DOI: 10.31276/VJST.2024.0019 ***Khoa học tự nhiên*** | *Khoa học máy tính và thông tin, Kỹ thuật điện, kỹ thuật điện tử, kỹ thuật thông tin*

Phát hiện lỗ hổng phần mềm nhờ học sâu

Bùi Văn Công[1*], Vũ Thảo Nguyên[2], Vũ Đức Minh[3], Nguyễn Phương Lan[4]

[1]*Khoa Công nghệ Thông tin, Trường Đại học Kinh tế - Kỹ thuật Công nghiệp, 456 Minh Khai, phường Vĩnh Tuy, quận Hai Bà Trưng, Hà Nội, Việt Nam*

[2]*Trường Đại học Công nghệ Nanyang, 50 Đại lộ Nanyang, Singapore*

[3]*Trường THPT Cầu Giấy, 8/118 Nguyễn Khánh Toàn, phường Quan Hoa, quận Cầu Giấy, Hà Nội, Việt Nam*

[4]*Trường Quốc tế Nhật Bản, 84A Nguyễn Thanh Bình, phường Vạn Phúc, quận Hà Đông, Hà Nội, Việt Nam*

Ngày nhận bài 21/10/2024; ngày chuyển phản biện 23/10/2024; ngày nhận phản biện 5/11/2024; ngày chấp nhận đăng 29/11/2024

Tóm tắt:

Việc phát triển các dự án phần mềm thành công luôn là mối quan tâm hàng đầu của các tổ chức, doanh nghiệp. Trong đó, việc đảm bảo chất lượng của phần mềm là ưu tiên cao nhất trong toàn bộ quá trình phát triển cũng như vận hành của phần mềm. Bài báo đề cập việc phát hiện các lỗ hổng của mã nguồn và tập trung vào phân tích cú pháp và ngữ nghĩa của các câu lệnh trong mã nguồn. Mô hình phát hiện lỗ hổng mã nguồn được thực hiện theo quy trình: (i) biểu diễn cú pháp và ngữ nghĩa; (ii) trích xuất đặc trưng của mã nguồn; (iii) cân bằng dữ liệu; (iv) phân loại mã nguồn. Đầu ra của mô hình cho biết mã nguồn đó hoặc bình thường, hoặc có lỗ hổng. Mô hình sử dụng bộ dữ liệu SART để huấn luyện mô hình học sâu. Mạng học sâu sử dụng mô hình BERT, mô hình Word2Vec kết hợp LSTM và mô hình Word2Vec với BiLSTM theo ba kịch bản. Kết quả phân loại được đưa qua một hàm softmax để trả về một vector chứa dự đoán xác suất xảy ra của từng loại lỗ hổng. Với tỷ lệ phát hiện lỗ hổng mã nguồn cho kết quả chính xác lên đến 82,63%, tương ứng với tỷ lệ bỏ sót chỉ còn 17,37%. Đây là một kết quả có thể chấp nhận được, chứng minh hiệu quả vượt trội đối với bài toán phát hiện lỗ hổng mã nguồn.

Từ khóa: **học sâu, học tương phản, lỗ hổng, phần mềm.**

Chỉ số phân loại: **1.2, 2.2**

[*]Tác giả liên hệ: Email: bvcong@uneti.edu.vn

DOI: 10.31276/VJST.2024.0019 ***Khoa học tự nhiên*** | *Khoa học máy tính và thông tin, Kỹ thuật điện, kỹ thuật điện tử, kỹ thuật thông tin*

Detecting software vulnerabilities using deep learning

Van Cong Bui[1*], Thao Nguyen Vu[2], Duc Minh Vu[3], Phuong Lan Nguyen[4]

[1]*Faculty of Information Technology, University of Economics - Technology for Industrial, 456 Minh Khai Street, Vinh Tuy Ward, Hai Ba Trung District, Hanoi, Vietnam*

[2]*Nanyang Technological University, 50 Nanyang Ave, Singapore*

[3]*Cau Giay High School, 8/118 Nguyen Khanh Toan Street, Quan Hoa Ward, Cau Giay District, Hanoi, Vietnam*

[4]*Japanese International School, 84A Nguyen Thanh Binh Street, Van Phuc Ward, Ha Dong District, Hanoi, Vietnam*

Received 21 October 2024; revised 5 November 2024; accepted 29 November 2024

Abstract:

Developing successful of software projects is always a top concern for organizations and enterprises. Among these concerns, ensuring software quality is the highest priority throughout the entire development and operation process. This paper addresses the detection of source code vulnerabilities and focuses on analyzing the syntax and semantics of statements within the source code. The source code vulnerability detection model follows a structured process: (i) syntactic and semantic representation; (ii) feature extraction from source code; (iii) data balancing; and (iv) source code classification. The model's output indicates whether the source code is normal or contains vulnerabilities. The model is trained using the SART dataset and incorporates deep learning approaches. Specifically, it employs the BERT model, the Word2Vec model combined with LSTM, and the Word2Vec model with BiLSTM across three scenarios. Classification results are passed through a softmax function to generate a vector containing the probability predictions for each type of vulnerability. The detection model achieves an accuracy rate of up to 82.63% for identifying source code vulnerabilities, with a corresponding omission rate of only 17.37%. This result is considered acceptable and demonstrates the superior effectiveness of the approach in the task of source code vulnerability detection

Keywords: **contrastive learning, deep learning, software, vulnerabilities.**

Classification numbers: **1.2, 2.2**

1. Đặt vấn đề

Ngày càng nhiều lỗ hổng trên phần mềm máy tính là rất lớn, dễ dàng trở thành mục tiêu cho các cuộc tấn công cố ý, trong khi các tin tặc lại có vô số các phương pháp khai thác các lỗ hổng này. Báo cáo cho thấy mức độ thiệt hại và sự gia tăng đột biến về số lượng lỗ hổng phần mềm ngày càng nghiêm trọng. Khi hệ thống đang tồn tại lỗ hổng, nếu người quản lý không

phát hiện ra và cài đặt bản vá trước khi tin tặc biết cách khai thác, chúng có thể bị sử dụng để thực hiện các hành vi nguy hiểm như đánh cắp các dữ liệu quan trọng, gây từ chối dịch vụ… và nguy hiểm nhất là nắm quyền kiểm soát hệ thống và phát tán virus. Để phát hiện lỗ hổng mã nguồn, nghiên cứu chỉ ra có nhiều CVE (Common Vulnerabilities and Exposures) được phát hiện và công bố. Đặc biệt mức độ nghiêm trọng của các CVE là rất lớn. Đồng nghĩa với việc khi các CVE này bị tấn công và khai thác sẽ dẫn đến hậu quả rất nguy hiểm cho các hệ thống của tổ chức.

Vậy nên bài báo đề cập việc phát hiện lỗ hổng trong mã nguồn của phần mềm; gọi là lỗ hổng phần mềm. Có hai phương pháp chính để phát hiện lỗ hổng mã nguồn đó là phương pháp thủ công và phương pháp tự động hóa [1-4]. Đối với hướng tiếp cận thủ công có ưu điểm là hiệu quả cao nhưng nhược điểm là yêu cầu người phân tích phải có trình độ cao đồng thời tốn rất nhiều thời gian. Đối với vấn đề tự động hóa phát hiện lỗ hổng mã nguồn thì đang được nghiên cứu và áp dụng nhiều hiện nay do có nhiều công nghệ tính toán cũng như phương pháp hỗ trợ mạnh mẽ. Có thể nhận thấy rằng các hướng tiếp cận gần đây thường tập trung vào phân tích mã nguồn theo dạng ngữ nghĩa và cú pháp rồi sử dụng các thuật toán học máy hoặc học sâu để tiếp tục phân tích và dự đoán lỗ hổng. Tuy nhiên, chúng tôi cho rằng các hướng tiếp cận hiện nay đang gặp phải một số vấn đề sau [5-11].

- Thiếu đặc trưng và thuộc tính về lỗ hổng bảo mật: Thực tế cho thấy, một số nghiên cứu đã đề xuất được một số phương pháp mang lại hiệu quả tương đối tốt. Tuy nhiên, khi áp dụng các phương pháp này vào các bộ dữ liệu khác thì tỷ lệ phát hiện nhầm rất cao. Điều này thể hiện, phương pháp đề xuất đó chỉ phù hợp với một số bộ dữ liệu cụ thể nên rất khó áp dụng trong các bộ dữ liệu khác. Hay nói cách khác, các thuộc tính và đặc trưng được các nghiên cứu đề xuất chưa thể hiện được đầy đủ và khác biệt về các lỗ hổng phần mềm

- Bộ dữ liệu thực nghiệm thiếu thực tế: Các nghiên cứu trước đây thường lựa chọn các bộ dữ liệu có sẵn để đánh giá sự hiệu quả của mô hình đề xuất. Bên cạnh đó, các bộ dữ liệu này thường có đặc điểm là có sự cân bằng giữa các nhãn lỗ hổng và nhãn bình thường. Điều này dẫn đến các nhà nghiên cứu đã không tính toán đến trường hợp làm sao để phát hiện được lỗ hổng của mã nguồn trên các bộ dữ liệu mất cân bằng. Đây là vấn đề rất quan trọng vì trong thực tế, khi kiểm tra và đánh giá lỗ hổng phần mềm thì có sự chênh lệch rất lớn giữa về số lượng bản ghi của các mã nguồn chứa lỗ hổng bảo mật và mã nguồn không chứa lỗ hổng bảo mật. Chính vì vậy, khi áp dụng các phương pháp này vào thực tế thì tỷ lệ phát hiện sai rất cao về cả khả năng dự đoán chính xác lỗ hổng mã nguồn cũng như mã nguồn bình thường. Cá biệt có trường hợp không thể phát hiện được lỗ hổng phần mềm nào do mô hình bị overfitting.

Để giải quyết các vấn đề trên, các nghiên cứu gần đây thường tập trung áp dụng các kỹ thuật phân tích đồ thị kết hợp với các thuật toán học máy hoặc học sâu [7, 8, 12-15]. Theo đó, các hướng tiếp cận thường phân tích mã nguồn thành các dạng đồ thị như AST (Abstract syntax tree) [16], CFG (Control Flow Graph) [17], PDG (Program Dependence Graph) [18] sau đó

dùng các kỹ thuật học máy hoặc học sâu để phân loại mã nguồn bình thường và mã nguồn chứa lỗ hổng bảo mật. Khi quan sát và phân tích về các hướng tiếp cận này chúng tôi nhận thấy một số vấn đề cần cải thiện như sau:

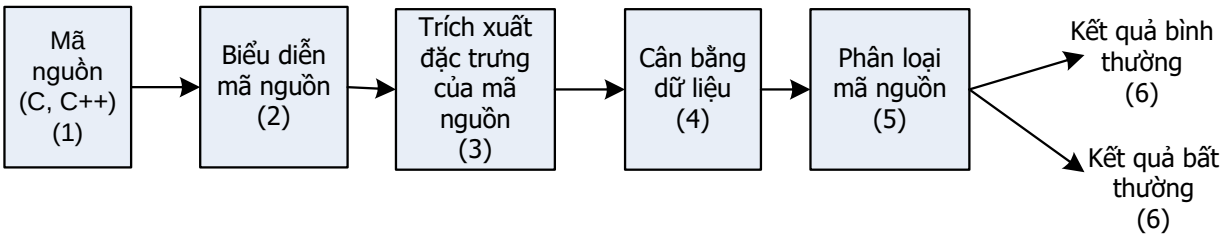
- Vấn đề biểu diễn mã nguồn: Chúng tôi cho rằng việc phân tích và biểu diễn mà nguồn thành một trong các kiểu như AST, CFG, PDG sẽ không thể hiện được đầy đủ mối quan hệ giữa các thành phần trong hàm và các biến của mã nguồn. Nguyên nhân của vấn đề này là do các cách biểu diễn AST, CFG, PDG sẽ chỉ có khả năng biểu diễn được một vài mối quan hệ giữa thành phần của mã nguồn. Do đó, việc chỉ áp dụng một trong các cách biểu diễn này sẽ dẫn đến tình trạng mã nguồn không được biểu diễn một cách đầy đủ và rõ ràng. Từ đó dẫn đến thông tin được trích xuất kém hiệu quả.

- Vấn đề phân loại lỗ hổng mã nguồn trong các bộ dữ liệu mất cân bằng: để giải quyết các vấn đề liên quan đến mất cân bằng dữ liệu, các hướng tiếp cận thường đề xuất 2 giải pháp: i) lựa chọn các bộ dữ liệu có sự cân bằng về mặt định lượng giữa các nhãn. ii) tiến hành phân loại mà không áp dụng kỹ thuật hoặc phương pháp cân bằng dữ liệu. Cả 2 giải pháp này sẽ làm cho sự hiệu quả của mô hình phân loại không đạt kết quả tốt.

- Phương pháp phát hiện lỗ hổng mã nguồn: Theo đó, ngay sau khi kết thúc quá trình phân tích và trích xuất đặc trưng của mã nguồn thì các đặc trưng này thường được đưa vào mô hình phân loại nhằm phát hiện ra các mã nguồn bình thường và bất thường. Chúng tôi nhận thấy với cách tiếp cận truyền thống như vậy mặc dù sẽ mang lại hiệu quả tốt, nhưng cũng cần khắc phục vì các bộ dữ liệu mất cân bằng sẽ dẫn đến tình trạng mô hình phát hiện dễ bị phát hiện nhầm. Từ đó dẫn đến sự hiệu quả của các mô hình phát hiện sẽ không được cao.

Để phát hiện lỗ hổng, bài báo đề xuất mô hình sử dụng phân tích học sâu dựa trên luồng xử lý nhằm bổ sung thêm cho phân tích theo cú pháp còn được trích xuất qua biểu đồ luồng điều khiển CFG, biểu diễn được logic cũng như cả mối quan hệ giữa các đỉnh với nhau thông qua sự kết hợp của 3 kỹ thuật xử lý chính: Word2Vec kết hợp LSTM, Word2Vec kết hợp BiLSTM, phát hiện lỗ hổng phần mềm sử dụng BERT.

Tiếp theo mô hình Word2Vec + LSTM và Word2Vec + BiLSTM được đề xuất nhằm tính toán và trích xuất các đặc trưng của mã nguồn. Trong mô hình này, các thành phần xử lý chính bao gồm tiền xử lý, sử dụng Word2Vec để chuyển chuỗi mã đã được xử lý thành các vector nhúng. Sau đó các vector nhúng được đưa vào mạng LSTM và BiLSTM để tiến hành học và phân loại lỗ hổng. Kết quả phân loại sẽ được đi qua một hàm softmax để trả về một vector chứa dự đoán xác suất xảy ra của từng loại lỗ hổng đã cho (hình 1).



Hình 1. Mô hình tổng quan về phát hiện lỗ hổng mã nguồn.

2. Phần mềm và lỗ hổng phần mềm

2.1. Phần mềm

Phần mềm máy tính là tập hợp các chương trình, dữ liệu và chỉ dẫn mà máy tính sử dụng để thực hiện các nhiệm vụ [19].

Phần mềm có thể chia thành nhiều loại khác nhau, dựa trên chức năng và mục đích sử dụng: Phần mềm hệ thống; Phần mềm ứng dụng; Phần mềm tiện ích; Phần mềm lập trình; Phần mềm nhúng; Phần mềm mã nguồn mở và thương mại.

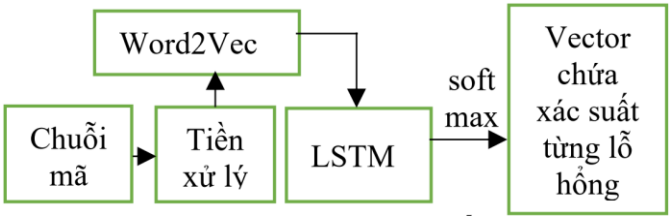
2.2. Lỗ hổng phần mềm

Có các loại lỗ hổng như: lỗi tràn bộ đệm là khi bộ nhớ bị ghi đè nhiều lần trên ngăn xếp; Lưu trữ dữ liệu không an toàn; thiếu chế độ bảo vệ tầng giao vận; rò rỉ dữ liệu; phân quyền và xác thực yếu; mã hóa bị bẻ khóa, không mã hóa; đầu vào không tin cậy; xử lý phiên không đúng, quản lý phiên yếu; thiếu bảo mật nhị phân; thiếu mã hóa dữ liệu; Không xác thực dữ liệu đầu vào; lỗi, lỗi trong phần mềm; sử dụng mật khẩu yếu; lỗ hổng trong định dạng chuẩn đầu ra cho dữ liệu; lỗi thiết kế không bảo mật, sử dụng thuật toán mật mã yếu; lỗ hổng không đồng bộ trong đa luồng; sử dụng dữ liệu hoặc tài nguyên sai cách [19]. Kết quả đúng trên một dữ liệu cụ thể, nhưng khi có dữ liệu khác, bị xung đột; truy cập vào các tài nguyên hoặc chức năng không cho phép.

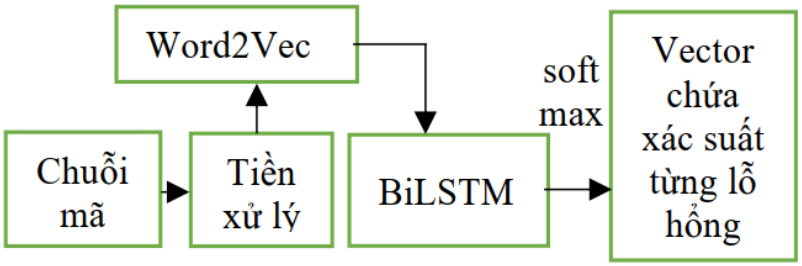
3. Mô hình phát hiện lỗ hổng phần mềm

3.1. Kiến trúc mô hình đề xuất

Trong hình 2 và 3 là mô hình Word2Vec kết hợp với LSTM:



Hình 2. Mô hình Word2Vec kết hợp LSTM tiếp theo là mô hình Word2vec kết hợp với BiLSTM.



Hình 3. Mô hình Word2Vec kết hợp BiLSTM.

Sau cùng là mô hình phát hiện lỗ hổng phần mềm sử dụng BERT (hình 4).

Hình 4. Các thành phần chính mô hình BERT.

Chức năng và nhiệm vụ các khối con trong mô hình đề xuất thể hiện qua các hình như sau:

- Chuỗi mã: là dữ liệu phục vụ quá trình huấn luyện và phát hiện lỗ hổng. Mã nguồn sẽ bao gồm nhiều dạng ngôn ngữ và lỗ hổng khác nhau. Chúng tôi sử dụng bộ dữ liệu tham chiếu về bảo hiểm phần mềm (SARD) do Viện Tiêu chuẩn và Công nghệ Quốc gia (NIST) [20] tạo ra để thử nghiệm các mô hình đề xuất.

- Khối tiền xử lý: Với bộ dữ liệu tạo thành từ các tệp văn bản chứa mã thô gồm các tệp chứa lỗ hổng và các tệp không chứa lỗ hổng, tiến hành gắn nhãn để phân loại. Sau đó tiến hành loại bỏ các thành phần nhiễu khỏi dữ liệu bằng cách xóa những chú thích, đặt lại tên biến, tên hàm để tạo nên một cấu trúc mã nhất quán, không chứa những thông tin ảnh hưởng tới quá trình đào tạo mô hình.

- Khối Word2Vec: là mô hình giúp tạo ra các biểu diễn embedding của từ trong một không gian có số chiều thấp hơn nhiều lần so với số từ trong từ điển. Ý tưởng đó là hai từ (từ con) xuất hiện trong những văn cảnh giống nhau thường có ý nghĩa gần với nhau. Hay có thể đoán được một từ (từ con) nếu biết các từ xung quanh nó trong câu. Như vậy có hai cách khác nhau để xây dựng mô hình Word2vec đó là dự đoán những từ ngữ cảnh nếu biết trước từ đích (Skip-gram) hoặc CBOW (Continuous Bag of Words) dựa vào những từ ngữ cảnh để dự đoán từ đích.

- Khối BERT: Khối BERT sử dụng Transformer là một mô hình attention học mối tương quan giữa các từ (hoặc một phần của từ) trong một văn bản [3].

- Khối LSTM: Khối này được đề xuất là một trong những giải pháp cho vấn đề phát hiện lỗ hổng phần mềm từ mã nguồn. Điểm chính trong kiến trúc LSTM chính là các ô nhớ

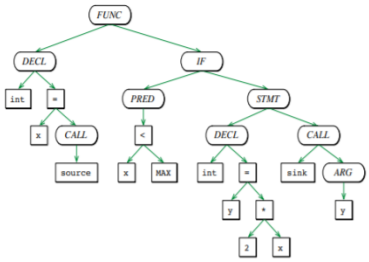
với các cổng cho phép lưu trữ hoặc truy xuất thông tin. LSTM chỉ có thể dự đoán nhãn của từ hiện tại dựa trên thông tin có được từ các từ nằm trước đó.

- Khối BiLSTM: Như đã nêu, điểm hạn chế của LSTM chính là chỉ dự đoán nhãn của từ hiện tại dựa trên thông tin có được từ các từ nằm trước đó. Trong khi, việc nhận dạng chính xác cấu trúc và ngữ cảnh của một đoạn mã nguồn không chỉ cần thông tin ở phía trước của từ đang xét mà còn cả các thông tin phía sau. Điều này LSTM lại không thể nhận dạng được. Chính vì thế BiLSTM đã được tạo ra để khắc phục điểm yếu trên. Một kiến trúc BiLSTM thường chứa 2 mạng LSTM đơn được sử dụng đồng thời và độc lập để mô hình hóa chuỗi đầu vào theo 2 hướng: từ trái sang phải và từ phải sang trái.

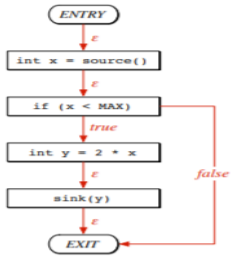
3.2. Biểu diễn mã nguồn phần mềm

Mô hình trong bài báo sử dụng biểu đồ luồng điều khiển CFG [17] là dạng biểu đồ trong đó mỗi nút đại diện cho một lệnh hay nói cách khác CFG là một đồ thị biểu diễn đồ họa của luồng điều khiển hoặc tính toán trong quá trình thực thi các chương trình hoặc ứng dụng của một chương trình. Các nút trong CFG là các khối cơ bản, nhóm các lệnh có thể thực thi tuần tự mà không có bất kỳ chuyển hướng điều khiển nào và các cạnh đại diện cho các luồng điều khiển giữa các khối (hình 5).

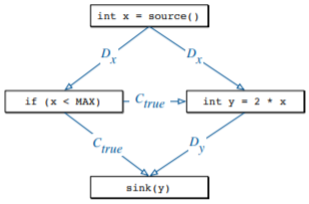
(A)



(B)



(C)



Hình 5. Biểu diễn mã nguồn dưới một số dạng AST (A), CFG (B) và PDG (C).

3.3. Trích xuất đặc trưng của mã nguồn phần mềm

Các nghiên cứu cho thấy đặc trưng của đỉnh và cạnh trong CPG (Code Property Graph) [4, 21] là một biểu diễn quan trọng của mã nguồn. CPG kết hợp các phần tử từ luồng điều khiển, biểu đồ luồng dữ liệu, cùng với AST và PDG cho phép biểu diễn tổng hợp ngữ nghĩa

của mã nguồn. Về mặt hình thức, CPG được ký hiệu là G = (V, E), trong đó V là tập hợp các đỉnh (nút) và E là tập hợp các cạnh trong đồ thị. Mỗi đỉnh chứa thông tin về loại đỉnh.

3.4. Cân bằng dữ liệu

Sau khi thu được hồ sơ đặc trưng của mã nguồn, chúng tôi dùng nó để phân loại nhằm phát hiện các mã nguồn bình thường và mã nguồn chứa lỗ hổng. Từ hồ sơ thu được, chúng tôi nhận thấy có sự chênh lệch rất lớn giữa mã nguồn chứa lỗ hổng và mã nguồn bình thường. Nếu vấn đề này không được giải quyết, sẽ dẫn đến một sai lệch không mong muốn trong mô hình làm hạn chế hiệu suất dự đoán của nó. Hơn nữa, các vector đặc trưng được trích xuất của các mẫu có lỗ hổng và các mẫu không có lỗ hổng thể hiện sự chồng chéo đáng kể trong không gian tính năng. Điều này gây khó khăn cho việc phân định các mẫu có lỗ hổng và các mẫu không có lỗ hổng. Để giải quyết tình trạng mất cân bằng dữ liệu trong quá trình huấn luyện và giúp tăng tính đa dạng của dữ liệu chúng tôi sử dụng thuật toán SMOTE (Synthetic Minority Oversampling Technique) [22].

3.5. Phân loại lỗ hổng phần mềm

Dựa trên các đặc trưng của mã nguồn được trích xuất, kết hợp với các vector đặc trưng được sinh ra từ thuật toán SMOTE bài báo sẽ tiến hành phân loại các vector đặc trưng này để kết luận về lỗ hổng của mã nguồn. Để thực hiện mục tiêu này, bài báo sử dụng 2 lớp đơn giản là Fully Connected và Softmax [23]. Trong đó Fully Connected là một cách đơn giản hóa để học các tổ hợp phi tuyến tính của các đối tượng. Lớp Fully Connected đang học một hàm có thể phi tuyến tính trong không gian. Đầu ra được làm phẳng này được cung cấp cho một mạng nơ ron truyền thẳng và mạng nơ ron truyền ngược. Qua một loạt các epoch, mô hình có thể phân biệt các đặc điểm nổi bật trong dữ liệu và phân loại chúng bằng cách sử dụng kỹ thuật Phân loại Softmax [24].

4. Thử nghiệm

4.1. Dữ liệu thử nghiệm

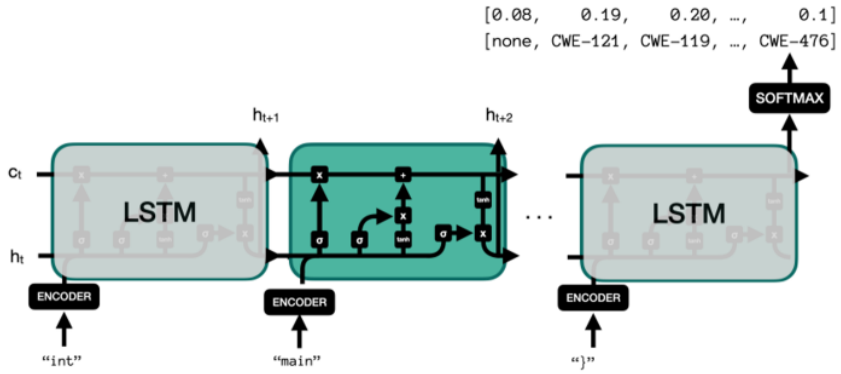
Bài báo này sử dụng bộ dữ liệu tham chiếu về bảo hiểm phần mềm (SARD) [20] do Viện Tiêu chuẩn và Công nghệ Quốc gia (NIST) tạo ra. SARD đặc biệt hấp dẫn vì nó không chỉ chứa các lỗ hổng bảo mật mà còn chứa các bản sao đã không tồn tại lỗ hổng của chúng, cho phép tìm hiểu chính xác sự khác biệt giữa lỗ hổng bảo mật và các giải pháp thay thế an toàn của nó. Bộ dữ liệu được xử lý trước khi đào tạo bằng cách loại bỏ tất cả các phần không mong muốn có thể khiến các mô hình bị quá tải và mất cân bằng. Và cuối cùng bộ dữ liệu còn lại hơn 100,000 tệp, với một nửa chứa một trong 118 lỗ hổng bảo mật và nửa còn lại chứa các lựa chọn thay thế không dễ bị tấn công. Tiến hành chia bộ dữ liệu thành các thành phần khác nhau, sau đó tiến hành thực nghiệm và đánh giá độ chính xác của các mô hình đề xuất. Toàn bộ quá trình chia tập dữ liệu thực nghiệm cho các kịch bản sẽ được chọn một cách ngẫu nhiên

trong đó 75% của tập dữ liệu sẽ được dùng trong quá trình huấn luyện, 25% còn lại sẽ được dùng trong quá trình kiểm tra.

4.2. Kịch bản đánh giá

Để đánh giá được kết quả thử nghiệm tác giả tiến hành ba kịch bản thực nghiệm như sau:

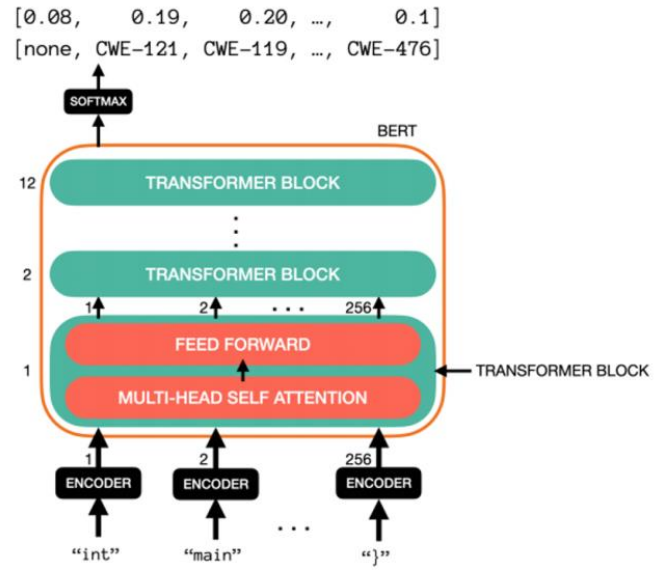
Kịch bản 1: Phát hiện lỗ hổng phần mềm sử dụng Word2vec kết hợp với LSTM: Thử nghiệm trên mô hình LSTM với số lượng các nút ẩn khác nhau (hình 6).



Hình 6. Mô hình phát hiện lỗ hổng sử dụng LSTM.

Kịch bản 2: Phát hiện lỗ hổng phần mềm sử dụng Word2vec kết hợp với BiLSTM: số lượng các nút ẩn khác nhau.

Kịch bản 3: Phát hiện lỗ hổng phần mềm sử dụng BERT: số lượng tầng khác nhau (hình 7).



Hình 7. Mô hình phát hiện lỗ hổng sử dụng BERT.

4.3. Tiêu chí đánh giá mô hình

Accuracy: Tỷ lệ giữa số lượng mẫu được phân loại đúng và tổng số mẫu. Độ chính xác được tính theo công thức sau:

$$accuracy = \frac{TP+TN}{TP+TN+FP+FN} \times 100\% \quad (1)$$

trong đó: TP - True positive: số lượng mẫu có lỗ hổng được phân loại chính xác; FN - False negative: số lượng mẫu có lỗ hổng được phân loại là không có lỗ hổng; TN - True negative: số lượng mẫu không có lỗ hổng được phân loại chính xác; FP - False positive: số lượng mẫu không có lỗ hổng được phân loại thành có lỗ hổng.

Precision: là tỷ lệ của true positive points trên tổng số điểm được phân loại là positive (TP + FP). Độ chính xác cao có nghĩa là độ chính xác cao của việc xác định points.

$$precision = \frac{TP}{TP+FP} \times 100\% \quad (2)$$

Recall: là tỷ lệ của các true positive points trên tổng số các real positive points (TP + FN). Recall cao có nghĩa là TP cao và tỷ lệ bỏ sót positive points là thấp.

$$recall = \frac{TP}{TP+FN} \times 100\% \quad (3)$$

F1-score: là trung bình hài hòa của độ chính xác và thu hồi. F1 càng cao, bộ phân loại càng tốt.

$$F1 = \frac{2 \times precision \times recall}{precision + recall} \quad (4)$$

4.4. Kết quả thực nghiệm

Kết quả thực nghiệm theo kịch bản 1: trong kịch bản thực nghiệm 1, phát hiện lỗ hổng phần mềm sử dụng kiến trúc mô hình Word2Vec kết hợp với LSTM trên bộ dữ liệu SARD cho kết quả theo bảng 1, tác giả thấy được độ chính xác của quá trình phân loại tiến trình cho kết quả khác nhau khi thay đổi số lượng các nút ẩn. Với số lượng các nút ẩn là 400 thì thuật toán LSTM mang lại hiệu quả cao nhất trên tất cả các độ đo. Mặc dù số lượng các nút ẩn thay đổi từ 200 đến 1000 nhưng độ chính xác của thuật toán lại không có nhiều thay đổi. Kết quả cao nhất và thấp nhất chỉ chênh lệch nhau: accuracy tăng 1,64%, precision tăng 1,82%, recall tăng 1,94% và F1 tăng 1,89%. Như vậy, mặc dù số lượng các nút ẩn tăng hay giảm cũng không làm cho độ chính xác trong quá trình phân loại lại thay đổi một cách đáng kể. Điều này cho thấy thuật toán LSTM cung cấp khả năng phân loại tốt và ổn định khi thay đổi số lượng các nút ẩn.

Bảng 1. Kết quả thực nghiệm phát hiện lỗ hổng phần mềm sử dụng kiến trúc mô hình Word2Vec kết hợp với LSTM.

Số các nút ẩn	Chính xác	Độ phủ	f1	Độ chính xác
1000	71,15	67,89	69,48	71,53
400	**72,97**	**69,83**	**71,37**	**73,17**
200	71,70	68,47	70,05	72,02

Kết quả thực nghiệm kịch bản 2: Kịch bản 2 sử dụng kiến trúc mô hình Word2Vec kết hợp với BiLSTM trên bộ dữ liệu SARD cho kết quả từ bảng 1 và 2 thấy được khi thay đổi số lượng các các nút ẩn thì LSTM và BiLSTM cho ra các kết quả tương tự nhau. Dựa trên kết quả kịch bản 1 và 2 đã chứng minh được mô hình NLP có ý nghĩa đối với bài toán phân loại lỗ hổng. Điều này được thể hiện trên độ chính xác tổng quát tối thiểu đều khoảng trên 71% và đối với giá trị recall tối thiểu là 67%. Nói chung, kết quả phân loại đối với kịch bản chỉ dừng ở mức không quá tệ và tạm chấp nhận được còn kịch bản 2 thì tương đối ổn. Kịch bản 2 sử dụng BiLSTM đã mang lại kết quả tốt hơn LSTM tới 6% cho thấy tầm quan trọng của việc lưu giữ thông tin theo ngữ cảnh trên từng phân đoạn. Bộ nhớ hai chiều trong BiLSTM có thể cải thiện đáng kể hiệu suất chính xác của việc phát hiện lỗ hổng phần mềm theo cách NLP vì thông tin theo ngữ cảnh được xem xét một cách toàn diện.

Bảng 2. Kết quả thực nghiệm phát hiện lỗ hổng phần mềm sử dụng kiến trúc mô hình Word2Vec kết hợp với BiLSTM.

Số các nút ẩn	Chính xác	Độ phủ	f1	Độ chính xác
1000	78,57	75,86	77,19	78,67
400	**80,13**	**77,56**	**78,83**	**79,83**
200	79,62	77,43	78,51	79,17

Kết quả thực nghiệm kịch bản 3: Bảng 3 cho thấy số lượng tầng tăng thì độ chính xác của mô hình cũng tăng theo. Tuy nhiên, việc tăng số tầng làm tăng hiệu suất mô hình lên khoảng 2% nhưng đồng thời cũng tăng thời gian xử lý của mô hình lên. Trong thực tế, để cân bằng giữa hiệu suất và thời gian thì cần phải cân nhắc lựa chọn được các tham số phù hợp để cho ra được kết quả như mong đợi. Đối với kịch bản 3 sử dụng mô hình BERT với cơ chế hai chiều và cơ chế chú ý (attention) cho kết quả phát hiện lỗ hổng tốt hơn hẳn. Điều này được thể hiện trên kết quả recall cao nhất trong 3 kịch bản là 82% tương ứng với tỷ lệ bỏ sót là 18%. Về

độ chính xác, mô hình BERT và các biến thể của nó đều hoạt động tốt hơn các mô hình học sâu NLP truyền thống đã được thử nghiệm. Điều này là do cơ chế chú ý trong các mô hình BERT có thể trích xuất thông tin nhận biết ngữ cảnh theo cách tốt hơn nhiều so với LSTM hoặc RNN. Kết quả này chứng minh được mô hình đề xuất có giá trị thực tiễn đối với bài toán phát hiện lỗ hổng mã nguồn.

Bảng 3. Kết quả thực nghiệm phát hiện lỗ hổng phần mềm sử dụng mô hình BERT.

Số các tầng	Chính xác	Độ phủ	f1	Độ chính xác
4	82,89	80,97	81,92	84,08
8	83,58	81,72	82,64	84,64
12	84,41	82,63	83,51	85,33

Chúng tôi đã sử dụng bộ dữ liệu FFmpeg+Qume để so sánh việc phát hiện lỗ hổng mã nguồn theo mô hình BERT, như kịch bản 3 nêu trên với các mô hình Russell [13], VulDeePecker [24], SySeVR [25]. Kết quả thể hiện trong bảng 4.

Bảng 4. So sánh mô hình đề xuất với một số mô hình của các tác giả khác.

Bộ dữ liệu	Tiếp cận	Acc (độ chính xác)	Pre (chính xác)	Rec (độ phủ)	F1 (chỉ số F1)
FFmpeg+Qume	Kịch bản 3 theo mô hình BERT	62,78	57,15	72,98	64,1
	Russell [13]	58,13	54,04	39,50	45,62
	VulDeePecker [25]	53,58	47,36	28,70	35,20
	SySeVR [25]	52,52	48,34	65,96	56,03

Các kết quả thực nghiệm trên bảng 4 cho thấy tiếp cận của chúng tôi được đề xuất trong bài báo không chỉ mang lại hiệu quả tốt trong nhiệm vụ phân loại lỗ hổng mã nguồn mà còn tối ưu hơn so với các mô hình, hướng tiếp cận khác. Cụ thể đối với bộ dữ liệu FFmpeg+Qume, dựa trên kết quả thực nghiệm thì thấy được rằng với độ đo Accuracy thì tiếp cận của chúng tôi hiệu quả hơn tất cả các nghiên cứu khác từ 0,2 đến 10%. Tương tự như vậy đối với độ đo Precision, tiếp cận này cũng tốt hơn từ 0,3 đến 9% so với các hướng tiếp cận khác. Còn đối với độ đo Recall thì tiếp cận của chúng tôi đề xuất cao hơn tất cả các tiếp cận còn lại từ 10 đến hơn 31%. Tương tự như độ đo Recall, độ đo F1_score theo tiếp cận của chúng tôi cũng cao hơn các nghiên cứu khác từ 7 đến 30%.

5. Kết luận

Nghiên cứu đã đề xuất sử dụng mô hình BERT với cơ chế hai chiều và chú ý cho kết quả phát hiện lỗ hổng với độ chính xác theo tỷ lệ phát hiện lỗ hổng mã nguồn cho kết quả chính xác lên đến 82,63% tương ứng với tỷ lệ bỏ sót chỉ còn 17,37%. Các kết quả thực nghiệm trong mục 4 của bài báo đã chứng minh được hiệu quả vượt trội đối với bài toán phát hiện lỗ hổng mã nguồn.

Thực nghiệm so sánh và đánh giá tính hiệu quả trong việc phát hiện lỗ hổng của mã nguồn trong bảng 3 cho thấy, số lượng tầng tăng từ 4 đến 8 thì F1-score tăng từ 81,92% đến 82,64%, tức là độ chính xác của mô hình cũng tăng theo. Tuy nhiên, việc tăng số tầng làm tăng hiệu suất mô hình lên khoảng 2% đồng thời cũng làm tăng thời gian xử lý của mô hình lên. Trong thực tế, để cân bằng giữa hiệu suất và thời gian cần lựa chọn các tham số phù hợp để cho ra được kết quả như mong đợi. Các kết quả thử nghiệm để lựa chọn các tham số phù hợp cũng như mô hình đã được trình bày.

TÀI LIỆU THAM KHẢO

[1] B. Chernis, R. Verma (2018), "Machine learning methods for software vulnerability detection," *Proceedings of The Fourth ACM International Workshop on Security and Privacy Analytics*, pp.31-39, DOI: 10.1145/3180445.3180453.

[2] B. Liu, W. Guan, C. Yang, et al. (2023), "Transformer and graph convolutional network for text classification", *International Journal of Computational Intelligence Systems*, **16**, DOI: 10.1007/s44196-023-00337-z.

[3] J. Devlin, M.W. Chang, K. Lee, et al. (2019), "Bert: Pre-training of deep bidirectional transformers for language understanding", *The 2019 Conference of The North American Chapter of The Association for Computational Linguistics: Human Language Technologies*, **1**, pp.4171-4186, DOI: 10.18653/v1/N19-1423.

[4] D.X. Cho, H.M. Dao, M.T. Cong, et al. (2023), "A novel approach for software vulnerability detection based on intelligent cognitive computing", *The Journal of Supercomputing*, **79**, pp.17042-17078, DOI: 10.1007/s11227-023-05282-4.

[5] G. Lin, S. Wen, Q.L. Han, et al. (2020), "Software vulnerability detection using deep neural networks: A survey", *Proceedings of The IEEE*, **108(10)**, pp.1825-1848, DOI: 10.1109/JPROC.2020.2993293.

[6] P. Zeng, G. Lin, L. Pan, et al. (2020), "Software vulnerability analysis and discovery using deep learning techniques: A survey", *IEEE Access*, **8**, pp.197158-197172, DOI: 10.1109/ACCESS.2020.3034766.

[7] H. Wang, G. Ye, Z. Tang, et al. (2021), "Combining graph-based learning with automated data collection for code vulnerability detection", *IEEE Transactions on Information Forensics and Security*, **16**, pp.1943-1958, DOI: 10.1109/TIFS.2020.3044773.

[8] X. Li, L. Wang, Y. Xin, et al. (2021), "Automated software vulnerability detection based on hybrid neural network", *Applied Sciences*, **11(7)**, DOI: 10.3390/app11073201.

[9] H. Wei, M. Li (2017), "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code", *Proceedings of The TwentySixth International Joint Conference on Artificial Intelligence*, pp.3034-3040.

[10] G. Siewruk, W. Mazurczyk (2021), "Context-aware software vulnerability classification using machine learning", *IEEE Access*, **9**, pp.88852-88867, DOI: 10.1109/ACCESS.2021.3075385.

[11] X. Li, L. Wang, Y. Xin, et al. (2020), "Automated vulnerability detection in source code using minimum intermediate representation learning", *Appl. Sci.*, **10**, DOI: 10.3390/app10051692.

[12] W. Zheng, J. Gao, X. Wu, et al. (2020), "The impact factors on the performance of machine learning-based vulnerability detection: A comparative study", *The Journal of Systems & Software*, **168(7)**, DOI: 10.1016/j.jss.2020.110659.

[13] R.L. Russell, L. Kim, L.H. Hamilton, et al. (2018), "Automated vulnerability detection in source code using deep representation learning", *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp.757-762, DOI: 10.1109/ICMLA.2018.00120.

[14] P. Haridas, G. Chennupati, N. Santhi, et al. (2020), "Code characterization with graph convolutions and capsule networks", *IEEE Access*, **8**, pp.136307-136315, DOI: 10.1109/ACCESS.2020.3011909.

[15] Z. Li, D. Zou, J. Tang, et al. (2019), "A comparative study of deep learning-based vulnerability detection system", *IEEE Access*, **7**, pp.103184-103197, DOI: 10.1109/ACCESS.2019.2930578.

[16] F. Yamaguchi, M. Lottmann, K. Rieck (2012), "Generalized vulnerability extrapolation using abstract syntax trees", *Annual Computer Security Applications Conference*, **28**, pp.358-368.

[17] H. Gascon, F. Yamaguchi, D. Arp, et al. (2013), "Structural detection of android malware using embedded call graphs", *ACM Workshop on Artificial Intelligence and Security*, pp.45-54.

[18] K. Ferrante, J. OttensteinJoe, D. Warren (1989), "The program dependence graph and its use in optimization", *ACM Transactions on Programming Languages and Systems*, **9(3)**, pp.319-349.

[19] D.X. Cho (2023), "A new approach to software vulnerability detection based on CPG analysis", *Cogent Engineering*, **10(1)**, DOI: 10.1080/23311916.2023.2221962.

[20] R. Krishna, Y. Ding, B. Ray (2022), "Deep learning based vulnerability detection: Are we there yet?", *IEEE Transactions on Software Engineering*, **48(9)**, pp.3280-3296, DOI: 10.1109/TSE.2021.3087402.

[21] F. Yamaguchi, N. Golde, D. Arp, et al. (2014), "Modeling and discovering vulnerabilities with code property graphs", *IEEE Symposium on Security and Privacy*, pp.590-604, DOI: 10.1109/SP.2014.44.

[22] N.V. Chawla, K.W. Bowyer, L.O. Hall, et al. (2002), "SMOTE: Synthetic minority over-sampling technique", *Journal of Artificial Intelligence Research*, **16**, pp.321-357.

[23] E. Hoffer, N. Ailon (2015), "Deep metric learning using triplet network", *International Workshop on Similarity-Based Pattern Recognition*, pp.84-92.

[24] Z. Li, D. Zou, S. Xu, et al. (2018), "VulDeePecker: A deep learning-based system for vulnerability detection", https://arxiv.org/abs/1801.01681, accessed 15 October 2024.

[25] Z. Li, D. Zou, S. Xu, et al. (2018), "SySeVR: A framework for using deep learning to detect software vulnerabilities", *IEEE Transactions on Dependable and Secure Computing*, DOI: 10.1109/TDSC.2021.3051525.