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: BinDeep: A deep learning approach to binary code similarity detection

Mã nhóm: G02 Mã đề tài: CK10

Lớp: NT521.N11.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	Lưu Gia Huy	21520916	21520916@gm.uit.edu.vn
2	Nguyễn Vũ Anh Duy	21520211	21520211@gm.uit.edu.vn
3	Nguyễn Văn Khang Kim	21520314	21520314@gm.uit.edu.vn

2. TÓM TẮT NỘI DUNG THỰC HIỆN:[1]

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

☑ 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:

BinDeep: A deep learning approach to binary code similarity detection.

[1] Ghi nội dung tương ứng theo mô tả

C. Dịch tên Tiếng Việt cho bài báo:

Dùng Deep Learing trong phát hiện sự tương đồng của mã nhị phân

D. Tóm tắt nội dung chính:

Paper đề xuất phương pháp sử dụng deep learning cho việc phát hiện sự tương đồng của mã nhị phân:

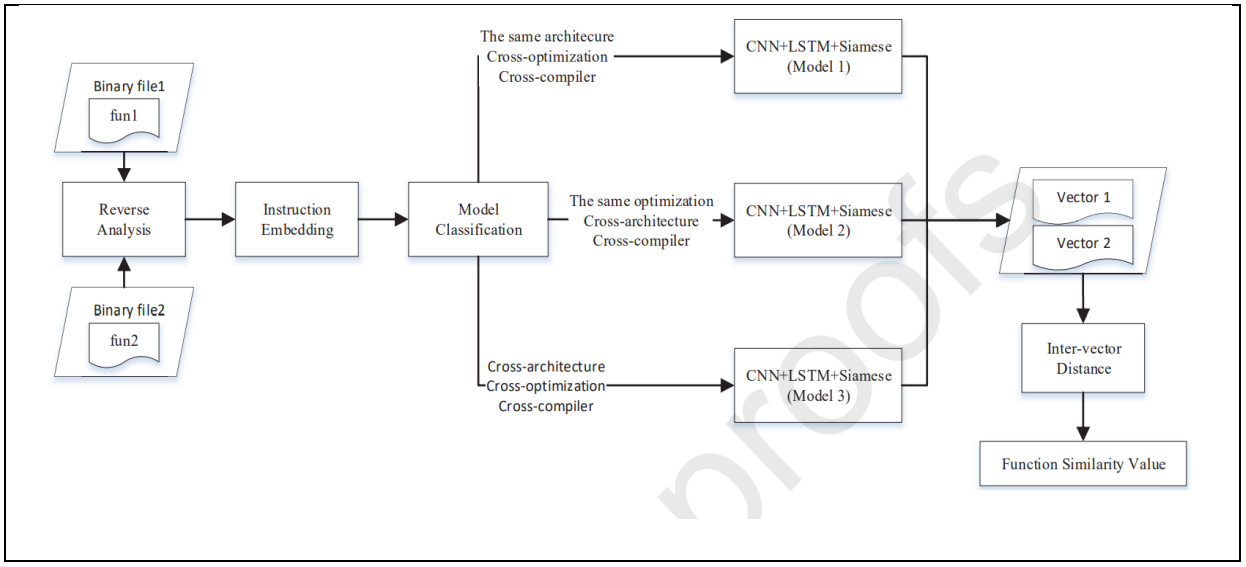
- Phương pháp này loại bỏ được các hạn chế của các phương pháp đã có trước đó như là: không dùng các cơ chế tiêu tốn tài nguyên cho hiệu suất thấp như là so khớp, mô phỏng biểu đồ. Một số phương pháp đã có trước đây không thể cho kết quả tốt khi mà binary code có các version khác nhau, được compile bởi các kiến trúc CPU, option khác nhau hoặc chỉ dùng một model
- Phương pháp này hữu ích trong việc phân tích malware, phát hiện ra các lỗ hỏng bảo mật. Bằng cách tính toán sự tương đồng với các hàm binary đã biết ta có thể xác định được lỗ hỏng bảo mật hay các function nguy hiểm mà không cần biết source code của phần mềm.
- Việc không cần biết rõ source code của các phần mềm là một ưu điểm cực lớn của mô hình này. Bởi vì xét về vấn đề bảo mật và sở hữu trí tuệ, nên là hầu hết các nhà phát triển phần mềm sẽ không cung cấp mã nguồn sản phẩm của họ để phân tích bảo mật.
- Mô hình paper đề xuất thì đầu vào là hai hàm trong hai binary file khác nhau, đầu ra sẽ là label đánh giá sự tương đồng của hai hàm đó, label là 1 nếu hai hàm đấy tương đồng và ngược lại
- Cùng một source code sẽ được compile bằng compiler khác nhau (gcc, clang), khác optimization levels (O0, O1, O2, O3), khác kiến trúc CPU (x86, x86-64, ARM)
- Mô hình này hoạt động không cần biết source code. Do đó ở đây dùng IDA pro để disassemble binary file ở cấp độ function. Sau đó normalize assembly code và dùng instruction embedding model để vector hóa các instructions

- Tiến hành nhiều thử nghiệm để so sánh đánh giá mức độ hiệu quả và đưa ra mô hình tốt nhất dựa trên các thông số accuracy, precision, recall, f1 score, FPR , trong đó chủ yếu thực hiện so sánh các mức dimension 100, 200, 300; so sánh giữa 3 model là CNN, LSTM, CNN+LSTM; so sánh giữa các GNU Core Utilities version

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

Bài báo sử dụng 3 kỹ thuật cho 3 giai đoạn chính trong phương pháp đề xuất, bao gồm:

- Kỹ thuật 01: Sử dụng IDA pro. Ở đây do là phương pháp paper đề xuất không cần biết source code do đó ta cần dùng trình IDA pro để disassemble binary file để trích xuất assembly code của function cần cho việc tính toán độ tương đồng. Sau đó Normalize assembly code của function chuẩn bị cho việc vector hóa. Ở các phương pháp trước đó người ta dùng các kĩ thuật để trích xuất features, nhưng ở đây paper sử dụng instruction sequence như là features. Ta có opcode thì có số lượng giới hạn, nhưng operands thì ứng với mỗi cách compile khác nhau sẽ cho ra kết quả khác và để tránh việc làm mất mát nhiều thông tin, hay làm mất sự tương quan giữa luồng thực thi function thì ở đây normalize là vô cùng cần thiết
- Kỹ thuật 02: Dùng instruction embedding model (word2vec) cụ thể là model skip-gram để convert các instruction đã được normalize sang vector
- Kỹ thuật 03: Dùng 3 model cho tập dữ liệu được phân thành loại khác nhau. Được trình bày cụ thể trong mô hình dưới đây:



F. Môi trường thực nghiệm của bài báo:

- **Cấu hình máy tính:** Dell T360 Server, Intel Xeon E5-2603 V4 CPUs, 16GB memory, 2TB hard drives, 1 NVIDIA Tesla P100 12GB GPU card
- **Các công cụ hỗ trợ sẵn có:** IDA Pro 7.0 dùng trong giai đoạn trích xuất các instruction assembly code từ
- **Ngôn ngữ lập trình để hiện thực phương pháp:** Python
- **Đối tượng nghiên:** Dùng các Linux packages phổ biến là coreutils, findutils, diffutils, sg3utils, and util-linux,...Sau khi có được source code thì sẽ tiến hành compile bằng 2 compiler là gcc, clang; 3 kiến trúc CPU là x86, x86-64, ARM; 4 optimization level là O0, O1, O2, O3. Thu được 4729140 mẫu. Chia data thành 5 loại như sau:

Table 2: Description of sample types

Sample type	Number	Remarks
Cross-compiler function pairs	855136	Only the compilers are different
Cross-optimization function pairs	855136	Only the optimization levels are different
Cross-version function pairs	501028	Only the function versions are different
Cross-architecture function pairs	1282704	Only the CPU architectures are different
Mixed function pairs	1235136	The Compilers, optimization levels,versions, and architectures may be all different

- **Tiêu chí đánh giá tính hiệu quả của phương pháp**: Có 2 label đầu ra 1 và 0, 1 nếu 2 function tương đồng và ngược lại. Đánh giá hiệu quả của phương pháp dựa trên accuracy, precision, recall, f1 score, FPR. Thực nghiệm trên các dimension là 100, 200, 300, và trên các GNU Core Utilities version. Dùng các model CNN, LSTM, CNN+LSTM để tìm ra mô hình cho kết quả tốt nhất.

G. Kết quả thực nghiệm của bài báo:

- Ở table 4,5 kết quả thực nghiệm cho thấy Embedding dimension 300 cùng với model CNN+LSTM cho kết quả tốt nhất

Table 4: Results under different embedding dimension in the LSTM model

Embedding dimension	Accuracy	Precision	Recall	$F_1 Score$	FPR
100	0.9062	0.8964	0.8526	0.8739	0.0609
200	0.9107	0.9021	0.8599	0.8804	0.0578
300	0.9248	0.9208	0.8776	0.8986	0.0466

Table 5: Results under different embedding dimension in the CLSTM model

Embedding dimension	Accuracy	Precision	Recall	$F_1 Score$	FPR
100	0.9820	0.9692	0.9844	0.9767	0.0195
200	0.9812	0.9690	0.9825	0.9757	0.0196
300	0.9843	0.9707	0.9888	0.9797	0.0185

- Ở table 6,7 cho thấy việc dùng 3 model CNN+LSTN cho 3 tập dữ liệu được phân loại sẽ cho kết quả tốt hơn so với việc dùng 1 model CNN+LSTM cho toàn bộ dữ liệu

Table 6: Results of the LSTM and classification model + LSTM

Model	Accuracy	Precision	Recall	F_1Score	FPR
LSTM	0.8830	0.8788	0.8493	0.8637	0.0908
Classification model + LSTM	0.9263	0.9128	0.8746	0.8932	0.0515

Table 7: Results of the CLSTM and classification model + CLSTM

Model	Accuracy	Precision	Recall	F_1Score	FPR
CLSTM	0.9206	0.9023	0.8829	0.8924	0.0591
Classification model + CLSTM	0.9844	0.9702	0.9895	0.9797	0.0187

- Ở table 8, 9, 10, 11, 12 cho thấy model CNN+LSTM luôn cho kết quả tốt nhất, và GNU Core Utilities version cho kết quả tốt nhất là version mới nhất ở hiện tại khi tác giả thực nghiệm là 8.30

Table 8: Comparison results of LSTM, CNN and CLSTM models on the mixed dataset

Model	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	0.9228	0.9110	0.8808	0.8956	0.0534
CNN	0.9029	0.8824	0.8699	0.8761	0.0900
CLSTM	0.9868	0.9700	0.9911	0.9804	0.0189

Table 9: Comparison results of LSTM, CNN and CLSTM models on the cross-architecture dataset

Model	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	0.9204	0.9025	0.8871	0.8947	0.0619
CNN	0.9093	0.8939	0.8649	0.8791	0.0897
CLSTM	0.9800	0.9690	0.9859	0.9804	0.0245

Table 10: Comparison results of LSTM, CNN and CLSTM models on the cross-optimization dataset

Model	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	0.9304	0.9179	0.8812	0.8991	0.0459
CNN	0.8998	0.9149	0.8679	0.8907	0.0718
CLSTM	0.9913	0.9860	0.9955	0.9769	0.0123

Table 11: Comparison results of LSTM, CNN and CLSTM models on the cross-compiler dataset

Model	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	0.9069	0.8959	0.8798	0.8878	0.0733
CNN	0.9023	0.9031	0.8699	0.8861	0.0721
CLSTM	0.9727	0.9312	0.9956	0.9623	0.0296

Table 12: Comparison results of LSTM, CNN and CLSTM on the cross-version dataset

Model	Version	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	5.0	0.6424	0.6576	0.6411	0.6599	0.2904
	6.0	0.7241	0.7267	0.7190	0.7254	0.2888
	7.0	0.7409	0.7399	0.7536	0.7404	0.2565
	8.0	0.8133	0.8402	0.8209	0.8265	0.1409
	8.30	0.9888	0.9735	0.9743	0.9738	0.0377
CNN	5.0	0.6289	0.6304	0.6253	0.6296	0.3104
	6.0	0.6920	0.7156	0.6889	0.7036	0.2818
	7.0	0.7165	0.7085	0.7212	0.7125	0.2765
	8.0	0.7787	0.7792	0.7867	0.7789	0.1963
	8.30	0.9798	0.9774	0.9659	0.9716	0.0584
CLSTM	5.0	0.7135	0.7103	0.7036	0.7069	0.1221
	6.0	0.7868	0.7834	0.7765	0.7799	0.1325
	7.0	0.8154	0.8053	0.8175	0.8113	0.0899
	8.0	0.8732	0.8942	0.8766	0.8854	0.0321
	8.30	0.9960	0.9925	0.9995	0.9908	0.0075

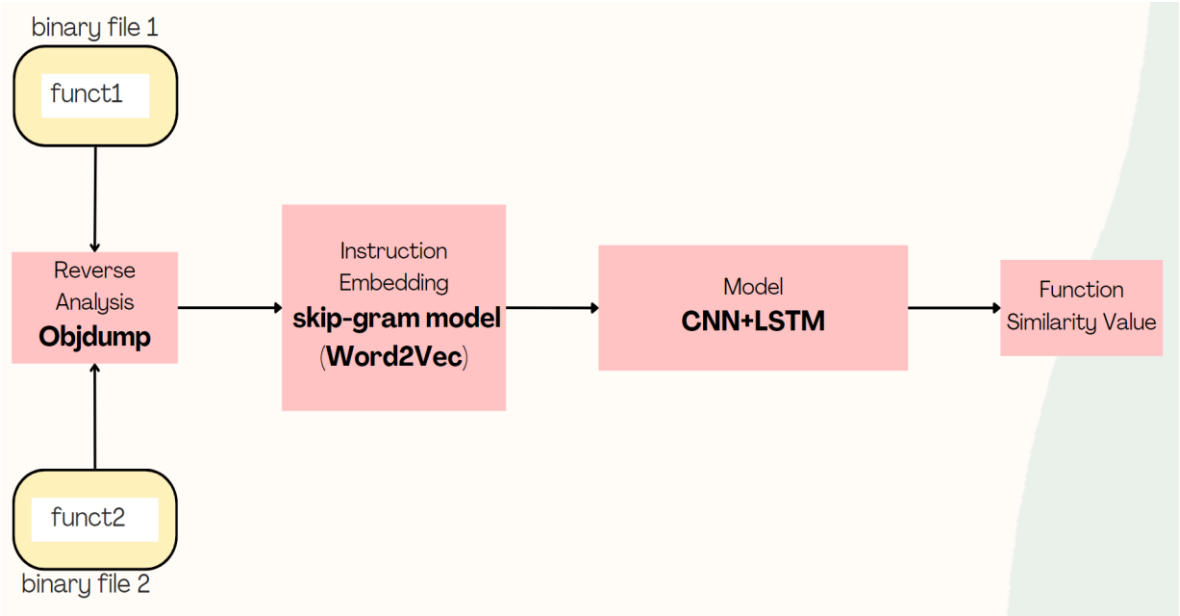
- Ưu điểm của bài báo:
  - Phát hiện sự tương đồng của mã nhị phân trên các source code lớn, phức tạp như các packages của Linux cho độ chính xác cao
  - Thực nghiệm so sánh trên nhiều model, trên nhiều trường hợp về các versions, các dimension để đưa ra kết quả tốt nhất
  - Việc không cần biết rõ source code của các phần mềm là một ưu điểm cực lớn của mô hình này. Bởi vì xét về vấn đề bảo mật và sở hữu trí tuệ, nên là hầu hết các nhà phát triển phần mềm sẽ không cung cấp mã nguồn sản phẩm của họ để phân tích bảo mật.
- Nhược điểm của bài báo:
  - Dữ liệu không thực tế, chỉ giải quyết được vấn đề phát hiện sự tương đồng của mã nhị phân chứ, chưa giải quyết được câu chuyện lớn hơn trong security như là phát hiện mã độc trong code, function nguy hiểm,...

- Nếu như file binary được compile làm rối code, làm rối mã nhị phân, tức là gây khó khăn cho việc reverse thì phương pháp paper để xuất sẽ không thể thực hiện được trên các file binary như vậy
- Không public thuật toán, code, dataset, cách thức xử lý data cụ thể

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

Link github toàn bộ code của bọn em: https://github.com/Hjn4Pwn/BinDeep-Deep-Learing-Approach-to-Binary-Code-Similarity-Detection.git

Mô hình thực tế triển khai:

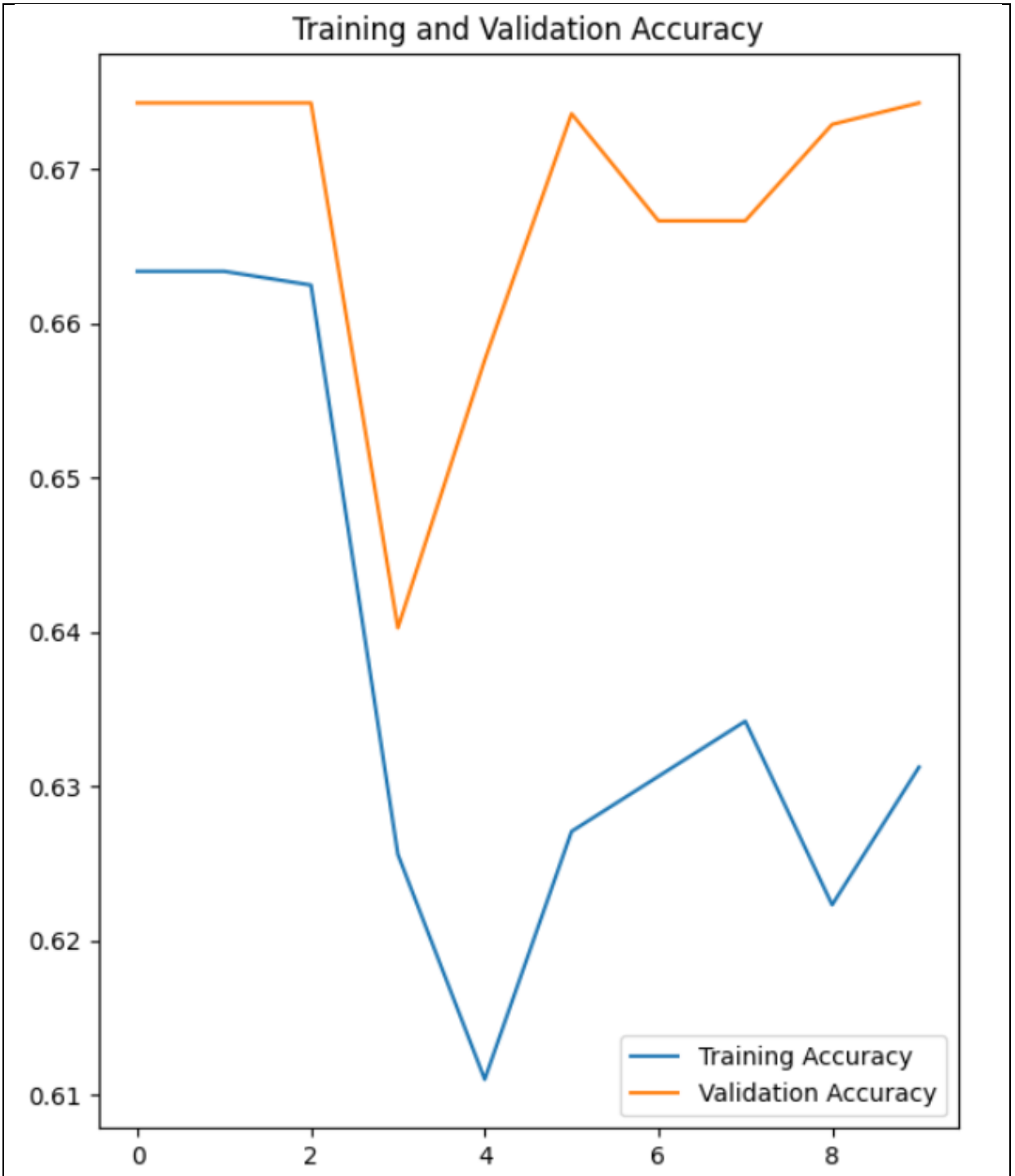


Paper triển khai	Thực tế triển khai

Dùng các files .c trong các pakage của Linux như: coreutils, findutils, diffutils, ... Thu thập được 4.729.140 mẫu	Dùng các files .c trong các challenges pwn. Thu thập được 4800 mẫu
Reverse bằng IDA Pro	Reverse bằng Objdump
Compile với nhiều options: compilation, CPU architectures, optimization	Compile với 2 options: compilation, optimization
Dùng 3 models, với mỗi model là gom nhóm input khác nhau, ví dụ như model1 train input giống nhau kiến trúc, khác về compiler, khác optimization	Dùng 1 model (CNN+LSTM) và input là toàn bộ data không chia nhóm

Cụ thể việc mà nhóm thực hiện như sau:

- Tự build dataset, tìm các source code .c từ các pwnable challenges thu thập được dataset gồm 4800 mẫu
- Thay vì disassemble binary file bằng IDA Pro 7.0 thì ở đây chúng em dùng Objdump. Sau đó tiến hành trích xuất assembly code của function cụ thể.
- Tiếp đến tiến hành normalize assembly code của function
- Sau khi normalize chúng em dùng model word2vec cụ thể là model skip-gram để vector hóa các instructions đã được normalize
- Sau đó không chia dataset thành 3 loại như là mô hình paper đề xuất để đưa vào 3 model, ở đây bọn em lấy toàn bộ dataset rồi đưa vào 1 model CNN+LSTM duy nhất để train và cho ra label là 1 nếu 2 functions là tương đồng và ngược lại
- Kết quả thực hiện:



Accuracy không được cao do vài lý do sau:

- Dataset nhỏ chỉ 4800 mẫu, các function cũng đơn giản, do đây được lấy từ các pwnable challenges
- Chỉ dùng 1 model cho toàn bộ dataset thay vì là 3 model như bài báo, vì lí do dữ liệu nhỏ chia ra sẽ không tối ưu

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

Các khó khăn của nhóm:

- Bài báo không hề cung cấp thuật toán cho việc xử lý dữ liệu, trích xuất dữ liệu chi tiết như thế nào, không có code sẵn, paper chỉ đưa ra rằng dùng IDA Pro, normalize các operands thành label gì, chứ chi tiết các thức không được chỉ rõ. Do đó cũng gây khó khăn. Tức là bọn em sẽ phải code và làm mọi thứ.
- Lần đầu 2/3 các thành viên trong nhóm tiếp cận ML, DL. Việc hiểu và thực hiện có phần khó khăn
- Một kỳ nhưng có quá nhiều đồ án cụ thể là 4, không có quá nhiều thời gian để tập trung trọn vẹn vào đồ án
- Chúng em cũng đã là sinh viên năm 3, bản thân phải tự có định hướng riêng cho chuyên ngành sẽ theo làm sau khi ra trường, do đó bản thân mỗi bạn đều dành thời gian tự học thêm nhiều thứ riêng như có bạn đã đi làm dev, có bạn theo hướng devops, cloudops,... tuy đây chỉ là lý do có phần personal, cũng có phần chủ quan cơ mà mong thầy có thể thông cảm.

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	Tìm các source file .c ở các pwnable challenges	Nguyễn Vũ Anh Duy, Nguyễn Văn Khang Kim
2	Kết hợp python, objdump để xử lý dữ liệu, disassemble và normalize assembly code của function trên máy local	Lưu Gia Huy, Nguyễn Văn Khang Kim

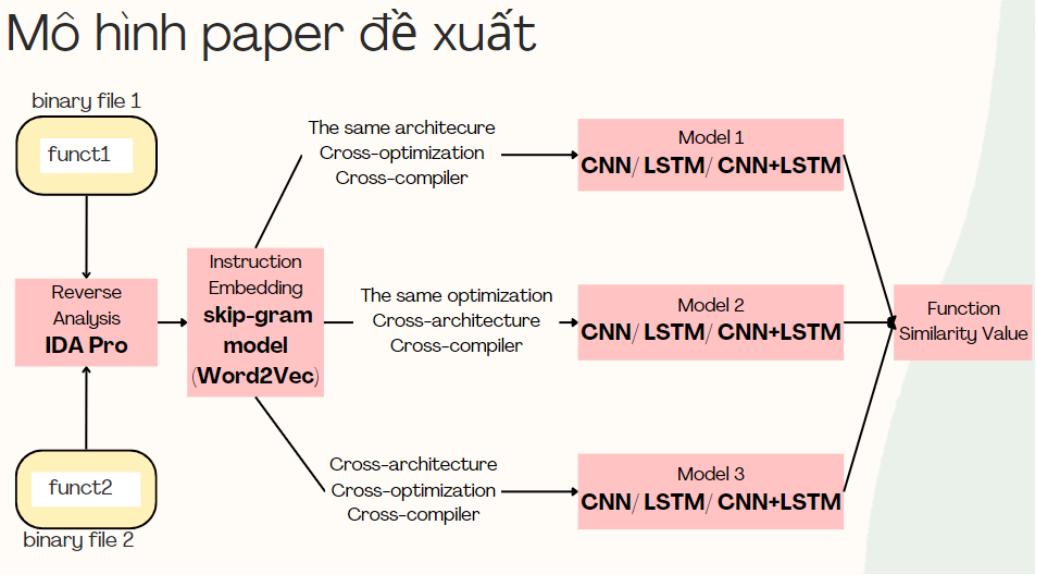
3	Tìm hiểu về Machine Learning, Deep Learning, tìm hiểu về tiền xử lý dữ liệu, word2vec, CNN, LSTM, CNN+LSTM	Lưu Gia Huy, Nguyễn Vũ Anh Duy, Nguyễn Văn Khang Kim
4	Build dataset từ các file asm code đã được normalize, dùng word2vec để vector hóa các asm code đã được normalize và nhúng vào model CNN+LSTM thực hiện train model trên colab	Lưu Gia Huy, Nguyễn Vũ Anh Duy
5	Đọc paper, làm slide, viết report, thuyết trình,	Lưu Gia Huy, Nguyễn Vũ Anh Duy, Nguyễn Văn Khang Kim

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.

A. Phương pháp thực hiện

1. Trình bày kiến trúc, thành phần của hệ thống trong bài báo

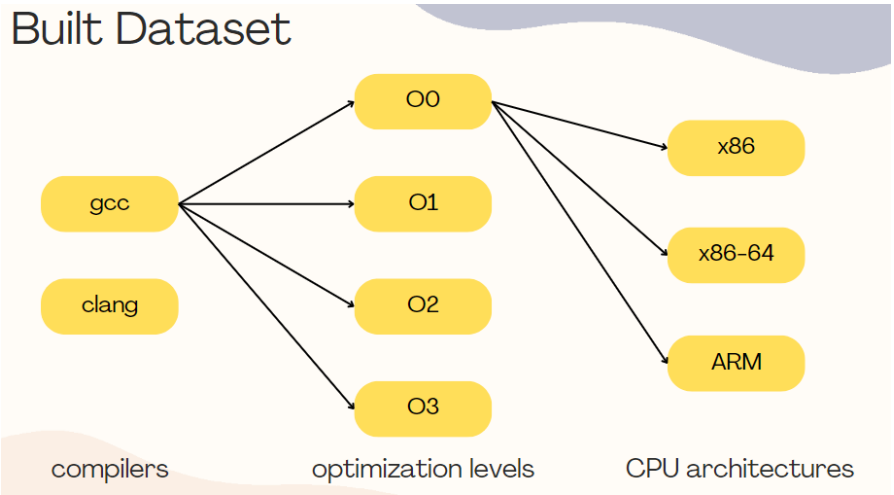


Mô hình hệ thống paper triển khai gồm các phần sau:

Thu thập dữ liệu: Ở đây paper thu thập các source code .c từ các Linux packages phổ biến như coreutils, findutils, diffutils, sg3utils, and util-linux,... thu được 4729140 mẫu.

Xử lý dữ liệu: Ở đây sẽ chia làm 2 giai đoạn:

- Compile source code: Compile bằng 2 compiler là gcc, clang; 4 optimization levels là O0, O1, O2, O3; 3 kiến trúc CPU (x86, x86-64, ARM). Theo Paper thì 1 source code .c sẽ được compile ra 24 versions



- Disassemble và normalize: Sau khi compile xong thì ở đây paper đề xuất dùng IDA Pro 7.0 để disassemble ra assembly code, và chỉ lấy ở cấp độ function. Rồi tiến hành normalize như sau:

Original operand	**Normalized operand**
General Register	TypeOne
Direct Memory Reference	TypeTwo
Memory Ref [Base Reg + Index Reg]	TypeThree
Memory Reg [Base Reg + Index Reg + Displacement]	TypeFour
Immediate Value	TypeFive
Immediate Far Address	TypeSix
Immediate Near Address	TypeSeven
Other Type	TypeEight

Ví dụ như sau:

Original instruction	**Normalized instruction**
push ebp	push TypeOne

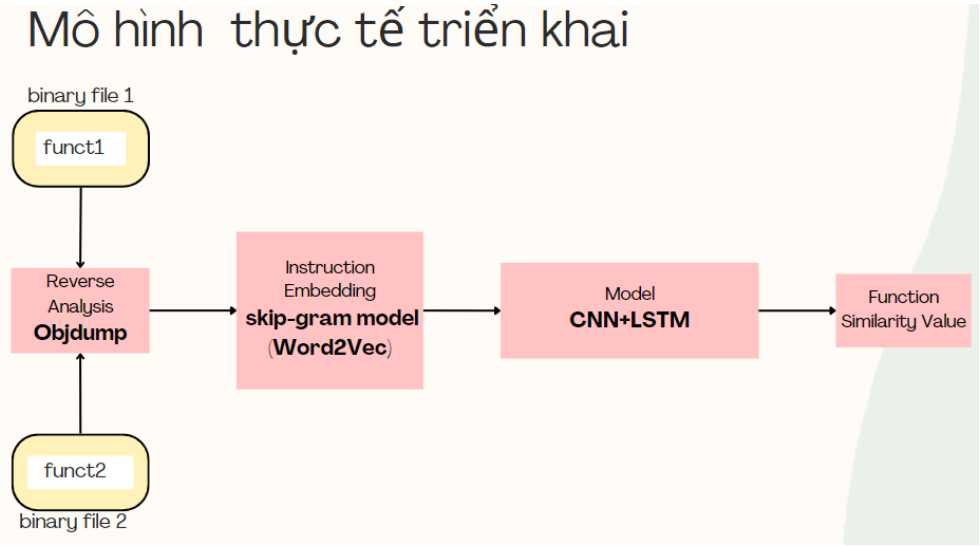
jl loc 804B271	jl TypeSeven
sub esp, 18h	sub TypeOne, TypeFive
mov dword ptr [esp+74h], 0	mov TypeThree, TypeFive
lea eax, [ebp+esi*2+0]	lea TypeOne, TypeFour
call exit	call TypeSix

- Word2vec: Dùng instruction embedding model (word2vec) cụ thể là model skip-gram để convert các instruction đã được normalize sang vector để nhúng vào model DeepLearning
- Chia tập dữ liệu ra 3 nhóm, mỗi nhóm sẽ được train với 1 model:
  - The same architecure, Cross-optimization, Cross-compiler
  - The same optimization, Cross-architecture, Cross-compiler
  - Cross-architecture, Cross-optimization, Cross-compiler

Ở đây paper sẽ thực nghiệm với 3 model CNN, LSTM, CNN+LSTM, thực nghiệm so sánh giữa các dimension 100, 200, 300, giữa các GNU Core Utilities version để đưa ra được kết luận là CNN+LSTM với dimension là 300 và GNU Core Utilities version 8.30 cho kết quả tốt nhất

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

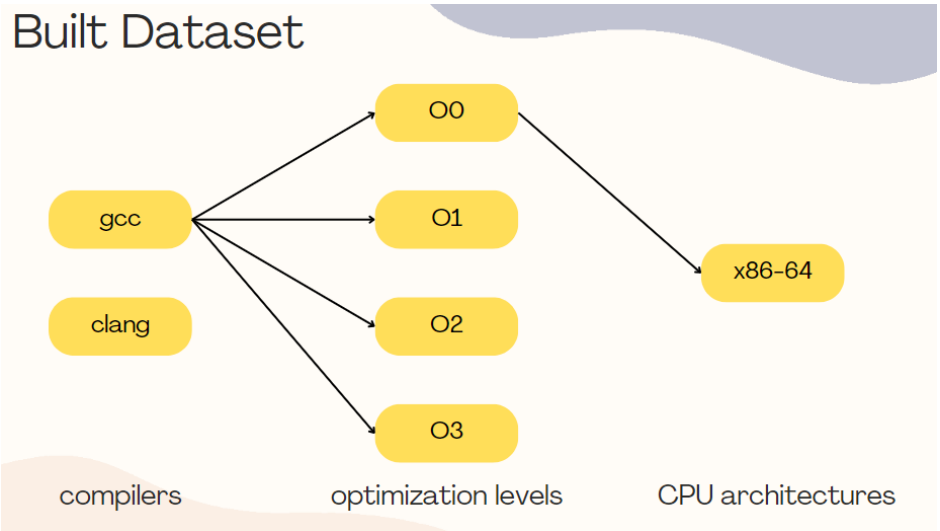
Mô hình thực tế nhóm đã triển khai:



Sự khác biệt của paper so với thực tế triển khai:

Paper đề xuất	**Thực tế triển khai**
Dùng các files .c trong các pakages phổ biến của Linux như: coreutils, findutils, diffutils, ... Thu thập được 4.729.140 mẫu	Dùng các files .c trong các challenges pwn. Thu thập được 4800 mẫu
Reverse bằng IDA Pro	Reverse bằng Objdump
Compile với nhiều options: compilation, CPU architectures, optimization. Cụ thể thì 1 file .c sẽ được compile ra 24 versions	Compile với nhiều options: compilation, CPU architectures, optimization. Cụ thể thì 1 files .c sẽ compile ra được 8 versions
Dùng 3 models, với mỗi model là gom nhóm input khác nhau, ví dụ như model1 train input giống nhau kiến trúc, khác về compiler, khác optimization	Dùng 1 model (CNN+LSTM) và input là toàn bộ data không chia nhóm

1 file source code .c được compile thành 8 versions bằng cách compile với 2 compiler (gcc, clang), 4 optimization levels (O0, O1, O2, O3) với kiến trúc CPU x86-64:



So với paper thực nghiệm nhiều trường hợp để đưa ra kết luận là CNN+LSTM với dimension 300, GNU Core Utilities version 8.30 mới nhất ở thời điểm tác giả thực hiện thì cho được kết quả tốt nhất. Ở đây bọn em dùng luôn model CNN+LSTM, dimension 300, và GNU Core Utilities version mới nhất lúc bọn em làm là 8.32.

B. Chi tiết cài đặt, hiện thực

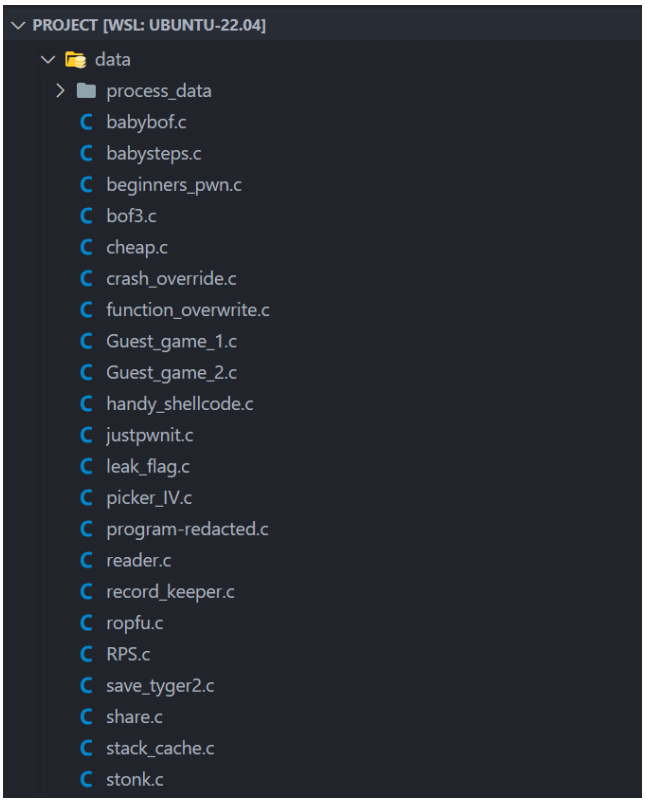
Toàn bộ code cho đồ án này là bọn em tự code và không hề có code sẵn nào từ paper, đây là link github mà em đã up tất cả code, dataset lên đây:
https://github.com/Hjn4Pwn/BinDeep-Deep-Learing-Approach-to-Binary-Code-Similarity-Detection.git

- Cấu hình máy tính:
  - CPU: 11th Gen Intel(R) Core(TM) i5-11400H @2.70GHz
  - Memory: 16 GB RAM
  - GPU: NVIDIA GeForce RTX 3050

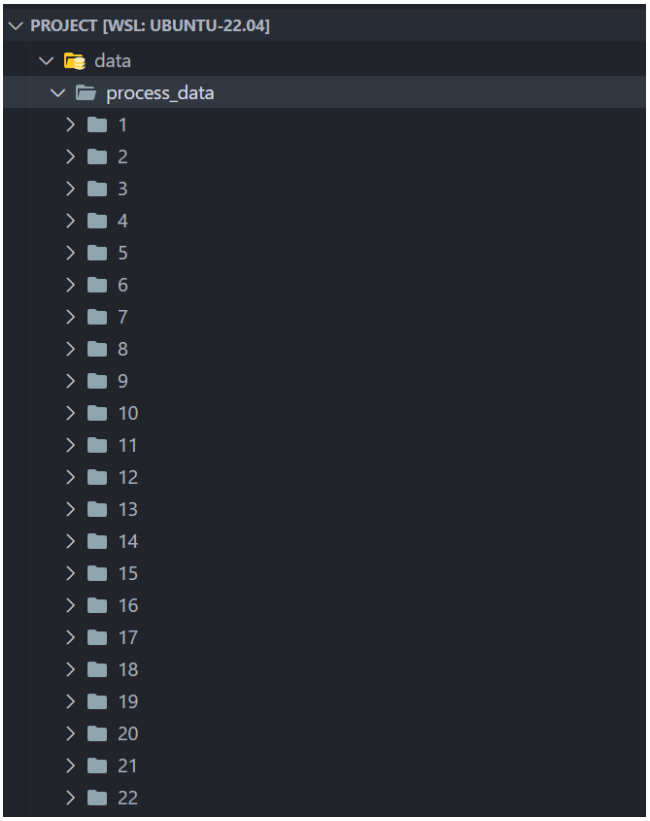
- Chuẩn bị dữ liệu: các source code .c được tìm từ các pwnable challenges, chủ yếu là ở trang web PicoCTF. Tổng cộng có 25 file code .c

Do paper không hề đưa ra các thức xử lý, thu thập dữ liệu nên dưới đây là cách nhóm em thực hiện dựa trên tư duy của nhóm

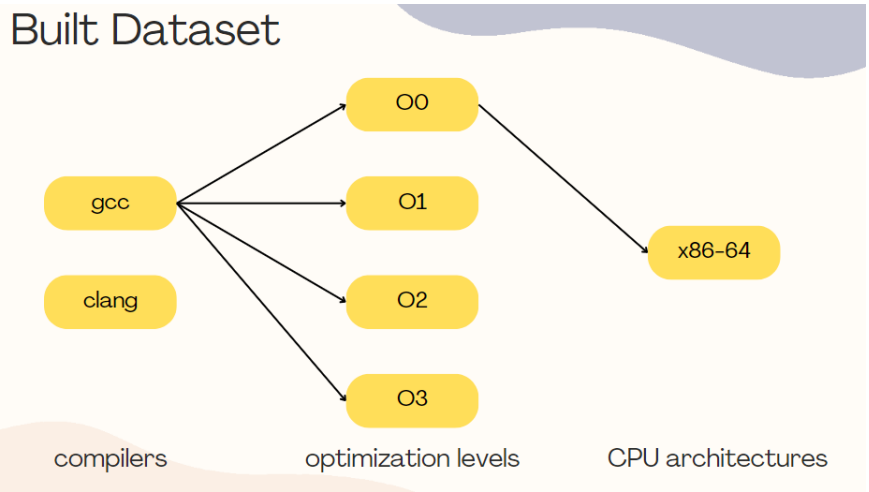
- Đầu tiên ta thu thập các file.c :



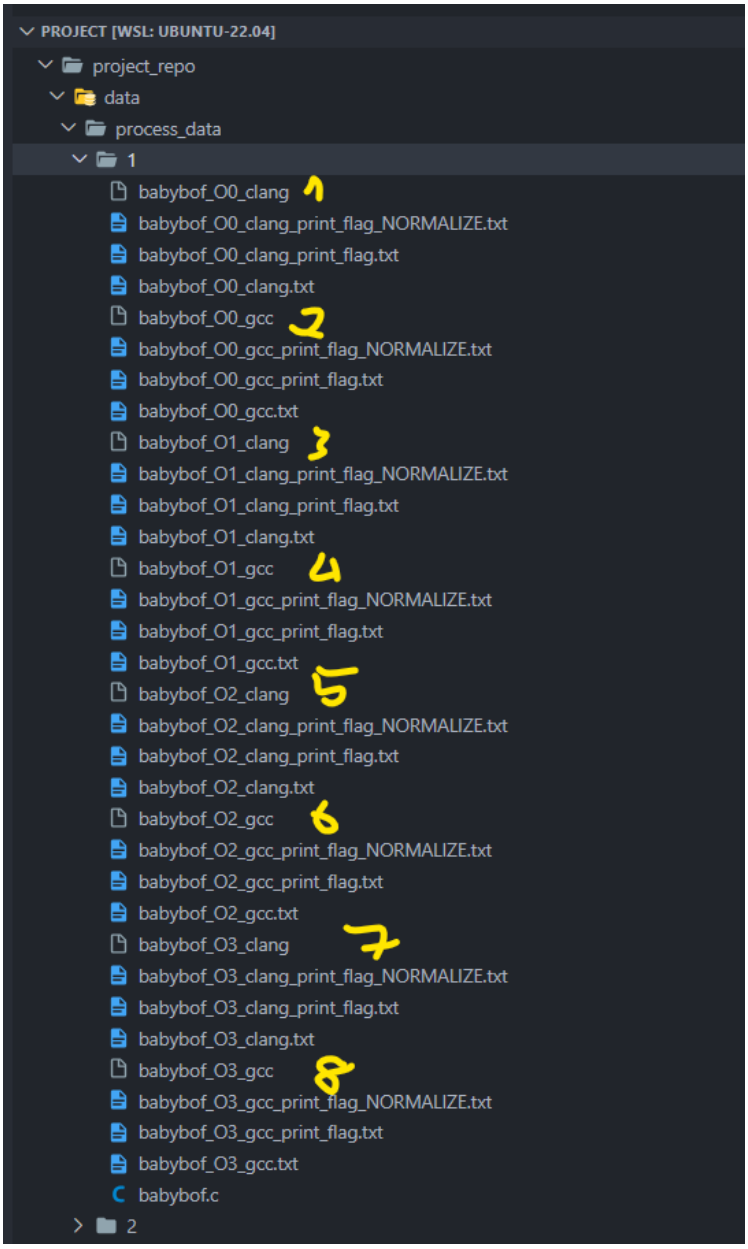
- Tiếp theo sẽ phân chia từng file .c vào từng thư mục đánh số như bên dưới, trong mỗi thư mục sẽ chứa 1 file source code .c :



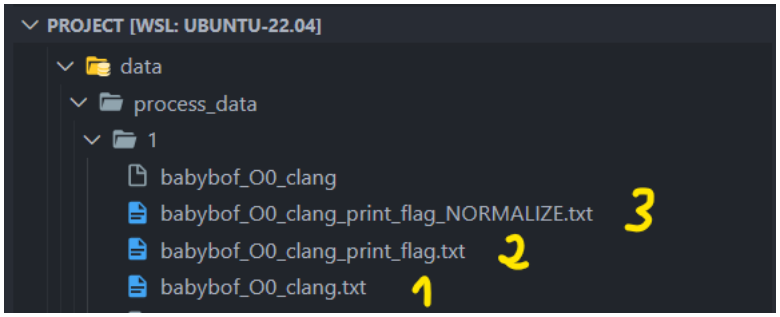
- Compile source code, normalize được thực hiện trên máy local với cấu hình đã được nêu như trên, dùng Python, cụ thể sẽ được trình bày dưới đây:
  - Với file .c trong mỗi thư mục ta sẽ tiến hành compile nó, dưới đây là cú pháp: **gcc/clang -{optimization_level} -w {file.c_path} -o {binaryfile_path}** .Compile với 2 compiler là gcc và clang cùng với 4 optimization levels là O0, O1, O2, O3, với kiến trúc CPU là x86-64. Tức là cùng 1 file .c sẽ được compile ra 8 versions.



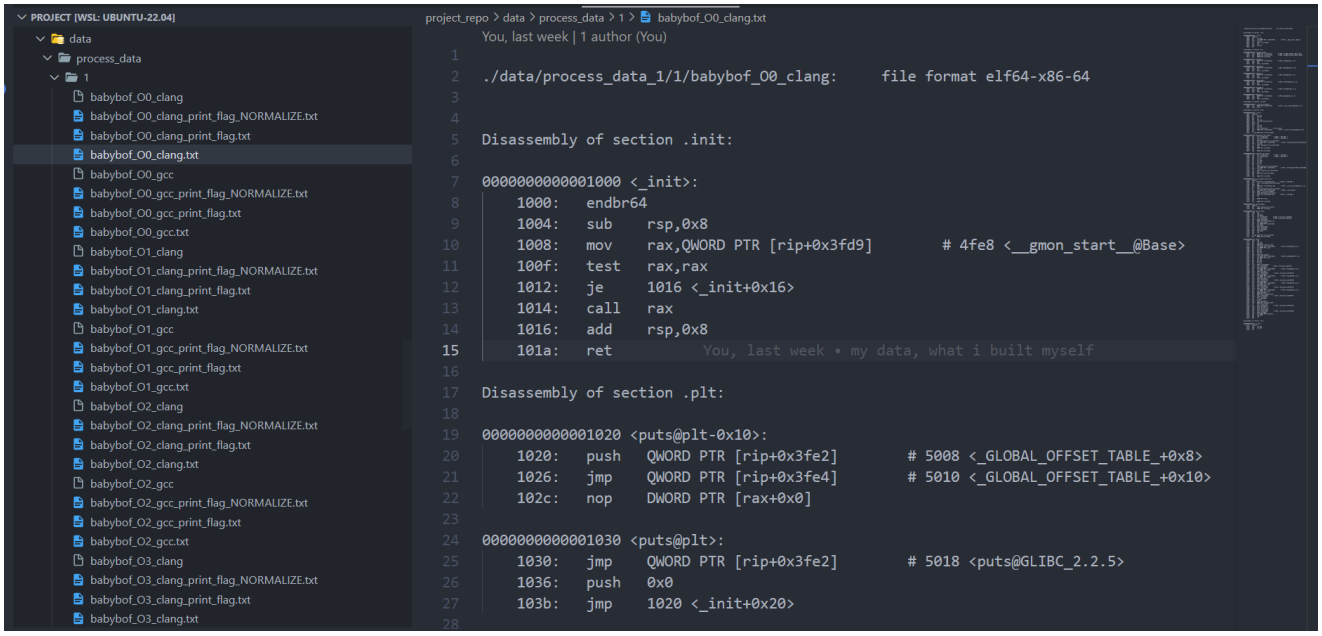
- Sau khi compile như trên thì mỗi thư mục được đánh số, ta sẽ có 8 binary files như bên dưới:



- Ứng với mỗi binary files ta sẽ tiến hành 3 bước trên nó:



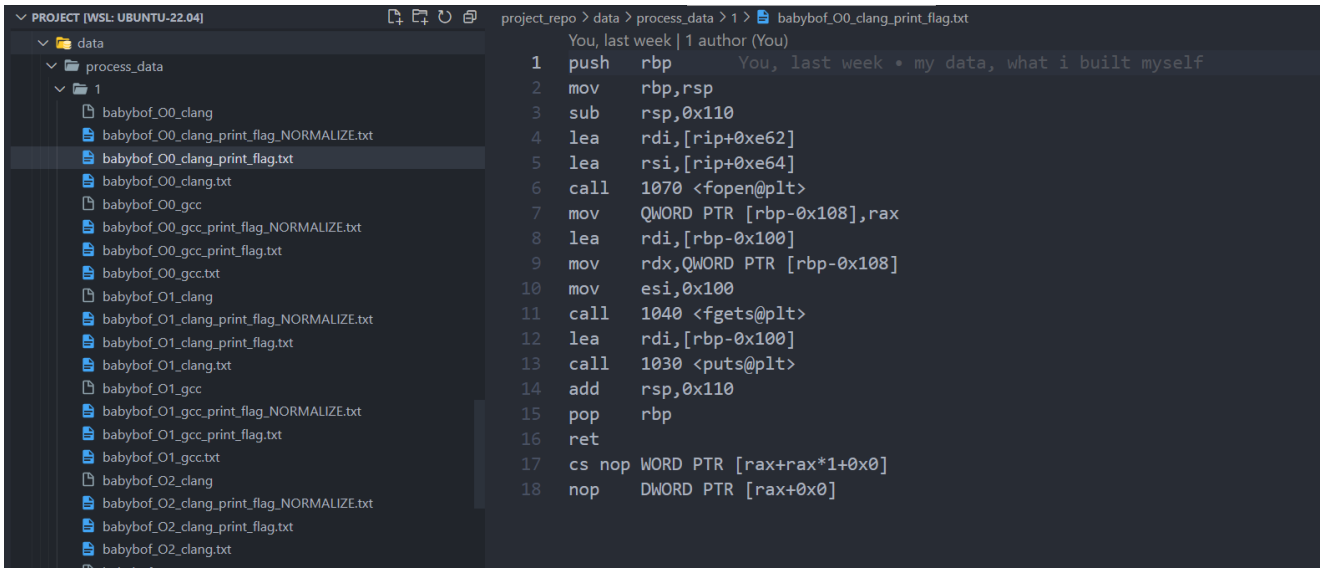
– Bước đầu tiên là disassemble file binary dùng objdump để lấy code assembly với cú pháp: **objdump --disassemble --no-show-raw-insn --disassembler-options=intel {binaryfile_path} > {asm_file}** , ta sẽ có được file như bên dưới:



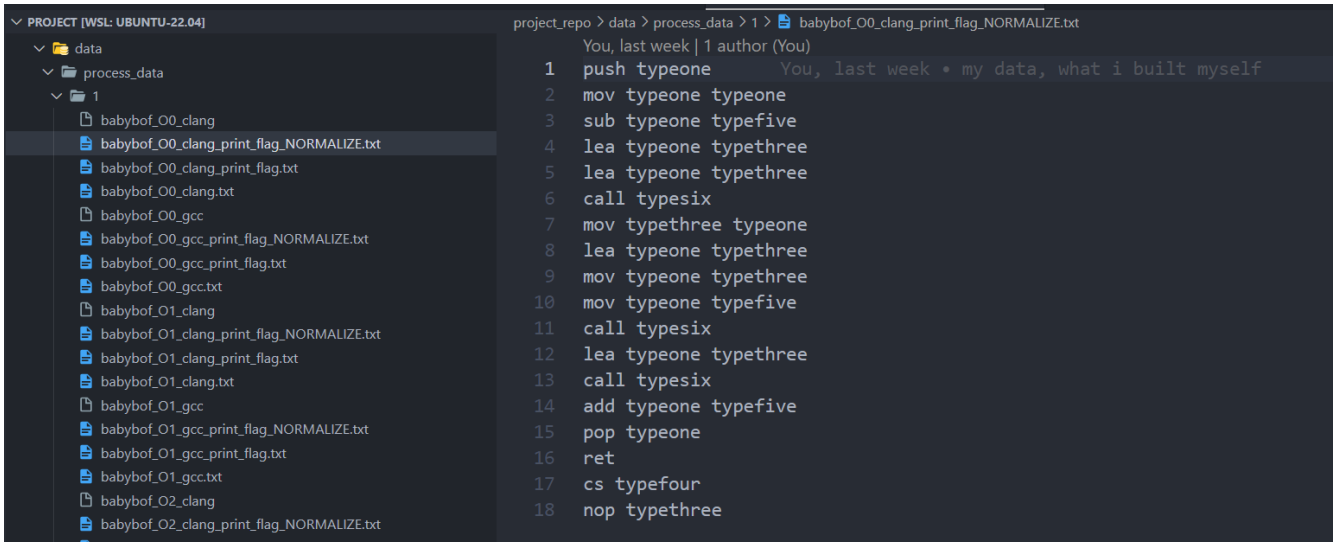
– Bước thứ 2 ta sẽ trích xuất assembly code của function cụ thể trong file assembly file vừa lấy được:
  - Đây là dict chỉ định các function cụ thể được lấy trong từng file .c:

```
get_funct = {"babybof": "print_flag",
             "babysteps": "ask_baby_name",
             "beginners_pwn": "win",
             "bof3": "read_canary",
             "cheap": "readn",
             "crash_override": "win_crash",
             "function_overwrite": "calculate_story_score",
             "Guest_game_1": "increment",
             "Guest_game_2": "do_stuff",
             "handy_shellcode": "vuln",
             "justpwnit": "justpwnit",
             "leak_flag": "vuln_leak",
             "picker_IV": "win_IV",
             "program-redacted": "tgetinput",
             "reader": "menu",
             "record_keeper": "get_record",
             "ropfu": "vuln_ropfu",
             "RPS": "play",
             "save_tyger2": "cell",
             "share": "check",
             "stack_cache": "UnderConstruction",
             "stonk": "free_portfolio",
             "stringzz": "printMessage1",
             "Unsubscriptions_Are_Free": "processInput",
             "x-sixty-what": "vuln_60"
             }
```

- Đây là file assembly code của function cụ thể được trích xuất ra từ file assembly code của cả binary file



– Bước thứ 3 ta sẽ tiến hành normalize để có được file như bên dưới:



- Cách thức normalize được giải thích cụ thể dưới đây:

Ở đây ta chia các operands thành 8 loại (cột bên trái), sẽ được gán nhãn tương ứng với cột bên phải. Ta buộc phải thực hiện normalize là vì:

- Tránh thất thoát dữ liệu, bởi vị operand sẽ có ý nghĩa trực tiếp với ngữ nghĩa của instruction, của function, của luồng thực thi function
- Việc normalize giúp giảm độ phức tạp của dữ liệu, tổng quát hóa nhưng lại không làm mất mát nhiều thông tin, giúp model học tốt hơn

Original operand	Normalized operand
General Register	TypeOne
Direct Memory Reference	TypeTwo
Memory Ref [Base Reg + Index Reg]	TypeThree
Memory Reg [Base Reg + Index Reg + Displacement]	TypeFour
Immediate Value	TypeFive
Immediate Far Address	TypeSix
Immediate Near Address	TypeSeven
Other Type	TypeEight

- Ví dụ các instructions sẽ được normalize như sau:

Original instruction	Normalized instruction
push ebp	push TypeOne
jl loc 804B271	jl TypeSeven
sub esp, 18h	sub TypeOne, TypeFive
mov dword ptr [esp+74h], 0	mov TypeThree, TypeFive
lea eax, [ebp+esi*2+0]	lea TypeOne, TypeFour
call exit	call TypeSix

- Build dataset, word2vec, train model CNN+LSTM được thực hiện trên **Colab** dùng Python, cụ thể sẽ được trình bày dưới đây
- Build dataset

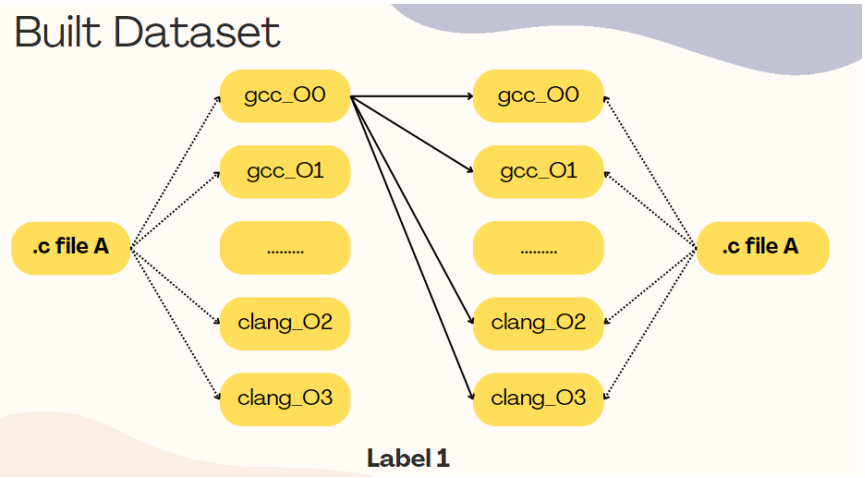
Dataset :

```
[79] feature_names = ['funct1', 'funct2', 'label']

csv_file_path = '/content/drive/MyDrive/process_data/dataset.csv'

with open(csv_file_path, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(feature_names)
    writer.writerows(data)

df = pd.read_csv(csv_file_path, index_col=False)
df = df.sample(frac=1, random_state=42)
df.head()
```

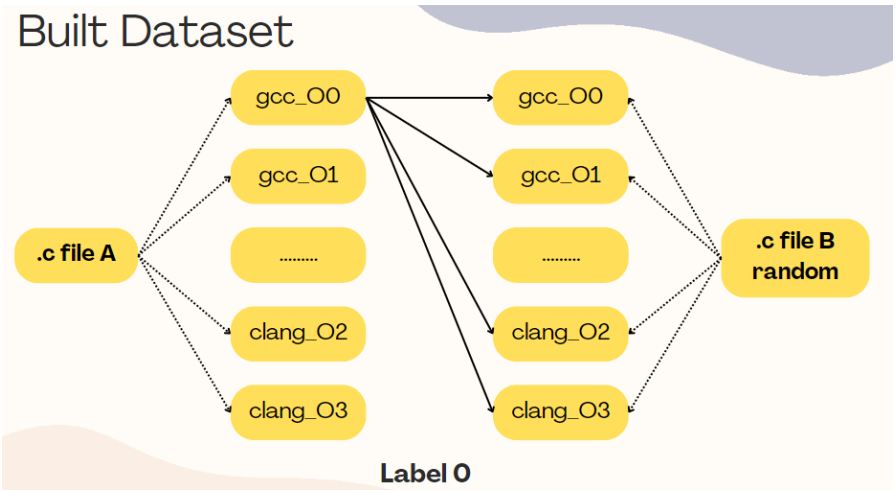
	funct1	funct2	label
596	endbr64 push typeone lea typeone typethree lea...	endbr64 push typeone mov typeone typeone sub t...	0
3370	push typeone push typeone sub typeone typefive...	push typeone mov typeone typeone sub typeone t...	0
3048	sub typeone typefive lea typeone typethree xor...	endbr64 push typeone mov typeone typeone sub t...	1
2908	sub typeone typefive lea typeone typethree xor...	endbr64 push typeone push typeone sub typeone ...	0
8	endbr64 push typeone lea typeone typethree lea...	endbr64 push typeone mov typeone typeone sub t...	0

Ở đây ta có 2 features: funct1, funct2. Đây là các assembly code đã normalize của các function.

Label là 0 hoặc 1 sẽ được xác định như sau:



Cùng 1 file source code .c được compile ra 8 versions binary file, thì cũng chính là bản thân nó, tức là có sự tương đồng, do đó ở đây label 1 được tạo bằng cách kết hợp xáo trộn các version của chính cùng 1 source code.



Label 0 được generate bằng cách dùng một file xác định và một file khác được random trong số file còn lại của tập data. Mỗi file đều có 8 versions, ta sẽ kết hợp, xáo trộn để có được label 1.

- Word2vec

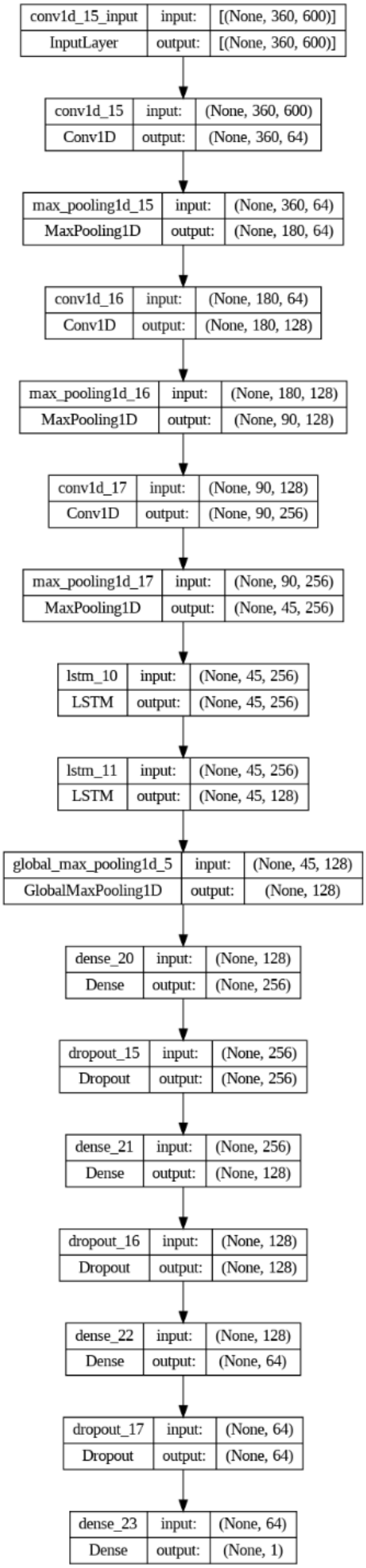
Ta có list 71 words trong toàn bộ dataset như sau:

Ở đây dùng model word2vec cụ thể là model skip-gram để vector hóa các instructions đã được normalize:

Word	Vector
push	(0.17, 0.34, ... 0.29)
typeone	(0.05, 0.27, ... 0.18)

- Model

Ở đây triển khai thực tế bọn em dùng model CNN+LSTM :

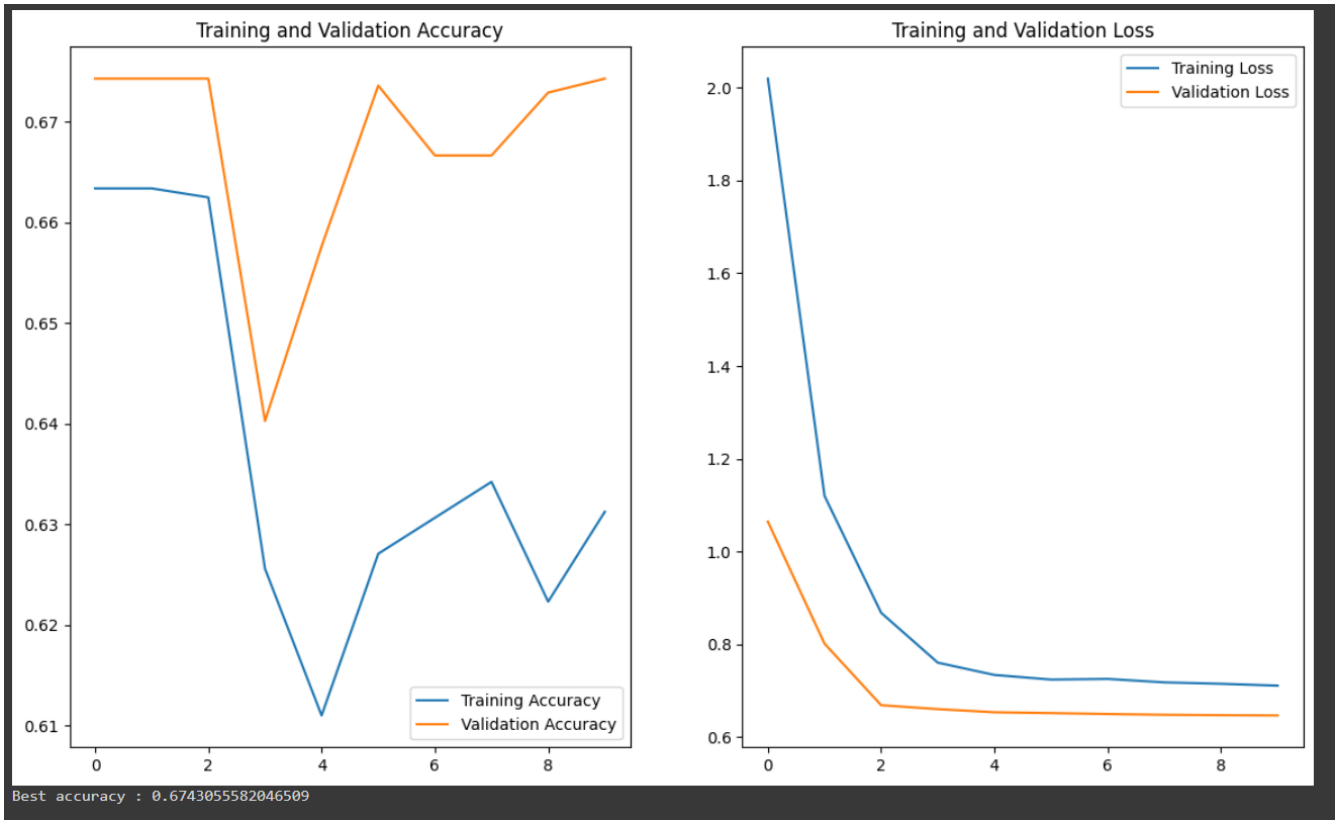

conv1d_15_input | input: | [(None, 360, 600)]
InputLayer | output: | [(None, 360, 600)]
conv1d_15 | input: | (None, 360, 600)
Conv1D | output: | (None, 360, 64)
max_pooling1d_15 | input: | (None, 360, 64)
MaxPooling1D | output: | (None, 180, 64)
conv1d_16 | input: | (None, 180, 64)
Conv1D | output: | (None, 180, 128)
max_pooling1d_16 | input: | (None, 180, 128)
MaxPooling1D | output: | (None, 90, 128)
conv1d_17 | input: | (None, 90, 128)
Conv1D | output: | (None, 90, 256)
max_pooling1d_17 | input: | (None, 90, 256)
MaxPooling1D | output: | (None, 45, 256)
lstm_10 | input: | (None, 45, 256)
LSTM | output: | (None, 45, 256)
lstm_11 | input: | (None, 45, 256)
LSTM | output: | (None, 45, 128)
global_max_pooling1d_5 | input: | (None, 45, 128)
GlobalMaxPooling1D | output: | (None, 128)
dense_20 | input: | (None, 128)
Dense | output: | (None, 256)
dropout_15 | input: | (None, 256)
Dropout | output: | (None, 256)
dense_21 | input: | (None, 256)
Dense | output: | (None, 128)
dropout_16 | input: | (None, 128)
Dropout | output: | (None, 128)
dense_22 | input: | (None, 128)
Dense | output: | (None, 64)
dropout_17 | input: | (None, 64)
Dropout | output: | (None, 64)
dense_23 | input: | (None, 64)
Dense | output: | (None, 1)

Mô hình này là một kết hợp giữa mạng Convolutional Neural Network (CNN) và Long Short-Term Memory (LSTM). Dưới đây là mô tả chi tiết về các phần chính của mô hình:

- Convolutional Neural Network (CNN):
  - Sử dụng ba lớp Conv1D với số lượng bộ lọc tăng dần là 64, 128, 256 và kernel size là 3.
  - Mỗi lớp Conv1D được theo sau bởi lớp MaxPooling1D để giảm kích thước của đầu ra.
- Long Short-Term Memory (LSTM):
  - Hai lớp LSTM được đặt liên tiếp nhau với số lượng units là 256, 128
  - Cả hai lớp LSTM đều có return_sequences=True, cho phép trả về chuỗi đầu ra từ mỗi bước thời gian.
- Global Max Pooling: Lớp GlobalMaxPooling1D được sử dụng để trích xuất đặc trưng quan trọng từ toàn bộ chuỗi thời gian sau khi các lớp CNN và LSTM đã được áp dụng.
- Fully Connected Layers:
  - Ba lớp Dense với số lượng đơn vị là 256, 128, và 64 tương ứng được sử dụng với activation "relu" để học các biểu diễn phi tuyến tính từ đặc trưng trích xuất trước đó.
  - Mỗi lớp Dense được theo sau bởi lớp Dropout với tỷ lệ 0.5 để ngăn chặn overfitting.
- Output Layer: Một lớp Dense cuối cùng với activation "tanh"

C. Kết quả thực nghiệm

- Dùng 4800 mẫu chia tỉ lệ train:test là **7:3**
- Dùng word2vec để convert các từ thành vector có dimension là **300**
- Accuracy đạt được là **0.6743055582046509**



Ta thấy là độ chính xác không được cao là do:

- Tập dữ liệu khá nhỏ chỉ 4800 mẫu từ 25 files code c, các files này nhỏ do đó các functions cũng đơn giản, chưa cho thấy sự khác biệt quá rõ rệt, dữ liệu không quá tốt thì model cũng không thể học tốt được
- Do giới hạn về phần cứng nên model cũng đơn giản, không phức tạp bằng paper

D. Hướng phát triển

- Hướng phát triển tiềm năng của đề tài này trong tương lai:
  - Obfuscation là một kỹ thuật được sử dụng để làm cho mã nguồn trở nên khó hiểu và khó phân tích, thường được áp dụng để bảo vệ mã nguồn khỏi việc dịch ngược và sao chép không cho phép. Do đó để loại bỏ hạn chế hiện có của phương pháp paper đề xuất ta sẽ dùng các kĩ thuật deobfuscation kết hợp với phương pháp hiện tại của bài báo nhằm loại bỏ sự làm rối, làm mờ mã nguồn hay mã nhị phân để đạt được hiệu suất, kết quả, độ chính xác tốt trên mỗi software mà không cần biết source code, cho dù có bị làm rối mã

- Để cải thiện độ chính xác của phương pháp này, ta có thể dùng các Model DeepLearning khác nhau cho các phân loại dữ liệu khác nhau. Ví dụ như: dùng BiLSTM model cho tập dữ liệu được compile khác nhau về kiến trúc CPU, dùng CNN+LSTM model cho tập dữ liệu khác nhau về compilers
- Thử nghiệm nhiều network models hơn. Để tính toán về sự tương đồng có thể dùng các phương pháp khác nhau như là Hamming distance of static binary features (Taheri et al., 2020a) cho việc phát hiện sự tương đồng nhị phân
- Xây dựng ứng dụng có User Interface, cho phép người dùng upload software của họ lên, mà không cần phải sợ lộ source code, ta không cần lo lắng rằng họ đã dùng các kĩ thuật làm rối code, ta vẫn có thể xây dựng model sẵn sàng cho việc kiểm tra từng function trong software của họ, với 1 thư viện chứa các function độc hại có tính thực tế cao của chúng ta.

Tính ứng dụng của đề tài

- Bảo mật Phần Mềm: Đề tài cung cấp một phương pháp để phát hiện sự tương đồng giữa các đoạn mã nhị phân. Điều này có thể hữu ích trong việc phân loại và phát hiện các biến thể của mã độc hại. Có thể sử dụng phương pháp của bài báo để xác định mối liên quan giữa các mẫu mã độc hại khác nhau.
- Có thể phát triển thành công cụ hữu ích cho quy trình phát triển phần mềm: Phương pháp paper đề xuất có thể phát triển thành công cụ hữu ích cho DevSecOps, có thể tích hợp vào quy trình CI/CD nhằm giảm thiểu rủi ro phần mềm có các lổ hỏng bảo mật, có chứa các functions độc hại,...
- Giúp phát hiện các lổ hỏng bảo mật, mã độc hại trong phần mềm mà không cần biết source code, đảm bảo tính bảo mật và sở hữu trí tuệ cho chủ sở hữu phần mềm
- Chống sao chép mã nguồn: Có thể dùng cho chống sao chép và đánh cắp mã nguồn. Nhà phát triển muốn bảo vệ mã nguồn của họ có thể sử dụng phương pháp phát hiện sự tương đồng để kiểm tra xem có bất kỳ đoạn mã nào giống với mã nguồn của họ.

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