Deep Learning-Based Code Vulnerability Detection: A New Perspective

Bachelor Thesis

Bachelor of Science
Department of Business Information Systems
Major Data Science
Baden-Wuerttemberg Cooperative State University

Amos Dinh

Prof. Dr. Maximilian Scherer, Academic Supervisor
Dr. rer. nat. Martin Härterich, SAP SE, Company Supervisor
12th of February - 6th of May 2024

Declaration of Originality

I herewith declare that I have composed the thesis

"Deep Learning-Based Code Vulnerability Detection: A New Perspective"

myself and without the use of any other than the cited sources and aids. Furthermore, the submitted electronic version of the thesis matches the printed version.

Speyer, 6th of May 2024

Amos Dinh

Abstract

Automatic code vulnerability detection is an ongoing research field. The employed algorithms detect whether a piece of source code contains a vulnerability that could render the whole application open to malicious attacks. Among recent methods, Deep Learning-based approaches have been proposed which leverage token- or graph-based source code representations to discover vulnerabilities.

In the current work, the performance of Deep Learning-based methods is investigated by employing Graph Neural Networks on a large vulnerability detection dataset. In detail, we examine the dimensions *data*, *architecture*, *training* and *evaluation* and show how a simple baseline which measures only the code complexity outperforms both Graph Neural Networks and Large Language Model-based approaches.

Further, performance is not improved with different architectures such as Graph Structure Learning-based and heterogeneous models, nor with specifically devised multitask- and multistage pretraining on the code graphs.

We demonstrate, how the dataset composition skews the performance and communicates overoptimistic results. Consequently, only rigorous evaluation, including careful train-test separation on code-project level, stratifying the the predictions by code complexity and the comparison against an appropriate baseline, depicts the models' detection capability more truthfully. The findings are not specific to the dataset but affect multiple other datasets in the field.

Now being able to measure the detection capability of models more precisely, we conclude with the findings that for increasing vulnerability detection performance, more data is needed, and simple model architectures suffice in the current setting.

Table of Contents

List of Abbreviations

AST:	Abstract Syntax Tree
CFG:	Control Flow Graph
CG:	Call Graph
CPG:	Code Property Graph
CVE:	Common Vulnerabilities and Exposures
CWE:	Common Weakness Enumeration
DFG:	Data Flow Graph
DL:	Deep Learning
GCN:	Graph Convolutional Network
GGNN:	Gated Graph Neural Network
GIN:	Graph Isomorphism Network
GNN:	Graph Neural Network
GSL:	Graph Structure Learning
GraphGLOW:	Graph Structure Learning Model for Open-World Generalization
HGP-SL:	Hierarchical Graph Pooling with Structure Learning
LLM:	Large Language Model
MLP:	Multilayer Perceptron
MSE:	Mean Squared Error
NLL:	Negative Log Likelihood
NN:	Neural Network
OSS:	Open Source Software
RGCN:	Relational Graph Convolutional Network
WL test:	Weisfeiler-Lehman Graph Isomorphism Test

List of Figures

List of Tables

1 Introduction

Security flaws in program code leave software and applications vulnerable to attacks with malicious intent. The applications are increasingly based on a variety of underlying Open Source Software (OSS) libraries. Thereby, the attack surface is drastically expanded, since software protocols differ and sufficient security checks might exist within a library but interfaces between libraries, as well as the modularity and complexity of today's applications, creates room for weaknesses. Simultaneously, the manual labor of security experts is costly, time consuming [1] and grows with project complexity. A a result it might be infeasible to verify the safety of all code manually.

Automatic methods have been developed to aid the manual vulnerability discovery process. Static methods such as "FlawFinder" [2], [3] analyze source code by matching it against a known list of tokens or patters which indicate vulnerabilities. Dynamic methods analyze code at runtime. Techniques such as fuzzing inject pseudo-random input into code and examine the output to discover bugs or vulnerabilties [4].

The availability of open source code as a data source permits the application of Deep Learning (DL)-based methods. However, how to effectivly gather and curate datasets for vulnerability discovery is still an unsolved problem. The promise of DL-based methods is that they may detect more vulnerabilities while reducing the number of false alerts. When applied effectively, they could alleviate the shortcomings of traditional static and dynamic detection such as limitations of manually defined rule sets or randomized compute-limited fuzzing that requires program code to be executable. More effective automatic means would both increase security detection coverage across the OSS landscape and correspond to many saved security expert-hours. Ultimately, they would help defend against cyber-security attacks, both of economical and political value.

For this purpose, the present work examines Graph Neural Networks (GNNs) to carry out the task of binary vulnerability classification on function level Code Property Graphs (CPGs). The graphs are extracted from function files of the relatively new DiverseVul dataset [5]. We employ GNNs because they are small in parameter size, allowing for fast experimentation. Additionally, authors of the Diversevul Dataset [5] have already examined Large Language Model (LLM) models' performance.

We focus on exploring the relevant factors *data*, *architecture*, *training* and *evaluation* within the domain of GNNs to gain a holistic understanding of the current state

of DL-based automatic vulnerability detection. Overall, it is discovered that careful evaluation of the models reveals their inability to learn significantly more than a naive baseline.

Followingly, the key contributions of this work include:

Data: The most important pillar of the work is the splitting of the DiverseVul dataset [5] on project level, preventing test-data leak. Project level splits were previously not practical, as former datasets did not exhibit sufficient size giving rise to inflated performance results. Earlier work such as [5] and [6] have examined the setting of out-of-training distribution testing. However, their work still measures performance and draws conclusions from the settings in which training and test samples share the same projects. We reason that the test set contaminations the authors notice in their own work make it difficult to draw any conclusions in the former setting. Therefore, we abandon it entirely in the present work.

The minor detail of train-test separation allows for a more truthful depiction and examination of DL-based performance in vulnerability detection measured in a realistic and arguably more useful scenario where methods are applied to unseen codebases.

Further, we empirically investigate the impact on performance of adding features such as node degree and triangle counts to the graph samples. Additionally, it is examined whether directed, undirected and heterogeneous variations of the underlying graphs yield best performance, also by leveraging the Relational Graph Convolutional Network (RGCN) [7] architecture.

Architecture: We investigate the performance of GNNs on the DiverseVul dataset [5]. In particular, different architectures including the state-of-the-art method ReVeal [6] as well as base architecures such as the Graph Convolutional Network (GCN) [8], Graph Isomorphism Network (GIN) [9] and Graph Structure Learning based methods.

Training: We derive and examine the performance of various pretraining methods, including multi-task and multi-stage pretraining.

Evaluation: An initial evaluation reveals suprisingly similar performance between the models. Delving deeper, we find different simple tools to be helpful to compare performances more accurately, controlling for the unwanted factors of variation in the datatset. These tools include a simple baseline, as well as stratification per vulnerability type.

In the following, in Chapter 2 we will first establish important concepts related to the pillars *data*, *architecture*, *training* and *performance*, such as how the CPG of program code is constructed, an introduction to GNNs and graph classification, as well as pretraining and the balanced accuracy metric. Chapter 3 introduces related work and in Chapter 4 we present the experiments. Finally, in Chapter 5 we re-evaluate and highlight the conclusions drawn.

The code for all experiments is available at https://github.com/AmosDinh/security-research-graph-learning.

2 Preliminaries

This chapter introduces the knowledge utilized throughout the work. First, general graph concepts are defined, followed by an overview over automatic vulnerability detection. Finally, GNNs in conjunction with related concepts such as graph classification, expressiveness of GNNs, Graph Structure Learning (GSL) and pretraining on graphs are reviewed, upon which attempts at increasing the vulnerability detection performance are based.

2.1 Graph Concepts

Graphs have become an important tool in areas such as geography, chemistry, sociology, linguistics or computer science to model and reason about concepts [10] or solve problems.

A simple graph G can be defined in terms of a set vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$ which connect vertices in a directed or undirected manner, (v_1, v_2) or $\{v_1, v_2\}$ [10]. For a graph in general, multiple edges between the same vertices might exist and the graph may contain self-loops connecting one node to itself.

$$\mathbf{A}_{ij} = \begin{cases} 1 \text{ if } (v_i, v_j) \in \mathcal{E} \\ 0 \text{ else} \end{cases} \tag{1}$$

For a simple directed graph the set of edges can be expressed in terms of the adjacency matrix Equation 1. The data representation as adjacency matrix allows for simple formulation of message passing in GNNs explained in a subsequent section.

A heterogeneous graph additionally includes the mapping functions $\tau(v) : \mathcal{V} \to \mathcal{A}$ and $\varphi(e) : \mathcal{E} \to \mathcal{R}$ which map vertices, also called nodes, and edges to their respective node and edge types [11]. This translates to the graph representation containing multiple adjacency matrices, one for each edge type. In a social network, where nodes might represent people, categories such as “child”, “adult” could be formulated as node types and edge types might represent relationships between people such as “is friend”, “is spouse”.

In the present paper, node degree and triangle count are examined for usefulness as features for DL-based vulnerability detection. The degree-count is the number of neighbors a node is connect to through edges. In the directed manner, there exist two degree statistics, an in-degree and out-degree for each node. The triangle count is the number of 3-cliques a node v participates in [12]. Node v is part of a triangle if there are distinct u and w such that v, u, w areconnected by an edge. The number of degree and triangle counts can directly be employed to determine the local clustering coefficient of a node [12]. The clustering coefficient can for instanc help detect spamming activity in web graphs [12].

In the following sections we use "1" to denote the identity matrix of appropriate size which has ones on the diagonal and zeros elsewhere.

2.2 Automatic Vulnerability Detection

This section introduces common vulnerability detection concepts.

Definition:

> *"[A code vulnerability can be defined as a] weakness in the computational logic […] found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability. […]"*
>
> — CVE Definition, National Institute of Standards and Technology [13]

The most common categorization of vulnerabilities is comprised by the **Common Vulnerabilities and Exposures (CVE)** database [14]. Companies and individuals can submit discovered instances of vulnerabilities in programm code to the CVE database, which are then mapped to one or multiple underlying weaknesses, categorized by the Common Weakness Enumeration (CWE) [15] classes in a hierarchical manner.

To give an example, common weaknesses include CWE-787 "Out-of-bounds Write" or CWE-79 "Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')" [16]. In CWE-787 the program writes data into computer memory past the intended bounds. An attacker can thereby overwrite parts of computer memory which were not intended to be writable. Subsequent access to the overwritten locations may now result in operations specified by the attacker.

The field of **automatic vulnerability detection** is concerned with discovering security relevant weaknesses in software by automatic means. Especially C++ and C code is in the focus of security researchers, because of their widespread use, and thereby data availability, and larger attack surface in comparision to higher-level programming languages like Python.

Code vulnerability detection is inherently a difficult task. Grounded in the halting problem, detecting all vulnerabilities in a program with the help of another program which terminates in finite time is theoretically intractable [17].

The project OSS-Fuzz [18] illustrates this claim in a practical manner. It tests the code of multiple OSS projects continuously against trillions of generated test cases every week to discover new vulnerabilities. The endeavor emphasizes that efficient and effective vulnerability detection is still a largely unsolved challenge. [19] group efforts of automatic vulnerability detection into three categories:

Static analysis examines a program without execution. Graph-based static analyzers model the program as a heterogeneous graph with vertices representing program statements. The implementation employed in this work [20], [21] converts C files to their graph representation, a Code Property Graph (CPG). This graph contains multiple directed edge types:

- The Abstract Syntax Tree (AST) contains the main edge type in the graph. It splits the program on syntactical symbol level. For example, the assignment `*prelink = '0';` is split into nodes `*`, `prelink`, `=` and `'0'` which are connected in an hierarchical structure with `=` as the root node.
- The Data Flow Graph (DFG) represents data dependencies between operations. It tracks access and modification of variables [22].
- The Control Flow Graph (CFG) marks a subset of the AST nodes as control nodes and connects them to model the program control flow. The flow describes all possible paths which could be taken over the course of execution and is determined by statements such as *if*, *for* or *switch* [22].
- The Call Graph (CG) represents method caller and receiver relationships.

Parsers such as [20] are "forgiving" in that they do not require programs to be executable to create the CPG. Graph-based static analyzers then model and identify vulnerabilities based on the CPG. They can examine high volumes of code but suffer from the lack of run-time information, making them subsceptible to a high false-positive rate [23].

Dynamic analysis checks programs for vulnerabilities during run-time. Fuzzers like the aforementioned OSS-Fuzz [18] inject random data into program code to induce unexpected behavior. The process is augmented with heuristic tools. Dynamic taint analysis tracks the dataflow of user-controlled data [19]. Any operation which uses the data is also marked as tainted. Pontential vulnerabilities are then discovered by identifying security critical operations such as control flow operations or system calls which come in contact with the tainted data. Drawbacks of dynamic analyzers include their low code coverage. Therefore, these methods suffer from a high false-negative rate, as they can not create different enough input data to show that the code sample is vulnerable [23].

Mixed analysis combines both static and dynamic methods. Concolic execution [19] executes a program with a random input while collecting symbolic constraints. A solver then creates a new input based on the symbolic information which steers the execution towards a different path. Therefore, mixed analysis can alleviate the drawbacks of static and dynamic analysis, combining high code coverage with access to run-time information.

2.3 Graph Neural Networks

In this section the concept of Graph Neural Networks GNNs as well as GNN architectures are introduced which we use for the vulnerability classification.

GNNs are utilized in various fields which use graphs to model concepts (Section 2.1). They have been employed to recommend content to users [24], model football dynamics [25] and accelerate fluid simulation [26].

They are amongst the most general classes of Artifical Neural Networks (ANNs) [27]. For example, architectures such as the Convolutional Neural Network (CNN) [28] as well as the Transformer [29] can be formulated as GNNs [30].

Most common GNNs use the adjacency matrix $\mathbf{A}$ of a graph to operate on a node feature matrix $\mathbf{X}$ which specifies an informative vector $\mathbf{x}$ per node. They apply permutation equivariant functions $\mathbf{F}(\mathbf{X})$ by the use of permutation invariant functions such as SUM, MEAN and MAX to aggregate local neighborhoods [30]. A Recurrent Neural Network [31] may also be used as aggregator [32].

$$
\begin{aligned}
H^{l+1} &= \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{l}W^{l}\right) \\
H^{l+1}_{i,\cdot} &= \sigma\left(\sum_{j\in\mathcal{N}(i)\cup\{i\}} \frac{\hat{A}_{ij}}{\sqrt{\hat{D}_{ii}\hat{D}_{jj}}} H^{l}_{j,\cdot}W^{l}\right)
\end{aligned}
\tag{2}
$$

The **GCN** [8] introduces the concept of graph convolution (Equation 2) where σ is a non-linear activation function, $H^0 = \mathbf{X}$ and subsequent H are learned node representations at each layer l, $\hat{A} = \mathbf{A} + I$, $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$ and the weight W^k is a learnable weight matrix. Similar to the convolution operation in CNNs, the graph convolution is an aggregation of the local neighborhood around each node i.

The weight matrices W^k can be seen as "message" weights which transform the neighbors' j node representations $X_{j,\cdot}$, as shown in the lower part of Equation 2. The neighbors' messages are aggregated by the SUM aggregator and finally transformed by a non-linearity σ, "sending" a single message to create node i's represenation $H^{l+1}_{i,\cdot}$ at layer $l+1$. Before aggregation, each neighbor j's message is normalized by the geometric mean of i's and j's (in-)degree. In combination with the SUM aggregator this operation does not amount to a mere averaging of neighbors of i but models more complex relationships.

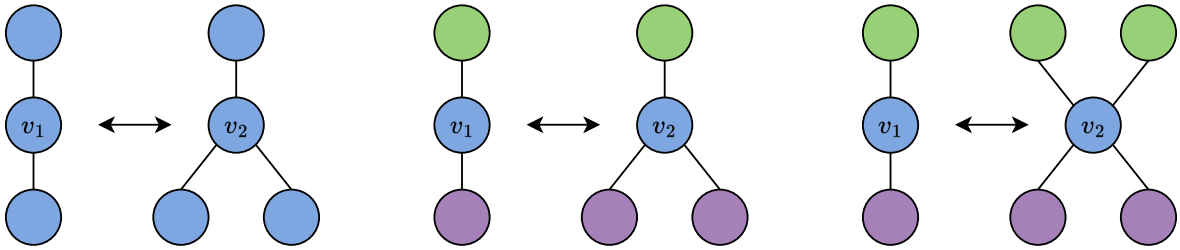
Besides a simple binary connection indicator, $\hat{A}_{ij}$ can represent continuously-weighted connections as well. Furthermore, the addition of the identity matrix I to A allows nodes' representations of the layer $l+1$ to include information of their own representation in the previous layer l.

The authors of the work introducing the **GIN** [9] examine the expressivity of GNNs and derive the GIN architecture as a result of their theoretical examination. Basis for their examination is the Weisfeiler-Lehman graph isomorphism [33] test. The test is used to determine if two graphs G and G' have the same structure.

1. Initially the same label is assigned to all n nodes in a graph
2. In each following iteration, the node labels of each neighbor are passed to the node, creating a multiset of node labels. This multiset, besides the node labels, also contains the information about the count of each label.
3. Then each node is assigned a new label based on a hash of the multiset of their neighbors' labels.

4. Steps 2 and 3 are repeated. If the algorithm converges before reaching n repetitions, meaning no new label-hashes are created and both graphs have identical label structure after sorting, the graphs can be declared as isomorphic.

The test is known to fail in some cases [34], but is overall able to distinguish graph structures.



a) Mean and Max fail b) Max fails c) Mean and Max fail

Figure 1: Common GNN aggregators fail to compute distinguishable node representations for v_1 and v_2 given their neighborhood structure and neighbor features represented by the color.

Examining the message passing mechanism commonly shared between GNN-architecture, [9] argue that GNNs can be maximally as powerful as the Weisfeiler-Lehman Graph Isomorphism Test (WL test) in distinguishing graph structures. Especially MEAN and MAX aggregators fail in computing distinguishable node representations based on the multiset of neighbors, which is illustrated in Figure 1.

$$
\begin{aligned}
H^{l+1} &= \mathrm{MLP}(\hat{A}H^{l}) \\
H^{l+1}_{i,\cdot} &= \mathrm{MLP}(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \hat{A}_{ij} X_{j,\cdot})
\end{aligned}
\tag{3}
$$

Followingly, they propose the GIN architecture (Equation 3), which utilizes the SUM aggregator and unlike conventional GNNs architectures employs a Multilayer Perceptron (MLP) [35], not a single-layer one, to pass the neighbors' messages. The MLP in combination with the SUM aggregator allows the model to maintain injectivity, by the universal approximation theorem [36], similar to the hash-function in the WL test. The authors prove empirically that the model is as expressive as WL test and

more expressive than models such as the GNN by measuring the models' overfitting performances.

Expressivity is beneficial in the case of model pretraining as observed by the authors in [37] following [38]. The paper employs the GIN attempting to improve model performance.

2.3.1 Graph Classification

GNNs can be utilized for a range of tasks on graph data. Widely used tasks include:

- link prediction to predict missing links between nodes such as to fill in missing knowledge in relational data [39],
- node classification to detect bots in social networks [40],

and graph classification, which is employed in the present paper as binary classification to classify if program code includes a vulnerability. Because the number of nodes varies between graphs, a fixed classification head such as a MLP is not sufficient.

$$\hat{Y} = \text{MLP}(\text{READOUT}(H)) \tag{4}$$

A "readout" layer is added before the MLP to aggregate the node representations H computed by the GNN along the node dimension (Equation 4).

$$\hat{Y} = \text{MLP}(\text{top-rank}(H, Z, n)) \tag{5}$$

$$\hat{Y} = \text{MLP}(\text{READOUT}(H \odot Z)) \tag{6}$$

Readout operations include the conventional aggregators SUM, MEAN and MAX, but can also be more complex like attention based pooling [41], which pools neighbors selectively using separately computed attention scores $Z = \text{GNN}(\text{A}, \text{X})$. In Equation 5 the top n node representations are selected to be passed to the MLP. In Equation 6 nodes are soft-selected based on the attention scores.

2.3.2 Metrics

$$\begin{aligned} \text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}} \\ \text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}} \\ \text{F1-Score} &= \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \end{aligned} \tag{7}$$

$$\text{Balanced Accuracy} = \frac{1}{2} \cdot \left(\frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{TN} + \text{FP}} \right) \tag{8}$$

The F1-Score (Equation 7) is a commonly employed metric for the binary classification scenario with imbalanced classes. However in cases where the number of correctly classified negatives (TN) is as relevant as the number of correctly classified positives (TP), balanced accuracy can be a more appropriate performance measure. Balanced accuracy reduces to the standard accuracy in case of a balanced dataset. A perfect classifier would attain 100% balanced accuracy, a random classifier 50%.

For illustration, we examine the task of vulnerability detection on the DiverseVul dataset utilized in this paper [5]. The dataset is imbalanced since only approximately 6% of the samples belong to the positive class. A classifier classifying all samples as positive reaches an F1-Score of 11% outperforming 9 out of 11 initial models on the task of detecting vulnerabilities in "unseen projects" ([5] Table 5). The outperformed models include CodeT5 [42] and GPT-2 [43] variations. The equivalent balanced accuracy amounts to exactly 50%. It can be concluded that the naive classifiers F1-Score depends on the class-ratio of the dataset, opposed to the balanced accuracy, which is independent of the class-ratio. As illustrated, the F1-Score can communicate a false sense of performance.

2.4 Graph Structure Learning

In the task of GSL the premise is that existing graph structures in a dataset are not optimal for learning-based optimization of downstream tasks using GNNs. Often, real-world datasets are noisy, incomplete or simply do not model structures beneficial for certain tasks.

$$\min_{\hat{A}} \mathbb{E}_{\hat{A} \sim G} \mathcal{L}\big(\mathrm{Y}, F_\theta\big(\mathrm{X}, \mathrm{A}, \hat{A}\big)\big) \tag{9}$$

Concretely, GSL aims at learning the optimal graph structure $\hat{A}$ in conjunction or separately to the downstream task, for which the negative performance is measured by $\mathcal{L}$. Different approaches for finding the optimal $\hat{A}$ exist [44]:

Metric-based GSL employs learned node embeddings and derives the adjacency matrix $\hat{A}$ by computing the pairwise similarity between learned node representations $\widetilde{H}_{i,\cdot}$ with the help of a metric function. Conceptually, such methods are closely related to attention-based mechanisms [45] such as the Graph Attention Network [46] or non-graph related architectures such as the Transformer [29]. However, different to the Graph Attention Network, they can consider all node-pair combinations in a graph and different to the Transformer, they enforce a graph-specific prior through regularization terms and the fact that the representations $\widetilde{H}_{i,\cdot}$ are learned on the original graph-structure.

Direct GSL treats the adjacency matrix $\hat{A}$ itself as a learnable parameter and optimizes it directly.

$$\min_{M} - \sum_{c=1}^{C} \mathbb{1}_{[y=c]} \log P_\theta(Y = y \mid A = \mathrm{A} \odot \sigma(M), X = \mathrm{X}) \tag{10}$$

In GNNExplainer [47] the end goal is not a downstream task but finding a suitable graph structure itself, which explains a separate GNN's classification predictions $F_\theta(\mathrm{A}, \mathrm{X})$ on a node, edge or graph level (Equation 10). Here, the mask M is directly optimized and mapped to $[0, 1]^{n \times n}$ by σ, $\odot$ denotes the Hadamard product.

$$\mathcal{L}_{sp} = \alpha \left\| \hat{A} \right\|_0 \tag{11}$$

$$\mathcal{L}_h\left(H, \hat{A}\right) = \frac{1}{2}\beta \sum_{i,j=1}^{N} \hat{A}_{ij}\left(h_i - h_j\right)^2 \tag{12}$$

Regularization terms include sparsity constraints (Equation 11) to reduce the number of edges in the learned adjacency matrix, the L_0 norm is often replaced by the L_1 norm such that optimization becomes tractable. $\mathcal{L}_h$ (Equation 12) regularizes the graph structure by modeling a homophily assumption, meaning that neighboring nodes should have similar representations h [48]. α and β are manually defined hyperparameters.

$$H^l = \lambda \cdot F_\theta^l\left(H^{l-1}, \mathrm{A}\right) + (1-\lambda) \cdot F_\theta^l\left(H^{l-1}, \hat{A}\right) \tag{13}$$

The initial graph structure A can be included as a prior in the computation of node representations H^l for the downstream task with hyperparameter λ (Equation 13) determining the influence of A [48].

As open challenges to GSL, [44] list the learning of heterogeneous, heterophilous and task-agnostic structures.

2.5 Pretraining Methods

In DL, *pretraining* describes the procedure of training a model first on an auxiliary task followed by *finetuning* the model on the actual task. Pretraining often increase the model's performance on the actual task [37], [49]. The combination of *pretraining* and *finetuning* may also be referred to as *transfer learning* [50].

Whereas in supervised machine learning, we are interested in learning $P_\theta(Y|X)$, pretraining learns some $P_\theta(X)$ or $P_\theta(Y_k|X)$ for some different task k. The extend to which pretraining is useful is therefore determined by how relevant the pretraining tasks are for the final supervised task [38]. Concretely, in pretraining, the model must learn representations of input data X which are useful for the later supervision task.

[38] posit that pretraining can both be seen as a form of regularization and as an improvement to the optimization procedure itself. Pretraining solely changes the

random initialization point of model weights to a "prelearned" initialization point at the start of the supervised stage.

From the regularization perspective, pretraining finds an initialization point in the parameter space which reduces the model's dependence on the first training samples seen. The dependence on the first data points stems from the fact that with more training steps, the magnitude of weights in the model increases, rendering the optimization surface increasingly non-convex and making it difficult for the learning procedure to escape the path taken [38]. Essentially, pretraining already narrows down the parameter location, such that during supervision, the dependence on the first samples is reduced. The regularization perspective implies that pretraining finds the "hills" in optimization space, which after descent increase the model's generalization ability but decrease its' training performance.

From the optimization perspective, pretraining finds specific "basins of attraction" [38] which increase the models training performance and thereby its' generalization ability as well.

[38] find evidence supporting both views. On the one hand pretraining decreases performance for small models while increasing larger models performance. As small models are not expressive enough in the first place, much like L_1 and L_2 regularization, pretraining diminishes performance. On the other hand, they find that even when the training distribution converges against the real distribution, by becoming increasingly large, the pretrained model's performance is higher than without pretraining. If pretraining could be described as a regularizer alone, it would hurt performance as the introduced bias would become counterproductive on increasingly large datasets. Contrary, this finding supports the optimization theory that pretraining finds regions in the parameter space which yield better performance overall.

2.5.1 Multitask Pretraining

Instead of pretraining a model on one task, one can employ multiple pretraining tasks with the idea that the model can find initial supervision data representations which incorporate information required for all pretraining tasks. These robust representations could lead to better performance, as the model can "choose" the learned features important for the following supervision task. The main concern centers around how one can jointly learn these pretraining tasks.

In this paper we experiment with four ideas:

1. Given three tasks A, B and C, the straightforward way to learn the tasks is to learn the tasks sequentially in a *blocked* fashion. The drawback of this idea is the loss of information from the earlier pretraining tasks [51].

2. Another way would be to formulate a new loss function, summing all participating losses: $\mathcal{L} = \mathcal{L}_A + \mathcal{L}_B + \mathcal{L}_C$.

   However, if the pretraining tasks have different loss formulations, such as the Negative Log Likelihood (NLL) for classification and the Mean Squared Error (MSE) of the regression task, the simple addition is not intuitively justifiable [52]. The reason is that both losses operate on different scales of magnitude and therefore would contribute unevenly to the gradient descent procedure.

3. One possibility to alleviate this problem is to learn tasks in an *interleaved* fashion, repeatedly changing the tasks, and resetting any momentum terms in the optimizer after n minibatch samples [51].

4. $$\mathcal{L}(\theta, \sigma_r, \sigma_c) = \frac{1}{2\sigma_r^2}\mathcal{L}_r(\theta) + \frac{1}{\sigma_c^2}\mathcal{L}_c(\theta) + \log \sigma_r + \log \sigma_c \tag{14}$$

   Along with the mentioned methods, we experiment with the approach of [52] (Equation 14). The equation combines the loss of a regression and classification task. The loss terms are derived from probabilistic formulations of the MSE as a Gaussian NLL and the classification likelihood as NLL with a temperatured softmax function. $\mathcal{L}_r$ represents the MSE for the regression task and $\mathcal{L}_c$ the negative logarithm of the softmax for the classification task. In the loss formulation, σ_r and σ_c are learned alongside θ. They model the variance in the respective pretraining tasks, which is determined by both the scale of the task loss and the difficulty of the task. When the difficulty of task $\mathcal{L}_r$ is high, σ_r will increase, to decrease $\mathcal{L}_r$'s influence on the total loss. At the same time $\log \sigma_r$ will act as a regularizer, penalizing large σ_r. We employ the modified version of [53] where the regularization terms become $\log 1 + \sigma^2$ to ensure the loss formulation can not exploit simple tasks, whereby $\log \sigma$ could become negative.

The loss formulation (Equation 14) allows the addition of arbitrarily many regression and classification tasks.

2.5.2 Pretraining on Graphs

GNNs can benefit from pretraining when training data is scarce [49]. The graph structure information naturally permits the formulation of un- or semi-supervised pretraining tasks. In general, *transfer learning* for GNNs can be modeled as follows, similar to [37]:

1. First, we select a GNN architecture, a set of pretraining tasks $T = \{A, B, C, ...\}$ as well as some pretraining schedule S which specifies the ordering and frequency of each task during pretraining, as well as the loss strategy used (In case of multitask pretraining). Further, for each task a *task head*, often a shallow MLP with correct dimensionality and depending on the task, a READOUT-layer, is specified: $H = \{\text{MLP}_A, \text{MLP}_B, \text{MLP}_C, ...\}$. Also, for each task, a loss function must be selected: $L = \{\mathcal{L}_A, \mathcal{L}_B, \mathcal{L}_C, ...\}$.
2. During pretraining for each task t, we compute $H^* = \text{GNN}(\text{A}, \text{X})$ and followingly $\hat{Y}_t = \text{MLP}_t(H^*)$ as well as $\ell_t = \mathcal{L}_t\left(\hat{Y}\right)$ in accordance with S. Then the GNN and task heads can be optimized with a stochastic gradient descent procedure [54]. Some synergies may arise in case of multitask pretraining as task head computations of different task can depend on the same H^*. The pretraining is terminated when a number of pre-defined epochs or another stopping-criterion is satisfied.
3. Finally, the task heads are discarded and the GNN is finetuned for the supervision task with a new task head.

Because Neural Networks (NNs) can be viewed as computing increasingly abstract representations of input X [38], this *transfer learning* approach encodes task-specific information in the MLP layers while the GNN learns to output more general features H^*.

GNN-specific pretraining tasks include link-prediction, feature masking [37] or graph contrastive learning [49]. In link-prediction the task is to classify if two nodes are connected in the real graph. When the size of the adjacency matrix $\text{A}^{n \times n}$ is sufficiently small, for example approximately $\text{n} < 3000$ on a NVIDIA T4 GPU, graph convolution and pretraining methods can be conducted in a full-graph fashion.

$$\hat{Y} = \sigma(H \cdot H^T) \tag{15}$$

$$\begin{aligned} H &= \mathrm{GNN}(\mathrm{X}, \mathrm{A} \odot (1 - \mathrm{M})) \\ \ell_p &= \mathcal{L}\big(\mathrm{Y} \odot \mathrm{M}, \hat{Y} \odot \mathrm{M}\big) \end{aligned} \tag{16}$$

In a full-graph fashion the link-prediction probabilities can be computed with Equation 15 where σ squashes the dot products into the range (0,1). As the GNN would otherwise have no reference for computing the representations H, only a certain amount of edges in $\mathrm{A}^{n \times n}$ is selected for supervision with the binary mask $\mathrm{M}^{n \times n}$ (Equation 16). Link prediction employs a contrastive loss with negative examples [24] to ensure representations of disconnected nodes are different. Here it is ensured that one node per edge is part of an existing edge in the positive examples to create a more difficult supervision objective.

For feature masking, we similarily apply a mask $(1 - \mathrm{M}^{n \times d})$ to the feature matrix $\mathrm{X}^{n \times d}$ masking nodes n with probability p and feature dimensions d with probability q to try to predict missing features. The features could be categorical or numerical features, requiring a classification or regression objective respectively.

Graph contrastive learning [49] leverages augmentations of the graph such as edge masking or feature masking and similar to link-prediction tries to maximize the agreement of graph level representations $\vec{y}$ and $\vec{y}_p$ while minimizing the agreement of $\vec{y}$ and $\vec{y}_n$, where negative examples are augmentations of a different graph. Graph contrastive learning can only be effective in the case when small augmentations do not alter the semantic meaning of the graph drastically, and is therefore unsuitable in certain tasks. The addition or deletion of connections in molecule graphs might drastically alter their chemical properties [37] and in code vulnerability detection single edits might render a program vulnerable. For these tasks, the former pretraining methods can find application.

3 Related Work

This chapter first introduces relevant work from the domain of DL-based automatic vulnerability detection. Later, also two GSL architectures are discussed which are experimented with in comparison to the conventional architectures.

3.1 Deep Learning-based approaches to Automatic Vulnerability Detection

Data for training training is collected through variations of the following procedure: The CVE [14] database deems as label source, since it collects vulnerabilities in source code reported by developers and companies. For OSS the vulnerability-fixing commits linked to the CVE are available. The authors can now declare all functions part of the vulnerability-fixing commit as benign and their counterparts in the previous commit as vulnerable. The label quality is diminished by factors such as that not every fixing-commit is able to fix the vulnerability and not every file or function changed in such a commit is relevant to the vulnerability.

In general DL-based approaches to Automatic Vulnerability Detection can be divided into token- and graph-based approaches [23]:

Graph-based approaches such as Devign [22] leverage the CPG of program code. The Devign architecture consists of a Gated Graph Neural Network (GGNN) [55] which considers a representation of the CPG with one directed adjacency matrix A_r for each edge type r.

$$\begin{aligned} a_r^{t-1} &= \mathrm{A}_r^T (W_r H^{t-1} + b) \\ H^t &= \mathrm{GRU}(H^{t-1}, \sum_r a_r^{t-1}) \end{aligned} \tag{17}$$

Concretely, the compute the state a_r separately for each edge type, and aggregate the states before computing the next timestep's nodes hidden states H with a Gated Recurrent Unit [45]. $\hat{y}$ is computed with a a custom architecture incorporating a CNN with 1-d convolution to select nodes and features important for the task.

The authors of REVEAL [23] apply a GGNN in a similar fashion. The node features X are composed of a categorical vector for the vertex type in the CPG, for example "ArithmeticExpression" or "CallStatement" and, a as in Devign, a Word2Vec [56] embedding of the respective node, learned on the same dataset. The authors separate the task of learning code representations from the vulnerability classification task: In a first stage, node representations are learned with the GGNN, classifying the node labels. In the second stage, a MLP head classifies the graphs into vulnerable or non-vulnerable based on the already learned node embeddings of each graph. They leverage a cross entropy loss to learn the true label and furthermore add a contrastive loss with weighting α, to encourage representations of the same class to be similar and of different classes to be dissimilar. Most importantly the authors make some key observations regarding the challenges for DL-based vulnerability detection.

They criticize previous approaches, because of the lack of out-of-training-distribution evaluation and curate a new dataset based on the two open source projects Linux Debian Kernel and Chromium. Evaluating models on the Reaveal dataset which were trained on the previous datasets, they find significant performance degradation by on average 73% for the F1-Score, compared to the models' performance on their own datasets. For example, a drop from 73.26% to 16.68% F1 for the Devign model.

Manually inspecting predictions of models, they state that many predictions are grounded in irrelevant features. Additionally, the note that GNNs learn more relevant features than LLMs, because they are able to use the CPG. To address the data imbalance, they propose to oversample the minority vulnerability class. Further they observe that the dataset curation method induces duplicate sample issues which degrades the dataset quality.

Opposed to the present work, the authors fail to establish a naive baseline and correspondingly conduct no examinations to relativize their approach. Further, the limited dataset size restrains the authors from evaluating real-world performance quantitatively in greater detail.

Token-based approaches leverage the text representations of source code to detect vulnerabilities. [57] employ Recurrent Neural Networks [31], CNNs and Decision Tree approaches. To increase the retrieval ability, to be employed on larger program code, [58] use a bidirectional Long Short Term Memory Network [59] only on a subset of the code, which they compose as "code gadgets". Focusing on code vulnerabilities

related to function calls, they extract the relevant information using data flow and control flow analysis tools.

[5] conduct large scale vulnerability detection experiments testing LLMs of the model families RoBERTa [60], GPT-2 [43] and T5 [42]. They create the DiverseVul vulnerability detection dataset which is 60% larger than the previously largest C++ and C dataset. To enlarge the training set they add multiple previous datasets and conclude that pretraining strategies can improve the LLMs' performance when the pretraining tasks are code specific and not only natural language ones. Comparing with REVEAL, they suggest that LLMs might be superior for vulnerability detection, especially when trained on large corpora. They further investigate the models' performance, when tested on samples from an isolated set of projects, and find similar performance drops as the REVEAL authors. Finally, they find that class weighting, as an alternative to minority-class oversampling can help the models' performance. We find several points for discussion in their approach and consequently the conclusions drawn, which we elaborate on in Chapter 4.

3.2 Respecting the Graph Structure

Previous CPG-based models for vulnerability detection only rely on the CPG of the given program. In the current work we investigate whether loosening this constraint provides benefits. The hypothesis would be that the provided CPG structure is not optimal for the task of vulnerability detection. Therefore, inferring different connections between code-nodes could create different neighborhood structures and thereby better inform the node representations learned by the GNNs for the task of vulnerability detection.

The the metric-based GSL approach of Graph Structure Learning Model for Open-World Generalization (GraphGLOW) [61] applies the "Iterative Deep Graph Learning" framework [48] in an inductive setting of node-classification in social networks.

In general, the architecture consists of two components, a GCN [8] and MLP head f_w and a structure learner g_θ, which learns to find an optimal adjacency matrix $A^* = g_{\theta^*}(\mathrm{A}, \mathrm{X})$. The training of GraphGLOW can be summarized as a nested optimization problem:

$$\theta^* = \arg\min_{\theta} \min_{w_1,..w_M} \sum_{m=1}^{M} \mathcal{L}\Big(f_{w_m}(g_\theta(\mathrm{A_m}, \mathrm{X_m}), \mathrm{X_m}), \mathrm{Y_m}\Big) \tag{18}$$

The approach is applied in an inductive setting, meaning training and testing are conducted on different graphs. During training, we aim to find the θ^* which minimizes the loss $\mathcal{L}\Big(f_{w_m^*}(A_m^*, \mathrm{X}_m), \mathrm{Y}_m\Big)$ on each training graph m. In detail, GraphGLOW trains the structure learner g_θ on a number of social graphs with $\mathcal{L}$ a node classification loss. Between the M graphs during training, one g_θ is learned while f_{w_m} is relearned for every m. During testing or inference on a new social graph, $A^* = g_{\theta^*}(\mathrm{A}, \mathrm{X})$ is computed with the found θ^* and a final f_w is learned based on A^*.

$$\alpha_{uv} = \delta\left(\frac{1}{K}\sum_{k=1}^{K} \mathrm{SIM}(w_k^1 \odot h_u, w_k^2 \odot h_v)\right) \tag{19}$$

Equation 19 depicts the node centric view of g_θ, determining the connection strength, the entry of the adjacency matrix A^* between nodes u and v using their learned representations h. It learns K heads with parameters w_k, SIM denotes the cosine similarity and δ converts the input into values within $[0, 1]$. During one forward pass g_θ and the GCN are applied in an iterative fashion: $A^t = g_\theta(A^{t-1}, H^t)$ and $H^t = \mathrm{GCN}(A^{t-1}, H^{t-1})$ until A^t converges measured by the frobenius norm of $A^t - A^{t-1}$, followed by computing $\hat{Y} = \mathrm{MLP}(H^t)$. The method further leverages graph specific regularization such as in Equation 12, the prior Equation 13 as well as a regularization term which penalizes certainty in g_θ.

Since GraphGLOW is applied on node classification in social networks, the regularization terms reflect an assumption of homophily as people often converse with like-minded individuals. Hierarchical Graph Pooling with Structure Learning (HGP-SL) [62] is applied on graph level and models the prior belief that redundant information in form of similar node representations in the graph can be pooled away. Therefore, it maximizes heterophily in the graph. In each layer HGP-SL pools the node representations H^l of the layer by some ratio ρ, only keeping the top $n \cdot \rho$ nodes which are most different to their neighbors.

$$\vec{p}^{l} = \| \left(1 - \left(D^{l}\right)^{-1} - A^{l}\right) H^{l} \|_{1} \tag{20}$$

After pooling, some of the nodes kept might not have any edges to other nodes, therefore a new graph structure is learned using a metric-based approach. The node-information score $\vec{p}^{l}$, determines how different nodes are to their neighbors and is computed with Equation 20. $\|...\|_1$ denotes the row-wise l_1 norm. $\vec{p}^{l}$ is simply the l_1 norm of the difference between h_i and the average of the neighbors' representation. HGP-SL is applied to molecule datasets and achieves state of the art results on the PROTEINS [63] dataset.

The RGCN [7] improves the GCN [8] to work on heterogeneous graphs which contain one adjacency matrix for each edge type r, $\mathrm{A_{r1}}, ..., \mathrm{A_{rn}}$. Exemplary, the idea to learn one weight W_r for each edge type and aggregate the neighbors' aggregated representations specified by each A_r which are computed in a similar fashion to the GCN. In heterogeneous graphs, some edge types might not have enough edges to learn the corresponding W_r well.

$$W_r = \sum_{b=1}^{B} a_{rb} V_b \tag{21}$$

Therefore, the authors propose to decompose all W_r into a series of shared basis matrices V_b (Equation 21), learning both V_b as well as the coefficients a_{rb} which depend on r. By the decomposition, information could be shared between edge types to learn better W_r.

4 Experiments

In the following, the experiments are presented which are conducted with the focus on the four dimensions *data*, *architecture*, *training* and *evaluation*. First, we revisit the motivation from Chapter 1, then examine the DiverseVul dataset [5] and proceed with each experiment, first stating the motivation, followed by the empirical results and interpretation thereof.

4.1 Experiment 1: The Importance of Project-Based Train-Test Separation

The promise of DL-based automatic vulnerability detection is to increase efficiency and effectiveness compared to conventional static and dynamic approaches. To support progress in the field, we aim to *evaluate* the models accurately. As described in the Chapter 1, we observe that previous work [5], [22], [6] evaluate DL-based approaches to automatic vulnerability detection mostly in the setting where both the training and test data stem from the same distribution, meaning from the same projects (the "same" setting). Upon evaluation on out-of-distribution samples, such as other datasets or unseen projects [5], [6] find unexpectedly large performance deterioration. Nevertheless, they proceed to draw conclusions based on the empirical results from the "same" setting.

Model	Training Set	F1	Precision	Recall	(Balanced Accuracy)
NatGen	CVEFixes	0.1183	0.3617	0.0707	(0.5300)
	Previous	0.4694	0.5181	0.4292	(0.6974)
	Prev. & DiverseVul	0.4715	0.5181	0.4325	(0.6989)

Table 1: Test results on Previous & DiverseVul from the Diversevul paper [5]. CVEFixes [64] is a dataset and "Previous" is a combination of datasets.

For example, [5] claim that increasing the dataset size increases model performance. Table 1 shows their results for the best LLM. It can be observed that in the first row where the model is not evaluated in the "same" setting, since training and test sets are distinct, the performance is near random: We calculated a balanced accuracy of 53% with the false-positive rate provided by the authors. It can not be ruled out that

the increase of performance by 14% in balanced accuracy in the second row is not simply due to a larger training set, but due to the fact that training and testing sets are not disjunct. Likewise, in the third row, if a larger training set were to increase performance, addition of roughly 50% more data only corresponds to an increase in performance by 0.015% balanced accuracy. Notably, about half of the samples of "Previous" and DiverseVul overlap, which would be in line with our hypothesis that the increase in performance is due to insufficient train-test separation (Since the model has seen many similar samples in the second and third row, explaining the almost-equal performance).

A small experiment assures our hypothesis, comparing the results of the GIN [9] network, in a project level split on DiverseVul [5] the balanced accuracy decreases by 15% from 80% to 65% (Table 4).

Determining the factors responsible for the diminished performance is left for further investigation. However, they correspond to some form of train-test leakage, as the model learns non-generalizable factors which are only relevant within one project. Consequently, in the following we only examine results on data which is split on project level, unless stated otherwise.

4.2 The DiverseVul Dataset and Splitting Approach

We utilize DiverseVul dataset [5] as the data source to train the DL-based models because of its size which allows for meaningful project level splits. The authors also manually check the vulnerable label quality: Only 25% of samples labeled as vulnerable which were manually checked are found to be vulnerable in the next largest dataset BigVul [65] with a size of 260,000 functions. DiverseVul has the highest rate of all compared datasets with 60%. The dataset contains 330,000 C and C++ functions of which 19,000 are classified as vulnerable. 85% of all functions are mapped to one or multiple of 150 CWE classes, the types of vulnerabilities. The data corresponds to 7500 commits from 800 different projects. To obtain the label for each function, the authors leverage websites which list vulnerability fixing commits similar to the approach mentioned in Chapter 3.1. Functions which are part of files changed in vulnerability fixing commits are labeled as benign, the same functions from the previous commit are labeled as vulnerable. To increase the dataset size, the authors

furthermore label the functions in all C and C++ files as benign, which are part of neither of both commits.

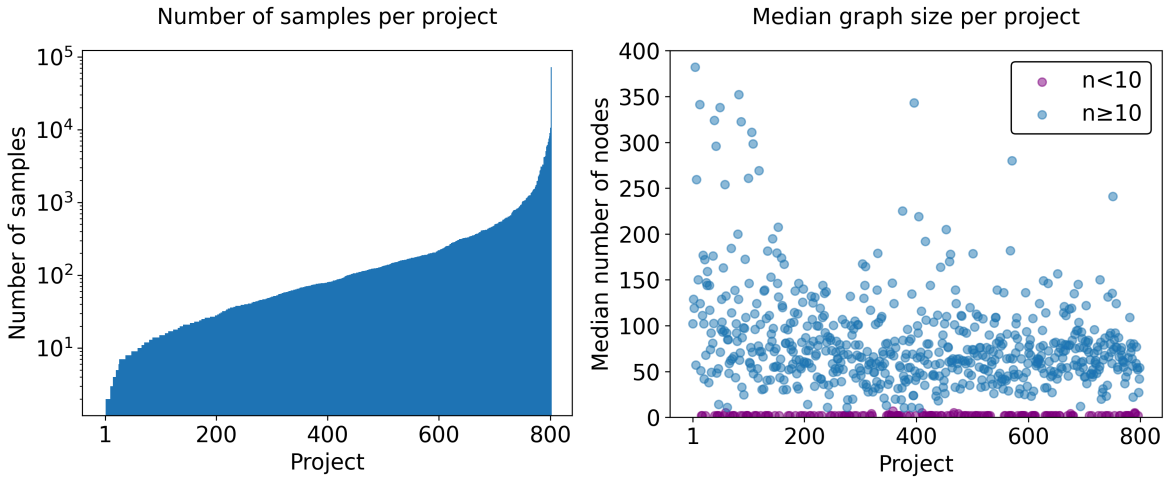


Figure 2: Left: Function samples per project on logarithmic scale, the largest project is Linux, Right: Median size of the extracted graphs per project, purple samples mostly represent functions for which the CPG-parsing failed.

Figure 2 on the left side depicts the number of functions extracted per project on a logarithmic scale. The project with the most samples is Linux with about 70,000. More than half of the projects have less than 100 samples. The CPGs for each function are extracted with the help of an extraction tool of Fraunhofer-AISEC [20], [21]. The tool contains a "forgiving" parser, which makes it possible to build the CPG even for incomplete or semantically incorrect source code. Nevertheless, the tool fails in some instances: We filter out all code graphs which contain less than 10 nodes. Figure 2 shows how the purple data points fall "out of distribution". The tool returns a category for each node, as described in Chapter 3.1. Further, each node's 100-dimensional Word2Vec [56] embedding is added, similar to [22], which was trained on data overlapping with the "Previous" datasets in [5].

We split the dataset into 6 folds on project level and try to ensure similar project sizes, benign and vulnerable ratios as well as CWE ratios. In detail, we randomly apply scikit-learns GroupKFold [66] 1000 times, selecting the variation with the smallest distance between the largest and smallest folds' sizes. Because of its size, the Linux project constitutes one whole fold. One of the other folds is randomly selected as test fold. Additionally to the functions on which the CPG parser failed, we remove

about 3000 graphs with $n > 1000$ which allows for faster experimentation with the GNN architecture. In total, the cross validation set contains now 205,000 graphs with 11,300 vulnerable samples and the separate test set 43,500 samples with 2900 vulnerable samples. This amounts to a vulnerable sample ratio of 5.6% and 6.4% respectively.

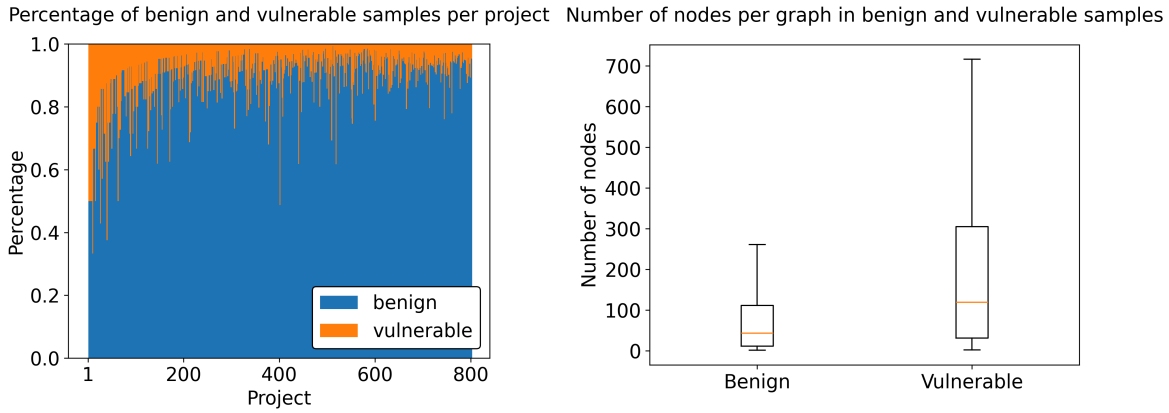


Figure 3: Left: Smaller projects contain relatively more vulnerable samples, Right: The number of nodes in the CPG is indicative of the vulnerability label.

The dataset is inspected further. Figure 3 on the left shows that smaller projects generally have larger vulnerability ratios. This could simply be related to the fact, that larger projects contain a larger codebase, thus the number of additionally added benign samples could be higher.

More importantly, Figure 3 on the right depicts that benign and vulnerable samples already have disparate distribution of number of nodes in the graph. The number of nodes in a graph should be proportionally related to file length. We believe it is also related to the fact that additionally to the benign and vulnerable pairs, all other unmodified functions are added to the dataset and declared as benign. Generally, larger functions are both simply more likely and because of increased complexity more prone to contain vulnerabilities, which leaves the simpler functions unmodified. These functions such as simple "setters" and "getters" are then added as benign samples. Notably, the original authors [5] do not examine the different models' performance respecting such an observation.

Further, it is examined whether graph sizes differ between small, medium and large projects, however the distributions appear visually similar.

The findings related to the dataset prompt multiple courses of action:

- The even visually distinct node count distributions of benign and vulnerable samples suggest the use of the "num_nodes" baseline: The naive classifier classifies samples as vulnerable if the number of nodes exceeds a threshold t. t is chosen to maximize the balanced accuracy on the validation folds. Plotting the balanced accuracy against the number of nodes, the optimal threshold is found to be a smooth curve with the peak at $t = 102$ which is evident from Figure 3. The baseline serves as point of comparison in the subsequent experiments. The performance of the baseline is reported in the next experiment.
- The imbalanced nature of the dataset prompts the use of balanced accuracy as primary metric. As elaborated in Section 2.3.2, balanced accuracy weighs positive and negative predictions equally. The F1-score is biased towards weighing correct positive predictions higher. In the case of the current dataset, a naive one-class predictor achieves an F1 of 11% conveys a false sense of performance, whereas the balanced accuracy of 50% communicates the random performance. To preserve comparability to previous works, the F1 score is reported as well.

4.3 Experiment 2: num_nodes Baseline

Here the performance of the num_nodes baseline is reported. The threshold t is determined on the cross-validation folds of the training set. The final performance on the test set, consisting of more than 100 projects, is only determined at the end together with all other models, to not influence the other experiments. But we report them earlier to establish it as reference.

Model	Balanced Accuracy	F1	Precision	Recall
num_nodes	0.6439	0.1950	0.1154	0.6284

Table 2: Test performance of num_nodes.

Surprisingly the baseline outperforms the best LLM, Code T5 Small [42] in the setting of evaluation on unseen projects. The DiverseVul authors report an F1-score of 17.21% [5] and we calculate a balanced accuracy of 57.12%, evaluated on 95 randomly chosen projects. Notably, in their setting the train and test set, correspond to

DiverseVul as well as previous datasets, such that performance can only be compared with lower confidence. However, as we infer from the context the Code T5 Small model is pretrained on C and C++ code as well as code specific tasks and finally finetuned on vulnerability detection. The experimental outcome raises the question, how well the models generalize at all in this setting.

Followingly, it can be confirmed, that LLMs are not necessarily the ideal architecture. For our experiments, we focus on GNNs with small parameter count in the single-digit millions as architecture, which allows for fast experimentation compared to LLM-based methods with over 100 million parameters [5]. GNNs-based methods have shown better generalization and robustness to variability in code-style and formatting compared to LLM-based approaches [67], [22], since they can access the CPG. Thus, by employing these architectures we can diminish the susceptibility to spurious features as unwanted factor of variation in the dataset to increase generalization.

4.4 Experiment 3: Graph Representation

Under the new setting of training data which is split on project level into 5 folds, we aim to evaluate if the information of edge-direction provided by the directed edges of the CPG is useful for vulnerability classification. In general CPG edge directions contain semantic meaning. In the control flow edges they indicate the order of operations and the data flow they indicate which node is a call node and which one a receiving node. The Devign model [22] considers the edges in the forward direction with the adjacency matrix A, however it is not mentioned whether the backward direction A^{T} is considered as well. From a node-level perspective the forward direction would inform a code node's representation, which other computational nodes it depends on in a backward perspective it would inform the node's representation which operations it gives rise to. [67] consider the adjacency matrix in an undirected fashion, meaning $\hat{A} = \min(\mathrm{A} + \mathrm{A}^{\mathrm{T}}, 1)$, which would help the model consider both views, however the information of direction is lost in this setting.

In a later experiment the effect of considering both directions separately and training on the CPG as heterogeneous graph with multiple adjacency matrices, one for each edge type, is examined. In the current setting, only a single matrix A summarizes all matrices of the heterogeneous CPG: $\hat{A} = \min(\mathrm{A}_{\mathrm{r1}} + \mathrm{A}_{\mathrm{r2}} + ... + \mathrm{A}_{\mathrm{rn}}, 1)$.

In addition we test the effect of including node degree and node triangle counts to X. The counts are added per adjacency direction and per edge type separately. These features could distinguish nodes with large involvement in the computational graph or might contain relevant patterns for the vulnerability detection.

Model & Dataset	Balanced Acc.	F1	Precision	Recall
GCN & dir. deg.	0.6407	0.1673	0.0976	0.6350 ▲
GCN & dir.	0.6266	0.1599	0.0982	0.5612
GCN & undir. deg.	0.6434 ◆	0.1723	0.1009	0.6127 ▲
GCN & undir.	0.6365	0.1862	0.1148	0.5051
GIN & dir. deg.	0.6173 ●	0.1643	0.0980	0.5324 ▲
GIN & dir.	0.6278	0.1733 ●	0.1045	0.5245
GIN & undir. deg.	0.6427 ● ◆	0.1737	0.1021	0.5998 ▲
GIN & undir.	0.6393	0.1817 ●	0.1099	0.5378

Table 3: Average performance of models trained on 20% of the training data on the five validation folds per dataset variation. Metrics are calculated as average over different model configurations (hidden dimension: [128, 256], dropout: [0, 0.3, 0.5]) (deg.: with degree features, dir.: directed graphs).

The GNN models GCN [8] and GIN [9] are trained of 20% of the data per training split and evaluated on the the full validation split in a 5-fold cross validation. They use a MLP task head with SUM-aggregation and are trained until the validation loss does not decrease for 10 epochs. For this and the following experiments Different model configurations are trained, and the metrics in Table 3 represent the average across these configurations as well as all cross-validation variations. The indicators “dir”, “undir” and “deg” denote whether a directed adjacency matrix is used and whether degree and triangle counts are added to X.

We make several observations (Table 3):

- ▲ In all four cases the addition of degree and triangle features improves recall.
- ● In the case of the GIN model, the addition of the features increases balanced accuracy but decreases F1, when considering the recall increase the model classifies more vulnerable samples correctly in favor for less correctly classified benign samples.

- For both architectures, the use of the additional features with an undirected adjacency matrix achieves the highest balanced accuracy.

Many GNN architectures such as the GCN and the GIN do not consider the inclusion of both A and A^T. One theory explaining the increase in balanced accuracy in Table 3 is that the undirected graphs allow the GNNs to pass neighbor-messages more efficiently than if an arbitrary direction of A was chosen. As we find later, a more likely reason is that the setup allows the models to learn naive features which describe the complexity of the graph and correlate with the number of nodes, similar to what the baseline num_nodes computes.

4.5 Experiment 4: Architecture

Several GNN architectures are implemented and evaluated in a cross-validation fashion, before finally being tested on the test set. The project-level split folds of the DiverseVul [5] are used. Before the final cross-validation, for each model, hyperparameter search attempts to find the hyperparameters achieving highest balanced accuracy. For this purpose, we cross-validate on all validation folds, training only on 20% of each fold to enable a broader hyperparameter search. Searched hyperparameters are provided in Appendix 1. The final cross-validation mirrors the variance of the performance between different projects, whereas the test set depicts the vulnerability detection performance on unseen data.

Some details of training include the class weighting according to [5] and the calibrating of the bias of the task head such that it initially reflects the class imbalance of 5.4% vulnerable samples in the predicted probability, to speed up the training. The Adam optimizer [68] with learning rate 0.001 and without weight decay is employed. For measuring the training performance, all models are trained on the same random split of the training data. Training is halted after 15 epochs of stagnating loss.

The performance of the baseline num_nodes, GCN, GIN, RGCN variants, REVEAL [6], and the GSL approaches GraphGLOW and HGP-SL are compared.

- The GCN and GIN represent baselines, because of their simple architecture.
- For the RGCN, we employ the basis decomposition (Equation 21) in the following way: In the heterogeneous CPG, each directed edge type r has a "forward" and "backward" representation A_{r_f} and $A_{r_b} = A_{r_f}^T$. All $r_{1f}, ..., r_{nf}, r_{1b}, ..., r_{nb}$ share a set of basis matrices $\mathcal{V}_{all}$, r_{xf} and r_{xb} share a set of basis matrices $\mathcal{V}_x$. Thus, the model

utilizes both the edge type information as well as edge direction information which was not accounted for in Experiment 3, Section 4.4. Two options with different aggregation methods are explored, RGCN-SUM and RGCN-MEAN.

- REVEAL [6] is trained without the contrastive loss component from the original paper in an end-to-end fashion.
- The application of GraphGLOW and HGP-SL investigates whether the methods can learn more optimal graph structures as elaborated on in Chapter 3.2. To restate, GraphGLOW could model homogeneous relationships while HGP-SL would summarize the graph, only keeping most distinct node representations. For GraphGLOW, the correct transition from node-classification in the original paper to graph-classification in the vulnerability detection task is not clear immediately. We experiment with keeping the structure learner g_θ constant while resetting the task head f_w every n minibatches without added benefit. Followingly, we resort to not resetting f_w.

Model	Balanced Acc.	F1	Precision	Recall
GCN	0.6475±0.017	0.1680±0.021	0.0963±0.013	0.6647±0.064
GIN	0.6467±0.015	0.1719±0.025	0.0996±0.015	0.6315±0.077
RGCN-mean	0.6478±0.019	0.1722±0.019	0.1001±0.013	0.6303±0.078
RGCN-sum	0.6411±0.020	0.1901±0.028	0.1178±0.021	0.5077±0.064
REVEAL	0.6365±0.023	0.1901±0.025	0.1186±0.017	0.4850±0.069
HGP-SL	0.6456±0.020	0.1722±0.019	0.0998±0.011	0.6322±0.092
GraphGLOW	0.6477±0.016	0.1681±0.018	0.0963±0.011	0.6649±0.079
num_nodes	0.6494±0.018	0.1699±0.027	0.0980±0.018	0.6547±0.056
GIN mixed data	0.7964±0.024	0.2869±0.040	0.1738±0.030	0.8409±0.034

Table 4: Validation performance of models trained on 100% of the respective splits' training data. The mixed-data model is trained and evaluated on non-projectwise split data.

Table 4 depicts the models' validation performance. No model is able to outperform the num_nodes baseline. Inspecting the individual validation splits, the performance among the models' is similar for each validation split, therefore the standard deviation shown is rather an effect of the specific split division. RGCN-sum and REVEAL

seem to have similar performance profiles. Both trade off a higher balanced accuracy in favor for a higher F1 score.

Model	Balanced Acc.	F1	Precision	Recall
GCN	0.6467	0.2011	0.1205	0.6061
GIN	0.6456	0.1992	0.1190	0.6106
RGCN-mean	0.6429	0.1910	0.1119	0.6507
RGCN-sum	0.6345	0.2104	0.1334	0.4976
REVEAL	0.6269	0.2177	0.1452	0.4350
HGP-SL	0.6410	0.1943	0.1153	0.6165
GraphGLOW	0.6441	0.1958	0.1161	0.6239
num_nodes	0.6439	0.1950	0.1154	0.6284

Table 5: Test performance of models.

Table 5 shows the testing performance of models. Again, no model is able to outperform the num_nodes baseline. Neither the information about edge type, nor edge direction allows the RGCN variants to outperform the baseline on the DiverseVul dataset. In parallel, the structure learning of the GSL variants provides no benefit.

These results, and the next experiment regarding model pretraining initiate the more thorough investigation of the factors affecting the outcome that all models achieve around 65% balanced accuracy in the last experiments.

4.6 Experiment 5: Pretraining

Pretraining can improve a DL model's downstream performance as elaborated on in Chapter 2.5 [38]. In the setting of graph classification, specifically molecular property prediction it has been shown to increase the model's performance [37]. In an abstract way, this task is similar to vulnerability detection, since both are graph classification tasks and even small changes in the adjacency matrix can lead to drastically different properties. Crucial to an effective pretraining setup in [37] is the multistage pretraining. The authors combine a first stage of node-level pretraining (attribute masking) with a second stage of graph-level pretraining (predicting domain-specific molecule attributes). In their setting, solely applying the graph-level pretraining has a negative impact on the downstream task. The reasoning is similar to REVEAL [6],

where node-level code features are learned independently from the downstream task, and only then the model is optimized for the graph-classification task. [37] believe, the node-level pretraining assists the GNN in learning relevant node representations, which increases the generalization ability. When the model is only pretrained on graph level, it could overfit on the node level, learning features which only maximize the graph-level pretraining performance but which are not relevant node-level features, which in turn would decrease the generalization ability.

Model & Dataset	Balanced Acc.	F1	Precision	Recall
GCN & deg.	0.8860	0.4022	0.2572	0.9327
GIN & deg.	0.8985	0.4266	0.2759	0.9428

Table 6: Performance of overfitting the training set for three-layer GCN and GIN models.

[37] find more expressive models to benefit the most from pretraining, which is in line with the observations of [38]. In particular, they suggest the GIN architecture. We test and confirm the model's expressivity compared to the GCN (Table 6). Because it has shown equal performance to other models in Experiment 4, and the computational efficiency of the architecture, the GIN is employed to test different pretraining strategies:

First, multiple pretraining tasks are devised. On the node-level, one can formulate link-prediction and feature masking as straightforward tasks. For the features-masking in $\mathbf{X}$, the triangle counts, node degrees and Word2Vec [56] embedding are learned minimizing the MSE, the node category minimizing the NLL.

On the graph-level, more than 85% of the samples in DiverseVul [5] are mapped to one or more CWE. For benign samples the mapping is determined by which CWEs the corresponding vulnerable samples belong to. The information is utilized for multi-label classification. Predicting if a sample is assigned to CWE is treated as single binary classification task, for each CWE. Also, each sample is weighted equally, regardless of how many CWEs it is categorized by. This pretraining task could help the model establish a focus on code patterns which are prevalent in each CWE, presumably beneficial for the final vulnerability detection task.

Since the code graphs used for training are rather small, $n < 1000$, all pretraining is conducted in a full graph fashion, and across a minibatch of 8 graphs. We pretrain and train only on one cross-validation split.

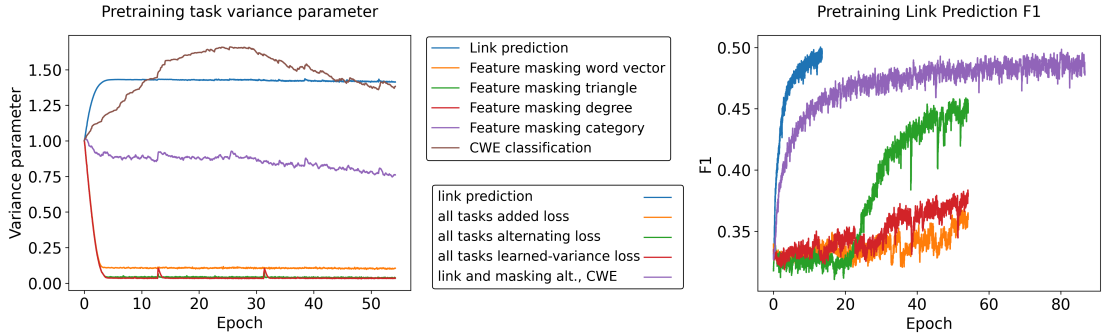


Figure 4: Left: Evolution of the σ parameter during pretraining when learning the "variance" of pretraining tasks, Right: Link prediction validation performance of different pretraining methods in conjunction with the GIN architecture.

To combine the pretraining tasks, three methods are experimented with, as presented in Chapter 2.5: Loss addition, alternating between pretraining tasks and learning a weight σ for each task (Equation 14), which can be interpreted as both learning the scale of the loss and the difficulty of its corresponding task. Figure 4 on the left shows the learned σ for each task during training, when all tasks are jointly learned with Equation 14. The tasks can be divided into two groups: regression tasks, feature masking of the embedding, degree and triangle counts, and classification pretraining tasks, link prediction, node category masking and CWE classification. The figure shows the different learned magnitudes for the classification-based vs regression-based loss. Within each group, the task difficulties can be compared: Predicting embedding features seems to be more difficult than predicting triangle counts, which in turn is more difficult than predicting degree counts. Predicting links and the graphs CWEs is more challenging than the node category prediction. Figure 4 on the right depicts link prediction performance for different pretraining routines. As expected, the pure link prediction pretraining task achieves the highest F1 (blue), followed by the routine which alternates between link prediction and feature masking followed by CWE classification, not shown in this figure. Comparing the “all” pretraining routines which pretrain with link prediction, feature masking and CWE prediction, alternating the loss is most efficient and learning the task variance is slightly better

than simple loss addition. For the "all" pretraining routines, the same patterns are observed when inspecting CWE classification and feature masking performance.

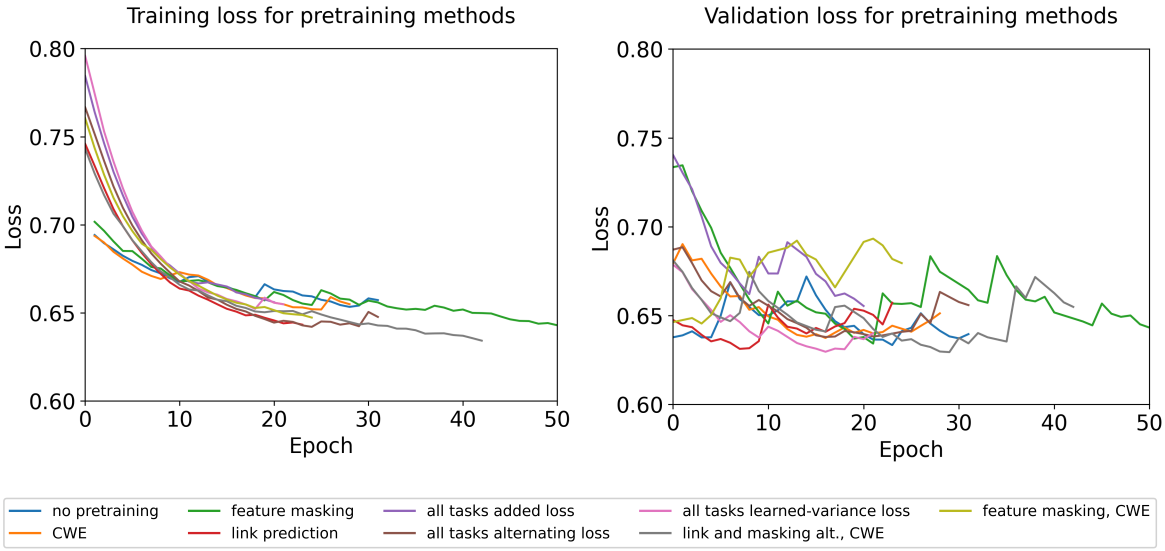


Figure 5: Training and validation loss of different pretraining methods with the GIN architecture. Exponential smoothing is applied for clearer comparability. Pretraining shows no clear benefit compared to no pretraining.

Figure 5 depicts both the training and validation loss in the setting when the pretrained GIN is trained on the final vulnerability detection task. Training is stopped after 10 epochs without validation loss improvement. The left plot illustrates that when pretrained with *"link prediction"*, *"all tasks in an alternating fashion"* or *"link prediction and feature masking in an alternating fashion, followed by CWE classification"*, the training loss achieved is the lowest (red, brown, grey). In the right plot, only *"pretraining with link prediction"*, *"all tasks with learned variance"* and *"alternating link prediction and feature masking followed by CWE classification"* (red, pink, grey) achieves minimally lower validation loss. When evaluated on the test set, models with pretraining achieve no higher balanced accuracy than the original GIN.

In summary, evaluating a broad range of pretraining tasks and pretraining schedules, pretraining on DiverseVul shows no clear benefit. However, in future experiments, link prediction or combining losses while learning σ seems and task alternation followed by graph level pretraining seem to be the most promising direction, compared to

the other pretraining tasks or simple loss addition, because those methods achieved lowest validation loss.

The findings hint at the hypothesis, that pretraining mostly only assists the models in learning how to "count" graph complexity faster, to in fact learn solely, what the num_nodes baseline counts: When inspecting the training loss in Figure 5, almost all pretraining schedules start at a higher loss value than the GIN baseline at around 0.75. But they quickly proceed to pass the GIN baseline in terms of training loss at around epoch 10. This phenomenon seems to be related to the link prediction task, because the training loss of the CWE classifcation and feature masking models progresses at a similar rate to the GIN baseline. Although their training loss is lower, no visibly lower validation loss is achieved, and therefore it is likely that pretraining in the current setting leads to the models overfitting faster. These findings prompt the following experiments.

4.7 Experiment 6: Stratification

Observing that all model architectures as well as the pretrained GIN models, without exception, achieve a balanced accuracy no higher than 65%, leads us to investigate further and establish the num_nodes baseline. Seeing how the num_nodes baseline also achieves a balanced accuracy of 65% (Table 5) , the question arises if the models learn anything besides a representation of the complexity in the graph, which correlates with the number of nodes the num_nodes baseline uses to detect vulnerable samples. For this purpose, the predictions of the models of the architecture and pretraining experiments are re-evaluated under stratification.

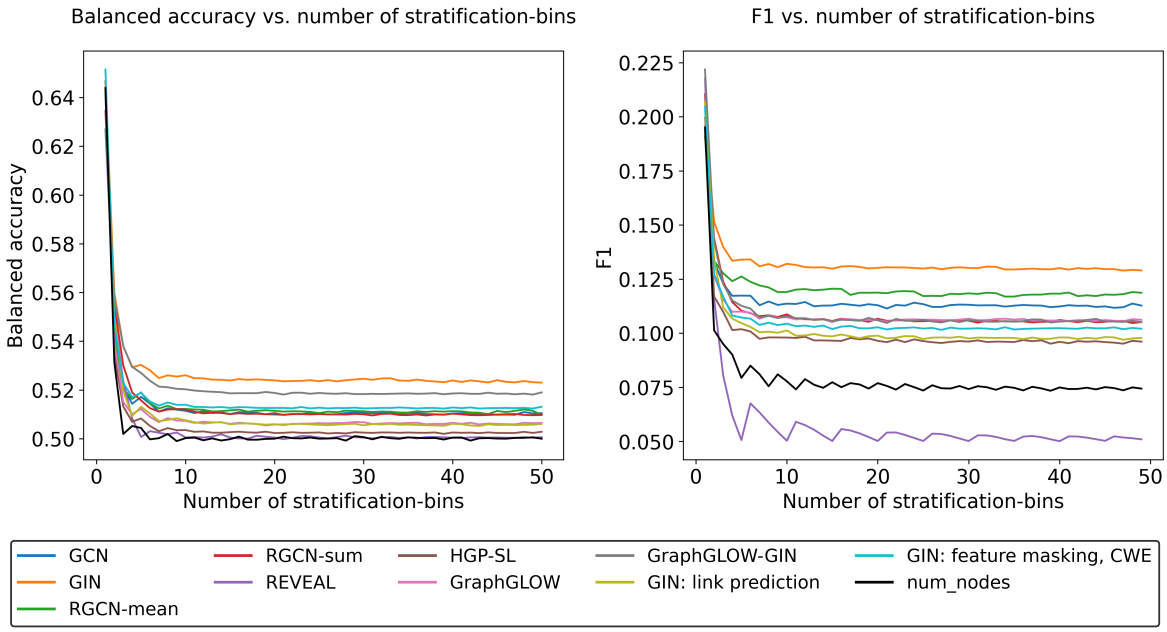


Figure 6: Predictions on the test set are stratified by the number of nodes in the graph with equal-count bins. "num_nodes" represents the baseline performance.

First, results are stratified by the number of nodes in the graph: For each model, all predictions are sorted by the number of nodes the code graph of the underlying sample has. Then they are assigned into equal-count bins. With a larger number of bins, the performance converges against the theoretical setting, in which models have no direct access to the information about the sample's graph size (Figure 6). This is because in each bin, only the performance on graphs of the same size is compared, measuring how well models can distinguish vulnerable samples from benign ones, when both have the same graph size. Subsequently the total performance is determined by averaging across all bins.

Figure 6 depicts both the balanced accuracy as well as the F1 score in this setting. The success of the stratification can be verified, since the balanced accuracy of the num_nodes classificator drops to a random guesser's performance of 50% already at around 3 bins (black). Similarly, it's F1 score drops to a performance worse than the 12% F1 score of a one-class classificator. Considering both plots, the GIN architecture performs best both in terms of balanced accuracy and F1 score (orange). It's performance, although modest, is above random in the balanced accuracy case and better than a one-class classificator. Also a GraphGLOW [61] model was trained (grey), exchanging the GCN [8] with the GIN architecture. However, it's performance

is worse than GIN. Notably the REVEAL [6] architecture, originally designed for vulnerability detection is unable to base predictions on more than on the graph complexity (purple).

Examining the balanced accuracy, GIN's performance drops by 12% points versus in the unstratified setting. Without access to the node count information, the model's performance is only 2.5% points higher than random, while it can simultaneously not be ruled out that the predictions are based on other naive features.

These findings are concerning, since the DL methods synthesize almost no complex information, which a simple baseline would not be able to detect. When comparing the num_nodes performance on unseen projects, 64% balanced accuracy and 20% F1 score, to the best LLMs performance in a similar setting, 57% balanced accuracy and 17% F1 [5], it seems unlikely that LLMs learn significantly more than the GNN-based models under stratified evaluation.

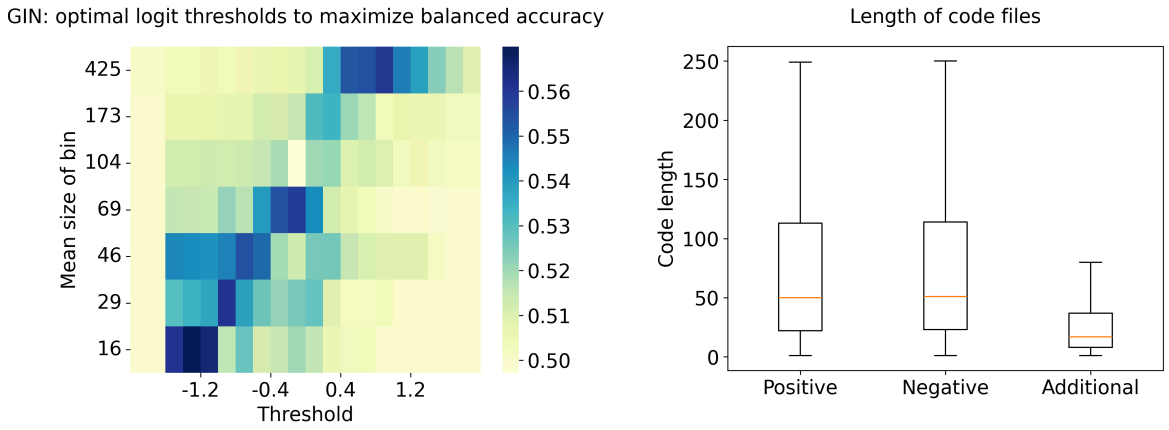


Figure 7: Left: Predictions on the test set are stratified by the number of nodes in the graph with equal-count bins. The optimal threshold increases in a linear pattern with increasing graph size, indicating that the original predictions are biased towards classifying smaller graphs as non-vulnerable. n Right: Code length of positive, negative and additional samples.

Figure 7 on the left confirms the following conclusion. The figure lists the optimal logit thresholds per graph size in a stratified setting with 7 bins for the GINs predictions (darker is better). An almost linear patter shows that with increasing graph size, higher thresholds would be more optimal , instead of the default threshold at 0. This indicates that the model primarily looks at features correlated with graph size,

since the optimal threshold changes drastically between graph sizes. The model has to compromise and predicts such that the optimal threshold is at 0 excatly for the median bin, which also resembles the average graph size.

In conclusion, the models primarily learn to approximate the the code length. The reason, num_nodes performs well, is the data collection strategy in DiverseVul. We confirm our hypothesis from Section 4.2 that the benign and vulnerable pairs in general have larger code length than the 88% functions which are modified in neither commit and are additionally collected in DiverseVul [5]: Based on the function names we are able to approximately identify benign functions which correspond to vulnerable ones, and retrieve 16541 of the 18945 corresponding benign functions. Consequently, we compare file lengths between the pairs and the other additional functions (Figure 7, the right plot). Indeed, the vulnerable and benign pairs have a similar median at 112 and 115 but the additional files a median of 35. A similar data collection of additional functions is present in a number of other works introducing new datasets [6], [65] and becomes a systematic issue in tainting performance measures when not accounted for by an appropriate baseline.

However, this dataset composition is not inherently an issue. Additional negative examples should presumably help the model. Multiple questions can be raised in response to the results, which we investigate:

Do the additional files mislead the model to only learn features based on the code length?

A GIN is trained under the setting where the graph size is explicitly added to the node features X. By initially providing the code length as feature, the model is not forced to focus on deriving it from other features such as degree and triangle counts. The GIN model is unable to achieve better performance and therefore we disregard the question.

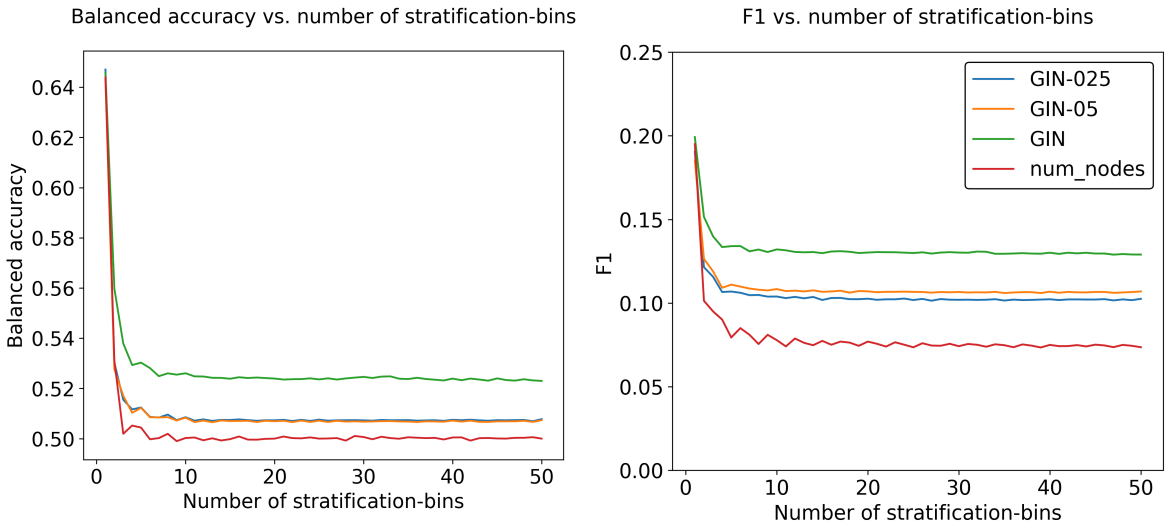


Figure 8: Larger dataset size increases model performance. Predictions on the test set are stratified by the number of nodes in the graph with equal-count bins.

Is it an issue of insufficient training data?

A GIN is trained with 25% and 50% of the training data. Figure 8 compares the performance with the 100% setting. Indeed, a larger training set size increases performance in the stratified setting. When disregarding the additional samples, DiverseVul only contains about 40,000 vulnerability pairs. It is plausible that significantly larger datasets will lead to better performance. The scalability under the stratified setting should be investigated in the future, adding more data through the use of the previous datasets.

Is it an issue of architecture, meaning the GNNs are unable to find relevant patterns? This question is left for future research. The references from the literature imply that token-based models such as LLMs perform no better [5], even when pretrained on code specific tasks. The experiments in the stratified setting suggest that at least within the GNN domain, of the evaluated models, the simple GIN architecture performs the best, outperforming more complex architectures such as the RGCN and GSL approaches. Perhaps because the expressivity of the GIN plays a crucial role, which expressive as the WL test, which is demonstrated both theoretically and empirically [9]. The other architectures are derived from the GCN.

Summarizing th experiment, it is found that when stratified by the number of nodes in the sample graphs, the prediction performance of the models decreases drastically. Although the GIN architecture performs the best, it is only 2.5% points better than

a random classifier. Examining the threshold heatmap and the different median code lengths of the vulnerable-benign pairs and the additional samples, it is concluded that the models mostly only learn the code graph length. Neither GNNs nor LLMs achieve good performance on unseen projects. But larger datasets might boost their performance.

4.8 Experiment 7: Performance per CWE

In the final experiment, the performance per CWE is studied. It is attempted to stratify both per CWE and graph size. However, the resulting bins of some CWEs are to small to derive interpretable results. Also, for some bin balanced accuracy and F1 score are undefined, because only a single class is present in the bin.

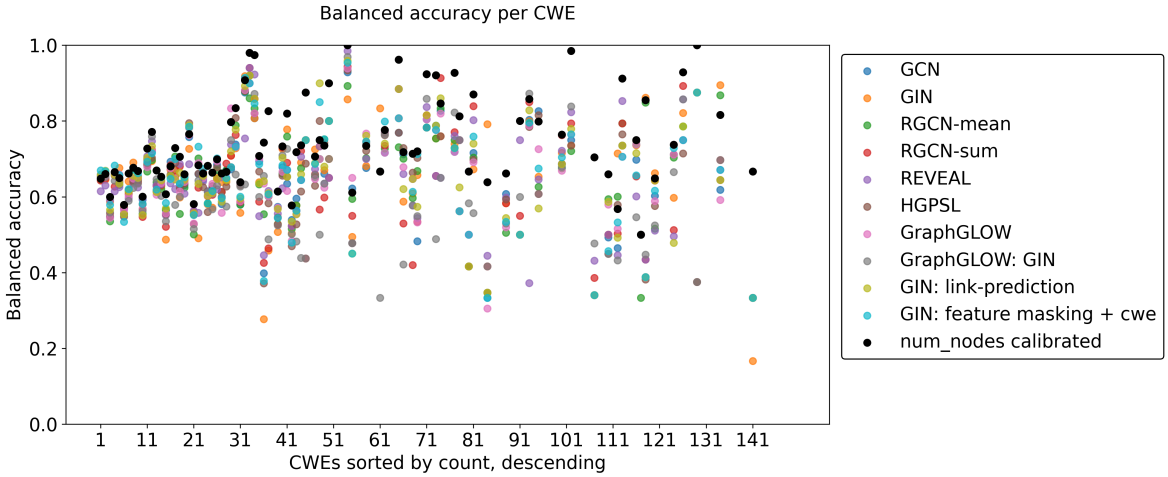


Figure 9: Predictions per CWE are shown. Not for every CWE samples are present in the test set. num_nodes is calibrated for each CWE separately to maximize balanced accuracy.

Instead, Figure 9 depicts the unstratified balanced accuracy of the models. For the most conservative comparison, the num_nodes baseline is tuned for the best threshold for each CWE separately. Under the presumption that the other models are only variations of the num_nodes baseline with a threshold which is less optimal than the globally optimal num_nodes baseline, they trade off balanced accuracy differently between CWEs. Therefore, the union of the models might “visually” outperform the

num_nodes baseline, which is avoided by setting the baseline to the most optimal thresholds for each CWE.

In Figure 9, the CWEs are ordered by their sample size in the whole DiverseVul [5] dataset. In many instances, the optimally calibrated num_nodes baseline outperforms other models. On the CWEs with the largest occurrence (on the left), some models learn additional facts about the data, indicating, that larger datasets might improve the detection capability. Supporting this claim is the fact, that compared to bigger CWEs for the smaller CWEs, the performance between models varies drastically, indicating that models have not trained on enough data.

4.9 Technical Details

Technical details regarding the experiments, such as pretraining and hyperparameters are provided in Appendix 1 and Appendix 2. We leave GraphGLOW-GIN as an interesting architecture for future research (Appendix 3).

5 Discussion

Followingly, we summarize the previous experiments and contextualize them to derive primary directives for future work in DL-based vulnerability detection.

In a theoretical analysis it is shown, how the balanced accuracy presents a better candidate to measure performance than the F1 score, because a one-class classifier always achieves a balanced accuracy of 50% but the associated F1 score changes with the data imbalance ratio. The plots under stratification in Experiment 6 illustrate how balanced accuracy is more interpretable in this setting.

Further, it is demonstrated, that to effectively evaluate DL-based methods, a project-level split to separate train and test data it is crucial to omit factors of variation specific to individual projects. A GIN network trained and evaluated on the same project achieves 15% points higher balanced accuracy compared to one trained in the project-split setting (Experiment 1).

In previous work, authors base their experiments on DiverseVul or similar datasets [5], [6] and fail to abandon the simple data setting and only for some experiments adopt the project-level split. They proceed to draw conclusions based on the un-separated data although attributing performance gains is difficult when models partially overfit to the testing data.

Supporting the argument from Experiment 1, we show that the num_nodes baseline which bases decisions only on the graph size outperforms even large, pretrained LLMs in the project-split setting (Experiment 2).

Different GNN-based and GSL-based approaches fail similar to the LLMs to outcompete the baseline. Including edge direction or edge type to train the RGCN [7] can not improve the outcome either (Experiment 4). Similarly, it is the case for pretraining (Experiment). At least, Graph structural features, including triangle counts and node degrees allow the models to perform on par with the baseline (Experiment 3).

When stratifying for the samples' graph sizes we find that models learn only marginally more than the baseline. The best performing model, the GIN only measures 2.5% points of balanced accuracy above num_nodes. The graph size is added as feature and subsequent training reveals that it is unlikely, that the models are "mislead" by the large portion of additional samples which are collected in DiverseVul to increase the dataset set. Rather, we attribute the insufficient performance to a lack of training data (Experiment 6).

Our findings demonstrate how DL-based vulnerability detection is still not effective enough to be applied in a real word setting on "unseen" data. However, the following observations might benefit its future progress.

The findings are interpreted in order of decreasing importance:

- ***Evaluation***: We illustrate how in the context of the DiverseVul dataset the right train-test separation and stratification is necessary in order to evaluate vulnerability detection methods and to validate one's approach. Future work would benefit from comparing against the right baselines and metrics.
- Regarding the ***data*** dimension, a shared topic in Chapter 4 is how improving the datasets both in quantity and quality is the most promising direction forward. In Experiment 7, models could not be evaluated in a setting stratified by CWE and graph size because of insufficient data. They showed drastically different performance for CWEs with the least data compared to CWEs with more data, inidicating that performance would increase with more data. Accordingly, also the GIN model achieved better balanced accuracy when trained with more data.

  However, publicly available vulnerability data is limited to OSS, of which the two largest projects, Linux and Chromium are already covered by vulnerability detection datasets [5], [6], [65]. The limited availability is best demonstrated by the overlapping percentage of over 50% of the DiverseVul dataset when joined with previous data sources [5].

  Thus, improving data quality above the 60% true-positive rate of DiverseVul [5] is equally as important.

  Related to data quality is the question whether the function level is the right representation to detect code vulnerabilities. [5] find that for many samples it is impossible to determine whether they are vulnerable without additional context. Other work explores higher level representations, such as an interprocedural one [1], [58]. Such representations provide detection methods with more context, for instance about which variables are user controlled. However, they come with the drawback of increased complexity and are selective about the data which can be used.
- ***Architecture*:** In context of the lack of training data and consequently also the difficulty of devising effective evaluation metrics, authors should be flexible when selecting and judging the performance of different architectures. In our experi-

ments, although simple, the GIN [9] architecture achieved the best results. Because of the small model size, experimentation regarding pretraining or the effect of increasing the dataset could be carried out quickly. Fast iteration would not have been possible with larger architectures such as GraphGLOW or LLM-based methods. After the *data* challenges have been solved, the RGCN represents an interesting choice of architecture, since it incorporates both the information about edge type and edge direction. LLM-based methods could show their effectiveness with larger abundance of vulnerability detection data. The 40,000 samples in the DiverseVul dataset [5] samples, disregarding the additionally collected data, did not suffice for outperforming the num_nodes baseline in the "unseen projects" setting.

- ***Training***: While the models pretrained in the current work achieve lower training loss, only in three instances their validation loss is slightly lower than the baseline's loss. Perhaps more sophisticated pretraining methods, for example ones which mimic code execution could be developed to aid performance. Other directions such as contrastive learning with vulnerable samples and their respective patches [6], [69], or [70] could show promising results as well.

In conclusion, the recipe of more careful evaluation in combination with simple model architectures and a special focus on data quality and quantity would greatly benefit progress in DL-based automatic vulnerability detection.

Acknowledgments

The author would like to thank Erik Imgrund for sharing his insights along the way and Dr. rer. nat. Martin Härterich for his guidance on the topic and feedback on the paper.

Bibliography

[1] T. Ganz, E. Imgrund, M. Härterich, and K. Rieck, "PAVUDI: Patch-based Vulnerability Discovery using Machine Learning," in *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023, pp. 704–717.

[2] D. A. Wheeler, "FlawFinder." [Online]. Available: https://github.com/david-a-wheeler/flawfinder

[3] T. Ganz, P. Rall, M. Härterich, and K. Rieck, "Hunting for Truth: Analyzing Explanation Methods in Learning-based Vulnerability Discovery," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroSP)*, 2023, pp. 524–541.

[4] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions," *Computers & Security*, vol. 120, p. 102813–102814, 2022.

[5] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 654–668.

[6] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[7] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*, 2018, pp. 593–607.

[8] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[9] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *arXiv preprint arXiv:1810.00826*, 2018.

[10] R. J. Wilson, *Introduction to graph theory*. Pearson Education India, 1979.

[11] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous graph transformer," in *Proceedings of the web conference 2020*, 2020, pp. 2704–2710.

[12] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008, pp. 16–24.

[13] N. I. of Standards and Technology, "Vulnerability Definition." [Online]. Available: https://nvd.nist.gov/vuln

[14] M. Corporation, "Common Weakness Enumeration." [Online]. Available: https://cwe.mitre.org/

[15] M. Corporation, "Common Vulnerabilities and Exposures." [Online]. Available: https://cve.mitre.org/

[16] M. Corporation, "Stubborn Weaknesses in the CWE Top 25." [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html

[17] J. M. Spring, "An analysis of how many undiscovered vulnerabilities remain in information systems," *Computers & Security*, vol. 131, p. 103191–103192, 2023.

[18] O. C. A. A. M. W. Mike Aizatsky Kostya Serebryany, "OSS-Fuzz." [Online]. Available: https://security.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[19] T. Ji, Y. Wu, C. Wang, X. Zhang, and Z. Wang, "The coming era of alpha-hacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques," in *2018 IEEE third international conference on data science in cyberspace (DSC)*, 2018, pp. 53–60.

[20] Fraunhofer-AISEC, "CPG Extractor." [Online]. Available: https://github.com/Fraunhofer-AISEC/cpg

[21] K. Weiss and C. Banse, "A Language-Independent Analysis Platform for Source Code," *arXiv preprint arXiv:2203.08424*, 2022.

[22] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[23] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[24] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[25] Z. Wang *et al.*, "TacticAI: an AI assistant for football tactics," *Nature Communications*, vol. 15, no. 1, pp. 1–13, 2024.

[26] Z. Li and A. B. Farimani, "Graph neural network-accelerated Lagrangian fluid simulation," *Computers & Graphics*, vol. 103, pp. 201–211, 2022.

[27] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386–387, 1958.

[28] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[29] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[30] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković, "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges," *arXiv preprint arXiv:2104.13478*, 2021.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation, Parallel Distributed Processing, Explorations in the Microstructure of Cognition, ed. DE Rumelhart and J. McClelland. Vol. 1. 1986," *Biometrika*, vol. 71, pp. 599–607, 1986.

[32] R. L. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro, "Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs," *arXiv preprint arXiv:1811.01900*, 2018.

[33] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels.," *Journal of Machine Learning Research*, vol. 12, no. 9, 2011.

[34] N. T. Huang and S. Villar, "A short tutorial on the weisfeiler-lehman test and its variants," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 8533–8537.

[35] S. Linnainmaa, "Taylor expansion of the accumulated rounding error," *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, 1976.

[36] K. Hornik, M. Stinchcombe, and H. White, "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks," *Neural networks*, vol. 3, no. 5, pp. 551–560, 1990.

[37] W. Hu *et al.*, "Strategies for pre-training graph neural networks," *arXiv preprint arXiv:1905.12265*, 2019.

[38] D. Erhan, A. Courville, Y. Bengio, and P. Vincent, "Why does unsupervised pre-training help deep learning?," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 201–208.

[39] H. Ren, W. Hu, and J. Leskovec, "Query2box: Reasoning over knowledge graphs in vector space using box embeddings," *arXiv preprint arXiv:2002.05969*, 2020.

[40] S. Feng *et al.*, "Twibot-22: Towards graph-based twitter bot detection," *Advances in Neural Information Processing Systems*, vol. 35, pp. 35254–35269, 2022.

[41] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *International conference on machine learning*, 2019, pp. 3734–3743.

[42] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[43] A. Radford *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9–10, 2019.

[44] Y. Zhu *et al.*, "A survey on graph structure learning: Progress and opportunities," *arXiv preprint arXiv:2103.03036*, 2021.

[45] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[46] P. Velickovic *et al.*, "Graph attention networks," *stat*, vol. 1050, no. 20, pp. 10–48550, 2017.

[47] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[48] Y. Chen, L. Wu, and M. Zaki, "Iterative deep graph learning for graph neural networks: Better and robust node embeddings," *Advances in neural information processing systems*, vol. 33, pp. 19314–19326, 2020.

[49] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, and Y. Shen, "Graph contrastive learning with augmentations," *Advances in neural information processing systems*, vol. 33, pp. 5812–5823, 2020.

[50] R. Entezari, M. Wortsman, O. Saukh, M. M. Shariatnia, H. Sedghi, and L. Schmidt, "The role of pre-training data in transfer learning," *arXiv preprint arXiv:2302.13602*, 2023.

[51] D. Mayo *et al.*, "Multitask learning via interleaving: A neural network investigation," in *Proceedings of the Annual Meeting of the Cognitive Science Society*, 2023.

[52] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7482–7491.

[53] L. Liebel and M. Körner, "Auxiliary tasks in multi-task learning," *arXiv preprint arXiv:1805.06334*, 2018.

[54] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[55] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[56] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[57] R. Russell *et al.*, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, 2018, pp. 757–762.

[58] Z. Li *et al.*, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[59] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM networks," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, 2005, pp. 2047–2052.

[60] Y. Liu *et al.*, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[61] W. Zhao, Q. Wu, C. Yang, and J. Yan, "Graphglow: Universal and generalizable structure learning for graph neural networks," in *Proceedings of the 29th*

ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2023, pp. 3525–3536.

[62] Z. Zhang *et al.*, "Hierarchical graph pooling with structure learning," *arXiv preprint arXiv:1911.05954*, 2019.

[63] C. Gallicchio and A. Micheli, "Fast and deep graph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, 2020, pp. 3898–3905.

[64] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 30–39.

[65] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "AC/C++ code vulnerability dataset with code changes and CVE summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[66] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[67] E. Imgrund, T. Ganz, M. Härterich, L. Pirch, N. Risse, and K. Rieck, "Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 2023, pp. 149–160.

[68] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[69] N. Risse and M. Böhme, "Limits of machine learning for automatic vulnerability detection," *arXiv preprint arXiv:2306.17193*, 2023.

[70] Y. Mirsky *et al.*, "{VulChecker}: Graph-based Vulnerability Localization in Source Code", in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6557–6574.

Index of Appendices

1 Appendix: Hyperparamter Search

All hyperparameter configurations can also be found in the repository linked in the Introduction.

For the GCN [8] and GIN [9] network which, because of their simplicity, represent baseline GNN performance, hyperparameters are searched in batch_size: [64, 128], hidden_channels: [64, 128], dropout: [0, 0.3, 0.7], depth: [2, 3].

For GraphGLOW [61] the number of iterations is searched in [1, 5], batch size is 32, GNN dropout is 0.3, the layers are 2, hidden channels are 129, the sparsity ratio is searched in [0, 0.2, 0.4] and the skip ratio searched in [0, 0.3, 0.7]

For HGP-SL [62] batch size is searched in [32, 128], the pooling ratio in [0.3, .05, 0.8], and the number of layers in [2, 3, 4].

For the RGCN [7] the optimal parameters from the GCN hyperparameter search are taken and additionally the number of globally shared basis matrices searched in [3, 8], the number of matrices shared between the forward and backward direction is searched in [1, 2], and for each direction separately the number of matrices is searched in [1, 2].

For REVEAL [23] we search the batch size in [32, 128], the feature dropout in [0, 0.5] and the global hidden channelsin [200, 256].

2 Appendix: Model Training

Here we briefly describe the training setup. Since all graphs with more than 1000 nodes are ignored (which removes about 3500 graphs), training is speed up significantly by implementing the models in a dense fashion. Further, this reduces memory footprint, such that GraphGLOW [61] can be trained faster, since setup allows us to batch graphs. For creating minibatches, adjacency matrices and feature vectors are padded to size 1000.

This further allows us to leverage the static-optimization options of pytorch, since the input has fixed dimensions, opposed to the sparse-matrix setting. Therefore, we can employ the structure learner and message-passing scheme of GraphGLOW in a non-approximate fashion, achieving similar speeds to the original approximate implementation. Furthermore, The setup enables full graph pretraining, instead of stochastic sampling for the node-level pretraining tasks.

3 Appendix: GIN-GraphGLOW

Another interesting research direction for the GraphGLOW architecture would be to study it's performance when the GIN architecture is used instead of the GCN as base model on the datasets from the original work [61]. Because the GIN applies no normalization terms to the aggregated neighborhood embeddings or more specifically the adjacency matrix, when the gradient of the loss is backpropagated through the learned adjacency matrix $\hat{A}$, it is independent of $\hat{A}$.

In the GCN's case, the adjacency matrix is normalized by the degree matrix: $\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}$, thus the gradient changes depending on the number of neighbors a node has. For the GCNs case this means that the gradient becomes smaller when connections to neighbors have higher probability. This means that for nodes for which many connections have been learned, their representations change more slowly. Therefore, the model could be more inflexible. Similarly, the gradient is really high for nodes without connection, which could lead the model to learn too many connections. Essentially, the rate of learning for edges becomes dependent on the other edges in the graph, which is not well justified.

For the GIN's case, the gradient does not change in response to more or less neighbors, thus it might be more flexible droping and learning new connections in the graph.