# Supplementary material

Due to company confidentiality requirements, the data cannot be copied or shared outside in any way. So we can only provide here a complete description of our experimental process and part of the code, which we hope will help you to review our work.

## A.Data collection

### scripts

- cfg_build.py : to obtain the CFG of all functions and the assembly instructions of each BB through angr
- cg_build.py : to obtain the CG of each program
- dfg_build.py : to obtain the DFG of each program

### run steps

- compile each target program according to the specified option (can be gcc-O2, gcc-O3, llvm-O2, and llvm-O3)
- run the following instruction to obtain the CFG of all functions and the instructions of each BB
  - ✧ python cfg_build.py --target_program="poj"/"spec" --output=output_path --comp_t=compiler --opti_t=compile_option
  - ✧ parameter
    - ➢ --target_program: program set categories, choose poj or spec
    - ➢ --output: storage path for output file
    - ➢ --comp_t: types of compilers
    - ➢ --opti_t: compile options (can be gcc-O2, gcc-O3, llvm-O2, and llvm-O3）
  - ✧ output
    - ➢ program_bb : records all the assembly instructions of the program
    - ➢ programI_adj : records the adjacency matrix of the CFG of each function of the program.
    - ➢ program_arg : records the node (BB) description of each CFG. Each CFG contains two lines. The first line has two parameters, which are the address of the first instruction of the function, and the total number of BBs contained in the CFG of the function. The second line is the number of instructions per BB of the CFG.

- run the following instruction to obtain the CG of the program

  - ✧ python cg_build.py --target_program="poj"/"spec" --output=output_path --comp_t=compiler --opti_t=compile_option

- output
  - program_cg_adj : records the adjacency matrix of the CG of the program
  - program_cg_arg : records the description of all nodes of the CG. The first line of this file is the total number of nodes, followed by one parameter per line, indicating the address of the first instruction of the function corresponding to each node
- run the following instruction to obtain the DFG of the program
  - python dfg_build.py --target_program="poj"/"spec" --output=output_path --comp_t=compiler --opti_t=compile_option
  - output
    - program_dfg_adj : records the adjacency matrix of the DFG of the program
    - program_dfg_arg : records the description of all nodes of the DFG. The first line of this file is the total number of nodes, followed by a parameter per line, indicating the address of the instruction corresponding to each node

- Detect all the dead stores in the program by DrCCTProf and output the functions they belong to. Use these outputs to label each function of the target program (label 0 or 1)

# B. BERT training

## scripts

- get_vocab.py : to get the vocabulary of the dataset
- create_bert_sample.py : to obtain the training data of BERT
- create_bert_train_sample.py : to obtain the training data of BERT
- BERT-BMM : BERT model to get initial embeddings of DFG and CFG nodes
- data_preparation_bert.py : to integrate all the initial data and generate the dataset

## run steps

- Execute the following command to get the vocabulary file of the dataset
  - python get_vocab.py
  - output
    - vocab_cnt.pkl
- Execute the following two instructions successively to obtain the training data of BERT-BMM
  - python create_bert_sample.py --sample_save_path = output_path_of_create_bert_sample
  - python create_bert_train_sample.py --sample_save_path = output_path_of_bert_sample --sample_input_path =

output_path_of_create_bert_sample
    - ✧ output
        - ➢ anp_corpus, big_corpus, gc_corpus : the dataset for three BERT-BMM training tasks
- Execute the following command to get the vocabulary file required by BERT-BMM
    - ✧ bert-vocab -c vocab_cnt.pkl -o bert_vocab
    - ✧ output
        - ➢ bert_vocab: vocabulary file required by BERT-BMM
- Execute the following command to train the BERT-BMM model
    - ✧ bert -c training_corpus_file_name (anp_corpus/big_corpus/gc_corpus) -v bert_vocab -o path_to_store_the_trained_model
- Execute the following instructions to obtain the initial embedding of CFG and DFG nodes via BERT-BMM
    - ✧ bert -v bert_vocab --load_model_path= path_to_store_the_trained_model --dataset_type=dataset_type (poj or spec_old)
    - ✧ output
        - ➢ program_bb_bert : The initial embedding of all CFG nodes in the program
        - ➢ program_dfg_bert : The initial embedding of all nodes of the program's DFG
- Construct the dataset by integrating all the initial data with the following instructions
    - ✧ python data_preparation_bert.py --pre_data_path=path_of_dataset
    - ✧ output
        - ➢ Each sample will output a file, the file contains the compilation category of the sample, the program number to which it belongs, CFG, DFG, CG and the initial embedding of the nodes of CFG and DFG. For the dead store prediction task, a label indicating whether the sample contains a dead store is also included.

# C.Training process

## scripts

- train.py : script for training model
- cfg.py : configuration file
- mystatistics.py : script to count and output results
- loader.py : script for data loading
- pair_sample.py : script for generating sample pairs
- model.py : BMM model

## run steps

- Start training and testing the model by executing the following instructions.
    - ✧ python train.py --task=task_name --pre_data_path= path_of_dataset

- ✧ Selection of parameter "task"
  - ➢ Binary similarity detection task: set task=binaryClassify
  - ➢ Compile the classification task, set task=funcClassify
  - ➢ Deadstore task, set task=deadstore
- ✧ output:
  - ➢ Binary similarity detection task: the similarity distance of each pair of samples in the test set, and their respective ground truth labels
  - ➢ Compilation classification task: accuracy of test set classification
  - ➢ Deadstore prediction task: precision, recall and accuracy of deadstore predictions

# D.Datasets and tool links

## Datasets:

- For the binary similarity detection task, we use the POJ-104 dataset as our test data
  POJ-104 : https://sites.google.com/site/treebasedcnn
- For the dead store prediction task, we choose the SPEC CPU 2017 benchmark as our dataset
  SPEC CPU 2017 : http://www.spec.org/cpu2017

## Tools:

- We use angr to generate three types of graphs
  Angr : https://github.com/angr/angr
- We use DrCCTProf to find all the dead stores that exist in target programs
  DrCCTProf : https://github.com/Xuhpclab/DrCCTProf