Faculty of Mathematics
and Computer Science

Machine learning course (ML)

Graph Neural Networks for Static Program Analysis

Selegean Victor

Department of Computer Science, Babeş-Bolyai University
1, M. Kogălniceanu Street, 400084, Cluj-Napoca, Romania
E-mail: victor.v.selegean@gmail.com,
victor.selegean@stud.ubbcluj.ro

Abstract

This paper presents the applications of graph neural networks in the field of static program analysis by analyzing some recent papers. Static program analysis is a crucial step in software development. It enables the early detection of bugs, vulnerabilities, and inefficiencies without executing the code. Through that, it empowers developers to build better, faster, and stronger systems. Traditional techniques used for SPA are often reliant on rigid rules and linear code representations, while struggling to capture complex structures that span large sequences. At the same time, models like large language models, while powerful at some tasks, only represent information through token sequences, requiring complex embeddings to simulate context. This paper explores the application of graph neural networks as a new approach to SPA. Through the modeling of code as complex graphs, integrating abstract syntax trees, control flow graphs, and data flow graphs, GNNs are able to learn rich semantic representations that outperform traditional methods. The articles reviewed in this paper demonstrate the effectiveness of GNNs across diverse domains, ranging from vulnerability detection with models like Devign and MVD, to maintenance tasks. Introducing GNNs as a step in the compilation process through tools like GraalNN was able to achieve significant runtime speedup. The report analyzes the methodologies, datasets, and inherent challenges of these approaches. Among these are dataset imbalances and computational overhead. The conclusion drawn is that GNNs offer superior performance, but their integration into real-time development workflows require addressing their limited scalability and their high latency.

Keywords: Machine Learning; Graphs; Neural Networks; Graph Neural Networks; Static Program Analysis

1. Introduction

1.1. Graph Neural Networks and Static Program Analysis

As machine learning techniques evolved over the past decade, so has the need for processing complex relationships within data. Traditional machine learning techniques generally work on arrays and struggle to represent and exploit the complexity of real-world applications. Many domains such as internet search, navigation, and bioinformatics [14], use

graphs as means to represent their data. Graph Neural Networks, networks which are able to work directly with graph-like structured data, have emerged as an impressive and convenient specialization of the traditional neural network architecture.

Static Program Analysis (SPA) is a method of analysis of source code without the need of runtime information [12]. This approach allows automated tools to detect and signal potential issues ahead of time and direct developers to fix them. The issues detected by such tools can range from syntax errors, potentially infinite cycles to failed assertions and many more[1]. Such inspections generally rely on some intermediate representation of a program such as Abstract Syntax Trees (ASTs) [13] and Control Flow Graphs (CFGs) [3]. As their name suggests, these structures lend themselves to a graph representation. Because of this fact, static analysis tools need to work directly with nodes and directed edges.

There are multiple directions from which the problem of static program analysis can be approached. The one that may come to mind to most readers is bug identification. Some tools are able to detect when code does not perform the intended functionality. Another one of them is Vulnerability Detection. As mentioned by Cao et al. [1], "Conceptually, vulnerabilities are different from bugs. Vulnerabilities represent abusive functionality, but bugs represent wrong or insufficient functionality". Examples of vulnerabilities in software include the SQL Injection Attack, where there is no explicit bug hindering the experience of "lawful" users, but the application is unfit for release since it can be easily hijacked by malicious actors. Another topic falling under the same umbrella term of SPA is Static Profiling. This refers to estimating how often a piece of code is executed without having access to runtime knowledge. Such information can be helpful for compilers, be they JIT or otherwise.

This paper aims to dive into the ways in which Graph Neural Networks have been used to improve user experience, developer experience, and software security over the past few years. GNNs were introduced in 2005 [6] and since then, numerous studies have looked into their applicability in regards to SPA. A significant part of the work has been focused on vulnerability prevention. In 2019, Devign, an acronym for **De**ep **V**ulnerability **I**dentification via **G**raph **N**eural Networks [16], a GNN model specialized in the C programming language, was able to work directly with a program's AST and outperformed the state of the art models of that time. Then in 2021, BGNN4VD, acronym for **B**idirectional **G**raph **N**eural **N**etwork **For V**ulnerability **D**etection [1], expanded the architecture of GNNs by adding backwards edges and using them to learn the difference in features between vulnerable and non vulnerable code in C/C++ codebases. In 2022, MVD, **M**emory-related **V**ulnerability **D**etection [2], used a Flow-Sensitive GNN to leverage more of the information available in the graph for detecting vulnerabilities pertaining to memory usage in the Linux kernel. Aside from vulnerability detection, GNNs have been successfully integrated for the purpose of detecting code smells [11], obtaining static profiling information [8], and bug localization [15].

1.2. Structure of the Paper

The remainder of this paper will be structured in three distinct parts.

Section 2, will discuss the types of tasks which may be solved with GNNs and why GNNs are particularly useful in certain contexts.

Then, section 3 will look into some recent implementations and results of the methods discussed previously. Additionally, it will provide an overview of the datasets used by these implementations and the respective metrics used for evaluating the models.

Section 4 will discuss some of the limitations of the topics discussed until that point.

Last, section 5 will draw some judgments on the effectiveness of these techniques and future works.

2. Motivation

The motivation for applying Graph Neural Networks to Static Program Analysis stems from the fact that source code has an inherently complex structure. This is in contrast to other application domains, such as images or text, in which all inputs share a certain structure (such as a pixel grid or a sequence of tokens). The structures and relationships

[1] A complete list of inspections available in JetBrains products is available here https://www.jetbrains.com/help/inspectopedia/WhatIsNewInspections.html

between modules, classes, methods, and even variables lend themselves closer to a graph structure. GNNs are designed to process such data natively.

2.1. Graph Neural Networks

At the theoretical level, GNNs work using a mechanism named message passing[10]. Unlike LLMs, which view code still as a sequence of tokens, they view code as a collection of nodes (variables, statements, modules) and edges (control flow, data dependencies). In the message passing stage, the graph is subject to an iterative process where nodes exchange information with their neighbors, learning the context in which each node exists.

There are three types of tasks that a GNN is able to solve given some input graph.

2.1.1. Graph-Level Tasks

In graph-level tasks, the objective is to predict a property for the entire graph structure. The model aggregates the properties of all individual nodes to generate a vector representing the whole graph. This approach can be used for solving tasks like Vulnerability Detection or Malware Classification. In these scenarios, an entire source file or function is converted into a graph, and the GNN is trained to classify the entire network as ”Vulnerable” or ”Clean” [16], rather than identifying specific locations within the code.

Figure 1 shows two graphs. Each, when subject to a graph level task gets a label associated to it in its entirety.

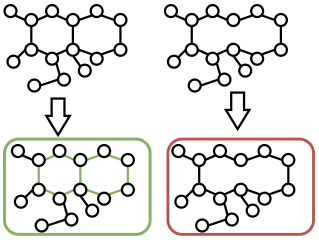

Fig. 1. Visual representation of a Graph-Level Task. The entire structure of the graph is aggregated to produce label.

2.1.2. Node-Level Tasks

In node-level tasks, the goal is to classify individual nodes within a graph. The model analyzes the context of a node’s neighborhood to determine its label. This translates to discovering specific locations of interest within a source file, as required by tasks like Bug Localization or Variable Misuse Detection. Here, the graph represents the code, and the nodes represent variables or statements. The GNN identifies which specific node is incorrect [15], highlighting the exact spot requiring attention.

Figure 2 shows a graph which, when subject to a node-level task gets a label associated to each of its nodes.

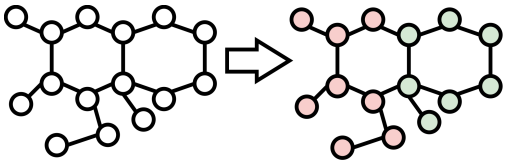

Fig. 2. Visual representation of a Node-Level Task. Individual nodes are classifies based on their context within the structure

2.1.3. Edge-Level Tasks

Edge-level tasks involve predicting the properties of edges. In the context of SPA, this is highly relevant for Static Profiling and compiler optimization. By modeling the Control Flow Graph, a GNN can predict the likelihood of a specific edge being taken during execution [8]. This allows compilers to optimize the most frequently traversed paths without needing to run the program dynamically.

Figure 3 shows a graph which, when subject to a edge-level task gets a label associated to each of its edges.

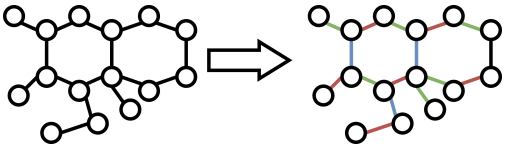

Fig. 3. Visual representation of a Edge-Level Task. Each relationship between nodes is classified based on the role it plays within the structure.

2.2. Application Domains

The alignment between GNN tasks and code structure brings improvements in several domains over traditional methods. Traditional vulnerability scanners often rely on linear pattern matching, which yields high false-positive rates [9]. Vulnerabilities like buffer overflows or use-after-free errors often depend on complex, non-local interactions that span multiple functions. **Graph-level** tasks can capture more subtle dependencies, allowing for the detection of syntactically correct but semantically dangerous functionality.

Additionally, maintaining code quality requires both knowledge of common coding patterns and of the project being worked on. Detecting ”code smells” requires understanding the structural complexity of classes and methods. **Node-level** GNNs can automate such tasks by learning the structural signatures of high-quality code versus ”bad-smelling” code [11], acting as an automated code reviewer that goes beyond simple style checking.

Last but not least, GNNs can impact the efficiency of the software itself. Compilers rely on heuristics to decide which functions to inline or how to allocate registers. By utilizing **edge-level** tasks to predict execution frequencies during static profiling, GNNs provide compilers with more insights during the compilation phase, leading to more efficient machine code generation.

3. Approaches and Results

3.1. Vulnerability Detection

The application of GNNs to vulnerability detection is primarily treated as a graph-level problem.’ The model is given a code snippet represented as a graph and it must determine if the input contains any vulnerabilities. This domain is a rapidly developing one, moving from simple representations to large, composite structures that are able to represent more complex semantic relationships.

3.1.1. Composite Semantics and Graph-Level Classification

A representative work in this domain is **Devign** (Deep Vulnerability Identification via Graph Neural Networks) [16]. Within the paper, the authors argue that traditional, flat, representations (like token sequences used in Recursive Neural Networks) are unable to capture the code’s structure. To address this, a new representation was introduced.

Devign represents a function not as a single graph type, but as a joint graph embedding four distinct layers of information:

- **Abstract Syntax Tree:** Captures the syntactic structure.
- **Control Flow Graph:** Models the execution paths.
- **Data Flow Graph:** Tracks variable usage and dependencies.
- **Natural Code Sequence:** the lexical order of source code tokens.

Figure 4, sourced from the original article [16], shows how nodes can be part of multiple graphs at once.

These sub-graphs are merged into a single graph which is then processed by a **Gated Graph Neural Network**. A key contribution of Devign was the introduction of a convolutional layer applied after the GNN layers to select relevant features for the final graph-level classification. Evaluated on large C-language projects (Linux Kernel, QEMU), Devign achieved an accuracy of 74.11% on detecting zero-day vulnerabilities, outperforming traditional static analyzers like Flawfinder [16] by a significant margin.

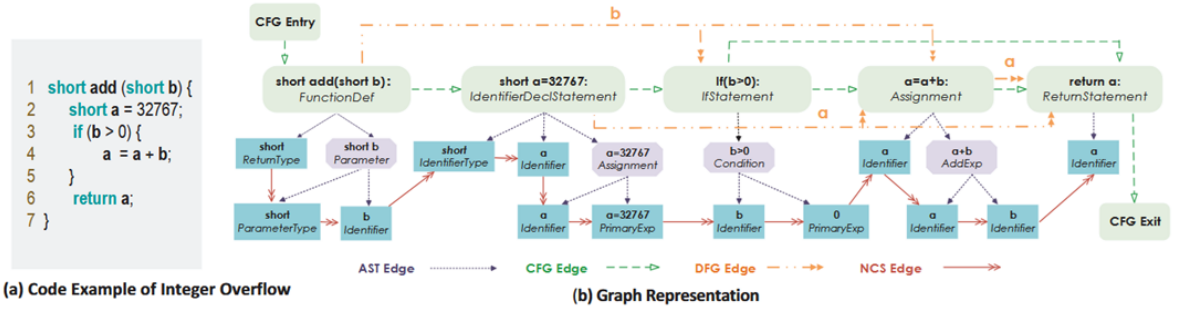

Fig. 4. Graph representation of a code section with integer overflow

3.1.2. Bidirectional Information Propagation

While Devign utilized multiple graph types, it relied on standard message passing which often treats edges as undirected in a way that may confuse "cause and effect". In 2021, it was proposed in "*BGNN4VD*: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection" [1] to address the limitations of propagation.

The main idea behind BGNN4VD is that vulnerability patterns are usually dependant on the context. A vulnerable statement is defined not just by what it affects (forward flow), but by what affects it (backward flow). The model explicitly adds backward edges to the AST, CFG, and DFG representations. This allows the message-passing mechanism to propagate information in both directions simultaneously. Figure 5, sourced from the original article [1], shows the composite code graph produced by combining the AST, CFG, and DFG of a short program for finding the larger of two numbers.

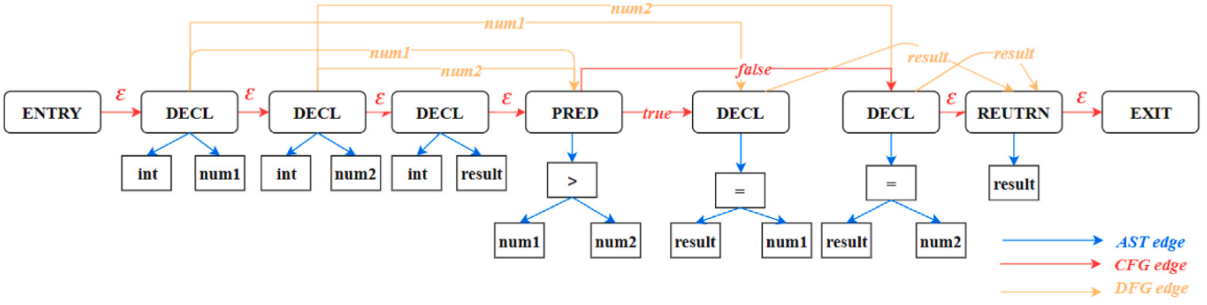

Fig. 5. Code Composite Graph

Experiments on the National Vulnerability Database (NVD) and the Software Assurance Reference Dataset (SARD) showed that this bidirectional approach allows the model to capture context more effectively. The model achieved an F1-measure of 76.8%, which was a 4.9% improvement over state-of-the-art sequence-based models like SySeVR [1].

3.1.3. Flow-Sensitivity and Specificity

As GNN models matured, research began to focus on specific classes of bugs that are harder to detect. 2022 saw the introduction of "Memory-related Vulnerability Detection" [2], specifically targeting memory errors such as buffer overflows and resource leaks.

MVD exploits a resource underutilized in previous models, flow information. Generic GNNs treat all edges similarly, potentially losing critical data-flow anomalies such as a variable being used after it is freed between unrelated syntax edges. MVD employs a **Flow-Sensitive GNN** that embeds unstructured information (source code tokens) with structured flow information (control and data dependencies). By breaking down the graph to focus only on statements relevant to memory operations, MVD ignores information it deems irrelevant.

This specialization allowed MVD to outperform other GNN detectors, achieving a detection accuracy of 74.1% on complex real-world memory vulnerabilities [2].

3.2. Code Smell Detection

While vulnerabilities represent security risks, ”code smells” represent maintenance risks. Detecting them may be a node-level or graph-level classification task, depending on whether the smell is localized to a function or a class.

In 2025, Slamet and Rochimah [11] applied GNNs to detect three specific types of smells in Python code, namely ”Long Method”, ”Large Class”, and ”Duplicated Code”. Traditional approaches to this problem relied on manually engineered metrics like Cyclomatic Complexity or Lines of Code and standard classifiers, which are sometimes not able to capture enough structural context.

The approach uses ASTs converted into Function Call Graphs to preserve the structural relationships between methods. The model uses a Graph Convolutional Network with three layers to learn the structure of smelly code.

When evaluated on real-world projects like ERPNext and Odoo, the model achieved an accuracy of 95.95% and an F1-score of 97.78%, outperforming traditional machine learning models like SVM and Decision Trees [11]. The results highlight the GNN’s ability to learn complex structural patterns—such as the ”spaghetti code” structure often found in long methods—without explicit feature engineering.

3.3. Static Profiling

Moving beyond classification, GNNs have also been applied to regression and probability estimation tasks, essential for compiler optimization. Static Profiling is a classic edge-level task, where the goal is to predict the probability of taking a specific edge in the Control Flow Graph.

Milikic et al. introduced **GraalNN** [8], a framework integrated into the GraalVM compiler. Unlike previous heuristics that rely on simple rules, GraalNN uses a GNN to learn execution probabilities directly from the CFG structure. A novel contribution of this work is its context-sensitive approach. GraalNN’s work is divided in two phases. First, the context-insensitive one, where the model predicts branch probabilities based on the CFG of a function during parsing. Then, in the second, context-sensitive, phase, the model refines these predictions by incorporating the caller’s graph, allowing it to understand the broader execution context. This approach allows the compiler to make better optimization decisions, such as which functions to inline.

GraalNN demonstrated a 10.13% runtime speedup on industry-standard benchmarks (DaCapo, Renaissance), overtaking previous state-of-the-art static profilers [8].

3.4. Bug Localization

Finally, GNNs can be used at the task of bug localization. Unlike vulnerability detection, which flags a whole file, bug localization must pinpoint the exact node (variable or statement) that is incorrect.

Yousofvand et al. [15] proposed a method for locating bugs in JavaScript code by treating the problem as a node classification task. The challenge of training a model in this domain is the extreme class imbalance: in any given file, nearly all nodes are ”bug-free,” and only one or two are ”buggy”. In order to solve this imbalance, the training data went through a three-step transformation. First, a graph was constructed by mapping JavaScript source code to a graph based on its AST. Then, the buggy AST was compared with the correct alternative and all nodes found to be non-matching were labeled as ”buggy”. At the end, the training examples got balances through oversampling of nodes labeled ”buggy”.

Using a 2-layer Graph Convolutional Network, the model achieved an accuracy of 90.1% in localizing bugs, detecting complex issues like undefined properties and variable misuse that traditional static analysis tools often miss [15].

3.5. Datasets and Evaluation Metrics

Measuring the performance of GNNs in Static Program Analysis depends heavily on the representativeness of the training data. Across the reviewed articles, a common pattern emerges. There is a move away from synthetic datasets

and towards complex, messy, real code repositories. This section contains an overview of the data sources and the specific metrics used to evaluate the discussed models.

3.5.1. Data Sources

The datasets used can be roughly categorized into synthetic collections and real-world software repositories.

Early research often relied on synthetic or semi-synthetic datasets to ensure a balanced distribution of code examples. BGNN4VD used the SARD, a collection of test cases with known vulnerabilities, alongside entries from the NVD [1]. Similarly, MVD constructed a custom dataset combining synthetic samples from SARD with real-world vulnerabilities from the Critical Vulnerability Exploit database (roughly, in a 5:2 ratio) to ensure sufficient volume for training deep learning models [2].

Other approaches have shifted focus to real-world applications. Devign was evaluated on very large open-source C and C++ projects including the Linux Kernel, QEMU, and FFmpeg [16]. These projects offer the advantage of rich and documented history of security vulnerabilities and associated patches. For code smell detection, the models used source code of large systems like ERPNext and Odoo [11]. These repositories mostly consist of Python and JavaScript [5], [4]. The reason for choosing these repositories is their maturity and complexity. When it comes to bug localization, the dataset used was Hoppity [15], which contains pairs of ”broken” and ”fixed” code for which the respective ASTs are as similar as possible [7]. Last but not least, for static profiling, GraalNN used both industry standard benchmarks of Java code as well as microservices frameworks like Spring and Quarkus [8].

3.5.2. Evaluation Metrics

The choice of metrics depended on the final task attempting to be solved: classification or regression. For tasks such as vulnerability detection, code smell detection, and bug localization, the standard metrics for binary and multi-class classification are employed. **Accuracy** is understood as the overall percentage of correctly classified instances. **Precision and recall** were used for security tasks where false negatives (missed vulnerabilities) are dangerous, and false positives (false alarms) waste developer time. **F1-Score** was used as a metric to compare model performance on imbalanced datasets due to its ability to provide a more reliable measure of effectiveness than accuracy when the majority class vastly outnumbers the minority class. **AUC-ROC** was used to evaluate the trade-off between true positive and false positive rates at various thresholds.

For Static Profiling, where the goal is performance improvement rather than labeling, standard classification metrics are insufficient. GraalNN instead uses operational metrics, measuring runtime speedup, binary size, and compilation time. **Runtime speedup** was the percentage reduction in execution time of the compiled program. **Binary size** showcased the trade-off between performance speed and executable size. **Compilation time** increased as a result of introducing GNN inference during the build process.

4. Discussion

The use of GNNs in SPA has demonstrated significant potential, but is still subject to several challenges. The reviewed approaches outperform traditional methods. However, an analysis reveals limitations regarding data dependency, scalability, and granularity.

4.1. Data Imbalance and Annotation Quality

High quality and labeled data is as of yet still scarce. In vulnerability detection and bug localization, fraction of ”buggy” code is statistically insignificant compared to code that is ”correct”. This class imbalance biases models toward predicting the majority class. In order to solve this issue, the techniques of synthetic augmentation, and manual labeling were used. Augmenting the dataset, while effective for balancing the training data, may not reflect the complex semantics of real world bugs, introducing unnecessary noise that may bear consequences later. Manual labeling relies on the intensive labor of human annotators, which limit the speed at which new datasets can be created and introduce human bias.

4.2. Scalability and Computational Overhead

GNNs are computationally expensive compared to traditional static analysis tools or simpler machine learning models. The training cost in itself is significantly higher than that of sequence-based models like VulDeePecker. While training is a one-time cost, high latency hinders rapid model iteration and tuning.

In the context of static profiling, GraalNN introduced a large compilation latency. This cost is justified for long running applications, but may be prohibitive for environments where build speed is an important facet, such as CI/CD pipelines.

4.3. Granularity and Context trade-off

As the model is able to capture more context, it loses the ability to analyze granular sections. This materializes in models picking a comfortable position on the spectrum between the two. Coarse-Grained Models like Devign operate at the function level. This allows them to process large codebases efficiently but fails to pinpoint the exact location of a bug, leaving the developer to manually search within the flagged function. Fine-Grained models like MVD work with statement(node)-level precision. However, this often requires expensive pre-processing steps like program slicing to reduce the graph size.

5. Conclusions

This report has explored the impact of Graph Neural Networks on Static Program Analysis. Treating source code as a structured graph rather than a linear sequence of tokens lends GNNs the ability to bridge the gap between syntax and semantics.

The articles upon which this report is based show that GNNs are versatile. They are able to solve tasks on graph, node, and edge level. Devign, BGNN4VD, and MVD are able to detect vulnerabilities. Slamet and Rochimah were able to create a GNN that detects code smell. GraalNN deals with static profiling. Not least, Yousofvand et al. have managed to deal with bug localization. While earlier models focused on generic graph classification, more recent ones incorporate refinements which are specific to the domain of SPA, among which bidirectionality and context-sensitive inlining.

The translation from academic research to industrial practice is still not complete. Ever-present bottlenecks are the reliance on large, balanced datasets, and a significant computational cost for inference. These issues pose challenges to integration in systems like IDEs or compilers. As it seems now, the balance between cost and improvements must be carefully considered and the choice to integrate them must be left to the end users. Developers of critical systems might appreciate the enhanced safety in the detriment of compilation speed, while the developers of internal tools might not. Future work should focus on lightweight GNN architectures that reduce latency and techniques that mitigate the dependency on labeled datasets.

In conclusion, despite these challenges, GNNs provide important insights and might become as ubiquitous as traditional SPA tools like cppcheck, eslint, and clang. Results like GraalNN's 10% runtime speedup and MVD's 74% detection accuracy suggest that GNNs represent the future of automated software analysis.

References

[1] Cao, S., Sun, X., Bo, L., Wei, Y., Li, B., 2021. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. 136. URL: https://doi.org/10.1016/j.infsof.2021.106576, doi:10.1016/j.infsof.2021.106576.

[2] Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C., 2022. Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks, in: Proceedings of the 44th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 1456–1468. URL: https://doi.org/10.1145/3510003.3510219, doi:10.1145/3510003.3510219.

[3] Control Flow for the GCC, . URL: https://gcc.gnu.org/onlinedocs/gccint/Control-Flow.html. online; Accessed October 2025.

[4] Github ERPNext Repository, . URL: https://github.com/frappe/erpnext. online; Accessed December 2025.

[5] Github Odoo Repository, . URL: https://github.com/odoo/odoo. online; Accessed December 2025.

[6] Gori, M., Monfardini, G., Scarselli, F., 2005. A new model for learning in graph domains, in: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., pp. 729–734 vol. 2. doi:10.1109/IJCNN.2005.1555942.

[7] Hoppity Dataset, . URL: https://github.com/AI-nstein/hoppity. online; Accessed December 2025.

[8] Milikic, L., Cugurovic, M., Jovanovic, V., 2025. Graalnn: Context-sensitive static profiling with graph neural networks, in: Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization, Association for Computing Machinery, New York, NY, USA. p. 123–136. URL: https://doi.org/10.1145/3696443.3708958, doi:10.1145/3696443.3708958.

[9] Park, J., Shin, J., Choi, B., 2023. Reduction of false positives for runtime errors in c/c++ software: A comparative study. Electronics 12. URL: https://www.mdpi.com/2079-9292/12/16/3518, doi:10.3390/electronics12163518.

[10] Sanchez-Lengeling, B., Reif, E., Pearce, A., Wiltschko, A.B., 2021. A gentle introduction to graph neural networks. Distill doi:10.23915/distill.00033. https://distill.pub/2021/gnn-intro.

[11] Slamet, J., Rochimah, S., 2025. Development of code smell detection model based on graph neural network to detect long method, large class, and duplicated code. Edelweiss Applied Science and Technology 9, 566–579. URL: https://learning-gate.com/index.php/2576-8484/article/view/8672, doi:10.55214/25768484.v9i7.8672.

[12] Static Code Analysis Guide, . URL: https://www.jetbrains.com/pages/static-code-analysis-guide/. online; Accessed October 2025.

[13] Structure of GCC, . URL: https://gcc.gnu.org/wiki/StructureOfGCC. online; Accessed October 2025.

[14] Tanis, J.H., Giannella, C., Mariano, A.V., 2024. Introduction to graph neural networks: A starting point for machine learning engineers. URL: https://arxiv.org/abs/2412.19419, arXiv:2412.19419.

[15] Yousofvand, L., Soleimani, S., Rafe, V., Nikanjam, A., 2025. Graph neural networks for precise bug localization through structural program analysis. Automated Software Engineering 33. doi:10.1007/s10515-025-00556-y.

[16] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, in: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (Eds.), Advances in Neural Information Processing Systems, Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf.