ĐẠI HỌC QUỐC GIA HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG

NGUYỄN PHÚC HẢI

KHÓA LUẬN TỐT NGHIỆP

NGHIÊN CỨU PHƯƠNG PHÁP PHÁT HIỆN SỰ TƯƠNG ĐỒNG MÃ NHỊ PHÂN CỦA CHƯƠNG TRÌNH PHẦN MỀM DỰA TRÊN CÁC MÔ HÌNH NGÔN NGỮ VÀ HỌC SÂU

A STUDY ON BINARY CODE SIMILARITY DETECTION USING LANGUAGE MODEL AND DEEP NEURAL NETWORKS

CỬ NHÂN NGÀNH AN TOÀN THÔNG TIN

TP. Hồ Chí Minh, 2024

ĐẠI HỌC QUỐC GIA HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG

NGUYỄN PHÚC HẢI - 20521281

KHÓA LUẬN TỐT NGHIỆP

NGHIÊN CỨU PHƯƠNG PHÁP PHÁT HIỆN SỰ TƯƠNG ĐỒNG MÃ NHỊ PHÂN CỦA CHƯƠNG TRÌNH PHẦN MỀM DỰA TRÊN CÁC MÔ HÌNH NGÔN NGỮ VÀ HỌC SÂU

A STUDY ON BINARY CODE SIMILARITY DETECTION USING LANGUAGE MODEL AND DEEP NEURAL NETWORKS

CỬ NHÂN NGÀNH AN TOÀN THÔNG TIN

GIẢNG VIÊN HƯỚNG DẪN:
ThS. Ngô Khánh Khoa

TP.Hồ Chí Minh - 2024

LỜI CẢM ƠN

Trong thời gian thực hiện nghiên cứu khóa luận tốt nghiệp, nhóm đã nhân được nhiều sự giúp đỡ, đóng góp ý kiến và chỉ bảo nhiệt tình của thầy cô, bạn bè. Với sự giúp đỡ này đã giúp nhóm rất nhiều trong việc cũng cố kiến thức và giải đáp những thắc mắc còn tồn đọng.

Nhóm xin gửi lời cám ơn chân thành đến thầy Ngô Khánh Khoa giảng viên khoa Mạng máy tính và truyền thông dữ liệu trường đại học Công Nghệ Thông Tin, người đã tận tình hướng dẫn, chỉ bảo nhóm trong suốt quá trình làm khóa luận tốt nghiệp vừa qua.

Với những kiến thức đã đạt được tại đồ án này nhóm báo cáo có thể tự tin hơn trên chặng đường tương lai sắp tới và trong cuộc sống sau này. Nhóm xin được chúc thầy và các cán bộ giảng viên đang công tác tại trường thật nhiều sức khỏe và thành công trong cuộc sống.

Nguyễn Phúc Hải

MỤC LỤC

DANH MỤC CÁC KÝ HIỆU, CÁC CHỮ VIẾT TẮT

CVE	**Common Vulnerabilities and Exposures**
CNN	**Convolutional Neural Network**
GNN	**Graph Neural Networks**
LSTM	**Long Short-Term Memory**
NLP	**Natural language processing**
CFG	**Control Flow Graph**
IR	**Intermediate representation**
bb	**Danh sách các câu lệnh trong 1 basic block**
S	**Danh sách các strand của một basic block**
MD5	**Hàm băm Message-Digest algorithm 5**
TP	**True Positive**
FP	**False Positive**
TN	**True Negative**
FN	**False Negative**
N	**Số lượng các chức năng tương đồng được xếp hạng**
$rank_i$	**Vị trí của chức năng thực sự tương đồng đầu tiên**
$\sum_{i=1}^{N}$	**Tổng của các thành phần từ $i = 1$ đến N**

DANH MỤC CÁC HÌNH VẼ

DANH MỤC CÁC BẢNG BIỂU

TÓM TẮT KHÓA LUẬN

Ngày nay phát hiện sự tương đồng trong mã nhị phân đang ngày càng trở nên là một kỹ thuật quan trọng trong nhiều lĩnh vực, bao gồm an ninh mạng, phân tích phần mềm và nghiên cứu bảo mật.

Đầu tiên trong an ninh mạng, phát hiện sự tương đồng trong mã nhị phân có thể được sử dụng để phát hiện và ngăn chặn các cuộc tấn công mạng và phần mềm độc hại. Ví dụ, các phần mềm độc hại thường được tái sử dụng hoặc chỉnh sửa từ các mã độc đã có sẵn. Bằng cách so sánh mã nhị phân của các phần mềm độc hại, các nhà nghiên cứu có thể phát hiện các mối tương quan và xác định các phần mềm độc hại mới.

Tiếp đến, trong phân tích phần mềm, phát hiện sự tương đồng trong mã nhị phân có thể được sử dụng để tìm kiếm các lỗ hổng bảo mật, lỗi và thiếu sót trong mã nguồn. Ví dụ, các nhà phát triển phần mềm có thể sử dụng kỹ thuật này để xác định các mã trùng lặp, từ đó tối ưu hóa mã nguồn và giảm thiểu rủi ro bảo mật.

Cuối cùng trong nghiên cứu bảo mật, phát hiện sự tương đồng trong mã nhị phân có thể được sử dụng để nghiên cứu các mô hình tấn công và phát triển các phương pháp bảo vệ mới. Ví dụ, các nhà nghiên cứu có thể sử dụng kỹ thuật này để phân tích các mã độc khác nhau và tìm ra các đặc điểm chung. Từ đó, họ có thể phát triển các phương pháp phát hiện và ngăn chặn các cuộc tấn công tương tự.

Cùng với phương pháp phát hiện sự tương đồng mã nhị phân thông thường thì phương pháp phát hiện tương đồng trong mã nhị phân dựa trên học sâu cũng đang ngày càng phát triển và càng minh chứng cho sự hiệu quả vượt trội của nó so với các phương pháp truyền thống. Các phương pháp phát hiện sự tương

đồng trong mã nhị phân truyền thống thường dựa trên các kỹ thuật thống kê hoặc kỹ thuật so sánh ký tự. Tuy nhiên, các kỹ thuật này có thể không hiệu quả trong trường hợp mã nhị phân được mã hóa hoặc biến đổi. Trong những năm gần đây, các phương pháp phát hiện sự tương đồng trong mã nhị phân dựa trên học sâu đã được phát triển và cho thấy hiệu quả cao hơn. Các phương pháp này sử dụng các mô hình học sâu để học các đặc điểm của mã nhị phân. Từ đó, chúng có thể phát hiện các tương đồng giữa các mã nhị phân ngay cả khi các mã này được mã hóa hoặc biến đổi. Tóm lại, phát hiện sự tương đồng trong mã nhị phân là một kỹ thuật quan trọng với nhiều ứng dụng trong thực tế. Các phương pháp phát hiện sự tương đồng trong mã nhị phân dựa trên học sâu đang ngày càng được phát triển và cho thấy hiệu quả cao hơn.

Trong khóa luận tốt nghiệp của mình, chúng tôi đã đề xuất một công cụ phát hiến sự tương đồng trong mã nhị phân dựa trên học sâu mang tên BinShoo. Công cụ này sử dụng phương pháp Proc2vec+ mà chúng tôi đã nâng cấp dựa trên nguyên mẫu Proc2vec được đưa ra của phương pháp Zeek đã được giới thiệu trước đó để chuyển đổi các bytes code của chức năng nhị phân thành vector, sau đó đưa vào mô hình học sâu mà chúng tôi đã nghiên cứu và kết hợp dựa trên các mô hình trước đó để bổ trợ đưa ra kết quả tối ưu nhất. Chúng tôi cũng đã thực hiện các thực nghiệm và đánh giá để so sánh độ hiệu quả của các thay đổi khi áp dụng vào mô hình của chúng tôi.

CHƯƠNG 1. TỔNG QUAN

Nội dung của chương này giới thiệu về vấn đề chúng tôi sẽ nghiên cứu và các nghiên cứu đã có trước đó. Đồng thời chúng tôi cũng sẽ trình bày phạm vi và cấu trúc của khóa luận tốt nghiệp.

1.1. Giới thiệu vấn đề

Phát hiện sự tương đồng của mã nhị phân (Binary code similarity detection) là vấn đề mà đầu vào là biểu diễn nhị phân của một cặp hàm, chương trình, v.v., và đầu ra là một giá trị số thể hiện mức độ "tương đồng" giữa chúng. Điều này có thể mở rộng ra để so sánh các tập tin nhị phân nhằm tìm ra các đoạn mã giống nhau hoặc tương tự nhau, hoặc tìm kiếm một đoạn mã trong tập tin nhị phân có sự tương đồng với một đoạn mã nhị phân được cung cấp (Bài toán tìm kiếm nhị phân)..

Mặc dù công việc này có vẻ đơn giản, nhưng nó lại có nhiều ứng dụng thực tiễn. Một ví dụ điển hình là việc phát hiện các lỗ hổng đã biết trong phần mềm, phát hiện đạo văn phần mềm, phân tích các bản vá, v.v. Trong bối cảnh hiện nay khi phần mềm độc hại và mã độc ngày càng gia tăng, lĩnh vực này cũng có thể được áp dụng để phát hiện mã độc. Tuy nhiên, đây là một lĩnh vực không dễ nghiên cứu vì phần mềm thường được biên dịch bằng các công cụ khác nhau, trên các kiến trúc hệ thống khác nhau, với các tùy chọn biên dịch khác nhau, và thậm chí phiên bản phần mềm trước khi biên dịch cũng có thể khác biệt nhiều.

1.2. Các nghiên cứu liên quan

Các phương pháp truyền thống để phát hiện sự tương đồng trong mã nhị phân thường dựa trên việc phân tích các đặc trưng cụ thể của mã nhị phân, đặc biệt là đồ thị luồng điều khiển (CFG - Control Flow Graph) của các hàm. Những phương pháp này chủ yếu tập trung vào việc phân tích và hiểu cú pháp của chương trình để so sánh sự tương đồng giữa các hàm.

Tuy nhiên, các phương pháp này gặp phải một số hạn chế và không đảm bảo hiệu quả trong mọi trường hợp. Một số hạn chế chính bao gồm:

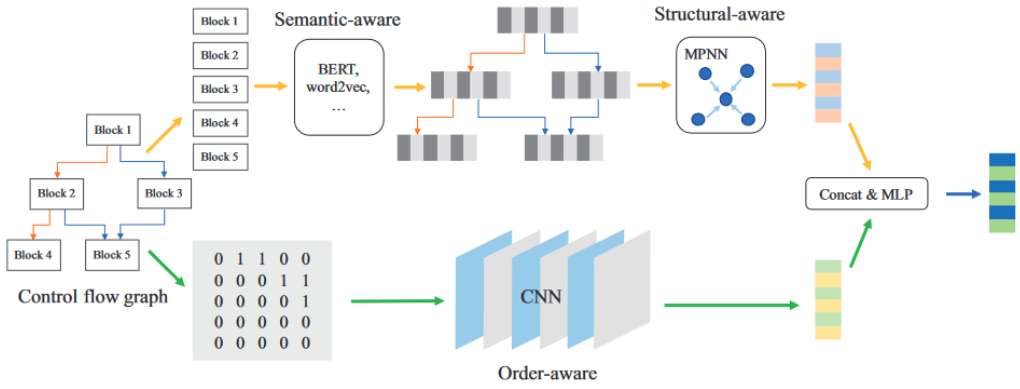
- Tính không ổn định: CFG của mã nhị phân có thể thay đổi dựa trên các tùy chọn tối ưu hóa của trình biên dịch, như tối ưu hóa đường đi, tối ưu hóa biểu thức, hoặc tối ưu hóa cấu trúc điều khiển. Ngay cả khi sử dụng các decompiler khác nhau cho cùng một tập tin nhị phân, CFG được trích xuất cũng có thể rất khác nhau. Điều này làm cho việc so sánh đồ thị để tìm kiếm các đường đi chung trở nên khó khăn và không ổn định.

- Tốn thời gian: Việc xây dựng CFG và so sánh đồ thị là một quá trình tốn thời gian, đặc biệt đối với các chương trình nhị phân lớn và phức tạp. Việc xử lý và so sánh các basic block, cấu trúc điều khiển và các điểm nhảy trong mã nhị phân đòi hỏi nhiều tài nguyên tính toán và bộ nhớ.

Vì những hạn chế trên, các phương pháp phát hiện tương đồng mã nhị phân truyền thống đã trở nên không hiệu quả và không ổn định trong việc so sánh sự tương đồng giữa các mã nhị phân. Do đó, nghiên cứu các phương pháp mới tiên tiến tiên tiến đã trở thành xu hướng để cải thiện khả năng phát hiện tương đồng mã nhị phân hiện tại.

Trong bối cảnh học máy (Machine Learning) ngày càng phát triển và trở thành trọng tâm của nhiều nghiên cứu, học máy và học sâu đã được áp dụng vào việc phát hiện sự tương đồng của mã nhị phân trong những năm gần đây. Các giải pháp hiện đại cho vấn đề này đều liên quan đến học máy.

Một số giải pháp lấy cảm hứng từ xử lí ngôn ngữ tự nhiên - Natural language processing (NLP) hay một số sử dụng mạng lưới thần kinh đồ thị - Graph Neural Networks (GNN). Thậm chí một số kết hợp cả 2 để tìm hiểu cách biểu diễn các khối cơ bản (Basic block) bằng kỹ thuật NLP và xử lý thêm các tính năng khối cơ bản trong CFG bằng GNN như Zeping Yu và nhóm của mình [14] họ cũng đã sử dụng BERT [12], một mô hình mạng thần kinh mở rộng xử dụng kiến trúc tranformer để đào tạo trước mã nhị phân, sử dụng mạng thần kinh tích chập - Convolutional Neural Network (CNN) trên các ma trận kề để trích xuất thông tin. Tuy hiệu quả có cải thiện nhưng, BERT cần một lượng dữ liệu lớn để đào tạo trước mô hình, phải đi qua một mạng lưới thần kinh phức tạp với nhiều tham số, tác động đáng kể đến hiệu suất và cần khả năng tính toán.



Hình 1.1: *Mô hình Order Matters - Kết hợp NLP và GNN trong phát hiện sự tương đồng mã nhị phân*

Shalev và các cộng sự đã cho phát triển Zeek [8] thực hiện phân tích luồng dữ liệu kết hợp với Vex-IR ở và chuyển đổi thành vector. Sau đó, dùng mạng nơ-ron hai lớp để huân luyện và thực hiện phát hiện tương đồng giữa các kiến trúc. Cách tiếp cận này là đề xuất tiên tiến nhất kết hợp các biểu diễn trung gian, phân tích luồng dữ liệu và học máy. Công trình này vượt trội so với nghiên cứu trước đây là Gitz [2] cả về độ hiệu quả và tốc độ.

Tiếp theo Luca và các cộng sự đã giới thiệu SAFE [7], một công cụ dựa trên sự tiến bộ của xử lí ngôn ngữ tự nhiên, tỏ ra hiệu quả cao trong việc phát hiện

sự tương đồng trong mã nhị phân kiên trúc chéo (cross-architecture, tức là biên dịch trên các kiến trúc khác nhau).

Năm 2020, công cụ BinDeep [9] của các tác giả Donghai Tian, Xiaoqi Jia, Rui Ma, Shuke Liu, Wenjing Liu, và Changzhen Hu, được xuất bản trong tạp chí Expert Systems with Applications, các tác giả đề xuất một phương pháp học sâu để phát hiện sự tương đồng trong mã nhị phân. Phương pháp này đầu tiên trích xuất chuỗi lệnh từ hàm nhị phân và sử dụng mô hình nhúng lệnh để vector hóa các đặc trưng của lệnh. Sau đó, BinDeep áp dụng mô hình học sâu Recurrent Neural Network (RNN) để xác định các loại cụ thể của hai hàm để so sánh sau này. Dựa vào thông tin loại, BinDeep chọn mô hình học sâu tương ứng để so sánh sự tương đồng. Cụ thể, BinDeep sử dụng mạng nơ-ron Siamese, kết hợp giữa LSTM và CNN để đo lường sự tương đồng của hai hàm mục tiêu. Khác với mô hình học sâu truyền thống, mô hình lai này tận dụng lợi thế của học cấu trúc không gian CNN và học chuỗi LSTM. Đánh giá cho thấy phương pháp của họ đạt được kết quả tốt trong việc so sánh mã nhị phân giữa các kiến trúc, trình biên dịch, mức tối ưu hóa và phiên bản khác nhau.

Vào năm 2022, Hao Wang và các cộng sự đã phát triển jTrans [11] một mô hình phát hiện sự tương đồng dựa trên Transformer, công cụ này sau đó đã được phát triển và cải tiến trở thành kTrans. Công cụ này kết hợp các mô hình NLP, nắm bắt ngữ nghĩa của các câu lệnh, cùng với trích xuất CFG để nắm bắt thông tin luồng điều khiển để suy ra ngữ nghĩa của mã nhị phân. Bên cạnh đó công cụ này cũng đề xuất kết hợp thông tin luồng điều khiển vào kiến trúc Transformer bằng cách sử đổi Transformer để nắm bắt thông tin luồng điều khiển. Phương pháp này đã tiến hành đá giá với một loạt các công cụ trước đó như Genius [4], Gemini [13], SAFE [7], Asm2Vec [3], GraphEm [6] và OrderMatters [14] và cho thấy hiệu quả vượt trội.

1.3. Tính ứng dụng

Nghiên cứu này được cảm hứng từ Zeek [8], chúng tôi dựa trên ý tưởng sử dụng biểu diễn trung gian và hàm băm để chuyển đổi chức năng nhị phân thành các vector có thể đưa vào mô hình học sâu.

Chúng tôi thấy rằng việc biểu diễn các chức năng nhị phân dưới dạng vector có thể mang lại nhiều lợi ích trong nhiệm vụ phát hiện sự tương đồng. Cùng với sự phát triển của học sâu, chúng tôi muốn khai thác tiềm năng của học sâu trong việc mô hình hóa và phân tích chức năng nhị phân.

Bằng cách áp phương pháp này chúng tôi hy vọng phát triển được một công cụ mạnh mẽ và chính xác trong việc xác định các chức năng nhị phân tương đồng. Mang lại kết quả tốt có thể đóng góp vào lĩnh vực phân tích mã độc, phát hiện lỗ hổng và kiểm tra bảo mật, giúp cải thiện quá trình phân tích mã nhị phân và tăng cường khả năng phát hiện các mối đe dọa trong phần mềm.

1.4. Những thách thức

Như đã trình bày, việc phát hiện sự tương đồng trong mã nhị phân không hề đơn giản. Một số lý do chính gây khó khăn bao gồm:

- Sự đa dạng của mã nhị phân: Mã nhị phân có thể có nhiều đặc điểm và cấu trúc khác nhau, từ các chương trình nhỏ đến các ứng dụng phức tạp, từ tập tin Windows đến tập tin Linux. Phát hiện sự tương đồng giữa các mã nhị phân đa dạng này yêu cầu khả năng xử lý và phân tích linh hoạt.

- Cấu trúc và độ phức tạp của mã nhị phân: Mã nhị phân thường có cấu trúc phức tạp với nhiều khối mã, hàm, và lời gọi hàm. Điều này làm cho việc phân tích và so sánh trở nên phức tạp hơn, đòi hỏi phải xem xét nhiều khía cạnh khác nhau của mã.

- Hiệu năng và tốc độ xử lý: Xử lý các mã nhị phân lớn và phức tạp đòi hỏi

nhiều tài nguyên tính toán và bộ nhớ. Việc tìm kiếm và so sánh sự tương đồng trong các mã nhị phân có thể tiêu tốn nhiều tài nguyên, đặc biệt là với các dự án lớn..

- Các tập tin nhị phân tuy cùng một mã nguồn, nhưng nếu biên dịch với các ngữ cảnh khác nhau (Mức độ tối ưu hóa, kiến trúc máy tính, trình biên dịch,...) cũng có thể cho ra các tập tin nhị phân rất khác.

1.5. Mục tiêu, đối tượng, và phạm vi nghiên cứu

1.5.1. Mục tiêu nghiên cứu

Mục tiêu của khóa luận này là sử dụng ứng dụng của học sâu để xây dựng và phát triển phương pháp phát hiện sự tương đồng trong mã nhị phân thông qua quá trình chuyển đổi các chức năng nhị phân thành vector.

1.5.2. Đối tượng nghiên cứu

Đối tượng nghiên cứu:

- Các thành phần liên quan đến tập tin nhị phân.
- Biểu diễn trung gian - Intermediate Representation.
- Học sâu.
- Proc2vec (Prov2vec+): Phương pháp chuyển từ bytes code sang vector.
- BinShoo: Mô hình phát hiện sự tương đồng trong mã nhị phân mới được phát triển dựa trên học sâu.

1.5.3. Phạm vi nghiên cứu

Trích xuất ngữ nghĩa, thuộc tính của các chức năng nhị phân thông qua quá trình chuyển đổi các chức năng nhị phân thành vector bằng phương pháp

Proc2vec cải tiến. Thực hiện kết hợp và xây dựng được một mô hình phát hiện sự tương đồng trong mã nhị phân mới dựa trên các mô hình học sâu đã biết trước đó. Cuối cùng thực hiện đánh giá khả năng phát hiện sự tương đồng của các chức năng của phương pháp đã nghiên cứu.

1.5.4. Cấu trúc của khóa luận tốt nghiệp

Chúng tôi trình bày nội dung của khóa luận theo cấu trúc như sau:

- Chương 1: Giới thiệu tổng quan về đề tài của khóa luận tốt nghiệp và những nghiên cứu có liên quan trước đó.
- Chương 2: Trình bày cơ sở lý thuyết và những kiến thức nền tảng phục vụ cho khóa luận.
- Chương 3: Trình bày phương pháp phát hiên sự tương đồng trong mã nhị phân mà chúng tôi đã nghiên cứu.
- Chương 4: Trình bày các thí nghiệm và đánh giá.
- Chương 5: Kết luận và hướng phát triển của đề tài.

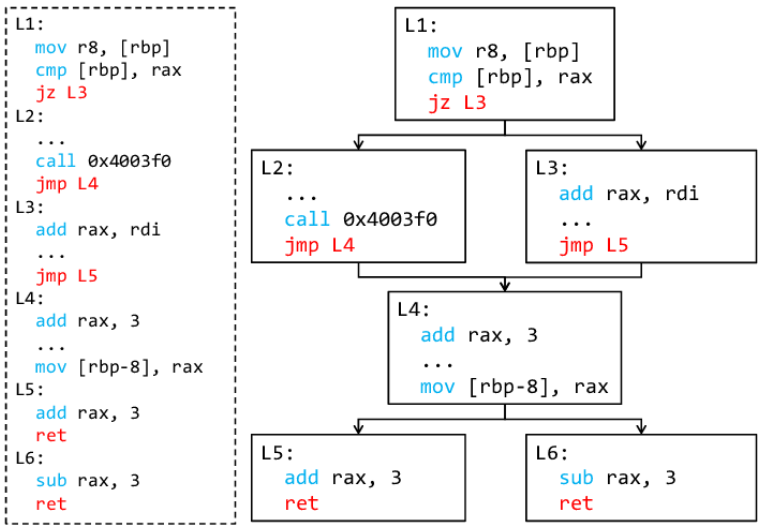
CHƯƠNG 2. CƠ SỞ LÝ THUYẾT

Trong chương này chúng tôi sẽ trình bày các cơ sở lý thuyết liên quan của đề tài: Bao gồm các thành phần liên quan đến tập tin nhị phân, ngôn ngữ biểu diễn trung gian (Intermediate representation) và mô hình học sâu.

2.1. Các thành phần liên quan đến tập tin nhị phân

2.1.1. Chức năng - Function

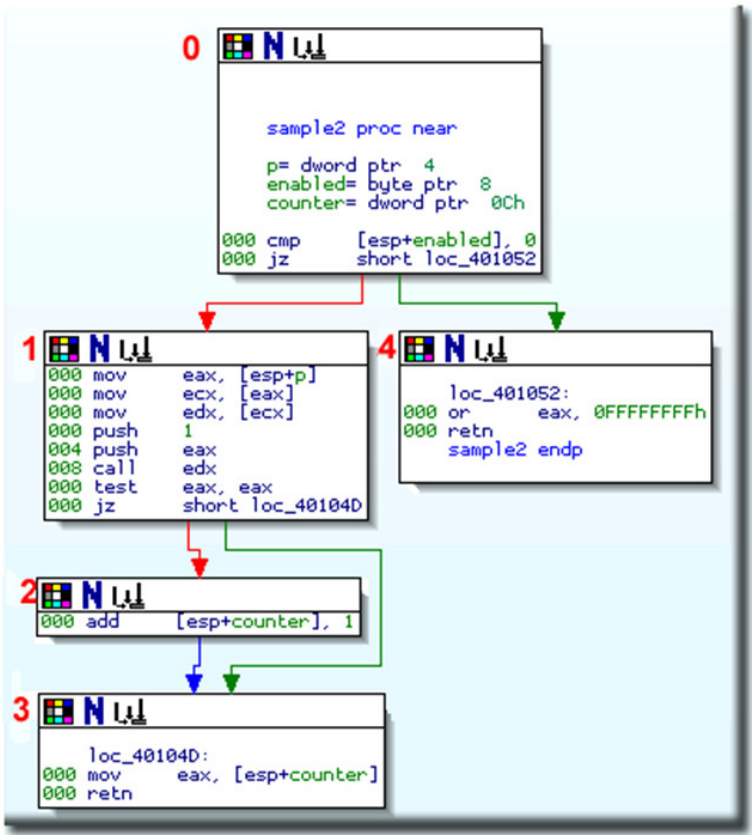
Trong nghiên cứu này, chúng tôi đề cập đến một hàm là một danh sách các câu lệnh (Instructions) được sắp xếp trong các tập tin nhị phân, và chúng được biên dịch từ một hàm mã nguồn. Do đó, nó có ngữ nghĩa cụ thể. Hơn nữa một chức năng có biểu đồ luồng điều khiển (Control-flow-graph) dùng để biểu thị thông tin luồng thực thi của nó (Tương tự như hình 2.1).



Hình 2.1: *Một ví dụ về biểu đồ luồng điều khiển của hàm nhị phân. Phần bên trái là mã tập hợp bố cục tuyến tính với các địa chỉ bước nhảy và phần bên phải là biểu đồ luồng điều khiển tương ứng. L1 và L4 là các nút bắt buộc.*

2.1.2. Basic block

Basic block là một chuỗi các lệnh liên tiếp trong mã máy, có một điểm vào và một điểm ra duy nhất. Trong file nhị phân, basic block là một đoạn mã máy được tạo ra từ một số lệnh máy và thường được xem như là một đơn vị dùng trong việc phân tích mã nhị phân (Hình 2.2).



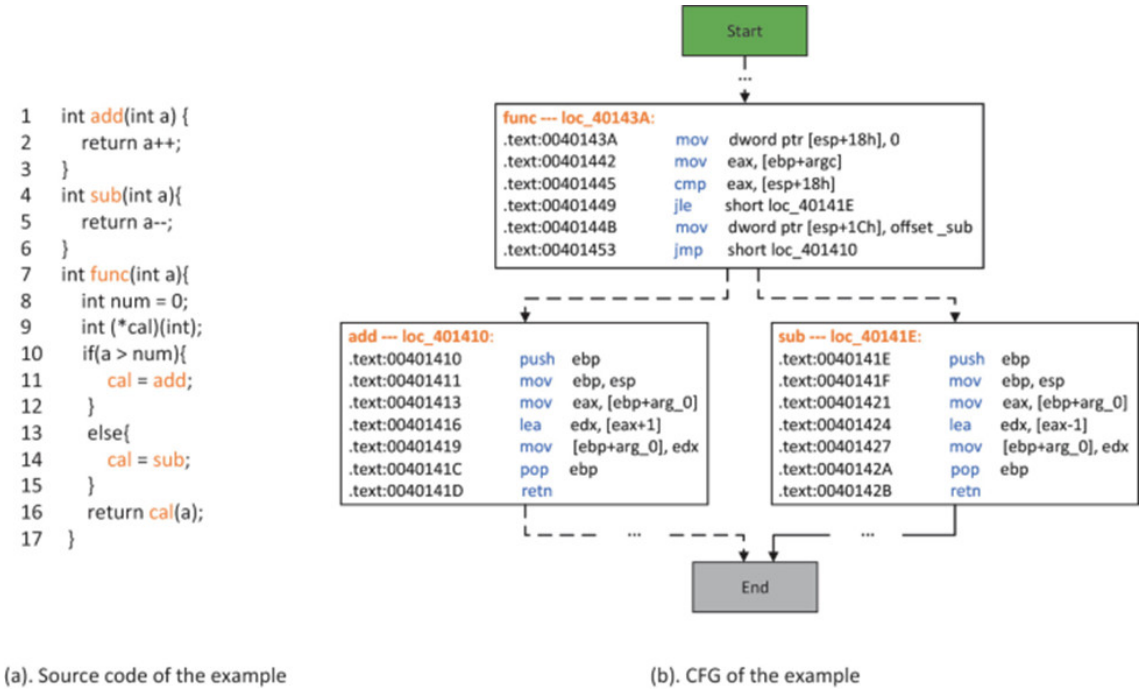
Hình 2.2: *Các basic block kế tiếp nhau được trích xuất khi dùng IDA*

2.1.3. Biểu đồ luồng điều khiển - Control flow graph

Control Flow Graph (CFG) là biểu đồ biểu diễn luồng điều khiển của chương trình máy tính hoặc chương trình nguồn dưới dạng đồ thị hướng điều khiển. Mỗi nút trong CFG đại diện cho một basic block, đại diện cho một chuỗi các lệnh liên tiếp không thể ngắt giữa các lệnh. Các cạnh trong CFG biểu thị các chuyển đổi giữa các basic block thông qua các nhánh điều kiện, nhánh vô điều

kiện hoặc các lệnh gọi hàm.

Trong phát hiện sự tương đồng trong mã nhị phân, CFG được sử dụng để so sánh cấu trúc điều khiển của các chương trình khác nhau và xác định sự tương đồng giữa chúng. Các phương pháp phát hiện sự tương đồng thường dựa trên việc so sánh các đặc trưng của CFG, quan hệ điều khiển và đường đi thực thi.



Hình 2.3: *Ví dụ về một đoạn chương trình với CFG tương ứng*

2.1.4. Strand

Trong ngữ cảnh phát hiện tương đồng mã nhị phân, thuật ngữ "Strand" thường được sử dụng để chỉ một đoạn mã nhị phân hoặc các lệnh cụ thể trong mã nhị phân. Strand là một tập hợp các câu lệnh liên quan nhau mà khi kết hợp lại, chúng sẽ tạo thành một chuỗi hoàn chỉnh thực hiện một nhiệm vụ nào đó trong chương trình.

Trong nghiên cứu này, quá trình phát hiện tương đồng mã nhị phân thường bắt đầu bằng việc chia nhỏ các mã nhị phân thành các strand. Sau đó, các thuật toán so sánh có thể được áp dụng để so sánh các strand này với nhau để xác định

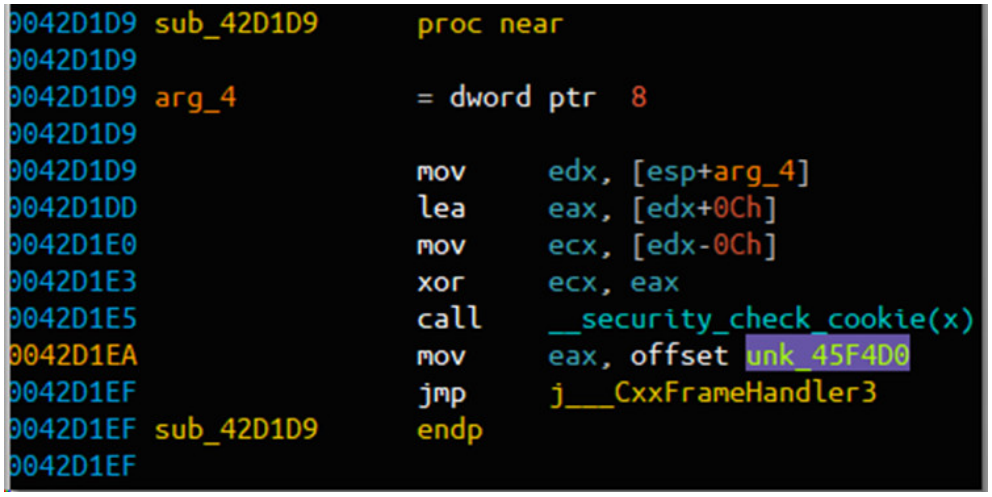
mức độ tương đồng giữa chúng. Các kỹ thuật như inlining, constant folding, và các phép tối ưu hóa khác có thể được áp dụng lên các strand để tăng cường hiệu quả của quá trình phân tích và so sánh mã nhị phân.

2.1.5. Decompiler

Decompiler là một công cụ hoặc chương trình máy tính dùng để chuyển đổi mã máy (machine code) hoặc mã nhị phân (binary code) trở lại thành mã nguồn gốc dưới dạng ngôn ngữ lập trình. Việc sử dụng decompiler có thể giúp phân tích và nằm bắt rõ hơn về chức năng và logic của một chương trình đã được biên dịch. Hiện nay có rất nhìu decompiler khác nhau như: IDA, Ghidra, Radare2, RetDec,...

Thông thường các decompiler có thể có các công dụng như sau:

- Phân tích ngược: Giúp nhà phát triển hiểu cấu trúc và logic của một chương trình khi mã nguồn không còn hoặc không có sẵn.
- Trích xuất chức năng: Decompiler cho phép chuyển đổi mã máy không cấu trúc thành dạng mã có cấu trúc (thường là assembly), giúp dễ dàng hiểu và phân tích các chức năng trong mã nhị phân.
- Phân tích bảo mật: Hỗ trợ các chuyên gia an ninh mạng phân tích mã độc, phát hiện lỗ hổng bảo mật và đánh giá rủi ro từ mã nhị phân.



Hình 2.4: *Ví dụ về một đoạn mã được chuyển đổi sang assembly bằng IDA*

2.1.5.1. IDA Pro

Ở nghiên cứu này, chúng tôi cần sử dụng decompiler để trích xuất các địa chỉ của các chức năng trong tập tin nhị phân, từ đó trích xuất ra các bytes code của các chức năng đó.

Để phục vụ cho yêu cầu này, chúng tôi quyết định chọn IDA Pro để thực hiện, ngoài là công cụ decompiler quen thuộc thì IDA Pro còn có các ưu điểm:

- Hỗ trợ phân tích mã nhị phân: IDA Pro cho phép người dùng phân tích các file thực thi (executable files), thư viện (libraries), hay các đoạn mã khác từ nhiều nền tảng khác nhau như Windows, Linux, macOS, và nhiều hệ điều hành nhúng.
- Phân tích đồ thị luồng điều khiển: Công cụ này giúp người dùng hình dung và phân tích các nhánh thực thi của chương trình, từ đó tìm ra các kết nối logic và điểm nhập nhằng (critical point) của chương trình.
- Hỗ trợ nhiều kiến trúc và định dạng file: IDA Pro hỗ trợ nhiều loại kiến trúc vi xử lý và định dạng file như ELF, PE, Mach-O, và COFF, giúp người dùng phân tích các loại file khác nhau dễ dàng.

2.1.5.2. IDA Python API

IDA Python API là một giao diện lập trình ứng dụng (API) được cung cấp bởi IDA Pro để cho phép người dùng tương tác với và điều khiển IDA Pro bằng ngôn ngữ lập trình Python. API này cho phép các nhà phân tích mã nhị phân và bảo mật sử dụng các tính năng của IDA Pro một cách tự động hóa và linh hoạt hơn thông qua việc viết các script và plugin.

Một số tính năng nổi bật của IDA Python API:

- Đọc và phân tích mã nhị phân: API cho phép người dùng mở các file thực thi và thực hiện các thao tác như phân tích đồ thị luồng điều khiển (CFG), đọc các hàm, khối lệnh, và các cấu trúc dữ liệu khác từ chương trình.
- Tự động hóa quy trình phân tích: Bằng cách sử dụng IDA Python API, chúng ta có thể viết các script để tự động hóa các quy trình phân tích, như tìm kiếm chuỗi, đặt nhãn, phát hiện các cấu trúc dữ liệu và các chuỗi lỗi.
- Tích hợp với các công cụ và workflow khác: IDA Python API có thể tích hợp và tương tác với các công cụ và quy trình làm việc khác, ví dụ như phân tích tĩnh (static analysis) từ các công cụ khác, tự động hóa các bước kiểm thử (testing automation), hoặc tạo ra các báo cáo phân tích.

2.2. Biểu diễn trung gian - Intermediate Representation

VEX Intermediate Representation (VEX-IR) là một biểu diễn trung gian được sử dụng trong phân tích mã nhị phân. Nó được phát triển bởi dự án Valgrind và hiện tại được sử dụng rộng rãi trong nhiều công cụ phân tích mã nhị phân như Angr. VEX-IR chuyển đổi mã máy cụ thể của các kiến trúc phần cứng khác

nhau thành một định dạng trung gian phổ quát, giúp dễ dàng phân tích và xử lý mã nhị phân.

2.2.1. Vex Intermediate Representation (Vex-IR)

Một trong những biểu diễn trung gian phổ biến được sử dụng là Vex Intermediate Representation (Vex-IR). Vex-IR là một ngôn ngữ trung gian dựa trên mã máy, được sử dụng để biểu diễn mã nhị phân dưới dạng các biểu thức và lệnh có cấu trúc.

Hiện nay có rất nhiều ngôn ngữ biểu diễn trung gian như LLVM-IR, BIL, nhưng trong nghiên cứu này chúng tôi lựa chọn Vex-IR vì các lí do sau:

- Kiến trúc trung gian: VEX-IR không phụ thuộc vào kiến trúc cụ thể, nghĩa là nó có thể biểu diễn mã từ nhiều kiến trúc CPU khác nhau (x86, ARM, MIPS, etc.) trong một định dạng nhất quán. Điều này giúp chuẩn hóa việc phân tích mã từ nhiều nguồn khác nhau.
- Ngôn ngữ trung gian cấp thấp: VEX-IR là ngôn ngữ trung gian cấp thấp, gần với mã máy nhưng vẫn dễ hiểu hơn đối với con người. Nó cung cấp một tập hợp các lệnh cơ bản để biểu diễn các thao tác như di chuyển dữ liệu, tính toán, và các lệnh điều khiển luồng..
- Hỗ trợ nhiều phép toán và kiểu dữ liệu: VEX-IR hỗ trợ các phép toán số học, logic, so sánh, và các phép toán trên bộ nhớ. Nó cũng hỗ trợ nhiều kiểu dữ liệu như số nguyên, số thực, và các kiểu dữ liệu vector.
- Ngoài ra Vex-IR còn có framework hỗ trợ rất mạnh mẽ là angr với pyvex được tích hợp.

2.2.2. Đặc điểm và cấu trúc của Vex-IR

VEX Intermediate Representation (VEX-IR) là một biểu diễn trung gian được thiết kế để tiêu chuẩn hóa và đơn giản hóa việc phân tích mã máy từ nhiều kiến

trúc phần cứng khác nhau. Nó cung cấp một nền tảng nhất quán cho việc phân tích tĩnh và động, tối ưu hóa, và các hoạt động khác liên quan đến mã nhị phân. Dưới đây là những đặc điểm và cấu trúc chính của VEX-IR. Dưới đây là một số đặc điểm của VEX-IR:

- **Độc lập kiến trúc:** VEX-IR có thể biểu diễn mã máy từ nhiều kiến trúc CPU khác nhau như x86, ARM, MIPS, v.v. Điều này giúp các công cụ phân tích dễ dàng làm việc với mã từ các nền tảng khác nhau mà không cần phải xử lý từng kiến trúc cụ thể..
- **Ngôn ngữ cấp thấp:** VEX-IR là ngôn ngữ trung gian cấp thấp, gần với mã máy nhưng dễ hiểu hơn và có cấu trúc rõ ràng hơn. Nó hỗ trợ các lệnh cơ bản như di chuyển dữ liệu, tính toán, và điều khiển luồng.
- **Đơn vị cơ bản là IRSB:** Một IRSB là một chuỗi các lệnh VEX-IR đại diện cho một khối mã gốc liên tục (basic block) từ mã nhị phân. Mỗi IRSB chứa một loạt các lệnh IRStmt và các biểu thức IRExpr.

Bảng sau đề cập đến một số loại biểu thức VEX-IR thường gặp.

Bảng 2.1: *Một số loại biểu thức Vex-IR*

Biểu thức Vex-IR	**Giá trị đã tính**	**Ví dụ**
Constant	Một giá trị hằng số	`0x4:I32`
Read Temp	Giá trị được lưu trữ trong biến tạm của VEX	`RdTmp(t10)`
Get Register	Giá trị được lưu trữ trong một thanh ghi	`GET:I32(16)`
Load Memory	Giá trị được lưu trữ tại một địa chỉ bộ nhớ, với địa chỉ được xác định bởi biểu thức Vex-IR khác	`LDle:I32 / LDbe:I64`
Operation	Kết quả của một phép toán Vex-IR cụ thể, áp dụng cho các đối số Biểu thức Vex-IR khác	`Add32`
If-Then-Else	Nếu một Biểu thức Vex-IR đã cho cho kết quả trả về bằng 0, trả về một biểu thức Vex-IR. Ngược lại, trả về biểu thức Vex-IR khác	`ITE`
Helper Function	VEX sử dụng các hàm trợ giúp của C cho một số phép toán, ví dụ như tính toán các thanh ghi cờ điều kiện . Các hàm này trả về biểu thức Vex-IR	`function_name()`

- **Thao tác (Operations):** Thao tác IR mô tả việc thay đổi Biểu diễn IR. Điều này bao gồm phép toán số nguyên, phép toán số dấu chấm động, phép toán bit,... Áp dụng một thao tác IR vào Biểu thức IR cho kết quả là một Biểu thức IR.

- **Biến tạm thời (Temporary variables):** Vex-IR sử dụng biến tạm thời như các thanh ghi nội bộ: Biểu thức IR được lưu trữ trong các biến tạm thời trước khi sử dụng. Ta có thể truy xuất nội dung của biến tạm thời bằng cách sử dụng Biểu thức IR. Các biến tạm thời này được đánh số từ t0.

- **Câu lệnh (Statements):** Câu lệnh IR mô hình các thay đổi trạng thái của kiến trúc ban đầu, chẳng hạn như việc lưu trữ dữ liệu vào bộ nhớ và ghi vào thanh ghi. Câu lệnh IR sử dụng Biểu diễn IR cho các giá trị mà chúng cần. Ví dụ, một câu lệnh IR lưu trữ dữ liệu vào bộ nhớ sử dụng biểu thức IR cho địa chỉ và một biểu thức IR khác cho nội dung. Một số loại câu lệnh Vex-IR được thể hiện ở bảng 2.2.

Bảng 2.2: *Vi dụ một số loại câu lệnh Vex-IR*

Câu lệnh Vex-IR	**Ý nghĩa**	**Ví dụ**
Write Temp	Đặt biến tạm thời Vex thành giá trị của biểu thức IR đã cho.	`WrTmp(t1) = (IR Expression)`
Put Register	Cập nhật một thanh ghi với giá trị của biểu thức IR đã cho.	`PUT(16) = (IR Expression)`
Store Memory	Cập nhật một vị trí trong bộ nhớ, được cung cấp dưới dạng biểu thức IR, với một giá trị, cũng được cung cấp dưới dạng biểu thức IR	`STle(0x1000) = (IR Expression))`
Exit	Thoát có điều kiện khỏi một basic block, với mục tiêu nhảy được chỉ định bởi biểu thức IR và điều kiện cũng được chỉ định bởi biểu thức IR	`if (condition) goto (Boring) 0x4000A00:I32`

- **Khối (Blocks):** Một Khối IR là một tập hợp các câu lệnh IR, đại diện cho một basic block trong kiến trúc ban đầu (gọi là "IR Super Block" hoặc "IRSB").

Để dễ hình dung hơn về các biểu diễn trong Vex-IR, chúng tôi để một đoạn mã Vex-IR đã được chuyển từ một đoạn bytes code ở bên dưới.

```
1 t12 = GET:I32(offset=20)
2 t13 = Sub32(t12,0x00000001)
3 t2 = Add32(t13,0x00000001)
4 PUT(offset=40) = 0x00000003
5 PUT(offset=44) = t13
6 PUT(offset=48) = 0x00000001
7 PUT(offset=52) = 0x00000000
8 PUT(offset=8) = t2
9 PUT(offset=340) = 0x08048431
```

2.3. Học sâu (Deep Learning)

Học sâu là một lĩnh vực của Trí tuệ nhân tạo (Artificial Intelligence) tập trung vào việc xây dựng và huấn luyện các mô hình máy học sâu để tự động học từ dữ liệu cho trước. Phương pháp này được lấy cảm hứng từ cách hoạt động của não người, trong đó mạng nơ-ron sinh học trong não xử lý thông tin qua việc kết nối các nút nơ-ron.

Mục tiêu chính của Học sâu là xây dựng các mô hình máy tính có khả năng học và hiểu dữ liệu phức tạp để đưa ra dự đoán, phân loại, và giải quyết các bài toán phức tạp. Học sâu có ứng dụng rộng rãi trong nhiều lĩnh vực, bao gồm nhận dạng hình ảnh và video, xử lý ngôn ngữ tự nhiên, nhận dạng giọng nói, phân tích dữ liệu y tế, xe tự hành, và nhiều lĩnh vực khác.

2.3.1. Một số thành phần chính của mạng nơ-ron

Mô hình Mạng Nơ-ron thông thường sẽ có thành phần cơ bản sau:

Lớp đầu vào (Input layer)

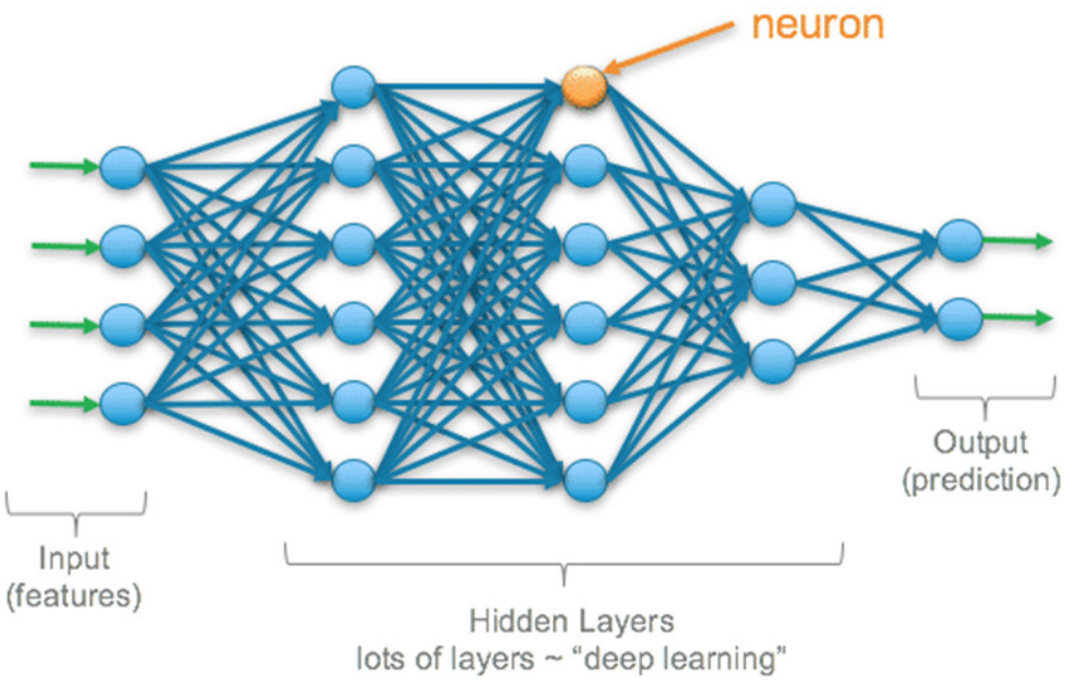
- Lớp đầu vào này nhận dữ liệu đầu vào và truyền chúng vào mạng nơ-ron để xử lý.
- Kích thước của lớp đầu vào phụ thuộc vào đặc điểm của dữ liệu và mục tiêu của bài toán cần giải quyết.

Lớp ẩn (Hidden layers)

- Lớp ẩn là nơi xảy ra quá trình biến đổi dữ liệu và trích xuất đặc trưng.
- Mỗi lớp ẩn sẽ có một số nút nơ-ron và các kết nối giữa chúng để truyền thông tin qua mạng nơ-ron.

Lớp đầu ra (Output layer)

- Lớp này nhận thông tin từ các lớp ẩn và đưa ra kết quả cuối cùng của mô hình. Hình 2.5



Hình 2.5: *Mô hình mạng nơ-ron*

2.3.2. Một số các khái niệm khác

Overfitting

Overfitting là hiện tượng trong mạng nơ-ron khi mô hình quá tinh chỉnh cho tập dữ liệu huấn luyện cụ thể mà nó không mang tổng quát hóa tốt cho dữ liệu mới mà không biết trước. Mạng nơ-ron có thể học các mẫu và quy tắc phức tạp trong tập dữ liệu huấn luyện, đến mức nó ghi nhớ cả các nhiễu và chi tiết không cần thiết. Điều này dẫn đến mô hình quá phức tạp và khả năng tổng quát hoá kém.

Underfitting

Underfitting là hiện tượng trong mạng nơ-ron không được cung cấp đủ thông tin và dữ liệu cho quá trình huấn luyện Trong trường hợp này, mạng nơ-ron chưa đủ khả năng học các mẫu phức tạp và quy tắc trong dữ liệu huấn luyện.

Lan truyền thuận (forward propagation)

Lan truyền thuận là quá trình dữ liệu khi được đưa vào mô hình sẽ đi từ các

lớp đầu vào đến các lớp ẩn rồi đến lớp đầu ra.

Trong quá trình lan truyền thuận, mạng nơ-ron tính toán giá trị đầu ra dự đoán cho mỗi mẫu dữ liệu đầu vào. Quá trình này cho phép mạng nơ-ron học và thực hiện các dự đoán dựa trên các trọng số được điều chỉnh trong quá trình huấn luyện

Lan truyền ngược (back propagation)

Lan truyền ngược là một thuật toán sử dụng để tính toán đạo hàm của hàm mất mát theo các trọng số của mạng nơ-ron, từ đó điều chỉnh các trọng số để tối ưu hóa mạng.

Quá trình lan truyền ngược bắt đầu từ việc tính toán đầu ra của mạng nơ-ron dựa trên dữ liệu huấn luyện và các trọng số hiện tại. Sau đó, đạo hàm của hàm mất mát được tính toán đối với các trọng số của mạng. Đạo hàm này cho biết mức độ ảnh hưởng của từng trọng số đến sự sai khác giữa kết quả dự đoán và giá trị thực tế.

Tiếp theo, đạo hàm được truyền ngược từ lớp đầu ra qua các lớp trước đó của mạng. Tại mỗi nơ-ron, đạo hàm được tính dựa trên đạo hàm của các nơ-ron trong lớp tiếp theo và trọng số kết nối tương ứng. Quá trình này tiếp tục cho đến khi đạo hàm được tính toán cho tất cả các nơ-ron trong mạng.

Sau khi đã tính toán được đạo hàm của hàm mất mát theo các trọng số, các trọng số được điều chỉnh sử dụng một thuật toán tối ưu hóa như gradient descent. Thuật toán gradient descent sẽ cập nhật các trọng số theo hướng giảm đạo hàm, từ đó làm giảm hàm mất mát và cải thiện hiệu suất của mạng.

Hàm kích hoạt (Activation function)

Hàm kích hoạt được áp dụng cho mỗi nút nơ-ron trong các lớp ẩn và lớp đầu ra để tạo ra các đầu ra phi tuyến tính và đưa ra quyết định. Chúng đóng vai trò quan trọng trong việc tạo ra độ linh hoạt và khả năng học của mạng nơ-ron.

- Hàm Sigmoid: Hàm sigmoid ánh xạ một giá trị đầu vào vào khoảng (0, 1). Tức là khi áp dụng hàm này cho một giá trị đầu vào, nó trả về

một giá trị trong khoảng từ 0 đến 1.

- Hàm Tanh: Tương tự như hàm sigmoid, hàm tanh ánh xạ giá trị đầu vào vào khoảng (-1, 1). Nó giúp tăng cường độ dốc của đầu ra so với hàm sigmoid.

- Hàm ReLU (Rectified Linear Unit): Hàm này trả về giá trị đầu vào nếu kết quả tính toán lớn hơn 0 và trả về 0 nếu nó nhỏ hơn hoặc bằng 0. Hàm này giúp mô hình học nhanh hơn và dễ dàng tính toán.

- Hàm Leaky ReLU: Tương tự như ReLU, hàm Leaky ReLU trả về giá trị đầu vào nếu kết quả tính toán lớn hơn 0 và trả về một giá trị nhỏ hơn 0 nhân với một hệ số nhỏ nếu nó nhỏ hơn hoặc bằng 0. Điều này giúp khắc phục vấn đề "neuron chết" trong hàm ReLU.

- Hàm Softmax: Được sử dụng trong lớp cuối cùng của mạng nơ-ron để ánh xạ đầu ra thành một phân phối xác suất. Hàm softmax tính toán đầu ra sao cho tổng các giá trị đầu ra bằng 1, giúp giải quyết bài toán phân loại đa lớp.

Hàm mất mát (loss function)

Hàm mất mát là một hàm số dùng để đo lường sự khác nhau giữa kết quả dự đoán của mô hình và kết quả thực tế của tập dữ liệu. Hàm này có nhiệm vụ đánh khả năng dự đoán của mô hình và từ đó cung cấp 1 phản hồi để thay đổi các trọng số của mô hình..

Thông thường trong huấn luyên, một trong các mục tiêu là tìm ra các tham số của mô hình sao cho hàm mất mát đạt giá trị nhỏ nhất

Có nhiều loại hàm mất mát khác nhau, dưới đây là một số hàm mất mát phổ biến:

- Hàm Cross-Entropy: Hàm này đo lường độ chính xác của dự đoán so với nhãn thực tế và cố gắng tối thiểu hóa khoảng cách giữa chúng. Thường được sử dụng trong các bài toán phân loại.

- Hàm Categorical Cross-Entropy: Được sử dụng trong bài toán phân loại đa lớp, hàm này đo lường chênh lệch giữa phân phối xác suất dự đoán và nhãn thực tế.

Batch size và Epoch

Batch size là số lượng mẫu dữ liệu được truyền qua mạng nơ-ron trong mỗi lần cập nhật trọng số. Thay vì truyền từng mẫu dữ liệu một, chúng ta chia dữ liệu huấn luyện thành các batch có kích thước nhất định. Batch size ảnh hưởng đến hiệu suất và tốc độ huấn luyện của mô hình.

Epoch là một vòng lặp hoàn chỉnh mà mô hình học được toàn bộ tập dữ liệu huấn luyện đã chuẩn bị. Số lượng epoch tức là số lần mà mô hình được huấn luyện trên toàn bộ dữ liệu

Ma trận nhầm lẫn (confusion matrix)

Được sử dụng để mô tả sự tương quan giữa các dự đoán của mô hình và các nhãn thực tế của dữ liệu.

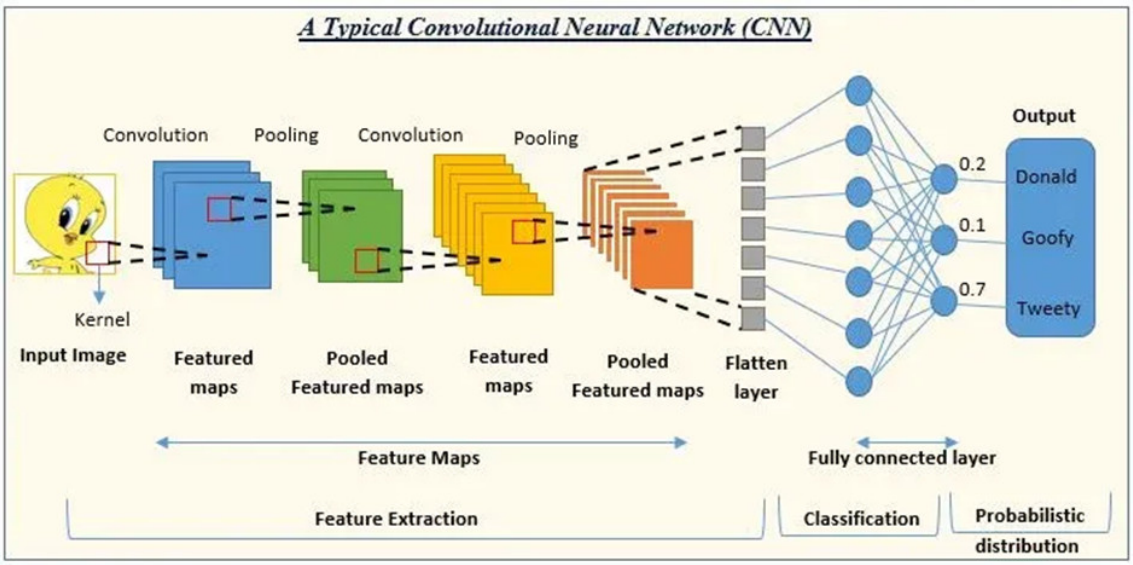
Confusion matrix có dạng một bảng vuông chứa các giá trị đại diện cho các kết quả phân loại. Bảng được chia thành các ô, trong đó hàng thể hiện nhãn thực tế và cột thể hiện nhãn dự đoán. Nó giúp chúng ta có cái nhìn tổng quan về khả năng phân loại của mô hình, từ đó tính toán các độ đo đánh giá hiệu suất như accuracy, precision, recall, F1-score.

2.3.3. Một số kiến trúc mạng nơ-ron phổ biến trong lĩnh vực học sâu

Mạng nơ-ron Tích chập (Convolutional Neural Network - CNN)

- Kiến trúc CNN được sử dụng chủ yếu trong xử lý ảnh.
- Bao gồm các lớp tích chập (convolutional) để trích xuất đặc trưng từ ảnh và các lớp gộp (pooling layers) để giảm kích thước của đặc trưng.

- Cuối cùng, các lớp kết nối đầy đủ (fully connected layers) được sử dụng để phân loại và dự đoán.



Hình 2.6: *Ví dụ về một mô hình CNN để phân loại hình ảnh.*

Mạng nơ-ron Hồi quy (Recurrent Neural Network - RNN)

- Kiến trúc RNN được sử dụng trong xử lý dữ liệu tuần tự như ngôn ngữ tự nhiên và dữ liệu chuỗi thời gian.
- Có khả năng lưu trữ thông tin trạng thái trước đó và áp dụng lặp lại cho dữ liệu mới.
- Mạng LSTM (Long Short-Term Memory) và GRU (Gated Recurrent Unit) là các biến thể phổ biến của RNN giúp khắc phục vấn đề mất thông tin dài hạn.

Mạng Long Short-Term Memory (LSTM)

- Mạng LSTM (Long Short-Term Memory) là một dạng của mạng nơ-ron hồi quy (RNN) đặc biệt được thiết kế để giải quyết vấn đề của hiện tượng biến mất đối với các mô hình RNN truyền thống.

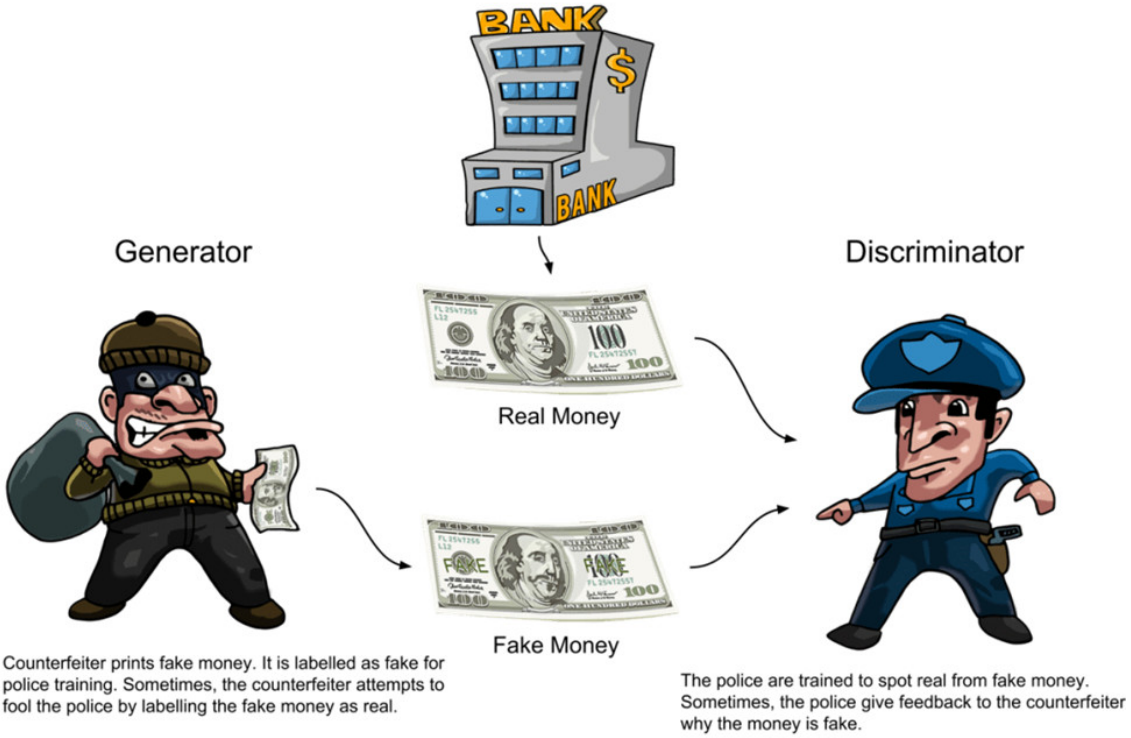
- Mạng LSTM bao gồm một số ô trạng thái (cell state) và các cổng (gates) để kiểm soát và xử lý thông tin.
- LSTM giúp giải quyết vấn đề biến mất gradient trong mô hình RNN thông thường, cho phép nó duy trì và sử dụng thông tin từ xa trong chuỗi dữ liệu, làm cho nó rất mạnh mẽ trong việc xử lý chuỗi dữ liệu có độ dài lớn.

Mạng nơ ron hồi quy Siamese Neural Network

- Siamese Neural Network là một kiến trúc mạng nơ-ron được thiết kế để học biểu diễn tương đồng giữa hai đối tượng
- Mạng Siamese Neural Network có thể học được biểu diễn tương đồng hiệu quả, đặc biệt là trong các nhiệm vụ liên quan đến so sánh và phân loại.

Mạng sinh đối nghịch (Generative Adversarial Network - GAN)

- Kiến trúc GAN bao gồm hai mạng nơ-ron: mạng sinh (generator) và mạng phân biệt (discriminator).
- Mạng sinh cố gắng tạo ra dữ liệu mới tương tự với dữ liệu huấn luyện, trong khi mạng phân biệt cố gắng phân biệt giữa dữ liệu thực và dữ liệu giả tạo.
- Quá trình huấn luyện GAN diễn ra thông qua việc tối ưu hóa đồng thời cả hai mạng, đạt được sự cân bằng giữa mạng sinh và mạng phân biệt.

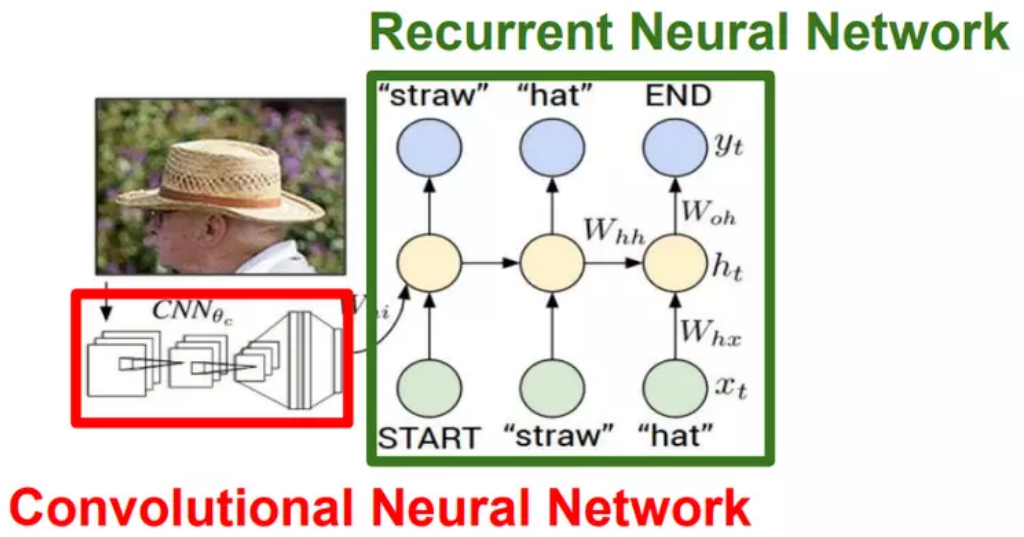


Hình 2.7: *Ví dụ về một mô hình GAN trong phát hiện ảnh.*

Kết hợp các kiến trúc

Ngoài việc sử dụng các kiến trúc riêng lẻ, chúng ta có thể kết hợp các kiến trúc lại với nhau để tận dụng các ưu điểm của mỗi kiến trúc, từ đó đem lại hiệu quả tốt hơn.

Ví dụ trong việc mô tả hình ảnh, có thể kết hợp CNN và RNN. CNN sẽ được sử dụng để phát hiện và trích xuất các tính chất có trong ảnh sau đó RNN sẽ sinh ra các câu có mô tả có ý nghĩa cho bức ảnh.



Hình 2.8: *Ví dụ về mô hình kết hợp CNN và RNN.*

2.3.4. Phương pháp học trong học sâu

Chuẩn bị dữ liệu

- Thu thập và xử lý dữ liệu: Thu thập dữ liệu mã nhị phân từ các nguồn khác nhau và tiền xử lý dữ liệu để chuẩn bị cho quá trình huấn luyện.
- Chia tập dữ liệu: Tách tập dữ liệu thành tập huấn luyện và tập kiểm tra. Tách tập dữ liệu vừa chuẩn bị thành 2 tập nhỏ hơn, 1 tập dùng để huấn luyện mô hình và tập còn lại đùng để đánh giá mô hình.

Lựa chọn mô hình học sâu

- Xác định kiến trúc mô hình: Chọn kiến trúc mô hình học sâu để phù phù hợp với bài toán cần giải quyết như CNN hoặc RNN
- Thiết kế lớp mạng: Xác định số lượng và kích thước các lớp nơ-ron trong mô hình, bao gồm các lớp đầu vào,lớp ẩn và lớp đầu ra.
- Lựa chọn hàm kích hoạt: Chọn hàm kích hoạt phù hợp cho mỗi lớp nơ-ron trong mô hình, như ReLU hoặc Sigmoid.

Quá trình huấn luyện

- Xác định hàm mất mát: Tìm hiểu và chọn hàm mất mát phù hợp với mô hình để đo lường sai số giữa các kết quả dự đoán của mô hình và giá trị thực tế.
- Tối ưu hóa mô hình: Điều chỉnh các trọng số trong mô hình, điều chỉnh lại các lớp,...
- Lặp lại quá trình huấn luyện: Thực hiện quá trình lan truyền thuận và lan truyền ngược trên tập huấn luyện nhiều lần để cải thiện mô hình theo từng vòng lặp.

2.4. Các ứng dụng của việc sử dụng phương pháp phát hiện sự tương đồng trong mã nhị phân

Phát hiện sự tương đồng trong mã nhị phân là một lĩnh vực quan trọng trong việc phân tích và bảo mật phần mềm, một số ví dụ như sau:

- Mã độc: Thông thường mã độc được tạo ra bằng cách chỉnh sửa mã nguồn hiện có. Bằng phát hiện sự tương đồng trong mã nhị phân, chúng ta có thể xác định được nhanh chóng đoạn mã độc cần phân tích. Không những thế ta còn có thể xác định nhanh chóng đâu là mã độc bằng cách so sánh với các mẫu mã độc đã biết,...
- Kiểm tra đạo văn phần mềm: Phát hiện sự tương đồng trong mã nhị phân có thể xác định sự sao chép hoặc đạo văn giữa các đoạn mã nguồn trong các phần mềm khác nhau chp dù chúng ta không có mã nguồn của chương trình cần kiểm tra.
- Phục hồi mã nguồn: Phát hiện sự tương đồng trong mã nhị phân có thể hỗ trợ quá trình phân tích và phục hồi mã nguồn bằng cách tìm

kiếm các đoạn mã tương đồng rồi đối chiếu với mã nguồn gốc để phục hồi cấu trúc, hàm, và logic của chương trình.

- Kiểm thử phần mềm: Phương pháp phát hiện sự tương đồng có thể được sử dụng để xác định các hàm trong chương trình có thể chứa các lỗ hổng đã biết hoặc các kịch bả đã được kiểm thử trước đó, từ đó làm tăng tốc quá trình kiểm thử.

Tất cả các ứng dụng trên nghe to lớn nhưng đều bắt nguồn từ công việc "So sánh độ tương đồng giữa 2 chức năng nhị phân". Một ví dụ minh họa cho việc áp dụng phương pháp này là trong kiểm thử phần mềm. Khi chúng tôi muốn tìm kiếm một chức năng bị lỗi trong một chương trình đã biết, chúng tôi cần so sánh độ tương đồng giữa các hàm trong chương trình với một hàm đã xác định là bị lỗi. Nếu một chức năng có độ tương đồng cao với hàm lỗi, khả năng cao là chức năng đó cũng chứa lỗi tương tự như hàm ban đầu.

Điều này đặt nền tảng cho nghiên cứu của chúng tôi về phát hiện sự tương đồng trong mã nhị phân cùng với phương pháp so sánh độ tương đồng giữa hai chức năng nhị phân. Cụ thể hơn chúng tôi sẽ trình bày ở chương 3 và 4.

CHƯƠNG 3. TỔNG QUAN VỀ BINSHOO

Trong chương này, chúng tôi sẽ trình bày về công cụ phát hiện sự tương đồng trong mã nhị phân có tên là Binshoo. Đầu tiên chúng tôi sẽ nói về định nghĩa phát hiện sự tương đồng trong mã nhị phân, nêu tổng quan về Binshoo, đồng thời sau đó đến chi tiết từng phần của phương pháp, cuối cùng là phương pháp đánh giá.

3.1. Định nghĩa vấn đề

Về cơ bản, phát hiện sự tương đồng trong mã nhị phân là việc tính toán độ tương đồng giữa 2 mã nhị phân. Nó có thể chia ra thành 3 loại như trong [5] như sau:

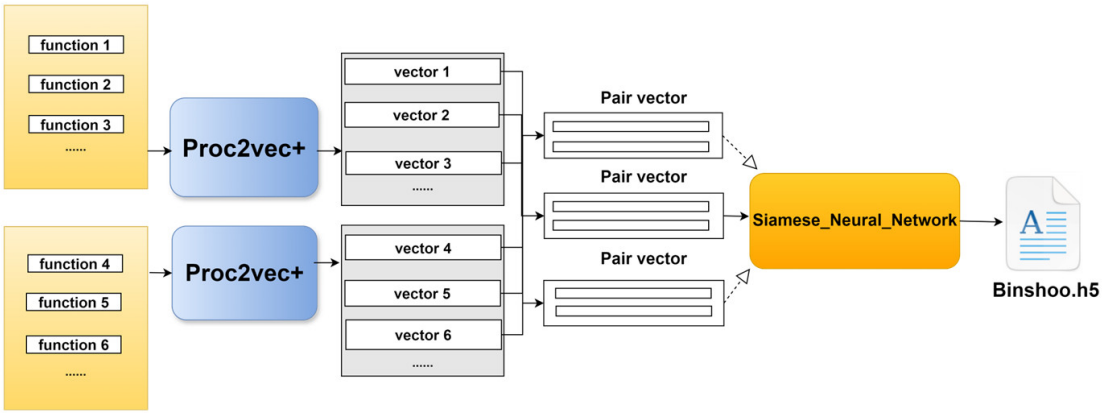
- One-to-one: Tính toán điểm số tương đồng giữa 2 chức năng bất kì từ đó xác định 2 chức năng có tương đồng hay không.
- One-to-many: Một nhóm các chức năng mục tiêu (pool) sẽ được sắp xếp dựa trên độ tương tự với chức năng nguồn ban đầu theo thứ tự giảm dần. Nhiệm vụ phương pháp của chúng ta sẽ tập trung vào vấn đề tìm ra hàm tương đồng nhất với hàm ban đầu và xếp vị trí của hàm đó lên đầu danh sách.
- Many-to-many: Một nhóm các chức năng sẽ được chia thành các nhóm nhỏ hơn dựa trên độ tương đồng của chúng với nhau.

Nghiên cứu này của chúng tôi tập trung vào 2 loại chính là One-to-one và One-to-many, vì về bản chất cả 3 vấn đề nay khá tương tự nhau. Trong One-to-many chúng tôi có thể giảm số lượng của các hàm mục tiêu thành 1 để trở thành

One-to-one và cũng có thể mở rộng các bài toán One-to-many thành Many-to-many bằng cách lấy từng hàm trong nhóm làm hàm nguồn và giải nhiều bài toán One-to-many. Thế nên dù mô hình của chúng tôi tuy chỉ được học về cách phân loại 2 chức năng nhị phân nhưng nó vẫn có khả năng giải quyết vấn đề one-to-many.

3.2. Binshoo - Phương pháp phát hiện sự tương đồng trong mã nhị phân dựa trên học sâu

Chúng tôi sẽ giới thiệu Binshoo, mô hình phát hiện sự tương đồng mã nhị phân cùng với phương pháp Proc2vec+, một phương pháp chuyển đổi từ bytes code của các chức năng nhị phân sang dạng vector được giới thiệu ở Zeek để có để đưa vào mô hình học máy. Mô hình tổng quát của chúng tôi được khái quát ở hình 3.1.



Hình 3.1: *Mô hình tổng quan của Binshoo.*

Cụ thể hơn đối với các tập tin nhị phân được biên dịch với các tình huống khác nhau, chúng tôi dùng IDA để trích xuất các bytes code của các chức năng này. Tiếp theo, chúng tôi tiến hành trích xuất các bytes code trong tập tin nhị phân để đưa vào phương pháp của chúng tôi – Proc2vec+ (Phương pháp chuyển đổi bytes code thành vector mà chúng tôi sẽ nói cụ thể hơn ở phần 3.2.2). Đối với các vector đã được trích xuất thông qua Proc2vec+, chúng tôi tiến hành tạo

các cặp vector và gán nhãn cho chúng với các cặp vector được biên dịch từ 1 mã nguồn (Tương đồng cặp vector đó được gán nhãn là 1, ngược lại đối với cặp vector được biên dịch từ 2 mã nguồn khác nhau (Không tương đồng) chúng sẽ được gán nhãn là 0. Cuối cùng các cặp vector này được đưa vào mô hình học sâu của chúng tôi (Sẽ bàn kĩ hơn ở phần 3.2.3) để tiến hành học tập và đánh giá.

3.2.1. Trích xuất các function

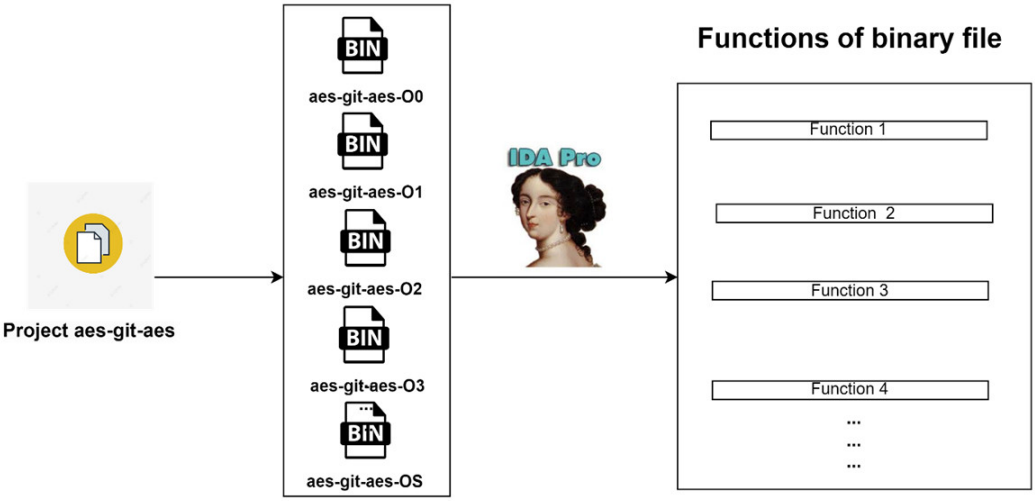
Trong nghiên cứu này chúng tôi sử dụng một dự án trong bộ dữ liệu BinaryCorp, sau đó biên dịch nó với các lựa chọn tối ưu hóa khác nhau để tạo thành các file nhị phân tương ứng. Thông tin về số lượng các dự án được các tác giả trình bày như sau:

Bảng 3.1: *Bảng trích xuất chức năng của bộ dữ liệu BinaryCorp.*

Dataset	**Project**	**Binaries**	**Functions**
BinaryCorp-3M Train	1,612	8,357	3,126,367
BinaryCorp-3M Test	354	1,908	444,574
BinaryCorp-26M Train	7,845	38,455	21,085,338
BinaryCorp-26M Test	1,974	9,675	4,791,673

Bởi vì đối tượng nghiên cứu là tập tin nhị phân, chúng tôi cần sự hổ trợ của một decompiler để thực hiện công việc này. Trong chương 2, chúng tôi đã giới thiệu IDA, một công cụ decompiler phổ biến và cung cấp nhiều tiện ích hữu ích. Vì vậy, để hỗ trợ quá trình trích xuất các chức năng, chúng tôi sẽ sử dụng IDA.

Bằng sự hổ trợ của các IDA API, việc tự động trích xuất vị trí của các chức năng trong tập tin nhị phân sẽ trở nên dễ dàng. Quá trình này giúp chúng tôi xác định địa chỉ bắt đầu và kết thúc của mỗi chức năng trong chương trình. Tiếp theo, từ các địa chỉ này, chúng tôi có thể đọc byte code của từng chức năng tương ứng. Chúng tôi biểu diễn quá trình này dưới hình 3.2.



Hình 3.2: *Quá trình trích xuất các chức năng*

3.2.2. Chuyển đổi các chức năng nhị phân sang vector để đưa vào mô hình học máy

Về tổng quan phương pháp này chúng tôi chuyển đổi các bytes code sang dạng biểu diễn trung gian VEX-IR, sau đó tiếp tục phân tách các chức năng thành các phân đoạn nhỏ hơn từ basic blocks thành strands và tiến hành tối ưu hóa từng phân đoạn, cuối cùng chúng tôi dùng hàm băm MD5 để nhúng các phân đoạn này thành vector biểu diễn cuối cùng.

3.2.2.1. Trích xuất Strand

Như đã giới thiệu ở chương 2, chúng tôi xem Strand như là một đơn vị nhỏ để trích xuất ngữ nghĩa của các chức năng nhị phân. Về lý do vì sao lại thực hiện quá trình này, chúng tôi sẽ giải thích kĩ hơn ở phần sau.

Thuật toán 3.1 biểu thị mã giả cho quy trình trích xuất Strand từ 1 tập hợp các câu lệnh. Các hàm Ref, Def trả về tập hợp các biến được tham chiếu hoặc xác định bởi một lệnh. Nó giống với những gì mà David đã giới thiệu ở [1].

Thuật toán 3.1 Thuật toán trích xuất Strands

Input: bb - danh sách các câu lệnh

Output: S - danh sách các Strand trả về bb

$uncovered \leftarrow [0, 1, ..., |bb| - 1]; S \leftarrow []$;
while $|uncovered| - 1] > 0$ **do**
  $last \leftarrow uncovered.pop\_last()$;
  $strand \leftarrow [bb[last]]$;
  $used \leftarrow Ref(bb[last])$;
  for $i \leftarrow (last - 1)..0$ **do**
    $needed \leftarrow Def(bb[i]) \cap used$;
    if needed $\neq \emptyset$ **then**
      $strand.append(bb[i])$;
      $used \cup Ref(bb[i])$;
      $uncovered.remove(i)$;
    end if
  end for
  $S.append(strand)$;
end while
return S

3.2.2.2. Proc2vec+: Phương pháp chuyển đổi bytecode thành vector.

Ở phần này chúng tôi sẽ sử dụng phương pháp Proc2vec+ để chuyển đổi các bytes code của các chức năng nhị phân thành các vector dựa trên phương pháp Proc2vec đã được giới thiệu ở Zeek và có cải tiến sử dụng thêm hai kỹ thuật tối ưu hóa mã là Inlining và Constant Folding. Phương pháp này gồm các bước được thể hiện trong hình 3.3 như sau:

(1) Từ các bytes code ban đầu, chúng tôi chuyển đổi chúng thành dạng biểu diễn trung gian. Chúng tôi chọn Vex-IR làm biểu diễn trung gian trong

phương pháp này. Từ những điểm mạnh của Vex-IR mà chúng tôi đã nếu ở chương 2, giai đoạn này chúng tôi chọn loại biểu diễn trung gian mà chúng tôi sử dụng là Vex-IR.

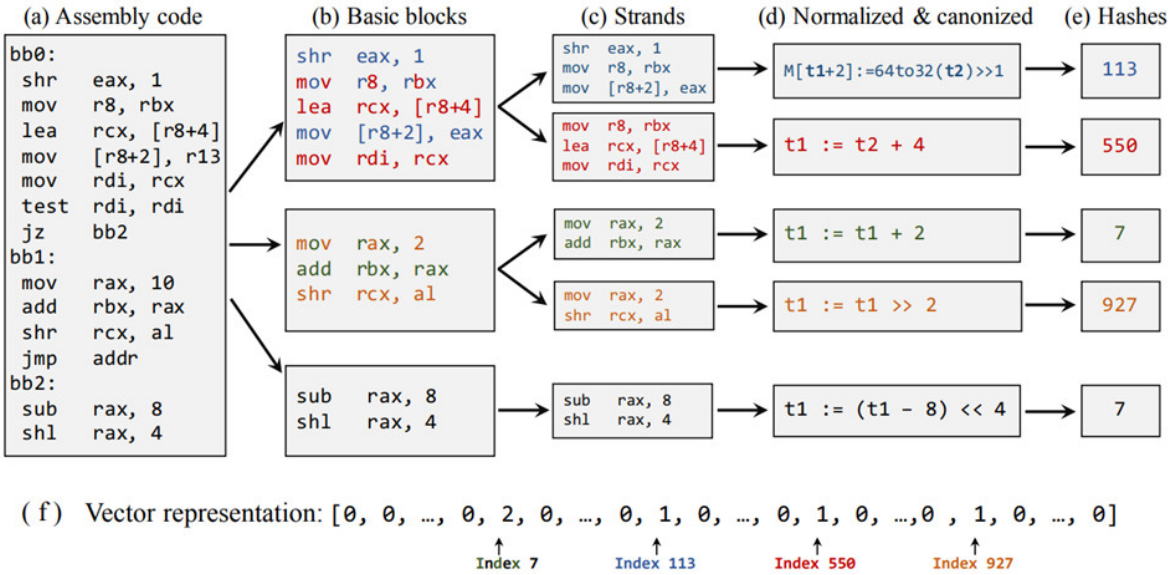
(2) Từ các đoạn mã biểu diễn trung gian của chức năng, chúng tôi gom lại tập hợp tất cả các câu lệnh

(3) Từ tập hợp các câu lệnh có được, chúng tôi áp dụng thuật toán extract strand để trích xuất thành các strand.

(4) Ở mỗi strand, chúng tôi tiến hành tối ưu hóa, giai đoạn này gồm 4 quá trình:

- Copy propagation và Dead code elimination
- Normalize
- Inlining
- Constant Folding

(5) Cuối cùng chúng tôi sử dụng MD5-b bit trên mỗi strand để chuyển đổi chúng thành 1 số nguyên có giá trị từ 0 -> 2^b -1 (Ở nghiên cứu này chúng tôi chọn b với giá trị bằng 10 do độ dài vector chúng tôi sử dụng là 1024 chiều). Từ các số nguyên có được, chúng tôi tiến hành tạo vector bằng cách duyệt qua danh sách các số nguyên có được, số nguyên mang giá trị thì vị trí tại của vector tăng giá trị thêm 1. Cuối cùng, đối với mỗi chức năng, chúng tối trích xuất được 1 vector tương ứng.



Hình 3.3: *Proc2vec+, phương pháp chuyển đổi bytes code thành vector dựa trên Zeek có cải tiến*

Chuyển đổi bytes code sang Vex-IR tạo ra một ngôn ngữ chung để thể hiện ngữ nghĩa các chức năng trong mã nhị phân. Việc này rất hữu ích trong các tình huống biên dịch khác nhau, bao gồm khác nhau về các option tối ưu hóa, phiên bản của trình biên dịch, và kiến trúc máy tính mục tiêu. Khi chuyển đổi từ bytes code sang Vex-IR, chúng ta có thể đạt được sự đồng nhất trong cách biểu diễn ngữ nghĩa của các chức năng do ngữ nghĩa của một chức năng trong mã nhị phân có thể thay đổi một cách đáng kể khi chúng ta chuyển đổi sang assembly trực tiếp trong khi để nguyên dạng bytes code như ban đầu không thể thể hiện được ngữ ngĩa.

Việc chia nhỏ chức năng thành các strand trong mô hình phát hiện sự tương đồng trong mã nhị phân được thực hiện với mục đích giới hạn khả năng hiểu sai ngữ nghĩa khi chuyển đổi từ bytes code thành Vex-IR. Thông thường đơn vị mà các công cụ phân tích mã nhị phân sử dụng là các basic block, tuy nhiên chúng tôi đánh giá basic block là đơn vị chưa đủ nhỏ để hiểu được ngữ nghĩa một cách chi tiết. Bằng cách này, ta tạo ra các đơn vị nhỏ hơn để so sánh, từ đó giảm thiểu sự ảnh hưởng của những khác biệt nhỏ đối với giá trị MD5 được tính toán. Bởu nếu ta sử dụng các basic block để tính toán, một thay đổi

nhỏ trong basic block sẽ ảnh hưởng đến việc hiểu ngữ nghĩa của toàn bộ basic block đó. Nếu như 2 chức năng tương đồng nhau, qua quá trình tối ưu 2 chức năng này chỉ khác nhau một câu lệnh, nếu ta sử dụng cả basic block để tính toán MD5 cả ngữ nghĩa của basic block đó sẽ khác nhau. Thay vì vậy nếu ta sử dụng Strand việc phân bố ngữ nghĩa sẽ đa dạng hơn, ngữ nghĩa chỉ khác nhau ở một Strand nhỏ có chứa câu lệnh đó. Bằng cách này, chúng ta tạo ra một biểu diễn chính xác hơn về sự tương đồng giữa các chương trình mã nhị phân, hạn chế tối đa ảnh hưởng của việc hiểu sai ngữ nghĩa.

Copy propagation là một kỹ thuật tối ưu hóa mã nhị phân nhằm loại bỏ sự sao chép không cần thiết của giá trị. Trong quá trình chuyển đổi từ bytes code sang Vex-IR sẽ xảy ra nhiều trường hợp mà một biến được gán một giá trị mới trong một điểm của Vex-IR và sau đó được sử dụng tại các điểm khác. Copy propagation thực hiện việc tìm và thay thế tất cả các biến dư thừa. Tuy nhiên kỹ thuật này cần một số điều kiện để được áp dụng như sau: Câu lệnh phải có độ dài nhỏ hơn 10 ký tự. Câu lệnh phải chứa đúng 2 biến. Biến đầu tiên phải là biến được định nghĩa (def). Đây là biến mà câu lệnh sẽ gán giá trị hoặc thay đổi. Biến thứ hai phải là biến tham chiếu (ref). Đây là biến mà giá trị của nó được sử dụng trong câu lệnh. Nếu tất cả các điều kiện này đều thỏa mãn, câu lệnh được coi là đủ điều kiện để thực hiện copy propagation. Ngược lại, nếu bất kỳ điều kiện nào không thỏa mãn, câu lệnh sẽ không được thực hiện copy propagation.

Dead code elimination là quá trình loại bỏ các câu lệnh không có tác động đến ngữ nghĩa và kết quả cuối cùng của chức năng. Điều này giúp giảm dung lượng mã và tăng tốc độ thực thi.

Việc kết hợp cả hai quá trình Copy propagation và Dead code elimination trong bytes2vec mang lại nhiều lợi ích như giảm nhiễu và tăng độ chính xác, tăng tốc độ, hiệu xuất, và cuối cùng là tăng khả năng hiểu và phân tích mã.

Normalize Đây là quá trình tái đặt tên các các biến đã được sinh ra bởi Vex-IR. Chúng tôi duyệt qua mã nhị phân và theo dõi sự xuất hiện của các biến.

Khi chúng ta gặp một biến mới, chúng ta đặt tên cho nó theo thứ tự xuất hiện.

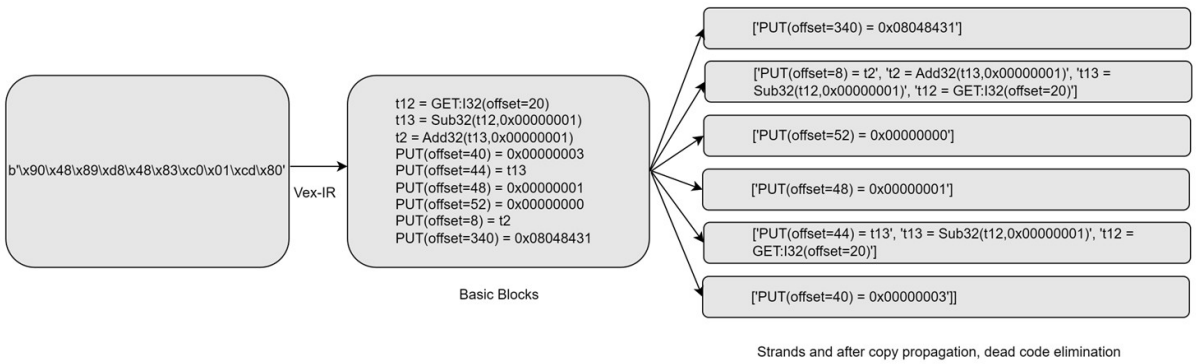
Inlining liên quan đến việc thay thế một cuộc gọi hàm bằng nội dung thực tế của hàm đó trong mã nguồn chương trình. Thực hiện inlining có thể tạo ra mã máy hiệu quả hơn và giảm độ phức tạp của mã nguồn. Một số lợi ích của Inlining bao gồm: Giảm chi phí cuộc gọi hàm, tối ưu hóa thanh ghi.

Constant Folding Thực hiện các phép toán số học trên các hằng số ngay trong quá trình biên dịch thay vì chờ đến thời điểm thực thi. Khi một biểu thức chứa hằng số được đánh giá ngay từ lúc biên dịch, kết quả của biểu thức có thể được tính toán trước khi chương trình chạy. Điều này giúp giảm thời gian thực thi và tăng hiệu xuất tổng thể.

Nếu không có quá trình này, tên các biến sẽ xuất hiện một các không có trật tự trên Vex-IR, khiến cho việc xác định ngữ nghĩa cho 2 chức năng tương tự nhau gặp khó khăn, và giảm hiệu suất của phương pháp.

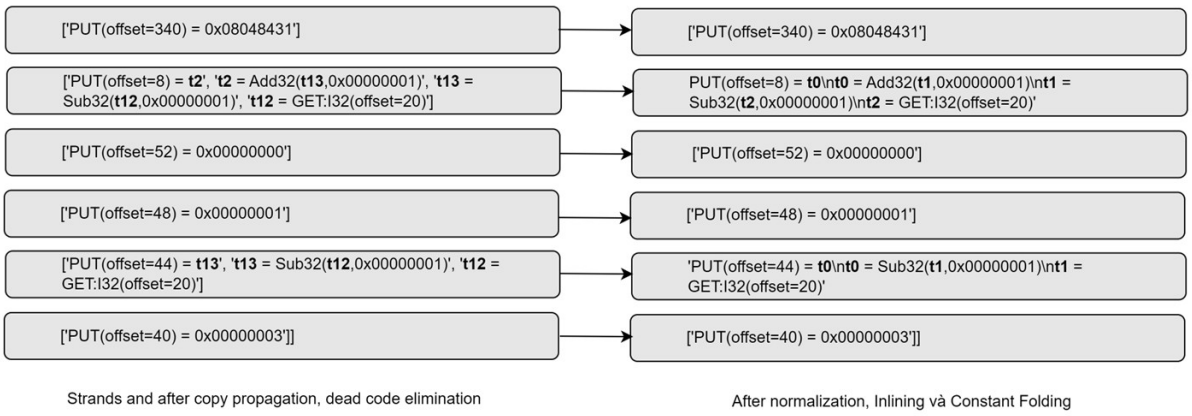
Các công việc tối ưu hóa trên là một phần rất quan trọng trong phương pháp bytes2vec của chúng tôi. Sau khi áp dụng các công việc này, những phần dư thừa không phù hợp của Vex-IR sinh ra sẽ bị loại bỏ, từ đó giúp dễ dàng trích xuất ngữ nghĩa hơn và tăng hiệu quả của phương pháp.

Để dễ hình dung các quá trình trong Proc2vec+ trên, chúng tôi sẽ lấy một ví dụ chuyển đổi từ một hàm ban đầu thành biểu diễn vector như sau:



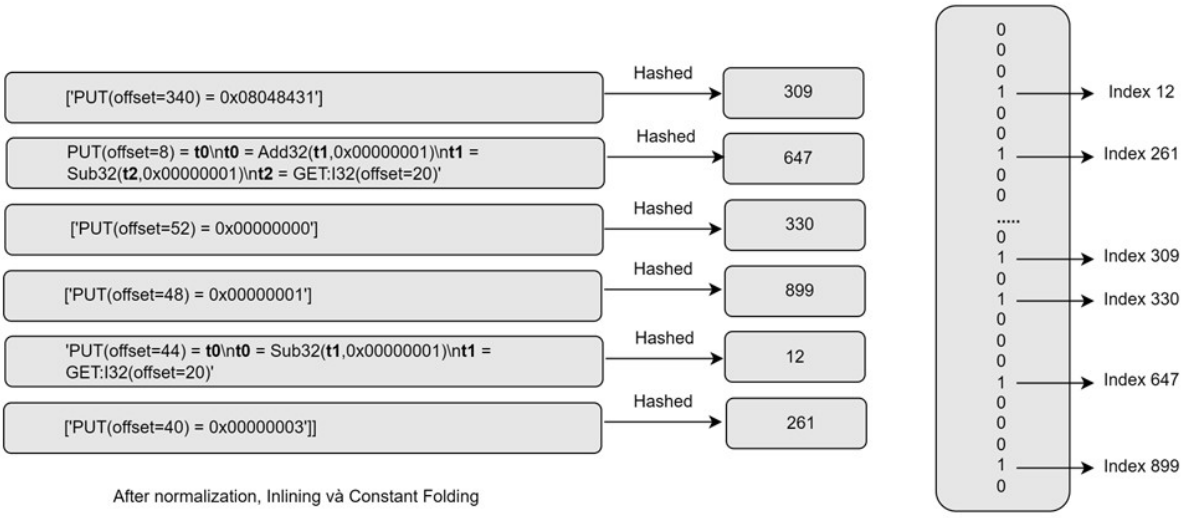
Hình 3.4: *Hình ảnh Function ban đầu sau khi kết thúc quá trình Dead code elimination.*

Raw bytes ban đầu được chuyển thành basic blocks thông qua biểu diễn trung gian VEX-IR. Basic block đó được áp dụng thuật toán trích xuất strands mà chúng tôi đã giới thiệu trước đó để tạo thành các Strands con. Strans này được giữ nguyên sau hai quá trình Copy Coporation và Dead Code Elimination. Strands không giữ nguyên do không đủ các điều kiện để thực hiện Copy Coporation như đã trình bày trước đó. Thứ hai strands giữ nguyên Dead code Elimination vì các strands được trích xuất ra ngắn và đã đủ nghĩa do đó không thực hiện rút gọn thêm.



Hình 3.5: *Định dạng strands sau khi kết thúc Constant Folding.*

Tiếp đến là hình ảnh Strands sau khi kết thúc Constant Folding. Ở đây sau khi thực hiện quá trình Nomalization thì các biến mặc định của VEX-IR đã được thay thế bằng các biến đơn giản hơn. Cụ thể đổi tên các biến tạm thời như t2, t13, và t12 thành t0, t1 và t2 theo thứ tự tuần tự và chuẩn hóa. Strands không đổi sau hai kỹ thuật còn lại vì Inlining chỉ thay thế các cuộc gọi hàm bằng giá trị thực tế của nó trong khi Strands của chúng ta không sử dụng hàm Call.



Hình 3.6: *Quá trình chuyển đổi từ Strands thành vector đầu ra.*

Sau khi thực hiện các quá trình tối ưu hóa xong lúc này mỗi strand sẽ thực hiện hiện hasher bằng hàm băm MD5 để tính toán các giá trị Hasher. Vector ban đầu được khởi tạo full 0 với 1024 chiều sẽ được duyệt qua và sẽ tăng lên một giá trị tại vị trí có giá trị bằng giá trị của strands được băm ra.

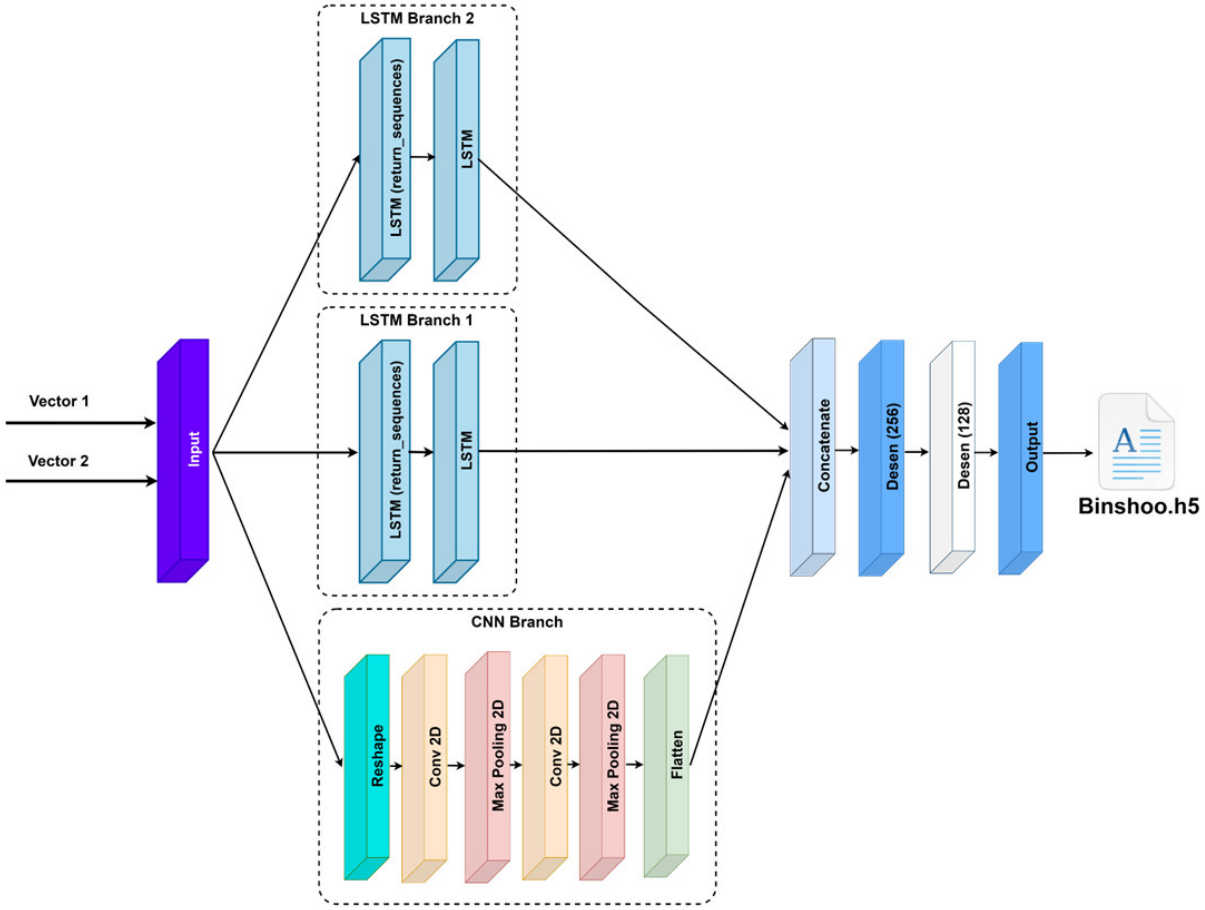
3.2.3. Lựa chọn mô hình học máy

Trong nghiên cứu của chúng tôi, xử lý hình ảnh bằng học sâu CNN đã cho thấy hiệu quả cao trong nhiều nghiên cứu. Do đó, chúng tôi cũng áp dụng kiến trúc CNN vào việc phát hiện sự tương đồng trong mã nhị phân. Để nâng cao hiệu quả, chúng tôi kết hợp CNN với LSTM trong mô hình Siamese Neural Network.

Mô hình của chúng tôi sử dụng một số lớp và kỹ thuật từ lĩnh vực xử lý hình ảnh và chuỗi thời gian. Đặc biệt, chúng tôi sử dụng các lớp Conv2D để trích xuất đặc trưng từ các vector đầu vào và lớp MaxPooling2D để giảm kích thước đặc trưng này. Bằng cách áp dụng các lớp này, chúng tôi có thể tận dụng các thông tin quan trọng và mối quan hệ không gian trong dữ liệu. Để làm cho dữ liệu đầu vào phù hợp với mô hình, chúng tôi sử dụng lớp Reshape để chuyển đổi đầu vào từ định dạng (2, 1024) thành định dạng (2, 1024, 1), cho phép mô hình

xem xét mã nhị phân như các hình ảnh với hai kênh và kích thước 1024x1. Lớp Conv2D với 64 bộ lọc và kích thước kernel (3, 1) được áp dụng để trích xuất đặc trưng từ dữ liệu, sau đó là lớp MaxPooling2D để giảm kích thước. Tiếp theo, một lớp Conv2D khác với 128 bộ lọc và kích thước kernel (3, 1) tiếp tục trích xuất đặc trưng, và lại một lần nữa, lớp MaxPooling2D được sử dụng để giảm kích thước.

Mô hình cũng áp dụng hai nhánh LSTM song song, mỗi nhánh có 100 đơn vị, để xử lý dữ liệu tuần tự từ đầu vào. Các nhánh LSTM này giúp mô hình nắm bắt các phụ thuộc dài hạn trong mã nhị phân. Sau đó, đầu ra từ các nhánh CNN và LSTM được kết hợp thông qua lớp concatenate, tạo thành một vectơ đặc trưng kết hợp. Vectơ này sau đó được đưa qua các lớp Dense với 256 và 128 đơn vị để học các biểu diễn đặc trưng sâu hơn. Cuối cùng, lớp đầu ra Dense với 2 đơn vị và hàm kích hoạt softmax được sử dụng để đưa ra dự đoán xác suất.



Hình 3.7: *Mô hình Siamese Neural Network của Binshoo.*

Việc lựa chọn mô hình Siamese Neural Network kết hợp CNN và LSTM cho phép chúng tôi tận dụng lợi thế của cả hai phương pháp, giúp mô hình học các đặc trưng không gian và tuần tự từ dữ liệu mã nhị phân một cách hiệu quả, từ đó cải thiện độ chính xác và khả năng phát hiện sự tương đồng trong mã nhị phân.

Ngoài ra trong mô hình của chúng tôi có sử dụng hai nhánh LSTM song song với mục đích cho hai nhánh này học dữ liệu một cách độc lập với nhau do đó tuy cùng là LSTM nhưng dữ liệu học được không hoàn toàn giống nhau. Khi chúng được kết hợp lại ở lớp concatenate mô hình có thể tạo ra một biểu diễn tổng thể của dữ liệu phong phú hơn. Những đặc trưng mà mỗi nhánh học được có thể bổ sung và bổ sung lẫn nhau, tạo nên một biểu diễn mạnh mẽ hơn về dữ liệu đầu vào... Thêm vào đó việc sử dụng hai nhánh LSTM độc lập sẽ giảm

thiểu được tình trạng overfitting. Hiện tượng này xảy ra khi mô hình quá phức tạp hoặc có quá nhiều tham số so với lượng dữ liệu huấn luyện có sẵn, dẫn đến việc mô hình "học nhớ" từng điểm dữ liệu trong tập huấn luyện một cách cụ thể, thay vì học được các đặc trưng tổng quát của dữ liệu. Do đó việc tạo hai mạng LSTM độc lập sẽ giúp giảm được tình trạng này.

Cuối cùng chúng tôi sử dụng hai mô hình LSTM trong mỗi nhánh LSTM này có sự khác biệt về lớp LSTM đầu tiên có return-sequences=True nghĩa là nó trả về đầu ra từng chuỗi thời gian. Mục đích của lớp LSTM này là học và trích xuất các đặc trưng từng bước thời gian của dữ liệu đầu vào. Điều này hữu ích khi mô hình cần biết sự thay đổi của dữ liệu. LSTM thứ 2 chỉ trả về đầu ra duy nhất tại bước thời gian cuối cùng. Lớp LSTM đầu tiên giúp mô hình học các đặc trưng phức tạp từng bước thời gian trong chuỗi, trong khi LSTM thứ hai có thể dùng để tổng hợp lại thông tin từ các đặc trưng đã học để đưa ra dự đoán và phân loại cuối cùng.

3.3. Phương pháp đánh giá

Thực tế như đã trình bày, phương pháp đánh giá One-to-many cũng chỉ là gộp lại của nhiều vấn đề One-to-one. Vì thế khi tiến hành đánh giá, đối với mỗi khía cạnh đánh giá chúng tôi chỉ đánh giá một loại, nhưng điều đó cũng có thể giúp ta có thể biết được hiệu quả khi đánh giá với phương pháp kia. Phần này chúng tôi sẽ trình bày về cách mà chúng tôi sẽ đánh giá hiệu quả của mô hình khi thực hiện đánh giá One-to-one hoặc One-to-many.

3.3.1. Phân loại One-to-one

Đối với phân loại One-to-one: Chúng tôi sẽ đánh giá hiệu suất của thông qua 4 thông số thông dụng là Accuracy, Precision, Recall và F1 Score.

Chúng ta sẽ khái quát một số chỉ số và khái niệm cơ bản:

- **True Positive (TP):** Mô hình dự đoán đúng cặp chức năng là tương đồng

và thực tế thì cặp chức năng đó cũng tương đồng.

- **False Positive (FP):** Mô hình dự đoán đúng cặp chức năng là không tương đồng nhưn thực tế thì cặp chức năng đó là cặp chức năng tương đồng.
- **True Negative (TN):** Mô hình dự đoán đúng cặp chức năng là không tương đồng và thực tế thì cặp chức năng đó cũng không tương đồng.
- **False Negative (FN):** Mô hình dự đoán đúng cặp chức năng là không tương đồng nhưng thực tế thì cặp chức năng đó là cặp chức năng tương đồng.

Các giá trị TP, FP, TN và FN tùy thuộc vào số lượng mẫu của dữ liệu. Nhưng trong hầu hết các trường hợp TP và TN càng cao sẽ càng tốt và ngược lại FB, FN càng thấp sẽ càng tốt. Tuy nhiên, vẫn có một số trường hợp đặc biệt, việc tăng giá trị TP có thể làm tăng FP hoặc tăng giá trị TN có thể làm tăng FN. Điều này phụ thuộc vào yêu cầu cụ thể của bài toán. Vì vậy, việc đánh giá hiệu suất của phương pháp không chỉ dựa trên các chỉ số TP, TN, FP và FN mà cần kết hợp với các chỉ số khác như sau:

- **Accuracy (Độ chính xác):** Đo lường khả năng phân loại trong việc dự đoán đúng các lớp (có và không có sự tương đồng) trong tổng số các mẫu

$$\textbf{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

- **Precision (Độ chính xác dương tính):** Giúp đánh giá khả năng trong việc tránh việc phân loại sai các mẫu không có sự tương đồng thành có sự tương đồng

$$\textbf{Precision} = \frac{TP}{TP + FP}$$

- **Recall (Độ bao phủ)** Giúp đo lường khả năng trong việc phát hiện tất

cả các mẫu thuộc lớp tích cực (có sự tương đồng) một cách chính xác

$$\textbf{Recall} = \frac{TP}{TP + FN}$$

- **F1 Score:**Giúp đánh giá hiệu suất tổng thể của mô hình phân loại, đồng thời cân nhắc cả Precision và Recall

$$\textbf{F1 Score} = 2 \times \frac{\textbf{Precision} \times \textbf{Recall}}{\textbf{Precision} + \textbf{Recall}}$$

3.3.2. Phân loại One-to-many

Trong bài toán phát hiện sự tương đồng trong mã nhị phân, sử dụng các chỉ số đánh giá như accuracy, precision, recall và F1-score có thể không còn phù hợp cho phân loại One-to-many vì các chỉ số này thường được sử dụng trong bài toán phân loại nhị phân hoặc đa lớp. Cho nên, đối với phân loại này chúng tôi đánh giá hiệu quả của phương pháp thông qua 2 thông số là MRR và Recall@1.

Mean Reciprocal Rank (MRR): MRR đo lường khả năng của phương pháp trong việc xếp hạng đúng chức năng tương đồng nhất. Giả sử bạn có một danh sách các chức năng tương đồng được sắp xếp theo mức độ tương đồng giảm dần. MRR tính giá trị nghịch đảo của vị trí đầu tiên mà một chức năng thực sự tương đồng được xếp hạng. Công thức tính MRR như sau:

$$MRR = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{\textbf{rank}_i}$$

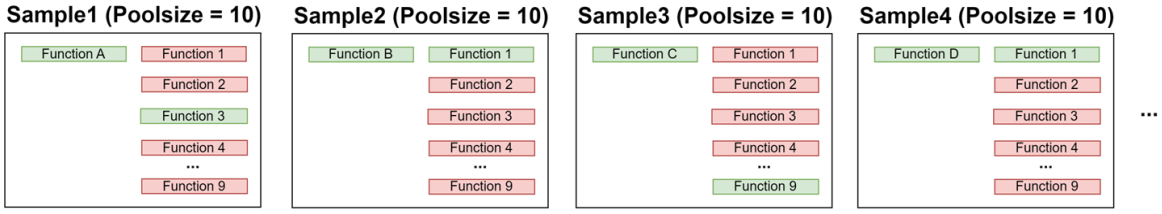
Trong đó, N là số lượng các chức năng tương đồng được xếp hạng, và rank_i là vị trí của chức năng thực sự tương đồng đầu tiên trong danh sách.

Recall@1: Recall@1 đo lường khả năng của hệ thống phát hiện sự tương đồng trong mã nhị phân để tìm ra chính xác một chức năng tương đồng. Nó đơn giản là tỷ lệ phần trăm chức năng thực sự tương đồng được tìm thấy ở vị trí đầu tiên trong danh sách các chức năng tương đồng được xếp hạng. Đây là chỉ

số quan trọng để đảm bảo rằng hệ thống có khả năng tìm ra chức năng tương đồng một cách chính xác. Recall@1 được tính bằng công thức:

$$\textbf{Recall@1} = \frac{\textbf{Số lượng chức năng tương đồng được xếp ở vị trí đầu tiên}}{\textbf{Tổng số mẫu thử}}$$

Để rõ hơn chúng tôi cung cấp một ví dụ về cách tính giá trị của MRR và Recal@1 như sau:



Hình 3.8: *Cách tính MRR và Recal@1.*

Giả sử chúng ta có một loạt các mẫu thử (Ở đây chúng tôi gọi là các sample). Với mỗi sample sẽ có một hàm ban đầu và 9 hàm khác để tạo thành poolsize = 10. Trong 9 hàm khác đó sẽ có một hàm thực sự tương đồng.

Giá trị Recal@1 sẽ bằng thương giữa số lần hàm tương đồng với hàm ban đầu được xếp ở vị trí đầu tiên trong danh sách kết quả trên tổng số sample được đánh giá.

Giá trị MRR sẽ bằng giá trị trung bình kết quả phép chia của 1 với vị trí kết quả mà mô hình trả về.

Cả hai chỉ số MRR và Recall@1 đều đánh giá được hiệu quả của hệ thống phát hiện sự tương đồng trong mã nhị phân. MRR đo lường mức độ chính xác trong việc xếp hạng chức năng tương đồng, trong khi Recall@1 đo lường khả năng tìm ra chính xác một chức năng tương đồng. Sự kết hợp của cả hai chỉ số này sẽ cung cấp cách đánh giá toàn diện về hiệu suất của phương pháp.

CHƯƠNG 4. THÍ NGHIỆM VÀ ĐÁNH GIÁ

Chương này chúng tôi tiến hành tạo môi trường, cài đặt và thực hiện các thí nghiệm để đánh giá hiệu quả của phương pháp mà chúng tôi đề xuất.

4.1. Thiết lập

4.1.1. Cài đặt môi trường

Đối với công viêc trích xuất các chức năng và áp dụng Proc2vec+ để tạo thành vector, chúng tôi thực hiện trên môi trường máy ảo với 16GB Ram, 64GB bộ nhớ trong. Trên máy ảo này, ngôn ngữ chính mà chúng tôi sử dụng là python 3.8.16 cùng với các thứ viện hổ trợ việc trích xuất Vex-IR là pyvex 9.2.52. Đối với công việc tìm ghi nhận địa chỉ của các chức năng trong tập tin nhị phân, chúng tôi dùng API của IDA PRO 7.7. Cuối cùng để huấn luyện và đánh giá mô hình học máy chúng tôi tiến hành cài thư viện TensorFlow 2.12.0.

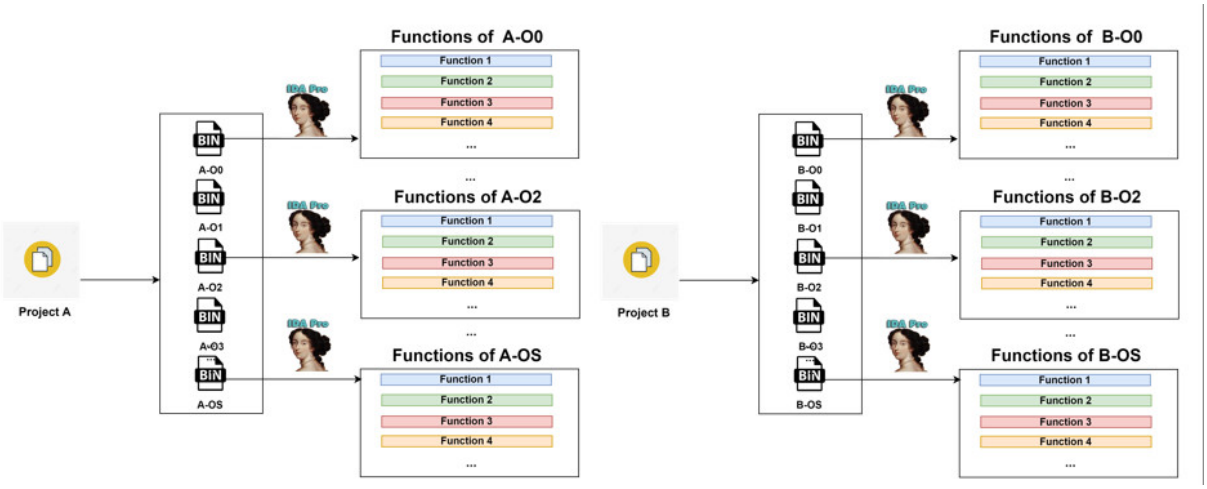
4.1.2. Các thông số của mô hình học sâu

Chúng tôi tiến hành đã xây dựng mô hình của chúng tôi như đã nói ở chương 2 trên thư viện TensorFlow chạy trên máy ảo. Hàm kích hoạt của tất cả các lớp đều là 'relu', riêng lớp cuối cùng là 'softmax'. Chúng tôi còn sử dụng hàm mất mát 'categorical_crossentropy' và thuật toán tối ưu 'adam' trong quá trình biên dịch mô hình

Các siêu tham số lần lượt là 'batch size' là 32 và 'epochs' là 3.

4.2. Quá trình tạo tập dữ liệu huấn luyện

Về tập dữ liệu cho Binshoo, chúng tôi sử dụng 1 phần của tập dữ liệu Binary Corp [10]. Theo như các tác giả, tập dữ liệu này có khả năng giảm thiểu tình trạng overfitting và thiếu đa dạng đặc trưng so với các tập dữ liệu hiện có. Các tác giả đã tự động thu thập tất cả các dự án c/c++ từ phần lớn các dự án mã nguồn mở phổ biến và xây dựng chúng với các tối ưu hóa trình biên dịch khác nhau (Os, O0, O1, O2, O3) để tạo ra các tệp nhị phân khác nhau. Đây được xem là một trong những bộ dữ liệu chương trình nhị phân lớn nhất và đa dạng nhất cho các nhiệm vụ phát hiện sự tương đồng trong mã nhị phân cho đến nay.



Hình 4.1: *Cách tạo hàm tương đồng và không tương đồng.*

Để tạo dữ liệu huấn luyện cho Binshoo. Chúng tôi chọn ra ngẫu nhiên 9 dự án trong BinaryCorp, mỗi dự án đã được biên dịch với 5 lựa chọn tối ưu hóa tình biên dịch khác nhau (Os, O0, O1, O2, O3). Về cách tạo hàm tương đồng và không tương đồng chúng tôi thể hiện như sau.

Với hàm tương đồng từ một function ban đầu sẽ được ghép cặp với function tương ứng của nó tại các tùy chọn biên dịch từ (Os, O0, O1, O2, O3). Với các hàm không tương đồng thì chúng tôi chọn ngẫu nhiên hai hàm bất kỳ từ hai dự án khác nhau trong tập dữ liệu để tạo cặp không tương đồng.

Số lượng hàm của môi dự án chúng tôi xin được tóm tắc trong bảng 4.1

Bảng 4.1: *Bảng tóm tắc kết quả trích xuất các chức năng có trong các dự án được chọn.*

Tên dự án	**Số lượng chức năng**
agensgraph-git-pgoutput.so	13
agensgraph-git-pg$_{c}onfig$	57
agensgraph-git-pgstattuple.so	43
abseil-cpp-libabsl$_{f}lags_{i}nternal.so$.2103.0.1	35
abseil-cpp-libabsl$_{s}ynchronization.so$.2103.0.1	97
abseil-cpp-libabsl$_{s}trings_{i}nternal.so$.2103.0.1	15
agensgraph-git-hstore$_{p}lpython2.so$	10
aes-git-aes	21
aften-git-libaften.so.0.0.8	97

Sau khi đã có các chức năng, chúng tôi áp dụng Proc2vec+ để chuyển đổi chúng thành các vector tương ứng và tạo ra các mẫu tương đồng hoặc không tương đồng. Đối vơi các mẫu tương đồng, từ 1 chức năng và 5 lựa chọn tối ưu hóa, chúng tôi tạo ra được 5 vector tương ứng, tiếp theo sẽ tạo ra các cặp tương đồng tự mỗi bộ 5 vector này với 2 vector sẽ thành 1 cặp. Hơn nữa, để dạy mô hình biết rằng mọi chức năng đều giống với chính nó, và thứ tự đầu vào của một cặp không quan trọng, từ 1 căp (x, y), chúng tôi tạo ra thêm 4 (x, y), (x, x), (y, y), (y, x). Chúng tôi dán nhãn các cặp này bằng cách đánh 1. Cuối cùng chúng tôi tạo ra được 9700 nghìn mẫu tương đồng. Đối với các mẫu không tương đồng, chúng tôi lấy ngẫu nhiên 2 chức năng trong toàn bộ tập dữ liệu dã lựa chọn và đảm bảo răng 2 chức năng trong cặp đó không được biên dịch từ cùng 1 dự án. Về khả năng tạo ra các mẫu không tương đồng là lớn hơn nhiều và có thể tạo ra số lượng rất lớn so với việc tạo mẫu tương đồng. Tuy nhiên để cân bằng dữ liệu tránh việc tỉ lệ giữa 2 loại mẫu chênh lệch quá cao, chúng tôi tạo ra 11396 ngìn mẫu không tương đồng và dán nhãn các cặp này là 0.

Về tổng quát, chúng tôi tạo ra 21096 mẫu cho tập dữ liệu huấn luyện bao gồm cả tương đồng và không tương đồng.

4.3. Hiệu quả của việc cải tiến phương pháp chuyển đổi vector Proc2vec+ so với nguyên mẫu Proc2vec được giới thiệu bởi Zeek.

4.3.1. Tập dữ liệu đánh giá

Phương pháp tạo dữ liệu đánh giá. Chúng tôi sẽ tiến hành chọn ngẫu nhiên các dự án khác trong tập BinaryCorp để trích xuất các chức năng, sau khi biểu diễn thành vector chúng tôi tiến hành tạo ra các cặp chức năng tương đồng và không tương đồng. Kết quả tổng hợp chúng tôi tóm tắc dưới bảng 4.4.

Bảng 4.2: *Bảng tóm tắc kết quả trích xuất các chức năng có trong các dự án được chọn để tạo thành dữ liệu đánh giá.*

Tên dự án	**Số lượng chức năng**
apache-git-mod-asis.so	5
apache-git-mod-negotiation.so	21
apache-git-mod-proxy-balancer.so	19
apache-git-mod-proxy-http.so	17
apache-git-mod-ratelimit.so	8
apcupsd-smtp-apcupsd	21
icu-git-genccode	7

Với các chức năng trên, chúng tôi tạo ra 2953 mẫu tương đồng và 2391 mẫu không tương đồng. Tổng cộng tập dữ liệu đánh giá có 5344 mẫu.

4.3.2. Kết quả so sánh

Đánh giá tác động của Inlining và Constant Folding đối với phương pháp Proc2vec trong bảng 4.3.

Bảng 4.3: *Kết quả so sánh giữa Proc2vec và Proc2vec+*

Mô hình	**Accuracy**	**Precision**	**Recall**	**F1 Score**
CNN	0.905127	0.934613	0.890620	0.912086
CNN+	0.912612	0.941720	0.897392	0.919022
LSTM	0.935067	0.917093	0.970200	0.942899
LSTM+	0.935629	0.916374	0.972232	0.943477
CNN+LSTM	0.923091	0.945028	0.913986	0.929248
CNN+LSTM+	0.933570	0.934448	0.946156	0.940266
CNN+GRU	0.868451	0.957689	0.797155	0.870079
CNN+GRU+	0.882672	0.939199	0.842194	0.888056
BINSHOO	0.975861	0.971925	0.984761	0.978301
BINSHOO+	0.991018	0.987580	0.996275	0.991908

Có thể dễ dàng thấy rằng việc sử dụng Proc2vec+ đã mang lại lợi ích rõ ràng cho việc biểu diễn vector so với Proc2vec trong hầu hết các trường hợp. Tuy nhiên, cũng cần lưu ý rằng hiệu suất của mô hình có thể phụ thuộc vào đặc điểm cụ thể của dữ liệu và mô hình được sử dụng.

4.4. So sánh độ hiệu quả giữa mô hình học máy của Binshoo với các mô hình học máy khác CNN, LSTM, CNN+LSTM, CNN+GRU, Zeek.

4.4.1. Tập dữ liệu đánh giá

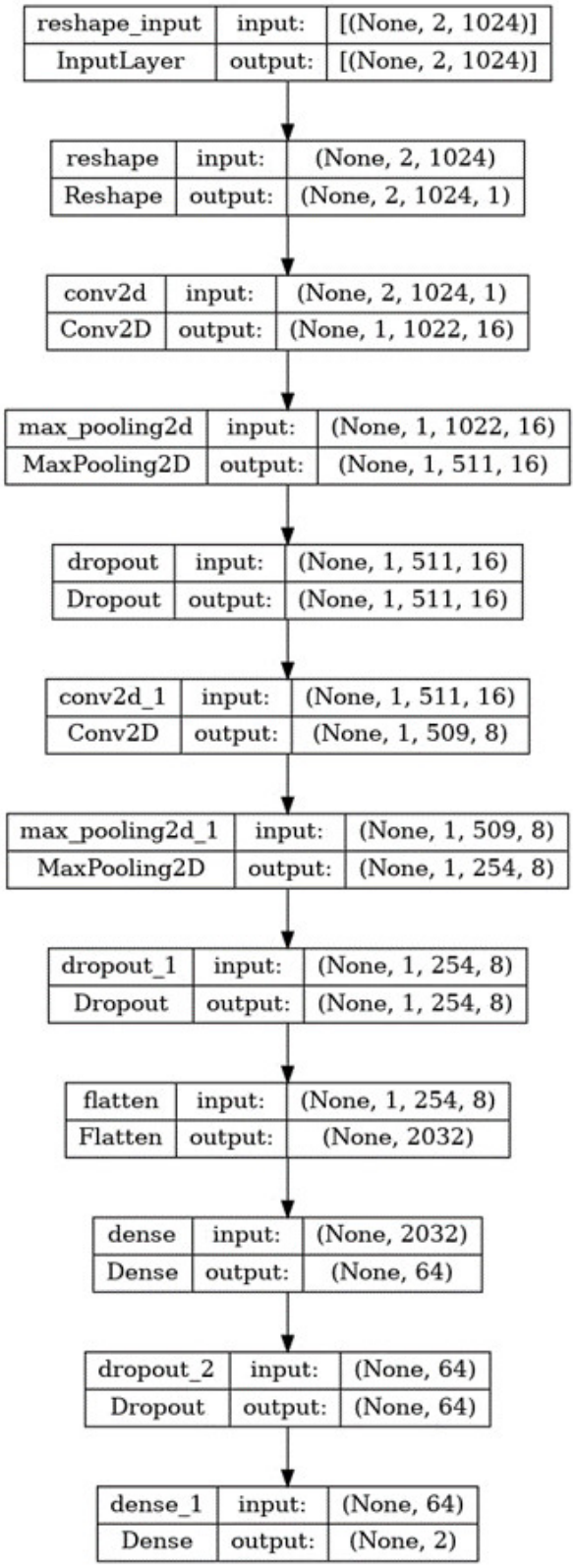
Phương pháp tạo dữ liệu đánh giá tương đồng với những gì chúng tôi đã làm ở phần 4.2. Chúng tôi sẽ tiến hành chọn ngẫu nhiên các dự án khác trong tập BinaryCorp để trích xuất các chức năng, sau khi biểu diễn thành vector chúng tôi tiến hành tạo ra các cặp chức năng tương đồng và không tương đồng. Kết quả tổng hợp chúng tôi tóm tắc dưới bảng 4.4.

Bảng 4.4: *Bảng tóm tắc kết quả trích xuất các chức năng có trong các dự án được chọn để tạo thành dữ liệu đánh giá*

Tên dự án	**Số lượng chức năng**
apache-git-mod-asis.so	5
apache-git-mod-negotiation.so	21
apache-git-mod-proxy-balancer.so	19
apache-git-mod-proxy-http.so	17
apache-git-mod-ratelimit.so	8
apcupsd-smtp-apcupsd	21
icu-git-genccode	7

Với các chức năng trên, chúng tôi tạo ra 2953 mẫu tương đồng và 2391 mẫu không tương đồng. Tổng cộng tập dữ liệu đánh giá này có 5344 mẫu.

4.4.2. Kiến trúc của mô hình của CNN



Hình 4.2: *Các layer trong mô hình CNN*

Sequential Model: Mô hình được xây dựng bằng cách sử dụng Sequential(), nó là một dạng của mô hình mạng nơ-ron theo kiểu tuyến tính, trong đó các layer được xếp chồng lên nhau theo thứ tự.

Reshape Layer: Layer này chuyển đổi đầu vào thành một kích thước cụ thể. Ở đây, đầu vào có kích thước (2, 1024), và nó được chuyển thành một tensor có kích thước (2, 1024, 1).

Conv2D Layer (1): Layer convolution với 16 filters, mỗi filter có kích thước (2, 3), và kích hoạt hàm là 'relu' (Rectified Linear Unit).

MaxPooling2D Layer (1): Layer max pooling với pool size là (1, 2). Max pooling giúp giảm kích thước của đầu ra từ layer trước đó và giữ lại giá trị lớn nhất trong mỗi cửa sổ.

Dropout Layer (1): Layer dropout với tỷ lệ dropout là 0.2. Dropout là một kỹ thuật chính để tránh việc quá mức học của mô hình và giảm nguy cơ quá mức đặc hóa (overfitting).

Conv2D Layer (2): Một lớp convolution khác với 8 filters, mỗi filter có kích thước (1, 3), và kích hoạt hàm 'relu'.

MaxPooling2D Layer (2): Max pooling với pool size là (1, 2).

Dropout Layer (2): Dropout với tỷ lệ là 0.2.

Flatten Layer: Chuyển từ tensor 2D sang vector 1D để chuẩn bị cho các layer fully connected.

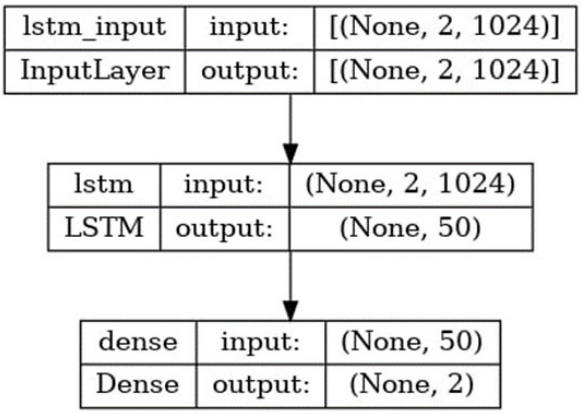
Dense Layer (1): Một layer fully connected với 64 neurons và kích hoạt hàm 'relu'.

Dropout Layer (3): Dropout với tỷ lệ là 0.5.

Dense Layer (2): Layer fully connected cuối cùng với 2 neurons được sử dụng để chuyển đổi đầu ra thành xác suất, tổng các xác suất bằng 1.

Compile Model: Cuối cùng, mô hình được biên dịch với hàm mất mát là 'categorical-crossentropy' , optimizer là 'adam', và đánh giá hiệu suất sử dụng chỉ số 'accuracy'.

4.4.3. Kiến trúc của mô hình của LSTM



Hình 4.3: *Các layer của mô hình LSTM*

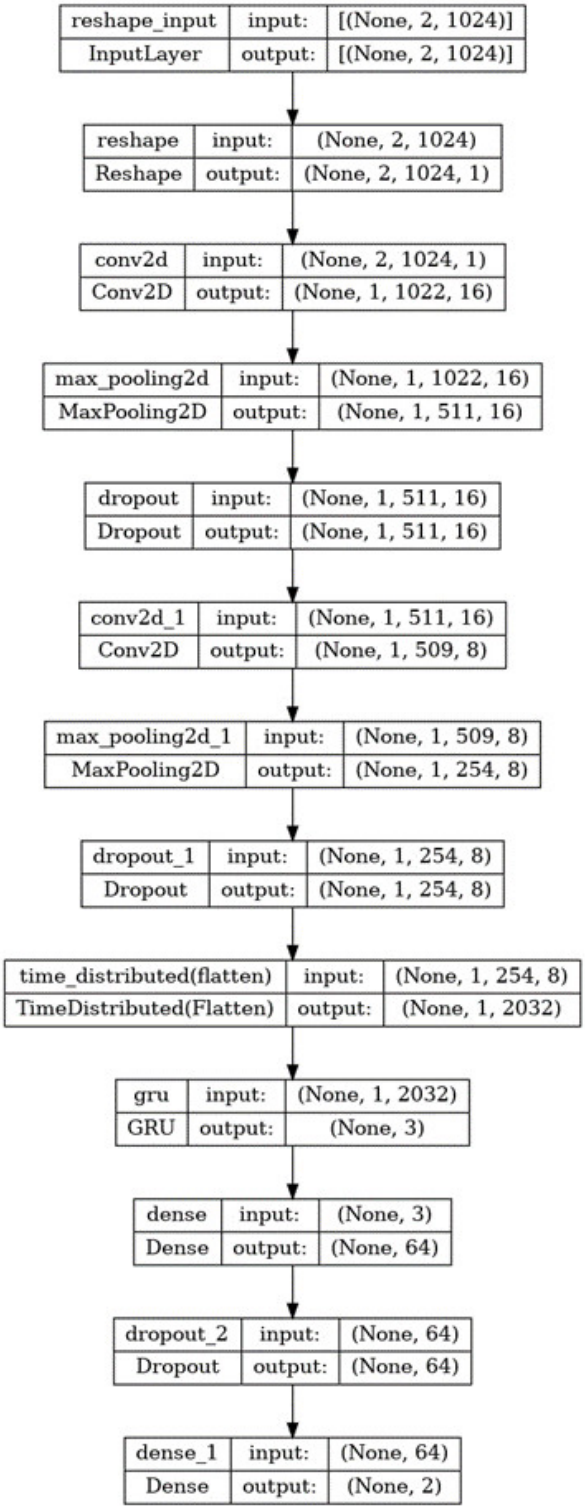
Layer LSTM sẽ được áp dụng cho dữ liệu đầu vào có kích thước (2, 1024), trong đó 2 là chiều dài chuỗi và 1024 là số chiều của mỗi điểm dữ liệu trong chuỗi.

Dense Layer: Layer này có 2 neurons và sử dụng hàm kích hoạt 'softmax'. Softmax được sử dụng để chuyển đổi đầu ra thành xác suất, tổng các xác suất bằng 1.

Compile Model: Xác định hàm mất mát là 'categorical-crossentropy' , bộ tối ưu hóa là 'adam', và đánh giá hiệu suất sử dụng chỉ số 'accuracy'.

Mô hình sau khi được định nghĩa và biên dịch có thể được sử dụng để huấn luyện trên dữ liệu và thực hiện dự đoán.

4.4.4. Kiến trúc của mô hình của CNN + GRU



Hình 4.4: *Các layer trong mô hình CNN + GRU*

Reshape Layer: Chuyển đổi đầu vào từ kích thước (2, 1024) thành (2, 1024, 1), biến nó thành một tensor ảnh đen trắng với chiều sâu là 1.

Conv2D Layer (1): Áp dụng một số bộ lọc (filters) có kích thước (2, 3) để tìm các đặc trưng trong ảnh. Kích hoạt các đặc trưng này bằng hàm kích hoạt 'relu'.

MaxPooling2D Layer (1): Giảm kích thước của ảnh và giữ lại giá trị lớn nhất trong mỗi cửa sổ, giúp giảm chiều sâu của dữ liệu và tăng tính trừu tượng của đặc trưng.

Dropout Layer (1): Ngẫu nhiên "tắt" một số neuron để giảm nguy cơ quá mức đặc hóa trong quá trình huấn luyện.

Conv2D Layer (2): Tiếp tục trích xuất đặc trưng từ dữ liệu, sử dụng các bộ lọc có kích thước (1, 3).

MaxPooling2D Layer (2): Giảm kích thước của ảnh một lần nữa để tăng tính trừu tượng và giảm độ phức tạp của mô hình.

Dropout Layer (2): Thêm một lớp dropout để ngăn chặn overfitting.

TimeDistributed Layer + Flatten Layer: Thông thường, layer Flatten được sử dụng để chuyển từ tensor 2D sang vector 1D trong mạng CNN. Tuy nhiên, khi kết hợp với TimeDistributed, nó áp dụng Flatten cho mỗi "thời điểm" trong chuỗi. Điều này hữu ích khi bạn có một loạt các dữ liệu theo thời gian.

GRU Layer: GRU là một kiểu mạng nơ-ron hồi quy thường được sử dụng để xử lý dữ liệu chuỗi. Trong trường hợp này, layer GRU sẽ được áp dụng cho dữ liệu được chuyển từ các layer trước. GRU giúp mô hình học được các mối quan hệ phức tạp trong dữ liệu chuỗi và duy trì thông tin trong thời gian dài.

Dense Layer (1): Tạo ra các kết nối đầy đủ giữa các neuron, giúp học các mối quan hệ phức tạp trong dữ liệu.

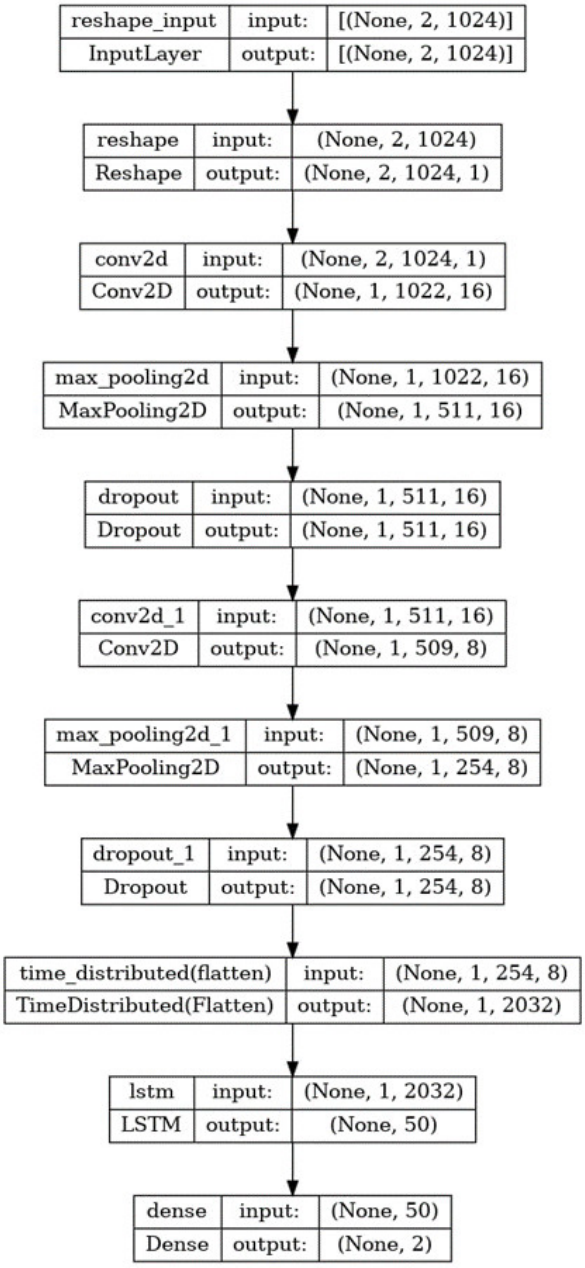
Dropout Layer (3): Tiếp tục sử dụng dropout để giảm overfitting trong các layer fully connected.

Dense Layer (2): Là layer output với 2 neurons và kích hoạt softmax để chuyển

đổi đầu ra thành xác suất, phục vụ cho bài toán phân loại thành hai lớp.

Compile Model: Xác định hàm mất mát là 'categorical-crossentropy', bộ tối ưu hóa là 'adam', và đánh giá hiệu suất sử dụng chỉ số 'accuracy'.

4.4.5. Kiến trúc của mô hình CNN + LSTM

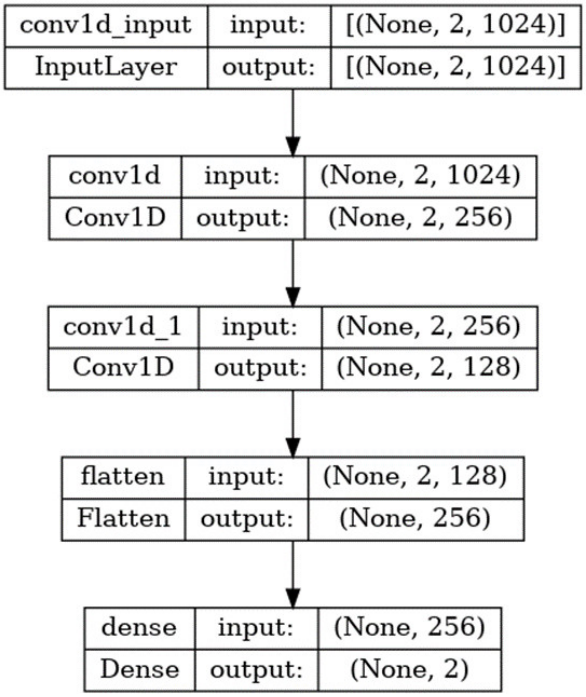


Hình 4.5: *Các layer và kích thước có trong mô hình của CNN + LSTM*

Tương tự như CNN+GRU nhưng thay GRU bằng LSTM.

LSTM là một kiểu mạng nơ-ron hồi quy thường được sử dụng để xử lý dữ liệu chuỗi. Trong trường hợp này, layer LSTM sẽ được áp dụng cho dữ liệu được chuyển từ các layer trước. LSTM giúp mô hình học được các mối quan hệ phức tạp trong dữ liệu chuỗi và duy trì thông tin trong thời gian dài.

4.4.6. Kiến trúc của mô hình Zeek

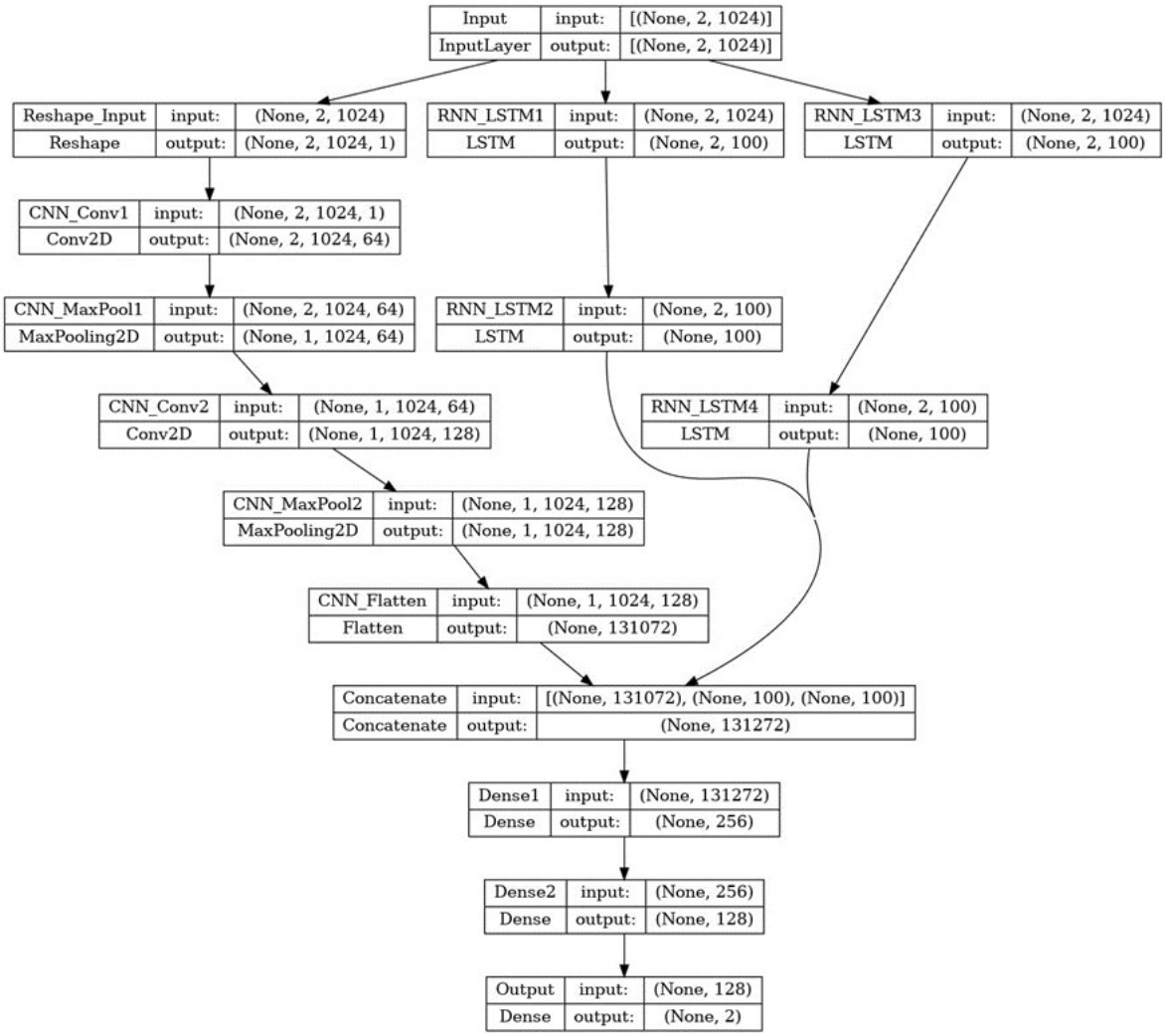


Hình 4.6: *Các layer và kích thước có trong mô hình của Zeek*

Mô hình của Zeek là một mô hình CNN có kiến trúc tuần tự. Mô hình bắt đầu với một lớp Conv1D có 256 bộ lọc, kernel size là 1 và hàm kích hoạt tanh. Đây là lớp đầu tiên trong mạng và nó giúp trích xuất các đặc trưng từ đầu vào có kích thước (2, 1024), tức là 2 vector đầu vào. Tiếp theo, một lớp Conv1D khác được sử dụng với 128 bộ lọc, kernel size là 1 và hàm kích hoạt tanh. Lớp này cũng giúp trích xuất thêm các đặc trưng từ đầu vào. Sau đó, đầu ra của lớp Conv1D này được làm phẳng (Flatten) để chuẩn bị cho việc đưa vào các lớp Fully Connected. Một lớp Fully Connected với 2 đơn vị và hàm kích hoạt

softmax được sử dụng làm lớp đầu ra của mô hình. Đầu ra của lớp này sẽ dự đoán xác suất xảy ra của hai lớp khác nhau. Mô hình được biên dịch với hàm mất mát categorical-crossentropy và tối ưu hóa adam, cùng với độ đo accuracy để đánh giá hiệu suất của mô hình. Đồng thời nó cũng được huấn luyện với batch size là 32 và 3 epochs tương tự như mô hình của chúng tôi.

4.4.7. Kiến trúc học máy của mô hình Binshoo



Hình 4.7: *Các layer và kích thước có trong mô hình Binshoo*

Cuối cùng sẽ là phần mô tả chi tiết về mô hình của chúng tôi.

Mô hình Siamese Neural Network được thiết kế để phát hiện sự tương đồng

trong mã nhị phân bằng cách kết hợp các lớp CNN và LSTM. Đầu tiên, lớp đầu vào nhận dữ liệu có hình dạng (2, 1024), sau đó lớp Reshape chuyển đổi dữ liệu thành định dạng phù hợp với Conv2D. Nhánh CNN gồm các lớp Conv2D với 64 và 128 bộ lọc cùng các lớp MaxPooling2D giúp trích xuất và giảm kích thước đặc trưng không gian từ dữ liệu. Kết quả từ nhánh CNN được làm phẳng bởi lớp Flatten. Đồng thời, hai nhánh LSTM song song, mỗi nhánh có 100 đơn vị, giúp nắm bắt các phụ thuộc tuần tự trong dữ liệu. Đầu ra từ nhánh CNN và hai nhánh LSTM được kết hợp bằng lớp concatenate. Sau đó, các lớp Dense với 256 và 128 đơn vị được áp dụng để học các đặc trưng sâu hơn từ dữ liệu kết hợp. Cuối cùng, lớp đầu ra Dense với 2 đơn vị và hàm kích hoạt softmax đưa ra dự đoán xác suất cho hai lớp. Mô hình được biên dịch với hàm mất mát categorical-crossentropy, trình tối ưu hóa adam và được đánh giá bằng độ chính xác. Sự kết hợp giữa CNN và LSTM giúp mô hình học tốt hơn các đặc trưng không gian và tuần tự, cải thiện độ chính xác và khả năng phát hiện sự tương đồng trong mã nhị phân.

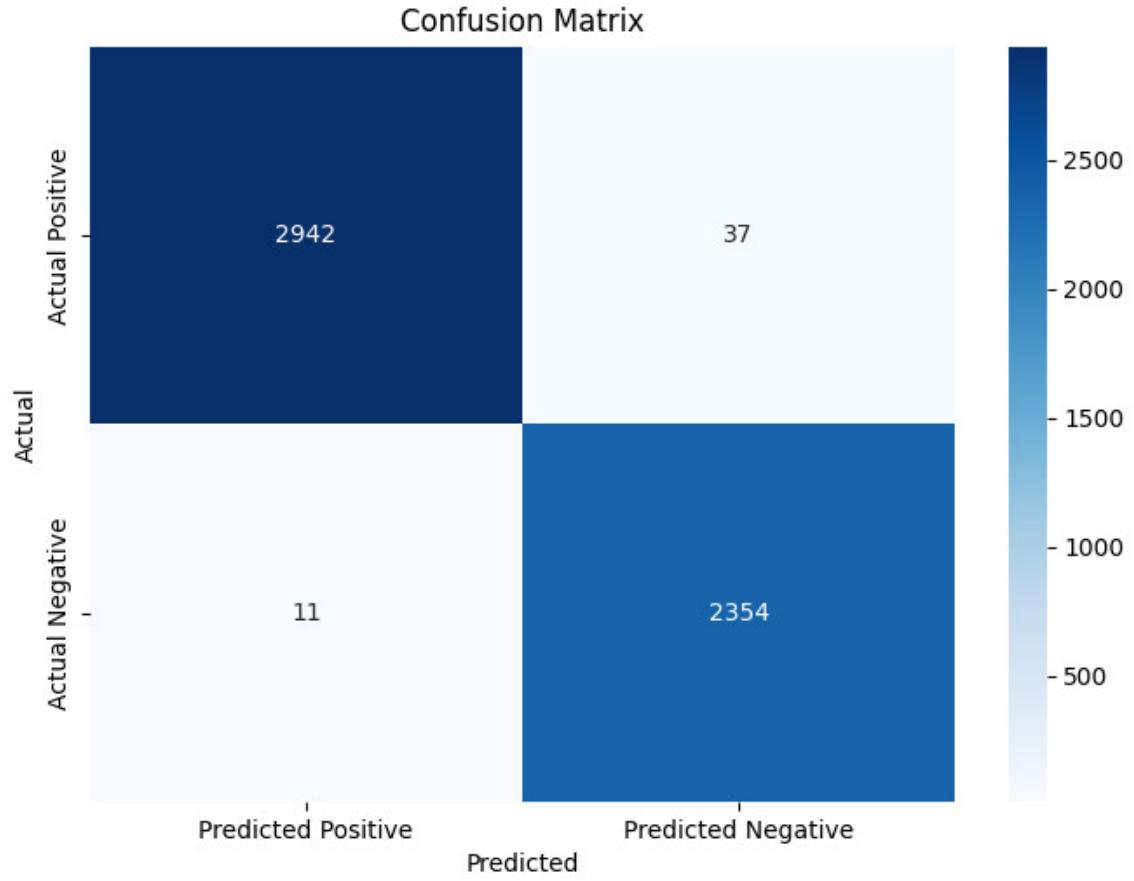
4.4.8. Kết quả so sánh

Chúng tôi ghi nhận kết quả như bảng 4.5.

Bảng 4.5: *Kết quả so sánh khi sử dụng các mô hình học máy khác nhau CNN, LSTM, CNN + GRU, CNN + LSTM và Binshoo*

Mô hình	**Accuaracy**	**Precision**	**Recall**	**F1 Scoe**
CNN	0.912612	0.941720	0.897392	0.919022
LSTM	0.935629	0.916374	0.972232	0.943477
CNN+LSTM	0.933570	0.934448	0.946156	0.940266
CNN+GRU	0.882672	0.939199	0.842194	0.888056
ZEEK	0.780501	0.832338	0.754826	0.791689
BINSHOO	0.991018	0.987580	0.996275	0.991908

Mô hình Binshoo xuất sắc hơn với giá trị cao nhất ở các chỉ số Precision, Recall, F1 Score, và Accuracy. Điều này cho thấy nó có khả năng phân loại đúng cao cả trong việc dự đoán positive và negative instances.



Hình 4.8: *Confusion matrix 4 chỉ số TP, FP, TN, FN khi đánh giá bằng mô hình Binshoo*

4.5. So sánh độ hiệu quả giữa mô hình học máy của Binshoo và mô hình học máy BERT.

BERT [11] (Bidirectional Encoder Representations from Transformers) là một mô hình học máy tiên tiến được phát triển bởi Google, nhằm mục đích cải thiện khả năng hiểu ngôn ngữ tự nhiên (NLP). BERT sử dụng kiến trúc Transformer, cụ thể là phần encoder của nó, và có khả năng học các biểu diễn ngữ nghĩa từ văn bản bằng cách xem xét ngữ cảnh của từ ở cả hai chiều (trái

và phải).

Phương pháp tạo dữ liệu huấn luyện. Chúng tôi sẽ tiến hành chọn ngẫu nhiên các dự án khác trong tập BinaryCorp để trích xuất các chức năng, sau khi biểu diễn thành vector chúng tôi tiến hành tạo ra các cặp chức năng tương đồng và không tương đồng. Kết quả tổng hợp chúng tôi tóm tắc dưới bảng 4.6.

Bảng 4.6: *Bảng tóm tắc kết quả trích xuất các chức năng có trong các dự án được chọn để tạo thành dữ liệu huấn luyện.*

Tên dự án	**Số lượng chức năng**
apache-git-mod-asis.so	5
apache-git-mod-negotiation.so	21
apache-git-mod-proxy-balancer.so	19
apache-git-mod-proxy-http.so	17
apache-git-mod-ratelimit.so	8
apcupsd-smtp-apcupsd	21
icu-git-genccode	7

Với các chức năng trên, chúng tôi tạo ra 2953 mẫu tương đồng và 2391 mẫu không tương đồng. Tổng cộng tập dữ liệu huấn luyện có 5344 mẫu.

4.5.1. Tập dữ liệu đánh giá

Trong phần này, đầu tiên cả hai mô hình học máy sẽ được đánh giá trên cùng một tập dữ liệu. Để đảm bảo tính khách quan, chúng tôi đã tạo ra một tập dữ liệu đánh giá từ các dự án khác trong BinaryCorp.

Để tạo tập dữ liệu đánh giá, chúng tôi đã chọn ngẫu nhiên các dự án khác từ tập dữ liệu BinaryCorp như sau.

Bảng 4.7: *Bảng tóm tắc kết quả trích xuất các chức năng có trong các dự án được chọn để tạo thành dữ liệu đánh giá*

Tên dự án	**Số lượng chức năng**
ptii-git-ptii	20
qgis-git-libauthmethod-identcert.so	63
libindi-indi-siefs-focus	35
kodi-platform-git-libkodiplatform.so.19.1.0	27
kiiro-notes-git-KiiroNotes	40

Đối với mỗi dự án, chúng tôi đã chọn ngẫu nhiên một chức năng và một tập hợp các chức năng khác (pool) để tạo thành một mẫu. Trong tập hợp các chức năng đó, chỉ có một chức năng duy nhất tương đồng với chức năng ban đầu. Nhiệm vụ của cả hai công cụ là tìm ra chức năng tương đồng đó và đặt nó ở vị trí đầu tiên

Để có kết quả chi tiết hơn, chúng tôi không trộn ngẫu nhiên các lựa chọn tối ưu hóa trình biên dịch với nhau. Thay vào đó, chúng tôi chỉ định các cặp chức năng được tối ưu hóa. Tức là ở mỗi dự án, chúng tôi chọn ra 2 chương trình biên dịch với lựa chọn tối ưu hóa khác nhau để làm 1 tâp dữ liệu, ví dụ O2 và O3 để xem khả năng phát hiện sự tương đồng giữa các chức năng biên dịch bằng lựa chọn tôi ưu O2 và O3, tương tự chúng tôi cũng làm vậy cho các lựa chọn khác. Bảng 4.8 tóm tắc lại các tập dữ liệu mà chúng tôi đã tạo được

Bảng 4.8: *Bảng tóm tắc các tập dữ liệu để so sánh mô hình học máy của BInshoo và mô hình học máy BERT.*

Cross option	**Number sample**
(O0,O2)	898
(O0,Os)	824
(O2,O3)	647
(O1,O2)	640
(O1,Os)	571

Lưu ý tuy là cùng trích xuất từ một tập các dự án nhưng số lượng mẫu không giống nhau là vì chỉ khi xuất hiện các chức năng tương đồng trong cả 2 lựa chọn tối ưu hóa, chúng tôi mới có thể tạo thành một mẫu. Do lựa chọn tối ưu hóa khác nhau nên kết quả các tập tin nhị phân cũng sẽ rất khác, từ đó số lượng các mẫu tạo được cũng sẽ không giống nhau.

Nhiệm vụ này được đánh giá qua 2 chỉ số là MRR và Recal@1.

4.5.2. Kết quả

Kết quả sau khi huấn luyện và đánh giá đối với mô hình học máy của chúng tôi và mô hình học máy BERT được thể hiện ở bảng 4.9

Bảng 4.9: *So sánh kết quả mô hình học máy của Binshoo và mô hình học máy BERT với poolsize = 10*

Công cụ	**Cross Option**	**MMR**	**Recal@1**
Binshoo	(O0,O2)	0.6032	0.4955
	(O0,Os)	0.5985	0.4733
	(O2,O3)	0.8278	0.7496
	(O1,O2)	0.6827	0.5422
	(O1,Os)	0.6475	0.5219
	Average	**0.67194**	**0.5565**
BERT	(O0,O2)	0.5888	0.4310
	(O0,Os)	0.5828	0.4223
	(O2,O3)	0.7105	0.5487
	(O1,O2)	0.6286	0.4406
	(O1,Os)	0.6129	0.4291
	Average	**0.6247**	**0.4543**

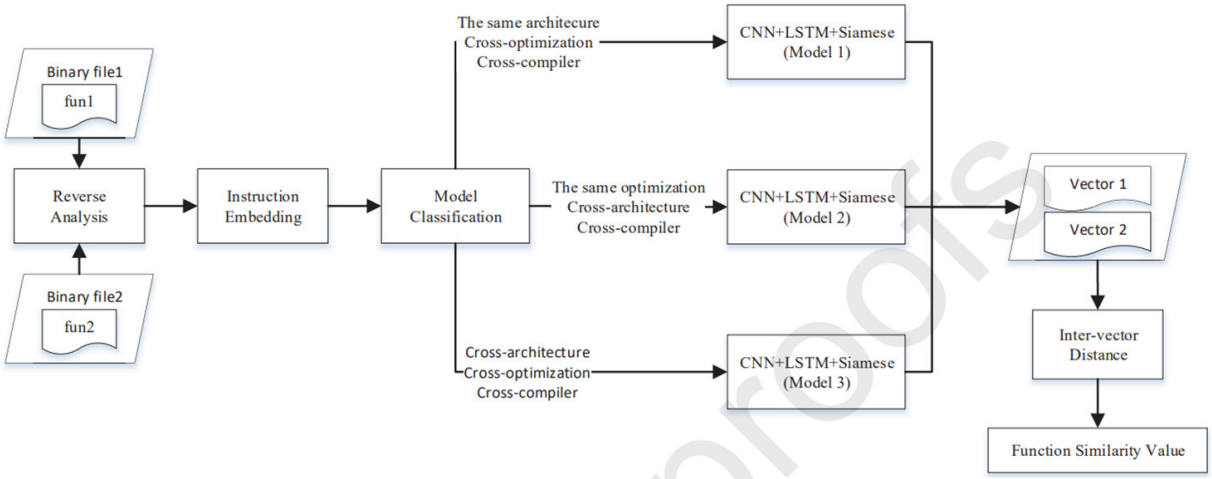
Từ các kết quả thống kê, chúng ta có thể nhận thấy rằng mô hình học máy của Binshoo hiệu quả hơn tương đối so với mô hình học máy BERT. Các chỉ số mô hình học máy của Binshoo đều tỏ ra vượt trội so với mô hình học máy BERT.

Tuy nhiên, dù có kết quả cao nhưng độ hiệu quả mô hình học máy của Binshoo tỏ ra không ổn định, độ hiệu quả của các nhóm (O0,O2), (O0,Os), (O1,O2), (O1,Os) khá thấp so với nhóm (O2,O3), . Điều này là do giữa các lựa chọn tối ưu hóa (O0,O2), (O0,Os), (O1,O2), (O1,Os) có sự khác biệt nhiều hơn so với các lựa chọn tối ưu (O2,O3) và do quá trình tối ưu hóa chưa được hoàn toàn tối ưu, công cụ gặp khó khăn khi cố gắng trích xuất ngữ nghĩa để xác định các chức năng tương đồng. Trong tương lai có thể cải thiện quá tình tối ưu hóa để có thể đạt được sự ổn định cao hơn.

4.6. So sánh độ hiệu quả giữa công cụ Binshoo và công cụ Bindeep

BinDeep[9] là một công cụ chuyên biệt được thiết kế để phát hiện sự tương đồng trong mã nhị phân bằng cách sử dụng các mô hình học sâu, đặc biệt là Mạng Nơ-ron Tích Chập (CNNs) và Mạng Nơ-ron Hồi quy Dài Ngắn hạn (LSTM), để phân tích mã nhị phân. CNNs được sử dụng để trích xuất các đặc trưng từ mã nhị phân, trong khi LSTM giúp xử lý các thông tin tuần tự và mối quan hệ dài hạn trong mã.

Mô hình Bindeep được các tác giả đề xuất và mô tả như sau.



Hình 4.9: *BinDeep: Phương pháp học sâu để phát hiện sự tương đồng mã nhị phân.*

Phương pháp tạo dữ liệu huấn luyện. Chúng tôi sử dụng lại bộ dữ liệu đã được nêu ở phần so sánh với mô hình học máy BERT[11]. Cụ thể chúng tôi sẽ tiến hành chọn ngẫu nhiên các dự án khác trong tập BinaryCorp để trích xuất các chức năng, sau khi biểu diễn thành vector chúng tôi tiến hành tạo ra các cặp chức năng tương đồng và không tương đồng. Kết quả tổng hợp chúng tôi tóm tắc dưới bảng 4.10.

Bảng 4.10: *Bảng tóm tắc kết quả trích xuất các chức năng có trong các dự án được chọn để tạo thành dữ liệu huấn luyện.*

Tên dự án	**Số lượng chức năng**
apache-git-mod-asis.so	5
apache-git-mod-negotiation.so	21
apache-git-mod-proxy-balancer.so	19
apache-git-mod-proxy-http.so	17
apache-git-mod-ratelimit.so	8
apcupsd-smtp-apcupsd	21
icu-git-genccode	7

Với các chức năng trên, chúng tôi tạo ra 2953 mẫu tương đồng và 2391 mẫu không tương đồng. Tổng cộng tập dữ liệu huấn luyện có 5344 mẫu.

4.6.1. Tập dữ liệu đánh giá

Trong phần này, đầu tiên cả hai mô hình học máy sẽ được đánh giá trên cùng một tập dữ liệu. Để đảm bảo tính khách quan, chúng tôi đã tạo ra một tập dữ liệu đánh giá từ các dự án khác trong BinaryCorp.

Để tạo tập dữ liệu đánh giá, chúng tôi đã chọn ngẫu nhiên các dự án khác từ tập dữ liệu BinaryCorp như sau.

Bảng 4.11: *Bảng tóm tắc kết quả trích xuất các chức năng có trong các dự án được chọn để tạo thành dữ liệu đánh giá*

Tên dự án	**Số lượng chức năng**
ptii-git-ptii	20
qgis-git-libauthmethod-identcert.so	63
libindi-indi-siefs-focus	35
kodi-platform-git-libkodiplatform.so.19.1.0	27
kiiro-notes-git-KiiroNotes	40

Đối với mỗi dự án, chúng tôi đã chọn ngẫu nhiên một chức năng và một tập hợp các chức năng khác (pool) để tạo thành một mẫu. Trong tập hợp các chức năng đó, chỉ có một chức năng duy nhất tương đồng với chức năng ban đầu. Nhiệm vụ của cả hai công cụ là tìm ra chức năng tương đồng đó và đặt nó ở vị trí đầu tiên

Để có kết quả chi tiết hơn, chúng tôi không trộn ngẫu nhiên các lựa chọn tối ưu hóa trình biên dịch với nhau. Thay vào đó, chúng tôi chỉ định các cặp chức năng được tối ưu hóa. Tức là ở mỗi dự án, chúng tôi chọn ra 2 chương trình biên dịch với lựa chọn tối ưu hóa khác nhau để làm 1 tâp dữ liệu, ví dụ O2 và O3 để xem khả năng phát hiện sự tương đồng giữa các chức năng biên dịch bằng lựa chọn tôi ưu O2 và O3, tương tự chúng tôi cũng làm vậy cho các lựa chọn khác. Bảng 4.12 tóm tắc lại các tập dữ liệu mà chúng tôi đã tạo được

Bảng 4.12: *Bảng tóm tắc các tập dữ liệu để so sánh công cụ Binshoo và công cụ Bindeep.*

Cross option	**Number sample**
(O0,O2)	898
(O0,Os)	824
(O2,O3)	647
(O1,O2)	640
(O1,Os)	571

Lưu ý tuy là cùng trích xuất từ một tập các dự án nhưng số lượng mẫu không giống nhau là vì chỉ khi xuất hiện các chức năng tương đồng trong cả 2 lựa chọn tối ưu hóa, chúng tôi mới có thể tạo thành một mẫu. Do lựa chọn tối ưu hóa khác nhau nên kết quả các tập tin nhị phân cũng sẽ rất khác, từ đó số lượng các mẫu tạo được cũng sẽ không giống nhau.

Nhiệm vụ này được đánh giá qua 2 chỉ số là MRR và Recal@1.

4.6.2. Kết quả

Kết quả so sánh công cụ Binshoo và công cụ Bindeep được thể hiện ở bảng 4.13.

Bảng 4.13: *So sánh kết quả của công cụ Binshoo và công cụ Bindeep với poolsize = 10*

Công cụ	**Cross Option**	**MMR**	**Recal@1**
Binshoo	(O0,O2)	0.6032	0.4955
	(O0,Os)	0.5985	0.4733
	(O2,O3)	0.8278	0.7496
	(O1,O2)	0.6827	0.5422
	(O1,Os)	0.6475	0.5219
	Average	**0.67194**	**0.5565**
Bindeep TH1	(O0,O2)	0.4089	0.2261
	(O0,Os)	0.4497	0.2706
	(O2,O3)	0.4017	0.1870
	(O1,O2)	0.3549	0.1453
	(O1,Os)	0.3882	0.1891
	Average	**0.4009**	**0.2036**
Bindeep TH2	(O0,O2)	0.3679	0.1760
	(O0,Os)	0.3823	0.1893
	(O2,O3)	0.3026	0.1252
	(O1,O2)	0.2999	0.1125
	(O1,Os)	0.2872	0.1051
	Average	**0.3278**	**0.1414**
Bindeep TH3	(O0,O2)	0.3120	0.1403
	(O0,Os)	0.3261	0.1359
	(O2,O3)	0.3095	0.1360
	(O1,O2)	0.3069	0.1359
	(O1,Os)	0.3024	0.1173
	Average	**0.3118**	**0.1335**

Trong đó Bindeep TH1 đại diện cho trường hợp Bindeep cùng một kiến trúc,

khác tối ưu hóa và khác trình biên dịch, Bindeep TH2 đại diện cho Bindeep cùng một tối ưu hóa, khác kiến trúc và khác trình biên dịch, và Bindeep TH3 đại diện cho trường hợp Bindeep khác kiến trúc, khác tối ưu hóa và khác trình biên dịch.

Dựa trên bảng so sánh kết quả giữa hai công cụ Binshoo và Bindeep với poolsize = 10, chúng ta có thể rút ra một số nhận xét về tính hiệu quả của chúng trong việc phát hiện sự tương đồng mã nhị phân. Đầu tiên, Binshoo đạt được MMR (Mean Reciprocal Rank) trung bình là 0.67194 và Recall@1 trung bình là 0.5565, cho thấy khả năng của nó trong việc xếp hạng chính xác và phát hiện mã nhị phân tương đồng ngay ở lần thử đầu tiên là khá cao. Trong khi đó, các phiên bản Bindeep (TH1, TH2, TH3) đều có MMR và Recall@1 thấp hơn đáng kể. Cụ thể, Bindeep TH1 đạt MMR trung bình là 0.4009 và Recall@1 trung bình là 0.2036; Bindeep TH2 đạt MMR trung bình là 0.3278 và Recall@1 trung bình là 0.1414; và Bindeep TH3 đạt MMR trung bình là 0.3118 và Recall@1 trung bình là 0.1335. Khi xem xét chi tiết các cài đặt cross option khác nhau, Binshoo luôn vượt trội so với Bindeep. Với các giá trị MMR và Recall@1 cao hơn đáng kể, Binshoo tỏ ra vượt trội trong việc phát hiện sự tương đồng mã nhị phân, cho thấy khả năng xếp hạng và nhận diện mã tương đồng tốt hơn.

Tóm lại, thử nghiệm cho thấy Binshoo đã đạt được kết quả cao và thể hiện rằng nó là một công cụ có tiềm năng phát triển.

CHƯƠNG 5. KẾT LUẬN

Phần này chúng tôi đưa ra những kết luận về phương pháp đã nghiên cứu, đưa ra những hạn chế, và đồng thời chỉ ra hướng cải thiện và phát triển.

5.1. Kết luận

Qua việc xây dựng phương pháp phát hiện sự tương đồng trong mã nhị phân lần này, nhóm chúng tôi đã có cơ hội tiềm hiểu thêm về các nghiên cứu liên quan, hiểu được các hạn chế để góp phần cải thiện dần. Nhìn chung, chúng tôi đã đạt được các kết quả sau:

- Hiểu về tập tin nhị phân, các thành phần liên quan của tập tin nhị phân, cách trích xuất các bytes code từ tập tin nhị phân.
- Sử dụng được decompiler với tập tin nhị phân như Ida,..
- Tìm hiểu về ngôn ngữ biểu diễn trung gian, cụ thể là Vex-IR
- Cải tiến được Proc2vec: Phương pháp chuyển đổi từ bytes code sang vector.
- Tìm hiểu về các kiến trúc học sâu.
- Xây dựng được mô hình học sâu mang lại hiệu quả cao hơn so với một số mô hình trước đó.
- Xây dựng được Binshoo công cụ phát hiện sự tương đông trong mã nhị phân đạt được kết quả tốt.

Từ các kết quả mà nhóm đã thu được qua quá trình thực nghiêm, có thể thấy rằng phương pháp của chúng tôi hoàn toàn có thể dùng để nghiên cứu sâu hơn, dựa vào đó để phát triển ra các công cụ hoàn thiện hơn.

Tuy vậy nghiên cứu của chúng tôi cũng có những hạn chế:

- Chỉ đánh giá được phương pháp đối với các chương trình nhị phân được biên dịch với các option khác nhau.
- Tập dữ liệu để huấn luyện và đánh giá chưa đủ lớn để đánh giá 1 cách chính xác, cũng như nâng cao hiệu suất cho phương pháp.

5.2. Hướng phát triển

Dù kết quả khả quan, nhưng so với một số phương pháp phát hiện sự tương đồng trong mã nhị phân hiện đại ngày nay, nghiên cứu của chúng tôi còn hạn chế. Trong tương lai chúng tôi sẽ đánh giá lại phương pháp, cải tiến phương pháp (Đặc biệt là quá trình tối ưu hóa) để có thể cho ra hiệu quả cao hơn. Ngoài ra, khả năng thực thi phương pháp trong nghiên cứu này còn thô sơ. Nếu có điều kiện chúng tôi sẽ thực nghiệm trên các tập dữ liệu lớn hơn để có kết quả chính xác nhất. Cuối cùng, từ phương pháp này, chúng tôi có thể xây dựng 1 công cụ phân tích mã độc với khả năng phát hiện mã độc nhanh chóng hoặc 1 công cụ hổ trợ kiểm thử phần mềm với khả năng tìm kiếm các chức năng với tốc độ cao.

TÀI LIỆU THAM KHẢO

Tiếng Anh:

[1] Yaniv David, Nimrod Partush, and Eran Yahav (2016), "Statistical similarity of binaries", *Acm sigplan notices*, 51 (6), pp. 266–280.

[2] Yaniv David, Nimrod Partush, and Eran Yahav, "Similarity of binaries through re-optimization", in: *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017, pp. 79–94.

[3] Steven HH Ding, Benjamin CM Fung, and Philippe Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization", in: *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 472–489.

[4] Qian Feng et al., "Scalable graph-based bug search for firmware images", in: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[5] Irfan Ul Haq and Juan Caballero (2021), "A survey of binary code similarity", *ACM Computing Surveys (CSUR)*, 54 (3), pp. 1–38.

[6] Luca Massarelli et al., "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis", in: *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019, pp. 1–11.

[7] Luca Massarelli et al., "Safe: Self-attentive function embeddings for binary similarity", in: *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, Springer, 2019, pp. 309–329.

[8] Noam Shalev and Nimrod Partush, "Binary similarity detection using machine learning", in: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, 2018, pp. 42–47.

[9] Donghai Tian et al. (2020), "BinDeep: A Deep Learning Approach to Binary Code Similarity Detection", *Expert Systems with Applications*, p. 114348, DOI: 10.1016/j.eswa.2020.114348, URL: https://doi.org/10.1016/j.eswa.2020.114348.

[10] vul337 Team, *BinaryCorp*, https://cloud.vul337.team:9443/s/cxnH8DfZTADLKCs.

[11] Hao Wang et al., "jTrans: jump-aware transformer for binary code similarity detection", in: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 1–13.

[12] Wikipedia contributors, *BERT (language model)*, https://en.wikipedia.org/wiki/BERT_(language_model).

[13] Xiaojun Xu et al., "Neural network-based graph embedding for cross-platform binary code similarity detection", in: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 363–376.

[14] Zeping Yu et al., "Order matters: Semantic-aware neural networks for binary code similarity detection", in: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 01, 2020, pp. 1145–1152.