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

TRỊNH MINH HOÀNG
VŨ TRUNG KIÊN

KHÓA LUẬN TỐT NGHIỆP

PHÁT HIỆN TỰ ĐỘNG LỖ HỔNG HỢP ĐỒNG THÔNG MINH DỰA TRÊN PHƯƠNG PHÁP HỌC SÂU

AUTOMATIC DETECTION OF SMART CONTRACT VULNERABILITIES BASED ON DEEP LEARNING METHODS

KỸ SƯ NGÀNH AN TOÀN THÔNG TIN

TP. Hồ Chí Minh, 2023

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

TRỊNH MINH HOÀNG - 19521548
VŨ TRUNG KIÊN - 19521722

KHÓA LUẬN TỐT NGHIỆP

PHÁT HIỆN TỰ ĐỘNG LỖ HỔNG HỢP ĐỒNG THÔNG MINH DỰA TRÊN PHƯƠNG PHÁP HỌC SÂU

AUTOMATIC DETECTION OF SMART CONTRACT VULNERABILITIES BASED ON DEEP LEARNING METHODS

KỸ SƯ NGÀNH AN TOÀN THÔNG TIN

GIẢNG VIÊN HƯỚNG DẪN:
TS. Phạm Văn Hậu
ThS. Nghi Hoàng Khoa

TP.Hồ Chí Minh - 2023

LỜI CẢM ƠN

Trong quá trình nghiên cứu và hoàn thành khóa luận, nhóm đã nhận được sự định hướng, giúp đỡ, các ý kiến đóng góp quý báu và những lời động viên của các giáo viên hướng dẫn và giáo viên bộ môn. Nhóm xin bày tỏ lời cảm ơn tới thầy Phạm Văn Hậu, thầy Nghi Hoàng Khoa, thầy Phan Thế Duy và các anh nghiên cứu viên tại Phòng Thí nghiệm An toàn Thông tin InsecLab đã tận tình trực tiếp hướng dẫn, giúp đỡ trong quá trình nghiên cứu.

Nhóm xin gửi lời cảm ơn đến gia đình và bạn bè đã động viên, đóng góp ý kiến trong quá trình làm khóa luận

Nhóm cũng chân thành cảm ơn các quý thầy cô trường Đại học Công nghệ Thông tin - ĐHQG TP.HCM, đặc biệt là các thầy cô khoa Mạng máy tính và Truyền thông, các thầy cô thuộc bộ môn An toàn Thông tin đã giúp đỡ nhóm.

Trịnh Minh Hoàng **Vũ Trung Kiên**

MỤC LỤC

DANH MỤC CÁC KÝ HIỆU, CÁC CHỮ VIẾT TẮT

AdaBoost	Adaptive Boosting
BERT	Bidirectional Encoder Representations from Transformers
BiLSTM	Bi-directional Long Short Term Memory
BS	Batch Size
CFG	Control Flow Graph
DL	Deap learning
DNN	Deep Neural Network
ETH	Đồng tiền mã hóa Ethereum
EVM	Ethereum Virtual Machine
GNN	Graph Neural Network
Gas	Chi phí thanh toán Smart Contract
KNN	K-Nearest Neighbor
M1	Mô hình học sâu đa phương thức BERT và Bi-LSMT
M2	Mô hình học sâu đa phương thức BERT và GNN
M3	Mô hình học sâu đa phương thức Bi-LSMT và GNN
ML	Machine learning
SC	Smart Contract
Smart Contract	Hợp đồng thông minh
TOD	Transaction Ordering Dependence
XGBoost	Extreme Gradient Boosting
SVM	Support Vector Machine

DANH MỤC CÁC HÌNH VẼ

DANH MỤC CÁC BẢNG BIỂU

TÓM TẮT KHÓA LUẬN

Tính cấp thiết của đề tài nghiên cứu:

Smart Contract (Hợp đồng thông minh) là các ứng dụng phi tập trung chạy trên Blockchain. Trong những năm gần đây, một số lượng rất lớn các hợp đồng thông minh đã được triển khai trên Ethereum. Bất cứ một ứng dụng nào được triển khai thì đi kèm với nó chính là vấn đề bảo mật. Tương tự với hợp đồng thông minh, cùng với tính chất bất biến và công khai của Blockchain. Các lỗi bảo mật trong các hợp đồng thông minh đã gây tổn thất rất lớn vì các ứng dụng được triển khai thường về giao dịch tiền bạc và tài sản, ngoài ra gây phá hủy sự ổn định sinh thái trên Blockchain.

Các phương pháp phát hiện lỗ hổng trong hợp đồng thông minh hiện nay thường chủ yếu dựa trên việc thực thi hoặc phân tích cú pháp, việc này thường mất khá nhiều thời gian. Vì vậy, trong đề tài này, chúng tôi đề xuất phương pháp phát hiện lỗ hổng dựa trên các kĩ thuật học sâu.

Trong đề tài này, chúng tôi trình bày một phương pháp toàn diện để phát hiện lỗ hổng hiệu quả trong các hợp đồng thông minh Ethereum bằng cách sử dụng phương pháp học sâu đa phương thức (multimodal deep learning), chúng tôi gọi mô hình này là VulnSense. Phương pháp mà chúng tôi đề xuất kết hợp 3 loại đặc trưng của hợp đồng thông minh, bao gồm: mã nguồn, mã opcodes và biểu đồ luồng điều khiển (CFG) được trích xuất từ bytecode. Chúng tôi sử dụng các mô hình BERT, Bi-LSTM và GNN để trích xuất và phân tích các đặc trưng. Lớp cuối cùng của phương pháp đa phương thức của chúng tôi là một lớp kết nối đầy đủ (fully connected layer) dùng để dự đoán lỗ hổng trong các hợp đồng thông minh Ethereum. Chúng tôi giải quyết các hạn chế của các phương pháp phát hiện lỗ hổng dựa trên kỹ thuật học sâu hiện có cho các hợp đồng thông minh, thường dựa vào một loại đặc trưng hoặc mô hình duy nhất, dẫn đến độ

chính xác và hiệu quả bị giới hạn. Kết quả thực nghiệm cho thấy phương pháp đề xuất của chúng tôi đạt được kết quả vượt trội so với các phương pháp học sâu đơn phương thức, chứng minh tính hiệu quả và tiềm năng của các phương pháp học sâu đa phương thức trong việc phát hiện lỗ hổng trong các hợp đồng thông minh.

CHƯƠNG 1. TỔNG QUAN

Chương này giới thiệu về vấn đề và các nghiên cứu liên quan. Đồng thời, trong chương này chúng tôi cũng trình bày phạm vi và cấu trúc của Khóa luận.

1.1. Giới thiệu vấn đề

Từ khoá Blockchain xuất hiện rất phổ biến trong thời đại công nghệ 4.0. Từ mục đích tốt cho đến mục đích xấu, ứng dụng của Blockchain đang ngày càng xuất hiện nhiều với vô vàn từ khoá. Điển hình nhất là các hợp đồng thông minh được triển khai trên Ethereum. Các hợp đồng thông minh được lập trình bằng ngôn ngữ Solidity, đây mặc dù là ngôn ngữ mới được phát triển trong thời gian gần đây, tuy nhiên bản thân ngôn ngữ này vẫn tồn tại nhiều lỗ hổng đã được Zou và cộng sự [30] chỉ ra. Khi triển khai trên hệ thống blockchain, hợp đồng thông minh thường thực hiện các giao dịch liên quan đến tiền mã hoá (crypto currency), cụ thể là đồng ether (ETH), cùng với tính chất không thể thay đổi và tính chất công khai của Blockchain, việc tồn tại lỗ hổng trong hợp đồng thông minh được triển khai trên hệ sinh thái Blockchain khiến kẻ tấn công có thể lợi dụng các hợp đồng thông minh có chứa lỗ hổng gây ảnh hưởng đến tài sản của các cá nhân, tổ chức cũng như sự ổn định trong hệ sinh thái Blockchain.

Cuộc tấn công DAO [21] được trình bày bởi Mehar và các cộng sự là một ví dụ rõ ràng về mức độ nghiêm trọng của các lỗ hổng này, vì nó đã gây thiệt hại đáng kể lên đến 50 triệu đô la Mỹ. Để giải quyết những vấn đề này, Kushwaha và cộng sự [17] đã tiến hành một cuộc khảo sát nghiên cứu về các lỗ hổng khác nhau trong hợp đồng thông minh và cung cấp một cái nhìn tổng quan về các công cụ hiện có để phát hiện và phân tích các lỗ hổng này. Để gia tăng hiệu suất phát hiện các lỗ hổng so với việc đọc từng dòng mã nguồn hợp đồng thông minh,

các nhà phát triển đã tạo ra một số công cụ để phát hiện các lỗ hổng trong mã nguồn hợp đồng thông minh, như Oyente [20], Slither [7], Conkas [22], Mythril [4], Securify [25],... Các công cụ này sử dụng phương pháp phân tích tĩnh hoặc phân tích động, nhưng có thể không bao phủ tất cả các luồng thực thi, dẫn đến các kết quả âm tính giả (False negatives). Bên cạnh đó, việc khám phá tất cả các luồng thực thi trong hợp đồng thông minh gây hao phí lượng lớn thời gian.

Đồng thời, sự xuất hiện các phương pháp học máy (machine learning - ML) trong việc phát hiện các lỗ hổng trong các ứng dụng phần mềm cũng được khám phá. Điều này hoàn toàn có thể áp dụng cho hợp đồng thông minh, nhiều công cụ và nghiên cứu khoa học về phát hiện lỗ hổng bảo mật dựa trên phương pháp học máy trong hợp đồng thông minh đã được phát triển như ESCORT của Lutz [19] , ContractWard của Wang [26] và nghiên cứu của Qian [23]. Các phương pháp phát hiện lỗ hổng dựa trên phương pháp học máy đã cải thiện đáng kể hiệu suất so với các phương pháp phân tích tĩnh và phân tích động đã được chỉ ra trong nghiên cứu của Jiang và cộng sự [13]. Tuy nhiên, nghiên cứu hiện tại vẫn còn một số hạn chế, chẳng hạn như chỉ sử dụng một loại đặc trưng duy nhất của hợp đồng thông minh làm đầu vào cho mô hình học máy. Ví dụ, hợp đồng thông minh có thể được biểu diễn và phân tích thông qua mã nguồn bằng cách sử dụng các mô hình xử lý ngôn ngữ tự nhiên (NLP) như Bi-GRU trong nghiên cứu của Khodadadi và cộng sự [15], nghiên cứu của Chen và cộng sự [3] sử dụng runtime bytecode của hợp đồng thông minh được công khai trên Ethereum. Trong khi đó, để phát hiện lỗ hổng, Wang và cộng sự [26] sử dụng các opcode được trích xuất bằng công cụ Solc [24] (trình biên dịch Solidity) dựa trên mã nguồn hoặc bytecode. Trên thực tế, các phương pháp như vậy được gọi là mô hình đơn phương thức (monomodal hoặc unimodal), mô hình này chỉ xử lý một loại dữ liệu, các mô hình này đã được nghiên cứu rộng rãi và đóng góp đáng kể cho các lĩnh vực như thị giác máy tính và xử lý ngôn ngữ tự nhiên và an ninh mạng. Mặc dù có hiệu suất đáng chú ý nhưng các mô hình đơn phương thức vẫn có những hạn chế do chỉ có một góc nhìn duy nhất đối với đối tượng

dữ liệu, dẫn đến sự phát triển của các mô hình đa phương thức (multimodal) được Jabeen và cộng sự trình bày tổng quan [11]. Cụ thể, các mô hình này tận dụng nhiều mô hình học máy khác nhau với nhiều loại đầu vào khác nhau của một đối tượng để học các biểu diễn của đối tượng đầy đủ và toàn diện hơn.

Trong thời gian gần đây, các mô hình phát hiện lỗ hổng đa phương thức trong hợp đồng thông minh là một lĩnh vực nghiên cứu mới đang phát triển, kết hợp các kỹ thuật khác nhau để xử lý dữ liệu đa dạng, bao gồm mã nguồn, bytecode và opcode, nhằm tăng cường độ chính xác và đáng tin cậy của các hệ thống trí tuệ nhân tạo.

Nhiều nghiên cứu đã chứng minh hiệu quả của việc sử dụng các mô hình học sâu đa phương thức để phát hiện lỗ hổng trong hợp đồng thông minh. Ví dụ, Yang và cộng sự [14] đã đề xuất một mô hình trí tuệ nhân tạo đa phương thức (multimodal AI) kết hợp mã nguồn, bytecode và luồng thực thi để phát hiện lỗ hổng trong hợp đồng thông minh với độ chính xác cao. Chen và cộng sự [15] đã đề xuất một mô hình lai đa phương thức mới có tên là HyMo Framework, kết hợp các kỹ thuật phân tích tĩnh và phân tích động để phát hiện lỗ hổng trong hợp đồng thông minh. Framework của họ sử dụng nhiều phương pháp, vượt trội hơn các phương pháp khác trên một số tập dữ liệu thử nghiệm.

Nhận thấy rằng các đặc trưng này phản ánh chính xác hợp đồng thông minh và sự xuất hiện của mô hình đa phương thức, chúng tôi đã quyết định sử dụng mô hình đa phương thức để xây dựng công cụ phát hiện lỗ hổng trên hợp đồng thông minh có tên là VulnSense. Trong đề tài này, chúng tôi sử dụng 3 loại đặc trưng của hợp đồng thông minh gồm mã nguồn, opcode và CFG được tạo ra từ bytecode. Với mỗi đặc trưng lần lượt tương ứng với một mô hình khác nhau là Bidirectional Encoder Representations from Transformers (BERT), mô hình BiLSTM và mô hình Graph Neural Network (GNN). Kết quả đầu ra của các mô hình này được kết hợp bằng một lớp Concatenate để tạo ra một kết quả duy nhất. VulnSense của chúng tôi được huấn luyện để có khả năng nhận diện các hợp đồng thông minh có lỗ hổng Reentrancy, Arithmetic, cũng như những hợp

đồng không có 2 lỗ hổng kể trên.

1.2. Giới thiệu những nghiên cứu liên quan

1.2.1. Các phương pháp truyền thống

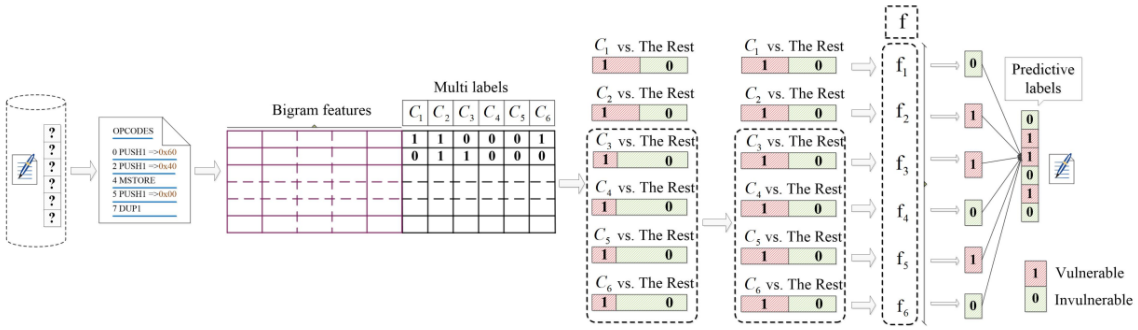
Các phương pháp truyền thống để phát hiện lỗ hổng trong hợp đồng thông minh liên quan đến phân tích tĩnh và phân tích động. Kushwaha và cộng sự [16] đã liệt kê một danh sách các công cụ sử dụng các kỹ thuật phân tích tĩnh như phân tích mã nguồn và bytecode, cũng như các kỹ thuật phân tích động như phân tích luồng điều khiển trong quá trình thực thi hợp đồng thông minh.

Các công cụ phân tích tĩnh như Oyente [20], Slither [7], Conkas [22], Mythril [4] và Securify [25] thường phân tích bytecode hoặc mã nguồn để phát hiện các loại lỗ hổng đa dạng trong hợp đồng thông minh, bao gồm reentrancy, time manipulation, arithmetic, TOD,... Trong khi đó, ContractFuzzer được Jiang và cộng sự [12] phát triển sử dụng phương pháp phân tích động gọi là fuzzing, tạo ra các đầu vào cho bytecode bằng cách sử dụng Application Binary Interface (ABI) của hợp đồng thông minh. Mặc dù hữu ích, các phương pháp truyền thống này thường có độ chính xác thấp và tính linh hoạt hạn chế trong việc xác định lỗ hổng. Điều này là do sự phụ thuộc vào kiến thức chuyên gia và các mẫu cố định, cũng như việc thực hiện tốn thời gian và chi phí.

1.2.2. Các phương pháp học máy

Từ sự phát triển mạnh mẽ việc sử dụng các phương pháp học máy trong việc phát hiện lỗ hổng phần mềm. Đã có nhiều nghiên cứu về việc sử dụng các phương pháp học máy để phát hiện lỗ hổng trong hợp đồng thông minh được công bố như Wang và cộng sự [26] đã đề xuất một công cụ mang tên ContractWard sử dụng các mô hình học máy, bao gồm: Random Forest, XGBoost, AdaBoost, SVM và K-Nearest Neighbors, để phân loại 6 loại lỗ hổng trên hợp đồng thông

minh ở mức opcode bằng cách trích xuất các đặc trưng bigram từ opcode đơn giản hóa. Tổng quan phương pháp phát triển ContractWard được nhóm tác giả thể hiện trong **Hình 1.1**. Bắt đầu từ việc chuyển đổi hợp đồng thông minh thành các opcodes, sau đó trích xuất bigram từ các opcodes này, cân bằng dữ liệu, gán nhãn và thực hiện phân loại hợp đồng thông minh.

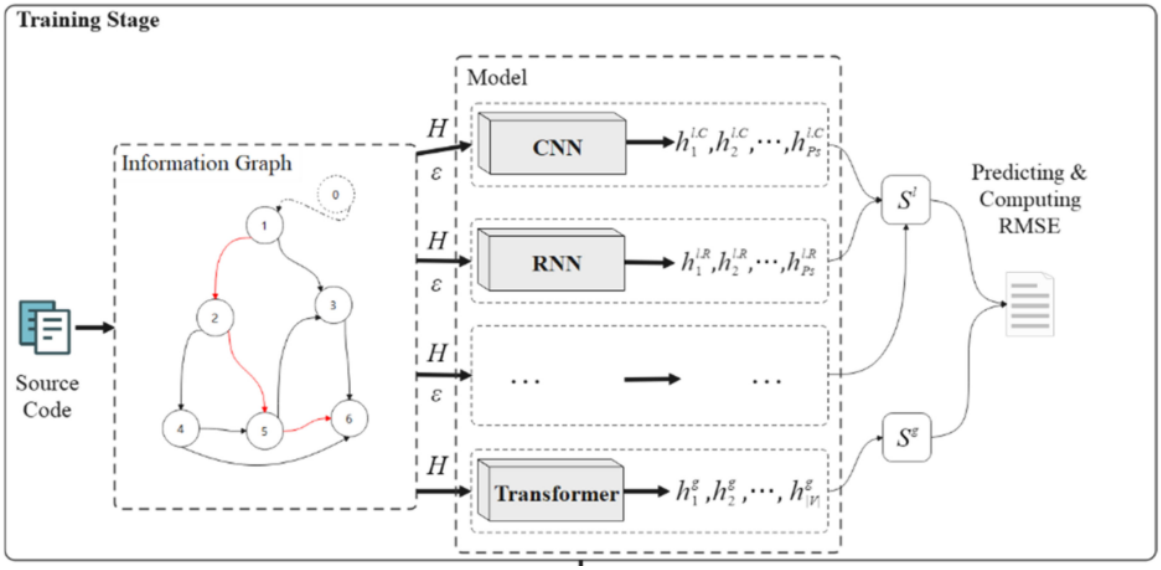


Hình 1.1: *Phương pháp thiết kế công cụ ContractWard*

Trong một nghiên cứu khác, Zhang, Lejun và cộng sự [29] đã sử dụng học kết hợp (Ensemble Learning) để phát triển một mô hình tích chập 7 tầng kết hợp các mô hình mạng thần kinh khác nhau, gồm: Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Recurrent Convolutional Network (RCN), Deep Neural Network (DNN), Gated Recurrent Unit Recurrent Neural Network (GRU), Bidirectional Gated Recurrent Unit Recurrent Neural Network (Bi-GRU), và Transformer dựa trên đặc trưng Graph của hợp đồng thông minh. **Hình 1.2** thể hiện đầy đủ kiến trúc mà Zhang và cộng sự đã xây dựng nên. Đặc trưng Graph của hợp đồng thông minh sẽ được trích xuất, đưa vào 7 mô hình cùng lúc. Sau đó, kết hợp lần lượt từng 2 mô hình với nhau và cuối cùng đưa ra được kết quả.

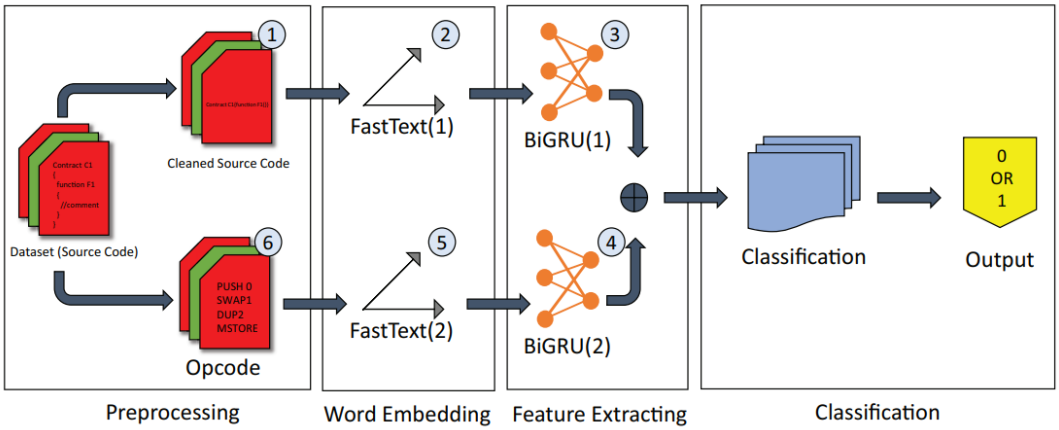
1.3. Mô hình đa phương thức

HyMo Framework [15] được giới thiệu bởi Chen và đồng nghiệp vào năm 2020 đã trình bày một mô hình học sâu đa phương thức dùng trong phát hiện lỗ hổng



Hình 1.2: *Kiến trúc mô hình của Zhang và cộng sự*

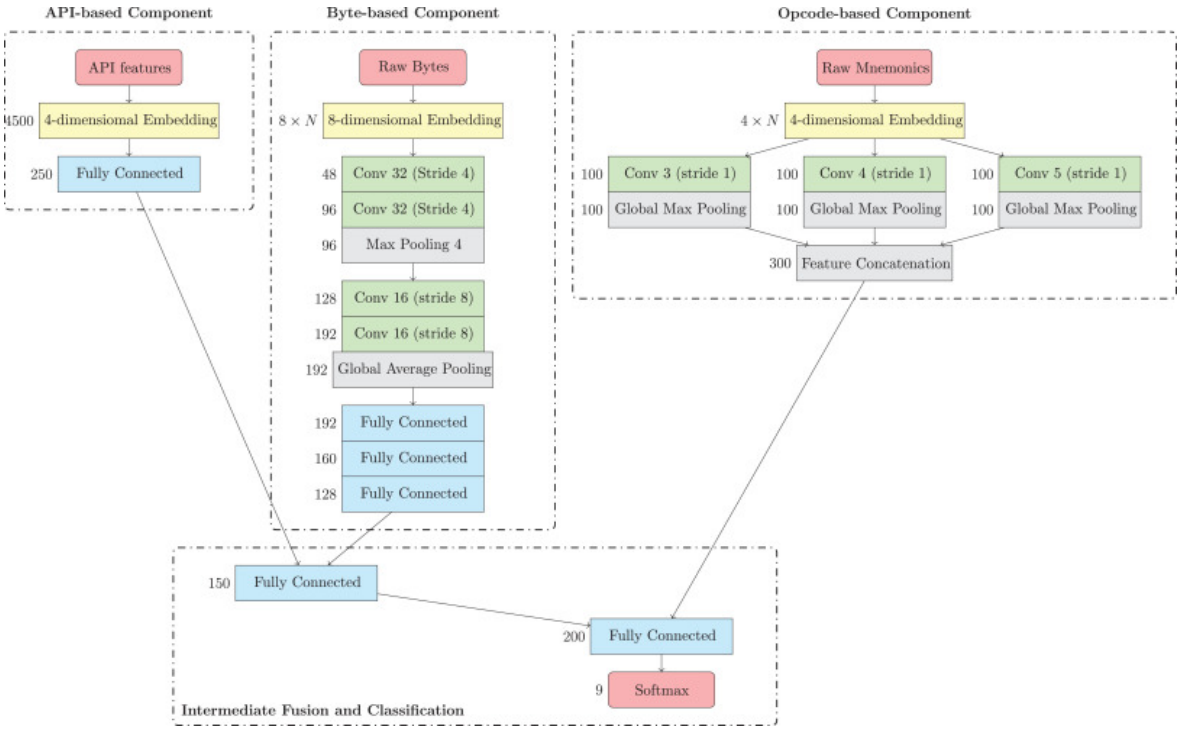
trong hợp đồng thông minh. **Hình 1.3** thể hiện các thành phần trong HyMo Framework. Framework này sử dụng 2 thuộc tính của hợp đồng thông minh gồm: mã nguồn, opcode. Sau khi tiền xử lý các thuộc tính, HyMo Framework dùng FastText cho word embedding và 2 mô hình Bi-GRU để trích xuất đặc trưng trên 2 thuộc tính này.



Hình 1.3: *Mô hình của HyMo Framework*

Một framework khác - HYDRA Framework, được đề xuất bởi Chen và đồng nghiệp [10], HYDRA sử dụng ba thuộc tính là API, bytecode và opcode làm đầu vào cho ba nhánh trong mô hình đa phương thức để phân loại phần mềm

độc hại.



Hình 1.4: *Mô hình của HYDRA Framework*

Hình 1.4 thể hiện chi tiết mô hình của HYDRA Framework. Mỗi nhánh xử lý các thuộc tính bằng cách sử dụng các mạng neural cơ bản, sau đó các đầu ra của những nhánh này được kết nối với nhau thông qua các lớp fully connected và cuối cùng thông qua hàm Softmax để nhận kết quả cuối cùng.

Kết quả của 2 mô hình này đều cho những kết quả khả quan về Accuracy, với HyMo đạt khoảng 0.79 và HYDRA đặt kết quả vượt bậc với khoảng 0.98. Với những kết quả đạt được, những nghiên cứu này đã chứng minh sức mạnh của các mô hình đa phương thức so với các mô hình đơn phương thức trong việc phân loại các đối tượng với nhiều thuộc tính.

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

1.4.1. Mục tiêu nghiên cứu

Ứng dụng mô hình đa phương thức học sâu để phát hiện được các lỗ hổng trong hợp đồng thông minh với thời gian phát hiện nhanh và độ chính xác cao hơn các công cụ hiện có.

1.4.2. Đối tượng nghiên cứu

Đối tượng nghiên cứu:

- Hợp đồng thông minh
- Lỗ hổng trong hợp đồng thông minh
- Mô hình học sâu
- Công cụ phát hiện lỗ hổng dựa trên học máy

1.4.3. Phạm vi nghiên cứu

Hợp đồng thông minh triển khai trên Ethereum được viết bằng ngôn ngữ Solidity. Phát hiện 3 loại nhãn gồm lỗ hổng Arithmetic, Reentrancy [18] và không có lỗ hổng. Nghiên cứu sử dụng mô hình đa phương thức học sâu làm thành phần chính xử lý đầu vào là các hợp đồng thông minh và đưa ra các dự đoán về lỗ hổng trong hợp đồng thông minh đó.

1.4.4. Cấu trúc khóa luận tốt nghiệp

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

- Chương 1: Giới thiệu tổng quan về đề tài của Khóa luận và những nghiên cứu liên quan.

- Chương 2: Trình bày cơ sở lý thuyết và kiến thức nền tảng liên quan đến đề tài.
- Chương 3: Trình bày chi tiết thiết kết mô hình học sâu đa phương thức - VulnSense.
- Chương 4: Trình bày thực 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 trình bày các cơ sở lý thuyết cần thiết cho khoá luận, bao gồm: hợp đồng thông minh, ngôn ngữ Solidity và các loại lổ hổng của ngôn ngữ Solidity khi triển khai hợp đồng thông minh, các đặc trưng của hợp đồng thông minh và các mô hình học sâu mà nhóm sử dụng trong khoá luận.

2.1. Hợp đồng thông minh

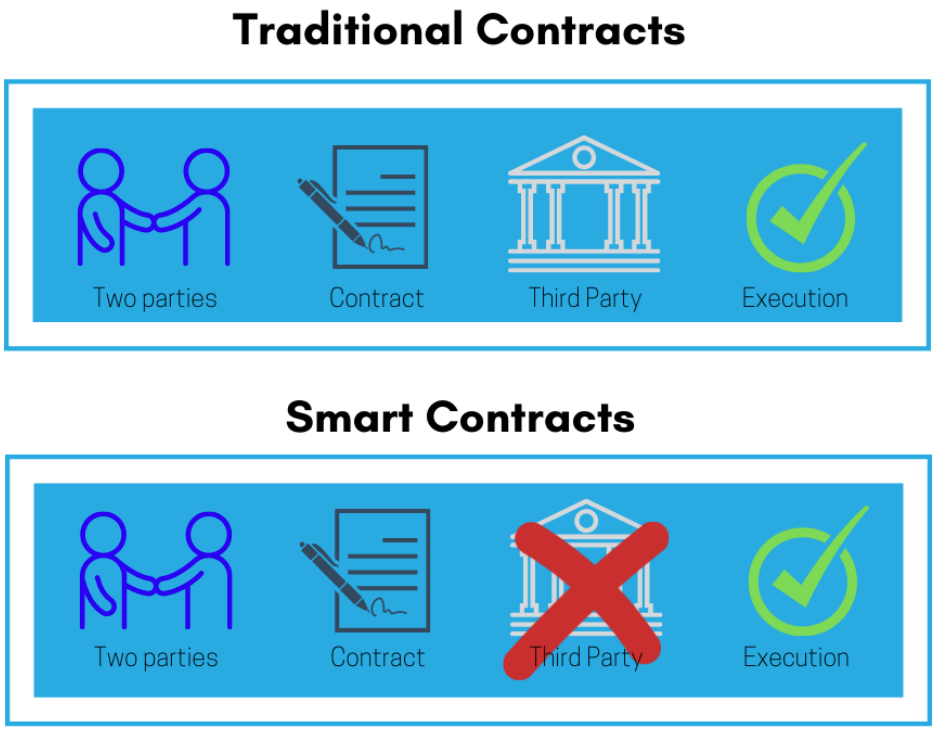
Hợp đồng thông minh là một khái niệm trong lĩnh vực Blockchain và crypto currency, đặc biệt phổ biến trên nền tảng Ethereum. Nó kết hợp cả lý thuyết và công nghệ để tạo ra các hợp đồng tự động, không cần đến sự can thiệp của bên thứ ba.

Cơ sở lý thuyết của hợp đồng thông minh xuất phát từ ý tưởng về hợp đồng truyền thống trong lĩnh vực pháp lý. Trong thực tế, một hợp đồng là một thỏa thuận giữa hai hoặc nhiều bên, đặt ra các điều kiện và cam kết cụ thể. Tuy nhiên, hợp đồng truyền thống phụ thuộc vào các tổ chức pháp lý và nhà trung gian để giám sát và thực thi (**Hình 2.1**).

Hợp đồng thông minh áp dụng cơ chế của Blockchain để tạo ra hợp đồng tự động, tự thực thi. Nó là một chương trình máy tính hoạt động trên nền tảng Blockchain, được viết bằng ngôn ngữ lập trình và chứa các điều kiện và hành động cụ thể. Khi một điều kiện trong hợp đồng thông minh được thỏa mãn, các hành động tương ứng sẽ được thực hiện tự động.

Một số tính chất của hợp đồng thông minh bao gồm:

1. Tính tự động hóa: Hợp đồng thông minh loại bỏ sự phụ thuộc vào sự can thiệp của bên thứ ba trong việc thực thi hợp đồng. Các điều khoản và điều



Hình 2.1: *Hợp đồng thông minh và hợp đồng thông thường*

kiện được định nghĩa rõ ràng và sẽ tự động thực hiện khi các điều kiện được đáp ứng.

2. Tính phân phối: Hợp đồng thông minh được sao chép và phân phối đến tất cả các nút trong mạng Blockchain. Điều này đảm bảo tính toàn vẹn và bảo mật của hợp đồng, giúp hợp đồng không thể bị thay đổi hay xóa bỏ một cách trái phép.

3. Tính bất biến: Sau khi triển khai, hợp đồng thông minh không thể bị chỉnh sửa.

4. Tính tin cậy: Hợp đồng thông minh hoạt động theo cách mà nó được lập trình, không có sự can thiệp của con người, không có bên thứ ba. Điều này đảm bảo tính tin cậy và tránh những mâu thuẫn hoặc phiên giải sai trong quá trình thực thi hợp đồng. Hơn nữa, hợp đồng thông minh được duy trì và thực thi bởi tất cả các nút trên mạng bằng cơ chế đồng thuận, loại bỏ

quyền kiểm soát từ tay bất kỳ một bên nào.

5. Tính minh bạch: Hợp đồng thông minh luôn được lưu trữ trên một sổ cái phân phối công khai gọi là Blockchain, nhờ đó mã nguồn là công khai với mọi người, cho dù họ có tham gia vào hợp đồng thông minh hay không.

6. Tính tiết kiệm thời gian và chi phí: Hợp đồng thông minh giảm thiểu sự phụ thuộc vào các bên trung gian và giúp tiết kiệm thời gian và chi phí trong việc thực hiện hợp đồng. Không cần phải tìm kiếm và truy cập đến các nguồn thông tin bên ngoài để xác minh và thực thi hợp đồng.

Tuy nhiên, hợp đồng thông minh không thể thay thế hoàn toàn hợp đồng truyền thống và không phải tất cả các loại hợp đồng đều phù hợp với hợp đồng thông minh. Điều này phụ thuộc vào tính chất và phạm vi của hợp đồng, cũng như các yếu tố pháp lý và quy định tại quốc gia hoặc khu vực tương ứng.

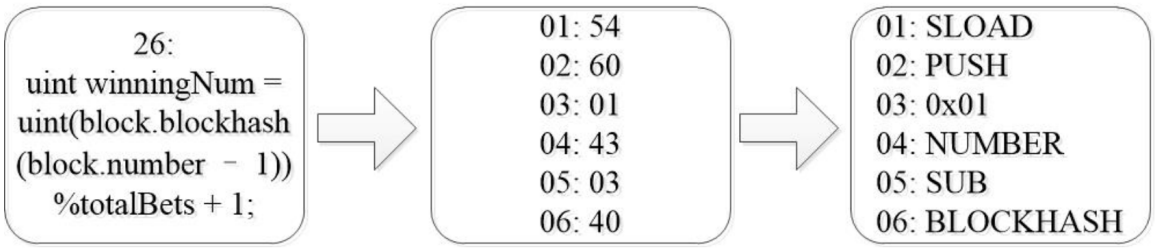
2.1.1. Ngôn ngữ Solidity

Solidity là một ngôn ngữ lập trình chính trong việc phát triển hợp đồng thông minh trên nền tảng Ethereum. Được phát triển bởi Gavin Wood vào năm 2014 [27], Solidity giúp xây dựng ứng dụng phi tập trung (Decentralized Applications - DApps) và thực hiện các hợp đồng tự động trên blockchain Ethereum.

Solidity có cú pháp tương tự như ngôn ngữ lập trình C++, với một số tùy chỉnh và đặc điểm riêng. Sau khi được biên dịch, bytecode của hợp đồng thông minh sẽ được tạo và thực thi bên trong máy ảo Ethereum (EVM). Quá trình này được thể hiện tổng quan trong **Hình 2.2**. Vì có một ánh xạ một-một giữa hoạt động blockchain và biểu diễn bytecode, nên việc phân tích luồng kiểm soát của hợp đồng thông minh ở bytecode là khả thi hơn. Khi được kích hoạt, việc thực hiện hợp đồng thông minh là tự chủ và có hiệu lực đối với tất cả các bên tham gia.

Bản thân EVM là một máy ảo dựa trên ngăn xếp có kích thước từ 256 đến 1024 bits. Bộ nhớ áp dụng một mô hình địa chỉ từ. Khi một hợp đồng thông

Hình 2.2: *Biên dịch Hợp đồng thông minh*



Hình 2.3: *Tương quan giữa source code, bytecode và opcode*

minh triển khai trên Blockchain, nó cần gas để hoạt động. Gas là đơn vị được sử dụng để thanh toán chi phí tính toán của những người khai thác chạy hợp đồng thông minh hoặc giao dịch và được thanh toán bằng Ether (ETH). **Hình 2.3** thể hiện 3 bước để triển khai hợp đồng thông minh trên EVM. Đầu tiên, mã nguồn sẽ được viết bằng ngôn ngữ bậc cao, ví dụ như Solidity. Thứ 2, mã nguồn này sẽ được trình biên dịch biên dịch thành các bytecodes. Bytecode (hay còn được gọi là EVM code) là một chuỗi thập lục phân. Cuối cùng, các bytecodes được chuyển thành các operation codes (opcodes).

2.1.1.1. Bytecode

Bytecode là một chuỗi các mã lệnh thập lục phân (hexadecimal) được sinh ra từ các ngôn ngữ lập trình bậc cao như C/C++, Python và không ngoại lệ đối với Solidity. Trong ngữ cảnh triển khai hợp đồng thông minh bằng Solidity, bytecode là phiên bản biên dịch của mã nguồn hợp đồng thông minh và được

thực thi trên môi trường Blockchain.

Bytecode đại diện cho các hành động mà hợp đồng thông minh có thể thực hiện. Nó chứa các câu lệnh và thông tin cần thiết để thực hiện các chức năng của hợp đồng. Bytecode thường được chuyển đổi từ ngôn ngữ Solidity hoặc các ngôn ngữ khác sử dụng trong việc phát triển hợp đồng thông minh.

Bytecode khi được triển khai trên Ethereum được chia thành 2 loại: creation bytecode và runtime bytecode.

1. Creation bytecode chỉ được chạy duy nhất 1 lần khi hợp đồng thông mình được triển khai lên hệ thống. Creation bytecode có nhiệm vụ thiết lập trạng thái ban đầu của hợp đồng thông minh như các biến khởi tạo, hàm constructor. Creation bytecode sẽ ko nằm trong thành phần hợp đồng thông minh được triển khai lên mạng Blockchain.

2. Runtime bytecode chứa các thông tin có thể thực thi của hợp đồng thông minh và được triển khai lên mạng Blockchain.

Một khi hợp đồng thông minh đã được biên dịch thành bytecode, nó có thể được triển khai lên Blockchain và được thực thi bởi các nút trong mạng. Các nút trong mạng sẽ thực hiện các câu lệnh trong bytecode để xác định hành vi và tương tác của hợp đồng thông minh.

Bytecode có tính xác định cao và không thể thay đổi sau khi đã được biên dịch. Nó cung cấp cho các bên tham gia trong mạng Blockchain khả năng xem và kiểm tra hợp đồng thông minh trước khi triển khai.

2.1.1.2. Opcode

Opcode trong hợp đồng thông minh là các mã lệnh thực thi được sử dụng trong môi trường blockchain để thực hiện các chức năng của hợp đồng thông minh. Opcode là các lệnh máy cấp thấp được sử dụng để điều khiển quá trình thực thi của hợp đồng trên một máy ảo blockchain như Ethereum Virtual Machine (EVM).

Mỗi opcode đại diện cho một tác vụ cụ thể trong hợp đồng thông minh, bao gồm các phép toán logic, phép tính số học, quản lý bộ nhớ, truy cập vào dữ liệu, gọi và tương tác với các hợp đồng khác trong mạng Blockchain, và nhiều tác vụ khác.

Opcode định nghĩa các hành động mà hợp đồng thông minh có thể thực hiện và quy định cách mà các dữ liệu và trạng thái của hợp đồng được xử lý. Các opcode được liệt kê và xác định trong biểu diễn bytecode của hợp đồng thông minh.

Việc sử dụng các opcodes cung cấp sự linh hoạt và tiêu chuẩn hóa cho việc thực hiện các chức năng của hợp đồng thông minh. Opcode giúp đảm bảo tính nhất quán và an toàn trong quá trình thực thi của hợp đồng trên mạng blockchain, và đóng vai trò quan trọng trong việc xác định hành vi và logic của hợp đồng thông minh.

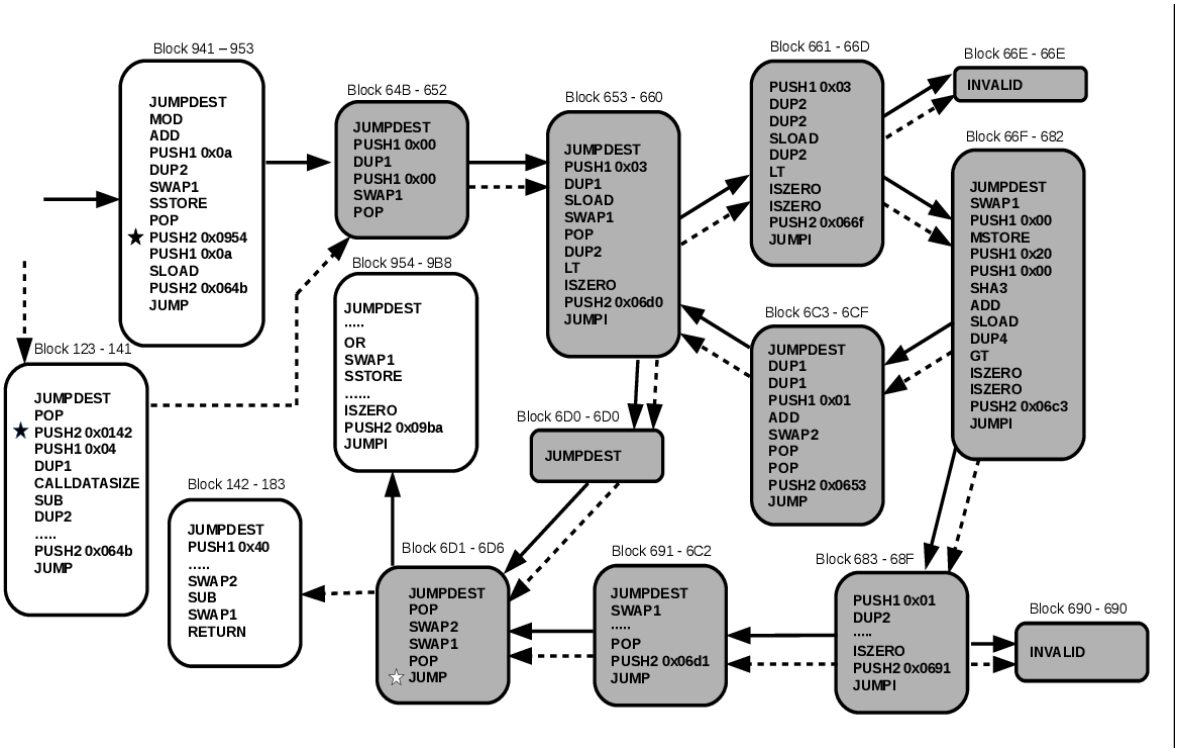
Theo Ethereum Yellow Paper [28], sẽ có 135 opcodes đại diện cho 10 chức năng bao gồm: namely, stop and arithmetic operations, comparison and bit-wise logic operations, SHA3 operations, environment information operations, block information operations, stack, memory, storage and flow operations, push operations, exchange operations, logging operations and system operations.

Bảng 2.1: *Giới thiệu về Opcodes trong ngôn ngữ Solidity*

Loại	Opcode
Quản lý Stack	POP, PUSH, UP, SWAP
Arithmetic/ Comparison/ Bitwise	ADD, SUB, GT, LT, AND, OR
Thông tin môi trường	CALLER, CALL VALUE, NUMBER
Opcode kiểm soát bộ nhớ	LOAD, STORE, MSSTORE 8, MSIZE
Quản lý bộ nhớ	LOAD, STORE
Opcodes liên quan đến bộ đếm chương trình	JUMP, JUMPTO, JUMPIF, JUMPSUB, JUMPSUBV, JUMPDEST
STOP Opcodes	STOP, RETURN, REVERT, INVALID, SELF, DESTRUCT

2.1.1.3. Control Flow Graph

Control Flow Graph (CFG) là một dạng dữ liệu mạnh mẽ trong phân tích mã nguồn Solidity để hiểu và tối ưu hóa luồng điều khiển của chương trình được trích xuất từ bytecode của hợp đồng thông minh. CFG giúp xác định cấu trúc



Hình 2.4: *Control flow graph của một Hợp đồng thông minh viết bằng ngôn ngữ Solidity*

và tương tác giữa các khối mã trong chương trình, từ đó cung cấp thông tin quan trọng về cách chương trình thực hiện và liên kết giữa các phần tử trong luồng điều khiển. **Hình 2.4** thể hiện một ví dụ về biểu diễn CFG của hợp đồng thông minh. Cụ thể, CFG xác định các điểm nhảy và điều kiện trong bytecode Solidity để xây dựng đồ thị dòng điều khiển. Đồ thị này mô tả các khối cơ bản và các nhánh điều khiển trong chương trình, từ đó tạo ra một hình dung rõ ràng về cấu trúc của chương trình Solidity. Với CFG, ta có thể tìm ra các vấn đề tiềm ẩn trong chương trình như vòng lặp vô hạn, điều kiện không chính xác, hay các lỗ hổng bảo mật. Bằng cách kiểm tra đường dẫn điều khiển trong CFG, ta có thể phát hiện được các lỗi logic hoặc khả năng xảy ra các tình huống không mong muốn trong chương trình Solidity. Ngoài ra, CFG còn hỗ trợ tối ưu hóa mã nguồn Solidity. Bằng cách phân tích và hiểu cấu trúc luồng điều khiển, ta có thể đề xuất các cải tiến về hiệu suất và tính đúng đắn cho chương trình Solidity. Điều này đặc biệt quan trọng trong việc phát triển các hợp đồng thông minh

trên nền tảng Ethereum, nơi hiệu suất và bảo mật đóng vai trò quan trọng. Tóm lại, CFG là một dạng biểu diễn mạnh mẽ cho phép ta phân tích, hiểu và tối ưu hóa luồng điều khiển trong chương trình Solidity. Bằng cách xây dựng đồ thị dòng điều khiển và phân tích cấu trúc luồng điều khiển, ta có thể tìm ra lỗi, kiểm tra tính đúng đắn và tối ưu hóa mã nguồn Solidity để đảm bảo hiệu suất và tính bảo mật.

2.1.2. Các lỗ hổng trong hợp đồng thông minh

Trong phần này, chúng tôi sẽ trình bày về hai lỗ hổng trong hợp đồng thông minh mà nhóm dùng để phát hiện trong khóa luận này. Bên cạnh đó, chúng tôi sẽ trình bày thêm một số lỗ hổng đáng chú ý trong hợp đồng thông minh.

2.1.2.1. Reentrancy

Reentrancy được coi là một trong những lỗi hổng nghiêm trọng nhất trong hợp đồng thông minh. Lỗ hổng này xuất hiện trong hợp đồng thông minh có thể dẫn đến phá hủy hoặc đánh cắp toàn bộ ETH trên hợp đồng thông minh đó. Mehar và cộng sự [21] đã trình bày nghiên cứu về cuộc tấn công DAO - một cuộc tấn công vào các lỗ hổng trong hợp đồng thông minh có sự ảnh hưởng lớn bởi lỗ hổng Reentrancy. Nghiên cứu đã chỉ ra các điểm yếu trên hợp đồng thông minh đã bị khai thác như thế nào cũng như chỉ ra cuộc tấn công này đã làm thiệt hại hơn 60 triệu USD vào tháng 6 năm 2016.

Hình 2.5 bên dưới là một đoạn code ví dụ có chứa lỗ hổng Reentrancy. Lỗ hổng này có thể xảy ra khi một hàm gọi đến một hợp đồng không đáng tin cậy được kiểm soát bởi kẻ tấn công thì chúng có thể gọi đệ quy lại hàm ban đầu và thực hiện lại hàm đó. Cụ thể, nếu một hợp đồng cho phép rút ETH tồn tại hàm chứa lỗ hổng Reentrancy, kẻ tấn công có thể gọi đệ quy hàm bị lỗi để rút hết tất cả số ETH có trong hợp đồng. Một cuộc tấn công Reentrancy xảy ra khi kẻ tấn công rút tiền từ mục tiêu bằng cách gọi đệ quy chức năng rút tiền mục tiêu, như trường hợp của DAO. Khi hợp đồng thông minh không cập nhật trạng thái số

dư người dùng trước khi gửi tiền, kẻ tấn công có thể liên tục gọi chức năng rút tiền để rút hết tiền trong ví hợp đồng. Bất cứ khi nào kẻ tấn công nhận ETH, hợp đồng thông minh của kẻ tấn công tự động gọi fallback function, nơi lại gọi hàm rút tiền một lần nữa. Lúc này cuộc tấn công đi vào vòng lặp đệ quy cho đến khi nào ví của hợp đồng không còn ETH.

```
pragma solidity ^0.4.18;
contract Reentrancy
{
    mapping (address -> uint256) public balances;

    function withdraw(uint _amount) public
    {
        if (balance[msg.sender] >= _amount) public
        {
            if (msg.sender.call.value(_amount)()) {
            _amount;
            }  // Khai thác lổ hỗng Reentrancy tại đây: rút
 ↪  tiền khỏi tài khoản
        }
        balances[msg.sender] -= _amount; // Trừ số tiền đã rút
 ↪  trong tài khoản. Nếu khai thác lỗi trên thành công, hàm này
 ↪  sẽ không bao giờ được thực thi.
    }
}
```

Hình 2.5: *Hợp đồng chứa lỗi Reentrancy*

Lỗ hổng Reentrancy còn có thể thực hiện bằng cách kết hợp cơ chế của một số hàm trong hợp đồng thông minh khiến cho hàm withdraw() chạy vào vòng lặp đệ quy tại vị trí call.value() trong **Hình 2.5** bằng cách viết ra một hợp đồng thông minh khai thác như **Hình 2.6**. Việc này rõ ràng khiến tiền bị rút hết mà số tiền của kẻ tấn công vẫn không thay đổi.

```
pragma solidity ^0.4.18;
contract ExploitReentrancy
{
    mapping (address -> uint256) public balances;

    function attack(target) payable {
        Reentrancy(target).withdraw(0.1 ether);
    }
    // Khi hợp đồng này nhận được tiền của hợp đồng Reentrancy,
 ↪  nó sẽ tự động gọi hàm Fallback này, đây là điểm thực hiện
 ↪  khai thác lỗ hổng Reentrancy
    function () payable {
        Reentrancy(target).withdraw(0.1 ether);
    }
}
```

Hình 2.6: *Hợp đồng khai thác lỗi Reentrancy*

2.1.2.2. Arithmetic

Arithmetic hay còn gọi là Integer Underflow/Overflow là một lỗi phổ biến trong nhiều ngôn ngữ lập trình, được biết đến do việc bộ nhớ lưu trữ của kiểu biến số nguyên có giới hạn, gây ra sự thay đổi giá trị bất thường dựa vào đầu vào của người dùng. Tương tự trong ngữ cảnh của hợp đồng thông minh, lổ hổng Arithmetic có thể gây ra hậu quả rất nghiêm trọng liên quan đến ETH trong ví của nạn nhân. Giá trị kiểu số nguyên trong ngôn ngữ máy tính có phạm vi từ số nhỏ nhất (min) đến số lớn nhất (max). Lỗ hổng Integer Underflow xuất hiện khi phép tính vượt quá giá trị min: min - 1 $\rightarrow$ max; còn Integer Overflow xuất hiện khi phép tính vượt quá giá trị max: $max + 1 \rightarrow min$. **Hình 2.7** là một ví dụ điển hình về lỗi Integer Overflow mà kẻ tấn công có thể dễ dàng khai thác. Khiến số dư thay đổi không đúng với bình thường.

```
pragma solidity ^0.4.10;
contract IntegerOverflow
{
    mapping (address -> uint256) public balanceOf;

    function transfer(address _to, uint256 _value)
    {
        require (balanceOf[msg.sender] >= _value);
        balanceOf[msg.value] -= _value;
        balanceOf[_to] += _value;
    }
}
```

Hình 2.7: *Hợp đồng chứa lỗi Arithnetic*

2.1.2.3. Transaction Ordering Dependency

Trên Blockchain, hiệu suất của các hợp đồng thông minh thay đổi theo các chuỗi giao dịch khác nhau. Thật không may, các phần tiếp theo có thể bị thao túng bởi những người khai thác. Hãy xem xét một trường hợp trong đó nhóm giao dịch đang chờ xử lý (gọi tắt là txpool) có hai giao dịch mới (ví dụ: T; Ti) và chuỗi khối đang ở trạng thái S và trạng thái S chỉ có thể được chuyển thành trạng thái S1 nếu giao dịch T được xử lý. Ban đầu, T phải được xử lý ở trạng thái S, và do đó trạng thái này là từ S đến S1. Nhưng những người khai thác có thể xử lý giao dịch Ti trước T theo mong muốn của riêng họ và khi đó trạng thái là từ S sang S2 thay vì từ S sang S1. Do đó, nếu T được xử lý tại thời điểm này, trạng thái sẽ được thay đổi thành một trạng thái mới khác S3. Trong trường hợp đã nói ở trên, T được xử lý ở các trạng thái khối khác nhau và các lỗ hổng xuất phát từ những thay đổi của chuỗi giao dịch dự kiến

Hình 2.8 là một hợp đồng có chứa lỗ hổng TOD. Trong ví dụ này, hợp đồng thông minh người dùng có thể giành được 1000 ether nếu có thể tìm thấy một chuỗi sau khi băm có giá trị là biến "hash". Khi người dùng nộp lời giải bằng một giao dịch trong hợp đồng thông minh. Giao dịch đó sẽ được đưa vào hàng

```
pragma solidity ^0.4.22;
contract FindThisHas
{
    bytes32 constant public hash =
↪   0xee5faa5b4fba766466c...cb564592dedc40b68d63ad98aebf4;

    constructor public payable {}
    function solve(string solution) public
    {
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

Hình 2.8: *Hợp đồng chứa lỗi TOD*

đợi để chờ đưa vào block tiếp theo trong Blockchain. Kẻ tấn công sẽ theo dõi hàng đợi đó và thu thập lời giải. Sau đó nộp đáp án với phí gas cao hơn và giành được giải thưởng.

2.1.2.4. Time manipulation

Lỗ hổng này xảy ra khi một hợp đồng thông mình sử dụng các biến khối làm điều kiện gọi để thực hiện một số hoạt động quan trọng (ví dụ: gửi mã thông báo) hoặc làm nguồn gốc để tạo số ngẫu nhiên. Một số biến bắt nguồn từ tiêu đề khối, bao gồm BLOCKHASH, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT và COINBASE, do đó, về nguyên tắc, chúng có thể được thực hiện bởi những người khai thác. Chẳng hạn, những người khai thác có quyền đặt khối TIMESTAMP trong vòng 900 giây bù đắp. Nếu tiền điện tử được chuyển dựa trên các biến khối, những người khai thác có thể khai thác lỗ hổng bằng cách can thiệp vào chúng.

Hình 2.9 là một ví dụ về hợp đồng thông minh có chứa lỗ hổng Time manipulation. Trong ví dụ này, hợp đồng **TimeManipulation** có một biến **targetTime** đại diện cho thời gian mục tiêu và một biến **reward** đại diện cho

```
contract TimeManipulation {
    uint256 public targetTime;
    uint256 public reward;

    constructor(uint256 _targetTime, uint256 _reward) {
        targetTime = _targetTime;
        reward = _reward;
    }

    function claimReward() public {
        require(block.timestamp >= targetTime, "Reward not yet
 ↪  available");
        // Transfer reward to the caller
    }
}
```

Hình 2.9: *Hợp đồng chứa lỗi Time manipulation*

phần thưởng. Hàm **claimReward** được sử dụng để yêu cầu nhận phần thưởng, nhưng chỉ khi thời gian hiện tại (được đo bằng **block.timestamp**) lớn hơn hoặc bằng **targetTime**.

Tuy nhiên, lỗ hổng xảy ra khi kẻ tấn công có khả năng thay đổi thời gian hệ thống (system time manipulation). Bằng cách điều khiển thời gian của máy tính hoặc tạo ra các giao dịch thông minh tùy chỉnh, kẻ tấn công có thể đặt thời gian hiện tại lớn hơn **targetTime** và gọi hàm **claimReward** để nhận phần thưởng mà không cần chờ đến thời gian thực tế.

2.1.2.5. Timestamp dependency

Mã lệnh của hợp đồng thông minh thực thi trên máy ảo của các Miner (thợ đào) là lý do xảy ra lỗ hổng này. Tức là bất cứ mã lệnh nào trong hợp đồng thông minh cần đến Miner quyết định kết quả đều có thể bị can thiệp và điều hướng. Kiểu lỗ hổng này được phát hiện trong các mã lệnh sử dụng thời gian của hệ thống đào. Khi hợp đồng sử dụng nhãn thời gian để tạo số ngẫu nhiên,

Miner thực sự có thể đóng nhãn thời gian trong vòng 15 giây khi block đang được xác thực, cho phép người thợ đào có thể tính toán trước các tùy chọn. Nhãn thời gian không phải là ngẫu nhiên và không nên được sử dụng trong ngữ cảnh đó. Nếu có thể thì không nên dùng block.timestamp. Nếu sự kiện hợp đồng được triển khai có thể thay đổi trong 15 giây và duy trì tính toàn vẹn, thì việc sử dụng block.timestamp là an toàn.

```
contract TimestampsDependency {
    uint256 public startTime;
    uint256 public endTime;
    uint256 public reward;

    constructor(uint256 _startTime, uint256 _endTime, uint256
↪   _reward) {
        startTime = _startTime;
        endTime = _endTime;
        reward = _reward;
    }
    function claimReward() public {
        require(block.timestamp >= startTime, "Reward not yet
↪   available");
        require(block.timestamp <= endTime, "Reward no longer
↪   available");
        // Transfer reward to the caller
    }
}
```

Hình 2.10: *Hợp đồng chứa lỗi Timestamp dependency*

Hình 2.10 là một ví dụ về hợp đồng thông minh có chứa lỗ hổng Timestamp dependency. Trong ví dụ này, hợp đồng **TimestampsDependency** có ba biến **startTime**, **endTime**, và **reward**. Hàm **claimReward** được sử dụng để yêu cầu nhận phần thưởng, nhưng chỉ khi thời gian hiện tại (được đo bằng **block.timestamp**) nằm trong khoảng từ **startTime** đến **endTime**.

Tuy nhiên, lỗ hổng xảy ra khi kẻ tấn công có khả năng dự đoán hoặc thay

đổi thời gian của các giao dịch để thỏa mãn điều kiện thời gian. Bằng cách điều khiển thời gian của máy tính hoặc tạo ra các giao dịch thông minh tùy chỉnh, kẻ tấn công có thể gọi hàm **claimReward** trong khoảng thời gian không hợp lệ.

Ví dụ, kẻ tấn công có thể thay đổi thời gian của máy tính để đặt **block.timestamp** là một giá trị nằm trong khoảng không hợp lệ, bên ngoài khoảng từ **startTime** đến **endTime**. Khi kẻ tấn công gọi hàm **claimReward**, điều kiện **block.timestamp >= startTime** và **block.timestamp <= endTime** có thể bị mở rộng và kẻ tấn công có thể nhận được phần thưởng mà không phải là trong thời gian hợp lệ ban đầu.

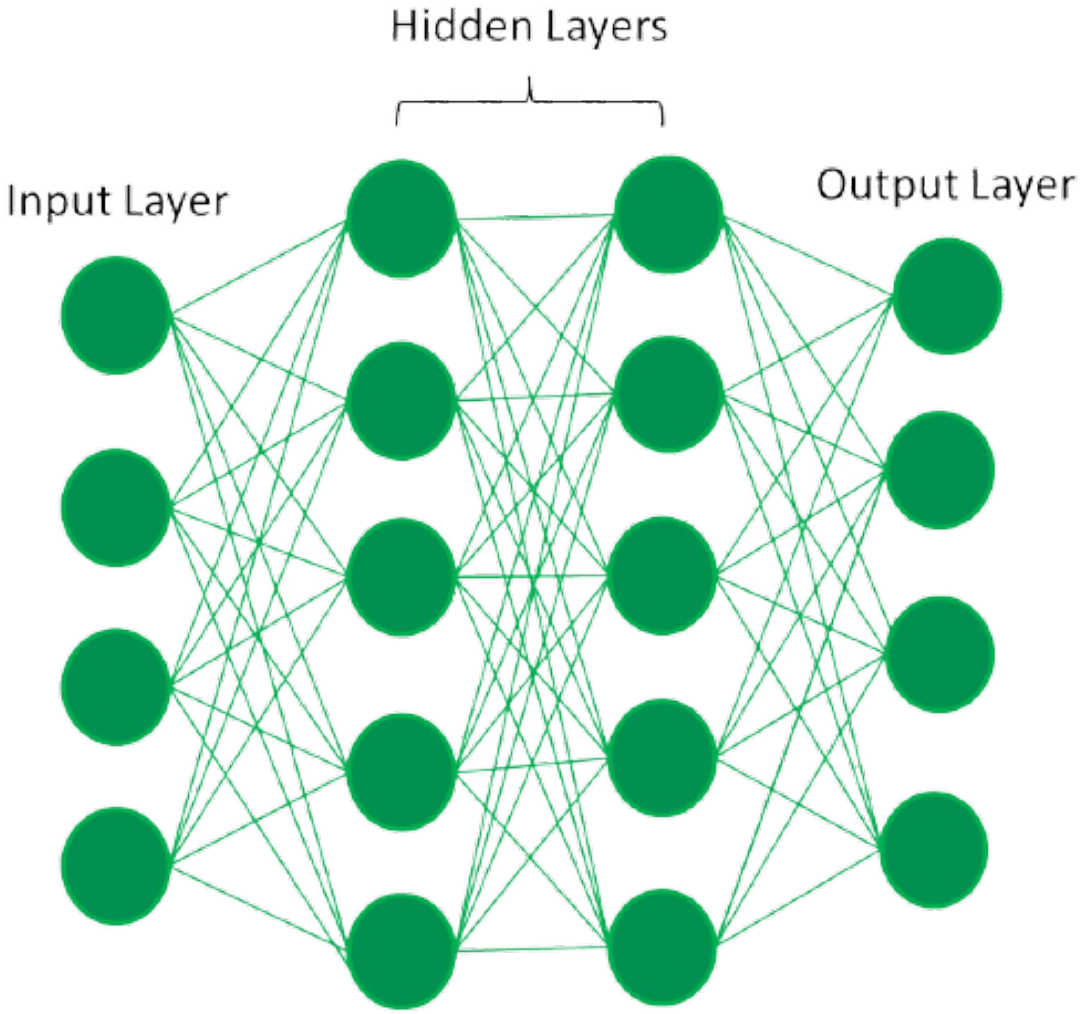
2.2. Các mô hình học sâu sử dụng

Trong phần này, chúng tôi tập trung giới thiệu về học sâu và các mô hình học sâu sử dụng. Trong Khoá luận này, chúng tôi sử dụng 3 mô hình học sâu cho 3 đặc trưng của hợp đồng thông minh là mã nguồn, opcode và CFG được trích xuất từ bytecode, lần lượt gồm: Bidirectional Encoder Representations from Transformers (BERT), Bi-directional Long Short Term Memory (BiLSTM) và Graph Neural Network (GNN).

2.2.1. Tổng quan về học sâu

Học sâu (Deep learning) là một lĩnh vực trong trí tuệ nhân tạo (Artificial Intelligence) tập trung vào việc xây dựng và huấn luyện các mô hình máy tính có khả năng học và tự cải thiện thông qua việc xử lý dữ liệu lớn. Nó được gọi là "sâu" vì mô hình học sâu có thể có hàng trăm hoặc hàng ngàn lớp ẩn (hidden layers), cho phép chúng khám phá và học các đặc trưng phức tạp từ dữ liệu. **Hình 2.11** là một ví dụ cho một mô hình học sâu đơn giản gồm 1 Input layer, 2 Hidden layers và 1 Output layer.

Một mô hình học sâu thường được xây dựng dựa trên mạng nơ-ron nhân tạo



Hình 2.11: *Kiến trúc của mô hình học sâu cơ bản*

(Artificial Neural Network), cấu trúc mô phỏng cách hoạt động của nơ-ron trong não bộ con người. Mạng nơ-ron nhân tạo bao gồm các nút nơ-ron liên kết với nhau thông qua các trọng số (weights), và các tầng (layers) để thực hiện các phép biến đổi dữ liệu.

Trong quá trình huấn luyện, mô hình học sâu tối ưu hóa các trọng số trong mạng nơ-ron dựa trên dữ liệu huấn luyện thông qua một quá trình gọi là lan truyền ngược (backpropagation). Thuật toán gradient descent được sử dụng để tối thiểu hóa hàm mất mát (loss function), đo lường sự sai khác giữa đầu ra dự đoán và đầu ra thực tế.

Một trong những ưu điểm chính của học sâu là khả năng học tự động các đặc trưng phức tạp từ dữ liệu. Thay vì phải thiết kế và chọn lọc các đặc trưng thủ công, mô hình học sâu có khả năng học các đặc trưng tầng cao từ dữ liệu đầu vào. Điều này giúp nâng cao hiệu suất và độ chính xác của mô hình trong nhiều tác vụ như nhận dạng hình ảnh, xử lý ngôn ngữ tự nhiên, dự báo và phân tích dữ liệu.

Học sâu cũng đòi hỏi một lượng lớn dữ liệu huấn luyện và tài nguyên tính toán. Tuy nhiên, với sự phát triển của công nghệ và quy mô dữ liệu, các mô hình học sâu ngày càng mạnh mẽ và được áp dụng rộng rãi trong nhiều lĩnh vực như công nghệ thị giác máy tính, xử lý ngôn ngữ tự nhiên, dữ liệu y tế, tự động hoá cũng như trong an toàn thông tin.

Đối với ngữ cảnh bảo mật trong hợp đồng thông mình, mô hình học sâu có thể được sử dụng để phân tích hợp đồng thông minh và tìm ra các lỗ hổng bảo mật. Thông qua việc học các mẫu từ các hợp đồng thông minh đã được kiểm tra, mô hình học sâu có thể nhận biết các lỗ hổng phổ biến như lỗi tràn bộ nhớ, truy cập không hợp lệ vào biến, hoặc lỗi trong quy trình xử lý giao dịch.

Một khía cạnh khác của mô hình học sâu trong bảo mật hợp đồng thông minh là phát hiện các cuộc tấn công. Hợp đồng thông minh có thể trở thành mục tiêu của các cuộc tấn công như tấn công kiểm thử hộp đen, tấn công kiểm thử hộp xám, hoặc tấn công tác nhân. Mô hình học sâu có thể được huấn luyện để xác

định các hành vi bất thường hoặc đặc trưng của các cuộc tấn công này. Thông qua việc phân tích lịch sử giao dịch và học các mẫu từ các cuộc tấn công đã xảy ra, mô hình học sâu có thể cung cấp cảnh báo sớm về các hoạt động đáng ngờ và giúp ngăn chặn các cuộc tấn công tiềm năng.

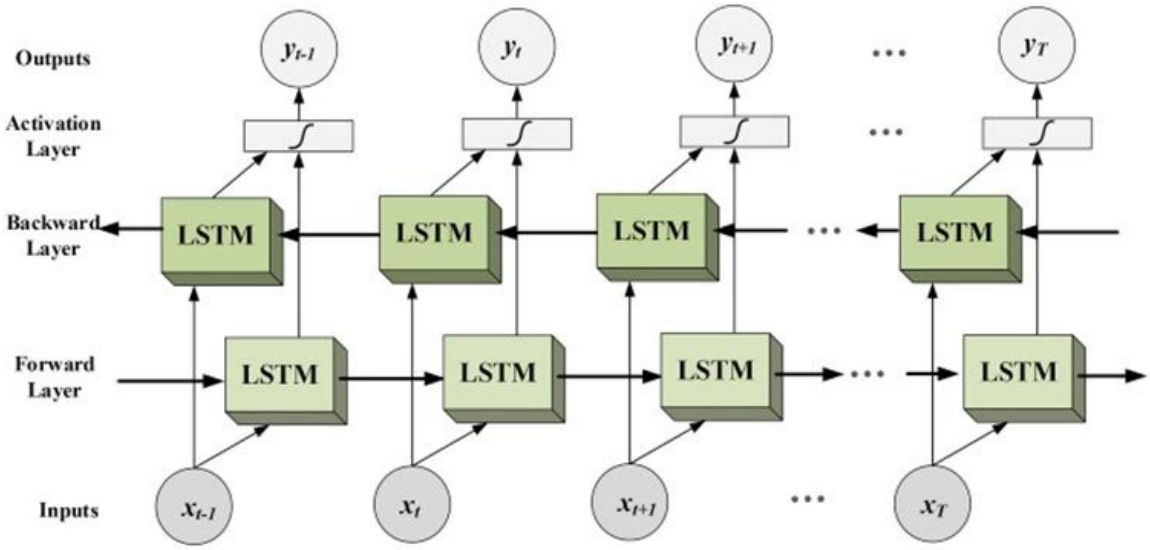
Ngoài ra, mô hình học sâu cũng có thể được sử dụng trong việc tăng cường bảo mật của hợp đồng thông minh thông qua quá trình học tăng cường (reinforcement learning). Mô hình học sâu có thể được huấn luyện để học cách tối ưu hóa các tham số và quy tắc trong quá trình thực thi hợp đồng thông minh, từ đó giảm thiểu rủi ro bảo mật và tăng cường tính bảo mật của hợp đồng.

Tóm lại, học sâu là một lĩnh vực quan trọng trong trí tuệ nhân tạo, với khả năng học và tự cải thiện thông qua việc xử lý dữ liệu lớn. Với sự phát triển của mạng nơ-ron nhân tạo và thuật toán gradient descent, học sâu đã mang lại nhiều tiềm năng và ứng dụng trong nhiều lĩnh vực và là một công cụ quan trọng cho việc giải quyết các bài toán phức tạp.

2.2.2. Bi-directional Long Short Term Memory (BiLSTM)

BiLSTM, hay Birectional Long Short-Term Memory, là một mô hình mạng neural sử dụng cho việc xử lý dữ liệu chuỗi. Đặc điểm nổi bật của BiLSTM là khả năng kết hợp hai mô hình LSTM, một theo hướng thuận và một theo hướng ngược, để xử lý thông tin từ cả hai phía của chuỗi đầu vào. LSTM (Long Short-Term Memory) là một biến thể nổi tiếng của mô hình RNN (Recurrent Neural Network) được thiết kế để giải quyết vấn đề biến mất gradient trong quá trình huấn luyện mạng neural khi xử lý các chuỗi dữ liệu dài. LSTM giữ một bộ nhớ nội tại để lưu trữ thông tin quan trọng trong quá khứ và áp dụng các cơ chế cập nhật thông tin (gọi là cổng) để điều chỉnh việc lưu trữ và truy cập vào bộ nhớ này. Trong mạng BiLSTM, chuỗi đầu vào được chia thành các thành phần riêng lẻ và truyền qua mạng LSTM theo hai hướng: thuận và ngược. Mạng LSTM thuận xử lý chuỗi từ trái sang phải, trong khi mạng LSTM ngược xử lý chuỗi từ phải sang trái. Điều này cho phép BiLSTM học được các mối

quan hệ giữa các thành phần trong chuỗi không chỉ theo hướng thời gian mà còn theo hướng ngược lại. **Hình 2.12** thể hiện tổng quan về các lớp của mô hình BiLSTM. Input từ input layer được lan truyền vào 2 mạng LSTM forward và backward.



Hình 2.12: *Tổng quan mô hình BiLSTM*

Trong mô hình BiLSTM thì ta sẽ có các công thức cho mạng LSTM thuận và LSTM nghịch. Đầu tiên, chúng ta xem xét mạng LSTM thuận (forward LSTM):

1. Cổng quên (Forget Gate):
   - Input: x_t (đầu vào tại thời điểm t), h_{t-1} (trạng thái ẩn tại thời điểm $t-1$)
   - Output: f_t (giá trị của cổng quên)
   - Công thức: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
2. Cổng đầu vào (Input Gate):
   - Input: x_t, h_{t-1}
   - Output: i_t (giá trị của cổng đầu vào)

- Công thức: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

3. Cổng đầu ra (Output Gate):

- Input: x_t, h_{t-1}
- Output: o_t (giá trị của cổng đầu ra)
- Công thức: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$

4. Cổng trạng thái (Cell Gate):

- Input: x_t, h_{t-1}
- Output: g_t (giá trị của cổng trạng thái)
- Công thức: $g_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

5. Trạng thái ẩn mới (New Cell State):

- Input: x_t, h_{t-1}
- Output: c_t (trạng thái ẩn mới)
- Công thức: $c_t = f_t \cdot c_{t-1} + i_t \cdot g_t$

6. Trạng thái ẩn (Hidden State):

- Input: x_t, h_{t-1}
- Output: h_t (trạng thái ẩn)
- Công thức: $h_t = o_t \cdot \tanh(c_t)$

Tiếp theo, chúng ta xem xét mạng LSTM nghịch (backward LSTM). Các công thức trong mạng LSTM nghịch tương tự như trong mạng LSTM thuận, nhưng với các trọng số và bias riêng (ký hiệu bằng W' và b'). Khi áp dụng mạng BiLSTM, chuỗi đầu vào được chia thành các thành phần riêng lẻ và truyền qua mạng LSTM thuận và LSTM nghịch. Đầu ra cuối cùng của BiLSTM là sự kết hợp của cả hai mạng LSTM, cung cấp thông tin từ cả quá khứ và tương lai trong quá trình xử lý dữ liệu chuỗi.

Khi đưa một chuỗi vào mạng BiLSTM, đầu ra của mỗi bước thời gian sẽ bao gồm thông tin từ cả hai mô hình LSTM, góp phần tăng cường sự hiểu biết về mối quan hệ giữa các thành phần trong chuỗi. Do đó, BiLSTM có khả năng nhìn thấy cả quá khứ và tương lai của mỗi thành phần trong chuỗi, điều này làm tăng khả năng dự đoán chính xác hơn. BiLSTM được sử dụng rộng rãi trong các lĩnh vực như xử lý ngôn ngữ tự nhiên, dịch máy, nhận dạng giọng nói, phân loại văn bản, và nhiều ứng dụng khác liên quan đến dữ liệu chuỗi. Nhờ khả năng xử lý thông tin chuỗi theo hai hướng, BiLSTM đã giúp cải thiện đáng kể hiệu suất của nhiều bài toán trong lĩnh vực này. Tóm lại, BiLSTM là một mô hình mạng neural mạnh mẽ cho việc xử lý dữ liệu chuỗi. Khả năng kết hợp thông tin từ cả hai phía của chuỗi giúp BiLSTM nắm bắt được các mối quan hệ phức tạp và đưa ra dự đoán chính xác.

2.2.3. Bidirectional Encoder Representations from Transformers (BERT)

BERT là một mô hình xử lý ngôn ngữ tự nhiên dựa trên mạng Transformer, được phát triển bởi Google vào năm 2018 [5]. Mô hình BERT được huấn luyện trên một lượng lớn dữ liệu văn bản giúp cho mô hình có khả năng hiểu được ngữ cảnh và ý nghĩa của các từ trong câu. BERT có tốc độ vượt trội khi đầu vào được đưa vào mô hình cùng lúc, chứ không tuần tự như các mô hình khác như RNN, LSTM. BERT đã đem lại các cải tiến đáng chú ý trong cộng đồng xử lý ngôn ngữ tự nhiên như:

- Tăng GLUE score (General Language Understanding Evaluation score), một chỉ số tổng quát đánh giá mức độ hiểu ngôn ngữ lên 80.5
- Tăng Accuracy trên bộ dữ liệu MultiNLI [1] đánh giá tác vụ quan hệ văn bản (text entailment) lên 86.7
- Tăng Accuracy và F1-score trên bộ dữ liệu SQuAD [2] v1.1 đánh giá tác vụ hỏi và trả lời lên đến 93.2

Mô hình BERT có một cấu trúc phức tạp và mạnh mẽ được xây dựng dựa trên mạng Transformer. Dưới đây là một phần mô tả về cấu trúc của BERT:

1. Lớp đầu vào: Đầu vào của BERT là một chuỗi văn bản được chia thành các từ và sau đó mã hoá thành các vector biểu diễn từ, gọi là embedding. Ngoài ra, BERT còn sử dụng một thẻ đặc biệt ở đầu câu là [CLS] (Classification) và thêm thẻ [SEP] (Separation) giữa hai câu trong các tác vụ có liên quan đến cặp câu.

2. Mạng transformer: BERT sử dụng mạng transformer như một thành phần chính để xử lý văn bản. Mạng transformer bao gồm nhiều lớp encoder, mỗi lớp encoder bao gồm một lớp self-attention và feed-forward network. Mỗi lớp encoder sẽ tạo ra một biểu diễn mới cho từng từ trong chuỗi đầu vào dựa trên ngữ cảnh toàn bộ câu.

3. Lớp mã hoá thông tin từ: Sau khi đi qua các lớp encoder, BERT sử dụng một lớp mã hoá thông tin từ (Token-Level Encoder) để tạo ra biểu diễn cuối cùng cho từng từ trong chuỗi văn bản.

4. Lớp mã hoá thông tin câu: BERT cũng có một lớp mã hoá thông tin câu (Sentence-Level Encoder) để tạo ra biểu diễn cho toàn bộ câu. Điều này giúp mô hình BERT hiểu được ngữ cảnh và mối quan hệ giữa các câu trong một đoạn văn.

5. Lớp phân loại: Cuối cùng, BERT sử dụng lớp phân loại để đưa ra dự đoán cho các tác vụ như phân loại văn bản. Lớp này có thể được kết nối với biểu diễn của từ [CLS] hoặc một tập hợp các biểu diễn từ khác để thực hiện tác vụ phân loại.

BERT được hiểu rằng đây là mô hình biểu diễn từ theo hai chiều sử dụng Transformer. Chính vì thế, Transformer chính là thành phần chính trong BERT. Transformer là một lớp mô hình seq2seq gồm 2 phần là Encoder và Decoder.

Transformer sử dụng các layers attention để thực hiện embedding các từ trong câu. Kiến trúc cụ thể của mô hình được thể hiện trong **Hình 2.13**.

Mô hình sẽ gồm 2 phần:

- **Encoder:** Khối Encoder này gồm 6 layers được xếp liên tiếp nhau. Mỗi layer sẽ có thêm một sub-layer được gọi là Multi-Head Attention kết hợp với fully-connected layer được mô tả ở nhánh bên trái của **Hình 2.13**. Kết thúc quá trình Encoder, sẽ thu được một vector embedding cho mỗi từ trong câu.

- **Decoder:** Khối Decoder cũng bao gồm các layers được xếp liên tiếp nhau nhưng khác số lượng so với khối Encoder. Mỗi layer trong khối Decoder cũng có các sub-layers tương tự như các layer trong khối Encoder nhưng bổ sung thêm sub-layer được gọi là Masked Multi-Head Attention có tác dụng loại bỏ các từ trong tương lai khỏi quá trình attention.

Hiện tại, có thể tạo ra nhiều phiên bản khác nhau của mô hình BERT bằng cách thay đổi kiến trúc của Transformer tập trung ở 3 tham số:

1. **L:** Là số lượng các block sub-layers trong Transformer.

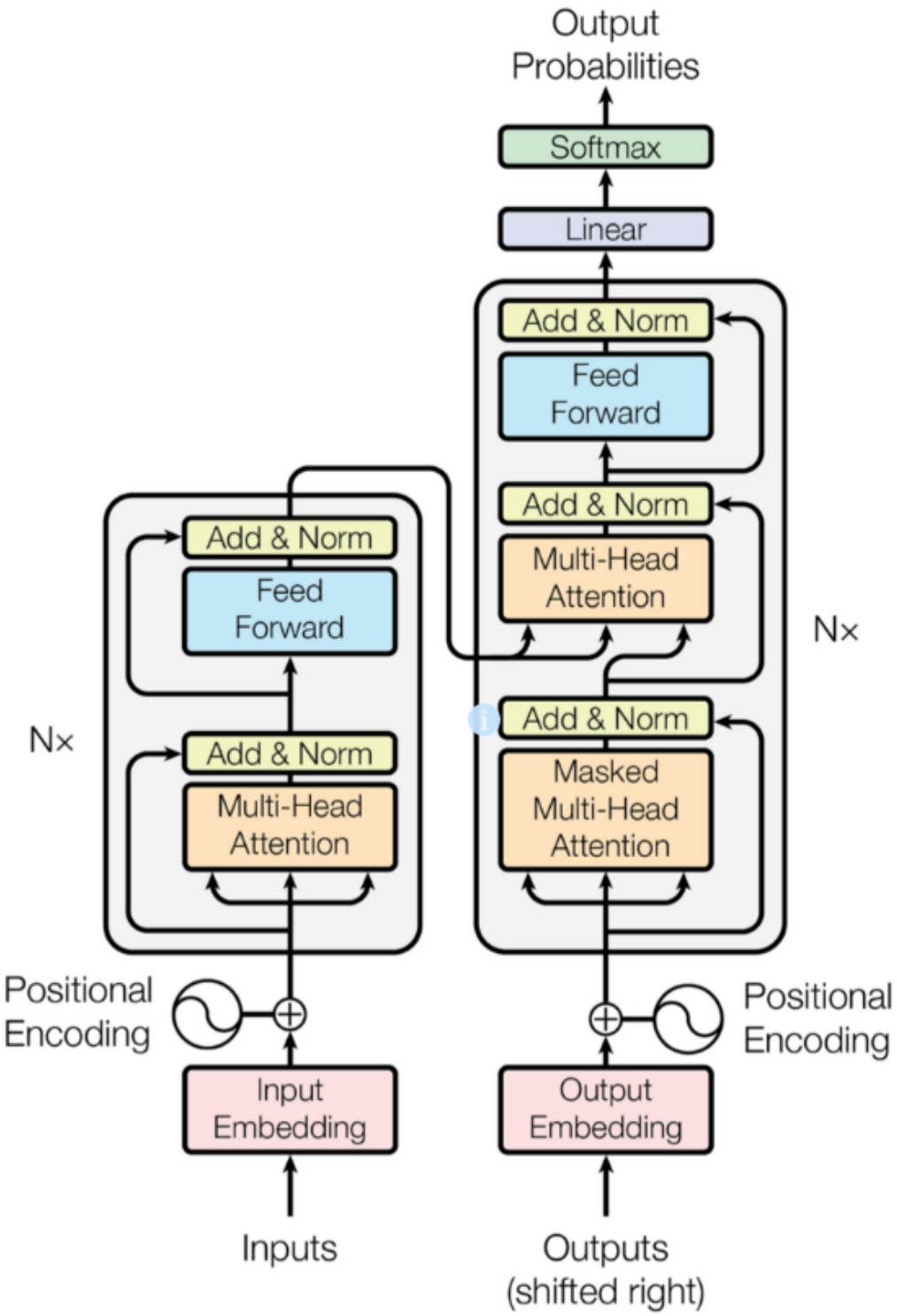
2. **H:** Là kích thước của embedding vector (hidden size).

3. **A:** Số lượng head trong multi-head layer, tại đây, mỗi head này sẽ thực hiện hành động self-attention.

Hiện nay, có hai mô hình với cấu hình ba tham số được sử dụng phổ biến nhất là:

- $BERT_{Base}(L = 12, H = 768, A = 12)$

- $BERT_{Large}(L = 24, H = 1024, A = 16)$

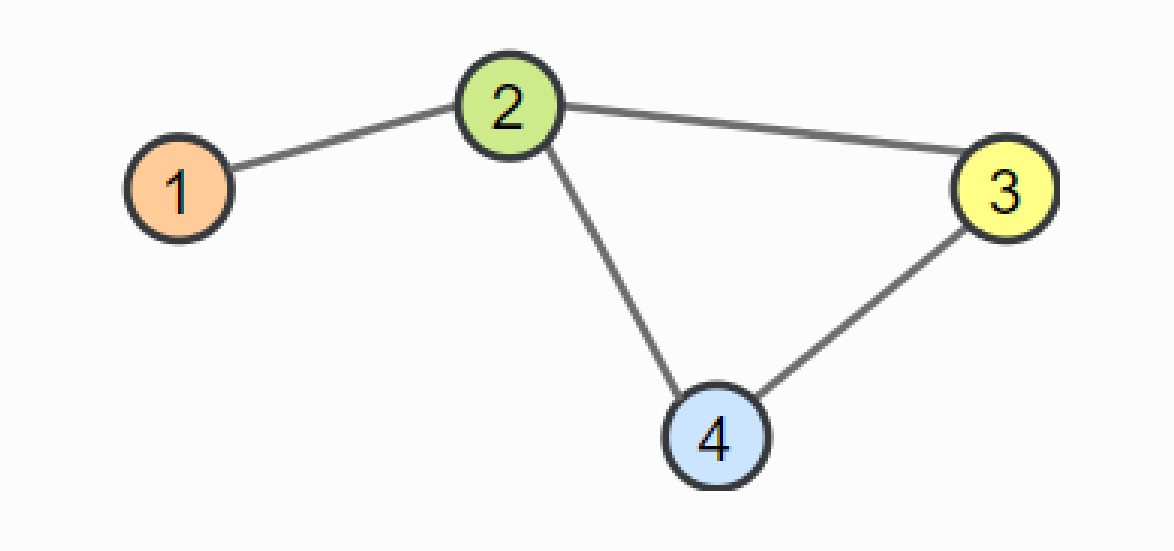


Hình 2.13: *Kiến trúc mô hình Transformer*

2.2.4. Graph Neural Network (GNN)

Graph Neural Network (GNN) là một mô hình dựa trên đầu vào là các thông tin về đồ thị. Trong toán học, đồ thị **G** được định nghĩa là tập hợp các đỉnh (nút) và các cạnh kết nối các đỉnh của đồ thị, được kí hiệu là G = (V, E), trong đó:

- **V** là tập hợp các đỉnh của đồ thị.
- **E** là tập hợp các cạnh của đồ thị, với mỗi cạnh nối 2 đỉnh trong đồ thị.



Hình 2.14: *Đồ thị vô hướng đơn giản*

Hình 2.14 thể hiện một đồ thị vô hướng đơn giản. Đồ thị này có tập hợp các đỉnh gồm $V = \{1, 2, 3, 4\}$, tập hợp các cạnh gồm $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$. Trong thực tế, các đỉnh và cạnh thường sẽ có những thuộc tính cụ thể và đồ thị thường là đồ thị có hướng thay vì vô hướng. Vấn đề được đặt ra là làm sao biểu diễn được sự đa dạng về đặc trưng của các đỉnh và các cạnh một cách hiệu quả cho các phép toán ma trận. Một phương pháp thường dùng chính là **Adjacency Matrix (ma trận kề)**. Gọi ma trận kề là A, A là một ma trận vuông với kích thước NxN với N là số đỉnh của đồ thị. Trong ma trận A, mỗi phần tử A_{ij} thể hiện việc có kết nối từ đỉnh i tới đỉnh j, giá trị 1 là có kết nối và giá trị 0 là

ngược lại. **Hình 2.15** là ma trận kề A biểu diễn cho đồ thị trong **Hình 2.14**.

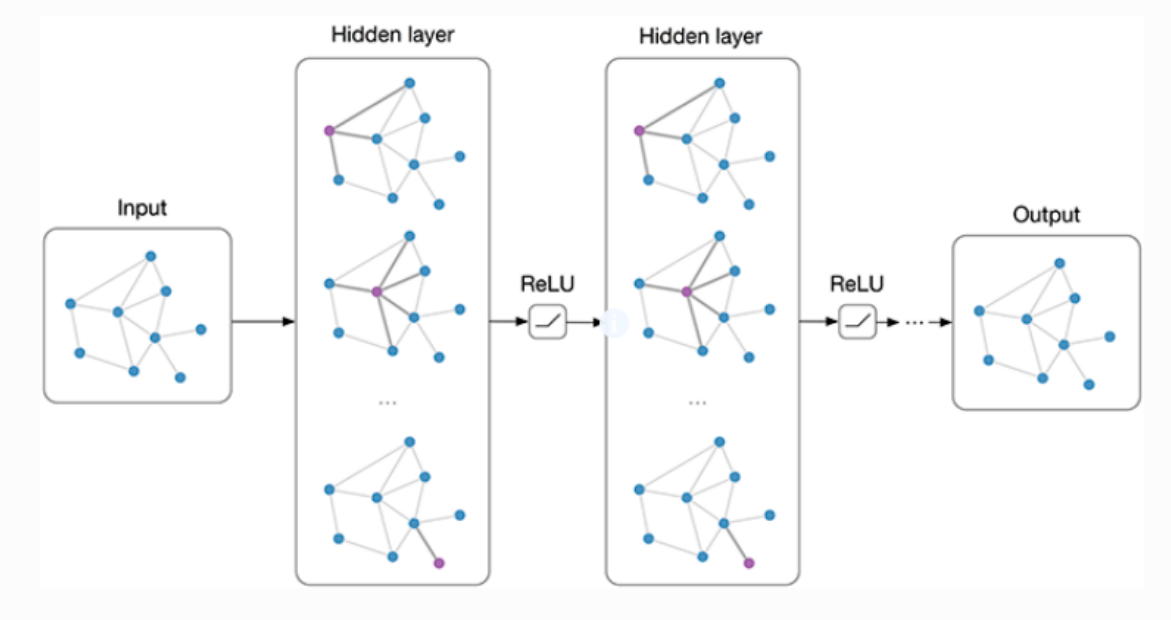
$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \tag{2.1}$$

Hình 2.15: *Ma trận kề A*

Ma trận A còn được gọi là ma trận trọng số, thể hiện trọng số của các cạnh trong đồ thị. Với ma trận minh họa trong **Hình 2.15** thì các cạnh có trọng số như nhau nhưng có thể thay đổi lại tùy bài toán và dữ liệu. Ngoài việc biểu diễn theo ma trận kề, có thể biểu diễn đồ thị dưới dạng một danh sách các cạnh. Trong khi việc biểu diễn một đồ thị dưới dạng một danh sách các cạnh sẽ hiệu quả hơn về mặt bộ nhớ và (có thể) tính toán, thì việc sử dụng ma trận kề sẽ trực quan hơn và đơn giản hơn để thực hiện. Ngoài ra, cũng có thể sử dụng danh sách các cạnh để xác định một ma trận kề thưa (sparse matrix) mà chúng ta có thể làm việc như thể nó là một ma trận dày đặc (dense matrix), nhưng cho phép các hoạt động sử dụng bộ nhớ hiệu quả hơn.

Mô hình GNN là một mô hình học sâu được thiết kế đặc biệt để làm việc với các đồ thị đã được biểu diễn đầy đủ thông tin và có giá trị tính toán. GNN có khả năng xử lý thông tin và học các đặc trưng trên các đỉnh và cạnh trong đồ thị.

GNN sử dụng một kiến trúc mạng neural network để lan truyền và cập nhật thông tin trong đồ thị. Quá trình lan truyền thông tin trong GNN được thực hiện thông qua các lớp neural network đặc biệt được gọi là lớp gộp (aggregation layer) hoặc lớp thông báo (message-passing layer). Các lớp này truyền thông tin từ các hàng xóm của mỗi đỉnh đến đỉnh đó, cho phép mô hình học cách xử lý thông tin từ các đỉnh kết nối trong đồ thị. Thông tin từ lớp trước được lan truyền sang lớp sau được lan truyền qua một hàm phi tuyến tính như ReLU. **Hình 2.16** thể hiện một kiến trúc mô hình GNN cơ bản.



Hình 2.16: *Kiến trúc mô hình GNN cơ bản*

GNN có thể học các đặc trưng phức tạp và tổng hợp thông tin từ cấu trúc đồ thị. Quá trình học của GNN bao gồm việc lặp lại các bước lan truyền thông tin trong đồ thị cho đến khi đạt được sự hội tụ. Quá trình này cho phép GNN tích hợp thông tin từ các đỉnh và cạnh xung quanh và áp dụng các phép toán học để học các đặc trưng đồ thị.

2.3. Mô hình học sâu đa phương thức

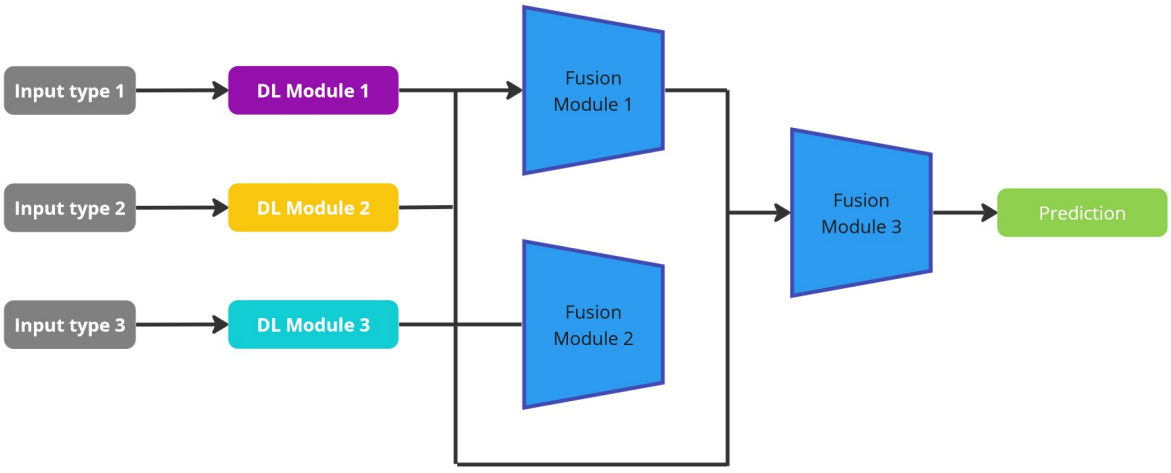
Mô hình học sâu đa phương thức là mô hình kết hợp và xử lý thông tin từ nhiều nguồn dữ liệu đa phương thức khác nhau của một đối tượng, chẳng hạn như hình ảnh, âm thanh, văn bản và video. Mục tiêu của mô hình là tận dụng sự đa dạng thông tin từ các nguồn dữ liệu khác nhau để cải thiện hiệu suất và khả năng dự đoán của mô hình. Thay vì chỉ dựa trên một nguồn dữ liệu đơn lẻ, mô hình học sâu đa phương thức có khả năng tìm ra các mối quan hệ phức tạp giữa các đặc trưng từ các nguồn dữ liệu khác nhau để đưa ra các dự đoán chính xác hơn và đa chiều hơn.

Mô hình học sâu đa phương thức khám phá và tận dụng sự tương tác và sự

tương quan giữa các đặc trưng từ các nguồn thông tin khác nhau. Trong quá trình huấn luyện, mô hình học sâu đa phương thức học cách kết hợp và tích hợp thông tin từ các biểu diễn khác nhau của đối tượng để đưa ra dự đoán hoặc rút ra thông tin mới. Điều này có thể được thực hiện thông qua việc tạo ra các biểu diễn đa phương thức (multimodal representations) bằng cách kết hợp các mạng neural network đơn lẻ và tạo ra một mô hình kết hợp (joint model) để đồng thời học từ tất cả các nguồn dữ liệu.

Mô hình học sâu đa phương thức đã được áp dụng trong nhiều lĩnh vực, bao gồm nhận dạng hình ảnh và video, xử lý ngôn ngữ tự nhiên, nhận dạng giọng nói, và nhiều ứng dụng khác. Điều quan trọng là mô hình này có khả năng khai thác sự phong phú và đa dạng của thông tin từ nhiều nguồn dữ liệu, từ đó cung cấp một cái nhìn toàn diện và cải thiện khả năng dự đoán và hiểu thông tin.

Mô hình học sâu đa phương thức thường sử dụng các kiến trúc mạng neural network phức tạp như mạng neural network tích chập (Convolutional Neural Network - CNN), mạng neural network tái phát (Recurrent Neural Network - RNN) hoặc mạng neural network transformer để xử lý dữ liệu từng nguồn riêng biệt. Sau đó, các đặc trưng trích xuất từ mỗi nguồn dữ liệu được kết hợp lại và đi qua các lớp fully connected, hàm kích hoạt và các phép biến đổi khác để tạo ra kết quả dự đoán cuối cùng.



Hình 2.17: *Kiến trúc mô hình học sâu đa phương thức cơ bản*

Hình 2.17 thể hiện cấu trúc của một mô hình học sâu đa phương thức cơ bản. Với mỗi đặc trưng của loại đối tượng, ta sử dụng các mô hình học sâu để trích xuất thành các vector thuộc tính. Sau đó, các vector thuộc tính này được kết hợp với nhau bằng các lớp kết nối, thường là lớp kết nối hoàn toàn (fully connection layers). Có thể kết nối đầu ra của các mô hình cùng một lúc, hoặc kết nối đầu ra của từng 2 mô hình một lần. Sau khi kết nối, ta có thể thực hiện các tác vụ khác như dự đoán, biểu diễn,...

CHƯƠNG 3. PHƯƠNG PHÁP THIẾT KẾ

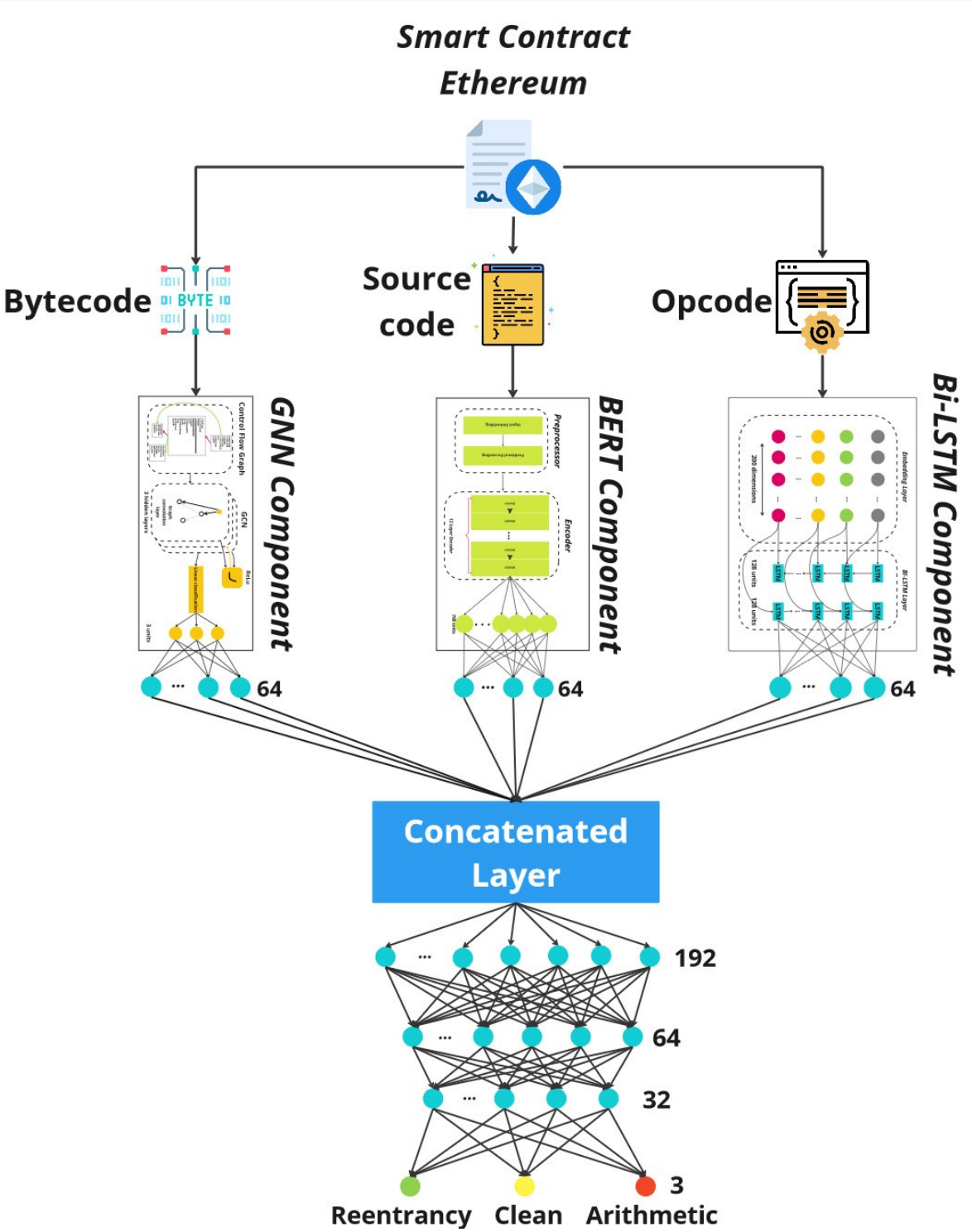
Ở chương này chúng tôi sẽ trình bày tổng quan về phương pháp của chúng tôi trong xây dựng công cụ VulnSense phát hiện lỗ hổng trên hợp đồng thông minh. Hơn nữa, bằng cách sử dụng mô hình học sâu đa phương thức để kết hợp các đặc trưng, chúng tôi tạo ra một biểu diễn toàn diện hơn về hợp đồng thông minh, từ đó cho phép chúng tôi trích xuất các đặc trưng liên quan hơn từ dữ liệu và cải thiện hiệu suất của mô hình.

3.1. Tổng quan mô hình

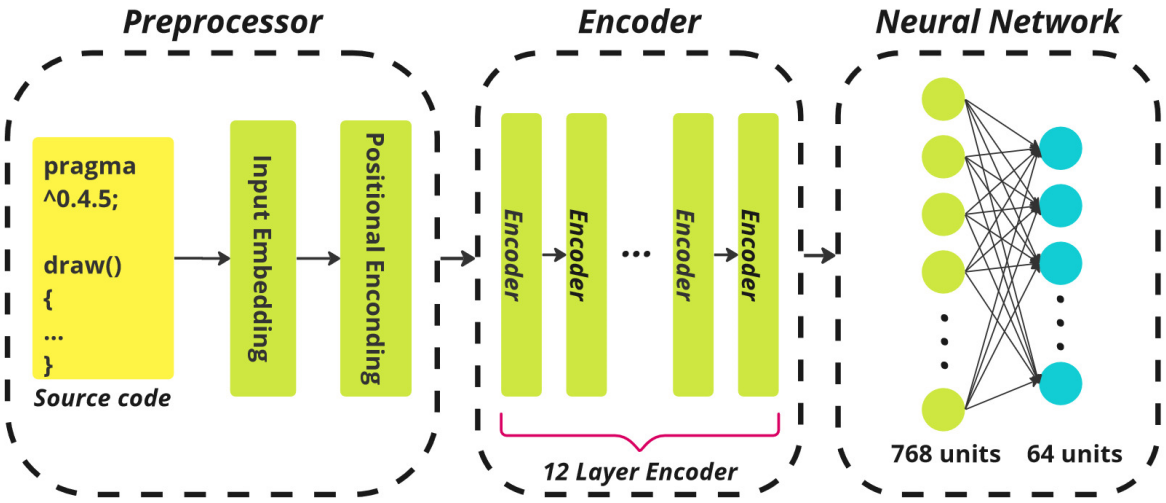
Phương pháp đề xuất VulnSense của chúng tôi được xây dựng dựa trên mô hình học sâu đa phương thức gồm ba nhánh bao gồm BERT, BiLSTM và GNN, như được mô tả trong **Hình 3.1**. Thứ nhất, mô hình BERT, dựa trên kiến trúc Transformer, được sử dụng để xử lý mã nguồn của hợp đồng thông minh. Thứ hai, để xử lý và phân tích ngữ cảnh của opcode, mô hình BiLSTM được áp dụng trên nhánh thứ hai. Cuối cùng, mô hình GNN được sử dụng cho kiểu biểu diễn CFG của hợp đồng thông minh.

3.1.1. Bidirectional Encoder Representations from Transformers (BERT)

Trong nghiên cứu này, chúng tôi sử dụng BERT làm một nhánh trong mô hình đa phương thức của chúng tôi với đầu vào là mã nguồn của SC. Như được mô tả trong **Hình 3.2**, mô hình BERT bao gồm 3 khối: Preprocessor, Encoder và Neural Network.



Hình 3.1: *Kiến trúc của mô hình VulnSense.*



Hình 3.2: *Phát hiện lỗ hổng bằng mã nguồn dùng BERT.*

Cụ thể hơn, khối Preprocessor xử lý đầu vào là mã nguồn của các hợp đồng thông minh. Đầu vào được biến đổi thành các vector thông qua lớp nhúng đầu vào (input embedding layer), sau đó đi qua lớp *positional_encoding* (3.1) để thêm thông tin vị trí cho các từ. Hơn nữa, đầu vào văn bản đã được chuẩn hóa về dạng không phân biệt viết hoa hay viết thường, có nghĩa là văn bản đã được chuyển thành chữ thường trước khi token hóa thành các phần từ ngữ, và tất cả comment cũng như khoảng trắng dư thừa trong code cũng được loại bỏ.

$$\textbf{preprocessed} = \mathit{positional\_encoding}(\mathbf{e}(\mathit{input})) \tag{3.1}$$

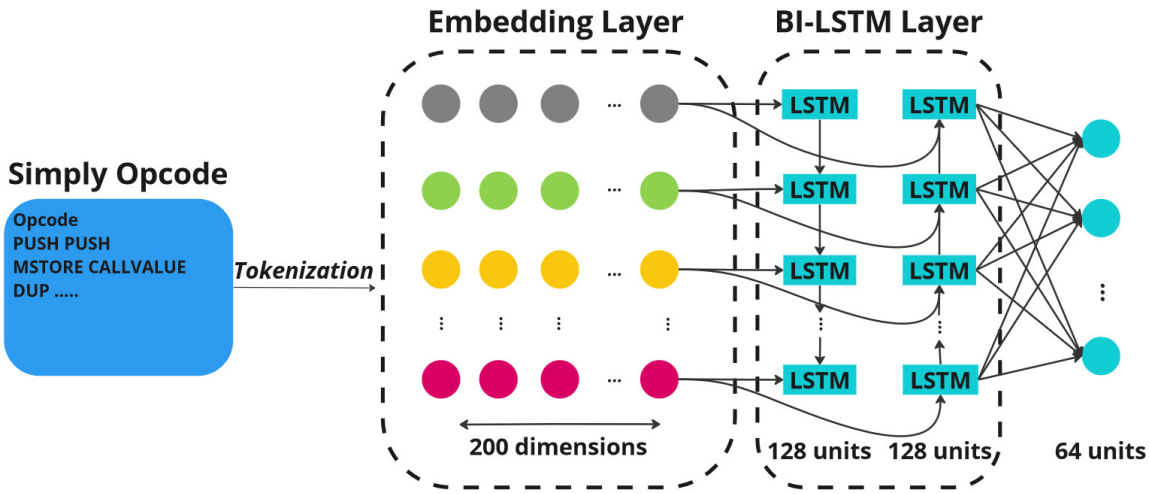
Sau quá trình tiền xử lý, để tính toán các mối quan hệ giữa các từ, giá trị **preprocessed** được đưa vào khối mã hóa (Encoder) (3.2). Toàn bộ khối mã hóa bao gồm 12 lớp mã hóa giống nhau được xếp chồng lên nhau. Mỗi lớp mã hóa bao gồm hai phần chính: self-attention layer và feed-forward neural network. Đầu ra **encoded** là một không gian vector có độ dài 768.

$$\textbf{encoded} = \mathit{Encoder}(\textbf{preprocessed}) \tag{3.2}$$

Tiếp theo, giá trị **encoded** sẽ được đưa vào một mạng neural đơn giản (3.3). Và giá trị **bert_output** là đầu ra của nhánh này trong mô hình đa phương thức.

$$\textbf{bert\_output} = NN(\textbf{encoded}) \tag{3.3}$$

3.1.2. Bidirectional long-short term memory (BiLSTM)



Hình 3.3: *Phát hiện lỗ hổng bằng opcode dùng BiLSTM.*

Chúng tôi áp dụng BiLSTM trong nhánh thứ hai của mô hình đa phương thức để phát hiện các lỗ hổng trong các hợp đồng thông minh dựa trên opcode, như trong **Hình 3.3**. Trước tiên, chúng tôi tokenize hóa (3.4) các opcode và chuyển chúng thành giá trị số nguyên bằng cách sử dụng từ điển.

$$\textbf{token} = Tokenize(\textbf{opcode}) \tag{3.4}$$

Các opcode sau đó được nhúng vào không gian vector dày đặc bằng một lớp nhúng (embedding layer) (3.5), tiếp theo là hai lớp BiLSTM với độ dài là 128.

$$\mathbf{vector\_space} = Embedding(\mathbf{token}) \tag{3.5}$$

Tiếp theo, một lớp Dropout được áp dụng sau lớp BiLSTM đầu tiên để giảm overfitting, như trong (3.6).

$$\begin{aligned} \mathbf{bi\_lstm1} &= Bi\_LSTM(128)(\mathbf{vector\_space}) \\ \mathbf{r} &= Dropout(dense(\mathbf{bi\_lstm1})) \end{aligned} \tag{3.6}$$

Sau đó, đầu ra của lớp BiLSTM cuối cùng được đưa vào một lớp dense với độ dài 64 và hàm kích hoạt ReLU, như trong (3.7).

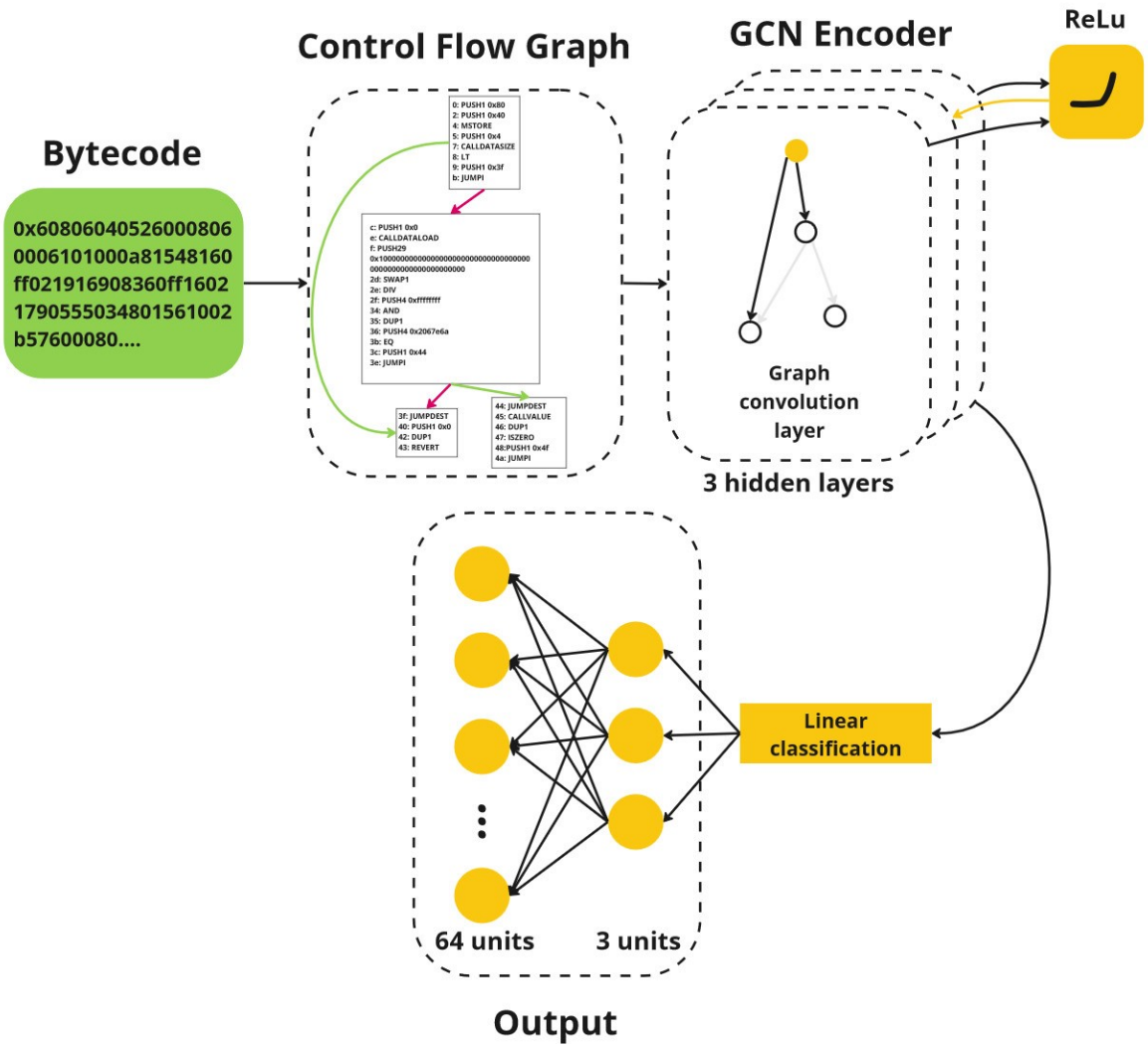
$$\begin{aligned} \mathbf{bi\_lstm2} &= Bi\_LSTM(128)(\mathbf{r}) \\ \mathbf{lstm\_output} &= Dense(64, relu)(\mathbf{bi\_lstm2}) \end{aligned} \tag{3.7}$$

3.1.3. Graph Neural Network (GNN)

Ở nhánh thứ ba của mô hình đa phương thức chúng tôi sử dụng GNN dựa trên CFG như trong **Hình 3.4**. Đầu tiên chúng tôi trích xuất CFG từ bytecode, rồi sử dụng API embedding của OpenAI để encode các nút và cạnh của CFG thành vector (3.8).

$$\mathbf{encode} = Encoder(edges, nodes) \tag{3.8}$$

Các vector sau khi encode có độ dài là 1536. Các vector này sau đó đi qua 3 lớp GCN với hàm kích hoạt ReLU (3.9), với lớp đầu tiên có độ dài đầu vào là 1536, độ dài đầu ra là biến hidden_channels tùy chỉnh.



Hình 3.4: *Phát hiện lỗ hổng bằng CFG dùng GNN.*

$$
\begin{aligned}
\mathbf{x} &= GCNConv(1536, hidden\_channels)(\mathbf{encode}) \\
\mathbf{x} &= Activation(relu)(\mathbf{x}) \\
\mathbf{x} &= GCNConv(hidden\_channels, hidden\_channels)(\mathbf{x}) \\
\mathbf{x} &= Activation(relu)(\mathbf{x}) \\
\mathbf{x} &= GCNConv(hidden\_channels, hidden\_channels)(\mathbf{x})
\end{aligned}
\tag{3.9}
$$

Sau đó, đầu ra của lớp GCN được đưa vào hàm phân loại tuyến tính để phân loại CFG với 3 loại nhãn (3.10).

$$\mathbf{linear} = Linear(hidden\_channels, 3)(\mathbf{x}) \tag{3.10}$$

Cuối cùng, để cho vào mô hình học sâu đa phương thức, chúng sẽ được đưa vào một lớp dense với độ dài là 64 và hàm kích hoạt ReLU rồi dùng hàm Flatten để làm phẳng nhằm đưa về dạng vector 1D làm đầu vào cho mô hình học sâu đa phương thức (3.11).

$$\begin{aligned} \mathbf{d\_gnn} &= Dense(64, relu)(\mathbf{linear}) \\ \mathbf{gnn\_output} &= Flatten()(\mathbf{d\_gnn}) \end{aligned} \tag{3.11}$$

3.1.4. Mô hình học sâu đa phương thức (VulnSense)

Cho đầu ra của mô hình BERT, BiLSTM và GNN được ký hiệu lần lượt là **bert_output** (3.3), **lstm_output** (3.7) và **gnn_output** (3.11) tương ứng. Sau đó, đầu ra kết hợp **c** có thể được biểu diễn như trong (3.12):

$$\mathbf{c} = \text{Concatenate}([\mathbf{bert\_output}, \mathbf{lstm\_output}, \mathbf{gnn\_output}]) \tag{3.12}$$

Đầu ra **c** sau đó được biến đổi thành tensor 3D có kích thước (batch_size, 194, 1) bằng cách sử dụng lớp Reshape (3.13):

$$\mathbf{c\_reshaped} = Reshape((194, 1))(\mathbf{c}) \tag{3.13}$$

Tiếp theo, tensor đã được biến đổi **c_reshaped** được truyền qua lớp tích chập 1D (3.14) với 64 filters và kích thước kernel là 3, sử dụng hàm kích hoạt tuyến tính chỉnh lưu:

$$\mathbf{conv\_out} = Conv1D(64, 3, relu)(\mathbf{c\_reshaped}) \tag{3.14}$$

Đầu ra từ lớp tích chập sau đó được làm phẳng (3.15) để tạo ra một vector 1D:

$$\mathbf{f\_out} = Flatten()(\mathbf{conv\_out}) \tag{3.15}$$

Tensor đã được làm phẳng **f_out** sau đó được truyền qua một lớp fully connected với độ dài 32 và hàm kích hoạt tuyến tính hiệu chỉnh như trong (3.16):

$$\mathbf{d\_out} = Dense(32, relu)(\mathbf{fl\_out}) \tag{3.16}$$

Cuối cùng, đầu ra được truyền qua hàm kích hoạt softmax (3.17) để tạo ra một phân phối xác suất trên ba lớp đầu ra.

$$\tilde{\mathbf{y}} = Dense(3, softmax)(\mathbf{d\_out}) \tag{3.17}$$

3.2. Xây dựng tập dữ liệu

Trong phần thực nghiệm trong đề tài này, chúng tôi thu thập được tập dữ liệu gồm 1769 hợp đồng thông mình có chứa các loại lỗi Arimetic, Reentrancy và không lỗi. Các hợp đồng chứa các loại lỗi đã đề cập trong **Phần 2.1.2**. **Hình 4.1** cung cấp cái nhìn tổng quan về tập dữ liệu của chúng tôi trong khoá luận lần này.

Trong tập dữ liệu này, chúng tôi kết hợp 3 tập dữ liệu sau: Smartbugs Curated [6, 8], SolidiFI-Benchmark [9], và Smartbugs Wild [6, 8]. Với tập dữ liệu Smartbugs Wild, chúng tôi chỉ thu thập các hợp đồng thông minh chỉ chứa duy nhất một lỗ hổng (hoặc lỗ hổng Arithmetic hoặc lỗ hổng Reentrancy). Việc hợp đồng thông minh đó được xác định có chứa lỗ hổng được xác nhận bởi ít nhất hai công cụ phát hiện lổ hổng trong hợp đồng thông mình hiện nay. Tổng cộng, tập dữ liệu của chúng tôi bao gồm 547 hợp đồng thông minh sạch, 631 hợp đồng chỉ có lỗ hổng Arimetic và 519 hợp đồng chỉ có lỗ hổng Reentrancy.

3.3. Xử lý dữ liệu

3.3.1. Mã nguồn

Khi lập trình, các lập trình viên thường có thói quen viết các comment để giải thích mã nguồn của họ, giúp họ và những lập trình viên khác có thể hiểu được đoạn mã nguồn. BERT là một mô hình xử lý ngôn ngữ tự nhiên và mã nguồn của hợp đồng thông minh là đầu vào cho mô hình BERT. Từ mã nguồn của hợp đồng thông minh, BERT tính toán sự liên quan của các từ trong mã nguồn. Những comment xuất hiện trong mã nguồn có thể gây nhiễu cho mô hình BERT, khiến mô hình tính toán những thông tin không cần thiết về mã nguồn của hợp đồng thông minh. Chính vì vậy, việc xử lý mã nguồn trước khi đưa vào mô hình là cần thiết. Ngoài ra, việc loại bỏ các comment khỏi mã nguồn cũng giúp giảm độ dài của đầu vào khi đưa vào mô hình. Để giảm độ dài mã nguồn thêm nữa, chúng tôi cũng loại bỏ các dòng trống thừa và khoảng trắng không cần thiết. **Hình 3.5** là một ví dụ cho một hợp đồng thông minh chưa được xử lý từ chúng tôi, hợp đồng này chứa các comment sau dấu "//", các dòng trống và các khoảng trắng dư thừa không đúng theo chuẩn lập trình.

Hợp đồng trên **Hình 3.5** sau khi được xử lý sẽ trở thành một hợp đồng không còn các đoạn comment, các dòng trống và khoảng trắng dư thừa như **Hình 3.6**.

3.3.2. Opcode

Chúng tôi tiến hành trích xuất bytecode từ mã nguồn của hợp đồng thông minh, sau đó trích xuất opcode thông qua bytecode. Các opcode trong hợp đồng được phân loại thành 10 nhóm chức năng với tổng cộng 135 opcodes theo Ethereum Yellow Paper [28]. Tuy nhiên, chúng tôi đã thu gọn chúng dựa trên **Bảng 3.1**. Trong quá trình tiền xử lý, chúng tôi đã loại bỏ các ký tự thập lục phân không cần thiết từ opcode. Mục đích của việc tiền xử lý này là sử dụng opcode để phát hiện các lỗ hổng trong hợp đồng thông minh bằng mô hình

```
pragma    solidity     ^0.4.10; // Version of this contract
contract IntegerOverflow
{
    mapping (address -> uint256) public    balanceOf;

    // This is a comment

    function transfer(address _to,       uint256 _value)
    {
        require (balanceOf[msg.sender] >= _value);
        balanceOf[msg.value] -= _value; // value use for ...
        balanceOf[_to] += _value;

    }

}
```

Hình 3.5: *Hợp đồng trước khi được xử lý*

```
pragma solidity ^0.4.10;
contract IntegerOverflow
{
    mapping (address -> uint256) public balanceOf;
    function transfer(address _to, uint256 _value)
    {
        require (balanceOf[msg.sender] >= _value);
        balanceOf[msg.value] -= _value;
        balanceOf[_to] += _value;
    }
}
```

Hình 3.6: *Hợp đồng sau khi được xử lý*

BiLSTM. Ngoài việc tiền xử lý opcode, chúng tôi cũng thực hiện các bước tiền xử lý khác để chuẩn bị dữ liệu cho mô hình BiLSTM. Đầu tiên, chúng tôi thực hiện token hóa opcode thành các chuỗi số nguyên. Sau đó, chúng tôi áp dụng padding để tạo ra các chuỗi opcodes cùng độ dài. Độ dài tối đa của các chuỗi opcode được đặt là 200, là độ dài tối đa mà mô hình BiLSTM có thể xử lý.

Opcodes thay thế	Opcodes ban đầu
ARITHMETIC_OP	ADD MUL SUB DIV SDIV SMOD MOD ADDMOD MULMOD EXP
CONSTANT1	BLOCKHASH TIMESTAMP NUMBER DIFFICULTY GASLIMIT COINBASE
CONSTANT2	ADDRESS ORIGIN CALLER
COMPARISON	LT GT SLT SGT
LOGIC_OP	AND OR XOR NOT
DUP	DUP1-DUP16
SWAP	SWAP1-SWAP16
PUSH	PUSH5-PUSH32
LOG	LOG1-LOG4

Bảng 3.1: *Bảng chuẩn hoá opcode*

Sau bước padding, chúng tôi sử dụng một lớp Word Embedding để chuyển đổi các chuỗi opcode được mã hóa thành các vector có kích thước cố định để làm đầu vào cho mô hình BiLSTM. Điều này cho phép mô hình BiLSTM học các biểu diễn của các chuỗi opcode tốt hơn.

Nói chung, các bước tiền xử lý mà chúng tôi thực hiện rất quan trọng trong việc chuẩn bị dữ liệu cho mô hình BiLSTM và cải thiện hiệu suất của nó trong việc phát hiện các lỗ hổng trong hợp đồng thông minh.

3.3.3. Control Flow Graph

Đầu tiên chúng tôi trích xuất bytecode từ hợp đồng thông minh, sau đó trích xuất CFG thông qua bytecode thành các file .cfg.gv như **Hình 3.7**. Từ file .cfg.gv này, chúng tôi tiếp tục trích xuất để lấy ra các nút và cạnh của CFG như **Hình 3.9** và **Hình 3.8**. Các nút trong CFG thường biểu thị các khối mã hoặc trạng thái của hợp đồng, trong khi các cạnh thể hiện các liên kết điều khiển giữa các nút.

Để huấn luyện mô hình GNN, chúng tôi mã hóa các nút và cạnh của CFG thành các vector số. Một cách tiếp cận là sử dụng phương pháp nhúng để biểu

```
digraph {
        subgraph global {
                node [fontname=Courier fontsize=30.0 rank=same
↪   shape=box]
                block_0 [label="0: PUSH1 0x80\l2: PUSH1 0x40\l4:
↪   MSTORE \l5: PUSH1 0x4\l7: CALLDATASIZE \l8: LT \l9: PUSH2
↪   0x4c\lc: JUMPI \l"]
                block_d [label="d: PUSH1 0x0\lf: CALLDATALOAD
↪   \l10: PUSH29 0x10 \l"]
                block_41 [label="41: DUP1 \l42: PUSH4
↪   0xf2fde38b\l47: EQ \l48: PUSH2 0xa8\l4b: JUMPI \l"]
                block_4c [label="4c: JUMPDEST \l4d: PUSH1
↪   0x0\l4f: DUP1 \l50: REVERT \l"]
                block_51 [label="51: JUMPDEST \l52: CALLVALUE
↪   \l53: DUP1 \l54: ISZERO \l55: PUSH2 0x5d\l58: JUMPI \l"]
                block_59 [label="59: PUSH1 0x0\l5b: DUP1 \l5c:
↪   REVERT \l"]
                block_5d [label="5d: JUMPDEST \l5e: POP \l5f:
↪   PUSH2 0x66\l62: PUSH2 0xeb\l65: JUMP \l"]
        }
        block_0 -> block_d [color=red]
        block_d -> block_51 [color=green]
        block_41 -> block_4c [color=red]
        block_0 -> block_4c [color=green]
        block_51 -> block_59 [color=red]
        block_d -> block_41 [color=red]
        block_51 -> block_5d [color=green]
}
```

Hình 3.7: *File code CFG được trích xuất từ bycode*

diễn các đối tượng này dưới dạng vector. Trong trường hợp này, chúng tôi sử dụng API embedding của OpenAI để mã hóa các nút và cạnh thành vector có độ dài 1536. Điều này có thể là một phương pháp tùy chỉnh dựa trên mô hình học sâu đã được huấn luyện trước của OpenAI. Khi các nút và cạnh của CFG đã được mã hóa thành vector, chúng tôi lấy chúng làm đầu vào cho mô hình GNN.

```
        block_0 -> block_d [color=red]
        block_d -> block_51 [color=green]
        block_41 -> block_4c [color=red]
        block_0 -> block_4c [color=green]
        block_51 -> block_59 [color=red]
        block_d -> block_41 [color=red]
        block_51 -> block_5d [color=green]
```

Hình 3.8: *Trích xuất cạnh từ CFG*

```
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH2
 ↪  0x4c JUMPI
PUSH1 0x0 CALLDATALOAD PUSH29 0x10
DUP1 PUSH4 0xf2fde38b EQ PUSH2 0xa8 JUMPI
JUMPDESTPUSH1 0x0 DUP1 REVERT
JUMPDEST CALLVALUE DUP1 ISZERO PUSH2 0x5d JUMPI
PUSH1 0x0 DUP1 REVERT
JUMPDEST POP PUSH2 0x66 PUSH2 0xeb JUMP
```

Hình 3.9: *Trích xuất nút từ CFG*

CHƯƠNG 4. THỰC NGHIỆM VÀ ĐÁNH GIÁ

Ở chương này chúng tôi trình bày thông tin về môi trường, cài đặt và đưa ra các tiêu chí đánh giá về mức độ hiệu quả của mô hình.

4.1. Thiết lập thực nghiệm

4.1.1. Môi trường thực nghiệm

Trong khoá luận này, thực nghiệm này thực hiện các xử lý phức tạp đòi hỏi chạy liên tục trong thời gian dài với hiệu suất CPU và GPU cao, dung lượng lưu trữ và bộ nhớ cao. Vì vậy, quá trình thực hiện thí nghiệm của chúng tôi nhận được sự hỗ trợ kịp thời về tài nguyên từ Phòng thí nghiệm An toàn thông tin InsecLab. Thông tin về môi trường được hỗ trợ được chúng tôi đề cập chi tiết trong **Bảng 4.1**.

Bảng 4.1: *Thông số môi trường thực nghiệm*

	InsecLab Vlab	Google Colab Pro Plus
CPU	4	2
GPU	N/A	V100
RAM	8GB	51GB
Hệ điều hành	Ubuntu 22.04	Ubuntu 20.04
Python	3.8.0	3.10.12

Thêm vào đó, thí nghiệm này còn cần thêm các thông tin về cài đặt tham số cho các mạng neural, được chúng tôi đề cập trong **Bảng 4.2**.

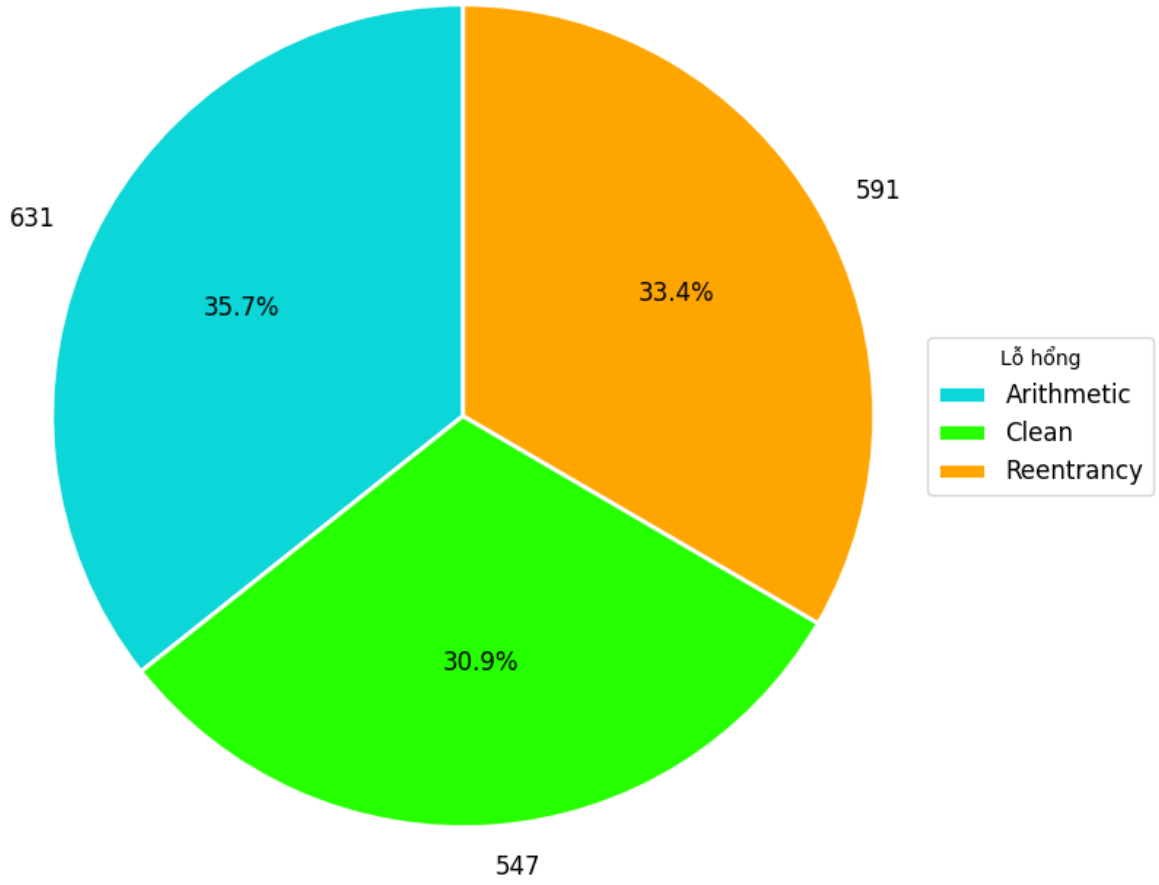
4.1.2. Tập dữ liệu

Trong các thí nghiệm của khoá luận này, chúng tôi xây dựng một bộ dữ liệu bao gồm ba thuộc tính: Mã nguồn, Opcode, CFG và Label, như đã đề cập trong

Bảng 4.2: *Giá trị các tham số chính trong quá trình thực nghiệm*

Tham số	Thông số
Epochs	30
Batch Size	32
Dropout	0.3
Optimizer	Adam

Phần 3.2. Cụ thể, chúng tôi có các thống kê chi tiết về tỷ lệ nhãn và các tham số dữ liệu liên quan khác cho bộ dữ liệu được thể hiện trong **Hình 4.1**.



Hình 4.1: *Phân bố các nhãn trong tập dữ liệu*

Hơn nữa, chúng tôi chia tập dữ liệu này thành hai tập, bao gồm tập huấn luyện và tập kiểm tra với tỷ lệ lần lượt là 80% và 20%. Phân bố các nhãn trong các tập huấn luyện và kiểm tra được hiển thị trong **Hình 4.1**.

4.1.3. Chỉ số đánh giá

Trong khoá luận này, để đánh giá hiệu suất của mô hình, chúng tôi sử dụng 6 chỉ số bao gồm: **Accuracy**, **Precision**, **Recall**, **F1-score**, **Thời gian đào tạo**, **Độ hội tụ** và **Thời gian dự đoán**. Để có thể tính toán được các giá trị Accuracy, Precision, Recall, F1-score, chúng tôi sử dụng thêm confusion matrix. Trong các bài toán phân loại, confusion matrix là một thành phần quan trọng giúp tính toán nhanh chóng các giá trị trên thông qua các khái niệm về True, False, Positive và Negative. Confusion matrix thể hiện được có bao nhiêu điểm dữ liệu thực sự thuộc vào một class, và được dự đoán là rơi vào một class. Từ confusion matrix, ta có được thêm các khái niệm, gồm:

- **True Positive:** là những mẫu được dự đoán đúng là mẫu A so với thực tế chính là mẫu A.
- **True Negative:** là những mẫu được dự đoán không phải là mẫu A và thực tế không phải mẫu A.
- **False Positive:** là những mẫu được dự đoán là mẫu A nhưng thực tế không phải là mẫu A.
- **False Negative:** là những mẫu được dự đoán không phải mẫu A nhưng thực tế chính là mẫu A.

Các chỉ số đánh giá Accuracy, Precision, Recall, F1-Score được định nghĩa và tính toán dựa trên số lượng các mẫu được xem là True Positive, True Negative, False Positive và Fasle Negative. Trong đó:

1. **Accuracy** là tỷ lệ phần trăm các mẫu được dự đoán đúng trong tổng số mẫu. Khoảng giá trị của Accuracy trong khoảng [0,1], giá trị càng lớn, khả năng dự đoán của mô hình càng tốt. Công thức của Accuracy được tính theo công thức:

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

2. **Precision** là tỷ lệ số mẫu được tính là True Positive với tổng số mẫu được phân loại là Positive. Khoảng giá trị của Precision trong khoảng [0,1], giá trị càng lớn, độ chính xác của các mẫu tìm được càng cao. Công thức của Precision được tính theo công thức:

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

3. **Recall** là tỷ lệ số mẫu True Positive được dự đoán đúng trong tất cả các mẫu được dự đoán. Khoảng giá trị của Precision trong khoảng [0,1], Recall cao có nghĩa tỉ lệ bỏ sót các mẫu positive thực tế thấp. Công thức của Recall được tính theo công thức:

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

4. **F1-score** là trung bình điều hòa giữa **Precision** và **Recall**. Như vậy chúng ta đã có hai khái niệm Precision và Recall và mong muốn hai giá trị này càng cao càng tốt. Tuy nhiên trong thực tế nếu ta điều chỉnh model để tăng Recall quá mức có thể dẫn đến Precision giảm và ngược lại, cố điều chỉnh model để tăng Precision có thể làm giảm Recall. F1-score sinh ra để cân bằng 2 đại lượng này. Trong đó giá trị gần 1 cho thấy mô hình phân loại có hiệu suất cao, và giá trị gần 0 cho thấy mô hình phân loại có hiệu suất thấp. Công thức của Recall được tính theo công thức:

$$\textbf{Recall} = \frac{2 * Precision * Recall}{Precision + Recall}$$

Bên cạnh những chỉ số được tính toán từ các giá trị trong confusion matrix, chúng tôi còn đánh giá các giá trị về mặt thời gian. Các giá trị này gồm:

1. **Thời gian đào tạo** là thời gian mô hình thực hiện đào tạo qua 30 epochs với các thông số được nhóm thiết lập và trình bày trong **Bảng 4.2**. Ngoài việc so sánh về các chỉ số liên quan đến hiệu suất, độ chính xác, việc quan tâm đến thời gian đào tạo cũng là một yếu tố thiết thực trong việc áp dụng mô hình vào thực tiễn.

2. **Độ hội tụ** là số lượng epochs ít nhất mà mô hình cần thực hiện để đạt giá trị tốt nhất. Tương tự với thời gian đào tạo, việc đạt được độ hội tụ sớm giúp giảm được thời gian đào tạo khi hiện nay có nhiều hàm **early_stop** giúp chúng ta dừng quá trình đào tạo khi mô hình đã đạt độ hội tụ tốt.

3. **Thời gian dự đoán** là thời gian mô hình thực hiện dự đoán trên tập test của tập dữ liệu. Bên cạnh **Thời gian đào tạo** và **Độ hội tụ**, **Thời gian dự đoán** cũng là một đại lượng cần quan tâm khi ứng dụng mô hình trong thực tiễn.

4.1.4. Ngữ cảnh thực nghiệm

Trong quá trình thực nghiệm, để đi tìm đáp án cho câu hỏi: "Liệu rằng việc sử dụng mô hình học sâu kết hợp đa phương thức biểu diễn của hợp đồng thông minh có đạt kết quả tốt hơn so với mô hình học sâu đơn phương thức?".

Chúng tôi thực hiện huấn luyện 7 mô hình, được chia làm 2 loại: mô hình học sâu đơn phương thức và mô hình học sâu đa phương thức.

Trong đó, mô hình học sâu đơn phương thức là các mô hình trong từng nhánh của VulnSense:

- BiLSTM
- BERT

- GNN

Các mô hình học sâu đa phương thức chúng tôi tiến hành thực nghiệm là các mô hình kết hợp theo tổ hợp chập 2 của 3 mô hình học sâu đơn phương thức và VulnSense, trong đó:

- **M1:** Multimodal BERT - BiLSTM
- **M2:** Multimodal BERT - GNN
- **M3:** Multimodal BiLSTM - GNN
- **VulnSense:** đã được đề cập trong **Phần 3.1.4**

Quá trình thực nghiệm các mô hình được thực hiện trên tập dữ liệu đã được đề cập trong **Phần 4.1.2**. Nhằm đảm bảo tính công bằng, chúng tôi thực hiện tổng cộng 30 epochs trên từng mô hình, và với 10 epochs, chúng tôi dừng lại và ghi nhận kết quả là các chỉ số đánh giá được đề cập trong **Phần 4.1.3** để có thể so sánh được độ hội tụ của các mô hình trong kịch bản thực nghiệm.

4.2. Kết quả thực nghiệm

Trong phần này, chúng tôi thực nghiệm, ghi lại kết quả của các mô hình học sâu mà chúng tôi đã đề cập trong **Phần 4.1.4** dựa theo các tiêu chí trong **Phần 4.1.3**.

4.2.1. So sánh điểm Accuracy, Precision, Recall và F1

Qua các **Hình** [**4.2, 4.3, 4.4, 4.5**], một cách trực quan, có thể trả lời được câu hỏi đặt ra trong **Phần 4.1.4** rằng khả năng nhận diện lỗ hổng trong hợp đồng thông minh sử dụng mô hình học sâu đa phương thức hiệu quả hơn mô hình học sâu đơn phương thức trong ngữ cảnh thực nghiệm của Khoá luận này. Dựa vào **Bảng 4.3**, có thể tính được các giá trị **Accuracy**, **Precision**, **Recall**

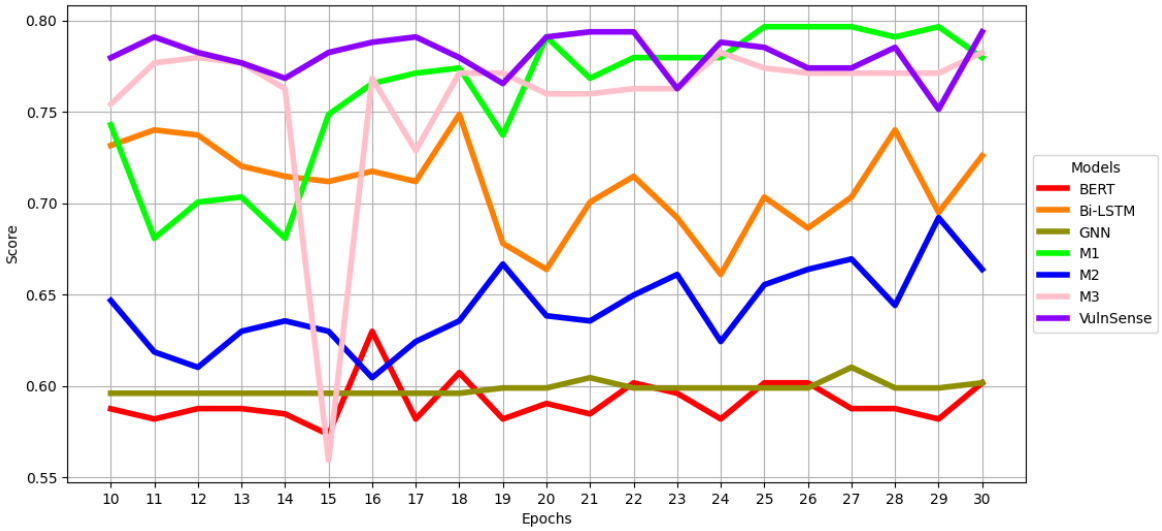
và **F1-Score** trung bình của 4 mô hình học sâu đa phương thức cao hơn trung bình của 3 mô hình học sâu đơn phương thức, cụ thể:

- **Accuracy:** Cao hơn khoảng 16.57%
- **Precision:** Cao hơn khoảng 19.13%
- **Recall:** Cao hơn khoảng 16.36%
- **F1-Score** Cao hơn khoảng 23.41%

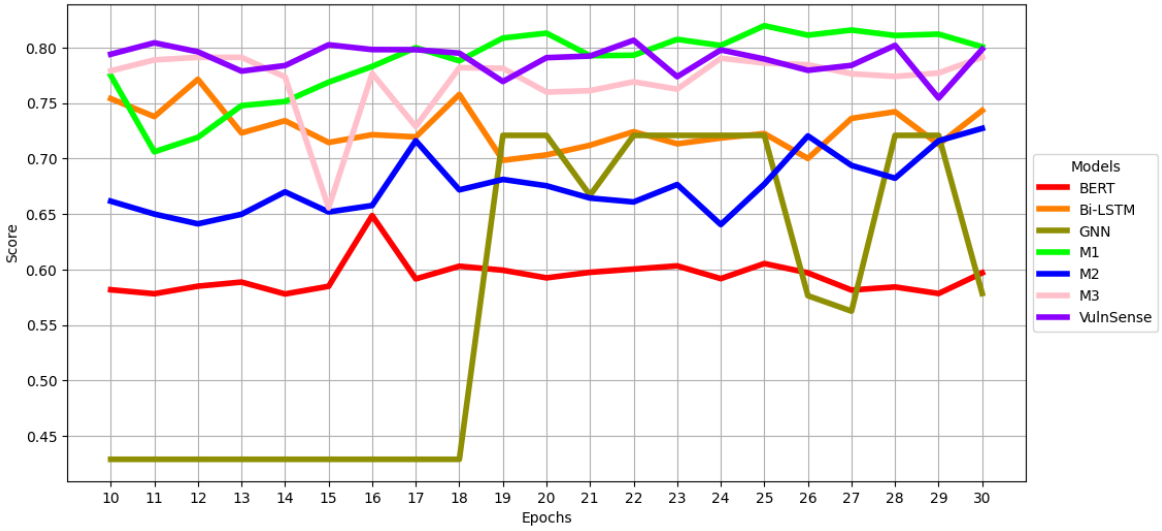
Ngoài ra, tập trung vào các chỉ số giữa 4 mô hình học sâu đa phương thức, có thể thấy rằng mô hình VulnSense đạt điểm cao nhất trong 4 chỉ số **Accuracy**, **Precision**, **Recall** và **F1-Score**. Bên cạnh đó, mô hình học sâu đa phương thức M3 và M1 cũng cho thấy kết quả tốt khi theo sát kết quả của VulnSense. Riêng mô hình M2 các chỉ số so sánh khá thấp so với phần còn lại. Cụ thể:

- **Accuracy:** VulnSense đạt giá trị cao nhất là 0.7796 hơn 1.59% so với M1, hơn 1.72% so với M3, và 19.66% so với M2. (sắp xếp theo thứ tự điểm giảm dần giữa M1, M2, M3) (**Hình 4.2**)
- **Precision:** VulnSense đạt giá trị cao nhất là 0.794, hơn 1.19% so với M1, hơn 1.59% so với M3 và hơn 15.89% so với M2. (sắp xếp theo thứ tự điểm giảm dần giữa M1, M2, M3) (**Hình 4.3**)
- **Recall:** VulnSense đạt giá trị cao nhất là 0.7797, hơn 1.12% so với M3, hơn 1.62% so với M1 và hơn 19.05% so với M2. (sắp xếp theo thứ tự điểm giảm dần giữa M1, M2, M3) (**Hình 4.4**)
- **F1-Score** VulnSense đạt giá trị cao nhất là 0.7830, hơn 1.02% so với M3, hơn 1.35% so với M1 và hơn 18.91% so với M2. (sắp xếp theo thứ tự điểm giảm dần giữa M1, M2, M3) (**Hình 4.5**)

Đây chính là các giá trị trung bình của mô các mô hình, để có thể tính toán chi tiết hiệu suất phát hiện các loại nhãn, chúng tôi đã cung cấp các confusion matrix của các mô hình trong **Hình 4.8**.



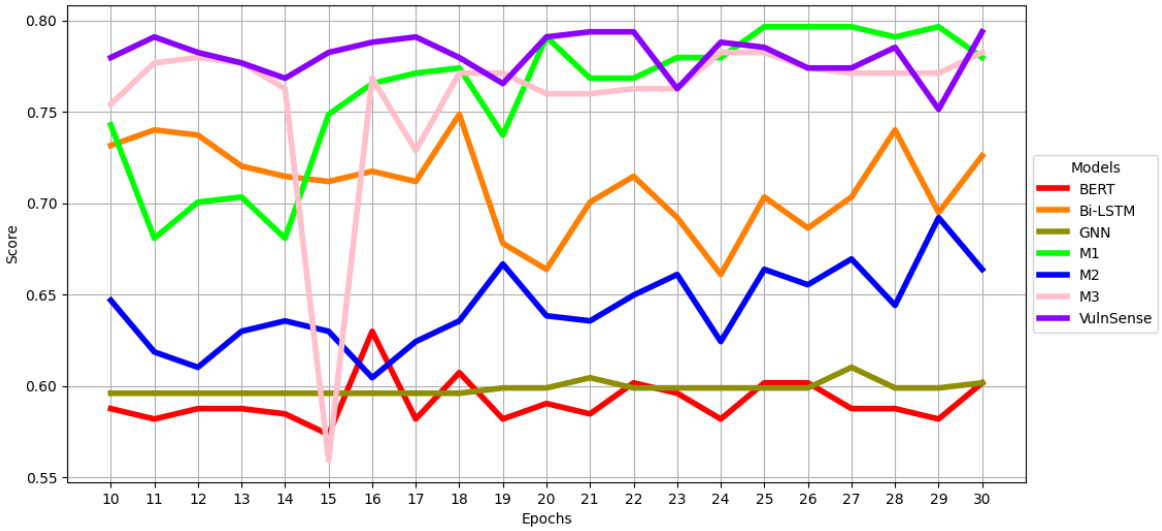
Hình 4.2: *Biểu đồ so sánh Accuracy của các mô hình*



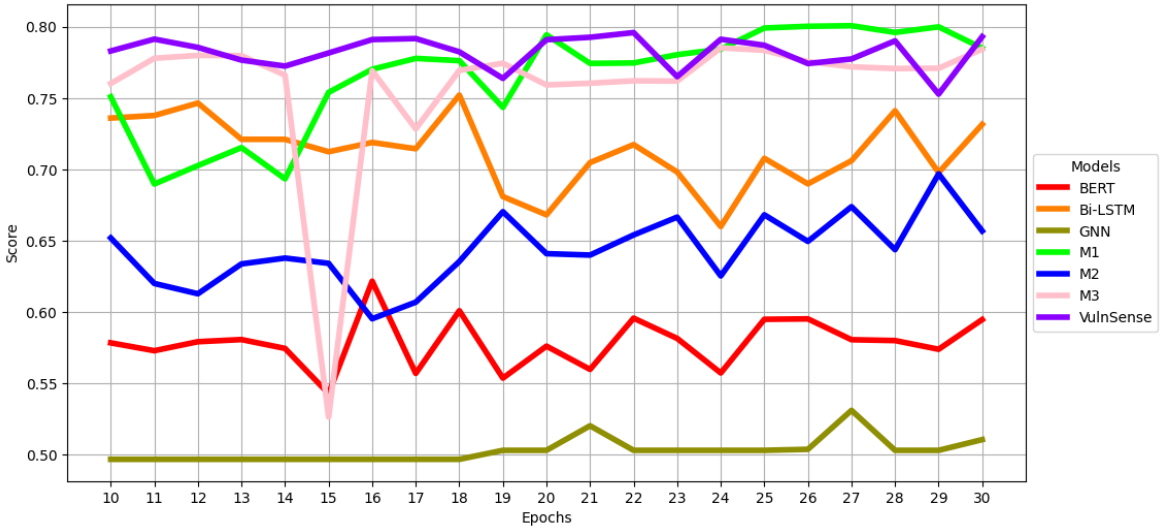
Hình 4.3: *Biểu đồ so sánh Precision của các mô hình*

4.2.2. So sánh thời gian đào tạo

Hình 4.6 thể hiện thời gian huấn luyện 30 epochs của mỗi mô hình. Xét về thời gian đào tạo, không có sự phân hoá rõ ràng giữa mô hình học sâu đa phương thức và mô hình học sâu đơn phương thức. Điểm đáng chú ý là thời gian đào tạo của mô hình GNN là rất ngắn chỉ 7.114 giây, trái ngược với đó thời gian đào tạo của mô hình BERT là rất cao lên tới 252.814 giây.



Hình 4.4: *Biểu đồ so sánh Recall của các mô hình*

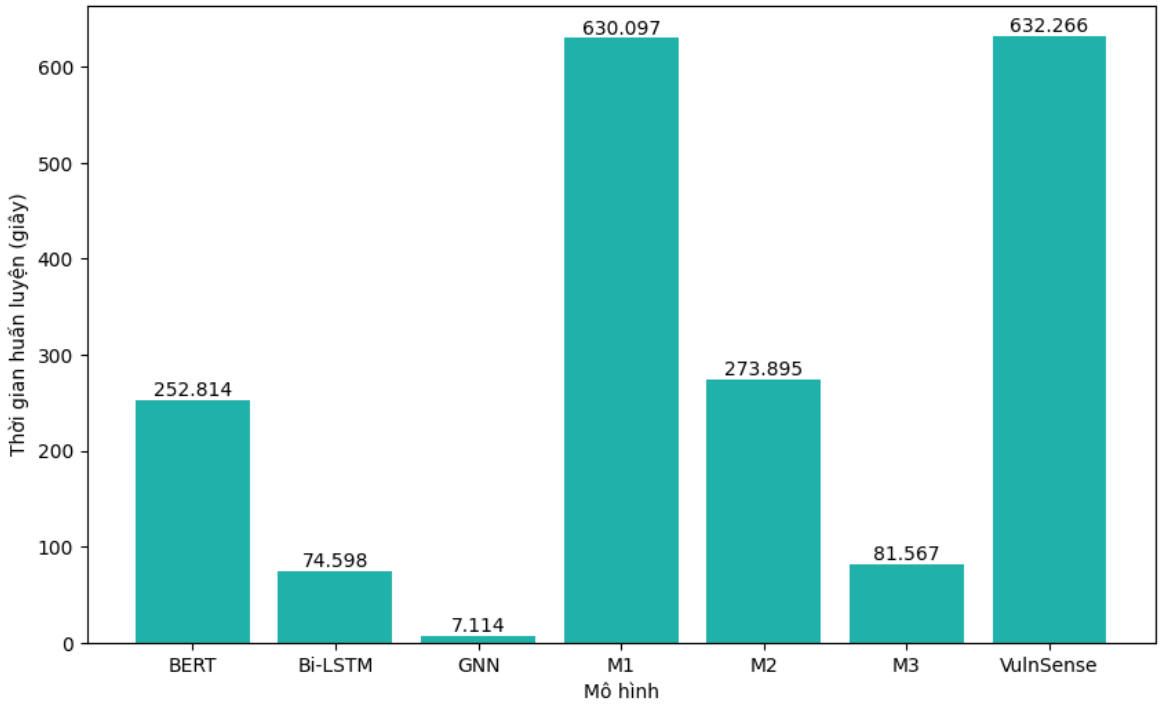


Hình 4.5: *Biểu đồ so sánh F1 của các mô hình*

Khi so sánh giữa các mô hình đa phương thức. Cùng với sự dẫn đầu về điểm số thì mô hình VulnSense cũng có thời gian huấn luyện lâu nhất với 632.266 giây cho 30 epochs, xếp theo sau là mô hình M1 (mô hình đa phương thức BERT và BiLSTM) với 630.097 giây. Mô hình có thời gian huấn luyện ngắn nhất là M3 (mô hình đa phương thức BiLSTM và GNN) với thời gian là 81.567 giây.

Có thể thấy rằng thời gian huấn luyện của mô hình đơn phương thức ảnh hưởng khá nhiều đến thời gian huấn luyện mô hình đa phương thức mà mô hình

đơn phương thức đó tham gia. Như trong **Hình 4.6**, mô hình M1, M2, và mô hình VulnSense có sự tham gia của mô hình BERT nên có thời gian huấn luyện khá lâu (trên 200 giây cho 30 epochs).



Hình 4.6: *Biểu đồ so sánh thời gian huấn luyện 30 epochs của các mô hình*

4.2.3. So sánh độ hội tụ

Trong **Bảng 4.3**, VulnSense và mô hình đơn phương thức đạt được giá trị cao nhất khi hoàn thành 10 epochs. Trong khi đó, 2 mô hình đơn phương thức còn lại là BERT và GNN cần đến 30 epochs mới có thể đạt giá trị tốt nhất. Về các mô hình đa phương thức M1, M2, M3, các mô hình này cần 20 epochs để đạt được giá trị tốt nhất.

Từ đó có thể thấy được các mô hình học sâu đa phương thức đạt được giá trị tốt với số lượng epochs ít hơn so với các mô hình học sâu đơn phương thức. Điều đó sẽ làm rút ngắn thời gian huấn luyện trên các mô hình học sâu đa phương thức.

Bảng 4.3: *Hiệu xuất của 7 mô hình trong các kịch bản thực nghiệm khác nhau.*

Score	Epoch	BERT	BiLSTM	GNN	M1	M2	M3	VulnSense
Accuracy	**E10**	0.5875	0.7316	0.5960	0.7429	0.6468	0.7542	**0.7796**
	E20	0.5903	0.6949	0.5988	0.7796	0.6553	0.7768	**0.7796**
	E30	0.6073	0.7146	0.6016	0.7796	0.6525	0.7683	**0.7796**
Precision	**E10**	0.5818	0.7540	0.4290	0.7749	0.6616	0.7790	**0.7940**
	E20	0.6000	0.7164	0.7209	0.7834	0.6800	0.7800	**0.7922**
	E30	0.6000	0.7329	0.5784	0.7800	0.7000	0.7700	**0.7800**
Recall	**E10**	0.5876	0.7316	0.5960	0.7429	0.6469	0.7542	**0.7797**
	E20	0.5900	0.6949	0.5989	0.7797	0.6600	0.7800	**0.7797**
	E30	0.6100	0.7147	0.6017	0.7700	0.6500	0.7700	**0.7700**
F1	**E10**	0.5785	0.7360	0.4969	0.7509	0.6520	0.7602	**0.7830**
	E20	0.5700	0.6988	0.5032	0.7809	0.6600	0.7792	**0.7800**
	E30	0.6000	0.7185	0.5107	0.7700	0.6500	0.7700	**0.7700**

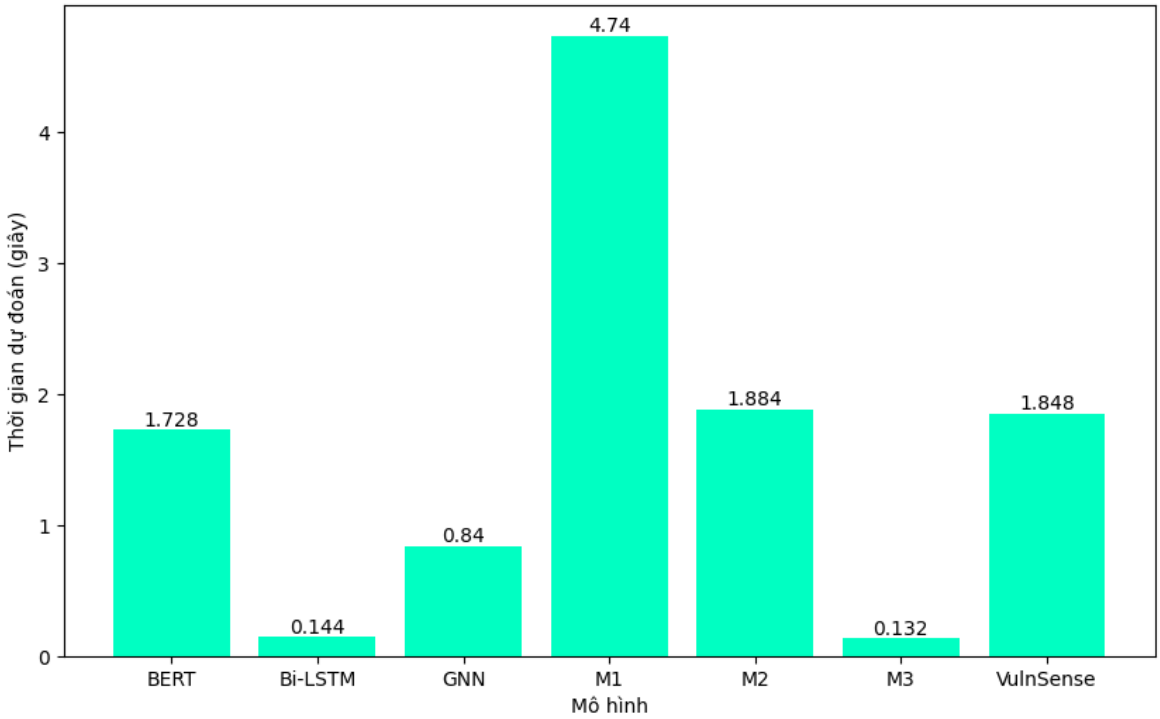
4.2.4. So sánh thời gian dự đoán

Hình 4.7 thể hiện thời gian dự đoán trên tập test của mỗi mô hình. Xét về thời gian dự đoán, không có sự phân hoá rõ ràng giữa mô hình học sâu đa phương thức và mô hình học sâu đơn phương thức. Điểm đang chú ý là thời gian dự đoán của mô hình Bi-LSTM và M3 là rất ngắn chỉ khoảng 0.144 và 0.132 giây, trái ngược với đó thời gian đào tạo của mô hình M1 là cao nhất lên tới 4.74 giây. Về thời gian dự đoán ở mức trung bình, mô hình BERT, M2 và VulnSense khá tương đồng nhau với thời gian dự đoán lần lượt là 1.728, 1.884 và 1.848 giây.

Khi so sánh giữa các mô hình đa phương thức. có sự cách biệt giữa mô hình M1 và M3 khi thời gian dự đoán của mô hình M1 hơn M3 lên đến 4,608 giây. Trong khi đó M2 và VulnSense có thời gian dự đoán khá tương đồng nhau khoảng 1.8 giây.

Có thể thấy rằng thời gian dự đoán của mô hình đơn phương thức không trực tiếp ảnh hưởng đến thời gian dự đoán của mô hình đa phương thức mà mô hình đơn phương thức đó tham gia. Như trong **Hình 4.7**, mô hình M1, M2, và mô hình VulnSense có sự tham gia của mô hình BERT nhưng chỉ có thời gian dự đoán của mô hình M1 là tăng đột biết, còn M2 và VulnSense không chênh lệch

nhiều so với mô hình BERT.



Hình 4.7: *Biểu đồ so sánh thời gian dự đoán trên tập test của các mô hình*

4.3. Thảo luận

Kết thúc quá trình thực nghiệm, kết quả thu được trong việc phân loại hợp đồng thông mình với ba nhãn: Clean, Arithmetic và Reentrancy với từng mô hình được trình bày chi tiết trong **Bảng 4.3**. Có thể thấy VulnSense đạt được kết quả với 4 chỉ số **Accuracy**, **Precision**, **Recall** và **F1-Score** đều đạt khoảng 0.78 khả quan hơn so với ba mô hình đơn phương thức BERT, BiLSTM, GNN.

Xét về thời gian đào tạo trên toàn bộ 30 epochs, mặc dù VulnSense tốn nhiều thời gian hơn so với ba mô hình đơn phương thức, nhưng VulnSense chỉ cần 10 epochs để đạt độ hội tụ. Điều đó làm giảm đáng kể thời gian đào tạo của VulnSense so với ba mô hình đơn phương thức. Ngoài ra, thời gian huấn luyện của mô hình đơn phương thức ảnh hưởng khá nhiều đến thời gian huấn luyện mô hình đa phương thức mà mô hình đơn phương thức đó tham gia. Đặc biệt

là mô hình BERT tốn khá nhiều thời gian huấn luyện dẫn đến các mô hình đa phương thức có sử dụng mô hình BERT cũng có thời gian huấn luyện cao hơn.

Xét về thời gian dự đoán trên tập test, mô hình M3 đạt kết quả vượt trội hơn so với các mô hình khác kể cả VulnSense, khi chỉ tốn khoảng 0.1 giây trong việc dự đoán trên tập test gồm 340 hợp đồng thông minh. Về phương diện thời gian dự đoán, các mô hình học sâu đa phương thức chỉ có mô hình M3 là vượt trội so với các mô hình học sâu đơn phương thức.

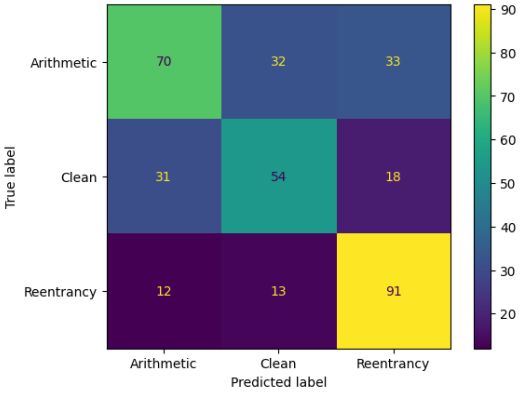
Khi cân bằng giữa thời gian huấn luyện, thời gian dự đoán và các chỉ số đánh giá khác, có thể nhận định rằng mô hình đa phương thức BiLSTM - GNN tối ưu hơn VulnSense trong việc ứng dụng thực tế.

Ngoài ra, để có thể chi tiết hơn trong việc nhận định mô hình nào có khả năng nhận diện loại nhãn nào tốt hơn, các confusion matrix trong **Hình 4.8** giúp đánh giá khía cạnh này một cách tốt hơn. Khi xét từng loại nhãn qua các mô hình trong kịch bản thử nghiệm, ta có:

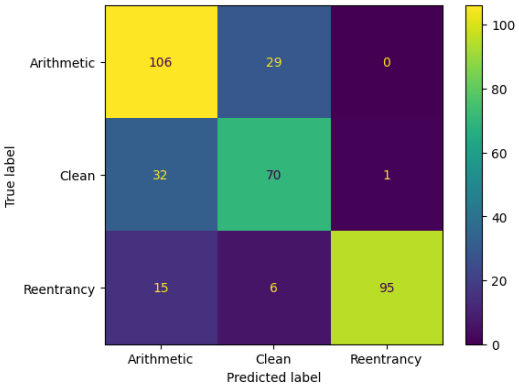
- Về **Arithmetic:** Mô hình GNN thể hiện giá trị tốt nhất khi nhận diện được chính xác 115 mẫu là Arithmetic và chỉ nhận diện sai 20 mẫu. VulSense chỉ chênh lệnh 1 mẫu đúng chuyển thành sai so với số mẫu từ GNN. Mô hình M2 thể hiện giá trị xấu nhất khi chỉ nhận diện đúng 54 mẫu và sai đến 81 mẫu.

- Về **Reentrancy:** Mô hình M3 thể hiện giá trị tốt nhất khi nhận diện được chính xác 99 mẫu là Reentrancy và chỉ nhận diện sai 17 mẫu. VulSense chỉ chênh lệnh 1 mẫu đúng chuyển thành sai so với số mẫu từ M3. Về lỗi Reentrancy, các mô hình khá đồng đều khi các mô hình đều nhận diện đúng từ 90 mẫu trở lên.

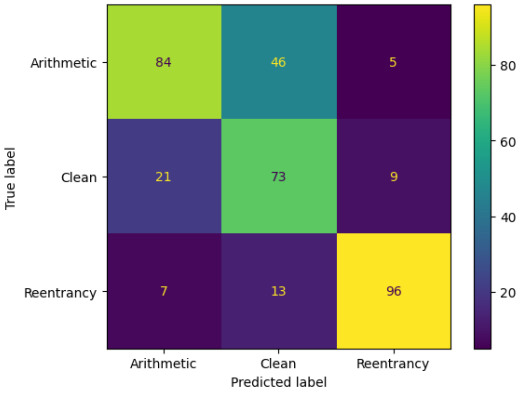
- Về **Clean:** Mô hình M2 thể hiện giá trị tốt nhất khi nhận diện chính xác 93 mẫu và chỉ nhận diện sai 10 mẫu. Trong khi đó, GNN thể hiện giá trị xấu nhất khi chỉ nhận diện đúng 2 mẫu và nhận diện sai lên đến 101 mẫu.



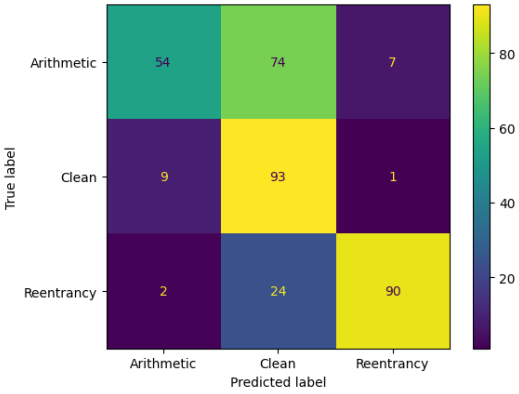
(a) BERT



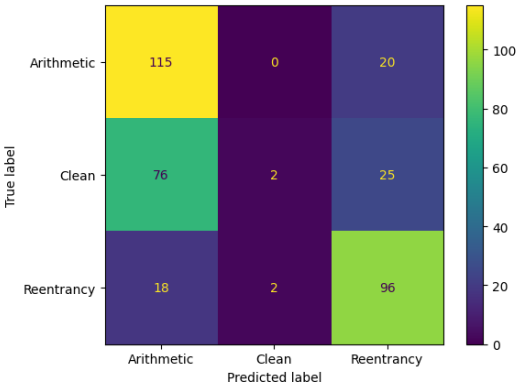
(b) M1



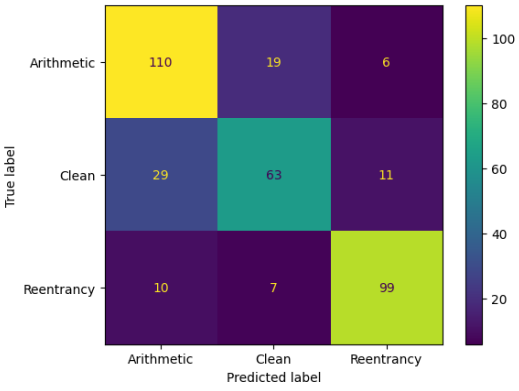
(c) BiLSTM



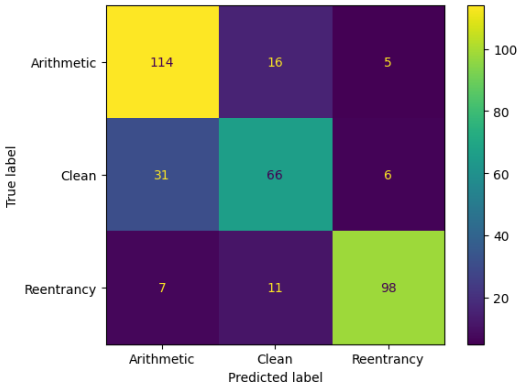
(d) M2



(e) GNN



(f) M3



(g) VulnSense

Hình 4.8: *Confusion matrices tại epoch thứ 30, với (a), (c), (e) là các mô hình đơn phương thức và (b), (d), (f), (g) là các mô hình đa phương thức*

CHƯƠNG 5. KẾT LUẬN

Ở chương này, chúng tôi đưa ra những kết luận về nghiên cứu, những hạn chế, và đồng thời đưa ra hướng cải thiện và phát triển.

5.1. Kết luận

Trong khoá luận này, chúng tôi đã trình bày một phương pháp mới cho việc phát hiện lỗ hổng hiệu quả trong hợp đồng thông minh bằng cách sử dụng mô hình học sâu đa phương thức mang tên VulnSense. Phương pháp đề xuất của chúng tôi kết hợp ba loại đặc trưng trong hợp đồng thông minh, bao gồm mã nguồn, Opcode và Control Flow Graph, và sử dụng các mô hình BERT, BiLSTM và GNN để trích xuất và phân tích các đặc trưng này. Chúng tôi đã tiến hành các thực nghiệm trên bộ dữ liệu tự thu thập chứa một số lượng lớn các hợp đồng thông minh Ethereum có tồn tại lỗ hổng Arithmetic, Reentrancy và không tồn tại lỗ hổng. Kết quả thực nghiệm cho thấy phương pháp đề xuất của chúng tôi đạt kết quả tốt hơn so với các mô hình đơn phương thức trong tập dữ liệu nhóm đã thu thập. Cụ thể, hiệu suất của mô hình của chúng tôi cao hơn khoảng 26% so với ba mô hình đơn phương thức trên trung bình tất cả các chỉ số và cao hơn khoảng 7% so với 3 mô hình đa phương thức còn lại trên trung bình tất cả chỉ số. Phương pháp của chúng tôi giải quyết các hạn chế của các mô hình phát hiện lỗ hổng dựa trên máy học hiện có cho hợp đồng thông minh, và cũng chứng minh tính hiệu quả và tiềm năng của các phương pháp học sâu đa phương thức trong việc phát hiện lỗ hổng trong hợp đồng thông minh. Tuy nhiên, bên cạch đó, mô hình VulnSense của chúng tối vẫn còn những hạn chế nhất định. Đó là, chỉ phát hiện được hợp đồng thồng minh có chứa lỗ hổng Arithmetic và Reentrancy được viết bằng ngôn ngữ Solidity và triển khai trên Ethereum và

hiệu suất của mô hình vẫn có thể cải thiện.

5.2. Hướng phát triển

Trong tương lai, chúng tôi dự định mở rộng mô hình VulnSense, giúp cho mô hình này có thể phát hiện thêm được các lỗ hổng khác có trong hợp đồng thông minh chẳng hạn như: Transaction Ordering Dependency, Time manipulation,.. Đồng thời, cải thiện hiệu suất của mô hình lên tối đa bằng cách tối ưu giai đoạn tiền xử lý dữ liệu, đặc biệt là xử lý mã nguồn trước khi đưa vào mô hình BERT. Xa hơn nữa, chúng tôi mong muốn VulnSense không chỉ dừng lại với việc phát hiện lổ hổng trên hợp đồng thông mình được viết bằng ngôn ngữ Solidity và triển khai trên Ethereum mà VulnSense còn có thể sử dụng cho các nền tảng blockchain khác và cung cấp một giải pháp đáng tin cậy và hiệu quả cho các nhà phát triển và kiểm toán viên hợp đồng thông minh. Ngoài ra, chúng tôi sẽ phát triển mô hình VulnSense thành mô hình Semi-supervised active learning nhằm giảm thời gian và công sức cho việc gán nhãn dữ liệu huấn luyện và cải thiện mô hình một cách tự động.

TÀI LIỆU THAM KHẢO

Tài liệu

[1] URL: https://cims.nyu.edu/~sbowman/multinli/.

[2] URL: https://rajpurkar.github.io/SQuAD-explorer/.

[3] Jiachi Chen et al. (), "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode", *IEEE Transactions on Software Engineering*, 48 (7).

[4] ConsenSys, *Consensys/mythril: Security Analysis Tool for EVM bytecode. supports smart contracts built for Ethereum, Hedera, quorum, Vechain, Roostock, Tron and other EVM-compatible blockchains.* URL: https://github.com/ConsenSys/mythril.

[5] Jacob Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", in: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, DOI: 10.18653/v1/N19-1423.

[6] Thomas Durieux et al., "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts", in: *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[7] Josselin Feist, Gustavo Grieco, and Alex Groce (2019), "Slither: A Static Analysis Framework For Smart Contracts".

[8] João F Ferreira et al., "SmartBugs: A framework to analyze solidity smart contracts", in: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[9] Asem Ghaleb and Karthik Pattabiraman, "How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection", in: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[10] Daniel Gibert, Carles Mateu, and Jordi Planes (2020), "HYDRA: A multimodal deep learning framework for malware classification", *Computers Security*, 95, p. 101873, ISSN: 0167-4048, DOI: https://doi.org/10.1016/j.cose.2020.101873.

[11] Summaira Jabeen et al. (2023), "A Review on Methods and Applications in Multimodal Deep Learning", *ACM Transactions on Multimedia Computing, Communications and Applications*, 19 (2s), pp. 1–41.

[12] Bo Jiang, Ye Liu, and W. K. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection", in: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, Montpellier, France: Association for Computing Machinery, 2018, 259–269, ISBN: 9781450359375, DOI: 10.1145/3238147.3238177.

[13] Fan Jiang et al. (2023), "Enhancing Smart-Contract Security through Machine Learning: A Survey of Approaches and Techniques", *Electronics*, 12 (9), p. 2046.

[14] Wanqing Jie et al. (2023), "A novel extended multimodal AI framework towards vulnerability detection in smart contracts", *Information Sciences*, 636, p. 118907, ISSN: 0020-0255, DOI: https://doi.org/10.1016/j.ins.2023.03.132.

[15] Mohammad Khodadadi and Jafar Tahmoresnezhad (2023), "HyMo: Vulnerability Detection in Smart Contracts using a Novel Multi-Modal Hybrid Model", *arXiv preprint arXiv:2304.13103*.

[16] Satpal Kushwaha et al. (2022), "Ethereum Smart Contract Analysis Tools: A Systematic Review", *IEEE Access*, pp. 1–1, DOI: 10.1109/ACCESS.2022.3169902.

[17] Satpal Singh Kushwaha et al. (2022), "Ethereum Smart Contract Analysis Tools: A Systematic Review", *IEEE Access*, 10, pp. 57037–57062, DOI: 10.1109/ACCESS.2022.3169902.

[18] Satpal Singh Kushwaha et al. (2022), "Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract", *IEEE Access*, 10, pp. 6605–6621, DOI: 10.1109/ACCESS.2021.3140091.

[19] Oliver Lutz et al. (2021), "ESCORT: ethereum smart contracts vulnerability detection using deep neural network and transfer learning", *arXiv preprint arXiv:2103.12607*.

[20] Loi Luu et al. (2016), "Making Smart Contracts Smarter", https://eprint.iacr.org/2016/633, DOI: 10.1145/2976749.2978309.

[21] Izhar Mehar et al. (2019), "Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack", *Journal of Cases on Information Technology*, 21, pp. 19–32, DOI: 10.4018/JCIT.2019010102.

[22] *Nveloso/conkas: Ethereum Virtual Machine (EVM) bytecode or solidity smart contract static analysis tool based on symbolic execution.*

[23] Peng Qian et al. (2020), "Towards automated reentrancy detection for smart contracts based on sequential models", *IEEE Access*, 8, pp. 19685–19695.

[24] *Solidity*, URL: https://docs.soliditylang.org/en/develop/index.html.

[25] Petar Tsankov et al., "Securify: Practical security analysis of smart contracts", in: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[26] Wei Wang et al. (2020), "Contractward: Automated vulnerability detection models for ethereum smart contracts", *IEEE Transactions on Network Science and Engineering*, 8 (2), pp. 1133–1144.

[27] Wikipedia contributors, *Solidity — Wikipedia, The Free Encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Solidity&oldid=1153327018`, [Online; accessed 18-June-2023], 2023.

[28] Daniel Davis Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER", in: 2014.

[29] Lejun Zhang et al. (2022), "A Novel Smart Contract Vulnerability Detection Method Based on Information Graph and Ensemble Learning", *Sensors*, 22 (9), ISSN: 1424-8220, DOI: 10.3390/s22093581.

[30] Weiqin Zou et al. (), "Smart contract development: Challenges and opportunities", *IEEE Transactions on Software Engineering*, 47 (10).