Trường Đại học Công nghệ Thông tin – ĐHQG Tp. HCM

Khoa Mạng máy tính và truyền thông

BÁO CÁO TỔNG KẾT ĐỒ ÁN MÔN HỌC

Môn học: Lập trình an toàn và khai thác lỗ hổng phần mềm

Tên chủ đề: Phát hiện lỗ hổng trong mã nguồn phần mềm

Mã nhóm: Nhóm 1 *Mã đề tài: CK25*

Lớp: NT521.O11.ANTN

1. THÔNG TIN THÀNH VIÊN NHÓM:

(Sinh viên liệt kê tất cả các thành viên trong nhóm)

STT	Họ và tên	MSSV	Email
1	Hà Thị Thu Hiền	21522056	21522056@gm.uit.edu.vn
2	Phạm Ngọc Thơ	21522641	21522641@gm.uit.edu.vn
3	Nguyễn Ngọc Nhung	21521248	21521248@gm.uit.edu.vn

2. TÓM TẮT NỘI DUNG THỰC HIỆN:[1]

A. Chủ đề nghiên cứu trong lĩnh vực An toàn phần mềm:

þ Phát hiện lỗ hổng bảo mật phần mềm

¨ Khai thác lỗ hổng bảo mật phần mềm

¨ Sửa lỗi bảo mật phần mềm tự động

¨ Lập trình an toàn

¨ Khác: ……..

B. Tên bài báo tham khảo chính:

Tang, W., Tang, M., Ban, M., Zhao, Z., & Feng, M. (2023). CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. Journal of Systems and Software, 199, 111623.

C. Dịch tên Tiếng Việt cho bài báo:

CSGVD: Phương pháp học sâu kết hợp nhúng trình tự và nhúng biểu đồ để phát hiện lỗ hổng mã nguồn.

D. Tóm tắt nội dung chính:

- Để bảo mật phần mềm, điều quan trọng là phải phát hiện lỗ hổng tiềm ẩn ở trong nó. Hiệu suất của các phương pháp phát hiện lỗ hổng tĩnh truyền thống bị giới hạn bởi các quy tắc xác định trước, phụ thuộc nhiều vào

[1] Ghi nội dung tương ứng theo mô tả

chuyên môn của các nhà phát triển. Các mô hình phát hiện lỗ hổng dựa trên deep learning hiện tại thường chỉ sử dụng một phương pháp nhúng biểu đồ hoặc trình tự (single sequence or graph embedding) duy nhất để trích xuất các tính năng lỗ hổng.

- Trong nghiên cứu này, mục tiêu là đào tạo được một mô hình có thể học cách biểu diễn đồ thị từ đồ thị luồng điều khiển ở cấp độ câu lệnh (statement-level control flow graphs).
- Đề xuất mô hình mạng thần kinh nơ-ron tên là CSGVD: CSGVD được thiết kế để xử lý việc phát hiện lỗ hổng ở mức độ hàm (function) trong source code thông qua việc phân loại đồ thị nhị phân, sử dụng đồ thị luồng điều khiển của mã nguồn làm đầu vào. Mục tiêu là tìm ra các lỗ hổng tiềm ẩn trong mã nguồn một cách chính xác và hiệu quả, đồng thời xác định xem function-level code có dễ bị tấn công hay không
- Đề xuất một mô-đun PE-BL thay thế cho Word2Vec và Doc2Vec để nhúng đỉnh, mô-đun này sử dụng trọng số lớp nhúng từ (word) của một mô hình pre-trained language model để khởi tạo lớp nhúng của CSGVD và một BiLSTM.
- Đề xuất the mean biaffine attention pooling (M-BFA) cho graph embedding. Kết quả cho thấy nó có thể cải thiện hiệu suất của mô hình.

E. Tóm tắt các kỹ thuật chính được mô tả sử dụng trong bài báo:

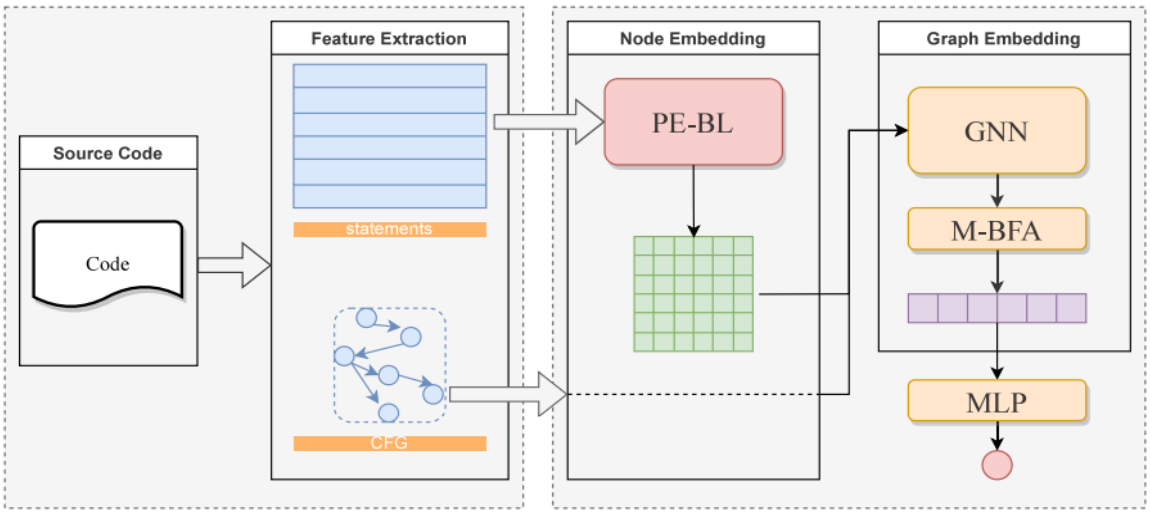


Fig. 1. Overall framework.

Các kỹ thuật cho 4 giai đoạn chính trong phương pháp đề xuất, bao gồm:

- Feature extraction:
- Để huấn luyện neural network, chúng ta cần phải chuyển đổi raw function-level source code thành định dạng dữ liệu được neural network model chấp nhận.

- Sử dụng **Joern** để phân tích mã nguồn và trích xuất biểu đồ luồng điều khiển. Vì biểu đồ do Joern trích xuất quá lớn nên bài báo hợp nhất các node được trích xuất theo hàng. Bằng cách nén biểu đồ và chuyển đổi biểu đồ luồng điều khiển ban đầu thành biểu đồ luồng điều khiển ở mức câu lệnh (stament-level control flow graph), số node trong biểu đồ nén sẽ nhỏ hơn hoặc bằng số dòng mã, điều này sẽ làm giảm đáng kể thời gian và tài nguyên tính toán.

- Node embedding: Bài báo đề xuất **PE-BL module** (bao gồm lớp Pre-Embedding và lớp BiLSTM) để tạo ra một bộ biểu diễn mã đầy đủ thông tin và toàn diện cho mã nguồn. Cụ thể:

- Thay vì sử dụng lớp nhúng truyền thống, bài báo giới thiệu mô hình ngôn ngữ được đào tạo trước (pre-trained programming language model) là **CodeBERT**, để khởi tạo lớp nhúng. Bài báo tạo các trọng số của lớp nhúng với các trọng số nhúng từ (word embedding weights) đã được huấn luyện từ CodeBERT. Bài báo sử dụng một hàm (function) duy nhất của mã nguồn làm đầu vào thô (raw input). Chúng ta chia function thành một tập hợp các câu lệnh S = {S1, S2, S3, …, Sn) và CFG ở cấp độ câu lệnh (statement-level CFG), trong đó *si* tương ứng với node *vi* của CFG. Mỗi câu lệnh trước tiên phải được mã hoá thông qua mã thông báo BPE được đào tạo trước của CodeBERT. Sau đó, CSGVD lấy một lớp nhúng, sử dụng các trọng số của lớp nhúng từ (word embedding layer weights) của CodeBERT để thu được biểu diễn vector.
- LSTM hai chiều (**BiLSTM**) được sử dụng để hợp nhất thông tin ngữ nghĩa cục bộ của mã và thu được biểu diễn vector *xi* của câu lệnh *si* tương ứng với nút *vi*.

- Graph embedding:

- Sử dụng **multi-layer GNN** với khả năng kết nối dư (residual connectivity), bài báo thu được biểu diễn ẩn của từng node trong biểu đồ luồng điều khiển trong khi tổng hợp thông tin node của các node lân cận.
- Trước khi thực hiện nhiệm vụ phân loại biểu đồ (graph classification), chúng ta cũng cần tổng hợp thông tin của các node để thu được biểu diễn vector của biểu đồ. Bài báo đã đề xuất **mean biaffine attention (M-BFA) pooling**. Đầu tiên, bài báo giả sử sự tồn tại của supernode *vs*, supernode này có vai trò chủ đạo đối với việc tổng hợp thông tin biểu đồ. Thứ hai, tính điểm chú ý (attention score) của *vs* cho mỗi node. Cuối cùng, tổng hợp bao gồm trọng số được thực hiện cho mỗi node dưới dạng biểu diễn vector của biểu đồ.

- Classifier learning:

- CSGVD sử dụng a **multilayer perceptron (MLP)** như là lớp phân loại.
- 2 lớp MLP được xây dựng trên 2 công thức (14)-(15) để dự đoán hàm đã cho có dễ bị tấn công hay không:

$$\boldsymbol{o} = softmax(\sigma(U \cdot \boldsymbol{h_g}) \cdot V) \quad (14)$$

$$y = argmax(\boldsymbol{o}) \quad (15)$$

where U and V are two learnable weight matrix, $\sigma(\cdot)$ is an activation function and $y \in \{0, 1\}$ denotes the final predicted result.

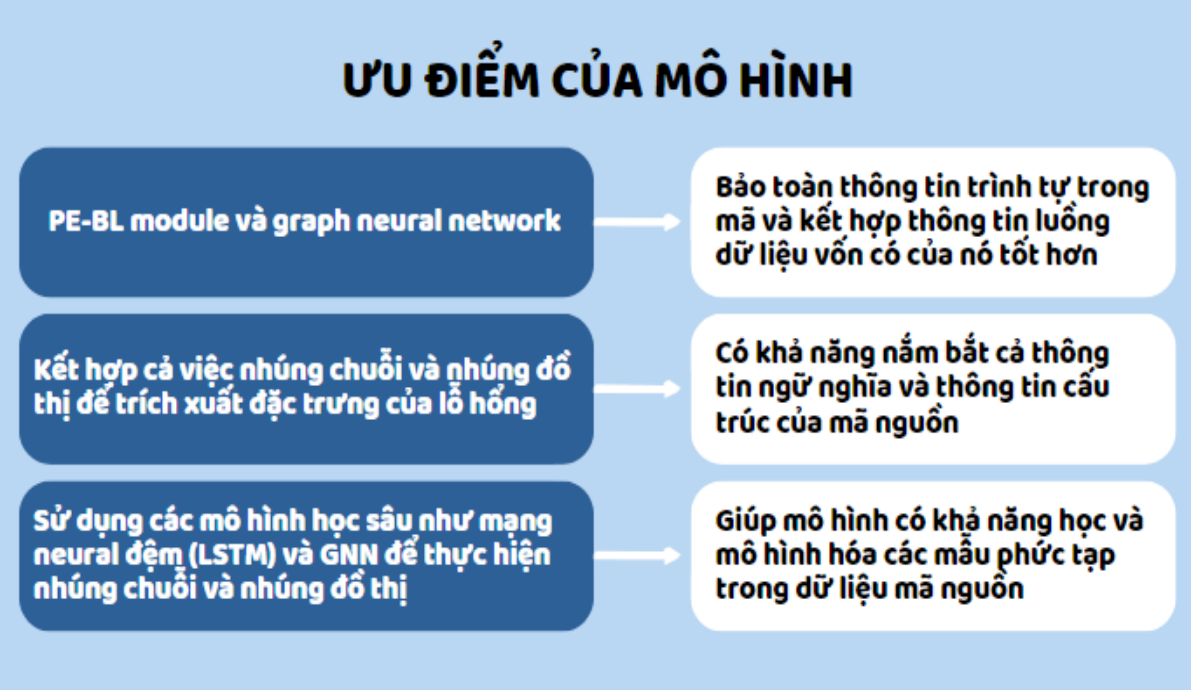
F. Môi trường thực nghiệm của bài báo:

- Cấu hình máy tính: Ubuntu 18.04 desktop workstation với một NVIDIA RTX 3090 GPU và một Intel(R) Core(TM) i9-10900X CPU hoạt động ở tốc độ CPU là 3.70 GHz.
- Các công cụ hỗ trợ sẵn có:
- Công cụ Joern để trích xuất thuộc tính.
- Trình tối ưu hóa Adam và bộ lập lịch tốc độ học tập tuyến tính trong quá trình train model với epoch là 100 và learning rate là 5e-4.
- Xây dựng một mô hình GNN 2 lớp, với batch size = 128 và dropout là 0.5.
- Kích thước ẩn cho tất cả các lớp ẩn của LSTM, GNN và MLP là 768, bằng với chiều của pre-trained embedding.
- Ngôn ngữ lập trình để hiện thực phương pháp: bài báo không đề cập.
- Đối tượng nghiên cứu (chương trình phần mềm dùng để kiểm tra tính khả thi của phương pháp/tập dữ liệu – nếu có): tác giả dùng tập dữ liệu từ CodeXGLUE (https://github.com/microsoft/CodeXGLUE), ban đầu bao gồm 27318 mẫu, sau đó được tổ chức và chia lại thành train/validation/test với tỉ lệ 8:1:1.
- Tiêu chí đánh giá tính hiệu quả của phương pháp: đánh giá dựa trên Accuracy, Recall, F1-score.

G. Kết quả thực nghiệm của bài báo:

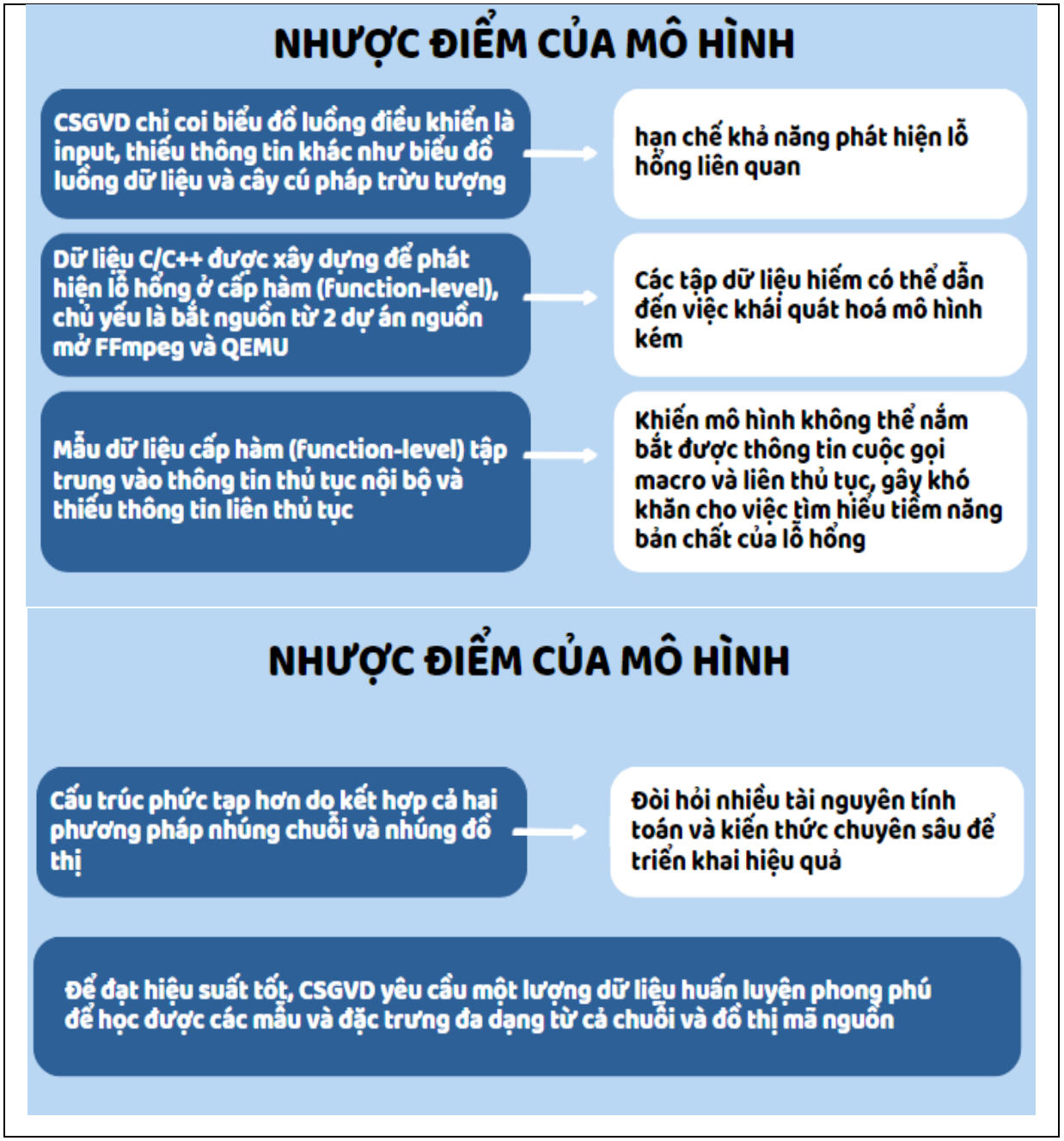
- Kết quả: CSGVD tốt hơn đáng kể so với các mô hình cơ sở và đạt độ chính xác cao nhất (Accuracy) là 64.46%, Recall 58.99% và F1-score 60.39% trên tập dữ liệu.
- Ưu điểm của mô hình:
- CSGVD sử dụng PE-BL module và graph neural network nên bảo toàn thông tin trình tự trong mã và kết hợp thông tin luồng dữ liệu vốn có của nó tốt hơn.
- Kết hợp cả nhúng chuỗi và nhúng đồ thị: Phương pháp CSGVD kết hợp cả việc nhúng chuỗi và nhúng đồ thị để trích xuất đặc trưng của lỗ hổng. Bằng cách kết hợp hai phương pháp này, nó có khả năng nắm bắt cả thông tin ngữ nghĩa và thông tin cấu trúc của mã nguồn.

- Sử dụng học sâu: CSGVD sử dụng các mô hình học sâu như mạng neural đệm (LSTM) và GNN để thực hiện nhúng chuỗi và nhúng đồ thị. Sử dụng học sâu giúp mô hình có khả năng học và mô hình hóa các mẫu phức tạp trong dữ liệu mã nguồn.



- Nhược điểm của mô hình:

- CSGVD chỉ coi biểu đồ luồng điều khiển là input, thiếu thông tin khác như biểu đồ luồng dữ liệu và cây cú pháp trừu tượng, có thể hạn chế khả năng phát hiện lỗ hổng liên quan.
- Bài báo sử dụng dữ liệu C/C++ được xây dựng để phát hiện lỗ hổng ở cấp hàm (function-level), chủ yếu là bắt nguồn từ 2 dự án nguồn mở FFmpeg và QEMU. Các tập dữ liệu hiếm có thể dẫn đến việc khái quát hoá mô hình kém.
- Mẫu dữ liệu cấp hàm (function-level) tập trung vào thông tin thủ tục nội bộ và thiếu thông tin liên thủ tục. Lỗ hổng này khiến mô hình không thể nắm bắt được thông tin cuộc gọi macro và liên thủ tục, gây khó khăn cho việc tìm hiểu tiềm năng bản chất của lỗ hổng.
- Phức tạp hơn trong triển khai: Phương pháp CSGVD có cấu trúc phức tạp hơn so với ReGVD do kết hợp cả hai phương pháp nhúng chuỗi và nhúng đồ thị. Điều này có thể đòi hỏi nhiều tài nguyên tính toán và kiến thức chuyên sâu để triển khai hiệu quả.
- Đòi hỏi dữ liệu huấn luyện phong phú: Để đạt hiệu suất tốt, CSGVD yêu cầu một lượng dữ liệu huấn luyện phong phú để học được các mẫu và đặc trưng đa dạng từ cả chuỗi và đồ thị mã nguồn.



H. Công việc/tính năng/kỹ thuật mà nhóm thực hiện lập trình và triển khai cho demo:

Chúng em thực hiện theo mô hình ReGVD (cắt bớt giai đoạn nhúng trình tự trong CSGVD) như sau:

Giai đoạn tiền xử lý dữ liệu:

- Đầu vào: Đầu vào của ReGVD là mã nguồn được biểu diễn dưới dạng đồ thị, trong đó các thành phần của mã (hàm, biến, lớp, phương thức) được biểu diễn bằng các đỉnh và mối quan hệ giữa chúng được biểu diễn bằng các cạnh của đồ thị.

Giai đoạn nhúng đồ thị:

- Nhúng đỉnh: Đầu tiên, ReGVD sử dụng một phép nhúng đỉnh để biểu diễn thông tin của từng thành phần trong đồ thị mã nguồn. Thông thường, các phép nhúng đỉnh như GraphSAGE, GAT (Graph Attention Network) hoặc GCN (Graph Convolutional Network) được sử dụng để biểu diễn các đặc trưng của đỉnh dựa trên thông tin cục bộ và môi trường xung quanh.
- Nhúng cạnh: Sau đó, ReGVD thực hiện nhúng cạnh để biểu diễn các mối quan hệ giữa các thành phần trong đồ thị. Các phương pháp nhúng cạnh, chẳng hạn như GraphSAGE hoặc GAT, có thể được sử dụng để tính toán biểu diễn dựa trên thông tin của đỉnh kề và trọng số của cạnh.

Giai đoạn trích xuất đặc trưng:

- Kết hợp đặc trưng: ReGVD kết hợp đặc trưng của các đỉnh và cạnh để tạo ra biểu diễn tổng thể cho mỗi thành phần trong đồ thị mã nguồn. Phương pháp thông thường là kết hợp các đặc trưng bằng cách ghép nối, cộng hoặc trung bình hóa để tạo ra biểu diễn tổng hợp.

Giai đoạn phát hiện lỗ hổng:

- Phân loại: Cuối cùng, ReGVD sử dụng một mô hình phân loại hoặc học máy để phân loại và phát hiện lỗ hổng trong mã nguồn dựa trên biểu diễn tổng hợp. Các mô hình phân loại thường được huấn luyện trên tập dữ liệu được gán nhãn để nhận diện các mẫu lỗ hổng và phân loại các thành phần mã nguồn.

Tóm lại, ReGVD sử dụng mạng neural đồ thị để nhúng và trích xuất đặc trưng từ đồ thị mã nguồn, bao gồm nhúng đỉnh và nhúng cạnh. Sau đó, các đặc trưng được kết hợp và sử dụng để phân loại và phát hiện lỗ hổng trong mã nguồn.

Vậy phương pháp này có những ưu và nhược điểm gì so với CSGVD?

Ưu điểm của ReGVD so với CSGVD:

- Sử dụng mạng neural đồ thị: ReGVD tận dụng hiệu quả kiến thức về đồ thị trong mã nguồn bằng cách sử dụng mạng neural đồ thị (GNN) để nhúng và trích xuất đặc trưng từ đồ thị mã nguồn. Điều này giúp ReGVD có khả năng phát hiện mối quan hệ phức tạp giữa các thành phần mã nguồn và cấu trúc chương trình.
- Tính toán biểu diễn đỉnh và cạnh: ReGVD không chỉ nhúng đỉnh mà còn nhúng cạnh trong đồ thị mã nguồn. Điều này cho phép ReGVD nắm bắt thông tin về mối quan hệ giữa các thành phần và trọng số của cạnh, tăng cường khả năng trích xuất đặc trưng và phát hiện lỗ hổng.

Ưu điểm của CSGVD so với ReGVD:

- Kết hợp nhúng chuỗi và nhúng đồ thị: CSGVD kết hợp cả việc nhúng chuỗi và nhúng đồ thị để trích xuất đặc trưng. Việc kết hợp hai phương pháp này cho phép CSGVD tận dụng thông tin ngữ nghĩa từ chuỗi và cấu trúc từ đồ thị mã nguồn, cung cấp cái nhìn toàn diện hơn về mã nguồn và tăng cường khả năng phát hiện lỗ hổng.
- Sử dụng mô hình học sâu cho nhúng chuỗi: CSGVD có thể sử dụng các mô hình học sâu như LSTM để nhúng chuỗi và biểu diễn thông tin ngữ nghĩa. LSTM có khả năng lưu trữ thông tin về sự tương quan giữa các từ trong chuỗi, giúp CSGVD hiểu được ngữ cảnh và ý nghĩa của từng thành phần mã nguồn.

Kết quả khi đánh giá trên tập dataset từ CodeXGLUE (https://github.com/microsoft/CodeXGLUE), ban đầu bao gồm 27318 mẫu, sau đó được tổ chức và chia lại thành train/validation/test với tỉ lệ 8:1:1.

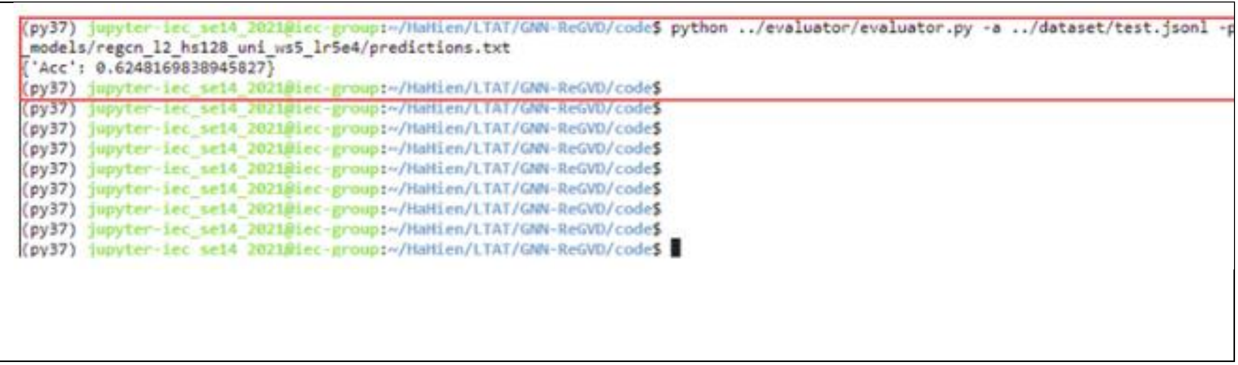
```
(py37) jupyter-iec_se14_2021@iec-group:~/HaHien/LTAT/GNN-ReGVD/code$ CUDA_VISIBLE_DEVICES="0" python run.py --output_dir=./saved_models
/regcn_l2_hs128_uni_ws5_lr5e4 --model_type=roberta --tokenizer_name=microsoft/graphcodebert-base --model_name_or_path=microsoft/graphco
debert-base \
        --do_eval --do_test --do_train --train_data_file=../dataset/train.jsonl --eval_data_file=../dataset/valid.jsonl --test_data_fil
e=../dataset/test.jsonl \
        --block_size 400 --train_batch_size 128 --eval_batch_size 128 --max_grad_norm 1.0 --evaluate_during_training \
        --gnn ReGCN --learning_rate 5e-4 --epoch 100 --hidden_size 128 --num_GNN_layers 2 --format uni --window_size 5 \
        --seed 123456 2>&1 | tee $logp/training_log.txt
tee: /training_log.txt: Permission denied
12/23/2023 10:59:34 - WARNING - __main__ -   Process rank: -1, device: cuda, n_gpu: 1, distributed training: False, 16-bits training: F
alse
Some weights of the model checkpoint at microsoft/graphcodebert-base were not used when initializing RobertaForSequenceClassification:
['lm_head.layer_norm.bias', 'lm_head.dense.bias', 'lm_head.layer_norm.weight', 'lm_head.bias', 'lm_head.decoder.bias', 'lm_head.dense.w
eight', 'lm_head.decoder.weight']
- This IS expected if you are initializing RobertaForSequenceClassification from the checkpoint of a model trained on another task or w
ith another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing RobertaForSequenceClassification from the checkpoint of a model that you expect to be ex
actly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
Some weights of RobertaForSequenceClassification were not initialized from the model checkpoint at microsoft/graphcodebert-base and are
 newly initialized: ['classifier.dense.weight', 'classifier.dense.bias', 'classifier.out_proj.weight', 'classifier.out_proj.bias']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
12/23/2023 10:59:37 - INFO - __main__ -   Training/evaluation parameters Namespace(adam_epsilon=1e-08, alpha_weight=1.0, att_op='mul',
block_size=400, cache_dir='', config_name='', device=device(type='cuda'), do_eval=True, do_lower_case=False, do_test=True, do_train=Tru
e, epoch=100, eval_all_checkpoints=False, eval_batch_size=128, eval_data_file='../dataset/valid.jsonl', evaluate_during_training=True,
feature_dim_size=768, format='uni', fp16=False, fp16_opt_level='O1', gnn='ReGCN', gradient_accumulation_steps=1, hidden_size=128, learn
ing_rate=0.0005, local_rank=-1, logging_steps=50, max_grad_norm=1.0, max_steps=-1, mlm=False, mlm_probability=0.15, model='GNNs', model
_name_or_path='microsoft/graphcodebert-base', model_type='roberta', n_gpu=1, no_cuda=False, num_GNN_layers=2, num_classes=2, num_train_
epochs=1.0, output_dir='./saved_models/regcn_l2_hs128_uni_ws5_lr5e4', overwrite_cache=False, overwrite_output_dir=False, per_gpu_eval_b
atch_size=128, per_gpu_train_batch_size=128, remove_residual=False, save_steps=50, save_total_limit=None, seed=123456, server_ip='', se
rver_port='', start_epoch=0, start_step=0, test_data_file='../dataset/test.jsonl', tokenizer_name='microsoft/graphcodebert-base', train
_batch_size=128, train_data_file='../dataset/train.jsonl', training_percent=1.0, warmup_steps=0, weight_decay=0.0, window_size=5)
12/23/2023 11:00:14 - INFO - __main__ -   *** Total Sample ***
12/23/2023 11:00:14 - INFO - __main__ -         Total: 21854    selected: 21854 percent: 1.0
12/23/2023 11:00:14 - INFO - __main__ -   *** Sample ***
```

```
12/23/2023 11:00:14 - INFO - __main__ -   *** Total Sample ***
12/23/2023 11:00:14 - INFO - __main__ -         Total: 21854    selected: 21854 percent: 1.0
12/23/2023 11:00:14 - INFO - __main__ -   *** Sample ***
12/23/2023 11:00:14 - INFO - __main__ -   Total sample
12/23/2023 11:00:14 - INFO - __main__ -   idx: 0
12/23/2023 11:00:14 - INFO - __main__ -   label: 0
12/23/2023 11:00:14 - INFO - __main__ -   input_tokens: ['<s>', 'static', '_av', '_', 'cold', '_int', '_v', 'd', 'ade', 'c', '_', 'init
', '(', 'AV', 'Cod', 'ec', 'Context', '_*', 'av', 'ctx', ')', '_{', '_V', 'D', 'AD', 'ec', 'oder', 'Context', '_*', 'ctx', '_=', '_av',
 'ctx', '->', 'priv', '_', 'data', ';', '_struct', '_v', 'da', '_', 'context', '_=', 'v', 'da', '_', 'ctx', '_=', '_&', 'ctx', '->', 'v
', 'da', '_', 'ctx', ';', '_OS', 'Status', '_status', ';', '_int', '_ret', ';', '_c', 'tx', '->', 'h', '264', '_', 'initialized', '_=',
 '_0', ';', '_/*', '_init', '_p', 'ix', '_', 'fm', 'ts', '_of', '_codec', '_*/', '_if', '_(!', 'ff', '_', 'h', '264', '_', 'v', 'da', '
', 'dec', 'oder', '_', 'p', 'ix', '_', 'fm', 'ts', ')', '_{', '_if', '_(', 'k', 'C', 'FC', 'ore', 'Found', 'ation', 'Version', 'Number
', '_<', '_k', 'C', 'FC', 'ore', 'Found', 'ation', 'Version', 'Number', '10', '_', '7', ')', '_ff', '_', 'h', '264', '_', 'v', 'da', '_
', 'dec', 'oder', '_', 'p', 'ix', '_', 'fm', 'ts', '_=', '_v', 'da', '_', 'p', 'ix', 'fm', 'ts', '_', 'pri', 'or', '_', '10', '_', '7',
 ';', '_else', '_ff', '_', 'h', '264', '_', 'v', 'da', '_', 'dec', 'oder', '_', 'p', 'ix', '_', 'fm', 'ts', '_=', '_v', 'da', '_', 'p',
 'ix', 'fm', 'ts', ';', '_}', '_/*', '_init', '_v', 'da', '_*/', '_mem', 'set', '(', 'v', 'da', '_', 'ctx', ',', '_0', ',', '_sizeof',
'(', 'struct', '_v', 'da', '_', 'context', '));', '_v', 'da', '_', 'ctx', '->', 'width', '_=', '_av', 'ctx', '->', 'width', ';', '_v',
'da', '_', 'ctx', '->', 'height', '_=', '_av', 'ctx', '->', 'height', ';', '_v', 'da', '_', 'ctx', '->', 'format', '_=', '_-', 'av', 'c
', '1', '";', '_v', 'da', '_', 'ctx', '->', 'use', '_', 'sync', '_', 'dec', 'oding', '_=', '_1', ';', '_v', 'da', '_', 'ctx', '->', 'us
e', '_', 'ref', '_', 'buffer', '_=', '_1', ';', '_c', 'tx', '->', 'p', 'ix', '_', '_', 'f', 'mt', '_=', '_av', 'ctx', '->', 'get', '_', 'for
mat', '(', 'av', 'ctx', '->', '_av', 'ctx', '->', 'cod', 'ec', '->', 'p', 'ix', '_', 'fm', 'ts', ');', '_switch', '_(', 'ctx', '->', 'p'
, 'ix', '_', 'f', 'mt', ')', '_{', '_case', '_AV', '_', 'P', 'IX', '_', 'F', 'MT', '_', 'U', 'Y', 'V', 'Y', '422', ':', '_v', 'da', '_'
, 'ctx', '->', 'cv', '_', 'p', 'ix', '_', 'f', 'mt', '_=', '_"', '2', 'V', 'uy', '";', '_break', ';', '_case', '_AV', '_',
'P', 'IX', '_', 'F', 'MT', '_', 'Y', 'U', 'Y', 'V', '422', ':', '_v', 'da', '_', 'ctx', '->', 'cv', '_', 'p', 'ix', '_', 'f', 'mt', '_
', 'type', '_=', '_"', 'yu', 'vs', '";', '_break', ';', '_case', '_AV', '_', 'P', 'IX', '_', 'F', 'MT', '_', 'NV', '12', ':', '</s>']
12/23/2023 11:00:14 - INFO - __main__ -   input_ids: 0 42653 6402 1215 33912 6979 748 417 1829 438 1215 25153 1640 10612 47436 3204 485
22 1009 1469 49575 43 25522 468 495 2606 3204 15362 48522 1009 49575 5457 6402 49575 46613 25943 1215 23687 131 29916 748 6106 1215 467
96 1009 705 6106 1215 49575 5457 359 49575 46613 705 6106 1215 49575 131 8192 47731 2194 131 6979 5494 131 740 43820 46613 298 29137 12
15 49722 5457 321 131 48565 45511 181 3181 1215 40523 1872 9 45797 48404 114 48209 3145 1215 298 29137 1215 705 6106 1215 11127 15362 4
 642 3181 1215 40523 1872 43 25522 114 36 330 347 5268 1688 29991 1258 47322 43623 28696 449 347 5268 1688 29991 1258 47322 43623 698 1
```

```
12/23/2023 11:00:17 - INFO - __main__ -   Num Epochs = 100
12/23/2023 11:00:17 - INFO - __main__ -   Instantaneous batch size per GPU = 128
12/23/2023 11:00:17 - INFO - __main__ -   Total train batch size (w. parallel, distributed & accumulation) = 128
12/23/2023 11:00:17 - INFO - __main__ -   Gradient Accumulation steps = 1
12/23/2023 11:00:17 - INFO - __main__ -   Total optimization steps = 17100
using default unweighted graph
12/23/2023 11:02:38 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:02:38 - INFO - __main__ -     Num examples = 2732
12/23/2023 11:02:38 - INFO - __main__ -     Batch size = 128
12/23/2023 11:02:52 - INFO - __main__ -     eval_loss = 0.6884
12/23/2023 11:02:52 - INFO - __main__ -     eval_acc = 0.5655
12/23/2023 11:02:52 - INFO - __main__ -     ********************
12/23/2023 11:02:52 - INFO - __main__ -     Best acc:0.5655
12/23/2023 11:02:52 - INFO - __main__ -     ********************
12/23/2023 11:02:53 - INFO - __main__ -   Saving model checkpoint to ./saved_models/regcn_l2_hs128_uni_ws5_lr5e4/checkpoint-best
del.bin
12/23/2023 11:02:53 - INFO - __main__ -   epoch 0 loss 0.69214
12/23/2023 11:05:12 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:05:12 - INFO - __main__ -     Num examples = 2732
12/23/2023 11:05:12 - INFO - __main__ -     Batch size = 128
12/23/2023 11:05:27 - INFO - __main__ -     eval_loss = 0.6876
12/23/2023 11:05:27 - INFO - __main__ -     eval_acc = 0.5655
12/23/2023 11:05:27 - INFO - __main__ -   epoch 1 loss 0.69096
12/23/2023 11:07:47 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:07:47 - INFO - __main__ -     Num examples = 2732
12/23/2023 11:07:47 - INFO - __main__ -     Batch size = 128
12/23/2023 11:08:01 - INFO - __main__ -     eval_loss = 0.6803
12/23/2023 11:08:01 - INFO - __main__ -     eval_acc = 0.56
```

```
12/23/2023 11:51:32 - INFO - __main__ -   epoch 20 loss 0.52619
12/23/2023 11:53:38 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:53:38 - INFO - __main__ -     Num examples = 2732
12/23/2023 11:53:38 - INFO - __main__ -     Batch size = 128
12/23/2023 11:53:51 - INFO - __main__ -     eval_loss = 0.6342
12/23/2023 11:53:51 - INFO - __main__ -     eval_acc = 0.6453
12/23/2023 11:53:51 - INFO - __main__ -     ********************
12/23/2023 11:53:51 - INFO - __main__ -     Best acc:0.6453
12/23/2023 11:53:51 - INFO - __main__ -     ********************
12/23/2023 11:53:52 - INFO - __main__ -   Saving model checkpoint to ./saved_models/regcn_l2_hs128_uni_ws5_lr5e4/checkpoint-best-
del.bin
12/23/2023 11:53:52 - INFO - __main__ -   epoch 21 loss 0.51762
12/23/2023 11:55:58 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:55:58 - INFO - __main__ -     Num examples = 2732
12/23/2023 11:55:58 - INFO - __main__ -     Batch size = 128
12/23/2023 11:56:11 - INFO - __main__ -     eval_loss = 0.6428
12/23/2023 11:56:11 - INFO - __main__ -     eval_acc = 0.6332
12/23/2023 11:56:11 - INFO - __main__ -   epoch 22 loss 0.51013
```

```
12/23/2023 17:05:35 - INFO - __main__ -     eval_loss = 1.2147
12/23/2023 17:05:35 - INFO - __main__ -     eval_acc = 0.6069
12/23/2023 17:05:35 - INFO - __main__ -   epoch 96 loss 0.23299
12/23/2023 17:09:31 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 17:09:31 - INFO - __main__ -     Num examples = 2732
12/23/2023 17:09:31 - INFO - __main__ -     Batch size = 128
12/23/2023 17:09:58 - INFO - __main__ -     eval_loss = 1.2137
12/23/2023 17:09:58 - INFO - __main__ -     eval_acc = 0.6054
12/23/2023 17:09:58 - INFO - __main__ -   epoch 97 loss 0.22439
12/23/2023 17:13:54 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 17:13:54 - INFO - __main__ -     Num examples = 2732
12/23/2023 17:13:54 - INFO - __main__ -     Batch size = 128
12/23/2023 17:14:21 - INFO - __main__ -     eval_loss = 1.2309
12/23/2023 17:14:21 - INFO - __main__ -     eval_acc = 0.6058
12/23/2023 17:14:21 - INFO - __main__ -   epoch 98 loss 0.22609
12/23/2023 17:18:19 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 17:18:19 - INFO - __main__ -     Num examples = 2732
12/23/2023 17:18:19 - INFO - __main__ -     Batch size = 128
12/23/2023 17:18:46 - INFO - __main__ -     eval_loss = 1.2211
12/23/2023 17:18:46 - INFO - __main__ -     eval_acc = 0.6098
12/23/2023 17:18:46 - INFO - __main__ -   epoch 99 loss 0.22809
12/23/2023 17:18:57 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 17:18:57 - INFO - __main__ -     Num examples = 2732
12/23/2023 17:18:57 - INFO - __main__ -     Batch size = 128
12/23/2023 17:19:24 - INFO - __main__ -   ***** Eval results *****
12/23/2023 17:19:24 - INFO - __main__ -     eval_acc = 0.6453
12/23/2023 17:19:24 - INFO - __main__ -     eval_loss = 0.6342
12/23/2023 17:19:32 - INFO - __main__ -   ***** Running Test *****
12/23/2023 17:19:32 - INFO - __main__ -     Num examples = 2732
12/23/2023 17:19:32 - INFO - __main__ -     Batch size = 128
12/23/2023 17:19:58 - INFO - __main__ -   ***** Test results *****
12/23/2023 17:19:58 - INFO - __main__ -     test_acc = 0.6248
```



I. Các khó khăn, thách thức hiện tại khi thực hiện:

- Khó khăn đầu tiên là bài báo khá là nhiều giai đoạn và đòi hỏi kiến thức chuyên sâu mới có thể làm lại được.
- Khó khăn nhất là bài báo không hề có bất kỳ thông tin nào về mã nguồn hay ngôn ngữ mà họ thực hiện.
- Là một bài báo còn quá mới, mới public năm 2023 nên chưa có nhiều tài liệu để tụi em tiếp cận.
- Kỳ này đối với tụi em là lần đầu tiên có nhiều đồ án như vậy nên tụi em hơi đuối ạ.

3. TỰ ĐÁNH GIÁ MỨC ĐỘ HOÀN THÀNH SO VỚI KẾ HOẠCH THỰC HIỆN:

90%

4. NHẬT KÝ PHÂN CÔNG NHIỆM VỤ:

STT	Công việc	Phân công nhiệm vụ
1	Đọc báo và tìm hiểu các giai đoạn trong bài báo	Cả nhóm
2	Thực hiện test Joern để tìm cách thực hiện trích xuất và thử gộp các node trên 1 dòng lại thành 1 node	Hà Thị Thu Hiền
3	Tìm kiếm các giải pháp cho giai đoạn nhúng node và nhúng đồ thị	Phạm Ngọc Thơ, Nguyễn Ngọc Nhung
4	Tiếp cận các bài báo khác như FUNDED, REGVD để tìm ra phương pháp phù hợp so sánh ưu nhược điểm, các điểm giống nhau và đánh giá hiệu suất của các model	Cả nhóm
5	Chạy thực nghiệm model và tiến hành đánh giá so sánh model	Hà Thị Thu Hiền

BÁO CÁO TỔNG KẾT CHI TIẾT

Phần bên dưới của báo cáo này là tài liệu báo cáo tổng kết - chi tiết của nhóm thực hiện cho đề tài này.

Qui định: Mô tả các bước thực hiện/ Phương pháp thực hiện/Nội dung tìm hiểu (Ảnh chụp màn hình, số liệu thống kê trong bảng biểu, có giải thích)

A. Phương pháp thực hiện

<Trình bày kiến trúc, thành phần của hệ thống trong bài báo>

1. Approach overview

- Trong phần này, bài báo sẽ trình bày một mô hình mạng thần kinh kết hợp nhúng trình tự và biểu đồ (combined sequence and graph embedding neural network model), được đặt tên là CSGVD, để xác định xem function-level code có dễ bị tấn công hay không.
- Trong CSGVD, các phụ thuộc kiểm soát được xác định giữa các câu lệnh có thể đóng vai trò đầy đủ dưới dạng thông tin theo ngữ cảnh cho nhiệm vụ function-level vulnerability detection. Hơn nữa, bài báo sử dụng phương pháp núng trình tự (sequence embedding) để trích xuất các đặc điểm ngữ nghĩa cục bộ (local semantic features) của mã khi biểu diễn nhúng node của biểu đồ. Trong khi đó, bài báo xây dựng kiến trúc dựa trên mạng thần kinh biểu đồ (graph neural netwok-based architecture) để tận dụng tốt hơn các mối quan hệ cấu trúc trong câu lệnh và giữa các câu lệnh, như trong hình 1 sau:

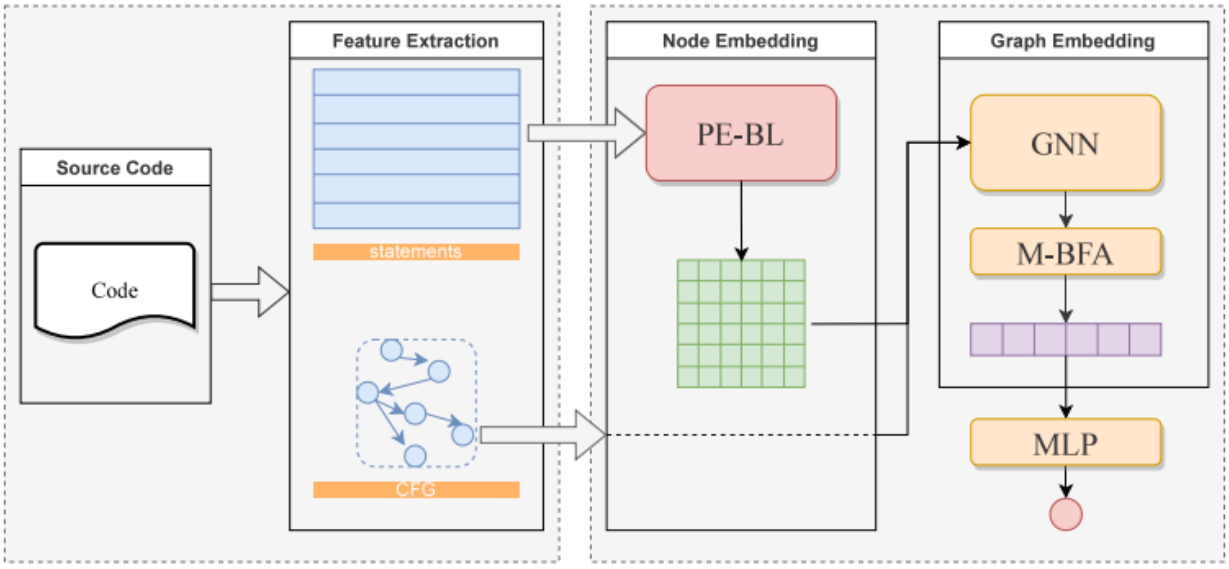


Fig. 1. Overall framework.

PE-BL: Pre-Embedding – BiLSTM

BiLSTM là viết tắt của **Bidirectional Long Short-Term Memory**[1]. Nó là một kiến trúc mạng nơ-ron sử dụng trong học sâu để xử lý các chuỗi dữ liệu. BiLSTM bao gồm hai mạng LSTM chạy song song với nhau, một mạng xử lý chuỗi theo chiều thuận và một mạng xử lý chuỗi theo chiều ngược lại. Kiến trúc này cho phép mạng học được thông tin từ cả hai phía của chuỗi đầu vào, giúp cải thiện khả năng dự đoán của mạng

- Cấu trúc tổng thể được chia chủ yếu thành hai phần, trích xuất đặc trưng (feature attraction) ở bên trái và mô hình mạng thần kinh (neural network model) ở bên phải. Kiến trúc mạng thần kinh (The neural network architecture) được minh hoạ ở trong hình 2 sau:

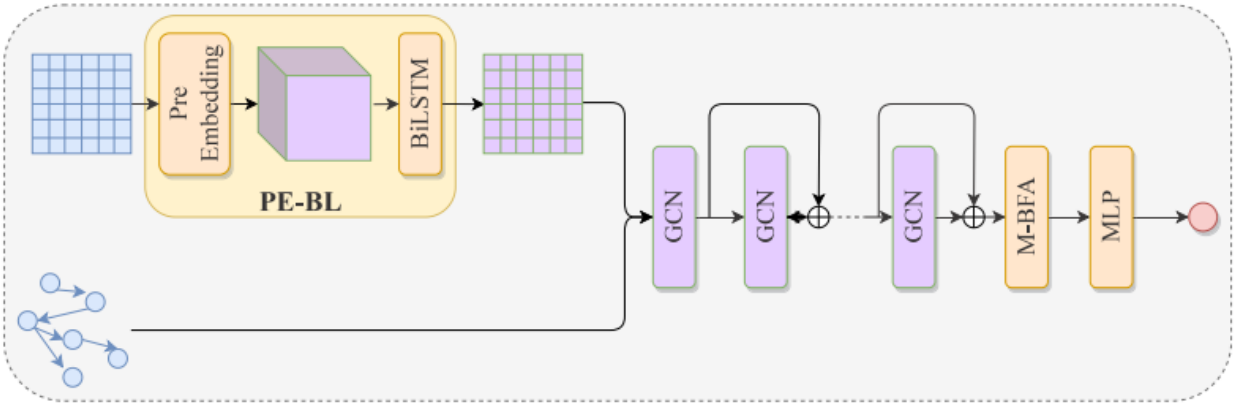


Fig. 2. Model architecture.

1.1 Feature extraction

- Để huấn luyện neural network, chúng ta cần phải chuyển đổi raw function-level source code thành định dạng dữ liệu được neural network model chấp nhận. Trong bài báo sử dụng GitHub - joernio/joern: Open-source code analysis platform for C/C++/Java/Binary/Javascript/Python/Kotlin based on code property graphs. Discord https://discord.gg/vv4MH284Hc để **phân tích mã nguồn và trích xuất biểu đồ luồng điều khiển**. Vì biểu đồ do Joern trích xuất quá lớn nên bài báo hợp nhất các node được trích xuất theo hàng. Bằng cách nén biểu đồ và chuyển đổi biểu đồ luồng điều khiển ban đầu thành biểu đồ luồng điều khiển ở mức câu lệnh (stament-level control flow graph), số node trong biểu đồ nén sẽ <= số dòng mã, điều này sẽ làm giảm đáng kể thời gian và tài nguyên tính toán.

- Ví dụ: Khi dùng Joern để phân tích một câu lệnh mẫu Ci như:

$$for\ (expression\_1;\ expression\_2;\ expression\_3) \qquad (1)$$

- Joern sẽ chia câu lệnh này thành 3 node khác nhau (như trong bảng 1). Nó có thể làm cho cfgi trương ứng với mẫu ci rất lớn. Để giảm kích thước của cfgi, bài báo hợp nhất các node nằm trên cùng một dòng tương ứng với raw code thành một node mới vnew. Trong khi chúng tôi sử dụng câu lệnh gốc (1) tương ứng với dòng làm nội dung của node vnew. Hơn nữa vnew sẽ kế thừa tất cả các mối quan hệ biên
(edge relations).

1.2 Node embedding

- Đối với các nhiệm vụ tiếp theo, điều quan trọng là tạo ra một biểu diễn mã đầy đủ thông tin và toàn diện của một mã nguồn nhất định. Như được hiển thị trong hình 2, bài báo đề xuất PE-BL module, bao gồm lớp Pre-Embedding và lớp BiLSTM. Cụ thể, thay vì lớp nhúng truyền thống, bài báo giới thiệu mô hình ngôn ngữ được đào tạo trước (pre-trained programming language model), CodeBERT, để khởi tạo lớp nhúng. Bài báo tạo các trọng số của lớp nhúng với các trọng số nhúng từ (word embedding weights) đã được huấn luyện của nó.
- Bài báo sử dụng một hàm (function) duy nhất của mã nguồn làm đầu vào thô (raw input). Chúng ta chia hàm (function) thành một tập hợp các câu lệnh S = {S1, S2, S3, ..., Sn) và CFG ở cấp độ câu lệnh (statement-level CFG), trong đó si tương ứng với node vi của CFG. Mỗi câu lệnh trước tiên phải được mã hoá thông qua tokenized BPE được đào tạo trước của CodeBERT (Each statement is firstly tokenized via CodeBERT's pretrained BPE tokenizer).

$$tokens_i = BPE - Tokenizer(s_i). \quad (2)$$

- Sau đó, CSGVD lấy một lớp nhúng, lớp này sử dụng các trọng số của lớp nhúng từ (word embedding layer weights) của CodeBERT để khởi tạo các trọng số của bản thân, nhằm thu được biểu diễn vector $\boldsymbol{e_{ij}} \in E_i = \{\boldsymbol{e_{i1}}, \boldsymbol{e_{i2}}, \boldsymbol{e_{i3}}, \ldots, \boldsymbol{e_{in}}\}$ của mỗi mã thông báo (each token).

$$E_i = Embedding(tokens_i). \quad (3)$$

- Cuối cùng, LSTM hai chiều được sử dụng để hợp nhất thông tin ngữ nghĩa cục bộ của mã và thu được biểu diễn vector xi của câu lệnh si tương ứng với nút vi:

$$\overrightarrow{\boldsymbol{h_i}}, \overleftarrow{\boldsymbol{h_i}} = BiLSTM(E_i) \quad (4)$$

$$\boldsymbol{x_i} = Sum(\overrightarrow{\boldsymbol{h_i}}, \overleftarrow{\boldsymbol{h_i}}) \quad (5)$$

where $\overrightarrow{\boldsymbol{h_i}}, \overleftarrow{\boldsymbol{h_i}}$ is the final output of BiLSTM.

1.3 Residual graph neural network (Mạng neural đồ thị dư)

- Bài báo tập trung vào thông tin phụ thuộc điều khiển mà bài báo xây dựng mô hình mạng neural đồ thị có kết nối dư (graph neural network model with residual connectivity). Không giống như mạng neural tuần tự (sequential neural network), mạng neural đồ thị (graph neural network – CNNs) tận dụng cơ chế khuếch tán thông tin để tìm hiểu dữ liệu có cấu trúc biểu đồ, cập nhật trạng thái node theo kết nối biểu đồ. Như được hiển thị trong hình 2, mạng tích chập đồ thị (graph convolution network – GCN) được sử dụng để tìm hiểu thông tin luồng điều khiển từ biểu đồ.
- Lớp GCN trước tiên sẽ lấy câu lệnh đầu ra n được nhúng từ BiLSTM cùng với các cạnh giữa mỗi node. Thông tin cấu trúc biểu đồ của mã nguồn, bao gồm thông tin về node và cạnh, được trích xuất và đưa vào GCN. Đặc biệt, bài báo thêm các liên kết tự lặp (seft-looping links) vào từng node trong biểu đồ. CSGVD sẽ truyền bá thông tin theo trạng thái của các node và mối quan hệ luồng điều khiển giữa các node lân cận theo cách tăng dần. Kiến trúc mô hình (the model architecture) bao gồm 2 mạng GCN (graph convolution network – mạng tích chập đồ thị) với khả năng kết nối dư. Nhìn chung, GCN dư được xác định bằng cách sử dụng kết nối bỏ qua trên các lớp GCN khác nhau trong biểu thứ (6). Trong đó:
  - l: số lớp hiện tại
  - $H^{(l)} = \{\boldsymbol{h}_1^{(l)}, \boldsymbol{h}_2^{(l)}, \boldsymbol{h}_3^{(l)}, \ldots, \boldsymbol{h}_n^{(l)}\}$: là ma trận trạng thái ẩn của lớp thứ l
  - A: là ma trận kề

$$H^{(l+1)} = GCN(H^{(l)}, A) + H^{(l)} \quad (6)$$

Specially, in CSGVD,

$$H^{(0)} = X \quad (7)$$

$$H^{(1)} = GCN(H^{(0)}, A) \quad (8)$$

wherein $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \ldots, \boldsymbol{x}_n\}$ is the node embedding matrix.

1.4 Graph embedding

- Sử dụng multi-layer GNN với khả năng kết nối dư (residual connectivity), bài báo thu được biểu diễn ẩn của từng node trong biểu đồ luồng điều khiển trong khi tổng hợp thông tin node của các node lân cận.
- Trước khi thực hiện nhiệm vụ phân loại biểu đồ (graph classification), chúng ta cũng cần tổng hợp thông tin của các node để thu được biểu diễn vector của biểu đồ. Do đó, như được hiển thị trong hình 3, lấy cảm hứng từ biaffine attention mechanism, bài báo đã đề xuất mean biaffine attention (M-BFA) pooling. Đầu tiên, bài báo giả sử sự tồn tại của supernode vs, supernode này có vai trò chủ đạo đối

với việc tổng hợp thông tin biểu đồ. Thứ hai, bài báo tính điểm chú ý (attention score) của vs cho mỗi node. Cuối cùng, việc tổng hợp có trọng số được thực hiện cho mỗi node dưới dạng biểu diễn vector của biểu đồ. Các bước tính toán được thể hiện trong phương trình sau:

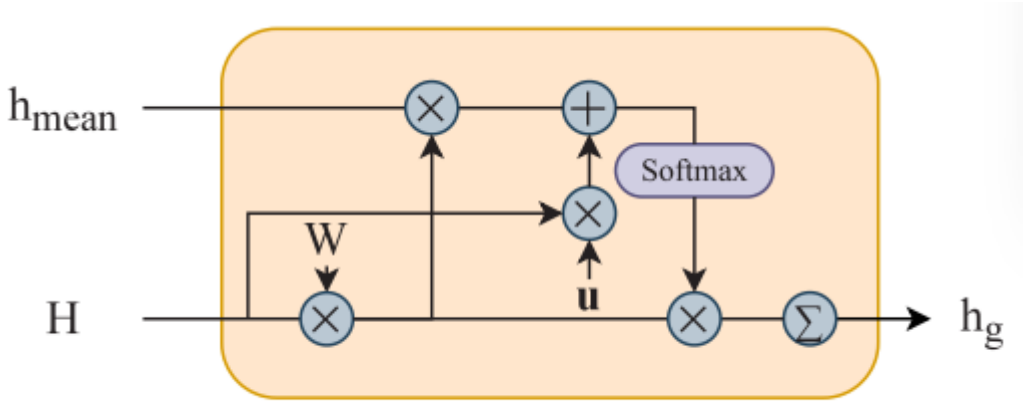


Fig. 3. M-BFA pooling.

$$\boldsymbol{h_{mean}} = \frac{\sum_{i=1}^{n} \boldsymbol{h_i}}{n} \tag{9}$$

$$\boldsymbol{h_{fi}} = \boldsymbol{h_i^\top} \cdot W \tag{10}$$

$$e_i = \boldsymbol{h_{fi}^\top} \cdot \boldsymbol{h_{mean}} + \boldsymbol{h_i^\top} \cdot \boldsymbol{u} \tag{11}$$

$$a_i = \frac{\exp(e_i)}{\sum_{j=0}^{n} \exp(e_j)} \tag{12}$$

$$\boldsymbol{h_g} = \sum_{i=0}^{n} a_i \cdot \boldsymbol{h_{fi}} \tag{13}$$

- W: ma trận trọng số có thể học được (a learnable weight matrix)
- u: vector trọng số có thể học được
- hi: trạng thái ẩn cuối cùng của node vi
- hg: biểu thị cách trình bày vector của đồ thị
- hmean: giá trị trung bình hmean của tất cả các biểu diễn ẩn của node làm biểu diễn ẩn của supernode vs.

1.5 Classifier learning

- CSGVD sử dụng a multilayer perceptron (MLP) như là lớp phân loại, đây là phân loại phổ biến nhất.

- Trong nghiên cứu này, mục tiêu là đào tạo được một mô hình có thể học cách biểu diễn đồ thị từ đồ thị luồng điều khiển ở cấp độ câu lệnh (statement-level control flow graphs)
- 2 lớp MLP được xây dựng trên 2 công thức (14)-(15) để dự đoán hàm đã cho có dễ bị tấn công hay không.

$$\boldsymbol{o} = softmax(\sigma(U \cdot \boldsymbol{h_g}) \cdot V) \quad (14)$$

$$y = argmax(\boldsymbol{o}) \quad (15)$$

where U and V are two learnable weight matrix, $\sigma(\cdot)$ is an activation function and $y \in \{0, 1\}$ denotes the final predicted result.

<Trình bày kiến trúc, thành phần đã thực hiện (nội dung mà nhóm đã thực hiện)>

Chúng em thực hiện theo mô hình ReGVD (cắt bớt giai đoạn nhúng trình tự trong CSGVD) như sau:

Giai đoạn tiền xử lý dữ liệu:

- Đầu vào: Đầu vào của ReGVD là mã nguồn được biểu diễn dưới dạng đồ thị, trong đó các thành phần của mã (hàm, biến, lớp, phương thức) được biểu diễn bằng các đỉnh và mối quan hệ giữa chúng được biểu diễn bằng các cạnh của đồ thị.

Giai đoạn nhúng đồ thị:

- Nhúng đỉnh: Đầu tiên, ReGVD sử dụng một phép nhúng đỉnh để biểu diễn thông tin của từng thành phần trong đồ thị mã nguồn. Thông thường, các phép nhúng đỉnh như GraphSAGE, GAT (Graph Attention Network) hoặc GCN (Graph Convolutional Network) được sử dụng để biểu diễn các đặc trưng của đỉnh dựa trên thông tin cục bộ và môi trường xung quanh.
- Nhúng cạnh: Sau đó, ReGVD thực hiện nhúng cạnh để biểu diễn các mối quan hệ giữa các thành phần trong đồ thị. Các phương pháp nhúng cạnh, chẳng hạn như GraphSAGE hoặc GAT, có thể được sử dụng để tính toán biểu diễn dựa trên thông tin của đỉnh kề và trọng số của cạnh.

Giai đoạn trích xuất đặc trưng:

- Kết hợp đặc trưng: ReGVD kết hợp đặc trưng của các đỉnh và cạnh để tạo ra biểu diễn tổng thể cho mỗi thành phần trong đồ thị mã nguồn. Phương pháp thông thường là kết hợp các đặc trưng bằng cách ghép nối, cộng hoặc trung bình hóa để tạo ra biểu diễn tổng hợp.

Giai đoạn phát hiện lỗ hổng:

- Phân loại: Cuối cùng, ReGVD sử dụng một mô hình phân loại hoặc học máy để phân loại và phát hiện lỗ hổng trong mã nguồn dựa trên biểu diễn tổng hợp. Các mô hình phân loại thường được huấn luyện trên tập dữ liệu được gán nhãn để nhận diện các mẫu lỗ hổng và phân loại các thành phần mã nguồn.

Tóm lại, ReGVD sử dụng mạng neural đồ thị để nhúng và trích xuất đặc trưng từ đồ thị mã nguồn, bao gồm nhúng đỉnh và nhúng cạnh. Sau đó, các đặc trưng được kết hợp và sử dụng để phân loại và phát hiện lỗ hổng trong mã nguồn.

B. Chi tiết cài đặt, hiện thực

<cách cài đặt, lập trình trên máy tính, cấu hình máy tính sử dụng, chuẩn bị dữ liệu, v.v>

- Cài đặt các gói:
  - Python 3.7
  - Pytorch 1.9
  - Transformer 4.4
- Thực nghiệm trên server có cấu hình:

```
jupyter-iec_se14_2021@iec-group:~$ uname -a
Linux iec-group 5.15.0-91-generic #101-Ubuntu SMP Tue Nov 14 13:30:08 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
jupyter-iec_se14_2021@iec-group:~$ nvidia-smi
Tue Jan  2 17:57:50 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 545.23.06              Driver Version: 545.23.06    CUDA Version: 12.3     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 3060        On  | 00000000:09:00.0 Off |                  N/A |
| 80%   69C    P2             148W / 170W |   6585MiB / 12288MiB |     90%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce RTX 3060 Ti     On  | 00000000:0A:00.0 Off |                  N/A |
| 56%   73C    P2             190W / 220W |   5643MiB /  8192MiB |     96%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A   2848755      C   /opt/tljh/user/bin/python                  6578MiB |
|    1   N/A  N/A   2909359      C   /opt/tljh/user/bin/python                   742MiB |
|    1   N/A  N/A   2989732      C   /opt/tljh/user/bin/python                  4892MiB |
+---------------------------------------------------------------------------------------+
jupyter-iec_se14_2021@iec-group:~$
```

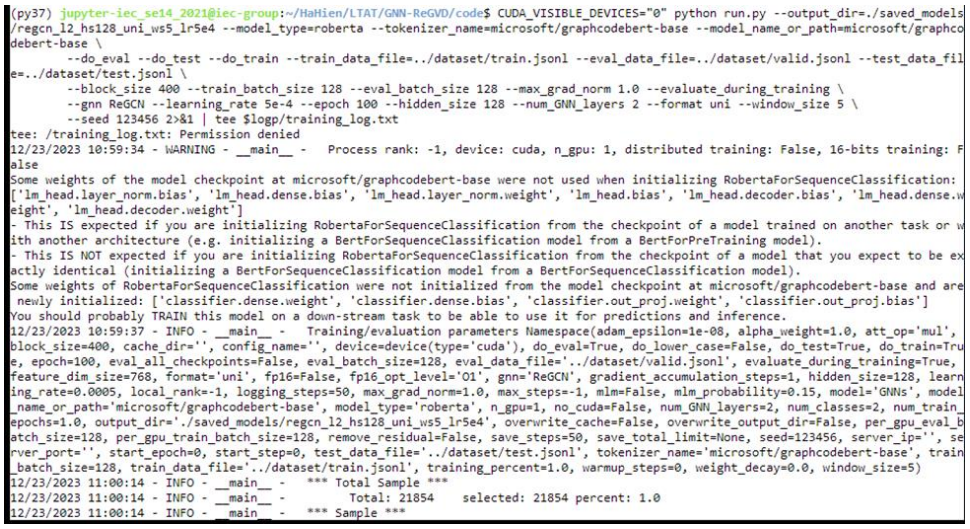
- Chuẩn bị dữ liệu:
  - Dataset: **https://github.com/microsoft/CodeXGLUE**

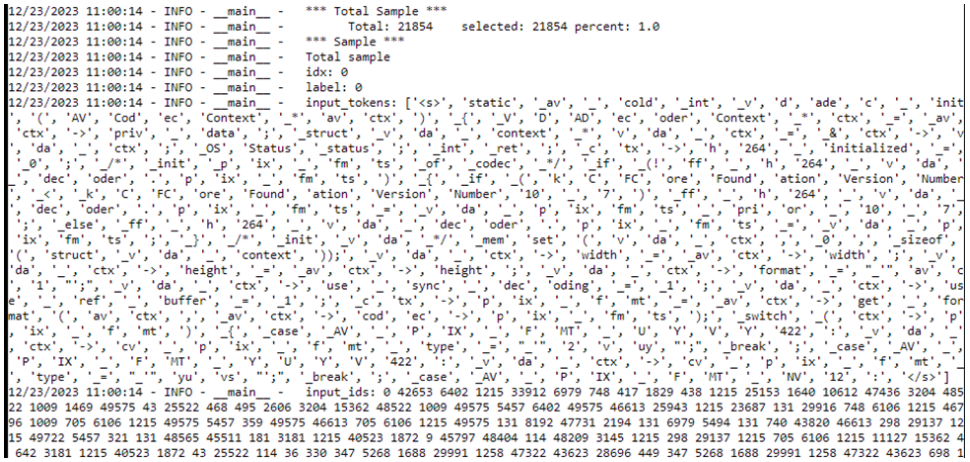
- **Ban đầu bao gồm 27318 mẫu, sau đó được tổ chức và chia lại thành train/validation/test với tỉ lệ 8:1:1**

C. Kết quả thực nghiệm

<mô tả hình ảnh về thực nghiệm, bảng biểu số liệu thống kê từ thực nghiệm, nhận xét về kết quả thu được.>

- Kết quả khi đánh giá trên tập dataset từ CodeXGLUE (https://github.com/microsoft/CodeXGLUE), ban đầu bao gồm 27318 mẫu, sau đó được tổ chức và chia lại thành train/validation/test với tỉ lệ 8:1:1.





```
12/23/2023 11:00:17 - INFO - __main__ -   Num Epochs = 100
12/23/2023 11:00:17 - INFO - __main__ -   Instantaneous batch size per GPU = 128
12/23/2023 11:00:17 - INFO - __main__ -   Total train batch size (w. parallel, distributed & accumulation) = 128
12/23/2023 11:00:17 - INFO - __main__ -   Gradient Accumulation steps = 1
12/23/2023 11:00:17 - INFO - __main__ -   Total optimization steps = 17100
using default unweighted graph
12/23/2023 11:02:38 - INFO - __main__ -  ***** Running evaluation *****
12/23/2023 11:02:38 - INFO - __main__ -    Num examples = 2732
12/23/2023 11:02:38 - INFO - __main__ -    Batch size = 128
12/23/2023 11:02:52 - INFO - __main__ -    eval_loss = 0.6884
12/23/2023 11:02:52 - INFO - __main__ -    eval_acc = 0.5655
12/23/2023 11:02:52 - INFO - __main__ -    ********************
12/23/2023 11:02:52 - INFO - __main__ -    Best acc:0.5655
12/23/2023 11:02:52 - INFO - __main__ -    ********************
12/23/2023 11:02:53 - INFO - __main__ -   Saving model checkpoint to ./saved_models/regcn_l2_hs128_uni_ws5_lr5e4/checkpoint-best-acc/mo
del.bin
12/23/2023 11:02:53 - INFO - __main__ -   epoch 0 loss 0.69214
12/23/2023 11:05:12 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:05:12 - INFO - __main__ -    Num examples = 2732
12/23/2023 11:05:12 - INFO - __main__ -    Batch size = 128
12/23/2023 11:05:27 - INFO - __main__ -    eval_loss = 0.6876
12/23/2023 11:05:27 - INFO - __main__ -    eval_acc = 0.5655
12/23/2023 11:05:27 - INFO - __main__ -   epoch 1 loss 0.69096
12/23/2023 11:07:47 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:07:47 - INFO - __main__ -    Num examples = 2732
12/23/2023 11:07:47 - INFO - __main__ -    Batch size = 128
12/23/2023 11:08:01 - INFO - __main__ -    eval_loss = 0.6803
12/23/2023 11:08:01 - INFO - __main__ -    eval_acc = 0.56
```

```
12/23/2023 11:51:32 - INFO - __main__ -   epoch 20 loss 0.52619
12/23/2023 11:53:38 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:53:38 - INFO - __main__ -    Num examples = 2732
12/23/2023 11:53:38 - INFO - __main__ -    Batch size = 128
12/23/2023 11:53:51 - INFO - __main__ -    eval_loss = 0.6342
12/23/2023 11:53:51 - INFO - __main__ -    eval_acc = 0.6453
12/23/2023 11:53:51 - INFO - __main__ -    ********************
12/23/2023 11:53:51 - INFO - __main__ -    Best acc:0.6453
12/23/2023 11:53:51 - INFO - __main__ -    ********************
12/23/2023 11:53:52 - INFO - __main__ -   Saving model checkpoint to ./saved_models/regcn_l2_hs128_uni_ws5_lr5e4/checkpoint-best-acc/mo
del.bin
12/23/2023 11:53:52 - INFO - __main__ -   epoch 21 loss 0.51762
12/23/2023 11:55:58 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 11:55:58 - INFO - __main__ -    Num examples = 2732
12/23/2023 11:55:58 - INFO - __main__ -    Batch size = 128
12/23/2023 11:56:11 - INFO - __main__ -    eval_loss = 0.6428
12/23/2023 11:56:11 - INFO - __main__ -    eval_acc = 0.6332
12/23/2023 11:56:11 - INFO - __main__ -   epoch 22 loss 0.51013
```

```
12/23/2023 17:05:35 - INFO - __main__ -    eval_loss = 1.2147
12/23/2023 17:05:35 - INFO - __main__ -    eval_acc = 0.6069
12/23/2023 17:05:35 - INFO - __main__ -   epoch 96 loss 0.23299
12/23/2023 17:09:31 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 17:09:31 - INFO - __main__ -    Num examples = 2732
12/23/2023 17:09:31 - INFO - __main__ -    Batch size = 128
12/23/2023 17:09:58 - INFO - __main__ -    eval_loss = 1.2137
12/23/2023 17:09:58 - INFO - __main__ -    eval_acc = 0.6054
12/23/2023 17:09:58 - INFO - __main__ -   epoch 97 loss 0.22439
12/23/2023 17:13:54 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 17:13:54 - INFO - __main__ -    Num examples = 2732
12/23/2023 17:13:54 - INFO - __main__ -    Batch size = 128
12/23/2023 17:14:21 - INFO - __main__ -    eval_loss = 1.2309
12/23/2023 17:14:21 - INFO - __main__ -    eval_acc = 0.6058
12/23/2023 17:14:21 - INFO - __main__ -   epoch 98 loss 0.22609
12/23/2023 17:18:19 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 17:18:19 - INFO - __main__ -    Num examples = 2732
12/23/2023 17:18:19 - INFO - __main__ -    Batch size = 128
12/23/2023 17:18:46 - INFO - __main__ -    eval_loss = 1.2211
12/23/2023 17:18:46 - INFO - __main__ -    eval_acc = 0.6098
12/23/2023 17:18:46 - INFO - __main__ -   epoch 99 loss 0.22809
12/23/2023 17:18:57 - INFO - __main__ -   ***** Running evaluation *****
12/23/2023 17:18:57 - INFO - __main__ -    Num examples = 2732
12/23/2023 17:18:57 - INFO - __main__ -    Batch size = 128
12/23/2023 17:19:24 - INFO - __main__ -   ***** Eval results *****
12/23/2023 17:19:24 - INFO - __main__ -    eval_acc = 0.6453
12/23/2023 17:19:24 - INFO - __main__ -    eval_loss = 0.6342
12/23/2023 17:19:32 - INFO - __main__ -   ***** Running Test *****
12/23/2023 17:19:32 - INFO - __main__ -    Num examples = 2732
12/23/2023 17:19:32 - INFO - __main__ -    Batch size = 128
12/23/2023 17:19:58 - INFO - __main__ -   ***** Test results *****
12/23/2023 17:19:58 - INFO - __main__ -    test_acc = 0.6248
```

⇨ Dựa trên kết quả trên thì so sánh hiệu suất của ReGVD và CSGVD ta có được kết quả là: CSGVD vẫn có độ chính xác cao hơn ReVGD nhưng sự chệnh lệch về độ chính xác đó khoảng 2% nên cũng không đáng kể lắm, điều này chứng minh ReGVD vẫn là model đáng để chúng ta sử dụng trong vấn đề phát hiện lỗ hổng mà không cần mất quá nhiều công sức và giai đoạn như CSGVD (về sự so sánh ưu nhược điểm đã có so sánh ở phần trước)

D. Hướng phát triển

<Nêu hướng phát triển tiềm năng của đề tài này trong tương lai. Nhận xét về tính ứng dụng của đề tài>.

Đề tài CSGVD (Combining Sequence and Graph embedding for Vulnerability Detection) đề xuất một phương pháp kết hợp nhúng chuỗi và nhúng đồ thị để phát hiện lỗ hổng trong mã nguồn. Dưới đây là những hướng phát triển tiềm năng và nhận xét về tính ứng dụng của đề tài này trong tương lai:

- Hướng phát triển tiềm năng của CSGVD:
  - Nâng cao hiệu suất phát hiện: Một hướng phát triển tiềm năng là tăng cường hiệu suất phát hiện lỗ hổng của CSGVD. Điều này có thể được thực hiện bằng cách cải tiến phương pháp nhúng chuỗi và nhúng đồ thị, tăng cường mô hình phân loại hoặc sử dụng các kỹ thuật học sâu mới nhất.
  - Mở rộng phạm vi ứng dụng: CSGVD có thể được mở rộng để ứng dụng trong các lĩnh vực khác nhau liên quan đến phân tích mã nguồn và bảo mật thông tin. Ví dụ, ngoài việc phát hiện lỗ hổng, CSGVD có thể được áp dụng để phân loại mã nguồn theo tính bảo mật, tìm kiếm các lỗ hổng có sẵn trong các cơ sở dữ liệu lỗ hổng công khai, hoặc hỗ trợ đánh giá rủi ro bảo mật của mã nguồn.
  - Tích hợp công cụ hỗ trợ: CSGVD có thể được phát triển thành một công cụ hỗ trợ phân tích mã nguồn, cung cấp khả năng phát hiện lỗ hổng trong thời gian thực và tích hợp với các công cụ phân tích mã nguồn hiện có. Điều này sẽ giúp nhà phát triển phần mềm và chuyên gia bảo mật tăng cường quy trình phát hiện lỗ hổng và cải thiện chất lượng mã nguồn.
- Nhận xét về tính ứng dụng của CSGVD:

- CSGVD có tính ứng dụng cao trong việc phát hiện lỗ hổng trong mã nguồn. Bằng cách kết hợp cả nhúng chuỗi và nhúng đồ thị, phương pháp này khai thác được thông tin ngữ nghĩa và cấu trúc của mã nguồn, cung cấp cái nhìn toàn diện và tăng cường khả năng phát hiện lỗ hổng.
- Đối với các tổ chức phát triển phần mềm, CSGVD có thể hỗ trợ trong việc tìm kiếm và sửa lỗi lỗ hổng, giảm nguy cơ bảo mật và tăng cường chất lượng mã nguồn. Ngoài ra, CSGVD cũng có thể hỗ trợ các chuyên gia bảo mật trong việc phân tích mã nguồn, tìm kiếm lỗ hổng mới và đánh giá rủi ro bảo mật của hệ thống.

Tuy nhiên, như bất kỳ phương pháp phát hiện lỗ hổng nào, CSGVD cũng có thể gặp phải một số hạn chế như tỷ lệ phát hiện sai cao hoặc khó áp dụng cho các ngôn ngữ lập trình mới. Để đạt được hiệu suất tốt nhất, việc nghiên cứu tiếp về CSGVD cần tiếp tục nghiên cứu và phát triển các phương pháp cải thiện và tối ưu hóa, đồng thời thử nghiệm và đánh giá trên các bộ dữ liệu mã nguồn đa dạng và thực tế.

Sinh viên báo cáo các nội dung mà nhóm đã thực hiện, có thể là 1 phần hoặc toàn bộ nội dung của bài báo. Nếu nội dung thực hiện có khác biệt với bài báo (như cấu hình, tập dữ liệu, kết quả,...), sinh viên cần chỉ rõ thêm khác biệt đó và nguyên nhân.

Sinh viên đọc kỹ yêu cầu trình bày bên dưới trang này

YÊU CẦU CHUNG

- Sinh viên tìm hiểu và thực hiện bài tập theo yêu cầu, hướng dẫn.
- Nộp báo cáo kết quả chi tiết những việc (**Report**) bạn đã thực hiện, quan sát thấy và kèm ảnh chụp màn hình kết quả (nếu có); giải thích cho quan sát (nếu có).
- Sinh viên báo cáo kết quả thực hiện và nộp bài.

Báo cáo:

- File .PDF. Tập trung vào nội dung, không mô tả lý thuyết.
- Đặt tên theo định dạng: [Mã lớp]-Project_Final_NhomX_Madetai. (trong đó X và Madetai là mã số thứ tự nhóm và Mã đề tài trong danh sách đăng ký nhóm đồ án).

  Ví dụ: [NT521.N11.ANTT]-Project_Final_Nhom03_CK01.
- Nếu báo cáo có nhiều file, nén tất cả file vào file .ZIP với cùng tên file báo cáo.
- Nộp file báo cáo trên theo thời gian đã thống nhất tại courses.uit.edu.vn.

Đánh giá:

- Hoàn thành tốt yêu cầu được giao.
- Có nội dung mở rộng, ứng dụng.

Bài sao chép, trễ, … sẽ được xử lý tùy mức độ vi phạm.

HẾT