# Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection

Xiao Cheng
University of Technology Sydney
Sydney, NSW, Australia

Guanqin Zhang
University of Technology Sydney
Sydney, NSW, Australia

Haoyu Wang
Huazhong University of Science and Technology
Wuhan, China

Yulei Sui
University of Technology Sydney
Sydney, NSW, Australia

## ABSTRACT

Machine learning and its promising branch deep learning have shown success in a wide range of application domains. Recently, much effort has been expended on applying deep learning techniques (e.g., graph neural networks) to static vulnerability detection as an alternative to conventional bug detection methods. To obtain the structural information of code, current learning approaches typically abstract a program in the form of graphs (e.g., data-flow graphs, abstract syntax trees), and then train an underlying classification model based on the (sub)graphs of safe and vulnerable code fragments for vulnerability prediction. However, these models are still insufficient for precise bug detection, because the objective of these models is to produce classification results rather than comprehending the semantics of vulnerabilities, e.g., pinpoint bug triggering paths, which are essential for static bug detection.

This paper presents ContraFlow, a selective yet precise contrastive value-flow embedding approach to statically detect software vulnerabilities. The novelty of ContraFlow lies in selecting and preserving feasible value-flow (aka program dependence) paths through a pretrained path embedding model using self-supervised contrastive learning, thus significantly reducing the amount of labeled data required for training expensive downstream models for path-based vulnerability detection. We evaluated ContraFlow using 288 real-world projects by comparing eight recent learning-based approaches. ContraFlow outperforms these eight baselines by up to 334.1%, 317.9%, 58.3% for informedness, markedness and F1 Score, and ContraFlow achieves up to 450.0%, 192.3%, 450.0% improvement for mean statement recall, mean statement precision and mean IoU respectively in terms of locating buggy statements.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Computing methodologies → Machine learning**; • **Software and its engineering → Automated static analysis**.

## KEYWORDS

Path sensitive, code embedding, contrastive learning, vulnerabilities

**ACM Reference Format:**
Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534371

## 1 INTRODUCTION

Although considerable efforts have been made to improve software security, vulnerabilities remain a major concern within modern software development. Existing static bug detectors (e.g., Checkmarx [35], RATs [60], ITS4 [70], CoBOT [25], Coverity [66], SVF [63], Infer [18]) rely heavily on user-defined rules and domain knowledge for effectiveness, making them labour-intensive [51]. Although they have shown their successes in detecting conventional well-defined bugs (e.g., use-after-frees, null dereferences), these detectors still have difficulty in finding a wider range of vulnerabilities (e.g., naming issues [32] and incorrect business logic [10]) and report a large portion of false positives/negatives [10, 51]. The recent success of deep learning techniques has opened new opportunities to develop more intelligent detection systems by learning vulnerability patterns that capture the correlation between vulnerable programs and their extracted code features through prediction models.

***Existing Efforts and Limitations.*** Code embedding, which aims to represent code semantics through distributed vector representations, has recently been proposed for source code analysis and bug detection. Initially, the embedding approach treats a program as textual tokens [49–51] by applying natural language processing techniques to learn code semantics without code structural information. Later, several approaches [7, 10, 48, 81] improve the embedding results by preserving structural information, for example, through program dependence graphs, and then use graph neural networks (GNNs) [41, 47] to classify whether a (sub)graph of a code fragment is vulnerable or not. However, although learning a graph representation of code can be used for code classification or summarization tasks, it is still insufficient for path-based vulnerability detection. This is because the input graph representation does not distinguish program paths, which are opaque to backend GNNs. The graph features are learned from message passing between all connected node pairs in GNNs, but unfortunately, without the knowledge of any feasible/infeasible value-flow (program dependence) paths.

*ISSTA '22, July 18–22, 2022, Virtual, South Korea*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534371

Therefore, these prediction models are unaware of potential buggy paths, which indicate how a bug originates and is triggered. This is one of the major aims of static bug detection: to help a practitioner quickly locate and fix the reported vulnerabilities.

***Insights and Challenges.*** To address the above limitations, the detection approach needs to work on a precise learning model that can preserve value-flow paths rather than the entire graph which does not distinguish feasible/infeasible program dependence paths. Inspired by the idea of a bag of tokens in word embedding [54], some recent code embedding approaches embed a bag of paths on abstract syntax trees (ASTs) [1] or value-flow graphs (VFGs) [62] for code classification and summarization. The approaches randomly sample a small fraction of paths to produce their embedding vectors, and then aggregate them to form the final representation of a code fragment. However, these approaches can not be directly used for sophisticated tasks such as path-based bug detection, due to a potentially unbounded number of program paths which need to be embedded. The effectiveness of the path-based model lies in the strategy of path selection. It is challenging yet important to identify and preserve individual feasible paths [14] rather than the aggregation of infeasible or bug irrelevant paths via random sampling, to avoid imprecision during embedding. This requires selectively learning paths with discriminative features which contribute to bug semantics during model training, to yield precise embedding for path-based vulnerability detection.

***Our Solution.*** We present ContraFlow, a path-sensitive code embedding approach which uses self-supervised contrastive learning to pinpoint vulnerabilities based on value-flow paths. To avoid training a large number of value-flow paths, ContraFlow enables contrastive learning to first work with value-flow paths extracted from unlabeled code fragments, by learning a pretrained representation (or path encoder) such that semantically similar paths stay close to each other while dissimilar ones are far apart. The resulting pretrained path encoder is then used to guide the path selection process through self-supervised active learning by reducing a substantial number of value-flow paths required for training the downstream fine-tuning task, i.e., vulnerability detection. In the path selection process, we introduce sparse and guarded value-flow analysis into code embedding to further refine the selected value-flows by checking their path feasibility [12, 64]. Thus, only feasible value-flow paths are preserved in the embedding space to precisely represent a code fragment. Our vulnerability detection is then able to report likely buggy paths based on the path-sensitive representation and interpret important value-flow paths contributing to a vulnerability, hence bridging the gap between existing path-unaware learning approaches and path-based static vulnerability detection.

***Framework Overview.*** Figure 1 provides an overview of our framework consisting of a training process and a prediction process. The training process comprises the following three major phases:

***(a) Contrastive Value-Flow Embedding.*** This phase aims to train a value-flow embedding model, Value-flow Path Encoder (VPE), using contrastive learning. Given a set of value-flow paths extracted from unlabeled source code using an existing static analyzer SVF [63], we first perform data augmentation to generate contrastive value-flow representations [26], and then utilize the standard Noise Contrastive Estimate (NCE) loss function [9] to maximize the agreement between semantically similar value-flow

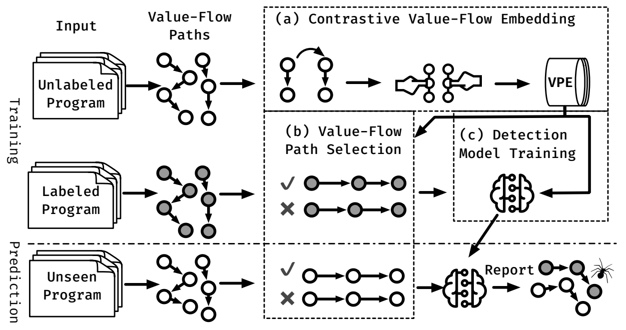


**Figure 1: Framework overview.**

path vectors. This updates the parameters of our VPE to prompt it to preserve the deep semantics of value-flow paths. The pretrained VPE is used in the next two phases.

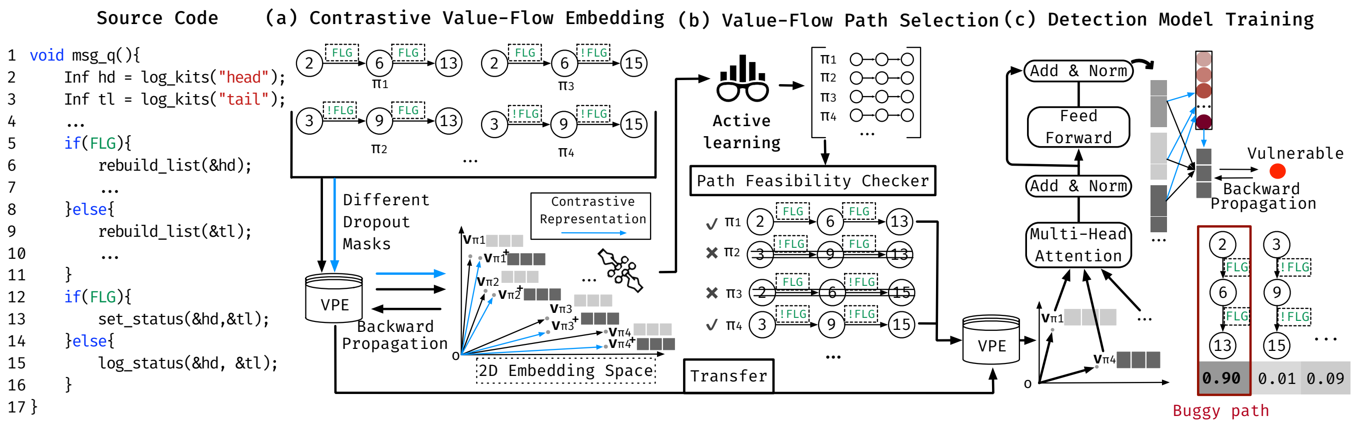
***(b) Value-Flow Path Selection.*** This phase is designed to precisely select feasible and representative value-flow paths to represent the code fragment to support fast training of the path-based detection model. We first use the pretrained VPE from Phase (a) to generate the feature vectors of the input paths, and sample the paths using self-supervised active learning [45] to capture the most representative paths and make the embedding diverse and informative. We then perform path-sensitive code embedding given the sampled paths as a reachability problem over the guarded value-flow graph (VFG) [12, 64]. The VFG captures def-use relations in a sparse manner and guards edges with annotations describing control-flow transfer conditions. The feasibility checking is then reduced to a reachability problem over the guarded VFG, to embed only feasible paths in the low-dimensional embedding space.

***(c) Detection Model Training.*** Given the selected paths produced by Phase (b) and the VPE model transferred from Phase (a), this phase will train a precise detection model by only using the selected paths of a program and its label (i.e., vulnerable or safe). We first obtain the embedding vector from VPE for each selected value-flow path, and then leverage the transformer architecture [69] to produce a contextual vector for each path to capture the interaction between paths. These vectors are then fed into a soft attention layer [1] to score and aggregate them into one vector for the final detection model training. The model can also interpret important value-flow paths and statements according to their contribution to the model outputs, which is ranked by the learned attention scores.

Once the model is trained, in the prediction process, we apply the model to detect potential vulnerabilities in an unseen program with the selected feasible and representative value-flow paths generated as Phase (b). For a detected vulnerability, our approach reports top-ranked value-flow paths as buggy paths according to the attention scores. Top-ranked statements are also identified as the top influential statements in each buggy path.

Our major contributions are as follows:

- We present ContraFlow, a new path-sensitive code embedding utilizing a pretrained value-flow path encoder via self-supervised contrastive learning to significantly boost the performance and reduce the training costs of later path-based prediction models to precisely pinpoint vulnerabilities.
- We formulate path-sensitive code embedding as a reachability problem over a guarded sparse value flow graph to



```
1  void msg_q(){
2      Inf hd = log_kits("head");
3      Inf tl = log_kits("tail");
4      ...
5      if(FLG){
6          rebuild_list(&hd);
7          ...
8      }else{
9          rebuild_list(&tl);
10         ...
11     }
12     if(FLG){
13         set_status(&hd,&tl);
14     }else{
15         log_status(&hd, &tl);
16     }
17 }
```

**Figure 2: A motivating example extracted from a real-world project POCO (a library for network-based applications). It illustrates and details the three phases of our framework described in Figure 1.**

support lightweight yet precise embedding of feasible paths in the low-dimensional embedding space.

- We evaluated ContraFlow by comparing it with eight state-of-the-art learning-based vulnerability detection approaches. Experimental results show that our approach outperforms the method/slice-level baselines by up to 334.1% informedness, 317.9% markedness, 58.3% F1 Score, and up to 450.0% mean statement recall, 192.3% mean statement precision and 450.0% mean IoU in terms of locating buggy statements.

## 2 A MOTIVATING EXAMPLE

Figure 2 illustrates the key idea of ContraFlow by going through the three phases in Figure 1 using a Business Logic Error (CWE-840) [55] extracted from a real-world project POCO (a library for network applications) [31]. The vulnerability is caused by the API misuse when calling `set_status(&hd)` after `rebuild_list(&hd)`, where hd is first defined at line ②, then modified at line ⑥ and used at line ⑬. This buggy value-flow path of hd can cause unexpected behavior and result in denial of service.

Note that the value-flow paths of different variables extracted from the original code fragment are large in size and contain many paths including infeasible or bug irrelevant paths needed for feasibility checks and embedding. Our contrastive value-flow embedding in Phase (a) first pretrains a VPE, to preserve the semantics of paths (e.g., $\pi_1 - \pi_4$) in the latent space, and then selects the most representative paths using active learning in Phase (b), followed by feasibility checking which removes infeasible paths $\pi_2$ and $\pi_3$ through sparse and guarded value-flow analysis. Phase (c) further fine-tunes and interprets $\pi_1$ as the likely buggy path according to the ranked attention scores (90% for $\pi_1$) of the training model.

***(a) Contrastive Value-Flow Embedding.*** As shown in Figure 2 (a), the input of this phase is a bag of value-flow paths extracted from the code fragment, e.g., $\pi_1$ (②→⑥→⑬), $\pi_2$ (③→⑨→⑬), $\pi_3$ (②→⑥→⑮) and $\pi_4$ (③→⑨→⑮). We feed them into the value-flow path encoder (VPE) twice using different dropout masks [61] in VPE, to obtain their vector representations, e.g., $\mathbf{v}_{\pi_1}$, $\mathbf{v}_{\pi_2}$, $\mathbf{v}_{\pi_3}$ and $\mathbf{v}_{\pi_4}$, and their corresponding contrastive representations [26], e.g., $\mathbf{v}^+_{\pi_1}$, $\mathbf{v}^+_{\pi_2}$, $\mathbf{v}^+_{\pi_3}$ and $\mathbf{v}^+_{\pi_4}$. VPE is pretrained using contrastive learning to capture the semantics of value-flow paths such that the pretrained similar embedding vectors (e.g., $\mathbf{v}_{\pi_1}$ and $\mathbf{v}^+_{\pi_1}$) stay close to each other while dissimilar pairs (e.g., $\mathbf{v}_{\pi_1}$ and $\mathbf{v}^+_{\pi_3}$) are far apart, as the two-dimensional feature space depicted in Figure 2(a). The parameters of VPE are automatically updated in the backward propagation process [33] by minimizing the NCE loss [9] that encodes the similarities between value-flow embedding vectors.

***(b) Value-Flow Path Selection.*** This phase uses the pretrained VPE from Phase (a) to transform value-flow paths into embedding vectors. After this, we sample a proportion of representative value-flow paths, e.g., $\pi_1 - \pi_4$, according to the rankings learned from self-supervised active learning [45]. These paths are further fed into the path-feasibility checking to remove infeasible value-flow paths. For instance, path ③→⑨→⑬ is infeasible because the control flow guards `!FLG` at ③→⑨ and `FLG` at ⑨→⑬ contradict with each other. Likewise, ②→⑥→⑮ is also infeasible. Finally, only feasible value-flow paths $\pi_1$ and $\pi_4$ are preserved to train the detection model in Phase (c).

***(c) Detection Model Training.*** The input of this phase is the selected feasible and representative value-flow paths produced by Phase (b), which are first transformed into vectors using the VPE model transferred from Phase (a). These embedding vectors then go through a transformer architecture [69] to produce a contextual vector for each value-flow path. For example, the contextual vector of $\pi_1$ is computed by attention with the other vectors, e.g., $\pi_4$, to add their influence on $\pi_1$. After this, a soft attention layer [1] is applied to merge these contextual vectors into one vector, which is used for training the detection model. The ranked attention weights indicate the contribution of different value-flow paths to the model output. For instance, the value-flow path $\pi_1$ (②→⑥→⑬) poses the highest attention weights (90%) while the others are negligible, indicating that this value-flow path is likely a buggy path.

Let us demonstrate that our approach can embed more precisely the structural code information than path-unaware learning-based models by comparing ContraFlow with two recent approaches VulDeeLocator [49] and IVDetect [48]. VulDeeLocator and its previous work VulDeePecker [51] utilize a slicing criteria traversing forward and backward from a user-specified API call to obtain a subgraph on a program dependence graph (PDG). The resulting subgraph merges multiple infeasible/feasible paths. In our example, performing slicing given an API function, e.g., `set_status` will

**Algorithm 1:** Contrastive Value-Flow Embedding

```
1 for $epoch \leftarrow \{1, 2, ...\}$ do
2     Generate contrastive vector representations using data
        augmentation [26]
3     Compute the contrastive loss $\mathcal{L}$ with Equation 3
4     Update parameters by applying stochastic gradient
        ascent to minimize $\mathcal{L}$
5 return Well-trained VPE
```

produce a subgraph containing ②, ③, ⑤, ⑥, ⑨, ⑫, ⑬. It neither distinguishes value-flows of different variables nor removes infeasible paths. IVDetect [48] performs edge masking to identify important edges on the PDG based on the classification model trained with GNNs. However, the approach is unaware of infeasible program dependence (e.g., ⑨→⑬) and the resulting masked graph can contain disconnected value-flows which are insufficient or incomplete when reporting buggy paths. Section 4 provides more evaluations to compare ContraFlow with the state-of-the-arts.

## 3 CONTRAFLOW APPROACH

This section details the three phases of our approach, namely contrastive value-flow embedding (Section 3.1), value-flow path selection (Section 3.2) and detection model training (Section 3.3).

### 3.1 Contrastive Value-Flow Embedding

The goal of our contrastive value-flow embedding is to pretrain a value-flow embedding model, aka value-flow path encoder (VPE), from the value-flow paths extracted from the code without the need for manual labels, to support value-flow path selection (Section 3.2) and to train the detection model (Section 3.3). We first describe the contrastive value-flow embedding algorithm in Section 3.1.1 and then elaborate the design of VPE in Section 3.1.2.

*3.1.1 Contrastive Value-Flow Embedding Algorithm.* Contrastive value-flow embedding aims to learn discriminative vector representations $\mathbf{v}_\pi$ of similar/dissimilar guarded value-flow paths $\pi$ extracted from unlabeled code fragments by pretraining the VPE. A guarded value-flow path $\pi$ consists of a sequence of program statements representing a def-use chain between variables, with the guard on each edge between two statements to indicate control-flow transfer conditions [12, 64]. The guards will be used during path-feasibility solving in Phase (b). Algorithm 1 summarizes the learning algorithm. For each learning epoch, we generate contrastive vector representations (Line 2) and compute the contrastive loss amongst contrastive value-flow paths (Line 3). The parameters of VPE are automatically updated in the training process (Line 4). The following paragraphs describe contrastive value-flow representations and the contrastive loss function.

***Contrastive Value-Flow Representations.*** We utilize the minimal data augmentation approach [26] to generate contrastive value-flow vector representations using independently sampled dropout masks, i.e., feeding the same value-flow path to the VPE twice with independently sampled (different) dropout masks [61], to produce an embedding pair $(\mathbf{v}_\pi, \mathbf{v}_\pi^+)$, where $\mathbf{v}_\pi^+$ stands for the contrastive representation regarding $\pi$. The embedding pair generated from

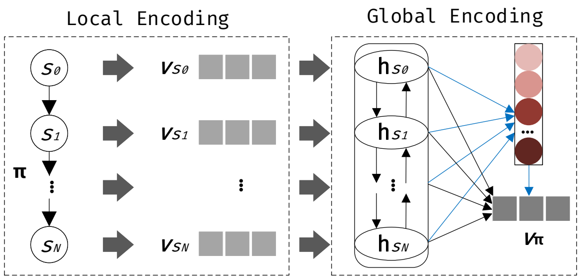


**Figure 3: Value-Flow Path Encoder.**

the same $\pi$ are considered as *positive* while that produced from different paths are treated as *negative*. Note that, this augmentation approach largely outperforms common data augmentation techniques (e.g., element deletion or substitution [52]) because it can avoid potential representation collapse [26].

***Contrastive Value-Flow Embedding Loss.*** We use the Noise Contrastive Estimate (NCE) loss function [9] as the learning objective to maximize the agreement between positive value-flow representations. For two value-flow path vectors $\mathbf{v}_{\pi_i}$ and $\mathbf{v}_{\pi_j}$, we measure their similarity using cosine similarity:

$$sim(\mathbf{v}_{\pi_i}, \mathbf{v}_{\pi_j}) = \frac{\mathbf{v}_{\pi_i}^\top \mathbf{v}_{\pi_j}}{||\mathbf{v}_{\pi_i}|| \cdot ||\mathbf{v}_{\pi_j}||} \tag{1}$$

The loss of $\pi_i$ is defined as :

$$loss(\pi_i) = -log\frac{exp(sim(\mathbf{v}_{\pi_i}, \mathbf{v}_{\pi_i}^+))}{\sum_{k=1}^{B} exp(sim(\mathbf{v}_{\pi_i}, \mathbf{v}_{\pi_k}^+))} \tag{2}$$

where $B$ is the batch size of value-flow paths. The total value-flow contrastive loss can be computed as:

$$\mathcal{L} = \frac{1}{B}\sum_{i=1}^{B} loss(\pi_i) \tag{3}$$

During pretraining, the vector representations of semantically similar paths produced by the pretrained VPE stay close to each other as the contrastive loss $\mathcal{L}$ diminishes through backward propagation to a minimum stationary point [40]. Once the VPE is trained, we use it to guide value-flow path selection in Section 3.2 and transfer it to the detection model for fine-tuning in Section 3.3.

*3.1.2 Value-Flow Path Encoder.* To produce the feature vector $\mathbf{v}_\pi$ to represent a value-flow path $\pi$, we first introduce the processing pipeline of VPE and then elaborate the design of the statement encoder.

***Processing pipeline.*** Figure 3 shows the pipeline of VPE consisting of two steps:

- ***Local encoding:*** We apply a statement encoder to capture the syntactic structure of each statement's AST-subtree [79] of a value-flow path $\pi$. This encoder produces *local features* in the form of vector representations $\mathbf{v}_{s_0}, \mathbf{v}_{s_1}, .., \mathbf{v}_{s_N}$ to represent the sequence of statements $s_0, s_1, ..., s_N$ in $\pi$. We will detail the design of the statement encoder later in this section.
- ***Global Encoding:*** To model the sequential naturalness of the def-use chain on a value-flow path, we leverage Bidirectional Gated Recurrent Unit (BGRU) [13] to generate hidden state vectors $\mathbf{h}_{s_0}, \mathbf{h}_{s_1}, ..., \mathbf{h}_{s_N}$ from $\mathbf{v}_{s_0}, \mathbf{v}_{s_1}, .., \mathbf{v}_{s_N}$. These generated vectors encode the *global features* of a value-flow path, i.e., def-use feature

propagation between statements. We then add weights to each $\mathbf{h}_s$ and merge them with the standard attention mechanism [1], to produce a fixed length vector $\mathbf{v}_\pi$ representing $\pi$'s semantics.

***Statement Encoder.*** Each statement $s$ is represented as an AST-subtree, a subtree of the AST at the statement level [79]. The statement encoder aims to model the in-depth feature of $s$ including the AST node tokens, their types, and the syntactic structure (AST edges). This can extract more fine-grained and holistic features by perceiving syntax-level code elements, making the local encoding of a value-flow path more precise. We first compute the feature vector $\mathbf{v}_{n_i}$ of each node $n_i$ in the AST-subtree (Equation 4) by considering node properties and the tree structure. We then aggregate the vectors of all nodes to produce vector $\mathbf{v}_{sm}$ to represent a summary of the tree (Equation 6). Finally we merge all the summary vectors [73], and pass it to a fully connected layer and a dropout layer to produce $s$'s feature vector $\mathbf{v}_s$.

Let $C_i$ denote the children of node $n_i$ in the AST-subtree. Each node has two properties: the node type and node tokens. We first obtain the property embedding vectors by looking up the embedding matrices, where each row represents an embedding vector in relation to a certain object [1]. We then use the element-wise sum to produce $n_i$'s initial vector $\mathbf{v}_{n_i}$, which is then updated to $\mathbf{v}'_{n_i}$ by considering its children's features. Formally,

$$\mathbf{v}'_{n_i} = \sum_{j \in C_i \cup \{i\}} \alpha_{ij} \cdot \mathbf{W}\mathbf{v}_{n_j} \tag{4}$$

where $\mathbf{W}$ is a weighted matrix. $\alpha_{ij}$ means the attention weights between nodes $n_i$ and $n_j$, and is computed as:

$$\begin{aligned} \alpha_{ij} &= \frac{exp(\sigma(e_{ij}))}{\sum_{k \in C_i \cup \{i\}} exp(\sigma(e_{ik}))} \\ e_{ij} &= \mathbf{a}_s^\top [\mathbf{W}^a \mathbf{v}_{n_i} || \mathbf{W}^a \mathbf{v}_{n_j}] \cdot \sigma((\mathbf{W}^a \mathbf{v}_{n_i})^\top (\mathbf{W}^a \mathbf{v}_{n_j})) \end{aligned} \tag{5}$$

Here $\mathbf{a}_s$ is a global learnable attention vector, $\mathbf{W}^a$ is a weighted matrix, $||$ represents concatenation and $\sigma(\cdot)$ denotes the activation function. Finally, we use the concatenation of the element-wise mean and max pooling to aggregate the vectors of all nodes (assuming that we have $N$ nodes) and generate the summary vector:

$$\mathbf{v}_{sm} = [\frac{1}{N} \sum_{i=1}^{N} \mathbf{v}'_{n_i} || max_{j=1}^{N} \mathbf{v}'_{n_j}] \tag{6}$$

Note that there can be multiple processing layers (Equations 4 and 6), so a JK-net architecture [73] is used to aggregate the summary vector at different scales of processing to produce the final statement vector: $\mathbf{v}_{jk} = \sum_{l=1}^{L} \mathbf{v}_{sm}^l$, where $L$ represents the number of layers and $\mathbf{v}_{sm}^l$ represents the summary vector of $s$ after layer $l$. Finally, a fully connected layer and a dropout layer [61] are applied on $\mathbf{v}_{jk}$ to produce the final statement vector $\mathbf{v}_s = dropout(\mathbf{W}\mathbf{v}_{jk})$.

**Example 1.** Figure 4 gives the AST-subtree of `rebuild_list(&hd);` at Line 6 in Figure 2 to illustrate our statement encoder. The root node $n_1$ is of type `CallExpression` and has two children, a `Callee` node $n_2$ with tokens `rebuild`, `list` and an `ArgumenList` node $n_3$ with tokens `&`, `hd`, which has its children $n_4$-$n_6$. First, we look up the type and token(s) of each node in the two embedding space (matrices) to retrieve their corresponding vectors. We then perform the element-wise sum to form one embedding vector for each node. The resulting vector is then updated according to Equation 4. For

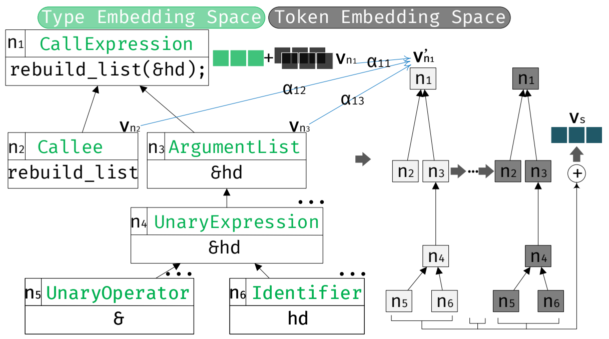


**Figure 4: An example to illustrate statement encoder.**

example, $n_1$'s vector is computed by adding attention weights from itself ($\alpha_{11} \cdot \mathbf{v}_{n_1}$) and its children $n_2$ ($\alpha_{12} \cdot \mathbf{v}_{n_2}$) and $n_3$ ($\alpha_{13} \cdot \mathbf{v}_{n_3}$). At last, all the hidden vectors of nodes are pooled as in Equation 6 and summed at different scales to produce the final statement vector $\mathbf{v}_s$.

## 3.2 Value-Flow Path Selection

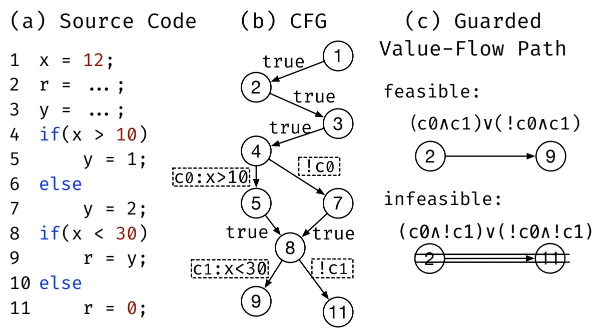
*3.2.1 Value-Flow Active Learning.* We select a portion of representative paths using a recent self-supervised active learning approach [45] such that the number of paths for code embedding is reduced while important program semantics are well-preserved. Given a set of value-flow paths $\mathbf{H} = [\mathbf{v}_{\pi_1}, ..., \mathbf{v}_{\pi_n}] \in \mathbb{R}^{n \times d}$, we aim to select a representative subset $\mathbf{H}' \in \mathbb{R}^{k \times d} \subseteq \mathbf{V}$. The approach introduces an encoder-decoder model [45] to reconstruct the input based on two reconstruction coefficient matrices $\mathbf{Q}, \mathbf{P} \in \mathbb{R}^{n \times n}$. These two matrices are trained using a joint reconstruction loss [45] by minimizing the distance between (1) $\mathbf{H}$ and the reconstructed matrix $\mathbf{QH}$, (2) the cluster centroid matrix $\mathbf{C}$, which is obtained using K-means clustering [39] on $\mathbf{H}$, and the reconstructed matrix $\mathbf{PH}$, and (3) $\mathbf{H}$ and the decoder outputs $\mathbf{G}$. After training, we calculate the $l_2$-norm [72] for each column of $\mathbf{Q}$ and $\mathbf{P}$, and normalize the value into $[0, 1]$, to produce two ranking vectors $\hat{\mathbf{q}}, \hat{\mathbf{p}} \in \mathbb{R}^n$. Finally, $\hat{\mathbf{q}}$ and $\hat{\mathbf{p}}$ are merged and sorted in descending order, and the top-$k$ value-flow paths are the most representative, which are fed into the next phase for feasibility analyses.

*3.2.2 Value-Flow Path Feasibility Analysis.* We then perform path-sensitive analysis for the value-flow paths selected using the aforementioned active learning to further refine the value-flows for precise code embedding. Feasibility checking is reduced to a reachability problem over the guarded value-flow graph [12, 64] whose edges are annotated with guards to describe control-flow transfer conditions on the control-flow graph (CFG).

Given a value-flow path $\pi : s_0, s_1, \ldots, s_N$, we define $guard_v(\pi)$, a Boolean function encoding the reachability of the control-flow paths between each consecutive pair along the value-flow path in the program, from $s_0$ to $s_N$. Hence, $\pi$ is feasible if $guard_v(\pi)$ returns true, and infeasible otherwise.

$$guard_v(\pi) = \bigwedge_{i=0}^{N-1} \bigvee_{p \in CP(s_i, s_{i+1})} \bigwedge_{e \in CE(p)} guard_e(e) \tag{7}$$

where $CP(s_i, s_{i+1})$ denotes all the control flow paths between $s_i$ and $s_{i+1}$, $CE(p)$ represents the control-flow edges in $p$ and $guard_e(e)$ denotes the control-flow transfer condition for edge $e \in CE(p)$. A



```
1  x = 12;
2  r = ...;
3  y = ...;
4  if(x > 10)
5      y = 1;
6  else
7      y = 2;
8  if(x < 30)
9      r = y;
10 else
11     r = 0;
```

**Figure 5: An example of guarded value-flow analysis.**

guard between two nodes (e.g., $s_i$ and $s_{i+1}$) on the value-flow path is computed based on the disjunctions of the guards of each control-flow path $p$ on the CFG. For a conditional control-flow transfer (e.g., if-else branch), we encode the branch condition $c$ as a unique Boolean guard for the true branch and $!c$ for the else branch. An unconditional control-flow transfer is assigned a true value.

**Example 2.** Figure 5 gives an example of our feasibility analysis of the value-flow path from ② to ⑨ and ② to ⑪. For the first path, as shown on the CFG in Figure 5(b), there are two control flow paths in $CP$(②, ⑨), i.e., $p_1$ : ⟨②, ③, ④, ⑤, ⑧, ⑨⟩ and $p_2$ : ⟨②, ③, ④, ⑦, ⑧, ⑨⟩. Thus the feasibility of $\pi$ is determined by the disjunction of $\bigwedge_{e \in CE(p_1)} guard_e(e)$ and $\bigwedge_{e \in CE(p_2)} guard_e(e)$ according to Equation 7, i.e., $(c_0 \wedge c_1) \vee (!c_0 \wedge c_1)$, where $c_0 : x > 10$ is the guard at ④ to ⑤ and $c_1 : x < 30$ is the guard from ⑧ to ⑨. Hence, ② to ⑨ is feasible because $c_1$ is a true value. In contrast, the value-flow path from ② to ⑪ is an infeasible one because $!c_1$ is a false value, thus this path is excluded for code embedding.

### 3.3 Detection Model Training

Our model training requires labeled code fragments (i.e., vulnerable or safe) with their selected feasible value-flow paths (Section 3.2). Each code fragment and its selected paths $\pi_1, \pi_2, ..., \pi_N$ are first fed into the VPE transferred from Section 3.1.2 and go through a multi-head self-attention layer as in Equation 8. A soft attention [1] is then used to aggregate them into one vector, which is used for classification (Equation 9) to predicted buggy paths.

Specifically, we first obtain the embedding matrices of each value-flow path from VPE, $\mathbf{V} = [\mathbf{v}_{\pi_1}, \mathbf{v}_{\pi_2}, ..., \mathbf{v}_{\pi_N}] \in \mathbb{R}^{N \times d_{embed}}$, where each row $\mathbf{v}_\pi$ denotes the feature vector (Section 3.1.2) of a selected $\pi$. $N$ represents the number of value-flow paths and $d_{embed}$ means the embedding dimension of the value-flow paths. We first pass $\mathbf{V}$ into a multi-head self-attention layer [69] to produce a contextual feature matrix $\mathbf{V}' \in \mathbb{R}^{N \times d_{ctx}}$:

$$\begin{aligned} \mathbf{V}' &= [\mathbf{h}_1 || ... || \mathbf{h}_h] \mathbf{W}^o \\ \mathbf{h}_i &= Attn(\mathbf{V}\mathbf{W}_i^Q, \mathbf{V}\mathbf{W}_i^K)(\mathbf{V}\mathbf{W}_i^V) \end{aligned} \tag{8}$$

Here, $Attn(\mathbf{Q}, \mathbf{K}) = softmax(norm(\mathbf{Q}\mathbf{K}^\top))$ stands for a $N \times N$ attention weights matrix, each row of which represents the attention weights for its corresponding value-flow path. $\mathbf{W}^o \in \mathbb{R}^{h \cdot d_{hid} \times d_{ctx}}$, $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V \in \mathbb{R}^{d_{embed} \times d_{hid}}$ are learnable weight matrices. $h$ denotes the attention head number and $d_{hid}$ is the hidden dimension.

**Table 1: Labeled sample Distribution.**

| Dataset | granularity | # Vulnerable | # Safe | # Total |
|---|---|---|---|---|
| D2A | Method | 21,396 | 2,194,592 | 2,215,988 |
| | Slice | 105,973 | 10,983,992 | 11,089,965 |
| Fan | Method | 8,456 | 142,853 | 151,309 |
| | Slice | 42,527 | 713,239 | 717,496 |
| FQ | Method | 8,923 | 9,845 | 18,768 |
| | Slice | 45,627 | 50,125 | 95,752 |
| **Total** | Method | 38,775 | 2,347,290 | 2,386,065 |
| | Slice | 194,127 | 11,747,356 | 11,903,213 |

Finally, we use a global learnable attention vector $\mathbf{a}_c$ to compute the attention weights $\alpha_i^c = \frac{exp(\mathbf{v}_{\pi_i}^\top \mathbf{a}_c)}{\sum_{j=1}^N exp(\mathbf{v}_{\pi_j}^\top \mathbf{a}_c)}$ for $\pi_i$ and merge them into a code vector $\mathbf{v}_c = \sum_{i=1}^N \alpha_i^c \cdot \mathbf{v}_{\pi_i}$ which is used for the final prediction through learning the label distribution conditioned on the code $c$, i.e., $P(y_i|c)$, where $y_i$ is one of the vulnerability tag sets $Y$. The predicted distribution $q(y_i)$ is computed using a softmax function, the dot product between the code vector $\mathbf{v}_c$ and the vector representation $\mathbf{lb}_i$ of each label $y_i \in Y$:

$$\forall\, y_i \in Y : q(y_i) = \frac{exp(\mathbf{v}_c^\top \mathbf{lb}_i)}{\sum_{y_j \in Y} exp(\mathbf{v}_c^\top \mathbf{lb}_j)} \tag{9}$$

To interpret the vulnerability at the statement level, the buggy path set is obtained with top-$k$ indexing via attention weights $\alpha_i^c$. For each buggy path we use top-$k$ indexing based on the attention weights of statements (Section 3.1.2) to obtain important statements.

## 4 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of ContraFlow at detecting vulnerabilities in real-world projects by comparing it against eight state-of-the-art learning-based approaches. ContraFlow improves up to 334.1%, 317.9%, 58.3% informedness, markedness and F1 Score, respectively, and up to 450.0%, 192.3%, 450.0% mean statement recall, mean statement precision and mean IoU, respectively.

### 4.1 Datasets and Implementation

***Datasets.*** We evaluate ContraFlow using 288 real-world open-source projects extracted from three datasets, namely D2A [80], Fan [20] and FFMPeg+Qemu (FQ) [81], consisting of 275K programs with 30M lines of code. Note that our datasets are built upon real-world vulnerabilities: Fan is built upon CVEs [20], and FQ and D2A comprise vulnerabilities labeled by domain experts [80, 81]. We label a sample as vulnerable if it contains at least one vulnerable statement, and safe otherwise. For D2A, the bug traces (paths) are already labeled for vulnerable statements. For Fan and FQ, we label buggy statements based on the code fragment before and after its bug-fixing patch on GitHub. The labeled method/slice distribution is presented in Table 1. The average number of paths produced by SVF is 453. We perform evaluations across projects (training on some projects and validation and detection on different projects) by randomly splitting the projects of our datasets into 80%, 10% and 10% for training, validation and detection respectively. We also compared the mixture of training, validation and detection on all the projects. We utilize SMOTE [8] to alleviate the data imbalance problem during training, while the ratios of the data are unchanged for validation and detection.

***Implementation.*** The interprocedural value-flow graph of a program is generated by SVF [63]. We use Z3 SMT solver [15] to solve path feasibility on the guarded sparse VFG. We use `clang` to compile each file of a project into an LLVM bitcode file, which is then fed into SVF to extract the VFG (Note that we extract buggy samples by compiling different buggy versions of a project based on its GitHub commits). We use JOERN [17] to generate the AST-subtree of statements. The neural networks are implemented on top of PyTorch Lightning [19] and PyTorch Geometric [24]. The experiments are conducted on an Ubuntu 18.04 server with an NVIDIA GeForce GTX 1080 GPU and an Intel Xeon E5-1620 3.50GHz CPU with 512 GB memory. The neural network is trained in a batch-wise fashion with a batch size of 64. The embedding dimension is set to 128. The number of selected paths is set to a maximum of 100. The dropout rate is set to 0.1. We use stochastic gradient descent [40] to train the neural network with a learning rate of 0.002 for a maximum number of training epoch with 500. We utilize AutoML [30] to automatically tune the hyper-parameters of our model.

For the open-source approaches (SySeVR [50], Reveal [7], DeepWukong [10], IVDetect [48] and VulDeeLocator [49]), we directly use their shared implementations. For the implementations of VGDetector [11], Devign [81] and VulDeePecker [51] which are not publicly available, we re-implemented them by strictly following their methods elaborated in the original papers. We do not compare ContraFlow with non-learning-based detectors, e.g., Checkmarx [35] and Infer [18], because our baselines (i.e., VulDeeLocator [49], VGDetector [11], SySeVR [50] and DeepWukong [10]) already report fewer false alarms and false negatives than these non-learning-based static analyzers, which are only effective for specific vulnerability types (e.g., use-after-frees and null-dereference) with analysis rules predefined by domain experts [10, 48, 51].

## 4.2 Research Questions

Our evaluation aims to answer the following research questions:

RQ1 **Can ContraFlow outperform existing learning-based vulnerability detection approaches?** We would like to investigate (RQ1.1) whether ContraFlow outperforms existing method/slice-level vulnerability detectors, and (RQ1.2) whether ContraFlow achieves consistently better performance in terms of locating buggy statements [49].

RQ2 **How do different settings affect ContraFlow's overall performance?** We conduct ablation analysis to understand the influence of different components of ContraFlow, including (RQ2.1) the performance with and without contrastive learning, (RQ2.2) the impact of path selection strategies (active learning or random sampling, path sensitive or insensitive), and (RQ2.3) the effectiveness under different pretrainings (CodeBert [22], BLSTM [29], BGRU [13]).

RQ3 **How do different dataset scales for training affect the performance of ContraFlow?** We perform data sensitive analysis to understand the performance with and without contrastive learning under different dataset sizes.

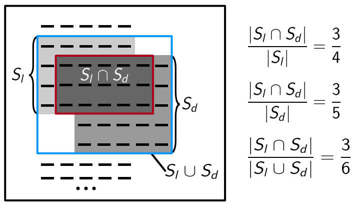

$S_l$
$S_l \cap S_d$
$S_d$
$S_l \cup S_d$
...
$\frac{|S_l \cap S_d|}{|S_l|} = \frac{3}{4}$
$\frac{|S_l \cap S_d|}{|S_d|} = \frac{3}{5}$
$\frac{|S_l \cap S_d|}{|S_l \cup S_d|} = \frac{3}{6}$

**Figure 6: An example to illustrate LBS metrics. Each dashed line represents a statement.**

## 4.3 Evaluation Methodology

Existing learning-based detectors can be generally summarized into two categories (1) method-level detection, i.e., predicting whether a program method is vulnerable or not, such as VGDetector [11], Devign [81], Reveal [7], and (2) slice-level detection, i.e., predicting whether a program slice (e.g., a subgraph of PDG) is vulnerable or not, including VulDeePecker [51], SySeVR [50], DeepWukong [10], VulDeeLocator [49] and IVDetect [48].

ContraFlow provides a fine-grained path-based model which can report the potentially buggy value-flow paths and statements of a vulnerable program. Due to different detection granularities, we perform a fair cross-comparison between ContraFlow and the above approaches under their metrics, aiming to show that ContraFlow still performs better than current approaches under the following existing metrics [48, 49]:

- **Informedness (IF)** is an unbiased variant of Recall and TNR (proportion of predicted truly safe samples) and is calculated as IF = Recall + TNR − 1.
- **Markedness (MK)** is an unbiased variant of Precision and TNA (correctness of predicted safe samples) and is computed as MK = Precision + TNA − 1.
- **F1 Score (F1)** means the overall effectiveness, which is the harmonic mean of Recall and Precision: $\text{F1} = 2 * \frac{\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}}$.

Here, $\#TP$ ($\#FP$) denotes the number of vulnerable (safe) methods/slices correctly (incorrectly) detected as vulnerable. $\#TN$ ($\#FN$) denotes the number of safe (vulnerable) methods/slices correctly (incorrectly) predicted as safe. **Recall** $= \frac{\#TP}{\#TP+\#FN}$ denotes the proportion of detected truly vulnerable samples in all vulnerable samples. **Precision** $= \frac{\#TP}{\#TP+\#FP}$ is the correctness of detected vulnerable samples.

We use the above metrics to compare with method- and slice-level approaches. Our approach predicts a method as vulnerable if ContraFlow finds one buggy path contained in the method. Similarly, we predict a slice as vulnerable if the slice and our reported vulnerable value-flow paths have an overlapping statement.

We further evaluate ContraFlow against IVDetect and VulDeeLocator based on the same metrics used in VulDeeLocator [49] to quantitatively compare the effectiveness of locating buggy statements (LBS), which is inspired by the Jaccard Index [36]. For each correctly detected vulnerable sample, let $S_l$ denote the set of labeled vulnerable statements and $S_d$ denote the set of reported vulnerable statements. The LBS metrics are defined as follows:

- **Mean Statement Recall (MSR)** $\text{MSR} = \frac{1}{N}\sum_{i=1}^{N} \text{SR}_\text{i}$ where $\text{SR} = \frac{|S_l \cap S_d|}{|S_l|}$ denoting the proportion of detected vulnerable statements in labeled vulnerable statements.

**Table 2: Comparison of method- and slice-level approaches under informedness (IF), markedness (MK), F1 Score (F1), Precision (P) and Recall (R). ContraFlow-method/slice denotes the evaluation at method- and slice-level respectively.**

| Model Name | IF (%) | MK (%) | F1 (%) | P (%) | R (%) |
|---|---|---|---|---|---|
| VGDetector | 31.1 | 29.3 | 56.7 | 52.6 | 61.4 |
| Devign | 30.1 | 28.8 | 58.7 | 54.6 | 63.4 |
| Reveal | 34.2 | 33.8 | 63.4 | 61.5 | 65.5 |
| **ContraFlow-method** | **60.3** | **58.2** | **75.3** | **71.5** | **79.4** |
| VulDeePecker | 17.3 | 17.3 | 52.3 | 52.2 | 52.4 |
| SySeVR | 24.3 | 24.2 | 55.0 | 54.5 | 55.4 |
| DeepWukong | 48.1 | 48.4 | 67.0 | 67.4 | 66.5 |
| VulDeeLocator | 38.4 | 38.1 | 62.0 | 61.4 | 62.5 |
| IVDetect | 37.4 | 37.3 | 64.1 | 64.0 | 64.6 |
| **ContraFlow-slice** | **75.1** | **72.3** | **82.8** | **79.5** | **86.4** |

- **Mean Statement Precision (MSP)** MSP $= \frac{1}{N}\sum_{i=1}^{N} \mathrm{SP_i}$ where SP $= \frac{|S_l \cap S_d|}{|S_d|}$, which stands for the locating precision of detected vulnerable statements.
- **Mean Intersection over Union (MIoU)** MIoU $= \frac{1}{N}\sum_{i=1}^{N} \mathrm{IoU_i}$ where IoU $= \frac{|S_l \cap S_d|}{|S_l \cup S_d|}$, which reflects the overlap degree of detected vulnerable statements and labeled vulnerable ones.

**Example 3.** Figure 6 illustrates the meaning of the LBS metrics. In this example, $S_l$ contains 4 statements (the number of dashed lines) and $S_d$ contains 5. They have a total of $|S_d \cup S_l| = 6$ statements and an overlapping of $|S_d \cap S_l| = 3$ statements. Note that we only have one sample here. Thus MSR = 3/4 (the proportion of labeled vulnerable statements that a detection model can successfully report). MSP = 3/5 (how precise a model is at locating vulnerable statements). MIoU = 3/6 (an overall degree at which the detected statements overlap with the labeled statements). The closer MSR, MSP, MIoU are to 1, the better the performance is in terms of locating buggy statements.

***Efficiency.*** It takes about 72 hours to pre-train the value-flow path encoder using contrastive learning. For detection model training (i.e., fine-tuning), it takes much less time at approximately 18 and 34 hours to train ContraFlow-method and ContraFlow-slice respectively. The vulnerability prediction phase is very fast with an average prediction rate at about 10 methods/sec and 25 slices/sec respectively.

## 4.4 Comparison with State-of-the-Arts (RQ1)

*4.4.1 Method- and Slice-Level Comparison (RQ1.1).* Table 2 compares the results using method- and slice-level metrics in Section 4.3. Figure 7 presents the average ranking [48] comparison results. It is clear that ContraFlow outperforms both our method- and slice-level baselines under the existing metrics, including IF, MK, F1, precision, recall, AVR and ASR.

***Results.*** As shown in Table 2, ContraFlow outperforms all our baselines with an average improvement of F1 Score at around 22.9% (e.g., 32.8% for VGDetector and 58.3% for VulDeePecker), indicating an overall better effectiveness for vulnerability detection. The informedness of ContraFlow is the largest at 75.1%, which is more than double that of IVDetect at 37.4% and over four times the number of VulDeePecker at 17.3%. ContraFlow

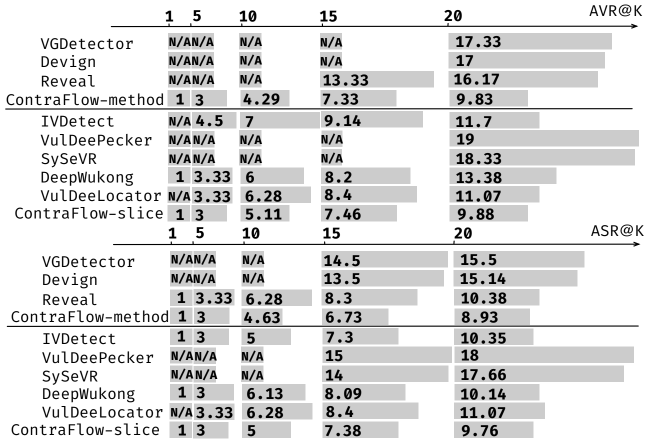


**Figure 7: Comparison with IVDetect and VulDeeLocator under AVR@k (ASR@k) [48]. AVR@k (ASR@k) represents the average top-k ranking of the correctly predicted vulnerable (safe) samples. N/A means that there is no correctly predicted sample in the top-ranked list.**

also has a significantly higher markedness at 72.3% compared to 37.3% for IVDetect and merely 17.3% for VulDeePecker. A higher informedness means that our tool can correctly recall more vulnerabilities and safe programs, and a higher markedness demonstrates that the prediction result of ContraFlow is more trustworthy. The precision and recall of ContraFlow are also better than the other eight approaches by 18.0–52.3% in precision and 29.9–64.9% in recall. Fig. 7 compares the AVR@{1-20} and ASR@{1-20} of these approaches. It is clear that ContraFlow achieves the best average vulnerable and safe rankings across all the ranges. Notably, the AVR@1 for most of our baselines (7/8) is not applicable (N/A) since their first-ranked detected method/slice (i.e., most confident prediction) is a false positive.

***Analysis.*** The reason for the better performance of ContraFlow is that it can preserve more comprehensive features of the input program by considering path-sensitive value-flow paths, which approximate the program runtime behaviour and bug semantics. The other approaches either utilize textual representation [49–51] or graph embedding [7, 11, 48, 81], which is insufficient for distinguishing feasible/infeasible or buggy/safe value-flow (program dependence) paths. Therefore, the learned vulnerable/safe code pattern is not as precise as ContraFlow.

*4.4.2 Locating-Buggy-Statements Comparison (RQ1.2).* IVDetect, ContraFlow and VulDeeLocator can pinpoint and report different lengths of vulnerable statements (LOS). Figure 8 and Figure 9 present the experimental results for LBS under different LOS. Table 3 further compares these approaches using SA, MFR and MAR [48]. Overall, ContraFlow is superior to the other two approaches (IVDetect and VulDeeLocator) in terms of all the Locating-Buggy-Statements (LBS) metrics and SA, MFR and MAR.

***MSR and MSP.*** It is clear that MSR increases dramatically from 1 LOS to 4 LOS and still rises steadily during the rest range of LOS, reaching an upper bound, i.e., reporting all the embedded statements, at about 89.5% for VulDeeLocator and ContraFlow,

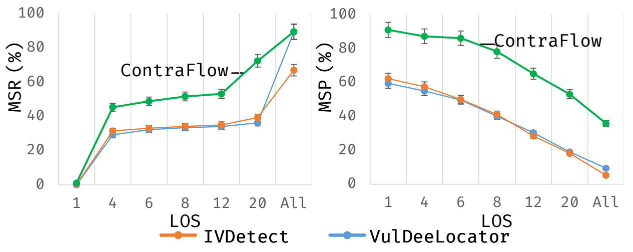


**Figure 8: MSR and MSP under different LOSs.**

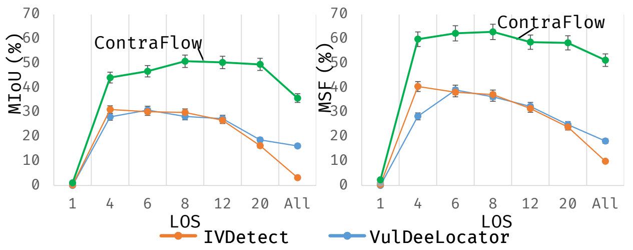


**Figure 9: MIoU and MSF under different LOSs. MSF is the harmonic mean of MSP and MSR.**

**Table 3: Comparison with IVDetect and VulDeeLocator under SA, MFR and MAR [48]. Statement Accuracy (SA) counts a correct detection if one labeled vulnerable statement is reported. MFR/MAR are the mean value of the first/average ranks of correctly detected statements.**

| Model Name | SA(%) | | | | MFR | MAR |
|---|---|---|---|---|---|---|
| | 1 LOS | 4 LOS | 6 LOS | 12 LOS | | |
| VulDeeLocator | 1.3 | 46.7 | 50.2 | 54.4 | 6.9 | 10.5 |
| IVDetect | 2.1 | 55.5 | 59.7 | 63.5 | 6.8 | 9.5 |
| **ContraFlow** | **15.1** | **73.9** | **78.2** | **84.1** | **2.1** | **5.7** |

and around 64.5% for IVDetect. This is because more LOS is likely to cover more vulnerable statements. Particularly, ContraFlow can recall more lines of vulnerable statements than the others across all the LOS by up to 89.6%. However, more LOS observes a sharp decrease of 50.1% MSP for VulDeeLocator, which is capped at 9.5% while IVDetect has an even lower MSP at about 5.3%. In comparison, ContraFlow achieves a significantly higher MSP with 91.2% at 1 LOS and is consistently better than the other two as the LOS grows larger. This is because ContraFlow only selects feasible value-flow paths as inputs so the embedded statements contain less noise. In contrast, at the upper bound, IVDetect detects the whole method as vulnerable while VulDeeLocator reports all the slices without distinguishing infeasible/feasible paths.

***MIoU and MSF.*** As shown on Figure 9, IVDetect and VulDeeLocator report their best performance at 4 and 6 LOS respectively in terms of MIoU. However, there is still an average gap of approximately 15.4% MIoU and 24.6% MSF between these two approaches and ContraFlow. By comparison, the best performance in terms of MIoU and MSF for ContraFlow appears at about 8 LOS, meaning that ContraFlow is capable of reporting a larger proportion of vulnerable statements with fewer false positives.

***SA, MFR and MAR.*** Table 3 compares the SA, MFR and MAR [48] of these detectors (IVDetect, VulDeeLocator). We can see that ContraFlow performs better than VulDeeLocator and IVDetect in terms of these metrics. There is a notable rise from 1 LOS to 4 LOS for all the approaches. This result is also consistent with the

**Table 4: Ablation Analysis Results. ContraFlow-CodeBert/BLSTM/BGRU means ContraFlow with CodeBert/BLSTM/BGRU as the value-flow path encoder.**

| Model Name | IF (%) | MK (%) | F1 (%) | MIoU (%) | MAR |
|---|---|---|---|---|---|
| Non-contrastive | 61.3 | 57.9 | 74.2 | 40.3 | 7.8 |
| Random-sampling | 63.2 | 59.6 | 75.0 | 42.9 | 7.1 |
| Path-insensitive | 49.3 | 47.2 | 68.6 | 33.2 | 9.8 |
| ContraFlow-CodeBert | 68.3 | 63.9 | 78.0 | 45.3 | 6.4 |
| ContraFlow-BLSTM | 56.3 | 54.4 | 73.2 | 42.3 | 7.5 |
| ContraFlow-BGRU | 58.3 | 56.2 | 74.2 | 43.1 | 6.9 |
| **ContraFlow** | **75.1** | **72.3** | **82.8** | **50.9** | **5.7** |

trend of MSR in Figure 8. The SA at 12 LOS is the largest at 84.1% for ContraFlow in comparison with 54.4% for VulDeeLocator and 63.5% for IVDetect. Note that these numbers are larger because the interpretation is considered as correct as long as one labeled vulnerable statement is reported. Regarding MFR and MAR, ContraFlow improves VulDeeLocator in terms of MFR and MAR by 4.8 and 4.7 ranks and IVDetect by 2.3 and 2.2 ranks, respectively.

The above LBS comparison shows that ContraFlow can detect a larger percentage of vulnerable statements with higher precision. This can be explained by the more precise embedding for vulnerability detection used by ContraFlow (i.e. path-sensitive value-flow paths), which is closer to the runtime execution feature of programs and thus can better manifest the potential vulnerable behaviour of programs and boost vulnerability semantic comprehension.

***Cross-project vs. Mixture-of-projects.*** We also evaluated two settings for training and detection (1) cross-project evaluation (training on some projects and detection on different projects), and (2) mixture-of-projects evaluation (training and detecting on the mixture of all projects). Mixture-of-projects shows a much better result than cross-project. The F1 Score for mixture-of-projects is 10.9% higher at 92.9%. It also reports a better informedness and markedness at 83.1% and 81.2%, respectively. This is because mixing projects during evaluation can have similar code fragments of the same project both appearing in the training, validating and detecting datasets, which can unexpectedly inflate our evaluation metrics due to overfitting. Thus, cross-project is a more objective setting to avoid favouring a model undesirably due to overfitting.

## 4.5 Ablation Analysis (RQ2)

As shown in Table 4, overall, the experimental results in terms of method/slice- and LBS metrics decline to varying degrees under different variants.

*4.5.1 Contrastive Learning (RQ2.1).* All the evaluation results reduce significantly when there is no pretrained value-flow path encoder with contrastive learning. Both the informedness and markedness drop to below 62% from over 72%. The F1 Score also decreases by 15.6% to 74.2%, and MIoU records a 26.3% decline to 40.3%. This demonstrates the performance gain with contrastive learning for precise code representation (feasible value-flow paths) to successfully boost the performance of the expensive downstream fine-tuning task, i.e., vulnerability detection.

*4.5.2 Value-Flow Path Selection (RQ2.2).* Path-insensitive ContraFlow reports a significant gap of 20.6% for F1 Score and 53.3%

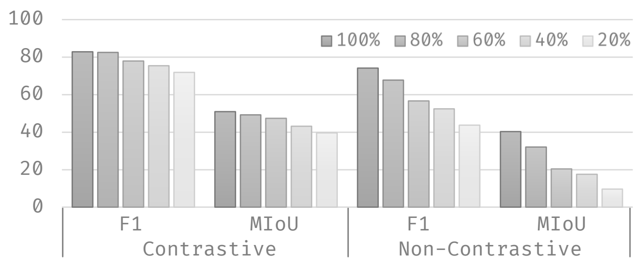


**Figure 10: Data sensitivity analysis.**

for MIoU. The informedness of path-insensitive ContraFlow is also 52.3% lower than path-sensitive ContraFlow at only 49.3%. This demonstrates the importance of preserving program execution properties (e.g., path-sensitivity) when learning to detect vulnerabilities. Regarding different sampling methods, the F1 Score of random sampling is about 10.4% smaller than active learning while the MIoU is around 18.6% smaller, showing that active learning can definitely improve the performance of our approach.

*4.5.3 Value-Flow Path Encoders (RQ2.3).* As for the performance of different encoders, CodeBert, BLSTM and BGRU perform worse than our VPE. For example, the markedness of ContraFlow-CodeBert is 63.9% (8.4% lower than ContraFlow), and ContraFlow-BLSTM has a 32.9% less markedness at 54.4%. Of these three encoders, CodeBert performs better than the other two with an improvement of about 5.8% for F1 Score, because CodeBert is built on the advanced RoBerta model [52] and can better capture the semantics of statement tokens than RNNs (BLSTM and BGRU). However, it still cannot fully capture the syntactic structure of AST-subtree and type information of an AST node, thus leaving room for improvement to represent the local features of statements.

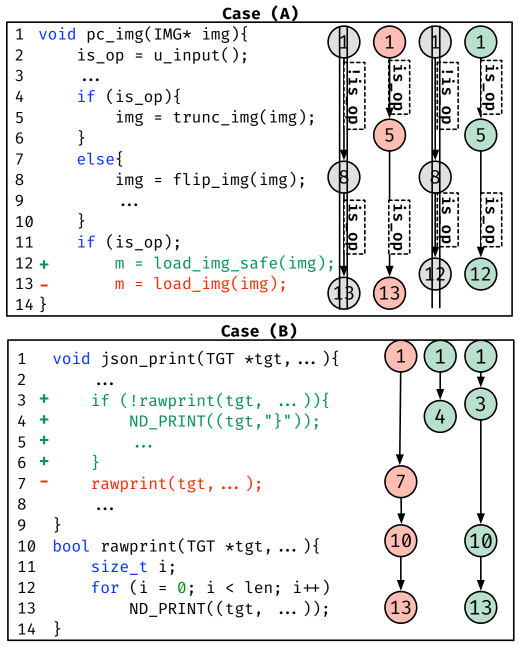
## 4.6 Data Sensitivity Analysis (RQ3)

Figure 10 presents the results of the data sensitivity analysis. There is a clear decline in both the F1 Score and MIoU when the dataset size decreases during training. For instance, the F1 Score for ContraFlow's contrastive version drops from 70.5% when using the full dataset to 59.6% when using only a fifth of the dataset. The non-contrastive version of ContraFlow records a sharper decline by nearly half of the F1 Score and more than half of MIoU, to below 10%. The reason for this is that the model with pretraining can exhibit better performance than the randomly initialized model when the size of the labeled data for fine-tuning is not sufficient. Thus, our pretraining and fine-tuning achieve consistently better performance under limited well-labeled data.

## 4.7 Case Study

Figure 11 shows two real-world code patches from the detection results of our ContraFlow, which are extracted from ImageMagick [34] and TCPDUMP [68] respectively. These cases are used to demonstrate the effectiveness of ContraFlow at reporting feasible buggy paths and important paths for bug fixing to detect real-world vulnerabilities (e.g., API misuse and inconsistent logic error) .

Figure 11 (a) shows an API misuse when calling `load_img(img)` to load an image (`img`) after truncating it with `trunc_img(img)`, which is similar to our motivating example in Fig. 2. ContraFlow



```
1  void pc_img(IMG* img){
2      is_op = u_input();
3      ...
4      if (is_op){
5          img = trunc_img(img);
6      }
7      else{
8          img = flip_img(img);
9          ...
10     }
11     if (is_op);
12 +       m = load_img_safe(img);
13 -       m = load_img(img);
14 }
```

```
1   void json_print(TGT *tgt, ... ){
2       ...
3 +     if (!rawprint(tgt,  ... )){
4 +         ND_PRINT((tgt,"}"));
5 +         ...
6 +     }
7 -     rawprint(tgt, ... );
8       ...
9   }
10  bool rawprint(TGT *tgt, ... ){
11      size_t i;
12      for (i = 0; i < len; i++)
13          ND_PRINT((tgt,  ... ));
14  }
```

**Figure 11: Important value-flow paths reported by ContraFlow are marked in red and green for pre-fix and post-fix versions respectively. Infeasible paths are highlighted in grey with strikethrough.**

can distinguish the feasible but vulnerable value-flow path (buggy path) ①→⑤→⑬, and infeasible but safe path ①→⑧→⑬. This vulnerability is fixed by replacing `load_img` with `load_img_safe`. It is interesting to find that this fix at the syntax level can be perceived by ContraFlow with local features well preserved by our value-flow path encoder (Section 3.1.2).

Figure 11 (b) presents another high-level logic error (incomplete logging). The function `json_print` calls `rawprint` to dump the cached logging file. In the pre-fix version, the log-dumping logic is incomplete due to a missing important terminator character "}" which leads to an invalid logging format and crashes the log reading program. ContraFlow's reported value-flow path ①→⑦→⑩→⑬ manifests this vulnerable behaviour, i.e., only dumping the body of the output file. This vulnerability is fixed by adding another printing operation at Line 4. Regarding this fix, ContraFlow reports two important value-flow paths, ①→③→⑩→⑬, and ①→④, showing a fixed intact data handling logic.

## 4.8 Threats to Validity

We discuss the threats to validity of ContraFlow: (1) the vulnerability labeling in existing benchmarks [20, 80, 81] might not be perfect and may contain mislabeled samples. Here, we trust the labeling results since they are labeled by domain experts [20, 81] and differential analysis [80], which runs before and after each bug-introducing commit using industrial-strength static analyzers [18]. In addition, our pretrained model with contrastive learning is robust against some mislabeled samples [16]. (2) Like [7, 48, 81], we only conduct experiments in C/C++ programs without distinguishing vulnerability types. In principle, our methodology can be adapted to other programming languages. (3) We need to compile the source code to extract more precise value-flow paths. In principle, ContraFlow accepts value-flow graphs as its inputs, which can also be produced by other tools that handle incomplete code [17]. (4) We used one typical algorithm [45] for path sampling. Though intuitive, the results demonstrate its effectiveness. It is interesting to further investigate more sampling algorithms. (5) Our feasibility checking is not full path-sensitive and resolves conditions only on the guarded VFG under the soundy assumptions [53]. We handle recursive calls and loops by bounding them to one iteration following [12, 57, 64].

## 5 RELATED WORK

***Static Vulnerability Detection.*** Static detection of vulnerabilities is a long-standing research topic with quite a number of static tools (e.g. Clang Static Analyzer [2], Infer [18], Checkmarx [35] and SVF [63]) aiming to pinpoint buggy paths of vulnerabilities by analyzing source code. Most of them [4, 44, 46, 57, 58, 65, 74–76] detect well-defined vulnerabilities like memory errors based on traditional program analysis techniques (e.g., abstract interpretation and symbolic execution), however, they require manually defined rules to detect a wider range of vulnerabilities.

***Learning-Based Vulnerability Detection.*** Recently, several studies have successfully applied learning techniques to automated vulnerability detection. They use different code representations (lexical tokens [56, 71], textual slice [49–51, 82], abstract syntax tree [81], control flow graph [11] and program dependence graph [7, 10, 48, 81]) to automatically learn vulnerability patterns under different granularities (method [7, 11, 56, 81], slice [10, 48–51, 82]). VulDeeLocator [49] and IVDetect [48] improve the detection results by performing post-processing (interpretation) on the trained detection model with the attention and edge-masking technique [78]. All the current approaches are path-unaware and learning a text/graph representation is insufficient for path-based vulnerability detection since the representation does not distinguish program paths, which is crucial for static vulnerability detection.

***Contrastive Learning.*** Being a dominant self-supervised learning approach, contrastive learning has recently made significant progress in a wide range of domains such as computer vision [23, 28, 43, 67] and natural language processing [21, 27, 42, 59, 77]. There are also approaches [6, 37] using contrastive learning to solve code retrieval and summarization tasks; however, these approaches are token-based and cannot be applied to path-based vulnerability detection. Typically, contrastive learning formulates the self-supervised learning task as teaching a model which data instances are similar/dissimilar, which is automatically labeled from existing unlabeled data, using learning techniques such as siamese neural networks [5], and NCE loss [9]. There are theoretical studies and literature reviews regarding the success of contrastive learning on a range of applications [3, 38].

## 6 CONCLUSION

This paper presents ContraFlow, a new path-sensitive code embedding approach via self-supervised contrastive value-flow embedding that precisely preserves path-sensitive value-flow paths in the embedding space to detect software vulnerabilities. The value-flow paths are embedded with an attention-based structure-aware encoder which is trained with contrastive learning to preserve the local and global semantics of value-flow paths. The pretrained path encoder is then used to support the subsequent vulnerability detection task with attention-based neural networks. We evaluated ContraFlow using a benchmark of over 2 million methods and 11 million slices extracted from popular open-sourced projects. The experimental results show that ContraFlow outperforms the eight recent learning-based vulnerability detection approaches by up to 334.1%, 317.9%, 58.3% in informedness, markedness and F1 Score, and up to 450.0%, 192.3%, 450.0% in mean statement recall, mean statement precision and mean IoU, respectively.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their reviews and suggestions. This research is supported by Australian Research Grants DP200101328 and DP210101348.

## REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[2] Apple Inc. 2021. Clang static analyzer. https://clang-analyzer.llvm.org/scan-build.html.

[3] Sanjeev Arora, Hrishikesh Khandeparkar, Mikhail Khodak, Orestis Plevrakis, and Nikunj Saunshi. 2019. A Theoretical Analysis of Contrastive Unsupervised Representation Learning. *CoRR* abs/1902.09229 (2019). arXiv:1902.09229 http://arxiv.org/abs/1902.09229

[4] M. Backes, B. Köpf, and A. Rybalchenko. 2009. Automatic Discovery and Quantification of Information Leaks. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 141–153. https://doi.org/10.1109/SP.2009.18

[5] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1993. Signature Verification Using a "Siamese" Time Delay Neural Network. In *Proceedings of the 6th International Conference on Neural Information Processing Systems (NIPS '93)*. ACM, 737–744. https://doi.org/10.5555/2987189.2987282

[6] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '21)*. ACM, 511–521. https://doi.org/10.1145/3404835.3462840

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning based Vulnerability Detection: Are We There Yet? *CoRR* abs/2009.07235 (2020). arXiv:2009.07235 https://arxiv.org/abs/2009.07235

[8] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority over-Sampling Technique. *Journal of Artificial Intelligence Research* (2002), 321–357. https://doi.org/10.5555/1622407.1622416

[9] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton. 2020. A Simple Framework for Contrastive Learning of Visual Representations. *CoRR* abs/2002.05709 (2020). arXiv:2002.05709 https://arxiv.org/abs/2002.05709

[10] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 38 (2021), 33 pages. https://doi.org/10.1145/3436877

[11] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui. 2019. Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding. In *ICECCS*. 41–50. https://doi.org/10.1109/ICECCS.2019.00012

[12] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical Memory Leak Detection Using Guarded Value-Flow Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 480–491. https://doi.org/10.1145/1250734.1250789

[13] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Association for Computational Linguistics, Doha, Qatar, 103–111. https://doi.org/10.3115/v1/W14-4012

[14] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. Association for Computing Machinery, 57–68. https://doi.org/10.1145/512529.512538

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*

*(TACAS'08/ETAPS'08)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[16] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. 2010. Why Does Unsupervised Pre-Training Help Deep Learning? *J. Mach. Learn. Res.* (2010), 625–660. https://doi.org/10.5555/1756006.1756025

[17] Fabian. 2021. joern. https://github.com/ShiftLeftSecurity/joern/.

[18] Facebook. 2021. Infer. https://fbinfer.com/.

[19] WA Falcon and .al. 2021. PyTorch Lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning* 3 (2021).

[20] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. ACM, 508–512. https://doi.org/10.1145/3379597.3387501

[21] Hongchao Fang and Pengtao Xie. 2020. CERT: Contrastive Self-supervised Learning for Language Understanding. *CoRR* abs/2005.12766 (2020). arXiv:2005.12766 https://arxiv.org/abs/2005.12766

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. ACL, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[23] Basura Fernando, Hakan Bilen, Efstratios Gavves, and Stephen Gould. 2016. Self-Supervised Video Representation Learning With Odd-One-Out Networks. *CoRR* abs/1611.06646 (2016). arXiv:1611.06646 http://arxiv.org/abs/1611.06646

[24] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[25] Qing Gao, Sen Ma, Sihao Shao, Yulei Sui, Guoliang Zhao, Luyao Ma, Xiao Ma, Fuyao Duan, Xiao Deng, Shikun Zhang, and Xianglong Chen. 2018. CoBOT: Static C/C++ Bug Detection in the Presence of Incomplete Code. In *IEEE/ACM 26th International Conference on Program Comprehension (ICPC '18)*. IEEE, 385–3853. https://ieeexplore.ieee.org/document/8973011

[26] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 6894–6910. https://aclanthology.org/2021.emnlp-main.552

[27] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. *CoRR* abs/2104.08821 (2021). arXiv:2104.08821 https://arxiv.org/abs/2104.08821

[28] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. 2018. Unsupervised Representation Learning by Predicting Image Rotations. *CoRR* abs/1803.07728 (2018). arXiv:1803.07728 http://arxiv.org/abs/1803.07728

[29] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks* 18, 5 (2005), 602 – 610. https://doi.org/10.1016/j.neunet.2005.06.042 IJCNN 2005.

[30] Isabelle Guyon, Lisheng Sun-Hosoya, Marc Boullé, Hugo Jair Escalante, Sergio Escalera, Zhengying Liu, Damir Jajetic, Bisakha Ray, Mehreen Saeed, Michéle Sebag, Alexander Statnikov, WeiWei Tu, and Evelyne Viegas. 2019. Analysis of the AutoML Challenge series 2015-2018. In *AutoML (Springer series on Challenges in Machine Learning)*. https://www.automl.org/wp-content/uploads/2018/09/chapter10-challenge.pdf

[31] Günter Obiltschnig. 2021. POCO. https://pocoproject.org/.

[32] Jingxuan He, Cheng-Chun Lee, Veselin Raychev, and Martin Vechev. 2021. *Learning to Find Naming Issues with Big Code and Small Supervision*. ACM, New York, NY, USA, 296–311. https://doi.org/10.1145/3453483.3454045

[33] Hecht-Nielsen. 1989. Theory of the backpropagation neural network. In *International 1989 Joint Conference on Neural Networks*. 593–605 vol.1. https://doi.org/10.1109/IJCNN.1989.118638

[34] ImageMagick Team. 2021. ImageMagick. https://imagemagick.org/.

[35] Israel. 2021. Checkmarx. https://www.checkmarx.com/.

[36] Jaccard. 2021. Jaccard index. https://en.wikipedia.org/wiki/Jaccard_index.

[37] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive Code Representation Learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 5954–5971. https://doi.org/10.18653/v1/2021.emnlp-main.482

[38] Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and Fillia Makedon. 2020. A Survey on Contrastive Self-supervised Learning. *CoRR* abs/2011.00362 (2020). arXiv:2011.00362 https://arxiv.org/abs/2011.00362

[39] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. 2002. An Efficient K-Means Clustering Algorithm: Analysis and Implementation. *IEEE Trans. Pattern Anal. Mach. Intell.* (2002), 881–892. https://doi.org/10.1109/TPAMI.2002.1017616

[40] J. Kiefer and J. Wolfowitz. 1952. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics* 23, 3 (1952), 462 – 466. https://doi.org/10.1214/aoms/1177729392

[41] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[42] Tassilo Klein and Moin Nabi. 2020. Contrastive Self-Supervised Learning for Commonsense Reasoning. *CoRR* abs/2005.00669 (2020). arXiv:2005.00669 https://arxiv.org/abs/2005.00669

[43] Bruno Korbar, Du Tran, and Lorenzo Torresani. 2018. Cooperative Learning of Audio and Video Models from Self-Supervised Synchronization. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS '18)*. ACM, 7774–7785. https://doi.org/10.5555/3327757.3327874

[44] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

[45] Changsheng Li, Handong Ma, Zhao Kang, Ye Yuan, Xiao-Yu Zhang, and Guoren Wang. 2020. On Deep Unsupervised Active Learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, Christian Bessiere (Ed.). International Joint Conferences on Artificial Intelligence Organization, 2626–2632. https://doi.org/10.24963/ijcai.2020/364 Main track.

[46] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-Sensitive and Alias-Aware Typestate Analysis for Detecting OS Bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 859–872. https://doi.org/10.1145/3503222.3507770

[47] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.05493

[48] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability Detection with Fine-Grained Interpretations *(FSE '21)*. ACM, 292–303. https://doi.org/10.1145/3468264.3468597

[49] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin. 2021. VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector. *IEEE Transactions on Dependable and Secure Computing* 01 (2021), 1–1. https://doi.org/10.1109/TDSC.2021.3076142

[50] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. 2021. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. (2021), 1–1. https://doi.org/10.1109/TDSC.2021.3051525

[51] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *NDSS* (2018). https://doi.org/10.14722/ndss.2018.23158

[52] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[53] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[54] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., USA, 3111–3119. http://dl.acm.org/citation.cfm?id=2999792.2999959

[55] MITRE. 2021. CWE840. https://cwe.mitre.org/data/definitions/840.html.

[56] R. L. Russell, Louis Y. Kim, Lei H. Hamilton, T. Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *ICMLA* (2018), 757–762.

[57] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 693–706. https://doi.org/10.1145/3192366.3192418

[58] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-Sensitive Sparse Analysis without Path Conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, 930–943. https://doi.org/10.1145/3453483.3454086

[59] Tian Shi, Liuqing Li, Ping Wang, and Chandan K. Reddy. 2020. A Simple and Effective Self-Supervised Contrastive Learning Framework for Aspect Detection. *CoRR* abs/2009.09107 (2020). arXiv:2009.09107 https://arxiv.org/abs/2009.09107

[60] Inc Secure Software. 2014. RATS. https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[61] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from

Overfitting. *J. Mach. Learn. Res.* 15, 1 (jan 2014), 1929–1958.

[62] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-Flow-Based Precise Code Embedding. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 233 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428301

[63] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC)*. ACM, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[64] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, 254–264. https://doi.org/10.1145/2338965.2336784

[65] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

[66] Synopsys. 2021. Coverity. https://scan.coverity.com/.

[67] Li Tao, Xueting Wang, and Toshihiko Yamasaki. 2020. Self-Supervised Video Representation Learning Using Inter-Intra Contrastive Framework. In *Proceedings of the 28th ACM International Conference on Multimedia (MM '20)*. ACM, 2193–2201. https://doi.org/10.1145/3394171.3413694

[68] The Tcpdump Group. 2021. TCPDUMP. https://www.tcpdump.org/.

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS '17)*. Curran Associates, Inc., 6000–6010. https://doi.org/10.5555/3295222.3295349

[70] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. 2000. ITS4: a static vulnerability scanner for C and C++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. 257–267. https://doi.org/10.1109/ACSAC.2000.898880

[71] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*. IEEE Press, Piscataway, NJ, USA, 334–345. https://doi.org/10.5555/2820518.2820559

[72] Wikipedia. 2021. Norm. https://en.wikipedia.org/wiki/Norm_(mathematics).

[73] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation Learning on Graphs with Jumping Knowledge Networks. *CoRR* abs/1806.03536 (2018). arXiv:1806.03536 http://arxiv.org/abs/1806.03536

[74] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. 797–812. https://doi.org/10.1109/SP.2015.54

[75] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security (CCS '13)*. ACM, 499–510. https://doi.org/10.1145/2508859.2516665

[76] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-after-Free Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, 327–337. https://doi.org/10.1145/3180155.3180178

[77] Zonghan Yang, Yong Cheng, Yang Liu, and Maosong Sun. 2019. Reducing Word Omission Errors in Neural Machine Translation: A Contrastive Learning Approach. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL '19)*. ACM, 6191–6196. https://doi.org/10.18653/v1/P19-1623

[78] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc. https://doi.org/10.5555/3454287.3455116

[79] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree *(ICSE)*. IEEE/ACM, 783–794. https://doi.org/10.1109/ICSE.2019.00086

[80] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, New York, NY, USA.

[81] YaQin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS '19)*. Curran Associates Inc. https://doi.org/10.5555/3454287.3455202

[82] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. $\mu$VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *TDSC* (2019), 1–1. https://doi.org/10.1109/tdsc.2019.2942930