CodeGraphSMOTE - Data Augmentation for Vulnerability Discovery

Tom Ganz[1], Erik Imgrund[1], Martin Härterich[1], and Konrad Rieck[2]

[1] SAP Security Research {firstname.lastname}@sap.com
[2] Technische Universität Berlin rieck@tu-berlin.de

Abstract. The automated discovery of vulnerabilities at scale is a crucial area of research in software security. While numerous machine learning models for detecting vulnerabilities are known, recent studies show that their generalizability and transferability heavily depend on the quality of the training data. Due to the scarcity of real vulnerabilities, available datasets are highly imbalanced, making it difficult for deep learning models to learn and generalize effectively. Based on the fact that programs can inherently be represented by graphs and to leverage recent advances in graph neural networks, we propose a novel method to generate synthetic code graphs for data augmentation to enhance vulnerability discovery. Our method includes two significant contributions: a novel approach for generating synthetic code graphs and a graph-to-code transformer to convert code graphs into their code representation. Applying our augmentation strategy to vulnerability discovery models achieves the same originally reported F1-score with less than 20% of the original dataset and we outperform the F1-score of prior work on augmentation strategies by up to 25.6% in detection performance.

Keywords: Vulnerability Discovery · Data Augmentation · Graph Neural Networks.

1 Introduction

The research in the field of automatic vulnerability discovery has made remarkable progress recently but is still far from complete. Traditional rule-based tools suffer from high false negative or false positive rates in their detection performance. Consequently, advances in deep learning spark interest in the development of learning-based vulnerability discovery models. For instance, recent models borrow techniques from natural language processing using recurrent neural networks (RNNs), in particular, long short-term memorys (LSTMs), where the source code is processed as a flat sequence of code tokens [6, 30, 31, 38]. Even more recent approaches use graph neural networks (GNNs) thereby leveraging code graphs as a compact structure to represent the syntactic and semantic properties of programs [6, 10, 43, 55]. Graph learning is still a young field with a big room for improvement, but a promising technique to foster further research, especially in software security [15].

However, one major obstacle in learning-based vulnerability discovery, is obtaining enough representative code samples since most datasets available are either too small, unrealistic or imbalanced [6, 34, 44]. While clean code samples are vastly available and can be gathered easily, vulnerable samples, on the other hand, are scarce [2]. GNNs architectures suffer under that shortage the most, as they tend to overfit very easily and hence need balanced labels for training. Chakraborty et al. [6] report that models trained on inappropriate and imbalanced datasets are less transferable and have disadvantageous detection capabilities. The question arises then, on how to apply vulnerability discovery models to projects that lack a large history of vulnerabilities.

In traditional machine learning domains, data augmentation techniques are commonplace: For image data, random crops, offsets and rotations generate slightly different images with the same underlying meaning [40]. In tabular data, Synthetic Minority Oversampling Technique (SMOTE) is used to interpolate between minority samples and thus generate new samples [7]. Natural language processing uses techniques, such as synonym replacement, random word swaps, deletions or back translation [39]. While graph-based deep learning provides a unified method for neural networks on grids, groups, geodesics and gauges, no augmentation method for full graphs and even less so for code graphs exists.

Although augmentation techniques for node-level [54] and edge-level [52] tasks are available, techniques for graph classification are still unexplored [51]. The graph-specific augmentation methods that have been developed so far, either only perturb graphs [32, 51], cannot generate new graphs with node attributes of the target domain [21] or only perturb the node attributes [26]. Even worse, these augmentation strategies are disconnected from the underlying vulnerability discovery task, causing the generated samples to be neither syntactically nor semantically correct rendering them effectively uninterpretable.

More promising approaches like Graph2Edit [50], SequenceR [9] or Hoppity [12] can generate new vulnerable samples by learning semantic edits applied to clean code samples [34]. Although they are better suited for balancing vulnerability datasets than random graph perturbations, they already require a large number of vulnerable samples for training, which is the problem we are trying to solve in the first place. Furthermore, Nong et al. [34] observe that neural code editing approaches for vulnerability injection only yield significant improvements if the generated samples are assessed and selected and thus require extensive manual labor. Hence, we need data augmentation strategies explicitly tailored for vulnerability discovery which do not require large amounts of vulnerable training data and produce human-readable code. We present CodeGraphSMOTE, a novel method to augment code graph samples for vulnerability discovery models. It generates new vulnerable samples for the minority class in a dataset by porting SMOTE to the graph domain, specifically for code graphs. It does so by interpolating in the latent space of a variational autoencoder.

Our approach focuses on interpretable and sound sample generation. In essence, the contributions we present are:

1. A novel method to generate sound and interpretable synthetic code graphs.
2. A graph-to-code transformer to translate code graphs back to source code.
3. An evaluation demonstrating the practicability of our method.

Moreover, we publish our implementation of this method to foster further research in this direction[3]. In the rest of this paper, we review Related Work in Section 2. Then we lay down the preliminaries for vulnerability discovery in Section 3 and for data augmentation in Section 4. We proceed to thoroughly describe our method in Section 5, then present our experimental evaluation in Section 6 and end with the Conclusion in Section 7.

2 Related Work

Some graph-specific data augmentation methods perturbing the given samples have been developed, while graph data augmentation methods that are extending the dataset by generating new graphs are heavily underdeveloped.

DropEdge [37] reduces overfitting and over-smoothing by removing random edges from the graph at training time, and several improvements over DropEdge have been made by choosing the dropped edges in a biased way [16, 41]. Other methods are based on adding and removing edges [8, 53], masking node attributes [56], sampling a random subset of the nodes [13, 20] and cropping sub-graphs [45]. DeepSMOTE interpolates images in the latent space of an autoencoder instead of the original pixel space.
This greatly improves downstream classification performance for imbalanced datasets by generating synthetic minority samples and works better than generating new samples based on generative adversarial networks [11]. The same idea is applied to graphs to generate new nodes for imbalanced node classification tasks in [54]. There, a GAE is trained to reconstruct the adjacency matrix, while simultaneously learning latent features of the edges. The nodes are then oversampled using SMOTE in the latent space obtained by the GAE, which is also used to generate edges connecting the new nodes with the rest of the graph. This method achieves better accuracy in the task of imbalanced node classification. However, no adaptation of this method has been published, that interpolates between graphs to be used in graph classification. Chakraborty et al. [6] already apply SMOTE on graph embeddings before using a vulnerability classifier, this was found to increase detection performance. However, their method is generally, hardly applicable since it uses intermediate representations from another vulnerability discovery model and does not reconstruct interpolated graphs let alone the underlying source code.

Other approaches have been proposed from different research branches. Neural code editing uses deep learning to generate syntactically and semantically valid code samples. Different approaches, for instance, Hoppity [12], SequenceR [9] and Graph2Edit [50] have been developed. A recent study found out, that these approaches do not work well for augmenting vulnerability datasets [34].

[3] https://github.com/SAP-samples/security-research-codegraphsmote

Furthermore, the generated additional samples by Graph2Edit were found to be unrealistic but still helpful as additional training data for a vulnerability discovery downstream task. Lastly, Evilcoder [36] allows for automatically inserting bugs using rule-based code modifications, for instance, modified/removed buffer checks. However, this method produces vulnerable code samples that are too trivial to distinguish from clean samples and hence not suitable for machine learning.

3 Vulnerability Discovery

The preliminary materials for our method concerning learning-based vulnerability discovery, program representations and representation learning on code graphs are discussed in this section.

3.1 Learning-based Static Analyzer

We start by formalizing the vulnerability discovery task in the following section: Given a particular representation of a program, a static vulnerability discovery method is a decision function f that maps a code snippet x to a label $y \in \{\textsc{vulnerable}, \textsc{clean}\}$.

Learning-based methods for vulnerability discovery build on such a decision function $f = f_\Theta$ parameterized by weights Θ that are obtained by training on a dataset of vulnerable and non-vulnerable code [18]. Compared to classical static analysis tools, learning-based approaches do not have a fixed rule set and can therefore adapt to the characteristics of different vulnerabilities in the training data. Current learning-based approaches differ in the program representation used as input and the inductive bias, that is, the way f depends on the weights Θ.

3.2 Program Representations

Different representations for programs have been used as a basis for vulnerability discovery models in the past. Popular natural language processing-based approaches represent a program as the natural token sequence that appears in the source code [38]. Since programs can be modeled inherently as directed graphs [1], more recent approaches make use of graph representations [10, 43, 55] for source code instead of flat token sequences achieving state-of-the-art performances [38]. We refer to the resulting program representation as a *code graph* and denote the underlying directed graphs as $G = G(V, E)$ with vertices V and edges $E \subseteq V \times V$. Moreover, for $v \in V$, we define $N(v)$ as the set of its neighboring nodes.

Code graphs differ in the syntactic and semantic features they capture. Recent works, for instance, rely only on syntactic features using the abstract syntax tree (AST) [1], while newer approaches also capture the semantic properties, as for instance using the control flow graph (CFG), which connects statements with edges in the order they will be executed or the data flow graph (DFG) connecting the usages of variables. Based on these classical representations, combined graphs have been developed. A popular one is the code property graph (CPG) [49], which resembles a combination of the AST, CFG and program dependence graph (PDG). Other approaches use different combinations [5, 46]. All these representations are denoted CPGs, however, to distinguish them from the original CPG proposed by Yamaguchi et al. [49], we formally define a code graph in Definition 1 as an attributed and combined graph structure representing programs.

Definition 1. *A code graph is an attributed graph $G = (V, E, X_V, X_E)$ derived from source code and providing a syntactic or semantic view of the program.*

Naturally, code graphs have attributes, for instance, a node could have code tokens or AST labels attached. Since deep learning algorithms expect input features to be numeric, recent works embed these attributes into vector spaces [6, 10, 43, 55]. Hence a code graph extends the pair (V, E) by node attributes $X_V \in \mathsf{R}^{|V| \times d_V}$ of dimensionality d_V and edge attributes $X_E \in \mathsf{R}^{|\mathbf{E}| \times d_E}$ of dimensionality d_E [47].

3.3 Learning on Code Graphs

Vulnerability discovery using code graphs as input representation is a graph classification task. Building on a set of labels y and a set of attributed code graphs G it aims to learn a function $f_\Theta \colon G \mapsto y$. A set $\mathcal{G}_{\text{train}}$ of training graphs with known labels for each of those is given through which the parameters Θ of the function are optimized.

To build a graph neural network for code graphs, a convolutional and a global pooling block are needed [3]. Many graph convolutional blocks have been developed, the simplest of which is the graph convolutional network (GCN) [25]. The GCN can be formulated based on:

$$X' = \widehat{D}^{-1/2} \hat{A} \widehat{D}^{-1/2} X \Theta, \tag{1}$$

where $\hat{A} = A + I$ is the adjacency matrix with added self-loops, $\widehat{D}_{ii} = \sum_j \hat{A}_{ij}$ is the degree matrix and X the initial node feature matrix. Other types of convolutional blocks might be a gated graph neural network (GGNN) [29] or a graph isomorphism network (GIN) [48], where the former uses gated recurrent units instead of a feed-forward network and the latter has a separate optimizable parameter for the weights applied to the self-loops. In vulnerability discovery, however, we often lack a representative amount of vulnerable samples and, in consequence, have to deal with imbalanced graph classification [22].

4 Data Augmentation

Since there are few examples of vulnerabilities in the wild and the datasets for vulnerability discovery are unbalanced, as a remedy, we discuss the basics of data augmentation in this section.

4.1 SMOTE

The Synthetic Minority Oversampling Technique (SMOTE) [7] extends a dataset by generating new samples for all minority classes based on feature-space interpolation in the input domain. This way, the imbalance ratio of the dataset can be reduced and generalization to minority classes improved. New samples are generated by randomly selecting a sample and choosing a second sample from a random subset of the k nearest neighbors of the same class. New samples are generated by linearly interpolating between the features of the two selected nodes, yielding a new feature vector $\tilde{x} = \lambda x_1 + (1 - \lambda)x_2$, where x_1, x_2 are the features of the original samples and $\lambda \in [0, 1]$ is a uniformly random number.

4.2 Graph Autoencoder

Due to their discrete nature, SMOTE is not readily applicable to code graphs. It is not directly evident how one would interpolate between two graphs with both having, e.g., different numbers of nodes or edges. Some recent works apply SMOTE on the compressed latent space representation learned by an autoencoder in the computer vision domain [11], which learns to generate meaningful latent variables for samples from a data distribution [23] consisting of an encoder and a decoder. Moreover, the encoder of a variational autoencoder infers a probability distribution of the latent representation, by choosing a parametric probability distribution as the prior distribution for the latent variables. During training, the encoder infers the parameters of that distribution. For example, a variational autoencoder with a Gaussian distribution as the prior for the latent space would have two encoders $e_\mu(x) = z_\mu$ and $e_{\sigma^2}(x) = z_{\sigma^2}$. Then the latent representation needs to be decoded by sampling from $z \sim \mathcal{N}(z_\mu, z_{\sigma^2})$. This way, the decoder can still operate on a continuous latent representation, where it then tries to reconstruct the original input [24].

Graph autoencoders (GAEs) take this idea to the domain of graphs. They encode a graph into a latent space representation and decode it back into a graph. The latent space can be structured as a node- or graph-wise latent representation. The latter implies a single constant-sized vector for the complete graph, while the former latent space representation consists of a vector per node. Furthermore, the reconstruction target can be the adjacency matrix [24], the node or edge feature matrix [28]. Just like for the classical autoencoder, a variational variant exists, called variational graph autoencoder (VGAE).

5 CodeGraphSMOTE

CodeGraphSMOTE is applied on code graphs since not only do they provide state-of-the-art performance results on vulnerability discovery but also retain semantic and syntactic information in a compressed structure. Moreover, CodeGraphSMOTE is also equipped with a transformer to convert graphs back to source code representations. In particular, it consists of an autoencoder, interpolation method and graph-to-code transformation model. Figure 1 shows an overview of those blocks and their interplay.

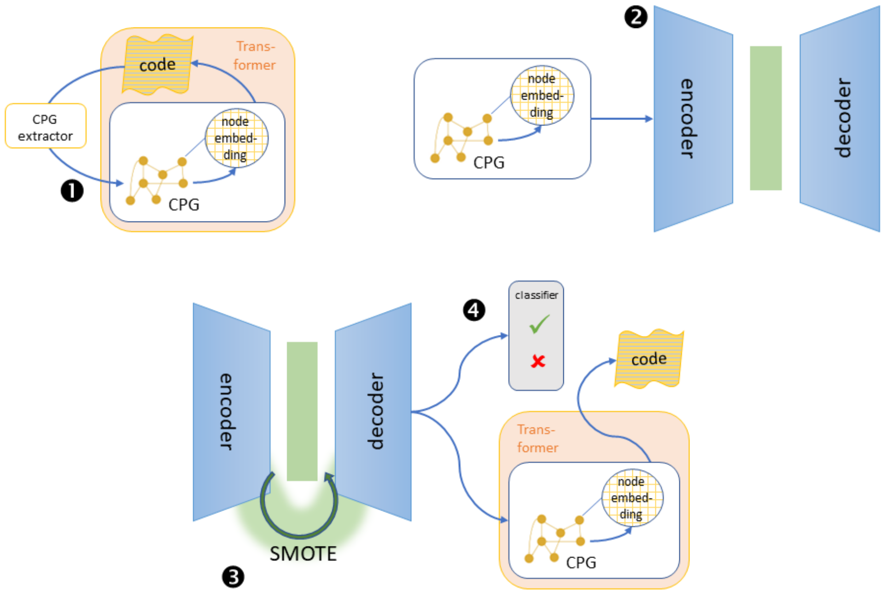


Fig. 1: Overview of the building blocks of CodeGraphSMOTE for training on imbalanced graph datasets.

5.1 Overview

The method has multiple training stages: The first stage is training a transformer model to convert code graphs to their original source code (1). The learned intermediate token embeddings of the transformer can then be used in the following stage to provide aggregated node embeddings for the VGAE. The second stage consists of training an autoencoder model to reconstruct code graphs in an unsupervised fashion (2). In the third stage, new samples can be generated by applying classic SMOTE to the VGAE latent representation (3). The last stage consists of training a vulnerability discovery model on an augmented version of the original dataset (4).

To augment the dataset, first, all code graphs of the original dataset are encoded by the autoencoder. Next, a balanced dataset is created by generating interpolated samples for the minority class. Lastly, the interpolated and original latent space representations are decoded by the autoencoder to generate new graphs for the vulnerability discovery downstream task. We proceed to explain our VGAE architecture and then provide insights into our transformer model.

5.2 Code Graph Generation

The input to the VGAE is a code graph, while the learning task is to pertain to as much information in the latent space as needed to reconstruct the original code graph. This ensures a semantically structured latent representation of vulnerable and clean code samples.

Encoder To produce latent space representations on the level of code graphs, a node-level autoencoder is implemented, where the intermediate node embeddings are calculated by applying a GNN to the input code graph. Conventional GNNs, such as GCNs, are low-pass filters and thus remove high-frequency features [35]. As it could be detrimental to decoding performance to smooth out the high-frequency features of the graph, an alternative architecture is used. The deconvolutional autoencoder [28] aims to preserve features of all components of the frequency spectrum by using more terms of the approximated eigendecomposition. Hence, a deconvolutional network with three layers and graph normalization [4] is used as the architecture for the encoder.

Decoder The decoder needs to decode not only the edge and node features but also the graph's topology. For the node level decoder, we use a GNN from Li et al. [28] which employs an approximate inverse convolution operation, restoring high-frequency features and consequently alleviating the problem of GCNs being mostly low-pass filters [35]. Since the node feature decoder depends on the graph's topology, we first reconstruct the adjacency matrix using a topology decoder.

The most prominent topology decoder is the inner product decoder which connects two distinct nodes with latent representations X'_i and X'_j by an edge with probability $\sigma(X'_i {X'}_j^T)$. Therefore, we can sample edges given these probabilities or, in a deterministic setting, draw an edge iff $\sigma(X'_i {X'}_j^T) > p_0$ (usually $p_0 = 0.5$), or in other words iff

$$X'_i {X'}_j^T > t \tag{2}$$

for some threshold t (usually $t = 0$). Note that increasing t leads to fewer edges.

During the reconstruction of the adjacency matrix, nodes with similar features tend to have a very high probability of an edge between them. As a solution, we decode the node features and topology separately, by splitting the latent representation of each node in half and using only one part for each decoder. The topology decoder is then trained to reconstruct the adjacency matrix using a weighted binary cross-entropy loss due to the natural sparsity of adjacency matrices.

Another problem arises since decoders based on the inner product can only reconstruct undirected graphs due to their inherent symmetry. Hence, we implement an asymmetric inner-product-based decoder, by splitting the adjacency matrix into two halves by its diagonal. One half of the topology decoder's latent space is used for the upper and one for the lower part of the adjacency matrix.

Finally, a third problem with inner-product-based decoders stems from the random node embeddings causing the expected average degree to increase proportionally with the number of nodes. This is problematic since the average

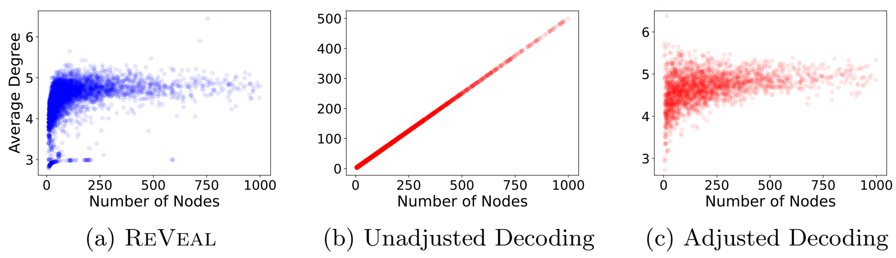


Fig. 2: Average degree compared against the number of nodes for code graphs, decoded naively and decoded with our correction.

degree will be higher for larger graphs, while in reality, code graphs have the property that their average node degree is independent of the number of nodes, which is illustrated in Figure 2a and Figure 2b.

The number of reconstructed edges for a particular graph can be seen as a random variable

$$|E| \sim B(p, |V|^2 - |V|) \tag{3}$$

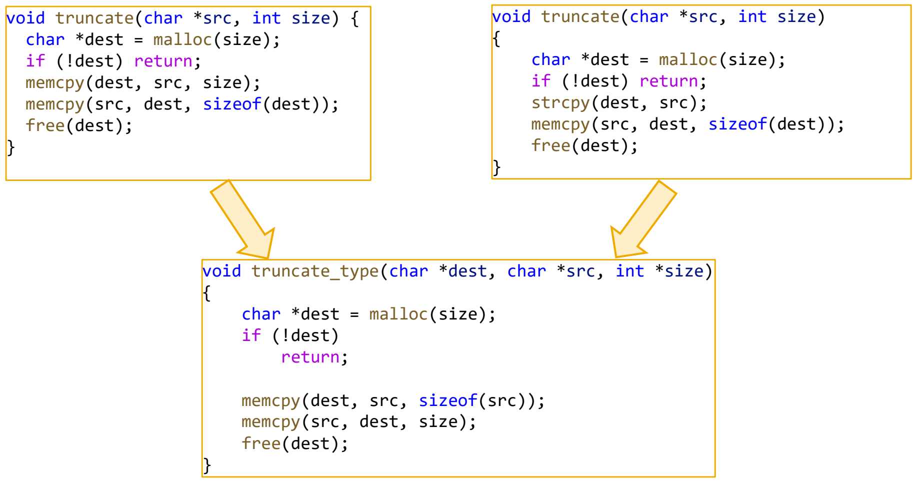
with p the binomial probability to decode a particular edge among the possible $|V|^2 - |V|$ edges. Note that we consider directed edges but no loops. Hence, we obtain $\mathsf{E}(|E|) = p \cdot (|V|^2 - |V|)$. Assuming a deterministic sampling, edges are decoded, when their similarity as defined by the inner product is above a certain threshold t. Adjusting the threshold by incorporating the expected average embedding distance of the closest embeddings reduces the effect of the proportionally increasing average degree as depicted in Figure 2c. The derivation of this adjustment can be found in Appendix A.1.

Interpolation Method To augment the code graph datasets, new samples need to be generated given a set of selected graphs. To do that, we propose a method to select and interpolate code graphs in their latent space representations.

A sample denotes an embedding matrix $X' \in \mathsf{R}^{|V| \times d_V}$ for a fixed latent space dimension d_V and number of nodes $|V|$. Since this matrix has different sizes for graphs with different numbers of nodes, no common distance metric can be applied to calculate the nearest neighbors. To mitigate this issue, the graphs are padded with zero vectors for non-existing nodes to the size of the largest graph in the dataset. Additionally, this same issue is found when interpolating the samples and solved in the same way. The interpolated embedding matrix

$$\hat{X'} = \lambda {X'}_a + (1 - \lambda){X'}_b \tag{4}$$

for two chosen code graphs G_a, G_b and a uniformly random $\lambda \in [0, 1]$ is truncated to a number of nodes interpolated in the same way: $|\hat{V}| = \lambda |V_a| + (1 - \lambda)|V_b|$ with the same λ. This interpolation method is not permutation-equivariant, thus the node ordering affects the results. Since we use the same method in the nearest



```
void truncate(char *src, int size) {
  char *dest = malloc(size);
  if (!dest) return;
  memcpy(dest, src, size);
  memcpy(src, dest, sizeof(dest));
  free(dest);
}
```

```
void truncate(char *src, int size)
{
    char *dest = malloc(size);
    if (!dest) return;
    strcpy(dest, src);
    memcpy(src, dest, sizeof(dest));
    free(dest);
}
```

```
void truncate_type(char *dest, char *src, int *size)
{
    char *dest = malloc(size);
    if (!dest)
        return;

    memcpy(dest, src, sizeof(src));
    memcpy(src, dest, size);
    free(dest);
}
```

Fig. 3: An interpolated sample in its code representation.

neighbor search, however, this results in the corresponding nodes already being close to each other.

Finally, new latent space representation samples are generated, which can then be decoded using the decoder. To train a graph vulnerability discovery model on the augmented code graphs, we have to reconstruct both, edges and nodes. The node features are recovered using the node decoder's output which is trained using a cosine embedding loss. The adjacency matrices need to be discretized by sampling from a Bernoulli distribution with a probability conforming to the edge probability in the reconstructed adjacency matrix.

5.3 Graph to Code Transformation

The ability to transform code graphs back to source code adds three beneficial properties to our method: First, we can produce human-readable samples, second, we are no longer limited to GNNs and third, we can use the latent node embeddings as a fixed size vector for each node in the VGAE. Thus, we train a transformer [42] to decode non-interpolated graphs and eventually apply it to interpolated samples.

Similar to the variational autoencoder (VAE), a transformer model consists of an encoder and a decoder comprising multiple blocks each. A single block consists of two components, namely a bidirectional multi-head self-attention mechanism and a feed-forward neural network. The attention mechanism generates an attention vector for each code token providing a weight on how much one token affects the other.

We propose an auto-regressive transformer model with a BART-like architecture consisting of six encoder and decoder layers, a width of 128 and only two heads per layer [27]. The code graph is linearized by sequentially extracting the tokens through a depth-first traversal of the AST and embedded using the pre-trained byte-pair tokenizer by Nijkamp et al. [33]. As each node consists of a variable number of tokens, the transformer has to learn a fixed-size token-level embedding of dimension $\mathsf{R}^{|N| \times d_V}$. To this end, an additional transformer encoder layer is trained jointly, that learns a normalized and pooled node vector. This way, inter-dependencies of the tokens are encoded in a token-level representation for the node-level features for the code graphs in the GAE training.

Figure 3 shows a generated synthetic sample at the bottom. We take the graph representation from the upper two samples and interpolate between both latent representations. Then we translate the resulting graph to its code representation using the transformer model.

Both source methods truncate a string but run into buffer overflows due to potential size mismatches between src, dest and size. Note, that the upper samples are taken from two real vulnerabilities. The resulting generated function has a different name and signature, but its body is similar: It is obvious, that the resulting method is also an example of string truncation with a buffer overflow. The unused parameter char *dest in the signature may be erroneously caused by either the generation of the graph or the conversion of the graph to code but may be negligible due to the nature of data augmentation.

6 Evaluation

This section introduces the experiments designed to tackle three research questions, in particular, we describe our experimental setting and provide empirical results answering the following questions:

RQ1 *Does CodeGraphSMOTE provide a sound latent representation?*
RQ2 *Can CodeGraphSMOTE improve detection performance when we lack data?*
RQ3 *Do the augmented datasets yield better model transferability?*

6.1 Experimental Setting

We rely on Fraunhofer-CPG [46] as a tool to generate code graphs and preprocess them using networkx [19]. We use the GNN implementations from Pytorch Geometric [14] and train them on AWS EC2 g4dn instances. All experiments are conducted using 10-fold cross-validation and our VGAE consists of 2 encoder and 2 decoder layers for topology and node features respectively with a dimension of 384. We use an Adam and AdamW optimizer for the VGAE and the transformer with learning rates of 0.0005 and 0.001 respectively.

Datasets and Models For our experimental evaluation, we use the following three datasets from recent publications around learning-based vulnerability discovery. All three datasets consist of a corpus of vulnerable and clean samples from C and C++ code repositories.

1. Chromium+Debian. The Chromium+Debian dataset consists of 1924 vulnerable and 17294 clean samples. Thus, the imbalance ratio is 10.01%. The dataset has been extracted from the Debian and Chromium bug tracker and hence contains C++ code samples [6].
2. FFmpeg+Qemu. The FFmpeg+Qemu dataset is nearly balanced with a ratio of 45.96% having 11466 and 9751 samples respectively for the clean and vulnerable class. The code was extracted using security-related keywords that have been matched against commits in the Github project repositories from Qemu and FFmpeg [55].
3. *PatchDB*. Finally, we utilize PatchDB [44], which consists of patches extracted from the national vulnerability database (NVD) for multiple C and C++ open-source projects. Vulnerable samples are labeled by their common weakness enumerations (CWEs). Overall, it has 3441 vulnerable and 30149 clean samples resulting in an imbalance ratio of 10.24%.

As ML models for the downstream vulnerability discovery task, we use ReVeal and Devign [6, 55]. They both rely on GGNNs and a pooling layer followed by a feed-forward neural network prediction head. We train the transformer, VGAE and downstream classifier on the same training set and test on a disjoint separate dataset containing only real samples.

Metrics We use two metrics recommended especially for imbalanced learning tasks to provide a comprehensive evaluation of the model's performance. By comparing these scores before and after augmentation, we can assess whether the augmentation has improved the model's performance.

1. *F1-score*. The F1-score is a commonly used metric to evaluate the performance of a classification model. It's a measure of the model's ability to correctly predict both positive and negative classes. The F1-score is calculated as the harmonic mean of precision (P) and recall (R).
2. *Balanced accuracy*. The second is balanced accuracy, which takes into account both, the true positive rate and true negative rate, and is calculated as the average of these two rates.

Baselines To compare our method for plausibility and practicability, we benchmark against four commonly used augmentation strategies for graphs in general and vulnerability discovery models in particular.

1. *SARD enrichment*. The software assurance reference dataset (SARD) is a synthetic vulnerability corpus containing about 30k vulnerable and 30k clean samples. The vulnerable samples are pattern generated, and consequently, ML models tend to overfit [6]. We use vulnerable samples from this dataset to enrich their original dataset as proposed by Nong et al. [34].
2. *Graph Perturbation*. Borrowed from the graph domain, we can augment the dataset by randomly dropping nodes and edges. This graph perturbation

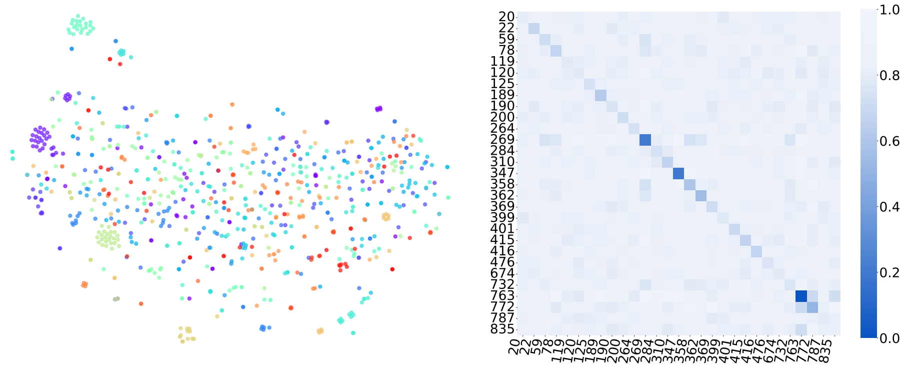


Fig. 4: Average normalized inter- and intra-cluster distance per CWE on PatchDB.

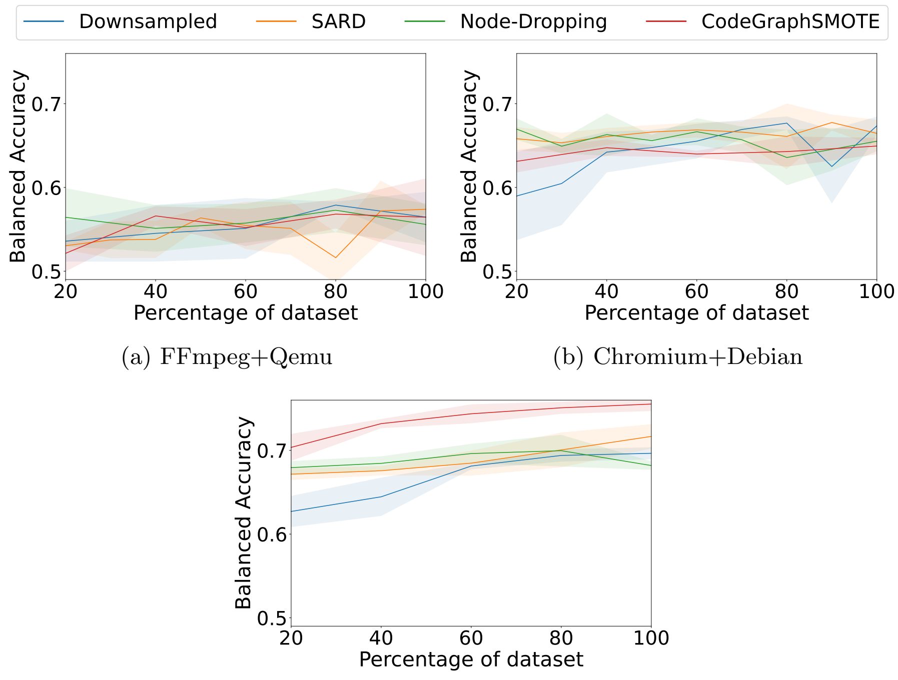
technique can be applied to vulnerable samples to augment the dataset. However, this technique will most likely break the code graph's semantics and generate unintelligible samples.

3. *Graph2Edit.* According to an empirical study [34], Graph2Edit [50] is currently state-of-the-art in neural code generation for vulnerability discovery. It uses a GGNN to learn graph and node embeddings and a LSTM network to predict edit actions on the AST. It is trained to convert ASTs of clean samples to vulnerable ones.
4. *Downsampling.* A naive approach is to downsample the majority class. That is, we remove clean samples until we have an imbalance ratio of 50%.

6.2 Results

The discussion of the experimental results is organized along the three research questions posed at the beginning of this section, which we try to answer in the following.

RQ1 | *Does CodeGraphSMOTE provide a sound latent representation?* First, we want to assess whether the learned latent space from the VGAE in CodeGraphSMOTE represents important features from the code graph and in particular for vulnerability discovery. The scatter plot on the left-hand side of Figure 4 shows a two-dimensional t-SNE embedding of the VGAE latent representation per vulnerable code graph of the training set from PatchDB. Each sample is colored by its CWE. We can reason about the quality of the interpolated vulnerable samples since it correlates with the quality of the cluster. At least five clusters are clearly visible, including CWE-269 (improper privilege management), CWE-347 (improper verification of cryptographic signature) and CWE-763 (release of invalid pointer or reference).The right-hand side of Figure 4 shows the average



(a) FFmpeg+Qemu

(b) Chromium+Debian

(c) PatchDB

Fig. 5: Dataset augmentation strategies by replacement.

inter- and intra-cluster distances between the CWEs. The matrix is diagonal-dominant suggesting that the learned representation places samples from the same type of vulnerability closer together. Overall, we can conclude that the latent representation encodes crucial information about the semantics of the code and vulnerability. Since SMOTE selects neighbors that are close to each other as interpolation candidates, it is safe to assume that it will automatically interpolate between vulnerabilities of the same type.

The latent space representation learned by the VGAE clusters code graphs by their vulnerability type, making it suitable for SMOTE on vulnerability datasets.

RQ2 | *Can CodeGraphSMOTE improve detection performance when we lack data?* We evaluate our method against simple downsampling, SARD enrichment and node-dropping. We simulate smaller datasets, by removing a partition of vulnerable samples from the original FFmpeg+Qemu, Chromium+Debian and PatchDB datasets and re-balance them by augmenting the remaining. Figure 5 shows the performance of the Chromium+Debian model measured in their bal-

Table 1: Cross dataset evaluation presenting the F1-score. *C+D* and *F+Q* denote the Chromium+Debian and FFmpeq+Qemu datasets respectively.

Model	Training	Testing	Downsampling	Graph2Edit	SARD	CodeGraphSMOTE
ReVeal	C+D	F+Q	31.86%	36.94%	15.16%	**41.37**%
	F+Q	C+D	19.31%	**21.31**%	20.07%	18.26%
Devign	C+D	F+Q	5.25%	7.11%	5.70%	**62.97**%
	F+Q	C+D	16.83%	**19.31**%	18.58%	18.20%

anced accuracy. The x-axis denotes the percentage of real samples remaining from the original datasets, while 100% corresponds to the original dataset rebalanced using the specific method. On PatchDB, the most realistic dataset, CodeGraphSMOTE achieves an overall area under balanced accuracy score of 73.9%, compared against 68.9%, 65.5% and 63.5% for SARD, Node-Dropping and Downsampling respectively. Hence CodeGraphSMOTE yields a significantly stronger improvement compared to the other approaches with nearly 24% improvement against simple downsampling. This is particularly interesting because PatchDB has the most diverse and realistic dataset containing samples from multiple projects collected directly from the NVD. Although no augmentation strategy is a clear winner on the FFmpeg+Qemu dataset and the overall performance is only slightly above 55% as already shown by Chakraborty et al. [6] and Ganz et al. [15], we can still see that SARD is slightly worse than the other methods. Other observations are not statistically significant due to their large standard deviations.
For the Chromium+Debian datasets all augmentation strategies have less influence on the model as depicted by the large standard deviation compared to their effect on PatchDB. There is no augmentation strategy that dominates another with statistical significance. Thus, no method provides a statistically significant improvement over another except for downsampling. Downsampling is the worst method on every dataset while simple graph perturbation is the second best approach.

Our method provides an improvement of up to 21% balanced accuracy against simple downsampling on realistic datasets and keeps the model performance constant at only 20% of the original dataset.

RQ3 | *Do the augmented datasets yield better model transferability?* Finally, we evaluate whether a pre-trained instance of CodeGraphSMOTE can be used to enhance vulnerability discovery when applied to different datasets that lack labeled or vulnerable samples. Despite recent publications showing that the Devign dataset (FFmpeg+Qemu) and model are unrealistic and underperforming [6, 15], we still include both to stay comparable with Graph2Edit.

In contrast, we excluded PatchDB, which contains over 300 C and C++ projects, to demonstrate the usability of CodeGraphSMOTE on small individual projects.

Table 1 shows the average F1-score for the models ReVeal and Devign using four different augmentation strategies to re-balance the datasets. While the models have been trained and tested on disjoint datasets, our results, as shown in Table 1 and Figure 5, indicate that training on the FFmpeg+Qemu dataset did not yield noteworthy detection capabilities. However, with F1-scores of 18.26% and 18.20%, respectively, CodeGraphSMOTE can be considered comparable to Graph2Edit with its F1-scores of 21.31% and 19.31%. Furthermore, the models trained on the Devign+Qemu dataset did not provide any transferability, highlighting the challenges associated with this dataset.

In contrast, training on the Chromium+Debian dataset reveals that CodeGraphSMOTE significantly improves detection capabilities by a factor of nearly 9 for Devign and 12% for ReVeal, as compared to the state-of-the-art method Graph2Edit. Interestingly, the Chromium+Debian dataset, with a higher degree of class imbalance than the FFmpeg+Qemu dataset, demonstrates the superior performance of CodeGraphSMOTE with increasing class imbalance.

CodeGraphSMOTE significantly improves model transferability by up to 800% measured by the F1-score. The performance enhancement scales with increasing class imbalance.

7 Conclusion

This work introduces CodeGraphSMOTE, a novel augmentation method designed to address imbalanced attributed code graph datasets. Our approach employs a variational graph autoencoder to interpolate between code graph samples in the latent space, and a transformer model to convert these graphs back to their source code representation. On the way, we also address several common issues with graph autoencoders in general, particularly in topology reconstruction. Through experimental evaluation, we demonstrate that our method not only achieves comparable vulnerability discovery performance with fewer data but also improves the models' generalizability and transferability to new datasets.

Acknowledgment. This work has been funded by the German Federal Ministry of Education and Research (BMBF) in the project IVAN (FKZ: 16KIS1165K).

A Appendix

A.1 Derivation of the Threshold Adjustment

Our goal is to adjust the threshold t in Equation (2) such that the average degree of a vertex in the reconstructed graph equals a given degree deg. Using $\mathsf{E}(|E|) = p\,(|V|^2 - |V|)$, since we consider directed edges but no loops, this is the case if $p\,(|V|^2 - |V|) = deg\,|V|$ or

$$p = \frac{deg}{|V| - 1} \quad .$$

Now $p = P(X'_i\,{X'}_j^T > t)$ where X'_i and ${X'}_j$ are d-dimensional latent representations of two nodes which (due to the targeted latent distribution of the VAE) we assume to be independent and identically distributed according to the standard normal distribution $\mathcal{N}(\mathbf{0}_d, \mathbf{I}_d)$. Hence the correct adjusted choice of t is given by

$$t = \mathrm{CDF}_Z^{-1}\left(1 - \frac{deg}{|V| - 1}\right) \tag{5}$$

where CDF_Z is the cumulative distribution function of the product $Z = XY$ of two i.i.d. vectors X and Y as above. By symmetry, we may assume that Y is parallel to the first coordinate axis and then X can be marginalized to this axis without affecting the inner product. Thus, we assume w.l.o.g. $d = 1$.

The density of $Z = XY = \frac{1}{4}\left((X+Y)^2 + (X-Y)^2\right)$ (a.k.a. variance gamma distribution) is known to be given by $\mathrm{PDF}_Z(z) = \frac{1}{\pi}K_0(z)$ where $K_0(z)$ is a modified Bessel function of the second kind (see [17]).

Finally, we use numerical integration to get $\mathrm{CDF}_Z(z) = \frac{1}{\pi}\int^z K_0(z)dz$ and solve numerically for t as in Equation (5).

Bibliography

[1] Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. ArXiv **abs/1711.00740** (2017)

[2] Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K.: Dos and don'ts of machine learning in computer security. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 3971–3988, USENIX Association, Boston, MA (Aug 2022), ISBN 978-1-939133-31-1

[3] Bronstein, M.M., Bruna, J., Cohen, T., Velivckovi'c, P.: Geometric deep learning: Grids, groups, graphs, geodesics, and gauges (2021)

[4] Cai, T., Luo, S., Xu, K., He, D., yan Liu, T., Wang, L.: Graphnorm: A principled approach to accelerating graph neural network training (2020)

[5] Cao, S., Sun, X., Bo, L., Wei, Y., Li, B.: Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. **136**, 106576 (2021)

[6] Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? IEEE TRANSACTIONS ON SOFTWARE ENGINEERING **TBD**, 1 (2020)

[7] Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: Synthetic minority over-sampling technique. J. Artif. Int. Res. **16**(1), 321–357 (jun 2002), ISSN 1076-9757

[8] Chen, D., Lin, Y., Li, W., Li, P., Zhou, J., Sun, X.: Measuring and relieving the over-smoothing problem for graph neural networks from the topological view (2019)

[9] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L., Poshyvanyk, D., Monperrus, M.: Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering **47**(09), 1943–1959 (sep 2021), ISSN 1939-3520

[10] Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y.: Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. **30**(3) (apr 2021)

[11] Dablain, D., Krawczyk, B., Chawla, N.: Deepsmote: Fusing deep learning and smote for imbalanced data. IEEE Transactions on Neural Networks and Learning Systems **PP**, 1–15 (01 2022), https://doi.org/10.1109/TNNLS.2021.3136503

[12] Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., Wang, K.: Hoppity: Learning graph transformations to detect and fix bugs in programs. In: International Conference on Learning Representations (2020)

[13] Do, T.H., Nguyen, D.M., Bekoulis, G., Munteanu, A., Deligiannis, N.: Graph convolutional neural networks with node transition probability-based message passing and DropNode regularization. Expert Systems with Applications **174**, 114711 (2021)

[14] Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)

[15] Ganz, T., Härterich, M., Warnecke, A., Rieck, K.: Explaining graph neural networks for vulnerability discovery. In: Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security, p. 145–156, AISec '21, New York, NY, USA (2021)

[16] Gao, Z., Bhattacharya, S., Zhang, L., Blum, R.S., Ribeiro, A., Sadler, B.M.: Training robust graph neural networks with topology adaptive edge dropping (2021)

[17] Gaunt, R.E.: Products of normal, beta and gamma random variables: Stein operators and distributional theory. Brazilian Journal of Probability and Statistics **32**(2), 437 – 466 (2018)

[18] Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L.: Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, p. 85–96, CODASPY '16, New York, NY, USA (2016)

[19] Hagberg, A., Swart, P., S Chult, D.: Exploring network structure, dynamics, and function using networkx (1 2008)

[20] Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: NIPS (2017)

[21] Han, X., Jiang, Z., Liu, N., Hu, X.: G-mixup: Graph data augmentation for graph classification. In: Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., Sabato, S. (eds.) Proceedings of the 39th International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 162, pp. 8230–8248, PMLR (17–23 Jul 2022)

[22] Johnson, J.M., Khoshgoftaar, T.M.: Survey on deep learning with class imbalance. Journal of Big Data **6**(1), 27 (Mar 2019)

[23] Kingma, D.P., Welling, M.: Auto-encoding variational bayes. CoRR **abs/1312.6114** (2014)
[24] Kipf, T.N., Welling, M.: Variational graph auto-encoders. NIPS Workshop on Bayesian Deep Learning (2016)
[25] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (ICLR) (2017)
[26] Kong, K., Li, G., Ding, M., Wu, Z., Zhu, C., Ghanem, B., Taylor, G., Goldstein, T.: Robust optimization as data augmentation for large-scale graphs. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 60–69 (June 2022)
[27] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L.: BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 7871–7880, Association for Computational Linguistics, Online (Jul 2020)
[28] Li, J., Li, J., Liu, Y., Yu, J., Li, Y., Cheng, H.: Deconvolutional networks on graph data. In: Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems (2021)
[29] Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. In: Bengio, Y., LeCun, Y. (eds.) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings (2016)
[30] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing **19**(4), 2244–2258 (2022)
[31] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society (2018)
[32] Luo, Y., McThrow, M., Au, W.Y., Komikado, T., Uchino, K., Maruhashi, K., Ji, S.: Automated data augmentations for graph classification (2022)
[33] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: Codegen: An open large language model for code with multi-turn program synthesis. In: The Eleventh International Conference on Learning Representations (2023)
[34] Nong, Y., Ou, Y., Pradel, M., Chen, F., Cai, H.: Generating realistic vulnerabilities via neural code editing: An empirical study. p. 1097–1109, ESEC/FSE 2022, New York, NY, USA (2022)
[35] NT, H., Maehara, T.: Revisiting graph neural networks: All we have is low-pass filters (2019)
[36] Pewny, J., Holz, T.: Evilcoder: Automated bug insertion. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, p. 214–225, ACSAC '16, New York, NY, USA (2016)
[37] Rong, Y., Huang, W., Xu, T., Huang, J.: Dropedge: Towards deep graph convolutional networks on node classification. In: ICLR (2020)
[38] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. pp. 757–762 (12 2018)
[39] Sennrich, R., Haddow, B., Birch, A.: Improving neural machine translation models with monolingual data. CoRR **abs/1511.06709** (2015)

[40] Shorten, C., Khoshgoftaar, T.M.: A survey on image data augmentation for deep learning. Journal of Big Data **6**(1), 60 (Jul 2019), ISSN 2196-1115

[41] Spinelli, I., Scardapane, S., Hussain, A., Uncini, A.: Biased edge dropout for enhancing fairness in graph representation learning (2021)

[42] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 30, Curran Associates, Inc. (2017)

[43] Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z.: Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Transactions on Information Forensics and Security **16**, 1943–1958 (2021), ISSN 15566021

[44] Wang, X., Wang, S., Feng, P., Sun, K., Jajodia, S.: Patchdb: A large-scale security patch dataset. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 149–160 (2021)

[45] Wang, Y., Wang, W., Liang, Y., Cai, Y., Hooi, B.: Graphcrop: Subgraph cropping for graph classification. CoRR **abs/2009.10564** (2020)

[46] Weiss, K., Banse, C.: A language-independent analysis platform for source code (2022)

[47] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE Transactions on Neural Networks and Learning Systems **32**(1), 4–24 (jan 2021), https://doi.org/10.1109/tnnls.2020.2978386

[48] Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations (2019)

[49] Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604 (2014)

[50] Yao, Z., Xu, F.F., Yin, P., Sun, H., Neubig, G.: Learning structural edits via incremental tree transformations. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net (2021)

[51] Zhao, T., Liu, G., Gunnemann, S., Jiang, M.: Graph data augmentation for graph machine learning: A survey (2022)

[52] Zhao, T., Liu, G., Wang, D., Yu, W., Jiang, M.: Counterfactual graph learning for link prediction. CoRR **abs/2106.02172** (2021)

[53] Zhao, T., Liu, Y., Neves, L., Woodford, O.J., Jiang, M., Shah, N.: Data augmentation for graph neural networks. In: AAAI (2021)

[54] Zhao, T., Zhang, X., Wang, S.: Graphsmote: Imbalanced node classification on graphs with graph neural networks. Proceedings of the 14th ACM International Conference on Web Search and Data Mining (2021)

[55] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32, Curran Associates, Inc. (2019)

[56] Zhu, Y., Xu, Y., Yu, F., Liu, Q., Wu, S., Wang, L.: Graph contrastive learning with adaptive augmentation. In: Proceedings of the Web Conference 2021, ACM (2021)