Học máy trong việc phát hiện lỗ hổng mã nguồn

Machine Learning for Code Vulnerability Detection

NHÓM G17 - Hồ Ngọc Thiện, Dương Phú Cường, Chu Nguyễn Hoàng Phương
GVHD: Phan Thế Duy

ABSTRACT

Ngày nay, các lỗ hổng phần mềm có thể gây ra hậu quả nghiêm trọng đối với một hệ thống máy tính. Chúng có thể dẫn đến sự sập hệ thống, rò rỉ thông tin riêng tư hoặc thậm chí gây thiệt hại vật lý. Việc xác định chính xác các lỗ hổng trong các mã nguồn lớn một cách kịp thời là điều kiện tiên quyết cần thiết để có thể đưa ra bàn vá kịp thời. Tuy nhiên, các phương pháp hiện tại để xác định lỗ hổng, cho dù là phương pháp cổ điển hay dựa trên học sâu (Deep learning), đều tồn tại một số các hạn chế nghiêm trọng, làm cho chúng không thể đáp ứng các yêu cầu ngày càng hiện đại được đặt ra bởi ngành công nghiệp phần mềm.

Trong bài báo cáo này, chúng em thực hiện nghiên cứu so sánh 2 bài báo và chọn ra 1 bài để tiến hành nghiên cứu chính. Hai bài báo chúng em chọn ra là:

+ VulChecker: Graph-basedVulnerability Localization in Source Code

+ DeepVulSeeker: A Novel Vulnerability Identification Framework via Code Graph Structure and Pre-training Mechanism

Dựa vào các chức n*u*ang và kỹ thuật được đề xuất trong cả hai bài báo, chúng em quyết định sử dụng DeepVulSeeker để tiếp tục nghiên cứu tiếp tục.

1 GIỚI THIỆU

Các lỗ hổng phần mềm từ lâu đã luôn là một mối đe dọa lớn trong ngành công nghiệp phần mềm. Chúng vẫn là nguyên nhân chính gây ra sự tổn hại của một hệ thống máy tính. Theo báo cáo của cơ quan thương mại Hoa Kỳ, viện tiêu chuẩn và công nghệ quốc gia (NIST) và cơ sở dữ liệu lỗ hổng quốc gia (NVD), số lượng lỗ hổng trong n*u*am 2021 đã đạt tới 18.378, một con số kỷ lục chưa từng có. Nếu không thể xác định và khắc phục kịp thời, chúng có thể gây ra những mối đe dọa nghiêm trọng tiềm ẩn đến cuộc sống hàng ngày của chúng ta, chẳng hạn như trộm cắp tài chính từ mua sắm trực tuyến, rò rỉ thông tin cá nhân hoặc thậm chí là tổn hại vật chất. Do đó, việc nhanh chóng và chính xác phát hiện các lỗ hổng phần mềm và khắc phục chúng là chìa khóa để đảm bảo an ninh cho một hệ thống máy tính.

Trong học thuật, nghiên cứu về việc xác định lỗ hổng đã được nghiên cứu liên tục trong nhiều n*u*am. Ở giai đoạn đầu, các phương pháp xác định lỗ hổng có thể được chia thành ba loại: phân tích tĩnh, phân tích động và các phương pháp học máy sơ khai. Phân tích tĩnh đề cập đến việc kiểm tra một chương trình phần mềm, hoặc bằng công việc thủ công hoặc với sự trợ giúp của một số dạng thuật toán tự động, mà không thực sự thực thi nó. Nhược điểm chính của phân tích tĩnh là nó hoặc yêu cầu nhiều công sức thủ công hoặc tiêu tốn nhiều thời gian do phân tích luồng hoạt động. Khác với phân tích tĩnh, phân tích động yêu cầu thực sự chạy chương trình để xác định lỗi. Do đó, phân tích động không phải là một phương pháp có thể mở rộng được do bản chất của nó. Các phương pháp học máy sơ khai đề cập đến những phương pháp sử dụng các kỹ thuật học máy cơ bản để xác định lỗ hổng với các đặc trưng đã được xác định trước. Điểm yếu của các phương pháp này là chúng có hiệu suất kém trên các bộ dữ liệu chương trình phức tạp nơi các đặc trưng có thể thay đổi đáng kể.

Trong các nghiên cứu gần đây, các nhà nghiên cứu có xu hướng giải quyết những thiếu sót nêu trên bằng cách áp dụng học sâu (Deep Learning), một phương pháp hoàn toàn tự động và có khả n*u*ang thích nghi với các tình huống phức tạp. Một số nghiên cứu sơ bộ đã cho thấy học sâu là một phương pháp khả thi để xác định nhiều lỗ hổng hơn so với các phương pháp nêu trên. Tuy nhiên, sau khi xem xét kỹ lưỡng các công trình dựa trên học sâu hiện có, chúng tôi nhận thấy rằng hiệu suất của chúng vẫn chưa đạt yêu cầu do một số lý do. Một số mô hình, chẳng hạn như các mô hình dựa trên mạng nơ-ron đồ thị (GNN) thuần túy, chỉ có thể nắm bắt được các đặc trưng cấu trúc của một chương trình. Nhược điểm của các mô hình này là chúng bỏ qua các đặc trưng ngữ nghĩa phong phú (chẳng hạn như tên hàm và các lời gọi hệ thống) ẩn trong mã nguồn. Một số mô hình chỉ trích xuất các đặc trưng ngữ nghĩa trong khi bỏ qua thông tin cấu trúc có trong mã nguồn. Một số công trình đã cố gắng xem xét cả đặc trưng cấu trúc và ngữ nghĩa, nhưng mô hình của họ vẫn chưa được đào tạo đầy đủ cho nhiệm vụ vì họ không nhận ra tầm quan trọng của các kỹ thuật tiền huấn luyện.

Để khắc phục những nhược điểm nêu trên, trong bài báo này, chúng em đề xuất nghiên cứu framework DeepVulSeeker, một mô hình dựa trên thông tin cấu trúc không đồng nhất với MetaPaths, các mô hình ngữ nghĩa được tiền huấn luyện, và mạng mã hóa tự chú ý biểu diễn đồ thị (Graph Representation Self-Attention - GRSA). DeepVulSeeker đầu tiên nắm bắt mối quan hệ thông tin ngữ nghĩa giữa từng nút chương trình và các nút lân cận của một đoạn mã với sự trợ giúp của mô hình tiền huấn luyện dựa trên Transformer hai chiều nhiều lớp có tên là UniXcoder model. UniXcoder có thể mã hóa hiệu quả các ngôn ngữ tự nhiên thành các ngôn ngữ lập trình,

ACM ISBN . . . $

cung cấp thông tin ngữ nghĩa liên quan và chính xác hơn cho việc đào tạo mô hình của chúng tôi. DeepVulSeeker sau đó sử dụng thông tin ngữ nghĩa để có được các biểu diễn cấu trúc của mã thông qua Đồ thị luồng dữ liệu (Data Flow Graph - DFG), Đồ thị luồng điều khiển (Control Flow Graph - CFG), và Cây cú pháp trừu tượng (Abstract Syntax Tree - AST). Cuối cùng, DeepVulSeeker kết hợp thông tin ngữ nghĩa đã thu được và thông tin cấu trúc, và đưa chúng vào một mạng nơ-ron tích chập và một mạng chuyển tiếp để dự đoán xem một đoạn mã cho trước có bị lỗ hổng hay không. Tóm lại, DeepVulSeeker không chỉ có thể học được thông tin cấu trúc của một chương trình, mà còn có thể hiểu được các ngụ ý ngữ nghĩa của nó.

Tập dữ liệu được tác giả sử dụng để đánh giá DeepVulSeeker là các bộ dữ liệu chuẩn Common Weakness Enumeration (CWE) và hai bộ dữ liệu chương trình rất phức tạp, là QEMU và FFMPEG, với tổng số 18.519 đoạn mã, trong đó có 8.125 đoạn có lỗ hổng và phần còn lại không có lỗi. Kết quả cho thấy DeepVulSeeker đạt được hiệu suất tốt trên các bộ dữ liệu CWE, với độ chính xác cao tới 0,99. Đối với các bộ dữ liệu QEMU và FFMPEG, DeepVulSeeker cũng đạt kết quả chấp nhận được với độ chính xác cao tới 0,64, cao hơn tất cả các phương pháp nổi tiếng hiện có. Nhóm tác giả cũng thực hiện các nghiên cứu trường hợp, chứng minh ba trường hợp điển hình không được xác định bởi các phương pháp cơ sở khác nhưng được DeepVulSeeker xác định thành công. Đóng góp của nhóm tác giả bao gồm ba điểm chính:

1. Đề xuất một khung nhận diện lỗ hổng mới, có tên là DeepVulSeeker, có khả n*u*ang học cả thông tin ngữ nghĩa và thông tin cấu trúc của một chương trình nhất định để xác định xem nó có lỗ hổng hay không. DeepVulSeeker có thể nắm bắt nhiều đặc trưng phát triển hơn so với các phương pháp hiện có theo cách tự động.
2. Thực hiện các nghiên cứu trường hợp mà hầu hết các nghiên cứu khác không có. Kết quả cho thấy DeepVulSeeker có khả n*u*ang tìm ra các lỗ hổng phức tạp hơn và hiểu rõ hơn về ý nghĩa của chúng.
3. Thực hiện nhiều thí nghiệm trên các bộ dữ liệu lớn và so sánh hiệu suất với các phương pháp nổi tiếng gần đây nhất. Kết quả cho thấy DeepVulSeeker vượt trội hơn hầu hết các phương pháp này trên các bộ dữ liệu CWE truyền thống và vượt trội hơn tất cả các phương pháp trên các bộ dữ liệu chương trình rất phức tạp, tức là QEMU và FFMPEG.

2 THIẾT KẾ CỦA DEEPVULSEEKER

Ở phần này, chúng ta sẽ đi vào chi tiết thiết kế của DeepVulSeeker. Cấu trúc tổng thể của DeepVulSeeker được biểu thị ở Hình 1 . Luồng hoạt động của DeepVulSeeker bao gồm 4 bước. Đầu tiên, DeepVulSeeker lấy những đoạn mã nguồn cần được phân tích để làm đầu vào và nhận các thông tin cấu trúc từ các đồ thị DFG, CFG và AST, biểu thị bằng các ma trận liền kề. Sau đó, DeepVulSeeker mã hóa các mã nguồn thô thành các token phù hợp với ngữ cảnh của đoạn code bằng cách sử dụng Programming Languages Sequencing (PLS), được xây dựng từ kỹ thuật tiền xử lý. Tiếp đó, DeepVulSeeker sử dụng kỹ thuật MetaPath cho các thông tin cấu trúc và đưa nó vào Graph Represent Self-Atttention Encode (GRSA) cùng với đầu ra của PLS. Cuối cùng, DeepVulSeeker chuyển tiếp những đầu ra từ bước thứ ba vào mạng tích chập, nơi được sử dụng để dự đoán xem liệu đoạn mã nguồn có lỗ hổng hay không.

2.1 Thông tin cấu trúc

Thông tin cấu trúc là một trong những phần quan trọng nhất của DeepVulSeeker. Dưới đây là phương pháp tác giả đề xuất cho việc trích xuất thông tin cấu trúc của mã nguồn.

2.1.1 Abstract Syntax Tree (AST): . AST là một biểu diễn dạng cây của cấu trúc cú pháp trừu tượng của một đoạn mã nguồn. Mỗi đỉnh trong cây đại diện cho một sự xuất hiện cú pháp của ngữ cảnh mã nguồn, và mỗi cạnh phản ánh mối quan hệ giữa một cặp đỉnh.

2.1.2 Control Flow Graph (CFG): . CFG mô tả tất cả các đường dẫn điều khiển có thể có trong một chương trình có thể được duyệt qua. Các đường dẫn được duyệt qua được chỉ định bởi các câu lệnh rẽ nhánh, ví dụ như câu lệnh if, while và do. Trong CFG, mỗi đỉnh đại diện cho một sự xuất hiện cú pháp, và mỗi cạnh biểu thị một nhánh có thể có.

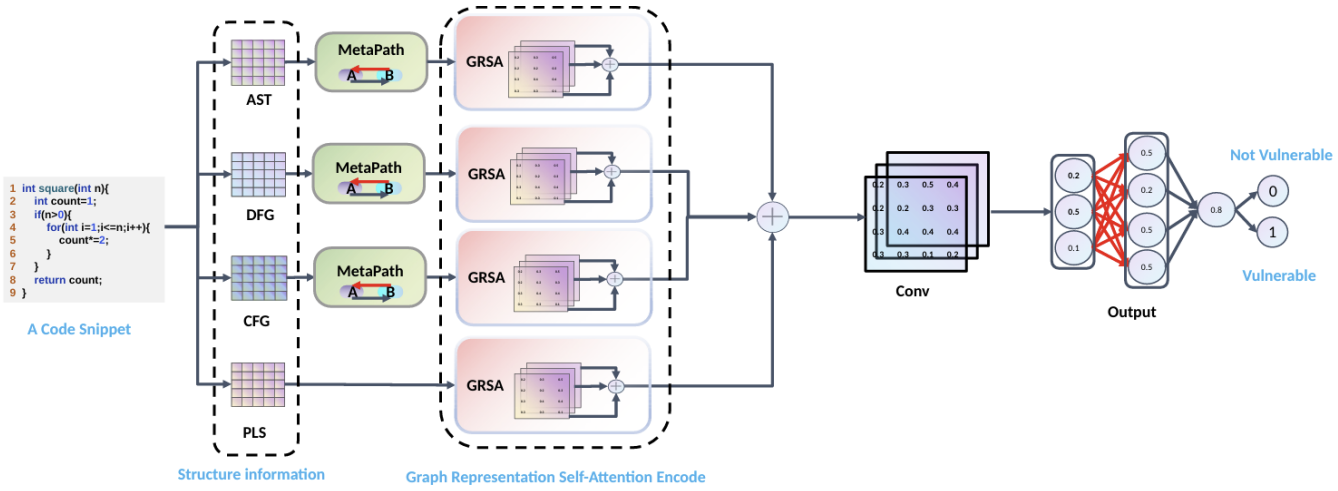
2.1.3 Data Flow Graph (DFG): . DFG theo dõi việc truy cập các biến trong một chương trình. Việc truy cập một biến bao gồm cả việc gán giá trị và sửa đổi giá trị của nó. Trong DFG, mỗi đỉnh là một sự xuất hiện cú pháp, và mỗi cạnh giữa một cặp đỉnh chỉ ra rằng hai đỉnh này có liên quan đến việc gán và sửa đổi giá trị của một biến.

2.2 Meta Path

Một MetaPath là một đường dẫn có dạng $\theta = A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} ... \xrightarrow{R_i} A_{i+1}$, trong đó A_i là một trạng thái và R_i là một quan hệ tổng hợp giữa A_i và A_{i+1}. Ký hiệu này biểu thị toán tử tổng hợp trên các quan hệ. Độ dài của một MetaPath được xác định bởi số lượng quan hệ theo một mô hình cụ thể. Trong mô hình, A_i đề cập đến một đỉnh từ các đồ thị đã đề cập trước đó, và R_i chỉ ra một cạnh mới nên được thêm vào giữa A_i và A_{i+1} dựa trên một quy tắc tùy chỉnh. Trong thiết kế của nhóm tác giả, DeepVulSeeker áp dụng quy tắc MetaPath length-2. Theo quy tắc này, nếu có một cạnh có hướng giữa hai đỉnh, một cạnh sẽ được thêm có hướng ngược lại giữa chúng. Hầu hết các đồ thị con từ AST, CFG và DFG có dạng cây, có nghĩa là có rất ít "vòng lặp" trong các đồ thị này. Việc huấn luyện các đồ thị dạng cây có thể dẫn đến vấn đề gradient vanishing. Do đó, việc sử dụng quy tắc MetaPath length-2 nhằm giảm thiểu vấn đề này bằng cách cải thiện tính hoàn chỉnh của các đồ thị.

2.3 Programming Languages Sequencing (PLS)

PLS là quá trình then chốt để DeepVulSeeker hiểu được các ngụ ý ngữ nghĩa đằng sau các đoạn mã. Nó mã hóa mỗi token chương trình thành một vector các đặc trưng, cụ thể là các biểu diễn token ngữ cảnh hóa, dựa trên một mô hình đã được tiền huấn luyện. So với một số phương pháp nhúng từ truyền thống yêu cầu đào tạo kỹ lưỡng, PLS sử dụng các



Hình 1: Kiến trúc mô hình

kỹ thuật tiền huấn luyện để tránh mô hình bị overfitting khi kích thước dữ liệu huấn luyện không đủ hoặc bị lệch. Do đó, các kỹ thuật tiền huấn luyện cung cấp các đặc trưng thích hợp hơn. Ký hiệu x_i là một đoạn mã nhất định, và các biểu diễn P_i thu được bởi Phương trình (1).

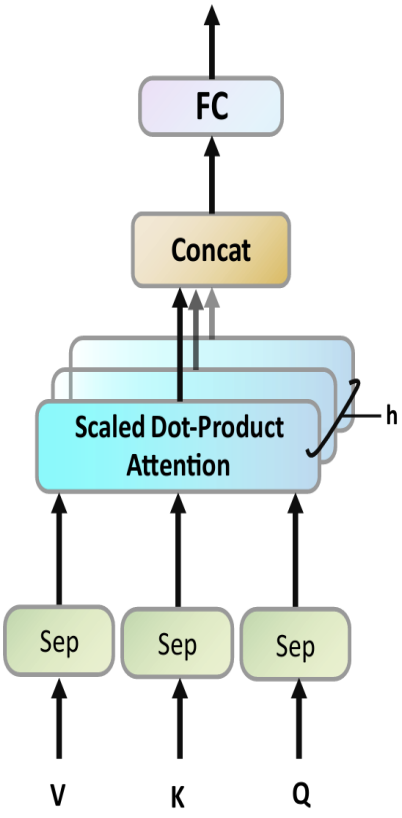
$$P_i = \text{model}(x_i) \tag{1}$$

trong đó model là biểu thức toán học của một mô hình đã được tiền huấn luyện.

2.3.1 Graph Represent Self-Attention Encoding (GRSA). Như đã đề cập, nhóm tác giả thu được ba thông tin cấu trúc (tức là AST, CFG và DFG), và thông tin ngữ nghĩa từ PLS. Bước tiếp theo là kết hợp các biểu diễn này và đưa chúng vào mô hình để tiến hành giai đoạn huấn luyện tiếp theo. Để làm điều này, tác giả đã thiết kế Mạng Mã Hóa Tự Chú Ý Biểu Diễn Đồ Thị (Graph Representation Self-Attention Encoding - GRSA) dựa trên cơ chế tự chú ý đa đầu (multi-head self-attention) . GRSA là một quy trình ba bước như được minh họa trong Hình 2. Đầu tiên, GRSA nhận ba ma trận Q, K và V làm đầu vào, và chia đều mỗi ma trận thành nhiều ma trận con theo số lượng đầu được xác định trước. Các ký hiệu Q, K và V lần lượt là ma trận truy vấn (query), ma trận khóa (key) và ma trận giá trị (value) trong mô hình dựa trên chú ý. Vì áp dụng cơ chế tự chú ý, các ma trận Q, K và V là giống nhau. Trong mô hình, cứ mỗi lần ta sẽ gán mã trận CFG, DFG, AST và PLS lần lượt cho Q, K và V. Thứ hai, GRSA truy xuất các ma trận con được xuất ra từ bước đầu tiên và đưa chúng vào một lớp scaled dot-product attention, trong đó điểm attention được tính toán như Phương trình (2).

$$\mathbf{h}_i = \text{softmax}(\frac{Q_i K_i^T}{\sqrt{d_k}}) \tag{2}$$

trong đó Q_i, K_i và V_i biểu thị ma trận con thứ i của Q, K và V tương ứng. Ký hiệu d_k chỉ số chiều của ma trận K. Cuối cùng, GRSA nối tất cả các ma trận con h_i xuất ra từ lớp chú ý sản phẩm chấm có tỷ lệ và đưa chúng vào một mạng nơ-ron kết nối đầy đủ. Sau quá trình của GRSA, tiến hành



Hình 2: Cấu trúc của GRSA

kết hợp tất cả thông tin từ bốn biểu diễn khác nhau thành một ma trận, chứa cả thông tin cấu trúc và ngữ nghĩa.

3 THỰC NGHIỆM

Trong phần này, ta sẽ nêu chi tiết việc thực hiện DeepVulSeeker được nêu ra ở Hình 1

3.1 Khởi tạo thông tin cấu trúc và ngữ nghĩa

Như đã được trình bày trong Mục 2, các bước đầu tiên và thứ hai là tạo thông tin về CFG, DFG, AST và PLS.

1. Tạo ma trận kề của CFG và DFG: Tác giả đã trích xuất mã C++ bằng cách sử dụng tree-sitter-c, một công cụ tạo trình phân tích cú pháp bao gồm nhiều thư viện hỗ trợ cho việc phân tích, xây dựng các cây cú pháp cụ thể cho các tệp nguồn và cập nhật hiệu quả các cây cú pháp khi chỉnh sửa các tệp nguồn. Sau đó, xây dựng một lớp Node ghi lại các nút đầu, nút cuối và nút tiếp theo của nút hiện tại, để sau này có thể duyệt qua tất cả các nút và các nút kết nối của chúng một cách thuận tiện. Sau đó sử dụng danh sách để lưu trữ tất cả các nút kết nối, thêm các cạnh có hướng giữa các nút này nếu chúng phụ thuộc vào các câu lệnh điều kiện, và tạo ra ma trận kề CFG dựa trên các cạnh có hướng này. Ví dụ, ta có một nút 1, có hai nút con là 2 và 3. Sau đó,thiết lập ma trận M(1,2) = 1, M(1,3) = 1. Quá trình tạo ra ma trận kề DFG tương tự như của ma trận kề CFG. Sự khác biệt duy nhất là thêm các cạnh có hướng dựa trên luồng dữ liệu thay vì các câu lệnh điều kiện.

2. Tạo ma trận kề AST: Tác giả sử dụng Joern, một công cụ phân tích mã nguồn mở, để phân tích các mã nguồn và tạo ra đồ thị thuộc tính mã nguồn (CPG), chứa tất cả thông tin về các cạnh dưới dạng $i \rightarrow j$, biểu thị rằng có một cạnh có hướng từ nút i đến nút j. Sau đó tạo ra ma trận kề tương ứng dựa trên thông tin về các cạnh này.

3. PLS Embedding: Ban đầu, tác giả sử dụng framework HuggingFace Transformers, một nền tảng tích hợp để chia sẻ và tải các mô hình đã được huấn luyện sẵn, cho việc triển khai PLS. Tuy nhiên, tác giả phát hiện rằng các mô hình huấn luyện trước từ HuggingFace cắt sai các token cho các tập dữ liệu, bằng cách áp dụng phương pháp cắt token cho NLP để xử lý mã chương trình. Trong quá trình thực hành, tác giả nhận ra rằng việc cắt token sai có thể làm giảm đáng kể độ chính xác của mô hình. Để giải quyết vấn đề này, tác giả đề xuất sử dụng NLTK, một công cụ xử lý ngôn ngữ tự nhiên mã nguồn mở, để thực hiện nhiệm vụ cắt token. Bằng cách sử dụng công cụ này, ta có thể cắt chính xác các token từ tất cả các đoạn mã nguồn, và đưa các token vào mô hình huấn luyện trước UniXcoder, một mô hình huấn luyện trước đa nền tảng hợp nhất cho các ngôn ngữ lập trình hỗ trợ hiểu các chuỗi token của mã nguồn. Cuối cùng, đối với mỗi đoạn mã nguồn, UniXcoder xuất ra một ma trận PLS embedding chứa các ý nghĩa ngữ nghĩa của nó. Bởi vì UniXcoder chỉ có thể chấp nhận tối đa 512 token. Do đó, tác giả đã cắt bớt các token từ những đoạn có hơn 512 token để đáp ứng yêu cầu của UniXcoder.

3.2 Hiện thực MetaPath và GRSA

Bước thứ 3 của quá trình DeepVulSeeker bao gồm viêc hiện thực MetaPath và GRSA.

- MetaPaths: Như đã đề cập trong Mục 2, tác giả đã sử dụng quy tắc MetaPaths length-2 để thực hiện MetaPaths. Để làm điều này, tác giả đã viết một chương trình Python để thêm các cạnh có hướng ngược lại cho tất cả các cặp nút kết nối. Sử dụng quy tắc MetaPaths length-2 có thể giảm thiểu vấn đề gradient vanishing mà không dẫn đến vấn đề path explosion như quy tắc MetaPaths length-n.
- GRSA: Như đã đề cập trong Mục III, thiết kế GRSA của tác giả được hiển thị trong Hình 4. Chúng tôi đã thực hiện thiết kế này dựa trên Keras, một thư viện học sâu mã nguồn mở dựa trên Python được hỗ trợ bởi TensorFlow. Cuối cùng, các đầu ra của lớp GRSA được đưa vào một mạng nơ-ron tích chập và một mạng nơ-ron fully-connected, được huấn luyện để dự đoán xem đoạn mã cho trước có dễ bị tấn công hay không. Tác giả đã sử dụng hàm mất mát cross entropy để huấn luyện mô hình của mình, được tính toán như sau:

$$\mathcal{L}_{CE} = \sum_{i=0}^{N} y_i \log p_i + (1 - y_i) \log(1 - p_i) \tag{3}$$

trong đó y_i biểu thị các nhãn thực tế và pi là xác suất của nhãn được mô hình tạo ra.

4 ĐÁNH GIÁ MÔ HÌNH

Trong phần này, tác giả tiến hành đánh giá chi tiết về đánh giá DeepVulSeeker. Đầu tiên, tác giả trình bày cấu hình thí nghiệm cho việc đánh giá. Sau đó, tác giả mô tả các phương pháp cơ sở hiện đại gần đây và so sánh hiệu suất tổng thể giữa các phương pháp này và mô hình DeepVulSeeker.

4.1 Cấu hình thí nghiệm

Tác giả trình bày các chỉ số hiệu suất, bộ dữ liệu và cấu hình môi trường được sử dụng để đánh giá.

1. Chỉ số hiệu suất: tác giả sử dụng hai chỉ số thường được dùng trong các nghiên cứu liên quan đến học máy, tức là Accuracy và F1 để đánh giá. Các định nghĩa của hai chỉ số này được định nghĩa ở bên dưới. Tác giả cũng có giải thích các định nghĩa của precision và recall trước khi giải thích F1.

- Accuracy: Tỷ lệ các trường hợp được gán nhãn chính xác so với tổng số trường hợp thử nghiệm.
- Precision: Tỷ lệ các mẫu được dự đoán chính xác so với tổng số mẫu được dự đoán có một nhãn cụ thể. Chỉ số này trả lời các câu hỏi như "Trong tất cả các phiên bản mã nguồn được gán nhãn liên quan đến lỗ hổng, bao nhiêu là đúng?". Precision cao cho thấy tỷ lệ dương tính giả thấp.
- Recall: Tỷ lệ các mẫu được dự đoán chính xác so với tổng số mẫu thử nghiệm thuộc một lớp. Chỉ số này trả lời các câu hỏi như "Trong tất cả các mẫu thử nghiệm có lỗ hổng, bao nhiêu được gán nhãn là tồn tại lỗ hổng?". Recall cao cho thấy tỷ lệ âm tính giả thấp.
- Điểm F1 (F1 Score): Trung bình của Precision và Recall. Đây là một chỉ số hiệu suất quan trọng của mô hình khi dữ liệu thử nghiệm có sự phân bố không đều của các loại lỗ hổng. F1 được tính như sau:

$$2 \text{ X } \frac{\text{Recall x Precision}}{\text{Recall + Precision}} \tag{4}$$

2. Tập dữ liệu: Tác giả đánh giá phương pháp của mình trên sáu bộ dữ liệu ngôn ngữ C/C++ được sử dụng trong

Project	Training Set	Validation Set	Test Set	Total	Vul	Not-vul
FFmpeg	3958	462	499	4919	2438	2481
QEMU	10903	1378	1318	13600	5678	7913
CWE-362	415	56	57	564	189	375
CWE-476	1298	162	162	1623	396	1227
CWE-754	4034	504	505	5043	1359	3684
CWE-758	1089	136	137	1362	367	995

Hình 3: Thông tin tập dữ liệu

nhiều nghiên cứu hiện đại. Bộ dữ liệu này bao gồm bốn bộ dữ liệu tham chiếu đảm bảo phần mềm thông thường, đó là CWE-362, CWE-476, CWE-754 và CWE-758. Bốn bộ dữ liệu này được thu thập từ các phần mềm khác nhau có cấu trúc mã nguồn và chức nuang tương đối đơn giản. Do đó, các lỗ hổng của chúng có thể dễ dàng được xác định hơn. Hai bộ dữ liệu còn lại là FFmpeg và QEMU. FFmpeg là một giải pháp mã nguồn mở mạnh mẽ đa nền tảng để ghi, chuyển đổi và truyền phát video. QEMU là một trình giả lập hệ điều hành cực kỳ mạnh mẽ cho phép chạy các hệ điều hành cho bất kỳ máy nào, trên bất kỳ kiến trúc được hỗ trợ nào. Mã từ hai bộ dữ liệu này rất phức tạp. Do đó, các lỗ hổng của chúng khó được xác định hơn so với các bộ dữ liệu CWE. Thông tin chi tiết, bao gồm kích thước của các tập huấn luyện, tập kiểm tra, tập thử nghiệm, số lượng lỗ hổng và số lượng không có lỗ hổng được liệt kê trong Hình 3.

3. Cấu hình môi trường: Cấu hình phần cứng của nhóm chúng em là máy tính cá nhân với cấu hình CPU là 13th Gen Intel Core i5-13500H (16 CPUs), 16GB RAM, Intel Iris Xe Graphic GPU và sử dụng hệ điều hành Windows 11. Vì có các hạn chế về mặt phần cứng nên nhóm chỉ tiến hành thực nghiệm 3/6 tập dữ liệu bao gồm: CWE-362, CWE-476 và CWE-758.

4.2 Kết quả thực nghiệm

Dựa vào mã nguồn được tác giả cung cấp cùng với việc tìm hiểu đoạn chi tiết và chỉnh sửa đoạn mã nguồn đó, nhóm đã tiến hành thực nghiệm thành công trên ba tập dữ liệu là CWE-362, CWE-476 và CWE-758. Kết quả thực nghiệm của nhóm so sánh với kết quả của tác giả ở ba tập dữ liệu nêu trên trên được đề cập trong Hình 4.

Kết quả trên cho thấy trên ba tập dữ liệu, các số liệu giữa nhóm và tác giả cũng không có khác biệt quá lớn. Mặc dù thông số có phần thấp hơn tác giả nhưng cùng không đáng kể.

Dựa vào kết quả nêu trên, có thể thấy DeepVulSeeker đã đạt được một bước đột phá trong việc phát hiện các lỗ hổng trên các bộ dữ liệu cực kỳ phức tạp, giúp nó phù hợp hơn cho các nhiệm vụ thực tế trong ngành công nghiệp.

Mã nguồn được nhóm tìm hiểu và chỉnh sửa từ tác giả được công khai ở Github. Ngoài ra thông tin công việc mà từng thành viên đóng góp được nêu ra ở Bảng 5

Dataset/Model	Authors		Group 17	
	F1	Accuracy	F1	Accuracy
CWE-362	0.8387	0.9123	0.8152	0.9203
CWE-476	0.8052	0.9080	0.8011	0.8994
CWE-758	0.9859	0.9927	0.9705	0.9854

Hình 4: Kết quả so sánh

Thành viên	Công việc
Hồ Ngọc Thiện	- Tìm hiểu lý thuyết - Chỉnh sửa mã nguồn, khởi chạy và thu kết quả - Viết báo cáo Latex - Làm slide
Dương Phú Cường	- Tìm hiểu lý thuyết - Làm poster
Chu Nguyễn Hoàng Phương	- Tìm hiểu lý thuyết - Chỉnh sửa mã nguồn, khởi chạy và thu kết quả

Hình 5: Đánh giá công việc

5 KẾT LUẬN VÀ NGHIÊN CỨU TRONG TƯƠNG LAI

Trong bài báo này, tác giả đề xuất DeepVulSeeker, một mô hình mới cho việc nhận diện lỗ hổng mã nguồn. DeepVulSeeker có khả nuang nhận diện thông tin cấu trúc và thông tin ngữ nghĩa từ mã nguồn bằng cách tích hợp các công nghệ tiên tiến như các mô hình được huấn luyện trước, mạng nơ-ron đồ thị và cơ chế tự chú ý. Tác giả thử nghiệm DeepVulSeeker trên các bộ dữ liệu đa dạng bao gồm mã nguồn thông thường và phức tạp, và kết quả thực nghiệm cho thấy độ hiệu quả cao hơn so với hầu hết các phương pháp cơ sở tiên tiến hiện nay.

Mặc dù mô hình của được đề xuất vượt qua các phương pháp khác trên các bộ dữ liệu QEMU và FFmpeg, độ chính xác và F1 vẫn cần được cải thiện. Tác giả đã nhận ra vấn đề và phát hiện ra rằng mã nguồn trong hai bộ dữ liệu này thường chứa ngôn ngữ assembly, mà hiện tại phương pháp của DeepVulSeeker chưa thể xử lý được. Trong tương lai, tác giả dự định sửa đổi mô hình của mình với các thuật toán mới để giải quyết vấn đề này.

https://github.com/nt1208/Machine-Learning-for-Security.git