DFA-Net: A Compiler-Specific Neural Architecture for Robust Generalization in Data Flow Analyses

Alexander Brauckmann
University of Edinburgh
Edinburgh, UK
alexander.brauckmann@ed.ac.uk

Anderson Faustino da Silva
State University of Maringá
Maringá, Brazil
andersonfaustino@gmail.com

Gabriel Synnaeve
Meta AI
Paris, France
gab@meta.com

Michael F. P. O'Boyle
University of Edinburgh
Edinburgh, UK
mob@inf.ed.ac.uk

Jeronimo Castrillon
TU Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Hugh Leather
Meta AI
Menlo Park, USA
hleater@meta.com

Abstract

Data flow analysis is fundamental to modern program optimization and verification, serving as a critical foundation for compiler transformations. As machine learning increasingly drives compiler tasks, the need for models that can implicitly understand and correctly reason about data flow properties becomes crucial for maintaining soundness. State-of-the-art machine learning methods, especially graph neural networks (GNNs), face challenges in generalizing beyond training scenarios due to their limited ability to perform large propagations. We present DFA-Net, a neural network architecture tailored for compilers that systematically generalizes. It emulates the reasoning process of compilers, facilitating the generalization of data flow analyses from simple to complex programs. The architecture decomposes data flow analyses into specialized neural networks for initialization, transfer, and meet operations, explicitly incorporating compiler-specific knowledge into the model design. We evaluate DFA-Net on a data flow analysis benchmark from related work and demonstrate that our compiler-specific neural architecture can learn and systematically generalize on this task. DFA-Net demonstrates superior performance over traditional GNNs in data flow analysis, achieving F1 scores of 0.761 versus 0.009 for data dependencies and 0.989 versus 0.196 for dominators at high complexity levels, while maintaining perfect scores for liveness and reachability analyses where GNNs struggle significantly.

CC '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1407-8/25/03
https://doi.org/10.1145/3708493.3712687

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Compilers**.

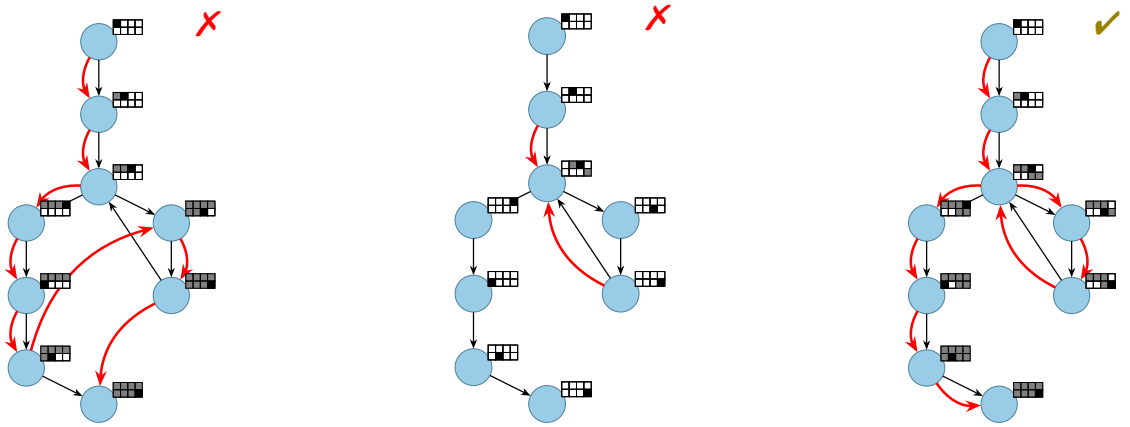
Keywords: Data Flow Analysis, Machine Learning, Neural Network Architecture

ACM Reference Format:
Alexander Brauckmann, Anderson Faustino da Silva, Gabriel Synnaeve, Michael F. P. O'Boyle, Jeronimo Castrillon, and Hugh Leather. 2025. DFA-Net: A Compiler-Specific Neural Architecture for Robust Generalization in Data Flow Analyses. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3708493.3712687

1 Introduction

Data flow analysis is a fundamental technique in compiler tasks [26, 28, 31], as it helps generate efficient and reliable code. It systematically examines how variables and expressions propagate through a program, providing crucial information for various optimizations and ensuring code correctness. At its core, data flow analysis investigates how values are defined, used, and modified during program execution [21]. A critical application is identifying optimization opportunities; compilers can make informed decisions about code transformations while preserving program semantics by understanding data flow. Standard optimizations such as constant folding, common subexpression elimination, and loop-invariant code motion heavily rely on data flow analysis information. Security analysis [2], which allows compilers to detect potential vulnerabilities like uninitialized variable use and buffer overflow [30], heterogeneous computing optimization [15], and smart contract verification on blockchain platforms [25] also utilize data flow analysis techniques. The importance of data flow analysis is growing as programming languages evolve and new hardware architectures emerge.

Recent research explored the incorporation of data flow understanding into machine learning (ML) models through GNNs representing program dependencies [1, 23], attention mechanisms focusing on relevant data flow patterns [19], and hybrid approaches combining traditional static analysis with



(a) RNNs **(b)** GNNs/Transformers **(c)** DFA-Net

Figure 1. Deep Learning models in predicting reachability. RNNs fail due to a lack of graph structure; GNNs/Transformers because of limited propagation reach. DFA-Net succeeds by using data-flow-inspired propagation, enabling the learning of graph and data-flow algorithms. Black edges show control flow; red edges show propagation flow.

learned components [18]. These methods aim to reconcile the pattern recognition ML capabilities with the stringent correctness requirements of compiler tasks. Developing ML models with robust reasoning capabilities regarding data flow is fundamental for ML compilers. Success in this area ensures code performance and the reliability that modern software systems demand. In this context, GNNs emerge as a promising approach for ML, demonstrating impressive capabilities in processing structured graph data. However, they display significant limitations when confronted with complex scenarios. While GNNs excel at handling straightforward analyses such as node classification [20, 42] and link prediction [24, 41], they often fall short when dealing with more intricate problems. This limitation is particularly evident in their inability to effectively generalize beyond their training data's specific patterns and structures [13]. The challenge lies in the fundamental architecture of GNNs, which typically relies on local neighborhood aggregation and may not effectively capture long-range dependencies or complex abstract relationships. Additionally, typical GNNs propagate fixed-sized tensors, regardless of the size of the graph, leading to memory bottlenecks. The generalization issue is exacerbated when dealing with scenarios that require complex reasoning capabilities.

This paper presents DFA-Net, a **D**ata **F**low **A**nalyses Neural **Net**work architecture for compilers that effectively addresses the generalization of data flow analyses in complex programs. DFA-Net distinguishes itself from conventional GNNs by its ability to mimic the reasoning process of a compiler while proportionally scaling memory allocation according to the program's complexity, thereby facilitating the management of increasingly complex code structures. Unlike traditional models, which follow specific propagation styles—such as RNNs that propagate linearly and GNNs and Transformers that aggregate local features—DFA-Net propagates according to data flow principles, allowing it to capture global features and providing a more comprehensive understanding of graph properties and data flows within programs, as showing in Figure 1.

The DFA-Net model captures data flow properties through a neural network architecture, enhancing the traditional data flow analysis framework with learnable components. Starting with a control flow graph (CFG), DFA-Net considers domains, features, direction, and flow functions. It operates within specific domains, including nodes, instructions, and variables, with each assigned distinct features. At the same time, the direction component facilitates propagation throughout the CFG to support both forward and backward analyses. By integrating these components, DFA-Net enhances the foundational data flow analysis framework, enabling effective generalization and knowledge acquisition regarding program properties.

Concretely, this paper makes the following contributions:

A novel neural network architecture. DFA-Net is inspired by how compilers analyze code – with data-flow analysis frameworks. It follows the same principles while keeping the framework's variation points learnable. It supports a variety of domains: Instructions, variables, and expressions, making it suitable for various predictive tasks in code analysis and compiler tasks.

A training methodology that enables strong generalization from simple to complex program graphs. The training methodology not only promotes strong generalization capabilities but also improves overall performance and model adaptability.

A systematic evaluation in data-flow analyses. We assess DFA-Net's learning capabilities in classical data flow analyses using the AnghaBench benchmark [11]. GNN+ProGraML [10] serves as the baseline. Our novel approach delivers previously unattained performance, enabling the model to learn general solutions applicable to larger, unseen graphs. DFA-Net achieves an F1-score of 0.999 across multiple data flow analyses, while ProGraML+GNN has difficulty generalizing beyond a small training set.

As a result of these contributions, we present findings demonstrating the feasibility of developing a strong generalization infrastructure that outperforms GNNs and well-established code representations.

2 Background

2.1 Data Flow Analysis

Data flow analysis is a framework used in compiler design and program analysis to systematically derive information about the possible values computed at various points in a program graph (Definition 2.1) [14]. It operates over a CFG (Definition 2.2).

Definition 2.1 (Graph)**.** A graph G is an ordered pair (N, E), where:

- N is a finite set of nodes.
- $E \subseteq N \times N$ is a set of edges representing node connections.

Definition 2.2 (CFG)**.** A control flow graph is a directed graph $CFG = (N, E, n_{entry}, n_{exit})$, where:

- N is the set of all program statements or basic blocks.
- E is the directed edges representing the control flow between nodes.
- $n_{entry} \in N$ is the unique entry node where execution begins.
- $n_{exit} \in N$ is the unique exit node where execution terminates.

In data flow analysis, a fact refers to a piece of information that can be determined about the state of a program at a particular point during its execution. These facts are typically represented as mathematical constraints or logical assertions, and they are used to reason about the program's behavior to perform compiler optimizations or analyses. Data flow analysis techniques, such as liveness analysis and reaching definitions, rely on accurately identifying and manipulating these facts to derive useful insights about the program's execution. The precise nature and representation of facts can vary depending on the data flow analysis. Still, they are a fundamental component of this widely-used program analysis and optimization approach. We can define the domain of data flow facts D as follows.

Definition 2.3 (Domain of data flow facts)**.** Let $\mathcal{P}$ be the set of all possible program properties or attributes that can be tracked during the data flow analysis. Then, the domain of data flow facts D is the power set of $\mathcal{P}$, that is $D = 2^{\mathcal{P}} = \{d \mid d \subseteq \mathcal{P}\}$. Each element $d \in D$ is a set of program properties or attributes representing a specific data flow fact. The set D contains all possible combinations of these properties, and each element of D represents a unique data flow fact.

Data flow equations give the relationships between facts and describe the relationships between data inputs and outputs within a system. They define how data transforms as it flows through a process, expressed as a set of equations specifying the output of each operation as a function of its inputs. In data flow analysis, IN (Definition 2.4) and OUT (Definition 2.5) sets are crucial for iterative algorithms that compute data flow facts by propagating information across the CFG.

Definition 2.4 (IN Set)**.** $\text{IN}(n \in N) \subseteq D$: The set of data flow facts entering node n before it executes.

Definition 2.5 (OUT Set)**.** $\text{OUT}(n \in N) \subseteq D$: The set of data flow facts exiting node n after it executes.

The relationships between these sets are given by the data flow Equations 1 and 2.

$$\text{OUT}(n) = f_n(\text{IN}(n)) \tag{1}$$

$$\text{IN}(n) = \bigoplus_{p \in \text{edges}(n)} \text{OUT}(p) \tag{2}$$

where:

- $f_n : 2^D \to 2^D$ is the transfer function at node n, modeling the effect of node n on the data flow facts.
- $\bigoplus$ is the meet operator that combines data flow facts from multiple predecessor nodes.
- $\text{edges}(n)$ is the set of immediate predecessor or successors nodes of n in the CFG, depending on the direction.

Data flow equations are solved iteratively until a fixed point is reached, meaning further iterations do not change the data flow sets as shown in Equation 3.

$$\forall n \in N: \quad \begin{cases} \text{IN}^{(i)}(n) & = \text{IN}^{(i-1)}(n) \\ \text{OUT}^{(i)}(n) & = \text{OUT}^{(i-1)}(n) \end{cases} \tag{3}$$

The convergence property of this framework is established through two essential conditions: monotonicity and finite height of lattice. Monotonicity ensures that successive iterations maintain a consistent direction of progression within the lattice structure. In contrast, the finite height condition provides an upper bound on the number of possible steps before reaching a fixed point. Together, these conditions prevent infinite ascending chains and guarantee that the iterative process eventually stabilizes at a solution.

2.2 ML for Compiler Tasks

The integration of ML techniques in compiler technology is rapidly advancing. Early applications utilized Recurrent Neural Networks (RNNs) primarily to predict potential bugs within functions based on their code structure [3] or assist in code completion suggestions [32]. While useful, RNNs struggled with the inherent complexity of program structures and long-range dependencies within code, limiting their broader application in compiler optimization.

GNNs, with their ability to model relationships within graph structures, have proven more suitable for compiler tasks. GNNs have been successfully applied to binary analysis [39, 43], vulnerability detection and code similarity checking [17, 36, 40]. Other applications involve identifying potential vulnerabilities through CFG analysis and predicting performance bottlenecks based on graph representations [9, 16, 27, 37]. Additionally, they include device mapping and determining optimal thread coarsening factors [8]. This underscores that graph-based representations can more effectively capture code's structural properties than sequence-based methods for compiler optimization tasks. Built upon GNN, GNN+ProGraML [10] effectively reasons about program-wide data flow, addressing issues faced by previous methods. While GNN+ProGraML enhances performance in downstream optimization tasks, GNNs can be computationally expensive, and their generalization to unseen code patterns remains challenging.

Transformers, known for capturing long-range dependencies, have emerged as powerful tools in compiler optimization. One application uses transformers to find binary code similarity [12]. Another is their use in code translation and text-to-code generation tasks [34]. Despite their strengths, the computational cost of training and inferring transformers and the need for extensive training data, which is significantly higher than for GNNs on domain specific graphs, remain significant hurdles.

Large Language Models (LLMs), with their capacity for understanding and generating natural language, are being explored for sophisticated compiler tasks. LLMs are being used for automated code refactoring, suggesting improvements to existing code based on best practices and style guidelines [22, 38]. They also show promise in automated code debugging, identifying and suggesting fixes for common programming errors [4, 5, 33]. In the context of compiler analyses, LLMDFA [35] investigates using LLMs for data flow analysis, aiming for a compilation-free and customizable approach by employing LLMs for source/sink extraction, data flow summarization, and path feasibility validation. This approach, however, encounters challenges such as LLM hallucinations and the difficulties of dealing with large functions or complex pointer operations. Further, the computational demands and data requirements for LLMs training and inference are substantial, motivating lightweight, task-specific models.

A key challenge across all these ML approaches lies in their ability to generalize beyond the training data. Compilers must handle various code styles, programming languages, and program structures. An ML model trained only on a specific dataset may fail to perform adequately when presented with unfamiliar code. Therefore, developing ML models that robustly generalize to unseen code patterns is crucial for their successful integration into real-world compiler systems, enabling the creation of more efficient and reliable software.

3 DFA-Net: Learnable Data-Flow Analyses

The DFA-Net model is designed to capture data flow properties through a neural network architecture. It is a model tailored for program analysis, enhancing the traditional data flow analysis framework, described in Section 2, with learnable components. DFA-Net merges traditional methodologies with cutting-edge ML techniques to create an architecture for robust generalization in data flow analyses, while being light-weight in comparison to Transformers and GNNs. Figure 2 presents an overview of the DFA-Net's architecture. Its components will be described in the following.

3.1 DFA-Net Architecture

In developing an effective learnable data flow framework, it is essential to address several key considerations, which will be elaborated upon in subsequent subsections. DFA-Net comprises core components, including the domain, features, direction, and learnable flow functions that utilize value set tensors to facilitate data propagation.

Domains. The analysis operates within specific domains, encompassing nodes, instructions, and variables.

Features. Each domain element, such as nodes, instructions, or variable identifiers, is associated with distinct features.

Direction. This component delineates the direction of propagation throughout the CFG.

Value sets. Value sets consist of the potential values that can be assigned during the runtime of the data flow analysis, specifically the IN and OUT sets. These sets serve as the working memory during the propagation process.

Learnable flow functions. These functions facilitate the propagation of facts through the graph. The *initialization* function initializes the value set, the *transfer* function dictates how values disseminate across nodes, the *meet* function combines data flow facts from multiple paths into a single value, and the *readout* function extracts pertinent information from the value set.

Integrating core components with learnable functions enhances data flow analysis, allowing the model to learn program properties. Additionally, the architecture employs a

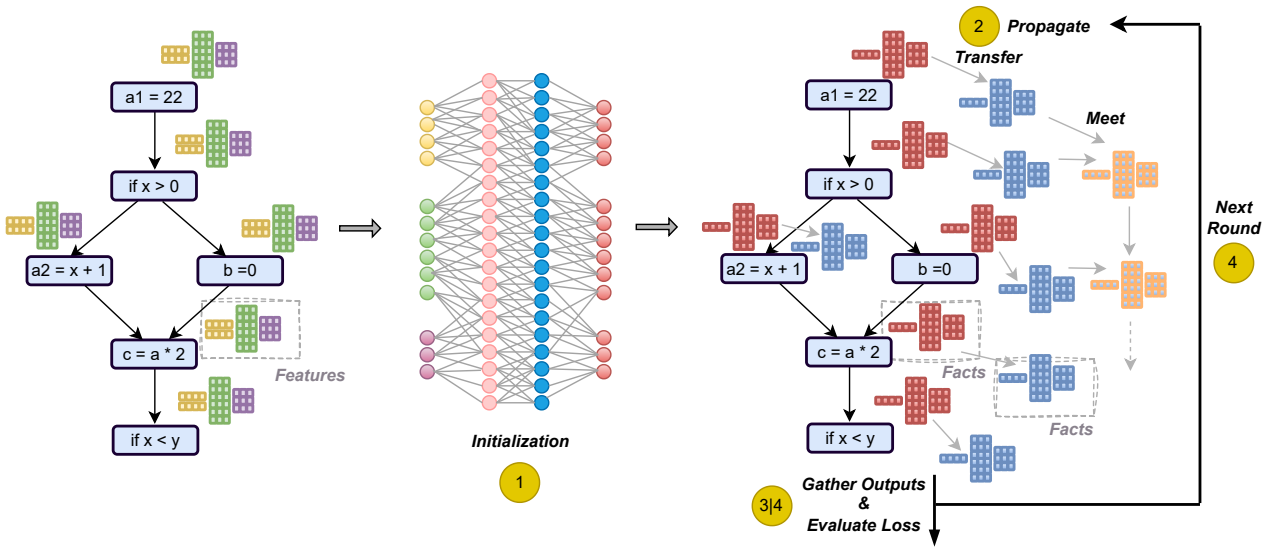


Figure 2. DFA-Net begins with an initialization function that generates starting values for each fact. In the propagation phase, an alternation of transfer and meet function applications determine how a node processes incoming values and produces output values. This is repeated until a fixed-point is reached. After propagation, the output function maps to a vector or scalar of the desired property. The training loop determines whether more rounds of optimization are needed.

curriculum learning strategy that gradually increases program complexity to ensure robust generalization.

3.2 Core Components: Domains, Features and Direction

Domains. In DFA-Net, facts are organized into domains $\mathbb{D} = \{D_1, D_2, \ldots, D_n\}$, which represent the key elements of interest within the CFG, such as the graph itself, nodes, instructions, and variables. The selection of domains significantly influences the model's expressiveness and memory usage, as different analyses may necessitate varying levels of domain complexity. Some facts can be multidimensional; for instance, if a node possesses a three-dimensional fact, it can accommodate three floating-point values per node. The analytical requirements and the computational resources at hand determine the dimensionality of these facts. The total number of facts is contingent upon the structure of the graph G and the defined domains $\mathbb{D}$. Facts are constructed based on examining the graph, meaning that the total count of facts cannot be established until the number of variables is identified. This characteristic distinguishes DFA-Net from conventional GNNs, which typically maintain a fixed memory size per node, causing a memory bottleneck when information from too many nodes is compressed into a finite tensor. In contrast, DFA-Net's memory allocation is dynamic and proportional to the number of elements within each CFG.

Features. Each domain is assigned specific features corresponding to their CFG roles. These features include node features, which provide unique identifiers for the nodes; instruction features, which capture fundamental attributes of each program instruction, such as its type; and variable features, which detail aspects such as whether a variable is being defined or utilized as an operand, along with other characteristics relevant to data flow facts, such as the variable's type.

Direction. To accommodate both forward and backward analyses within a cohesive framework, DFA-Net supports dual-direction propagation. This allows the model to learn both types of data flow analyses effectively. In a final stage, the most appropriate value set is selected based on the specific requirements of the analysis in question.

In the backward direction, values are propagated in post order, and in the forward direction, values are propagated in reverse post order. This is typical of standard data flow analyses but entirely different from GNNs.

3.3 Learnable Components: Value Sets and Flow Functions

DFA-Net passes around tensors to represent data flow values. It represents flow functions as many small learnable Multilayer perceptron (MLP) functions.

Value sets. The value sets serve as the working memory for propagating data flow facts across the CFG. Implemented as tensors, they encapsulate the state during the analysis. Consistent with the traditional data flow analysis framework, DFA-Net utilizes two types of Value Sets: the `IN` and `OUT` sets,

which correspond to the values entering and exiting each node after applying flow functions. At the beginning of the propagation process, Value Sets are initialized, providing the starting values for subsequent applications of flow functions. As information is propagated through the CFG, the contents of these Value Sets are dynamically updated, adapting based on previous values and the features encountered at each node. Importantly, Value Sets are designed as multidimensional tensors of arbitrary size, enabling the learning of complex features. Since multiple domains can coexist and each reserves different dimensions for their facts, a value set is allowed to be a ragged tensor.

Initialization function. The initialization function, denoted as $\mathbb{I} \rightarrow \mathbb{V}$, generates the starting value for each node. This is modeled using a set of functions, one for each combination of fact class and instruction type, represented as $A_{I,D}$ for instruction type I and fact class D. Each function takes the features from both the instruction and the fact, yielding a potentially multidimensional fact value. Specifically, the function is defined as $A_{I,D} : F_I \times F_D \rightarrow V_D$. For each node, we compute these values for every fact and concatenate them to obtain the complete value.

Transfer function. The transfer function, represented as $\mathbb{I} \times \mathbb{V} \rightarrow \mathbb{V}$, defines how a node processes an incoming value and produces a new output value. This function is more complex than the initialization function and is broken into several components. The value of each output fact is determined by its own features, the features of the node, and all incoming fact values along with their corresponding features. We define this with two functions:

1. $T_{I,D} : F_I \times F_D \times V_D \rightarrow V_D$ for situations where the input and output facts are identical regarding their values.
2. $T_{I,D_{in},D_{out}} : F_I \times F_{D_{in}} \times F_{D_{out}} \times V_{D_{in}} \rightarrow V_{D_{out}}$ for cases where the input fact differs from the output fact. This function is relevant for learning non-separable dataflow analyses, which require information about other facts.

To finalize the output for each fact, we introduce a reduction function: $R_{I,D} : V_D \times V_D \rightarrow V_D$. The initial value for this reduction comes from $T_{I,D}$, the V_D obtained when the input and output facts are the same; additional incoming values are then folded into this initial value using the reduction function.

Meet function. The meet function, represented as $\mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}$, combines the current node value with incoming values. This process serves as a reduction over all incoming values, with the initial value acting as the starting point for this reduction. Accordingly, we express this functionality through functions represented as $M_D : V_D \times V_D \rightarrow V_D$.

Readout function. To retrieve the final results of the data flow analysis, a readout function is employed. The approach for this involves selecting values from the appropriate domain. For instance, if a program-level readout is required, the function would extract results from the graph domain. Conversely, the relevant information would be pulled from the node domain for a per-node readout. In cases where the fact values are multidimensional, an additional learned function may be utilized to map these values to a scalar representation.

3.4 Curriculum Learning Strategy

DFA-Net employs a sophisticated curriculum learning strategy [29] that enables systematic learning progression from simple to complex program structures. This approach is fundamental to the architecture's ability to generalize data flow analyses comprehensively.

The implementation begins with training on simple program graphs with a low *complexity*, establishing a solid foundation in basic data flow patterns. As a measure of complexity, we use the number of data flow propagation steps, i.e. applications of the transfer and meet functions, that a standard data flow algorithm requires to reach a fixed point. The model must demonstrate perfect accuracy on the training set before proceeding to testing, ensuring thorough understanding at each complexity level. This stringent requirement means that testing occurs only when the model achieves flawless performance on the training data, resulting in selective test execution throughout the training process.

When the model successfully evaluates the test set but has not reached the stopping criterion, the complexity of training graphs automatically increments by one level.

This adaptive complexity mechanism continues challenging the model with progressively more complex programs to make training converge faster.

The progressive complexity increment ensures that the model builds strong foundational knowledge before tackling more intricate program structures, ultimately reaching more robust generalization capabilities than traditional GNN approaches. The curriculum learning approach thus forms a crucial component of our approach, but can also be part of other approaches.

4 Evaluation

In this section, we evaluate the performance of DFA-Net on established data flow analyses. We perform a comprehensive set of experiments designed to demonstrate how our approach addresses the challenges related to generalization from simple to complex programs and the limitations of GNNs. Our experimental evaluation focuses on the two following key aspects.

Generalization across program complexities. We investigate the ability of DFA-Net to generalize data flow analyses from simple training examples to more complex and diverse program structures. This assessment

Table 1. Data flow analyses and their types. Non Separable analyses (NSA) look across facts, making them harder to learn. Separable analyses (SA) look just at one fact.

Analysis	Meet Function	Transfer Function	Type
Data dependencies	$\text{in}[n] = \bigcup_{p \in \text{pred}(n)} \text{out}[p]$	$\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] \setminus \text{kill}[n])$ $\text{gen}[n] = e \mid e$ is a definition made in node n $\text{kill}[n] = e' \mid e'$is def(var) as in $\text{gen}[n], e' \neq e$	SA
Definite assignment	$\text{in}[n] = \bigcap_{p \in \text{pred}(n)} \text{out}[p]$	$\text{out}[n] = \text{def}[n] \cup \text{in}[n]$	SA
Dominators	$\text{in}[n] = \begin{cases} \{n\} & n \text{ is entry} \\ \bigcap_{p \in \text{pred}(n)} \text{out}[p] & \text{otherwise} \end{cases}$	$\text{out}[n] = \text{in}[n] \cup \{n\}$	SA
Liveness	$\text{out}[n] = \bigcup_{s \in \text{succ}(n)} \text{in}[s]$	$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$	SA
Strong liveness	$\text{out}[n] = \bigcap_{s \in \text{succ}(n)} \text{in}[s]$	$\text{in}[n] = \begin{cases} (n - \{y\}) \cup Opd(e) & n \text{ is } y = e, y \in n \\ n - \{y\} & n \text{ is input } (y) \\ n \cup \{y\} & n \text{ is use } (y) \\ n & \text{otherwise} \end{cases}$	NSA
Possibly undefined	$\text{out}[n] = \bigcap_{p \in \text{pred}(n)} \text{in}[p]$	$\text{in}[n] = (\text{out}[n] \cup \text{def}[n]) \setminus \text{use}[n]$	SA
Reachability	$\text{out}[n] = \bigcup_{p \in \text{pred}(n)} \text{in}[p]$	$\text{in}[n] = \text{out}[n] \cup \{n\}$	SA

is crucial to demonstrate the architecture's applicability to real-world compiler tasks.

Comparison with state-of-the-art models. We compare DFA-Net against existing GNN-based approaches on the AnghaBench dataset [11]. This comparison highlights accuracy and generalization capability, underscoring the advantages of incorporating compiler-specific knowledge into the architecture design.

Through the experiments, we aim to provide compelling evidence that our compiler-specific neural architecture offers significant improvements over state-of-the-art models.

4.1 Experimental Setup

Test system and benchmarks. Results discussed in this paper were obtained on an AMD Ryzen Threadripper 3960X 24-Core Processor 2.2 GHz, 64 GB of RAM, NVIDIA GeForce RTX 3090, running Ubuntu Linux v20.04. To rigorously evaluate DFA-Net's performance and generalization capabilities for data flow analysis, we employ AnghaBench [11]. AnghaBench comprises over one million compilable C programs with great diversity in terms of complexity and coding styles.

Dataset. DFA-Net utilizes data flow trace constructs during data flow analyses, which are sequences of dataflow operations that lead to a fixed-point. The pre-processed AnghaBench dataset contains traces for the analyses listed in Table 1. The dataset contains 426K entries, with complexity ranging from 3 to 448813. The 426K entries result from extracting unique traces from the one million benchmarks in AnghaBench (note that CFGs of similar *shapes* may generate the same exact trace).

Baseline. We compare our architecture with state-of-the-art GNN models [7, 8, 10], specifically GNN+CFG, GNN+CDFG, GNN+ProGraML. We use the same parameters as described in [10].

Metrics. In the evaluation, we employ three metrics – precision, recall, and F1-score – to assess model performance comprehensively rather than rely solely on accuracy. Precision measures the proportion of correctly identified positive predictions among all positive predictions, indicating how reliable the model's positive predictions are and helping us understand false positive rates. Recall, also known as sensitivity, quantifies the proportion of actual positive cases correctly identified, revealing how well the model captures all relevant instances and helping identify false negatives. The F1-score, the harmonic mean of precision and recall, provides a balanced metric particularly valuable in our compiler analysis context, where false positives and negatives can have significant implications for program optimization and correctness. This combination of metrics is especially crucial in data flow analyses, where simple accuracy metrics might mask poor performance and lead to incorrect compiler optimizations or missed optimization opportunities.

DFA-Net domains and features. We evaluate the model using three different domains. The domain of nodes, the domain of instructions, and the domain of variables. Each domain object is associated with features – the objects of

Table 2. Performance results of DFA-Net across various program analyses and complexities. The highlighted rows indicate the analyses where performance degradation occurred.

Analysis		Complexity 100	200	300	400	500	600	700	800	900	1000
Data dependencies	Precision	0.918	0.841	0.781	0.794	0.729	0.752	0.723	0.713	0.687	0.745
	Recall	0.851	0.824	0.804	0.814	0.769	0.838	0.799	0.776	0.750	0.779
	F1	0.883	0.832	0.793	0.804	0.748	0.793	0.759	0.743	0.717	0.761
Definite assignment	Precision	1.0	1.0	0.999	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	Recall	1.0	1.0	0.999	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	F1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Dominators	Precision	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999
	Recall	0.999	0.999	0.996	0.996	0.989	0.993	0.993	0.999	0.984	0.979
	F1	0.999	0.999	0.998	0.997	0.994	0.996	0.996	0.999	0.992	0.989
Liveness	Precision	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.999	1.0	1.0
	Recall	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	1.0
	F1	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	1.0
Strong liveness	Precision	0.976	0.968	0.969	0.968	0.926	0.972	0.952	0.943	0.945	0.952
	Recall	0.712	0.709	0.680	0.669	0.665	0.684	0.653	0.699	0.704	0.706
	F1	0.823	0.818	0.799	0.791	0.773	0.803	0.774	0.803	0.807	0.811
Possibly undefined	Precision	0.981	0.978	0.969	0.965	0.940	0.918	0.949	0.920	0.856	0.885
	Recall	0.998	0.998	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999
	F1	0.989	0.989	0.984	0.982	0.969	0.957	0.974	0.958	0.922	0.939
Reachability	Precision	1.0	1.0	1.0	1.0	1.0	0.999	1.0	1.0	1.0	0.999
	Recall	0.997	0.996	0.995	0.995	0.993	0.994	0.994	0.991	0.995	0.992
	F1	0.998	0.998	0.997	0.997	0.996	0.997	0.997	0.995	0.997	0.996

the node domain have node id as features, objects of the instruction domain have the instruction type as feature, and variable domain objects have use and defines of variables as features. The selected features allow learning the respective data flow analysis, e.g. a reachability analysis would learn how to leverage the node id feature, and a strong liveness analysis the instruction type, as well as the use and define features of variables.

DFA-Net parameters. The parameters of DFA-Net are as follows: The initialization function $A_{I,D}$ consists of 2 hidden layers of size 4, the transfer function $T_{I,D}$ of 4 hidden layers of size 4, the transfer function $T_{I,D_{in},D_{out}}$ 2 hidden layers of size 4, and the meet function M_D 2 hidden layers of size 4. The experiment parameters are as follows: the initial training complexity is 10, test complexities range from 100 to 1000 in increments of 100, the number of training samples is 50, and the number of test samples is 500. The stopping criterion is set to $step == 100,000$ or $f1 >= 0.99$, with a maximum batch size of 256. These parameter values are consistently used throughout the experiments. We further select the right analysis direction for each analysis.

4.2 DFA-Net Performance in Data Flow Analyses

As mentioned before, we evaluate DFA-Net on seven common analyses in compilers: data dependencies, definite assignment, dominators, liveness, strong liveness, potentially undefined variables, and reachability. The evaluation approach helps assess how effectively DFA-Net adapts learned patterns from simple cases to complex program structures. Table 2 summarizes the results of the dataflow analyses.

DFA-Net achieves near-perfect performance in four analyses: definite assignment, dominators, liveness, and reachability. For these analyses, the model consistently maintains scores of 0.999 across all evaluation metrics, regardless of the graph complexity.

The DFA-Net results show interesting patterns at various complexity levels and types of analysis. In the rows highlighted in grey, the model's performance varies, particularly depending on the analysis type and the complexity of the input graphs. For the data dependencies analysis, the model demonstrates its strongest performance with simpler graphs, accurately identifying correct relationships and avoiding false predictions. However, as the complexity of the graphs increases, there is a decline in performance, with the most

complex graphs showing nearly a fifth reduction in overall effectiveness.

The strong liveness analysis presents a more stable pattern across different complexity levels. While it still achieves its best results with simpler graphs, the decline in performance is more modest compared to the data dependencies analysis. The model's ability to maintain high accuracy in its predictions across all complexity levels is particularly noteworthy, given that this is a non-separable analysis, which is significantly more complex.

Finally, the model shows its most robust performance among the highlighted analyses in the case of possibly undefined analysis. The results are particularly good on simple graphs, with minimal decline as complexity increases. One interesting aspect is the model's consistent ability to identify almost all relevant cases across all complexity levels.

The observed performance degradation in higher complexities can be attributed to the substantial difference between training and testing data complexities (note that DFA-Net still generalizes orders of magnitude better on complex programs than the state of the art). The model was trained on relatively simple graphs but was tested on significantly more complex ones, creating a challenging generalization scenario. This performance degradation with increasing problem size arises from the numerical scalability limits of the learned data flow functions.

A key element of DFA-Net's training is its selective test execution strategy, leading to minimal test evaluations. For instance, out of the total training steps, the possibly undefined analysis generates test results in only two different moments. In comparison, the dominator analysis produces test results in thirty-nine different steps, and all other analyses generate test results in just one.

To illustrate how the progressive complexity mechanism improves DFA-Net's performance, we examine the training process for the dominators analysis. Figure 3 is an example of how incorrect facts decrease as the training progresses and complexity gradually increases. For dominators analysis, the training complexity ranged from 10 to 48 and was generalized to test complexities of 100-1000, as shown in the figure. Through this result, we can observe the effectiveness of the adaptive complexity approach in helping the model achieve and maintain high accuracy.

4.3 DFA-Net vs GNNs for Code

We compare DFA-Net with state-of-the-art GNNs, highlighting key differences in their learning and generalization capabilities.

Table 3 compares the DFA-Net and GNN models on the analyses implemented in the baseline [8, 10]. As can be seen from the table, DFA-Net greatly outperforms existing GNNs across the board. While working with a small dataset of only 50 training samples, DFA-Net showcases great performance, highlighting its ability to learn effectively from minimal data. This achievement is notable given the significant complexity difference between training and testing environments.

Traditional GNNs face two key problems, which reflect in the experimental results. First, their reachability is limited, as GNNs propagate can only a finite/pre-defined number of steps, whereas DFA-Net's learned flow functions can be applied a infinite number of times. Second, the bottleneck problem, where GNN model performance degrades with increasing propagation steps due to the exponential increase in arriving information. DFA-Net on the other hand limits the information arriving at a node to the relevant control-flow edges. In combination, these challenges make it difficult for GNNs to learn generalizable solutions, making their performance substantially lower when limited to small training samples. The situation becomes even more challenging for these networks when they encounter test cases with complexity levels far exceeding their training examples. In addition, the disparity between training and testing complexity poses a significant challenge for non-compiler-informed GNNs. While training occurs on graphs with complexity level 10, the test set contains graphs ranging from 100 to 1000. This substantial jump in complexity reveals the critical weakness in traditional GNNs, as they struggle to generalize their learned patterns to more complex scenarios. Instead of learning generalizable solutions, GNNs exhibit signs of overfitting to the specific characteristics of the training data, making them less effective when encountering new, more complex programs. This limitation is particularly problematic in real-world scenarios where training data may be scarce. Our results highlight the power of having a compiler-informed architecture that effectively leverages domain knowledge to learn even with minimal training examples. This is the way DFA-Net displays such robust generalization capabilities.

Comparing the different models, GNN+ProGraML has a substantially larger number of parameters, with 863,774, more than 26 times those of GNN+CFG (17,118) and GNN+CDFG (15,928) combined. Most notably, DFA-Net achieves superior performance while utilizing only 397 parameters, approximately 43 times less than GNN+CFG and 40 times less than GNN+CDFG. DFA-Net demonstrates that better results can be achieved with a more compact architecture, which is 3 orders of magnitude smaller than GNN+ProGraML while delivering enhanced performance. While GNN+ProGraML outperforms DFA-Net in computational efficiency, with a significantly shorter training time of 461.99 seconds (6 times faster than DFA-Net's 2852.51 seconds) and a faster inference time of 10.05 seconds (nearly 4.5 times quicker than DFA-Net's 45.68 seconds), the primary goal is to achieve better results. DFA-Net's ability to deliver superior performance with a smaller and simpler architecture ultimately outweighs the computational efficiency advantage of GNN+ProGraML, emphasizing the importance of model effectiveness over speed.

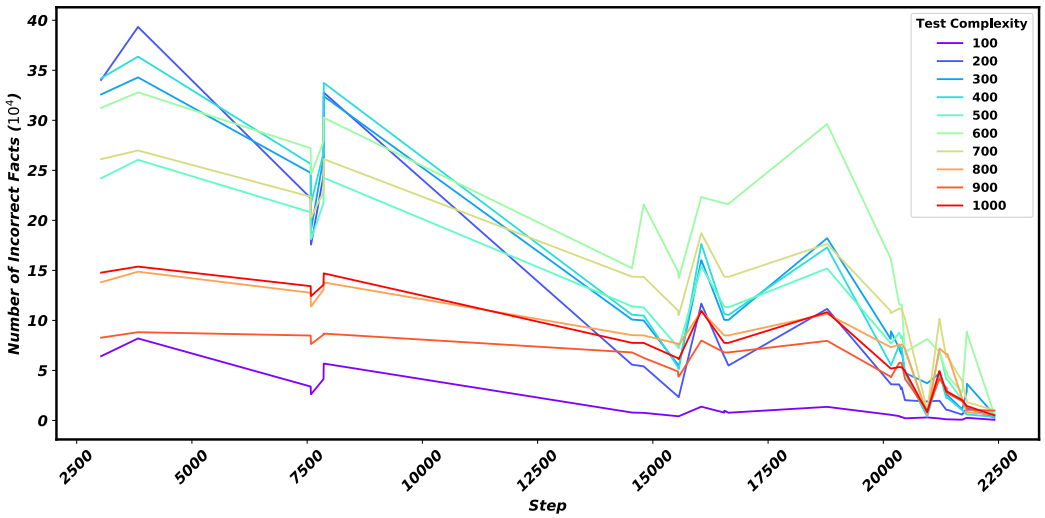


Figure 3. Performance of adaptive complexity approach in dominator analysis, with each color representing a test complexity.

Table 3. DFA-Net and GNN performance across different levels of complexity. Highlighted columns indicate the best models.

		Complexity = 100				Complexity = 1000			
Analysis		**CFG**	**CDFG**	**ProGraML**	**DFA-Net**	**CFG**	**CDFG**	**ProGraML**	**DFA-Net**
Data dependencies	Precision	0.103	0.058	0.114	0.918	0.03	0.015	0.005	0.745
	Recall	0.364	0.202	0.049	0.851	0.391	1.0	0.857	0.779
	F1	0.161	0.089	0.069	0.883	0.056	0.029	0.009	0.761
Dominators	Precision	0.293	1.0	0.237	0.999	0.231	0.164	0.109	0.999
	Recall	0.977	0.034	0.204	0.999	0.998	0.976	0.986	0.979
	F1	0.451	0.066	0.217	0.999	0.375	0.281	0.196	0.989
Liveness	Precision	–	–	0.151	0.999	–	–	0.061	1.0
	Recall	–	–	0.999	0.999	–	–	1.0	1.0
	F1	–	–	0.263	0.999	–	–	0.115	1.0
Reachability	Precision	0.439	0.421	0.458	1.0	0.685	0.348	0.426	0.999
	Recall	1.0	0.988	0.915	0.998	0.874	0.982	0.992	0.992
	F1	0.610	0.591	0.609	0.999	0.768	0.514	0.596	0.996

5 Conclusions

In this paper, we introduced DFA-Net, a novel neural network architecture inspired by traditional data flow analysis frameworks used in compilers. By employing a model tailored for program analysis and enhancing the traditional data flow analysis framework, DFA-Net provides strong generalization.

Our experiments demonstrated that DFA-Net consistently outperforms state-of-the-art GNNs in data flow analysis tasks. GNNs often struggle to generalize beyond the specific patterns present in their training data and require extensive datasets to learn meaningful representations. In contrast, DFA-Net showed remarkable generalization capabilities, effectively learning from minimal training data and maintaining high performance even as the complexity of the program graphs increased. In conclusion, DFA-Net represents a significant step forward in applying neural networks to compiler data flow analysis.

Future exploration includes using DFA-Net to predict code properties and analyze learned data flow functions.

Acknowledgments

This work was partially funded by the Center for Advancing Electronics Dresden (cfaed), and the AI competence center ScaDS.AI Dresden/Leipzig in Germany (01IS18026A-D).

Data-Availability Statement

The source code, benchmarks, and evaluation scripting that support our findings are openly available as an artifact [6].

References

[1] Yoav Alon and Cristina David. 2022. Using graph neural networks for program termination. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 910–921. https://doi.org/10.1145/3540250.3549095

[2] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. 2020. Actions Speak Louder than Words: Entity-Sensitive Privacy Policy and Data Flow Analysis with PoliCheck. In *USENIX Security Symposium*. USENIX, USA, 1–12. https://api.semanticscholar.org/CorpusID:212615510

[3] Hani Bani-Salameh, Mohammed Sallam, and Bashar Al shboul. 2021. A Deep-Learning-Based Bug Priority Prediction Using RNN-LSTM Neural Networks. *e-Informatica Software Engineering Journal* 15, 1 (Feb. 2021), 29–45. https://doi.org/10.37190/e-Inf210102 Available online: 08 February 2021.

[4] Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin Vechev. 2024. DeepCode AI Fix: Fixing Security Vulnerabilities with Large Language Models. *CoRR* abs/2402.13291 (2024), 1–12. arXiv:2402.13291 https://arxiv.org/abs/2402.13291

[5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *CoRR* abs/2403.17134 (2024), 1–12. arXiv:2403.17134 [cs.SE] https://arxiv.org/abs/2403.17134

[6] Alexander Brauckmann, Anderson Faustino da Silva, Gabriel Synnaeve, Michael F.P. O'Boyle, Jeronimo Castrillon, and Hugh Leather. 2025. DFA-Net: A Compiler-Specific Neural Architecture for Robust Generalization in Data Flow Analyses (Artifact). https://doi.org/10.5281/zenodo.14670956

[7] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. 2020. ComPy-Learn: A toolbox for exploring machine learning representations for compilers. In *2020 Forum for Specification and Design Languages (FDL)*. IEEE, Verona, Italy, 1–4. https://doi.org/10.1109/FDL50818.2020.9232946

[8] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *Proceedings of the 29th ACM SIGPLAN International Conference on Compiler Construction (CC 2020)* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[9] Zhaoyang Chu, Yao Wan, Qian Li, Yang Wu, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2024. Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 389–401. https://doi.org/10.1145/3650212.3652136

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the International Conference on Machine Learning*, Marina Meila and Tong Zhang (Eds.), Vol. 139. PMLR, Virtual Only, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

[11] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. 2021. ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. ACM, Virtual Only, 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

[12] Yuhong Feng, Haoran Li, Yixuan Cao, Yufeng Wang, and Haiyue Feng. 2024. CRABS-former: CRoss-Architecture Binary Code Similarity Detection based on Transformer. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware* (Macau, China) *(Internetware '24)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3671016.3671390

[13] Vikas K. Garg, Stefanie Jegelka, and Tommi S. Jaakkola. 2020. Generalization and Representational Limits of Graph Neural Networks. *CoRR* abs/2002.06157 (2020), 1–12. arXiv:2002.06157 https://arxiv.org/abs/2002.06157

[14] U. Khedker, A. Sanyal, and B. Sathe. 2017. *Data Flow Analysis: Theory and Practice*. CRC Press, USA.

[15] Hanxi Li, Ruohan Wu, Liang Qian, and Hong An. 2023. A Systematic Methodology for performance characterizing of Heterogeneous Systems with a dataflow runtime simulator. In *Proceedings of the 2022 4th International Conference on Robotics, Intelligent Control and Artificial Intelligence* (Dongguan, China) *(RICAI '22)*. Association for Computing Machinery, New York, NY, USA, 629–637. https://doi.org/10.1145/3584376.3584487

[16] Chen Liang, Qiang Wei, Zirui Jiang, Yisen Wang, and Jiang Du. 2024. A Source Code Vulnerability Detection Method Based on Adaptive Graph Neural Networks. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops* (Sacramento, CA, USA) *(ASEW '24)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/3691621.3694950

[17] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning Graph-Based Code Representations for Source-Level Functional Similarity Detection. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 345–357. https://doi.org/10.1109/ICSE48619.2023.00040

[18] Pengcheng Liu, Yifei Lu, Wenhua Yang, and Minxue Pan. 2024. Valar: Streamlining Alarm Ranking in Static Analysis with Value-Flow Assisted Active Learning. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE Press, Echternach, Luxembourg, 1940–1951. https://doi.org/10.1109/ASE56229.2023.00098

[19] Mingming Lu, Dingwu Tan, Naixue Xiong, Zailiang Chen, and Haifeng Li. 2019. Program Classification Using Gated Graph Attention Neural Network for Online Programming Service. *CoRR* abs/1903.03804 (2019), 1–12. arXiv:1903.03804 http://arxiv.org/abs/1903.03804

[20] Arpit Merchant and Carlos Castillo. 2023. Disparity, Inequality, and Accuracy Tradeoffs in Graph Neural Networks for Node Classification. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management* (Birmingham, United Kingdom) *(CIKM '23)*. Association for Computing Machinery, New York, NY, USA, 1818–1827. https://doi.org/10.1145/3583780.3614847

[21] Steven S. Muchnick. 1998. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[22] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 582–586. https://doi.org/10.1145/3663529.3663803

[23] Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. 2024. Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation. *Proc. ACM Softw. Eng.* 1, FSE, Article 86 (July 2024), 23 pages. https://doi.org/10.1145/3660793

[24] Ahmed E. Samy, Zekarias T. Kefato, and Sarunas Girdzijauskas. 2023. Graph2Feat: Inductive Link Prediction via Knowledge Distillation. In *Companion Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 805–812. https://doi.org/10.1145/3543873.3587596

[25] Aria Seo, Young-Tak Kim, Ji Seok Yang, YangSun Lee, and Yunsik Son. 2024. Software Weakness Detection in Solidity Smart Contracts Using Control and Data Flow Analysis: A Novel Approach with Graph Neural Networks. *Electronics* 13, 16 (2024), 1–12. https://doi.org/10.3390/electronics13163162

[26] Florian Sihler. 2024. Improving the Comprehension of R Programs by Hybrid Dataflow Analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2490–2493. https://doi.org/10.1145/3691620.3695603

[27] Zhihui Song and Jinchen Xu. 2024. BinVuGAL: Binary vulnerability detection method based on graph neural network combined with assembly language model. In *Proceedings of the 2024 3rd International Conference on Cryptography, Network Security and Communication Technology* (Harbin, China) *(CNSCT '24)*. Association for Computing Machinery, New York, NY, USA, 159–163. https://doi.org/10.1145/3673277.3673305

[28] Zirui Song, YuTong Zhou, Shuaike Dong, Ke Zhang, and Kehuan Zhang. 2024. TypeFSL: Type Prediction from Binaries via Interprocedural Data-flow Analysis and Few-shot Learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1269–1281. https://doi.org/10.1145/3691620.3695502

[29] Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. 2022. Curriculum learning: A survey. *International Journal of Computer Vision* 130, 6 (2022), 1526–1565.

[30] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/3597503.3623345

[31] Zewen Sun, Duanchen Xu, Yiyu Zhang, Yun Qi, Yueyang Wang, Zhiqiang Zuo, Zhaokang Wang, Yue Li, Xuandong Li, Qingda Lu, Wenwen Peng, and Shengjian Guo. 2023. BigDataflow: A Distributed Interprocedural Dataflow Analysis Framework. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1431–1443. https://doi.org/10.1145/3611643.3616348

[32] Kenta Terada and Yutaka Watanobe. 2019. Code Completion for Programming Education based on Recurrent Neural Network. In *2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA)*. IEEE, Hiroshima, Japan., 109–114. https://doi.org/10.1109/IWCIA47330.2019.8955090

[33] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating Debugging Capability of Large Language Models. *CoRR* abs/2401.04621 (2024), 1–12. arXiv:2401.04621 https://arxiv.org/abs/2003.13848

[34] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2024. StructCoder: Structure-Aware Transformer for Code Generation. *ACM Trans. Knowl. Discov. Data* 18, 3, Article 70 (Jan. 2024), 20 pages. https://doi.org/10.1145/3636430

[35] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. When Dataflow Analysis Meets Large Language Models. *CoRR* abs/2402.10754 (2024), 1–12. arXiv:2402.10754 [cs.SE] https://arxiv.org/abs/2402.10754

[36] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree . In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 261–271. https://doi.org/10.1109/SANER48275.2020.9054857

[37] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M. Zhang, and Qing Liao. 2023. Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 2275–2286. https://doi.org/10.1109/ICSE48619.2023.00191

[38] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1345–1357. https://doi.org/10.1145/3691620.3695508

[39] Shih-Yuan Yu, Yonatan Gizachew Achamyeleh, Chonghan Wang, Anton Kocheturov, Patrick Eisen, and Mohammad Abdullah Al Faruque. 2023. CFG2VEC: Hierarchical Graph Neural Network for Cross-Architectural Software Reverse Engineering. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, Melbourne, Australia, 281–291. https://doi.org/10.1109/ICSE-SEIP58684.2023.00031

[40] Yuguo Zhang, Jia Yang, and Ou Ruan. 2024. Cross-language Source Code Clone Detection Based On Graph Neural Network. In *Proceedings of the 2024 3rd International Conference on Cryptography, Network Security and Communication Technology* (Harbin, China) *(CNSCT '24)*. Association for Computing Machinery, New York, NY, USA, 189–194. https://doi.org/10.1145/3673277.3673310

[41] Tianyi Zhao, Jian Kang, and Lu Cheng. 2024. Conformalized Link Prediction on Graph Neural Networks. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Barcelona, Spain) *(KDD '24)*. Association for Computing Machinery, New York, NY, USA, 4490–4499. https://doi.org/10.1145/3637528.3672061

[42] Tianxiang Zhao, Xiang Zhang, and Suhang Wang. 2024. Disambiguated Node Classification with Graph Neural Networks. In *Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 914–923. https://doi.org/10.1145/3589334.3645637

[43] 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. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4283–4300. https://www.usenix.org/conference/usenixsecurity24/presentation/zhu-chang

Received 2024-11-11; accepted 2024-12-21