Proceedings of the International Conference on Formal Methods
and Foundations of Artificial Intelligence
Eszterházy Károly Catholic University, Eger, Hungary, June 5–7, 2025

Using GNN for Refactoring P4 Programs

Benedek Szabolcs Csüllög[a], **Máté Tejfel**[b],

[a]Eötvös Loránd University Budapest, ELTE
benedek.csullog@gmail.com

[b]Eötvös Loránd University Budapest, ELTE
matej@caesar.elte.hu

Abstract. P4 [2] is a domain-specific language for programming the data plane of network devices in a protocol-independent manner. To analyze and transform Programming Protocol-Independent Packet Processors (P4) programs, we use P4Query [5], a tool that performs both syntactic and semantic analyses and represents P4 source code as abstract syntax trees (ASTs) in the form of directed graphs. In this paper, we explore how Graph Neural Networks (GNNs) can be applied to these graph-structured ASTs to learn high-level code transformations. We introduce and evaluate three models: a variable renamer that learns to propagate identifier changes across the AST, a parameter reorderer that predicts function argument permutations, and a detector for semantically empty else branches. These tasks demonstrate the effectiveness of GNNs in understanding and transforming P4 code structures. Such models can support code optimization and standardization efforts by automating repetitive or error-prone transformations in P4 programs.

1. Introduction

The rapid evolution of computer networks has led to the increasing complexity of network protocols and the need for flexible, programmable solutions. The Programming Protocol-Independent Packet Processors (P4) [2] language has emerged as a powerful tool for defining how packets are processed in network devices. it follows the Software Defined Networking (SDN) approach, so it allows developers to specify the behavior of the data plane independently of the control plane. Unlike traditional programming languages, P4 mainly focuses on the data plane and makes it fully programmable, enabling more dynamic and adaptable network configurations.

As the use of P4 grows, so does the necessity for effective analysis and refactoring

tools that can assist developers in optimizing their code. The P4Query [5] tool provides a robust framework for static analysis of P4 programs, generating abstract syntax trees (ASTs) [3] that represent the structure of the code. These ASTs serve as a foundation for various analyses and transformations, facilitating the identification of potential improvements and refactoring opportunities.

In recent years, Graph Neural Networks (GNNs) [9] have gained prominence in the field of machine learning, particularly for tasks involving graph-structured data. By leveraging the inherent relationships within graphs, GNNs can learn to predict node attributes, modify graph structures, and uncover hidden patterns. This paper explores the integration of GNNs with P4Query-generated ASTs to enhance the refactoring process of P4 programs. By training GNN models on these syntax trees, we aim to develop a system that can intelligently suggest modifications, thereby improving code quality and maintainability.

GNNs operate on graph-structured data, where the input consists of a set of nodes and edges that define the relationships between those nodes. Each node is typically associated with a feature vector that encodes relevant information about that node. During the training process, GNNs utilize a message-passing mechanism, where nodes exchange information with their neighbors iteratively. In each iteration, a node aggregates the features of its neighboring nodes, allowing it to update its own feature vector based on the collective information from its local neighborhood.

This process enables GNNs to capture both local and global structural patterns within the graph. After several iterations of message passing, the updated node features can be used for various tasks, such as node classification, link prediction, or graph classification. Ultimately, GNNs modify the input feature representations by learning to emphasize important relationships and patterns within the graph, leading to improved performance on tasks that require an understanding of complex relational data.

The P4 programming language is widely used for describing packet-processing logic in programmable network devices. Due to its domain-specific nature and structural richness [4], refactoring P4 programs presents unique challenges that cannot be effectively addressed with traditional string-based or token-based methods.

Our goal is to design a refactoring pipeline that learns from examples, generalizes across P4 code bases, and ultimately provides maintainability and performance improvements. In this paper, we focus on the graph representation learning task and training process.

In summary, this research seeks to bridge the gap between advanced machine learning techniques and the practical needs of network programming, contributing to the development of more efficient and reliable network applications.

The practical value of these models lies in their ability to enable standardization across large P4 codebases, ensure consistency in naming conventions, and reduce manual effort in code cleanup. These improvements directly support better performance, maintainability, and reduce potential bugs in programmable network

configurations.

2. Related works

The P4Query static analysis framework is centered around an extensible internal graph representation where the results of the different static analysis methods are also stored as part of the graph. The information in the knowledge graph is accessed using graph queries written in the Gremlin [7] query language. In this way, the framework guarantees a unique standard representation both for the stored data and for the data access mechanism. P4 programs can naturally be represented as abstract syntax trees (ASTs), which are directed graphs generated by the framework. The framework can also be used to implement P4-specific refactoring steps [8].

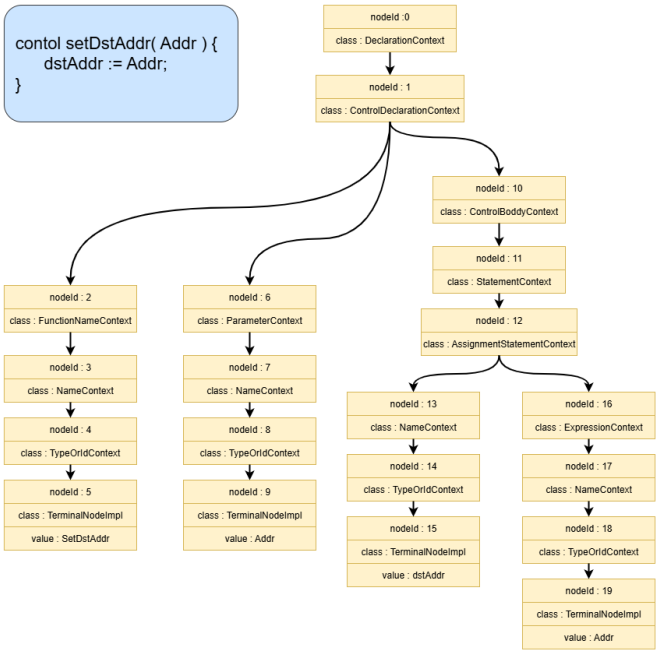

Figure 1. P4 function implementation and its AST

Figure 1 presents a simple P4 control declaration alongside its corresponding abstract syntax tree (AST). On the left, the P4 code defines a control block that assigns the `Addr` parameter to the `dstAddr` field. On the right, the simplified AST captures the structure of the program as a directed graph, including all relevant nodes, edges, and key node attributes such as `nodeId`, `class`, and `value`. The `class` attribute specifies the type of the node; if the class is `TerminalNodeImpl`, then the

`value` attribute contains a string literal that appears explicitly in the original P4 source code.

Graph Neural Networks (GNN) are designed to work with graph data structures. GNNs learn graph-level representations by iteratively aggregating information from neighboring nodes. The machine learning models try to modify the node attributes or the structure of the graph. The outputs are the modified graph data structures.

Code2Vec [1] is a framework that learns continuous feature representations for nodes in a graph. In the context of abstract syntax trees, the authors transformed the nodes of the syntax tree into vector representations. This allows machine learning models to capture the structural and semantic properties of the programs represented by the trees. By leveraging these vector representations, the model can effectively recognize and classify specific programs based on their syntax trees.

Code2Vec accomplished this by training models capable of recognizing well-known algorithms—such as sorting, searching, or counting—across multiple programming languages, including Java, C++ and Python. This approach involves generating a syntax tree from the source code, which is then analyzed by the trained model to identify the underlying algorithm. After the analysis, the model also provides a probability score for its prediction.

One notable example of applying Graph Neural Networks to program code analysis is Devign [10], in which the authors construct a code property graph (CPG) by merging Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Data Flow Graphs (DFG) into a single rich representation. The model—built upon a Gated Graph Neural Network—learns comprehensive program semantics from these graphs to identify security vulnerabilities in real-world C code. In empirical evaluations on large open-source projects, Devign significantly outperformed previous state-of-the-art vulnerability detection methods, improving accuracy by over 10% and F1 score by approximately 8–9%. This work exemplifies how merging multiple program-structure views and leveraging GNNs enables robust, semantics-aware analyses, and inspires similar approaches in tasks like variable renaming or parameter reordering in P4 ASTs.

A closely related work is the paper titled *P4 Specific Refactoring Steps* [8], which introduces a set of rule-based refactoring operations specifically tailored for P4 programs. The approach is built on the P4Query framework, utilizing its syntactic and semantic graph-based representations to perform transformations on abstract syntax trees (ASTs). These refactorings include table splitting, merging, and execution reordering, all governed by static analysis and predefined preconditions to ensure semantic correctness. While this method is based on manually designed transformations, our work extends this line of research by applying machine learning—specifically Graph Neural Networks—to automatically learn transformation patterns from data. This enables a shift from rigid, rule-based systems toward adaptive, data-driven refactoring strategies that generalize across diverse P4 programs.

3. Methodology

3.1. Graph Construction from P4 Programs

To obtain structured representations of P4 programs, we rely on P4Query, a comprehensive analysis tool that extracts detailed abstract syntax trees (ASTs) and other intermediate structures from P4 source code. The tool uses the official P4 language protocol to generate a standardized and richly annotated AST, which is interpreted as a large directed graph. The AST's structure is defined by the syntactic rules of the language, and its edges encode diverse semantic relations; for instance, control flow and data flow dependencies.

Each node in the graph has several attributes that capture both syntactic and positional information. These include a unique `nodeId`, a `line` attribute indicating the location in the source code, `begin` and `end` fields that mark the span of the syntactic construct, a `type` field (e.g., `TerminalNodeImpl`), and, in the case of terminal nodes, a `value` representing the literal content. When preparing this graph for training a Graph Neural Network (GNN), we encode only the `type` and `value` attributes (if present) to create initial node feature vectors. This selective encoding strikes a balance between expressiveness and efficiency, allowing the model to focus on meaningful structural relationships while leveraging the inductive bias of the syntax-driven graph topology.

3.2. Training of the GNN

To train our Graph Neural Network (GNN) models, we first generated several different P4 abstract syntax trees (ASTs) using P4Query. Each AST is treated as a directed graph, representing the dataset. As described earlier, each node's attributes have to be auto-encoded using its `type` and (if available) `value` attributes, producing a lightweight yet expressive node feature representation suitable for neural processing.

During the training procedure each epoch, the GNN is presented with a modified version of a graph and tasked with reconstructing the original structure. During the first epoch, the full graph is shown to the model, establishing a baseline understanding of the node representations and their connectivity. From the second epoch onward, we progressively apply structured degradation to the input graphs. Initially, we remove only terminal nodes (i.e., nodes of type `TerminalNodeImpl`), which are typically leaves of the AST. In later stages, the removal process targets increasingly complex subgraphs, thereby challenging the model to infer more abstract syntactic patterns.

Importantly, the complexity of the removed subgraphs increases exponentially as training progresses. That is, in each subsequent training phase, the size and structural depth of the deleted subgraphs grow according to an exponential schedule. For example, while early stages may remove only isolated leaf nodes or shallow branches, later phases may eliminate entire nested control structures, such as `if-else` blocks, loops, or parameter lists. This exponential degradation ensures a

curriculum-style learning process in which the model first learns to reconstruct simple local patterns, and only later faces the challenge of recovering deeply structured and semantically rich fragments of the AST.

The goal of this curriculum-style training is to gradually force the model to internalize deeper compositional structures in the input graphs. By learning to predict missing parts of the AST across a variety of P4 programs, the GNN develops a generalizable representation of the language's syntactic structure. Throughout training, the model is evaluated based on its reconstruction accuracy, which reflects how well it can recover node identities and their connections.

Using this training strategy, we developed three distinct GNN models, each targeting a specific code transformation or analysis task: a variable renamer, a parameter reorderer, and an empty `else` block detector. Although these models share the same underlying graph-based training framework, each addresses a different aspect of code semantics.

These tasks were carefully selected to reflect realistic challenges in network code development. By automating variable renaming, argument ordering, and dead-code detection, the models target areas where inconsistency and redundancy frequently arise in practice. Their utility lies not only in code transformation, but also in establishing and enforcing coding conventions across diverse teams and projects.

3.3. Variable Renamer

The Variable Renamer model is trained to recognize and rename variables across abstract syntax trees. It is implemented as a two-layer Graph Neural Network (GNN), which performs node classification over the AST graph. Each node is represented by a simple feature vector encoding whether it is a terminal node and whether it contains a value. The model learns to predict the original `value` attribute of `TerminalNodeImpl` nodes based on their surrounding context.

During training, graphs are labeled by the original variable names of terminal nodes, which serve as supervision targets. Once trained, the model takes as input a complete AST graph and a variable name, and it modifies the `value` attribute of all relevant nodes to a new name. This allows for consistent refactoring, such as renaming `egress_spec` to `egress_specific` across all its occurrences in the graph. After 20 epochs of training, the model successfully learned the graph structure and renaming patterns by leveraging structural differences between input graphs.

Renaming variables in header block

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ethernet_t {
    macAddr_t destinationAddr;
    macAddr_t sourceAddr;
    bit<16>   ethernetType;
}
```

The renaming decision is not based solely on the textual value of a node, but also on its structural context. This enables the model to generalize and identify semantically equivalent nodes in different programs, even when local syntax varies. The GNN's message-passing mechanism allows it to aggregate information from neighboring nodes, making the renaming behavior robust and context-aware.

3.4. Parameter Reorderer

The Parameter Reorderer model is designed to learn the correct ordering of function parameters in P4 programs. Unlike the Variable Renamer, this model is not a traditional GNN architecture. Instead, it uses a learned embedding layer in combination with a small feed-forward neural network to predict the target position of each parameter.

The training is performed using pairs of AST graphs: one representing the original parameter order, and one representing the desired (reordered) form. Each parameter is represented by its subtree in the AST, and the model learns to assign a position score to each based on its embedded identity. The loss function minimizes the mean squared error between the predicted positions and the ground truth permutation derived from the reordered graph. The model required 100 epochs of training to learn the parameter reordering logic from structural graph differences.

Once trained, the model receives a new AST and infers a new permutation of the parameters. It then rewrites the structure of the graph accordingly, updating the 'start' and 'end' positions of the affected nodes and regenerating comma separators where needed. This results in a syntactically valid AST that reflects the learned parameter order.

Reordering parameter list in control declaration

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}

control MyDeparser(in headers hdr, packet_out packet) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

The model's ability to operate directly on AST node embeddings without requiring handcrafted rules—enables flexible reordering strategies and supports the standardization of function signatures across large codebases.

3.5. Empty Else Block Detector

The Empty Else Block Detector is a binary classification model built to detect empty `else` branches in P4 program ASTs. It is implemented as a two-layer GNN, trained to classify specific nodes labeled as `else` into two categories: empty or non-empty.

The model is trained on a dataset of P4 ASTs, where each graph is automatically labeled using a rule-based search for syntactically empty `else` branches. During preprocessing, only nodes with the literal value `"else"` are included in the training set, and are labeled according to whether they lead to an empty block in the AST. Each node is encoded using a pair of categorical features: the class of the node (e.g., `TerminalNodeImpl`, `StatementContext`) and the textual value (if present). The model was trained for 400 epochs to capture the subtle structural patterns associated with empty else branches.

The GNN learns to classify these `else` nodes based on their surrounding context in the AST. Its performance is measured via accuracy on the classification task. Once trained, the model can analyze unseen ASTs and assign a confidence score to each `else` branch, indicating whether it is likely to be empty.

This model serves as a useful tool for detecting redundant or misleading conditional structures in P4 programs. By identifying empty branches that serve no semantic purpose, the model supports further code cleanup and optimization.

Empty else block in P4

```
if (hdr.ipv415.isValid()) {
    if (hdr.ipv416.isValid()) {
        ipv415.lpm.apply();
        ipv416.lpm.apply();
    } else {}
} else {}
```

All models presented in this work were implemented using the `PyTorch` [6] and `PyTorch Geometric` libraries, which provide efficient GPU-accelerated operations and high-level abstractions for graph-based deep learning. These libraries enabled rapid experimentation with neural architectures and streamlined the construction and training of Graph Neural Networks. We used standard components such as `GCNConv` layers, embedding modules, and training utilities provided by the framework.

4. Results

In our experiments, the abstract syntax trees (ASTs) generated by the P4QueryčiteP4Q tool were serialized as large JSON files. Each file contains a directed graph with thousands of nodes and edges, representing detailed syntactic and semantic information of a given P4 program. Due to the high complexity and richness of these graphs, even a small number of training examples proved sufficient for effective learning.

We observed that all three models trained on these AST graphs exhibited consistently decreasing loss values throughout training, indicating convergence and successful pattern extraction. The loss function approached values close to zero across multiple epochs, demonstrating that the models were able to learn the structural relationships within the graphs efficiently.

4.1. Variable Renamer Results

The Variable Renamer model was trained on a set of 13 AST graphs, each representing a different P4 program. These graphs served as supervised input, where variable renaming labels were provided for terminal nodes. Training was performed over 20 epochs, which proved sufficient for the model to converge — the loss function consistently decreased and reached zero by the end of training. This indicates that the GNN successfully learned to encode the structure of a given variable's representation, identifying the relevant nodes and edges responsible for that variable throughout the graph.

Once training, the model was evaluated on 17 previously unseen AST graphs. In all cases, the model was instructed to rename a variables per graph. The model

successfully performed all 17 renamings, modifying the corresponding `value` attributes in the appropriate `TerminalNodeImpl` nodes. Importantly, the renamings were consistent across all occurrences of the target variables within each graph.

These results highlight the model's ability to generalize renaming patterns across different P4 codebases, even when local syntactic variations are present. This confirms the effectiveness of GNN-based learning on AST structures.

4.2. Parameter Reorderer Results

To evaluate the effectiveness of the Parameter Reorderer model, we trained it on 5 pairs (a total of 10) of P4 AST graphs. Each pair consisted of an original function declaration with an incorrect or arbitrary parameter order, and a corresponding target graph in which the parameters were reordered according to a preferred or canonical sequence. Unlike the Variable Renamer, this model had to capture and interpret more complex structural patterns, as parameters are represented by entire subgraphs within the AST. As a result, learning the correct transformations required more epochs (100 in total), and convergence was slower.

In cases where the function had only two parameters, the model achieved correct reordering in 3 of the 5 example graphs, which were previously unseen ASTs. These are effectively binary swaps, which the model was often able to predict reliably. However, as the number of parameters increased, the model's accuracy declined. This was particularly evident in more complex declarations, where parameters had compound types or nested substructures. In such cases, the model struggled to infer the correct order, indicating a sensitivity to syntactic complexity.

These results suggest that while the model captures some general patterns of parameter ordering, it may lack sufficient contextual awareness to handle more elaborate structures. The approach, based on embedding and regression, shows promise for approximating permutation tasks but could benefit from stronger relational reasoning.

From these observations, we conclude that parameter reordering is more effectively approached as a regression problem over continuous embeddings than as a classification problem over discrete graph structures. Future improvements may involve incorporating structural attention or hybrid GNN-MLP architectures to better capture the interplay between parameter semantics and graph topology.

4.3. Empty Else Block Detector Results

The Empty Else Block Detector model was trained on a dataset of 24 AST graphs, each annotated with both empty and non-empty `else` branches. During training, the model performed binary classification on nodes with the literal value `"else"`, predicting whether they led to an empty code block. Due to the complexity of the task — which required the model to learn to recognize non-trivial subgraph patterns around each `else` node — the training process was conducted over 100 epochs. This extended training was necessary for the model to reliably capture the subtle structural cues associated with empty branches. The final training accuracy

reached 78%, demonstrating the model's ability to distinguish between semantically relevant and irrelevant `else` constructs.

To evaluate the model's generalization capability, we tested it on two previously unseen P4 programs, whose corresponding ASTs contained a total of 10 `else` blocks, including both empty and non-empty cases. The model correctly classified 12 out of the 15 instances, resulting in an 80% accuracy on this test set. While this reflects a slight decrease compared to its training performance, the result is still promising given the structural variability of `else` constructs in real-world P4 code.

These results suggest that the model successfully captures the contextual signals that distinguish empty `else` branches from meaningful ones. Its ability to detect redundant conditional structures can support automated refactoring pipelines and contribute to cleaner, more maintainable P4 codebases.

Beyond demonstrating technical feasibility, our results indicate that GNN-based learning provides a scalable path toward automated code improvement. Even with limited training data, the models extracted robust transformation patterns, suggesting practical applicability in real-world P4 codebases, where manual refactoring is costly and time-consuming.

5. Conclusion and Future Work

In this research, we have successfully developed three distinct Graph Neural Network (GNN) models: the Variable Renamer, the Parameter Reorderer, and the Empty Else Block Detector. Each of these models addresses specific aspects of P4 program refactoring, showcasing the versatility and potential of GNNs in this domain.

The Variable Renamer model receives the complete abstract syntax tree (AST) of a program as input and is tasked with renaming a specific variable throughout all its occurrences. It takes as parameters the graph representing the program and the names of the variable to be renamed along with the new name. By modifying the attributes of the relevant nodes in the AST, this model enhances code readability and consistency, which are crucial for maintaining large-scale software systems.

The Parameter Reorderer model operates at the level of function declarations, where it modifies the structure of the graph to rearrange the order of parameters. This capability is particularly beneficial for adhering to coding standards or managing default parameters, ultimately improving the clarity and readability of function definitions. By ensuring that parameters are organized in a logical manner, this model aids developers in writing more maintainable and understandable code.

The Empty Else Block Detector model focuses on prediction tasks within the graph. It identifies patterns that indicate the presence of empty else branches, which can be particularly useful for code optimization. By alerting developers to these non-essential branches, the model helps avoid unnecessary complexity in the code, thereby enhancing overall code quality and maintainability.

Through the development and evaluation of these models, we have gained valuable insights into how GNNs can learn to capture and manipulate different aspects

of graph data. This experience lays the groundwork for future research, where we aim to generalize these findings to create more complex models that integrate the strengths of each individual approach. By combining the capabilities of modifying node attributes, altering graph structures, and making predictions, we envision the development of sophisticated GNN architectures that can tackle a wider range of refactoring tasks.

Looking ahead, our goal is to explore the potential of hybrid models that leverage the strengths of the existing GNNs while introducing new mechanisms for learning and adaptation. Such models could incorporate advanced techniques such as attention mechanisms or reinforcement learning to further enhance their performance and applicability. By pushing the boundaries of what GNNs can achieve in the context of program analysis and refactoring, we hope to contribute to the creation of more intelligent and automated tools that assist developers in writing high-quality, maintainable code. By integrating these models into tooling pipelines, development teams can achieve faster iteration cycles, reduce manual overhead, and promote uniform, high-quality code across network applications.

References

[1] U. Alon, M. Zilberstein, O. Levy, E. Yahav: *code2vec: learning distributed representations of code*, Proc. ACM Program. Lang. 3.POPL (Jan. 2019), doi: 10.1145/3290353, url: https://doi.org/10.1145/3290353.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker: *P4: programming protocol-independent packet processors*, SIGCOMM Comput. Commun. Rev. 44.3 (July 2014), pp. 87–95, issn: 0146-4833, doi: 10.1145/2656877.2656890, url: https://doi.org/10.1145/2656877.2656890.

[3] J. Jones: *Abstract Syntax Tree Implementation Idioms*, Technical Report, PDF, University of Alabama, 2003, url: https://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf.

[4] A. Kheradmand, G. Roşu: *P4K: A Formal Semantics of P4 and Applications*, arXiv preprint (2018), arXiv:1804.01468.

[5] D. Lukács, G. Tóth, M. Tejfel: *P4Query: Static analyser framework for P4*, Annales Mathematicae et Informaticae 57 (2023), pp. 49–64, doi: 10.33039/ami.2023.03.002.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala: *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, arXiv:1912.01703, NeurIPS 2019, 2019, url: http://arxiv.org/abs/1912.01703.

[7] M. A. Rodriguez: *The Gremlin graph traversal machine and language (invited talk)*, in: Proceedings of the 15th Symposium on Database Programming Languages, DBPL 2015, Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 1–10, isbn: 9781450339025, doi: 10.1145/2815072.2815073, url: https://doi.org/10.1145/2815072.2815073.

[8] M. Tejfel, D. Lukács, P. Hegyi: *P4 Specific Refactoring Steps*, Acta Cybernetica 27.1 (Mar. 2025), pp. 53–65, doi: 10.14232/actacyb.308085, url: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4465.

[9] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun: *Graph neural networks: A review of methods and applications*, AI Open 1 (2020), pp. 57–81, ISSN: 2666-6510, DOI: https://doi.org/10.1016/j.aiopen.2021.01.001, URL: https://www.sciencedirect.com/science/article/pii/S2666651021000012.

[10] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu: *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*, in: Advances in Neural Information Processing Systems (NeurIPS), 2019.