TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHTN
ĐHQG-HCM

APPLYING ATTENTION-BASED MODEL TO DETECT VULNERABLE CODE

GRADUATION THESIS

Student
Trần Hoài Châu - 18125067

Advisors
Dr. Lê Kim Hùng (UIT)
Msc.Eng. Trần Anh Duy

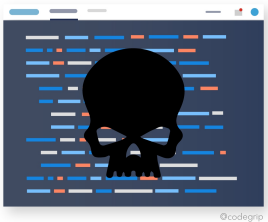
Table of Contents

Overview about source code Vulnerability detection task

Vulnerabilities in source code

Attackers can exploit vulnerabilities that are embedded in source code to compromise the integrity of an application or system. Here are some common vulnerabilities:

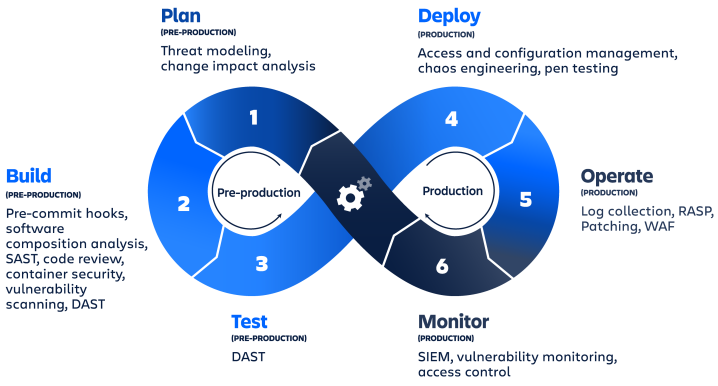
- SQL injection
- Cross-site scripting (XSS)
- Code injection
- Command injection
- And many more ...



Overview about source code vulnerability detection task

Source code Vulnerability detection (SVD) in software development lifecycle

DevSecOps



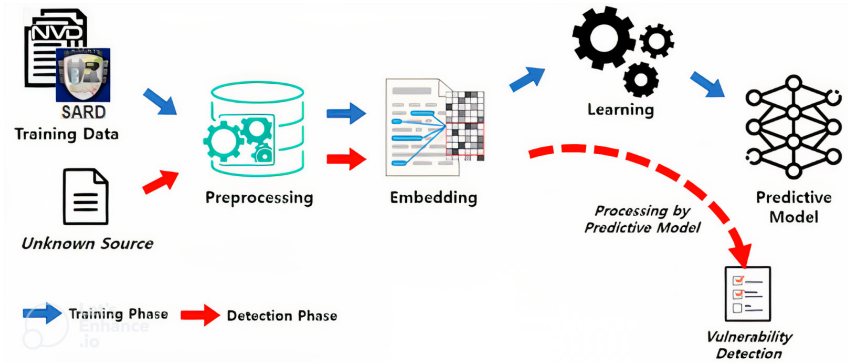
Approaches

Traditional approaches

- **Rule-based**: rule-based approach involves checking the code against a set of predefined rules that detect potential vulnerabilities.
- **Pattern-based**: The pattern-based approach involves searching for specific patterns of code that match known vulnerability patterns based on common attack techniques.
- **Limitations**: relying on predefined rules/patterns established by security experts -> limited in scalability, coverage, and lack of context.

Approaches

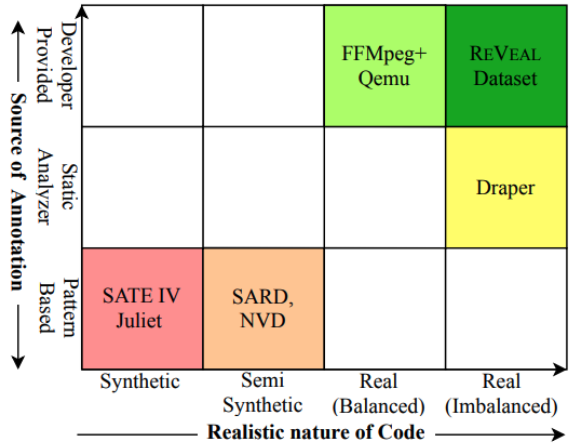
Deep learning based approaches



Challenges

Data challenges

- Unrealistic distribution of data
- Synthetic dataset
- High duplication rate
- Mainly focus on C/C++
- Contain noises such as variable and method names that might harm the performance of the model



Challenges

Model challenges

- Coarse level of granularity (i.e file level, function level, slice level,)
- Hard to capture the relationship between components in the code

```
def run_batch_mode(tweaks, args):
    for t in tweaks:
        if os_supported(t['os_v_min'], t['os_v_max']) \
                and is_executable(t['group'], args.groups, is_admin()) \
                and t['group'] != 'test':
            run_command(t['set'])


def run_command(cmd):
    try:
        subprocess.run(cmd, shell=True, timeout=60, check=True)
        dglogger.log_info(str(cmd))
    except subprocess.CalledProcessError as e:
#        dglogger.log_error(e, file=sys.stderr)
        dglogger.log_error(str(e)) # figure out deal w/file=sys.stderr!
    except subprocess.TimeoutExpired as e:
        dglogger.log_error(e, file=sys.stderr)
    except OSError as e:
        dglogger.log_error(e, file=sys.stderr)
    except KeyError as e:
        dglogger.log_error(e, file=sys.stderr)
```

Current solution & limitations

Data solutions

Current solution to collect data:

- Crawl raw source code from Git commits that are used to patch issues related to security vulnerabilities.
- The labeling process is based on the deleted/added lines in those commits.

Fix an XSS vulnerability in batch/views.py

As a Proof of Concept consider an AJAX POST request to https://pontoon.mozilla.org/batch-edit-translations/ with "action" parameter set to "<script>alert('xss')</script>".

master

Victor Chibotaru authored and **adngdb** committed on 21 Jan

Showing **1 changed file** with **1 addition** and **1 deletion**.

2 pontoon/batch/views.py

```
@@ -112,7 +112,7 @@ def batch_edit_translations(request):
112         """
113         form = forms.BatchActionsForm(request.POST)
114         if not form.is_valid():
115 -           return HttpResponseBadRequest(form.errors.as_json())
116
117         locale = get_object_or_404(Locale, code=form.cleaned_data['locale'])
118         entities = Entity.objects.filter(pk__in=form.cleaned_data['entities'])
```

Current solution & limitations

The limitations of data from Git commits

Limitations:

- There are commits that only fix a part of the problem
- Duplication is still unavoidable since there might be issues that consist of multiple commits that fix the same file or the code reuse practice in open-source projects
- There are changes in the commits that are irrelevant to the vulnerable pattern.

Current solution & limitations

Model solutions

To improve the prediction results at a more detailed, statement-level granularity, previous studies have employed graph neuron network(GNN) approaches, where each statement is represented as a node.

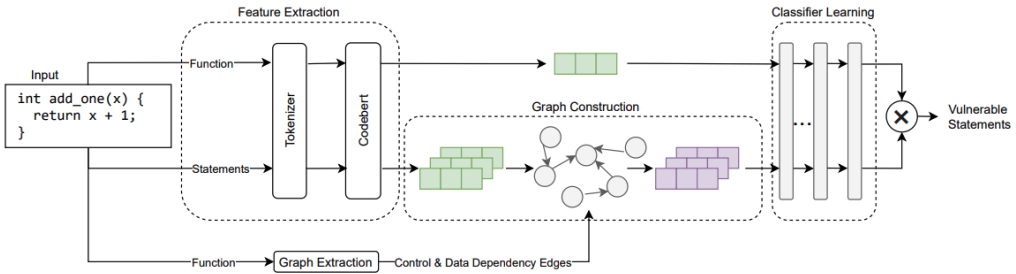


Figure 3: LineVD Overall Architecture

Current solution & limitations

Statement level SVD

```
1  void common_timer_get(struct k_itimer *timr, struct itimerspec64
   *cur_setting)
2  {
3    const struct k_clock *kc = timr->kclock;
4    ktime_t now, remaining, iv;
5    struct timespec64 ts64;
6    bool sig_none;
7
8    sig_none = timr->it_sigev_notify == SIGEV_NONE;
9    iv = timr->it_interval;
10
11   if (iv) {
12     cur_setting->it_interval = ktime_to_timespec64(iv);
13   } else if (!timr->it_active) {
14     if (!sig_none)
15       return;
16   }
17
18   kc->clock_get(timr->it_clock, &ts64);
19   now = timespec64_to_ktime(ts64);
20
21   if (iv && (timr->it_requeue_pending & REQUEUE_PENDING || sig_none))
22     timr->it_overrun += (int)kc->timer_forward(timr, now);
23     // Added: timr->it_overrun += kc->timer_forward(timr, now);
24
25   remaining = kc->timer_remaining(timr, now);
26
27   if (remaining <= 0) {
28     if (!sig_none)
29       cur_setting->it_value.tv_nsec = 1;
30   } else {
31     cur_setting->it_value = ktime_to_timespec64(remaining);
32   }
33 }
```

Current solution & limitations

Notable studies that used deep-learning model for SVD

Name	Dataset	Model type	Granularity	Language
Vuldeepeeker (2018)	SARD+ Draper	Sequence	Slice	C/C++
μVuldeepeeker (2018)	SARD + Draper	Sequence	Slice	C/C++
SySeVR (2018)	SARD + Draper	Sequence	Slice	C/C++
VulDeeLocator (2020)	SARD + NVD	sequence	Slice	C/C++
Vudenc (2022)	Vudenc	Sequence	Slice	Python
Devign (2019)	Devign	Graph	Function	C/C++
Reveal (2020)	Reveal	Graph	Function	C/C++
DeepWukong (2021)	SARD + Redis + Lua	Graph	Function	C/C++
IVdetect (2021)	SARD + NVD	Graph	Statement	C/C++
Linevd (2022)	Big-Vul	Graph	Statement	C/C++
MVD (2022)	MVD	Graph	Statement	C/C++

Current solution & limitations

Common step in GNN-based approaches

In general GNN-base approaches share some common steps:

- **Step** 1: Construct multiple graph structures (Control Flow Graph, Data Dependency Graph, Program Dependency Graph)
- **Step** 2: Feature vectors are extracted from each node using either static or dynamic embedding methods.
- **Step** 3: Extracted feature vectors are often combined with graph structure information to feed into a GNN-based approach to perform graph-level or node-level classification

Current solution & limitations

Limitations of GNN-based approaches

Although GNNs have demonstrated promising state-of-the-art performance compared to other methods, they still have some limitations:

- These models are often constrained by inductive biases, such as local connectivity and pre-defined graph structures
- Suffer from over-smoothing and over-squashing problems in GNNs
- The process of constructing graphs is challenging and hard to scale to other programming languages.

Table of Contents

2 Observation & Motivation

Goal

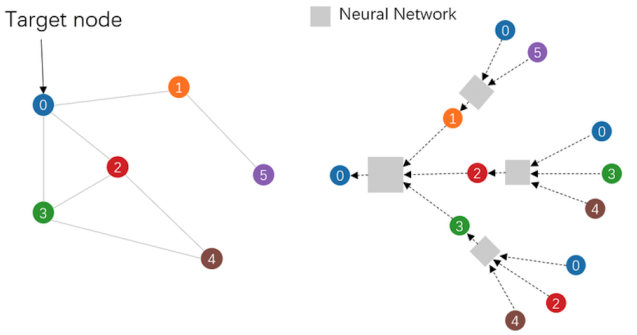
Address the limitations of GNN-base approaches

- A new approach is necessary to capture the relationship between statements without relying on any pre-defined graph structures.
- The target of this thesis is to come up with a new architecture that can achieve competitive performance with other GNN-based approaches with a fine-grained level of granularity (statement level)

Observation

Message-passing scheme in GNNs

In essence, the learning process of GNNs relies on the message-passing scheme, also known as neighborhood aggregation.



Observation

Message-passing scheme in GNNs

We can formulate message-passing operation in a general form as follow:

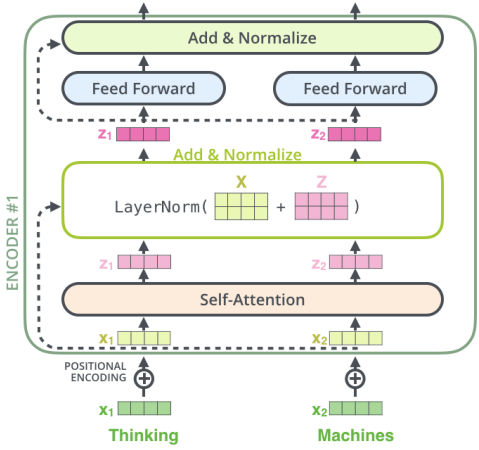
$$\vec{v}_i^{\,l} = \gamma^l\left(\vec{v}_i^{\,l-1}, \bigoplus_{j \in \mathcal{N}(i)} \phi^{l-1}(\vec{v}_i^{\,l-1}, \vec{v}_j^{\,l-1}, \vec{e}_{ij})\right)$$

Where:

- $\mathcal{N}(i)$ is the set of neighbor nodes of node i in a graph. The message-passing scheme can be formulated as follow:
- $\vec{v}_i^{\,l-1}$ is the feature vector of node i in the $(l-1)$-th layer of GNN model
- $\vec{e}_{ij}$ is edge features from node $j \in \mathcal{N}(i)$ to node i
- $\bigoplus$ is an aggregation function (typically a sum, mean, max or concatenate operator) .
- γ and ϕ are differentiable functions (Sigmoid, ReLu, MLPs, etc.)

Observation

Self-attention mechanism in Transformers



Observation

Self-attention mechanism in Transformers

Given a sentence that is tokenized in to n tokens, we can represent the embedding matrix for that sentence as $X \in \mathbb{R}^{n \times d}$ where d is the hidden size of the embedding vectors and n is. The entire process can be formulated as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, ..., head_h)W^O$$

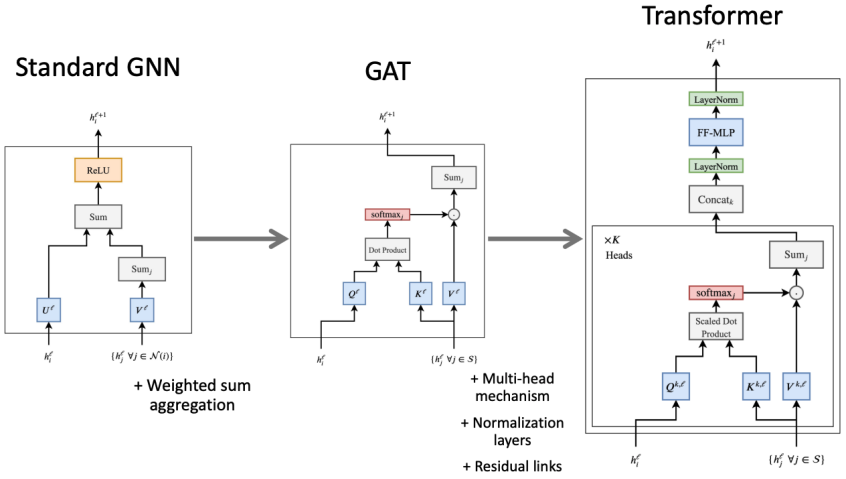
$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Here, Q, K, and V are the matrices obtained from the input sequence, and W_i^Q, W_i^K, and W_i^V are the weight matrices for the i^{th} attention head.

Motivation

Message passing scheme vs self-attention mechanism



Motivation

Message passing scheme vs self-attention mechanism

Let consider self-attention operator for one head in a transformer encoder layer. The self-attention formula can be written in the form of a message-passing formula as follows:

$$\vec{v}_i^l = \gamma\Big(\sum_{j \in \mathcal{N}(i) \cup i} \text{softmax}_j(W_Q^{l-1}\vec{v}_i^{l-1}, W_K^{l-1}\vec{v}_j^{l-1}) W_V^{l-1}\vec{v}_j^{l-1} \Big)$$

- γ include other components of the encoder (layer norm, MLP, residual connection).
- $\vec{v}_i^l$ for a statement i, and a set $\mathcal{N}(i)$ that contains all nodes in the graph (including i).
- Here, W_Q^{l-1},W_K^{l-1}, and W_Q^{l-1} are trainable weight matrices that produce query, key, and value vectors for the self-attention operation.
- The attention weights from node i to its neighbors j are rescaled to the range from 0 to 1 by the softmax$_j$ function.
- Finally, the next vector representation of node i ($\vec{v}_i^l$) is a weighted sum of all vectors $\vec{v}_j^{l-1}$ with $j \in \mathcal{N}(i) \cup i$.

Motivation

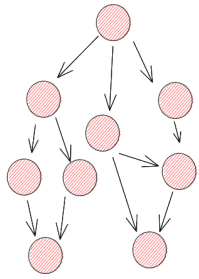
Message passing scheme vs self-attention mechanism

- We can see that while the model is in the message-passing system of GNNS, nodes obtain messages from their nearby neighbors and integrate them to generate updated contextual representations.
- Conversely, the self-attention mechanism creates a soft connection globally to all other nodes and calculates a node's contextual feature by weighting the features of its neighbors

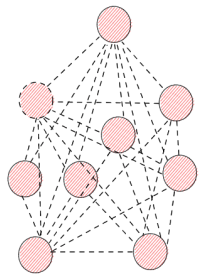
Motivation

DetectBERT: Self-attention is all we need!

- We can see that while the model in the message-passing system of GNNS, nodes obtain messages from their nearby neighbors and integrate them to generate updated contextual representations.
- Conversely, the self-attention mechanism creates a soft connection globally to all other nodes and calculates a node's contextual feature by weighting the features of its neighbors



Message passing scheme



Self-attention mechanism

Motivation

DetectBERT: Self-attention is all we need!

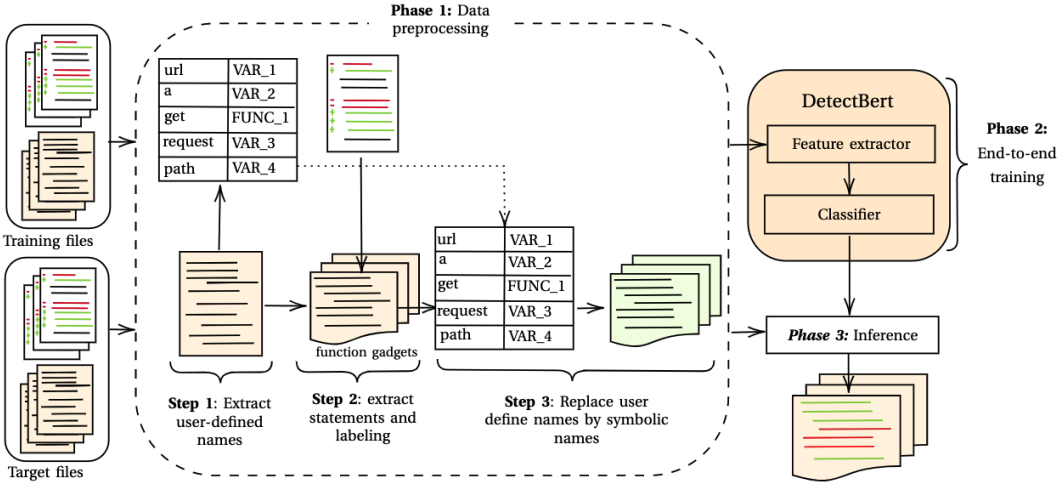
Our hypothesis: Self-attention mechanism is a potential approach that can be used to replace the message passing mechanism in GNN.

- We can consider each code statement in the code snippet as a token in a sentence
- Using self-attention mechanisms can help the model avoids any biases introduced by predefined graph structures, allowing the data to speak for itself.
- Enables the model to discover more meaningful patterns that may have been missed when identifying vulnerable statements using graph neural network (GNN) based approaches.

Table of Contents

Overview

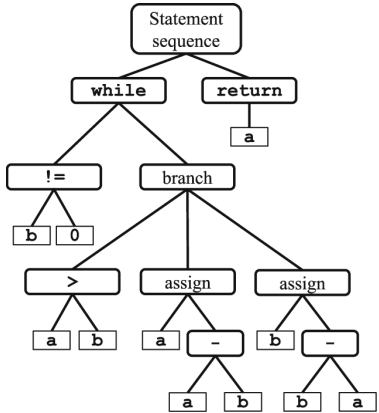
3 Methodology



Data preprocessing

Code normalizing

- In step 1 of the data preprocessing phase The Abstract Syntax Tree (AST) module of Python is used to collect all user-defined names from the code
- User-defined names can be extracted from code segments that create or assign variable values or define new functions while ignoring API and function calls.
- These names are saved in a dictionary for later use in step 3 to replace any other appearance or usage in the statement.



Data preprocessing

Statements extracting and labeling

Statement extracting

- The AST module is used to extract functions and statements.
- Docstrings, trailing space, tab, and line break are removed
- Import statements are also extracted and treated as a function gadget.

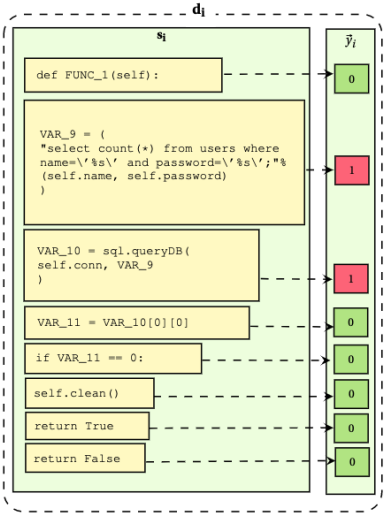
Labeling

- Only the last commit per issue is used.
- Deleted lines with only empty lines or comments are ignored.
- Deleted lines with valid content and having no identical line present in added lines are kept in the vulnerable lines list.

Data preprocessing

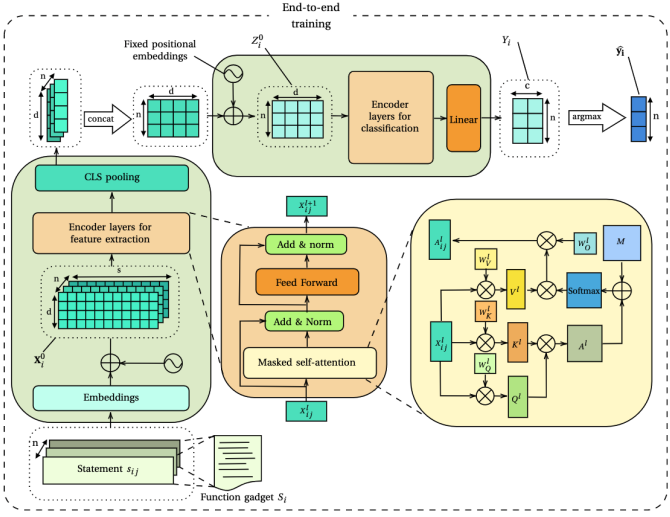
Function gadget

The output of phase 1 is a collection of N function gadgets, denoted as $\mathbf{D} = \{\mathbf{d_1}, \mathbf{d_2}, .., \mathbf{d_N}\}$, where each function gadget $\mathbf{d_i} = (\mathbf{s}i, \vec{y}i)$ consists of n normalized statements $\mathbf{s_i} = \{s_{i1}, s_{i2}, ..., s_{in}\}$ and a vector $\vec{y}_i = \{y_{i1}, y_{i2}, ..., y_{in}\}$ containing corresponding labels.



The architecture of DetectBERT

Overall architecture



The architecture of DetectBERT

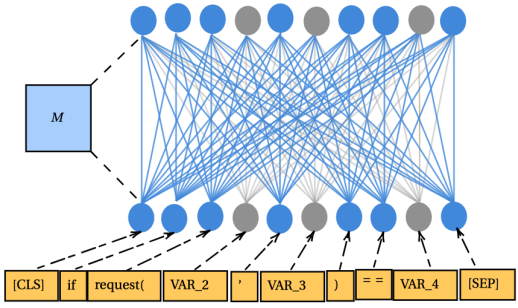
Feature extractor

- The DetectBERT training pipeline processes each function gadget as a batch of statements. Each statement is tokenized and added to positional embedding to produce an embedding tensor.
- The input embedding tensor $\mathbf{X}_i^0$ has dimensions of $n \times s \times d$, where n is the number of statements, s is the length of the longest statement, and d is the hidden dimension of each embedding vector.
- The tensor is then processed through multiple transformer encoder blocks. Each encoder block consists of a masked self-attention layer, a feed-forward layer, and layer normalizations.
- The output of the encoders is then pooled and concatenated which is used as input for the classifier model.

The architecture of DetectBERT

Attention mask

At every self-attention layer of each encoder block of the feature extractor, to emphasize the features that contribute to vulnerable patterns, all attention weights related to symbolic names, which are defined in the first step of the data preprocessing phase, are filtered out.



The architecture of DetectBERT

Classifier

- The processed output embeddings from the feature extractor are concatenated and added to a positional embedding matrix to create an embedding matrix $Z_i^0 \in \mathbb{R}^{d \times n}$.
- The processed embedding matrix is fed through a series of encoders to learn the connections between statements.
- The final output is projected by a linear layer and put through a softmax function to obtain an output matrix $\hat{Y}_i \in \mathbb{R}^{n \times c}$, where c is the number of classes.
- The predicted result is obtained by selecting the class with the highest probability.

End-to-end training

Training strategy

Let denote θ as the parameters of the deep learning model f. For each function gadget $\mathbf{d_i} = (\mathbf{s_i}, \vec{y}_i)$ that have n statements, the loss function is:

$$L_i(\theta) = \sum_{j=1}^{n} \frac{l_{ij}(f(s_{ij}, \theta), y_{ij})}{\sum_{i=1}^{n} w_{y_{ij}}} = \sum_{j=1}^{n} \frac{l_{ij}(\vec{\hat{y}}_{ij}, y_{ij})}{\sum_{i=1}^{n} w_{y_{ij}}}$$

$$l_{ij} = w_{y_{ij}} \log\Big(\frac{\exp(\hat{y}_{ij(y_{ij})})}{\sum_{k=0}^{c-1} \exp(\hat{y}_{ijk})}\Big)$$

Here:

- The loss function L_i is defined as the weighted sum of all statement losses.
- The mini-batch stochastic gradient descent was performed with a batch size of n.
- The adaptive gradient algorithm Adam is used with a linear scheduled learning rate of 1×10^{-5}.

End-to-end training

Training strategy

Let's $w_{y_{ij}} \in \{w_0, w_1, w_2, .., w_{c-1}\}$ denotes the weight assigned to each statement s_{ij} which is calculated based on the frequency of the label y_{ij} in the dataset, and the total number of samples.

$$w_{y_{ij}} = \frac{s}{c \times s_{y_{ij}}}$$

Here, s is the total number of statements of all functions gadgets in our dataset, c is the number of classes, and $s_{y_{ij}}$ is the number of statements that have label y_{ij}

Table of Contents

Experiments

Research questions

- **RQ1**: How does adjusting hyperparameters impact the performance of DetectBERT?
- **RQ2**: To what extent does incorporating a symbolic names attention mask improve DetectBERT's ability to identify vulnerable code?
- **RQ3**: How does DetectBERT's performance in statement classification compare to baseline GNN architectures that use graph data structures?
- **RQ4**: What are the statement types in which DetectBERT performs particularly well?
- **RQ5**: What is the performance of DetectBERT on the C/C++ dataset?

Dataset

Dataset for Python

- Vudenc and CVEfixes are chosen as the datasets for Python which contain multiple commits from real-world open-source project
- All commits from both dataset went through our data preprocessing steps for statement extracting and labeling.

Vulnerability	CVEFixes	Vudenc
Non vulnerable	5301	13020
CWE-22	68	325
CWE-77	n/a	328
CWE-79	102	58
CWE-89	n/a	1123
CWE-352	40	526
CWE-601	150	246
CWE-94	69	215
Total number of function gadgets	5730	15841
Total number of statements	57398	153919

Dataset

Dataset for C/C++

- In the Devign dataset, the authors did not classify the vulnerability type for each sample, however, most vulnerabilities are memory-related, like buffer overflow, memory leak, crash, and corruption.
- To extract statements and build function gadgets from C/C++ code, the PyClang module is used to extract ASTs. All statements are labeled using the sample data preprocessing pipeline as for Python

	No of function gadgets	No of statements
Non vulnerable	14858	1030313
Vulnerable	12460	31857

Dataset

Data splitting & evaluating

Throughout our experiments, the models were trained to perform multi-class classification tasks on the entire dataset instead of separately training multiple models for binary classification with only a subset of data related to a specific CWE

- The datasets were divided into training and testing sets with a ratio of $80:20$.
- During the evaluation phase, the k-Fold cross-validation technique with $k=5$ was used to ensure that our model was adequately validated and produced robust results.

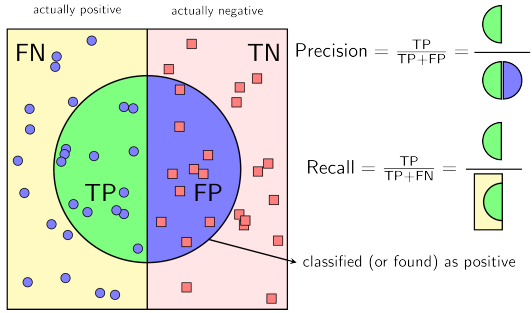
Metric

Precision & Recall

$$Precision_{macro} = \frac{1}{c}\sum_{i=0}^{c-1}\frac{TP_i}{TP_i + FP_i}$$

$$Recall_{macro} = \frac{1}{c}\sum_{i=0}^{c-1}\frac{TP_i}{TP_i + FN_i}$$

$$F1_{marcro} = \frac{2 \times Precision_{macro} \times Recall_{marco}}{Precision_{macro} + Recall_{macro}}$$



Metric

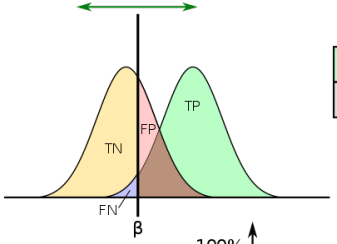
Area under the receiver operating characteristic curve (AUROC)

$$\text{sensitivity} = Recall = TPR = \frac{TP}{TP + FN}$$

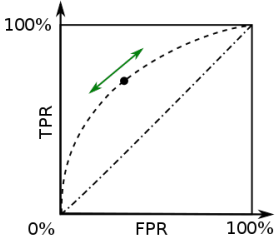
$$1 - \text{specificity} = FPR = \frac{TP}{TN + FP}$$

$$AUROC = \int_0^1 TPR(FPR)\, dFPR$$

$$\text{AUROC}_{macro} = \frac{\sum_{i=0}^{c-1} \text{AUROC}_i}{c}$$



TP	FP
FN	TN

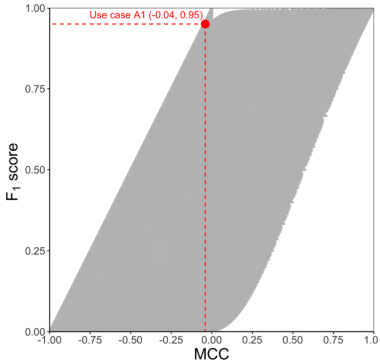


Metric

The Matthews correlation coefficient (MCC)

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

$$\mathrm{MCC}_{macro} = \frac{\sum_{i=0}^{c-1} \mathrm{MCC}_i}{c}$$



Experiment results

Model & Data preparation for RQ1

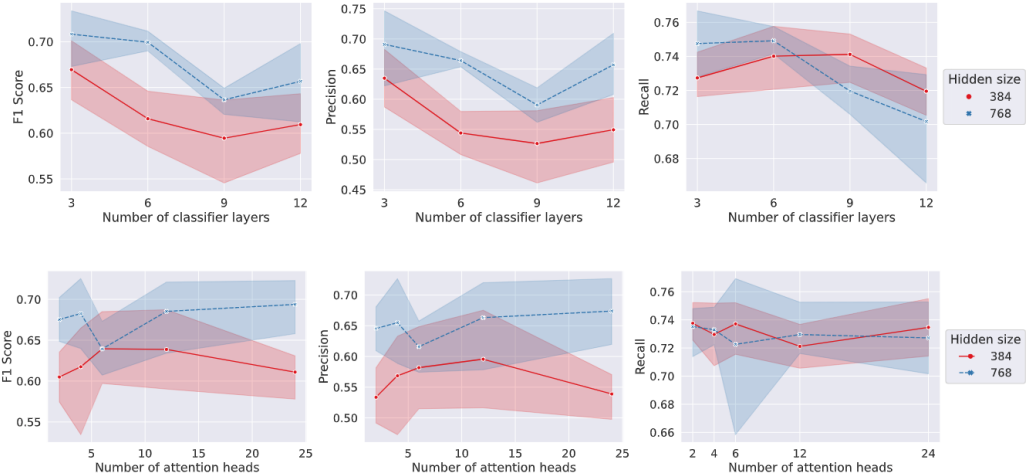
In this thesis, all experiments for the Python SVD task utilized two different language models, which are mpnet and MiniLM, as feature extractors of DetectBERT models.

Feature extractor	Num layers	Hidden size	Pooling	Model size
MiniLM	12	384	CLS pooling	120 MB
mpnet	12	768	CLS pooling	420 MB

Both models were pre-trained using contrastive learning objective on a massive corpus of over one billion sentence pairs, consisting of both natural languages and code. All DetectBERT models are trained with various hyperparameter settings are trained for 100 epochs on the concatenated dataset (Vudenc + CVEfixes).

Experiment results

Experiment results for RQ1



Experiment results

Data preparation for RQ2

Experiments are conducted separately with two versions of each dataset - VUDENC and CVEFixes on two different scenarios.

Dataset versions

- The first version contained unprocessed function gadgets with raw code.
- The second version replaced all inconsistencies such as user-defined variable names, method names, and class names with symbolic names like 'VAR_1' and 'FUNC_3'.

Scenarios

- In the first scenario, we split the data into a train and test set with an 80:20 ratio. The model is then trained using the train and evaluation splits respectively.
- In the second scenario, the datasets are also split into train and test sets with the same ratio, but the test set is augmented by replacing all user-defined variables and method names with random strings such as 'a', 'b', 'c', etc.

Experiment results

Model preparation for RQ2

- In this experiment, the pre-trained mpnet model was chosen as the feature extractor and a classifier that contains 3 layers of transformer encoders was also employed.
- For datasets without normalization, the standard self-attention layer was used for feature extraction.
- For datasets with normalized code, an attention mask was applied in the self-attention operator to eliminate the attention signal generated by the symbolic names.

Experiment results

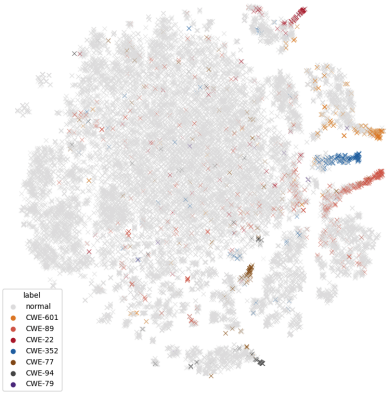
Experiment results for RQ2

- Particularly, our empirical results showed that models trained on normalized datasets outperformed models trained on unnormalized datasets in terms of recall scores in both scenarios.
- In contrast, models trained on unnormalized data had better precision than those trained on normalized data, except for CWE types such as CWE-352 and CWE-601 in the CVEfixes dataset.

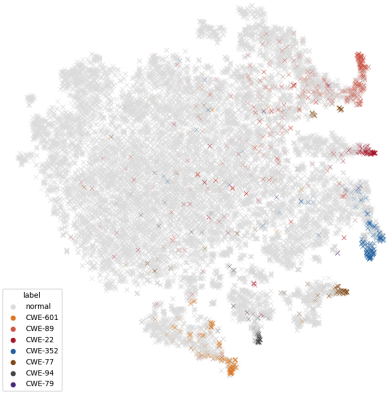
Scenario	Dataset	Normalized	Metric	Score
Normal	CVEFixes	False	F1	69.74±15.14
			Precision	**82.82±11.57**
			Recall	62.15±18.68
			AUROC	0.853±0.129
		True	F1	**70.30±19.51**
			Precision	72.67±21.91
			Recall	**69.83±18.50**
			AUROC	**0.898±0.093**
	Vudenc	False	F1	**66.04±9.69**
			Precision	**63.30±10.24**
			Recall	70.28±12.95
			AUROC	0.909±0.038
		True	F1	54.52±9.12
			Precision	45.19±11.28
			Recall	**72.32±13.36**
			AUROC	**0.917±0.055**
Augmented	CVEFixes	False	F1	49.20±20.07
			Precision	**73.71±21.25**
			Recall	37.64±17.25
			AUROC	0.829±0.147
		True	F1	**56.06±23.73**
			Precision	64.99±28.78
			Recall	**50.22±22.30**
			AUROC	**0.860±0.143**
	Vudenc	False	F1	**51.11±8.36**
			Precision	**66.12±6.75**
			Recall	42.63±11.68
			AUROC	0.858±0.063
		True	F1	45.20±7.41
			Precision	37.62±8.03
			Recall	**60.52±15.82**
			AUROC	**0.904±0.057**

Experiment results

Experiment results for RQ2



Before normalization

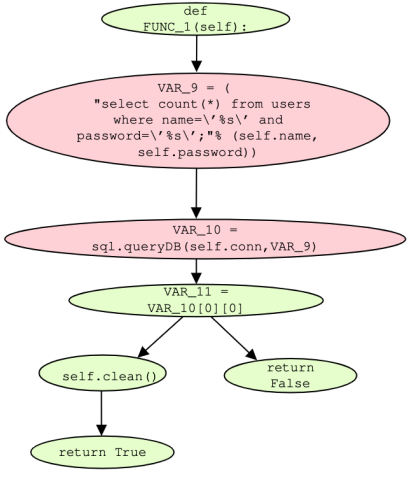


After normalization

Experiment results

Data preparation for RQ3

- Apart from the datasets used in previous RQs, to benchmark the performance of DetectBERT compared to GNN-based approaches, we need to construct a dataset that used graphs as the underlying data structure.
- In this experiment, we chose to build control flow graphs (CFG) from available function gadgets.



Experiment results

Model preparation for RQ3

Graph Convolutional Network (GCN):

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \frac{1}{\sqrt{\deg(i) \times \deg(j)}} W^{(l)} h_j^{(l)} \right)$$

Where $\deg(j)$ and $\deg(j)$ represent the degree of node i and its neighbor node j. The GCN computes an embedding vector of node i for the next layer by aggregate representation vectors of all neighbors and scales it based on the influence/degree of each node

Graph Attention Network (GAT):

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)} \right)$$

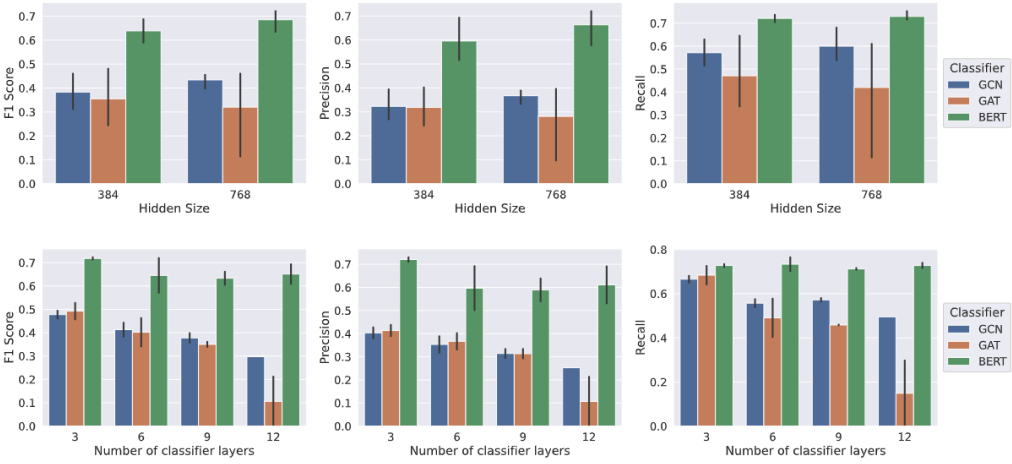
Similar to GCN, GAT also aggregates all neighbor features based on the attention coefficient $\alpha_{ij}^{(l)}$ between nodes i and j at layer l.

$$\alpha_{ij}^{(l)} = \text{Softmax}(\text{LReLU}(a^{(l)T}[W^{(l)} h_i^{(l)} || W^{(l)} h_j^{(l)}]))$$

where $a^{(l)}$ is the attention mechanism that computes the importance of the neighboring nodes, LReLU is the Leaky ReLU activation function, and $||$ denotes concatenation.

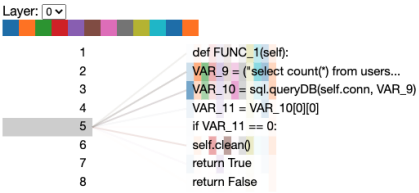
Experiment results

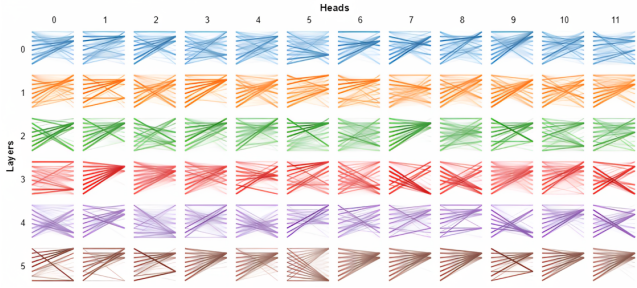
Experiment results for RQ3



Experiment results

Experiment results for RQ3





Experiment results

Experiment results for RQ3

Overall, in all metrics, the performance of DetectBERT outshined that of GNN-based models. By utilizing the power of transformers, we can avoid the over-smoothing and over-quashing issues often associated with GNN models, while still achieving high levels of accuracy and recall.

Classifier	F1	Prec	Rec
GCN	40.48 ± 6.70	34.18 ± 6.02	58.35 ± 6.35
GAT	33.75 ± 16.75	29.99 ± 13.99	44.53 ± 22.59
BERT (DetectBERT)	**64.88 ± 5.46**	**60.71 ± 8.08**	**73.08 ± 2.65**

Experiment results

Data & model preparation for RQ4

- Processed statement s_{ij} was categorized into statement types, such as "Return", "Condition", "For/While", "Assign", "Expression", "Assert", "Import From", "Augmented assignment", "Import", "Function declaration", and "Docstring", based on the node type of the AST object.
- The model architecture trained in the RQ2 as reused and evaluated. Through this analysis, we aim to identify the common types of vulnerable statements that typically contain vulnerable patterns, as well as the degree of agreement between the models and the ground truth labels for each statement type.

Experiment results

Experiment results for RQ4

Type	Number of statements	F1	MCC	AUROC
Condition	1660	83.99	0.8238	0.9819
Return	1110	87.20	0.8767	0.9441
Assign	3524	72.42	0.6928	0.8530
Expression	1697	73.07	0.6819	0.7694
For/While	251	49.96	0.8156	0.6606
Assert	79	41.33	0.6986	0.5480
Import From	365	33.12	0.4163	0.4571
Augmented assignment	44	16.66	0.0000	0.0000
Import	115	16.66	0.0000	0.0000
Function declaration	67	16.62	0.0000	0.0000
Docstring	393	16.64	0.0000	0.0000

Performance of DetectBERT on CVEFixes

Type	Number of statements	F1	MCC	AUROC
Expression	4983	69.42	0.5852	0.9082
Assign	9001	62.79	0.5940	0.9031
Return	2629	51.17	0.4768	0.8816
Import From	1122	48.87	0.4057	0.8045
For/while	756	47.52	0.5185	0.7717
Condition	3613	43.85	0.4492	0.7655
Augmented assignment	170	45.43	0.6758	0.6158
Import	674	29.17	0.3572	0.5503
Assert	192	48.07	0.7405	0.4998
Docstring	918	12.50	0.0000	0.0000
Function declaration	2827	12.48	0.0000	0.0000

Performance of DetectBERT on VUDENC

Experiment results

Baseline for RQ5

We used previous works conducted by David et al.(LineVD) which also built GNN-based solutions to perform statement-level SVD.

- Program dependency graph was used as the underlying graph structure.
- CodeBERT was used as the feature extractor model.
- GCN or GAT was used as the classifier model.

Experiment results

Data preparation for RQ5

Due to the limited time and resources, we only managed to conduct our experiments on the Devign dataset which is a subset of the Big-Vul dataset used in LineVD

- The same data preprocessing pipeline used for Python is also applied for C/C++ code.
- To extract and normalize function gadgets, PyClang was utilized to obtain the AST object C/C++ from C/C++ functions/snippets. Since the datasets and the data preprocessing pipeline are different, the empirical results from this experiment are just for demonstration purposes.

Experiment results

Model preparation for RQ5

Two versions of CodeBERT were used for feature extraction.

- The first model is the same model used in LineVD (CodeBERT-base)
- The second model is particularly pre-trained for C/C++ programming languages (CodeBERT-C)

The classifier model is a BERT model that contains three transformer encoder layers. Each model was trained on the Devign dataset for 100 epochs using the same strategy used for Python

Experiment results

Experiment results for RQ5

Dataset	Feature extractor	Classifier	F1	Prec	Rec	AUROC
Big-Vul	CodeBERT-base	GNNs(Avg)	21.1	15.8	33.8	**0.780**
Devign	CodeBERT-base	BERT	21.2	14.5	**38.9**	0.747
Devign	CodeBERT-C	BERT	**31.2**	**29.8**	32.7	0.760

Performance of DetectBERT on the Devign dataset compare to the average performance of GNN-based models trained on the Big-Vul dataset in LineVD. We can see that DetectBERT achieved competitive performance in every metric. Particularly, the best DetectBERT model outperforms models used in LineVD in terms of the precision score.

Experiment results

Experiment results for RQ5

```
105         if import_from_settings('OIDC_USE_NONCE', True):
106             nonce = get_random_string(
107                 import_from_settings('OIDC_NONCE_SIZE', 32)
108             )
109             params.update({
110                 'nonce': nonce
111             })
112             request.session['oidc_nonce'] = nonce
113
114         request.session['oidc_state'] = state
115         request.session['oidc_login_next'] = (
116             request.GET
117             .get(redirect_field_name)
118         )
119
120         query = urlencode(params)
121         redirect_url = (
122             '{url}?{query}'
123             .format(
124                 url=self.OIDC_OP_AUTH_ENDPOINT,
125                 query=query
126             )
127         )
128         return HttpResponseRedirect(redirect_url)
129
```

```
89          slice = av_mallocz(sizeof(*slice));
90          if (!slice)
91            return AVERROR(ENOMEM);
92          err = cbs_h265_read_slice_segment_header(
93            ctx,
94            &bc,
95            &slice->header
96          );
97          if (err < 0) {
98            av_free(slice);
99            return err;
100         }
101         pos = bitstream_tell(&bc);
102         len = unit->data_size;
103         if (!unit->data[len - 1]) {
104           int z;
105           for (
106             z = 0;
107             z < len && !unit->data[len - z - 1]
108             ; z++
109           );
110           av_log(ctx->log_ctx, AV_LOG_DEBUG,
111                  "Deleted %d trailing zeroes"
112                  "from slice data.\n", z);
113           len -= z;
114         }
115         slice->data_size = len - pos / 8;
116         slice->data = av_malloc(slice->data_size);
117         if (!slice->data) {
118           av_free(slice);
119           return AVERROR(ENOMEM);
120         }
121         memcpy(slice->data,
122                unit->data + pos / 8,
123                slice->data_size);
124         slice->data_bit_start = pos % 8;
125         unit->content = slice;
126     }
127     break;
```

Table of Contents

Limitations

Data limitations

- The VUDENC dataset may contain duplicate data that could lead to data leakage and overfitting during model training.
- The CVEFixes dataset has a limited number of commits that patch vulnerabilities in Python, and the dataset may not be up-to-date since the data collection pipeline needs to be re-run, which can be time-consuming.
- Both datasets only focus on vulnerabilities that are patched by source code changes and publicly disclosed, which may not accurately represent the full range of vulnerabilities present in software systems.

Limitations

Data preprocessing pipeline limitation

- Firstly, the data preprocessing pipeline used in this thesis heavily relies on the AST module leading to long processing times for large files.
- The AST module for Python3.x can not parse the source code for Python2.x.
- the pipeline parses files into function gadgets, which limits the context of the model to a functional unit, making it difficult for the model to detect more complex or nuanced vulnerabilities that require a broader context or reasoning from other functions.

Limitations

Model limitations

- One key limitation of DetectBERT is its computational complexity, which is primarily caused by the self-attention layers.
- DetectBERT is not suitable to be used in real-time or low-latency applications (e.g in IoT devices) where speed and efficiency are critical.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2.d)$	$O(1)$	$O(1)$
Recurrent	$O(n.d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k.n.d^2)$	$O(1)$	$O(\log_k(n))$
Recurrent	$O(r.n.d)$	$O(1)$	$O(n/r)$

Future work

Data-centric improvements

- Develop new methods for statement extracting without relying on any built-in library for a programming language, a very potential approach is to use part-of-speech (POS) tagging technique.
- Future works should also focus on creating more datasets from real-world projects to train statement-level SVD multiple programming languages.
- Applying data augmentation is also a potential idea. One way to do this is by extracting and replacing user-defined names with random strings, thus creating more training samples.

Future work

Model-centric improvements

- Different transformer-based architectures (XLNet, Reformer, Longformer, Fastformer,etc.) could be utilized to reduce DetectBERT's complexity.
- Pre-train strategies for feature extractors (e.g., contrastive learning objectives) could be explored to further enhance the model's performance.

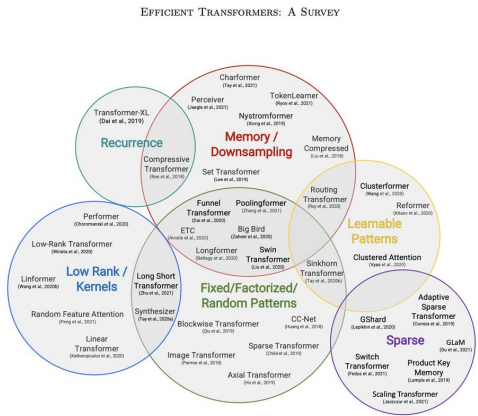


Figure 2: Taxonomy of Efficient Transformer Architectures.

Future work

Model-centric improvements

- Utilizing transformer architecture on graph data could be an avenue to explore tasks that require graphs as the underlying data structure.
- Previous works such as Graph Transformer, Graphormer, TokenGT, etc. have achieved impressive results in tasks that require graphs.

Table 1: A summary of papers that applied Transformers on graph-structured data. **GA**: GNNs as Auxiliary Modules; **PE**: Improved Positional Embedding from Graphs; **AT**: Improved Attention Matrix from Graphs.

Method	GA	PE	AT	Code
[Zhu *et al.*, 2019]			✓	✓
[Shiv and Quirk, 2019]		✓		
[Wang *et al.*, 2019]		✓	✓	
U2GNN [Nguyen *et al.*, 2019]			✓	✓
HeGT [Yao *et al.*, 2020]			✓	✓
Graformer [Schmitt *et al.*, 2020]			✓	✓
PLAN [Khoo *et al.*, 2020]			✓	✓
UniMP [Shi *et al.*, 2020]			✓	
GTOS [Cai and Lam, 2020]		✓	✓	✓
Graph Trans [Dwivedi and Bresson, 2020]		✓	✓	✓
Grover [Rong *et al.*, 2020]	✓			✓
Graph-BERT [Zhang *et al.*, 2020]	✓	✓		✓
SE(3)-Transformer [Fuchs *et al.*, 2020]			✓	✓
Mesh Graphormer [Lin *et al.*, 2021]	✓	✓		✓
Gophormer [Zhao *et al.*, 2021]			✓	
EGT [Hussain *et al.*, 2021]		✓	✓	✓
SAN [Kreuzer *et al.*, 2021]		✓	✓	✓
GraphiT [Mialon *et al.*, 2021]	✓	✓	✓	✓
Graphormer [Ying *et al.*, 2021]	✓	✓	✓	✓
Mask-transformer [Min *et al.*, 2022]			✓	✓
TorchMD-NET [Thölke and de Fabritiis, 2022]			✓	✓

Conclusion

Main contributions and conclusion

- I introduced DetectBERT, a versatile architecture for statement-level SVD. Our approach can easily adapt to multiple programming languages while still achieving competitive performance.
- New data preprocessing pipeline is created to extract, normalize, and label Python statements using commit history, with our own filtering criteria to reduce false positives and dataset duplication.
- Experiments show DetectBERT outperforms GNN-based approaches on a constructed CFG dataset for Python.
- DetectBERT is also applied to C/C++ code on the Devign dataset, showing competitive results compared to GNN-based approaches.
- Implementations, datasets, and models are publicly available on GitHub and Huggingface Hub to enable future research.

Thank you for listening!
Any questions?