0.185) but strongly positive in Struct (+4.168 to +5.166). For betweenness, all models assume negative values, spanning −0.0317 to −0.077 in Base and −1.727 to −2.83 in Struct. For in-degree, all models are positive in Base (+0.019 to +0.050) but negative in Struct (−0.221 to −0.463). For out-degree, all models remain almost zero under both Base and Struct conditions.

We next focus our analysis on struct variables that exhibited higher AUC results.

Table 12 through Table 15 list the logistic-regression coefficients and AUC scores for a type-identification task restricted to structure variables. Four back-end GNNs RGCN, GCN, GAT, and GGNN provide the input features.

Focusing first on struct*, its AUC is extremely high—above 0.96—in all four tables. Degree centrality is a large positive value, approaching two digits: +9.94 for RGCN, +11.91 for GCN, +13.82 for GAT, and +10.92 for GGNN, making it a notably strong feature compared with the others. The in-degree coefficient is strongly negative (-2.39 to -2.87), while the out-degree coefficient remains only slightly negative. In other words, a consistent connectivity pattern is extracted:

Type	AUC	in_deg	out_deg	deg_centrality	betweenness	eigen_centrality
type	0.8610	-0.0407	-0.0099	3.5359	-1.6765	-3.0799
void*	0.7860	-0.4733	-0.0709	1.7233	-0.2800	0.0676
struct	0.6730	0.0000	0.0000	-0.7982	0.5347	0.2521
union	0.9820	0.0000	0.0000	-3.6088	0.0000	-1.0578
enum	0.3850	0.0267	0.0010	-0.2015	0.2011	0.0000
bool	0.3430	2.0845	1.9238	-3.0047	-1.8449	-0.2770
char	0.8190	0.0000	0.0000	0.0000	0.0000	1.8131
i16	0.6830	0.0000	0.0000	-0.4123	0.1206	0.4439
i32	0.5370	-0.1308	-0.0733	0.3179	0.1469	-0.4497
i64	0.7010	0.4060	0.6202	0.4763	-0.4264	-0.9434
u16	0.8060	-0.7682	-0.0632	1.8887	0.0000	-0.2323
u32	0.7010	-0.3204	-0.1949	0.3072	0.9086	-1.2666
u64	0.9360	1.3089	0.4577	-2.5428	-0.2465	-3.6303
struct*	0.9670	-2.7390	-0.2682	9.9427	-3.9810	-2.0593
char*	0.9100	0.0000	-0.5789	3.9520	0.0000	-1.1484

Table 12: Logistic regression coefficient for each Struct type of RGCN model

Type	AUC	in_deg	out_deg	deg_centrality	betweenness	eigen_centrality
void*	0.8590	-0.9081	-0.0979	2.1168	-0.3564	0.0000
bool	0.9290	0.0000	0.0000	0.2345	1.3772	-7.2623
char	–	0.0000	0.0000	0.8176	-0.2739	0.0000
i64	0.7280	0.2684	0.2416	0.3244	0.0000	-2.2686
u16	0.7040	-0.2486	-0.1503	1.9488	-0.5429	-0.6058
u32	0.7400	0.2544	0.4702	-2.1724	0.0000	1.5836
u64	0.6150	-0.0000	-0.0005	0.1942	-0.0451	-0.1544
struct*	0.9680	-2.8774	-0.4142	11.9105	0.0000	-3.0433

Table 13: Logistic regression coefficient for each Struct type of GCN model

Type	AUC	in_deg	out_deg	deg_centrality	betweenness	eigen_centrality
type	0.7080	-0.3047	-0.3084	2.4053	-1.0323	-2.1710
void*	0.7100	-0.2574	-0.1317	1.0894	0.1455	-0.1154
struct	0.6730	0.0379	0.0420	-0.8511	0.4303	0.6680
union	0.8060	0.0000	-0.3940	3.9508	-0.3520	0.0000
enum	0.5660	0.1912	0.0234	0.0000	-0.2027	-0.2775
i16	0.8170	0.0000	0.0000	1.6397	-0.8720	-0.9736
i32	0.6310	-0.1318	-0.0000	-0.4623	-0.2988	0.4282
i64	0.8560	1.3412	1.2383	-0.3802	-1.5398	-2.6727
u16	0.5470	-0.4331	-0.0212	0.0000	0.0000	1.1046
u32	0.5420	0.0493	0.1723	-0.0899	-0.4303	0.2241
u64	0.6600	0.1471	0.1921	0.0000	-0.2786	-1.0348
struct*	0.9690	-2.3996	-0.0966	13.8245	1.0779	-3.2777
char*	0.8800	0.7943	0.4749	-3.0298	0.0000	-2.4509

Table 14: Logistic regression coefficient for each Struct type of GAT model

struct pointers are rarely written to from outside, yet they receive many internal reads and form local hubs. Betweenness is small in absolute value across all models, suggesting that struct pointers themselves are unlikely to serve as intermediaries on shortest paths.

For union nodes, the only striking feature is their extremely high degree centrality. Some models treat this degree centrality as a positive signal of +3.95, whereas others treat it as a negative one of -3.60; in both cases, the decision still hinges on the same cue: a union's unusually large degree centrality dominates the classification.

With char*, the sign of the degree-centrality coefficient flips by model—positive in RGCN, strongly negative in GAT, positive again in GGNN—while eigenvector centrality stays negative (-1.14 to -3.46) across the board. Thus a char* node operates as a hub locally but does not stand out globally. Differences in message-passing implementations invert the degree-centrality sign, yet all models exploit the same structural property.

Type	AUC	in_deg	out_deg	deg_centrality	betweenness	eigen_centrality
type	0.6850	-0.0571	-0.0003	0.0000	0.0000	-2.6667
void*	0.7100	-0.6527	-0.1605	1.5223	0.0953	-0.4661
struct	0.7530	0.0001	0.0461	-1.4496	0.0954	0.8170
union	0.9310	0.0000	0.0000	2.5666	0.0000	-2.2044
enum	0.8620	0.4786	0.2485	-1.8551	0.0000	-2.6526
bool	0.9290	0.0000	0.8863	3.4015	-5.1248	1.6265
i32	0.7060	-0.1570	-0.1145	1.3770	0.6107	-1.3717
i64	0.6990	0.0645	0.1962	0.9617	-2.3776	-1.4922
u16	0.6890	-0.1850	-0.3124	2.2342	-0.6217	-0.8120
u32	0.8320	1.0329	0.1677	-4.9475	1.5130	2.1410
u64	0.6070	-0.1924	-0.0041	0.4246	-0.2810	-0.1058
struct*	0.9600	-2.3914	-0.0112	10.9213	0.0000	-2.4106
char*	0.9370	-0.9948	-0.1406	3.0138	-0.9881	-3.4634
i32*	0.9400	-0.5247	-0.5322	0.0000	-4.4496	5.3288

Table 15: Logistic regression coefficient for each Struct type of GGNN model

The bool type shows strong model dependence. RGCN's AUC is only 0.343, barely better than random, whereas GCN and GGNN both reach 0.929. RGCN assigns positive in- and out-degree weights and a large negative degree-centrality term of -3.0. GGNN shows zero in-degree, positive out-degree, and a positive degree-centrality weight, while GCN displays near-zero degrees, a small positive centrality, betweenness +1.37, and eigenvector centrality -7.26. Each architecture captures Boolean broadcast patterns in its own way, reflecting how the model shares weights and distinguishes edge types.

For i16, under RGCN degree centrality is slightly negative -0.41 while betweenness +0.12 and eigenvector centrality +0.44 are weakly positive; in contrast, under GAT degree centrality is +1.63, betweenness -0.87, and eigenvector centrality -0.97, showing sign reversals and increased magnitudes and indicating clear differences in which features are influential across the models. In both models, in-degree and out-degree are approximately zero, suggesting that directed degrees contribute little to i16 discrimination. Taken together, these results imply that RGCN tends to capture i16 as quasi-bridge that connect to a small number of high-influence neighbors, whereas GAT tends to capture i16 as small, locally high-degree hub.

For i32*, the coefficients appear only in GGNN. Both degree terms are slightly negative, betweenness drops to -4.45, and eigenvector centrality climbs to +5.33. A 32-bit pointer thus forms a large local hub that never becomes a transit point. GGNN's sequential message passing captures this clear pattern most effectively, leading to high classification accuracy.

For u32, the coefficient on degree centrality is strongly negative (-2.17 to -4.95), so a highly connected node is less likely to be u32. Put differently, u32 nodes tend to form high-degree patterns that—paradoxically—work against correct type identification. This graph-attachment style, driven solely by bit width, directly shows up in the logistic-regression weights.

5.3 Analyzing Characteristics of Each GNN Model

Analysis of struct shows a consistent tendency across all models: in-degree coefficients are strongly negative, out-degree coefficients are slightly negative, and degree centrality is extremely strongly positive. We interpret the consistent elevation of degree centrality in every model as arising from the fact that struct is repeatedly used in the code in nearly identical ways. Concretely, the structure corresponds to a very clear and stable graph pattern in which the node supplies many local reads while receiving almost no external writes. Because this pattern is captured by simple, strong features such as degree and local density, it is detectable by any of the examined architectures: GCN and GGNN, which average and propagate neighborhood information; GAT, which emphasizes locality via attention weights; and RGCN, which handles edge-type–specific propagation. The combination of these indicators suggests that, on the DFG, the node behaves as a local hub that is rarely written to but fans out reads to multiple neighbors. In other words, struct* plays the role of providing many peripheral reads while not being actively updated, which in turn explains the consistent trend observed in structure-variable prediction performance.

For i16, under RGCN degree centrality is slightly negative while betweenness and eigenvector centrality are weakly positive. Under GAT these signs are reversed and the magnitudes increase. From this we infer that RGCN tends to capture i16 as a quasi-bridge that connects to a small number of high-influence neighbors, whereas GAT tends to capture i16 as a small, locally high-degree hub; thus i16 appears to exhibit at least two distinct occurrence patterns. Because GAT's mechanism emphasizes such locally high-degree hubs through attention weights, it is well matched to this pattern and therefore achieved better performance than RGCN.

For u32, both GGNN and GCN showed significantly higher eigenvector centrality compared to R-GCN and GAT, while also demonstrating relatively strong negative degree centrality. This suggests that u32 nodes function as concentrated junctions in the DFG—few in number, but heavily referenced by influential neighbors. Consequently, GCN, which captures features that reflect the influence of adjacent nodes, proved advantageous and achieved superior performance in structure type prediction tasks compared to other methods.

5.4 Limitations and Threats to Validity

The evaluation presented in this paper is limited to two components: (i) baseline results obtained by retraining the official TYGR implementation, and (ii) a simple statistical study in which we compute in-degree, out-degree, degree centrality, betweenness centrality, and eigenvector centrality for every variable node and compare their distributions across type labels. In other words, we have not yet carried out any clustering aimed directly at detecting structure boundaries, and the intended one-to-one correspondence between the number of structures and the number of clusters has not been verified quantitatively.

In our experiments we went beyond the standard type-identification task and examined how well node embeddings capture relationships inside structures. To that end we compared four GNN architectures and inspected the logistic-regression coefficients for each type. The analysis revealed several robust feature signatures that hold across models: structure pointers (struct*) are consistently characterized by extremely high degree centrality and almost negative in-degree; union nodes stand out as isolated giants; and char* nodes act as local hubs that fade at the global level. These findings suggest that TYGR embeddings encode not only type information but also latent cooperation among fields and characteristic access patterns.

As a next step we plan a two-stage pipeline. First, starting from struct* nodes, we will cluster neighboring nodes that share similar centrality patterns using a density-based algorithm such as DBSCAN. Second, we will automatically—or semi-automatically—learn whether each cluster truly represents members of the same structure. The rules extracted from the coefficient analysis (e.g., high degree centrality, low betweenness, matching eigenvector-centrality signs) can serve as an initial filter, while embedding proximity will be used as an integrated scoring metric in the subsequent stage.

6 Conclusion

In this paper, we presented a method for analyzing binary code by applying multiple graph neural network architectures to the task of inferring structure member types and identifying grouping relationships. Rather than treating the data flow graph as a passive intermediate form, we interpreted it as a learnable representation that reflects both semantic behavior and memory access dependencies. Through comparative experiments on real-world binary programs, we showed that the choice of neural network architecture significantly affects the quality and reliability of type inference results.

While previous work has mainly focused on a single model, our study examined additional architectures such as the gated graph neural network and the graph attention network. These models demonstrated valuable characteristics. The gated graph neural network was especially effective at capturing long distance dependencies in memory operations, which is important for understanding structure members that are accessed in nonlocal contexts. The graph attention network, on the other hand, offered robustness in the presence of noisy or ambiguous edges by adjusting the influence of each neighboring node. Although the standard graph convolutional network achieved lower accuracy, it served as a useful baseline and helped reveal the limitations of uniform message passing.

Beyond inference accuracy, our analysis of structural features in the graph revealed that variables associated with structure members tend to exhibit specific topological patterns. For example, nodes representing structure pointers commonly show high degree centrality and low inward connectivity, suggesting that structural roles are reflected in the geometry of the data flow graph itself. This insight supports the idea that graph neural networks are capable of capturing not only type-level distinctions but also deeper forms of structural organization.

As future work, we aim to extend this approach by constructing a clustering framework that detects and groups related variables based on both their learned embeddings and their structural properties in the graph. This will allow us to automatically infer structure layouts and model program state transitions in a more abstract and coherent way. Such a system could benefit a wide range of applications, including decompilation, vulnerability triage, firmware analysis, and protocol understanding. Our findings highlight the importance of using a variety of graph-based models to fully capture the complexity of compiled programs and recover the semantic structure that underlies low-level code.

References

[1] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. 2024. Evaluating the Effectiveness of Decompilers. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (Jul 2024), 491–502. doi:10.1145/3650212.3652144

[2] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting Decompiler Output with Learned Variable Names and Types. *Proceedings of the 31st USENIX Security Symposium* (Aug 2022), 4327–4343.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct 1991), 451–490. doi:10.1145/115372.115320

[4] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. 2011. Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. *Communications in Computer and Information Science* 200 (Aug 2011), 72–86.

[5] Christophe Hauser, Yan Shoshitaishvili, and Ruoyu Wang. 2017. Poster: Challenges and Next Steps in Binary Program Analysis with angr. *EuroS&P Poster Track* (Apr 2017).

[6] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (Oct 2018), 1667–1680. doi:10.1145/3243734.3243866

[7] Hex-Rays SA. 2024. IDA Pro Disassembler and Debugger. *Software Tool* (2024). https://hex-rays.com/ida-pro

[8] Ali R. Hurson and Krishna M. Kavi. 2007. Dataflow Computers: Their History and Future. *Wiley Encyclopedia of Computer Science and Engineering* (2007). doi:10.1002/9780470050118.ecse102

[9] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (Sep 2016). https://arxiv.org/abs/1609.02907

[10] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (Feb 2011), 251–268.

[11] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2015. Gated Graph Sequence Neural Networks. *arXiv preprint arXiv:1511.05493* (Nov 2015). https://arxiv.org/abs/1511.05493

[12] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering Types in Binary Programs Using Machine Learning. *Lecture Notes in Computer Science* 11543 (Jun 2019), 288–308. doi:10.1007/978-3-030-22038-9_14

[13] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery – A Case Study. *Proceedings of the ACM Asia Conference on Computer and Communications Security* (May 2022), 602–615. doi:10.1145/3488932.3497764

[14] National Security Agency. 2019. Ghidra. *Software Reverse Engineering Suite* (2019). https://ghidra-sre.org/

[15] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. *SIGPLAN Notices* 51, 6 (June 2016), 27–41. doi:10.1145/2980983.2908119 Originally appeared in PLDI '16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.

[16] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: Fine-Grained Type Recovery from Binaries using Generative State Modeling. *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Aug 2021), 769–781. doi:10.1145/3468264.3468607

[17] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. *Lecture Notes in Computer Science* 10843 (Jun 2018), 593–607. doi:10.1007/978-3-319-93417-4_38

[18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. *arXiv preprint arXiv:1710.10903* (Oct 2017). https://arxiv.org/abs/1710.10903

[19] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (Nov 2024), 4554–4568. doi:10.1145/3658644.3670340

[20] Bin Zeng. 2012. Static Analysis on Binary Code. *Technical Report LU-CSE-12-001, Lehigh University* (Feb 2012).

[21] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-Chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. *Proceedings of the IEEE Symposium on Security and Privacy* (May 2021), 813–832. doi:10.1109/SP40001.2021.00051

[22] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, Mayur Naik, Yan Shoshitaishvili, Ruoyu Wang, and Aravind Machiry. 2024. TYGR: Type Inference on Stripped Binaries using Graph Neural Networks. *Proceedings of the 33rd USENIX Security Symposium* (Aug 2024), 4283–4300. doi:10.5555/3698900.3699140