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

LÊ HỒNG HIỂN
NGUYỄN NHẬT QUANG

ĐỒ ÁN CHUYÊN NGÀNH

NGHIÊN CỨU PHƯƠNG PHÁP PHÁT HIỆN LỖ HỔNG BẢO MẬT CHƯƠNG TRÌNH PYTHON HIỆU QUẢ

RESEARCH ON AN EFFECTIVE METHOD FOR DETECTING SECURITY VULNERABILITIES IN PYTHON PROGRAMS

NGÀNH AN TOÀN THÔNG TIN

TP. Hồ Chí Minh, 2025

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

LÊ HỒNG HIỂN - 22520416
NGUYỄN NHẬT QUANG - 22521203

ĐỒ ÁN CHUYÊN NGÀNH

NGHIÊN CỨU PHƯƠNG PHÁP PHÁT HIỆN LỖ HỔNG BẢO MẬT CHƯƠNG TRÌNH PYTHON HIỆU QUẢ

RESEARCH ON AN EFFECTIVE METHOD FOR DETECTING SECURITY VULNERABILITIES IN PYTHON PROGRAMS

NGÀNH AN TOÀN THÔNG TIN

GIẢNG VIÊN HƯỚNG DẪN:
ThS. Phan Thế Duy
ThS. Đỗ Thị Thu Hiền

TP.Hồ Chí Minh, 2025

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 Phan Thế Duy, cô Đỗ Thị Thu Hiền 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.

Lê Hồng Hiển **Nguyễn Nhật Quang**

MỤC LỤC

DANH MỤC CÁC KÝ HIỆU, CÁC CHỮ VIẾT TẮT

ML	**Machine Learning**
DL	**Deep Learning**
LLM	**Large Language Models**
CBERT	**Mô hình CodeBERT**
QWEN	**Mô hình Qwen2.5-Coder**
GRAPH, G	**Mô hình Graph**

DANH MỤC CÁC HÌNH VẼ

DANH MỤC CÁC BẢNG BIỂU

CHƯƠNG 1. TỔNG QUAN

1.1. Bối cảnh vấn đề

Python là một trong những ngôn ngữ lập trình phổ biến nhất hiện nay, được sử dụng rộng rãi trong các hệ thống web, phần mềm máy chủ, trí tuệ nhân tạo và khoa học dữ liệu. Tuy nhiên, mã nguồn Python có thể dễ dàng phát sinh các lỗ hổng bảo mật do những đặc điểm sau:

- Không có kiểm soát chặt chẽ về kiểu dữ liệu và truy cập bộ nhớ.
- Lạm dụng các hàm nguy hiểm như `eval`, `exec`, `pickle`, v.v.
- Thiếu kiểm tra đầu vào, thao tác với tệp tin và xử lý mạng không an toàn.

Trong bối cảnh an toàn thông tin ngày càng trở nên quan trọng, nhu cầu phát hiện tự động các lỗ hổng bảo mật trong mã nguồn Python trở thành một vấn đề cấp thiết nhằm:

- Bảo vệ hệ thống phần mềm khỏi các cuộc tấn công khai thác lỗ hổng.
- Hỗ trợ kiểm thử bảo mật trong quy trình phát triển phần mềm (DevSecOps).
- Giảm thiểu chi phí và thời gian rà soát mã thủ công.

Vấn đề đặt ra

Các phương pháp truyền thống trong phát hiện lỗ hổng bảo mật có những hạn chế đáng kể:

- **Phân tích tĩnh (Static Analysis): nhanh và dễ tích hợp vào quy trình CI/CD, nhưng dễ bỏ sót các lỗi phức tạp liên quan đến ngữ cảnh hoặc logic chương trình.**
- **Phân tích động (Dynamic Analysis): có khả năng phát hiện lỗi khi chạy thực tế, nhưng yêu cầu môi trường chạy đầy đủ và tốn nhiều thời gian.**
- **Rà soát thủ công (Manual Code Review): có độ chính xác cao nếu do chuyên gia thực hiện, nhưng tốn nhân lực và không khả thi ở quy mô lớn.**

Trong thời gian gần đây, các nghiên cứu đã chuyển hướng sang ứng dụng các kỹ thuật học máy (Machine Learning), đặc biệt là:

- **Mô hình ngôn ngữ lớn (LLM) như CodeBERT, Qwen2.5 để học đặc trưng ngữ nghĩa từ mã nguồn.**
- **Mạng nơ-ron đồ thị (GNN), đặc biệt là RGCN, để khai thác cấu trúc điều khiển và phụ thuộc dữ liệu thông qua biểu diễn Program Dependence Graph (PDG).**

1.2. Mục tiêu nghiên cứu

Mục tiêu chính của đề tài là xây dựng một phương pháp hiệu quả để phát hiện lỗ hổng bảo mật trong mã nguồn Python bằng cách:

- **Ứng dụng các mô hình học sâu (Deep Learning) và mô hình ngôn ngữ lớn (LLM) trong bài toán phát hiện lỗ hổng.**
- **Phối hợp hai hướng tiếp cận:**
  - **Hướng ngữ nghĩa (semantic): biểu diễn nội dung mã nguồn bằng embedding từ các mô hình như CodeBERT và Qwen2.5.**

  - **Hướng cấu trúc (graph): biểu diễn quan hệ điều khiển và dữ liệu của chương trình thông qua đồ thị PDG.**

- **Kết hợp (fusion) embedding từ hai hướng để tăng khả năng học và dự đoán chính xác hơn.**
- **Đánh giá mô hình qua các chỉ số: Accuracy, Precision, Recall, F1-score, và AUC-ROC.**

1.3. Ý nghĩa thực tiễn

Đề tài có giá trị thực tiễn cao với các đóng góp như:

- **Tăng hiệu quả và độ chính xác trong việc phát hiện lỗ hổng bảo mật phần mềm.**
- **Góp phần tự động hóa quy trình kiểm thử bảo mật trong các hệ thống phần mềm hiện đại.**
- **Mở rộng hướng nghiên cứu ứng dụng mô hình LLM và GNN trong lĩnh vực phân tích mã nguồn và an toàn thông tin.**

CHƯƠNG 2. KIẾN THỨC NỀN TẢNG

2.1. Học sâu (Deep Learning) và Học máy (Machine Learning)

- **Học máy (Machine Learning) là lĩnh vực nghiên cứu các thuật toán giúp máy tính học được từ dữ liệu.**
- **Học sâu (Deep Learning) là một nhánh của học máy, sử dụng các mạng nơ-ron sâu (Deep Neural Networks – DNNs) để trích xuất đặc trưng phức tạp từ dữ liệu đầu vào như hình ảnh, văn bản, hoặc mã nguồn.**
- **Các mô hình học sâu đã đạt hiệu quả vượt trội trong nhiều bài toán như dịch máy, xử lý ngôn ngữ tự nhiên (NLP), và phát hiện mã độc.**

2.2. Mô hình ngôn ngữ lớn (Large Language Model – LLM)

- **LLM là các mô hình học sâu rất lớn (với hàng tỷ tham số), được huấn luyện trên tập dữ liệu văn bản khổng lồ nhằm học cách biểu diễn và hiểu ngôn ngữ tự nhiên hoặc ngôn ngữ lập trình.**
- **LLM có thể thực hiện các tác vụ như sinh mã (code generation), dịch mã (code translation), và phân tích mã nguồn.**

Một số mô hình LLM tiêu biểu được sử dụng trong đề tài:

- **CodeBERT: Mô hình pre-trained dựa trên kiến trúc BERT, được huấn luyện trên dữ liệu song ngữ gồm ngôn ngữ lập trình và ngôn ngữ tự nhiên. Hỗ trợ nhiều ngôn ngữ lập trình như Python, Java, JavaScript, v.v.**
- **Qwen2.5-Coder: Mô hình ngôn ngữ lớn do nhóm Qwen (thuộc Alibaba) phát triển, tối ưu hóa cho tác vụ liên quan đến mã nguồn. Có khả năng hiểu ngữ cảnh và cấu trúc mã tốt hơn các mô hình truyền thống.**

2.3. Kỹ thuật Fine-tuning và LoRA

- **Fine-tuning là quá trình huấn luyện lại một phần mô hình đã được pre-trained trên tập dữ liệu chuyên biệt, giúp điều chỉnh mô hình phù hợp hơn với bài toán cụ thể.**
- **LoRA (Low-Rank Adaptation) là một kỹ thuật fine-tune hiệu quả, tiết kiệm bộ nhớ. Nó chỉ thêm và huấn luyện một số lượng nhỏ các tham số có hạng thấp, mà không cần cập nhật toàn bộ mô hình gốc.**

2.4. Đồ thị phụ thuộc chương trình (Program Dependence Graph – PDG)

- **PDG là biểu diễn của chương trình dưới dạng đồ thị có hướng, mô tả các quan hệ:**
  - **Phụ thuộc điều khiển (control dependencies) giữa các câu lệnh có ảnh hưởng đến luồng thực thi.**
  - **Phụ thuộc dữ liệu (data dependencies) giữa các biến được sử dụng và gán giá trị.**

- **PDG giúp mô hình học được luồng điều khiển và dòng dữ liệu trong chương trình – những thông tin mà embedding ngữ nghĩa đơn thuần không thể hiện được.**

2.5. Mạng nơ-ron đồ thị (Graph Neural Network – GNN)

- **GNN là lớp mô hình học sâu thiết kế riêng cho dữ liệu đồ thị. Cho phép mô hình học được đặc trưng từ các node và edge thông qua quá trình truyền thông tin giữa các node lân cận.**
- **RGCN (Relational Graph Convolutional Network) là một biến thể mở rộng của GCN, hỗ trợ xử lý đồ thị có nhiều loại quan hệ (multi-relational graph), rất phù hợp với PDG trong bài toán này.**

2.6. Kỹ thuật Fusion (Kết hợp đặc trưng)

- **Fusion trong học máy là kỹ thuật kết hợp thông tin từ nhiều nguồn đặc trưng khác nhau để cải thiện hiệu suất, độ chính xác và tính ổn định của mô hình.**
- **Các phương pháp phổ biến:**
  - **Early Fusion: kết hợp đặc trưng ngay từ đầu trước khi đưa vào mô hình học.**
  - **Late Fusion: kết hợp kết quả đầu ra từ các mô hình khác nhau.**
  - **Hybrid Fusion: kết hợp cả hai cách trên.**
- **Trong đề tài này, đặc trưng từ hai hướng tiếp cận:**
  - **Ngữ nghĩa (semantic): sử dụng embedding từ CodeBERT hoặc Qwen,**

– **Cấu trúc đồ thị (graph): sử dụng embedding từ PDG, được kết hợp bằng kỹ thuật Early Fusion để tạo ra đầu vào chung cho mô hình phân loại.**

CHƯƠNG 3. PHƯƠNG PHÁP TRIỂN KHAI

Trong đề tài này, nhóm nghiên cứu đề xuất và triển khai một hệ thống phát hiện lỗ hổng bảo mật trong mã nguồn Python bằng cách kết hợp diễn ngữ nghĩa (semantic) và biểu diễn cấu trúc (structural) của đoạn mã. Phương pháp tiếp cận tổng thể bao gồm ba nhánh chính:

1. **Phân tích mã nguồn theo ngữ nghĩa bằng mô hình ngôn ngữ lớn (LLM),**
2. **Phân tích cấu trúc mã nguồn qua đồ thị phụ thuộc chương trình (Program Dependence Graph - PDG),**
3. **Kết hợp cả hai hướng tiếp cận trên (fusion-based).**

Hệ thống được thiết kế linh hoạt, có thể mở rộng, cho phép thay thế các thành phần embedding, mô hình đồ thị hoặc mô hình phân loại trong tương lai.

3.1. Tiền xử lý dữ liệu

Dữ liệu được thu thập từ tập `CodeParrot Github Code Clean (Python-only)` **trên HuggingFace, với hơn 645.000 đoạn mã Python. Trong đề tài này, nhóm lựa chọn ngẫu nhiên 128.000 mẫu để sử dụng.**

Quy trình tiền xử lý bao gồm:

– **Lọc dữ liệu lỗi: loại bỏ các đoạn mã rỗng hoặc sai cú pháp.**

- **Gán nhãn nhị phân: sử dụng quy tắc hoặc công cụ kiểm tra mã tự động để xác định nhãn 0 hoặc 1.**
- **Phân chia tập dữ liệu: theo tỉ lệ:**
  - **Tập huấn luyện (train): 70%**
  - **Tập xác thực (validation): 15%**
  - **Tập kiểm tra (test): 15%**

3.2. Hướng tiếp cận ngữ nghĩa (Semantic-based Approach)

Ở hướng này, mã nguồn được xem như một chuỗi ngôn ngữ và được biểu diễn bằng mô hình ngôn ngữ lớn (LLM). Quá trình triển khai:

- **Token hóa mã nguồn: sử dụng tokenizer tương ứng của mô hình LLM.**
- **Sinh embedding vector: đưa token vào mô hình CodeBERT hoặc Qwen2.5-Coder để sinh vector đặc trưng.**
- **Mô hình sử dụng:**
  - **CodeBERT: mô hình của Microsoft chuyên cho lập trình.**
  - **Qwen2.5-Coder: mô hình mã nguồn mở của Alibaba.**
- **So sánh fine-tuned và không fine-tuned:**
  - **Zero-shot inference: mô hình gốc không huấn luyện lại.**
  - **Fine-tuned: sử dụng kỹ thuật LoRA để tinh chỉnh trên tập huấn luyện.**
- **Phân loại: embedding vector được đưa vào mạng phân loại (ví dụ MLP) để dự đoán nhãn.**

3.3. Hướng tiếp cận cấu trúc đồ thị (Graph-based Approach)

Hướng tiếp cận này xây dựng đồ thị phụ thuộc chương trình (PDG) để mô hình hóa luồng điều khiển và phụ thuộc dữ liệu trong mã nguồn.

- **Xây dựng PDG:**
  - ***Control dependency*: giữa các node thể hiện cấu trúc điều kiện, vòng lặp, gọi hàm,...**
  - ***Data dependency*: giữa các node khai báo và sử dụng biến.**
- **Tạo node embedding: tên các node trong đồ thị được đưa vào CodeBERT để tạo vector 768 chiều.**
- **Xây dựng mô hình đồ thị:**
  - **Sử dụng RGCN để xử lý đồ thị có hai loại cạnh: control và data.**
  - **Áp dụng Global Attention Pooling để tổng hợp vector node thành một vector duy nhất.**
- **Phân loại: vector đầu ra được đưa vào mạng phân loại để dự đoán lỗ hổng.**

3.4. Hướng tiếp cận kết hợp (Fusion-based Approach)

Hướng kết hợp sử dụng cả embedding từ hai hướng semantic và structural để cải thiện hiệu quả mô hình.

- **Tổng hợp embedding:**
  - **Lấy vector ngữ nghĩa từ CodeBERT/Qwen.**

    - **Lấy vector cấu trúc từ RGCN.**

- **Kết hợp: sử dụng kỹ thuật *concatenation* hoặc *fusion layer*.**
- **Phân loại: đưa embedding kết hợp vào mạng phân loại MLP.**
- **Lý do kết hợp:**
    - **Ngữ nghĩa nắm bắt nội dung chi tiết từng dòng mã.**
    - **Đồ thị nắm bắt mối quan hệ logic tổng thể.**

3.5. Chiến lược huấn luyện và tối ưu

- **Focal Loss: giảm ảnh hưởng mẫu dễ, tập trung vào mẫu khó (giải quyết mất cân bằng nhãn).**
- **Weighted Sampling: tăng xác suất chọn mẫu thuộc lớp thiểu số trong mini-batch.**
- **Early Stopping: theo dõi chỉ số F1 trên tập validation để dừng huấn luyện sớm.**
- **Learning Rate Scheduler: giảm dần learning rate theo thời gian.**
- **Batch size tối ưu: chọn batch size lớn nhất có thể trong giới hạn bộ nhớ GPU.**

CHƯƠNG 4. THỰC NGHIỆM VÀ ĐÁNH GIÁ

Ở chương này chúng tôi tiến hành tạo môi trường, cài đặt, thực nghiệm với bộ datasets và đưa ra các tiêu chí đánh giá về mức độ hiệu quả của các mô hình.

4.1. Mục tiêu nghiên cứu

Nghiên cứu này hướng đến việc xây dựng một quy trình tự động phát hiện lỗ hổng bảo mật trong mã nguồn Python, với các mục tiêu cụ thể như sau:

4.1.1. Đánh giá khả năng phát hiện của các mô hình

Mục tiêu đầu tiên là thực hiện một loạt thí nghiệm có hệ thống trên bộ dữ liệu mã nguồn Python chứa cả các đoạn code an toàn và các đoạn code có lỗ hổng đã được gắn nhãn. Qua đó, chúng tôi sẽ:

- **Xây dựng pipeline tiền xử lý dữ liệu: làm sạch chuỗi ký tự, loại bỏ bình luận, chuẩn hóa cấu trúc thư mục dự án.**
- **Huấn luyện và tinh chỉnh siêu tham số cho từng mô hình.**
- **Ghi nhận các thông số (Accuracy, Precision, Recall, F1-Score) để đánh giá chi tiết mặt mạnh, mặt hạn chế của từng phương pháp.**

4.1.2. So sánh ba hướng tiếp cận chính

Để hiểu sâu hơn ưu – nhược điểm của từng nhóm phương pháp, nghiên cứu sẽ phân tích và đối chiếu ba hướng tiếp cận tiêu biểu:

1. **Ngữ nghĩa (Semantic-based):**
   - **Sử dụng embedding do các mô hình ngôn ngữ lớn (LLM) và mô hình học sâu như CodeBERT, Qwen2.5 sinh ra.**
   - **Khai thác ngữ cảnh của biến, hàm, lớp trong mã nguồn để phát hiện các mẫu code "độc hại" tiềm ẩn.**
2. **Đồ thị chương trình (Graph-based):**
   - **Xây dựng Program Dependence Graph (PDG) nhằm mô hình hóa dòng chảy điều khiển và dữ liệu giữa các câu lệnh.**
   - **Ứng dụng Relational Graph Convolutional Network (RGCN) để học các đặc trưng cấu trúc sâu, từ đó phát hiện các thao tác bất thường có thể dẫn đến lỗ hổng.**
3. **Kết hợp (Fusion-based):**
   - **Tích hợp đồng thời embedding ngữ nghĩa và biểu diễn đồ thị để tận dụng ưu điểm của cả hai.**
   - **Thiết kế mô-đun fusion-layer nhằm cân chỉnh thông tin và tối ưu hóa khả năng phân loại.**

4.1.3. Áp dụng và phân tích các chỉ số đánh giá

Để đảm bảo tính khách quan và toàn diện, nghiên cứu sẽ sử dụng đồng thời các metric chuẩn của lĩnh vực:

- **Accuracy: tỉ lệ dự đoán chính xác trên tổng số mẫu.**
- **Precision & Recall: đánh giá sự ổn định giữa việc phát hiện đúng lỗ hổng (Precision) và việc không bỏ sót lỗ hổng (Recall).**

– **F1-Score: chỉ số tổng hợp cân bằng giữa Precision và Recall.**

Các kết quả sẽ được biểu diễn dưới dạng bảng, kèm phân tích ý nghĩa thống kê khi so sánh các mô hình.

4.1.4. Xác định và triển khai mô hình tối ưu

Cuối cùng, dựa trên kết quả đánh giá và phân tích:

– **Lựa chọn mô hình hoặc chuỗi mô-đun có hiệu suất tổng hợp vượt trội ở cả độ chính xác và tốc độ xử lý.**
– **Thiết kế API hoặc tích hợp thành plugin để đưa vào quy trình DevSecOps, cho phép tự động quét mã nguồn khi commit hoặc deploy.**
– **Thử nghiệm thực tế trên một số dự án Python nguồn mở để đánh giá mức độ dễ dàng tích hợp và hiệu quả phát hiện lỗ hổng trong môi trường sản xuất.**

Với lộ trình này, nghiên cứu không chỉ đóng góp về mặt lý thuyết mà còn mang tính ứng dụng cao, giúp doanh nghiệp và tổ chức chủ động phát hiện, khắc phục lỗ hổng ngay từ giai đoạn phát triển, nâng cao tính an toàn và độ tin cậy của phần mềm.

4.2. Môi trường thực nghiệm và công cụ

Phần này mô tả chi tiết các thành phần phần mềm, phần cứng và công cụ được sử dụng để thiết lập, huấn luyện và đánh giá các mô hình phát hiện lỗ hổng bảo mật.

4.2.1. Môi trường phát triển

– **Nền tảng:**

    * **Kaggle Notebook (kernel) đảm bảo khả năng tái lập kết quả và chia sẻ công khai toàn bộ pipeline.**
    * **Giao diện chính: Jupyter Notebook, cho phép tích hợp linh hoạt giữa mã nguồn, đồ thị, bảng kết quả và chú thích.**
- **Phiên bản Python: 3.11.x — lựa chọn nhằm tận dụng các cải tiến về hiệu năng và cú pháp mới nhất.**
- **Quản lý môi trường: Sử dụng `conda` để cô lập các phụ thuộc và dễ dàng tái tạo môi trường trên các máy tính khác.**

4.2.2. Framework và thư viện chính

- **Pytorch (v2.x) — framework chính cho việc xây dựng và huấn luyện các mô hình học sâu (DL).**
- **Hugging Face Transformers & Datasets — cung cấp các mô hình pre-trained (CodeBERT, Qwen2.5) và tiện ích tải/tiền xử lý tập dữ liệu.**
- **Scikit-learn (v1.x) — hỗ trợ các công cụ tiền xử lý dữ liệu, đánh giá và so sánh các phương pháp truyền thống (baseline).**
- **TensorFlow (v2.x) — sử dụng trong một số thí nghiệm đối chứng khi cần tận dụng API Keras.**
- **Pandas, NumPy — thao tác và phân tích dữ liệu dạng bảng, ma trận; tính toán số học nhanh chóng.**
- **Bandit (v1.x) — công cụ tự động quét lỗ hổng Python, được dùng để gán nhãn nhị phân cho code snippets.**

4.2.3. Phần cứng và cấu hình

- **CPU: Intel Xeon 2.20GHz, 2 lõi / 4 luồng — đảm bảo khả năng song song ở mức vừa phải cho các bước tiền xử lý và**

huấn luyện nhỏ.

- **RAM: 30GB — đủ để lưu trữ các batch dữ liệu lớn trong quá trình huấn luyện và đánh giá.**
- **GPU: NVIDIA Tesla P100 (16GB VRAM) — phục vụ cho việc huấn luyện mô hình DL với batch size tối ưu, hỗ trợ CUDA 11.x và cuDNN 8.x.**
- **Ổ cứng: SSD 100GB (kaggle workspace) — lưu trữ mã nguồn, dữ liệu trung gian và kết quả thí nghiệm.**

4.2.4. Công cụ gán nhãn tự động: Bandit

- **Bandit là một công cụ mã nguồn mở do OpenStack phát triển, chuyên quét lỗ hổng trong mã Python.**
- **Cấu hình rule-set tiêu chuẩn (B101–B603) để phát hiện các anti-pattern như:** `eval()`, `exec()`, `pickle.loads()`, **mở file không an toàn, v.v.**
- **Xuất báo cáo dưới dạng JSON, từ đó trích xuất số lượng và mức độ nghiêm trọng của cảnh báo để gán nhãn:**

$$\text{label} = \begin{cases} 1, & \text{nếu có 1 cảnh báo bảo mật của mã nguồn}, \\ 0, & \text{nếu không có cảnh báo bảo mật nào}. \end{cases}$$

Với cấu hình phần mềm – phần cứng và quy trình gán nhãn như trên, toàn bộ thí nghiệm đảm bảo tính minh bạch, khả năng tái lập và đối chiếu công bằng giữa các phương pháp.

4.3. Tập dữ liệu - Datasets

Phần này trình bày chi tiết về nguồn, kích thước, quy trình tiền xử lý và cách chia tập dữ liệu sử dụng trong nghiên cứu.

4.3.1. Nguồn dữ liệu

Tập dữ liệu được trích xuất từ kho mã nguồn mở `codeparrot/github-code-cle` trên nền tảng Hugging Face[1]. Đây là một bộ sưu tập hàng trăm nghìn đoạn mã Python thu thập từ các repository công khai trên GitHub, đã qua bước làm sạch cơ bản (loại bỏ các file nhị phân, tập tin cấu hình, v.v.).

4.3.2. Quy mô và lựa chọn mẫu

- **Ngôn ngữ: Python.**
- **Số lượng mẫu: Chọn ngẫu nhiên khoảng 128.000 đoạn mã, tương đương khoảng 20% tổng số mẫu trong kho datasets trên nền tảng Hugging Face, nhằm đảm bảo tính đại diện và giảm thiểu thời gian huấn luyện.**

4.3.3. Tiền xử lý dữ liệu

Để tập trung vào logic và cấu trúc của mã, các bước tiền xử lý bao gồm:

1. **Loại bỏ thành phần không cần thiết:**
   - **Xóa toàn bộ `comments` (các dòng bắt đầu bằng #).**
   - **Loại bỏ `docstring` (các chuỗi ký tự đa dòng nằm ở đầu hoặc giữa hàm/lớp).**
   - **Bỏ qua các dòng trống hoặc chỉ chứa khoảng trắng.**
2. **Chuẩn hóa định dạng:**
   - **Đồng nhất kiểu thụt lề (4 khoảng trắng cho mỗi cấp).**
   - **Loại bỏ khoảng trắng thừa đầu/cuối dòng.**

[1] `https://huggingface.co/codeparrot/github-code-clean`

- **Thay thế các ký tự tab thành khoảng trắng.**

3. **Gán nhãn (Labeling): Nhãn nhị phân được gán dựa trên kết quả phân tích tĩnh sử dụng công cụ Bandit – một trình quét lỗ hổng chuyên dụng cho mã nguồn Python. Quy trình như sau:**
   (a) **Chạy Bandit với cấu hình rule-set tiêu chuẩn để sinh báo cáo các vấn đề an ninh (ví dụ: lệnh gọi `eval`, `exec`, thiếu kiểm tra đầu vào, v.v.).**
   (b) **Phân tích kết quả đầu ra của Bandit, đánh dấu mỗi code snippet:**

$$\text{label} = \begin{cases} 1, & \text{nếu phát hiện có lỗ hổng}, \\ 0, & \text{nếu không phát hiện lỗ hổng nào}. \end{cases}$$

   (c) **Đối với các trường hợp cảnh báo mơ hồ, kết quả được xác thực và hiệu chỉnh thủ công bởi chuyên gia an ninh phần mềm.**

4.3.4. Chia tập dữ liệu

Để đảm bảo tính cân bằng và khả năng tổng quát hóa, tập dữ liệu đã được chia theo tỉ lệ stratified như sau:

- **Train: 70%** (≈ 89.600 **mẫu**)
- **Validation: 15%** (≈ 19.200 **mẫu**)
- **Test: 15%** (≈ 19.200 **mẫu**)

Việc chia stratified đảm bảo rằng tỉ lệ mẫu có lỗ hổng và không có lỗ hổng trong từng phân đoạn (train/val/test) tương đương nhau, hạn chế sai số do mất cân bằng lớp.

4.3.5. Thống kê sơ bộ

Bảng 4.1 tổng hợp một số chỉ số thống kê quan trọng của tập dữ liệu sau tiền xử lý.

Bảng 4.1: *Thống kê cơ bản của tập dữ liệu sau tiền xử lý*

Thuộc tính	**Train**	**Validation**	**Test**
Tổng số mẫu	≈ 89.600	≈ 19.200	≈ 19.200
Tỉ lệ có lỗ hổng	$\approx 20\%$	$\approx 20\%$	$\approx 20\%$

Các bước chuẩn hóa, gán nhãn và chia dữ liệu được thực hiện bằng Python (pandas, scikit-learn), đảm bảo reproducibility thông qua thiết lập random seed cố định.

4.4. Tham số huấn luyện

Trong nghiên cứu này, chúng tôi sử dụng năm cấu hình mô hình khác nhau bao gồm: CodeBERT, Qwen2.5-Coder, Graph-based, mô hình kết hợp CodeBERT+Graph và Qwen2.5-Coder+Graph. Các siêu tham số (hyperparameter) quan trọng như learning rate, batch size, số epochs, mixed precision và tần suất logging được hiệu chỉnh riêng cho từng mô hình để đảm bảo hiệu năng và thời gian huấn luyện tối ưu. Việc lựa chọn các giá trị này dựa trên các khuyến nghị của tài liệu gốc, kết quả tuning sơ bộ trên tập validation, cũng như giới hạn phần cứng.

Diễn giải các tham số

– **Learning rate:**

Bảng 4.2: *Các siêu tham số huấn luyện cho từng mô hình*

Tham số	**CBERT**	**QWEN**	**GRAPH**	**CBERT+G**	**QWEN+G**
Learning rate	1e-4	1e-4	3e-5	3e-5	3e-5
Batch size	16	16	32	16	16
Epochs	3	3	100	3	3
Mixed precision	fp16	fp16	–	fp16	fp16
Logging steps	500	500	–	100	100

    - **Với các mô hình embedding-based (CodeBERT, Qwen2.5-Coder) thường khởi đầu ở mức 1e-4 để nhanh chóng hội tụ trên tập nhỏ, tránh cập nhật quá lớn làm mất ổn định.**
    - **Đối với Graph-based và các mô hình kết hợp, learning rate giảm xuống 3e-5 nhằm xử lý tốt hơn các đặc trưng cấu trúc và ngữ nghĩa, đồng thời tránh overfitting.**

- **Batch size:**
    - **Các mô hình transformer (CodeBERT, Qwen2.5) và fusion đều dùng batch size 16 để phù hợp với giới hạn VRAM (16GB).**
    - **Mô hình Graph-based cho phép batch size lớn hơn (32) nhờ yêu cầu bộ nhớ thấp hơn cho mỗi sample.**
- **Epochs:**
    - **Transformer-based và fusion chỉ cần 3 epoch nhờ khả năng fine-tune nhanh của pre-trained model.**
    - **Graph-based cần đến 100 epoch để mạng RGCN học đủ sâu các đặc trưng phụ thuộc chương trình.**
- **Mixed precision:**
    - **Sử dụng fp16 trên GPU P100 giúp tăng tốc độ huấn luyện và giảm tiêu thụ bộ nhớ, áp dụng cho tất cả các model transformer và fusion.**

  - **Graph-based không dùng mixed precision do khung triển khai hiện tại chưa ổn định với fp16.**

- **Logging steps:**
  - **Với transformer-based, đặt tần suất log sau mỗi 500 batch để giám sát loss và metric mà không làm gián đoạn huấn luyện.**
  - **Đối với mô hình kết hợp, do pipeline phức tạp hơn, chỉ log sau mỗi 100 batch để kịp thời điều chỉnh khi cần.**
  - **Graph-based bỏ logging tự động, kết quả lưu tự động sau mỗi epoch.**

Tất cả các thí nghiệm đều đặt `seed=42` cho Python, NumPy và PyTorch để đảm bảo reproducibility. Môi trường phần cứng (Intel Xeon + NVIDIA P100) và phần mềm (PyTorch v2.x, Transformers v4.x) được mô tả chi tiết trong mục trước.

4.5. Kết quả thực nghiệm

Bảng 4.3: *Kết quả thực nghiệm các mô hình trên tập dữ liệu phát hiện lỗ hổng mã Python*

Mô hình	**Accuracy**	**Precision**	**Recall**	**F1-score**
CodeBERT (chưa fine-tune)	0.4645	0.2438	0.8094	0.3747
CodeBERT (fine-tuned)	**0.8876**	**0.7181**	**0.7124**	**0.7152**
Qwen2.5-Coder (chưa fine-tune)	0.5725	0.2113	0.4232	0.2818
Qwen2.5-Coder (fine-tuned)	**0.9683**	**0.9468**	**0.8900**	**0.9175**
Graph-based	0.8255	0.6191	0.7209	0.6602
CodeBERT + Graph	**0.8107**	**0.7407**	**0.7872**	**0.7568**
Qwen2.5-Coder + Graph	**0.8744**	**0.8384**	**0.7867**	**0.8078**

4.6. Đánh giá kết quả thực nghiệm

Trong phần này, chúng tôi tiến hành phân tích và đánh giá hiệu quả của các mô hình được huấn luyện trong nhiệm vụ phát hiện lỗ hổng bảo mật trên mã nguồn Python. Các chỉ số đánh giá bao gồm: *Accuracy*, *Precision*, *Recall* và *F1-score*. Bảng 4.3 tổng hợp chi tiết kết quả của các mô hình: CodeBERT, Qwen2.5-Coder, Graph-based, và các mô hình kết hợp giữa biểu diễn ngữ nghĩa và biểu diễn cấu trúc chương trình.

4.6.1. So sánh giữa mô hình chưa fine-tune và fine-tuned

Kết quả cho thấy sự khác biệt rõ rệt giữa các mô hình ở trạng thái chưa fine-tune và sau khi được fine-tune. Cụ thể:

- **Với mô hình CodeBERT, accuracy tăng từ 46.45% (chưa fine-tune) lên đến 88.76% (fine-tuned). Đáng chú ý, precision cũng tăng mạnh từ 0.2438 lên 0.7181, cho thấy khả năng phân loại chính xác các mẫu có lỗ hổng được cải thiện đáng kể.**
- **Tương tự, mô hình Qwen2.5-Coder có sự tiến bộ vượt bậc sau khi fine-tune, với độ chính xác đạt 96.83%, precision là 0.9468, recall đạt 0.89 và F1-score lên tới 0.9175 — đây là kết quả cao nhất trong tất cả các mô hình, chứng minh hiệu quả của việc fine-tune trên tập dữ liệu đặc thù.**

4.6.2. Hiệu quả của mô hình dựa trên biểu diễn đồ thị

Mô hình Graph-based, được xây dựng dựa trên cấu trúc *Program Dependence Graph (PDG)* và huấn luyện bằng *Graph Neural Network*, đạt accuracy là 82.55%, cùng F1-score 0.6602. Dù không

vượt qua các mô hình ngữ nghĩa đã fine-tune, kết quả này vẫn cho thấy tiềm năng của hướng tiếp cận dựa trên cấu trúc chương trình, đặc biệt là khả năng phát hiện mối liên kết giữa các đoạn mã mà biểu diễn thuần túy theo token khó nắm bắt.

4.6.3. Hiệu quả của mô hình kết hợp

Hai mô hình kết hợp giữa biểu diễn ngữ nghĩa và cấu trúc gồm: CodeBERT + Graph và Qwen2.5-Coder + Graph đều cho thấy hiệu quả khả quan hơn so với từng thành phần riêng lẻ (khi chưa fine-tune). Mô hình Qwen2.5-Coder + Graph đạt F1-score 0.8078, trong khi CodeBERT + Graph đạt 0.7568.

Dù kết quả chưa vượt qua mô hình Qwen2.5-Coder fine-tuned đơn lẻ, nhưng việc kết hợp thông tin từ cả biểu diễn ngữ nghĩa lẫn cấu trúc chương trình vẫn mang lại độ chính xác và khả năng tổng quát tốt, mở ra hướng nghiên cứu và cải tiến trong tương lai.

4.6.4. Tổng quan và đề xuất

Từ kết quả thực nghiệm, có thể rút ra một số kết luận chính như sau:

- Việc fine-tune trên tập dữ liệu chuyên biệt có ảnh hưởng lớn đến chất lượng mô hình, đặc biệt với các mô hình ngôn ngữ lớn như CodeBERT hay Qwen2.5-Coder.
- Mô hình Qwen2.5-Coder sau khi fine-tune cho kết quả vượt trội so với các mô hình khác, đồng thời giữ được sự cân bằng giữa precision và recall, làm cho F1-score rất cao.
- Các mô hình kết hợp tuy chưa vượt trội hoàn toàn nhưng đã cải thiện đáng kể so với mô hình đơn lẻ chưa fine-tune, cho

thấy tiềm năng phát triển trong các hệ thống phát hiện lỗ hổng bảo mật phức tạp.

Từ những đánh giá trên, chúng tôi đề xuất trong tương lai nên tập trung khai thác mô hình ngữ nghĩa mạnh như Qwen2.5-Coder, kết hợp với biểu diễn đồ thị và cải tiến kỹ thuật tích hợp đa modal để nâng cao độ chính xác và khả năng phát hiện các lỗ hổng tinh vi trong mã nguồn Python.

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 trong tương lai.

5.1. Kết luận

Trong báo cáo này, chúng tôi đã thực hiện một nghiên cứu toàn diện về các phương pháp tự động phát hiện lỗ hổng bảo mật trên mã nguồn Python, bao gồm ba hướng tiếp cận chính: *semantic-based* (ngữ nghĩa), *graph-based* (cấu trúc đồ thị) và *fusion-based* (kết hợp). Trên cơ sở luận cứ lý thuyết vững chắc và các tập dữ liệu thực nghiệm được gán nhãn bằng công cụ Bandit, năm cấu hình mô hình – bao gồm CodeBERT, Qwen2.5-Coder, Graph-based, CodeBERT+Graph và Qwen2.5-Coder+Graph – đã được huấn luyện và đánh giá với các chỉ số chuẩn: Accuracy, Precision, Recall và F1-score.

Kết quả thực nghiệm cho thấy, sau khi fine-tune, mô hình Qwen2.5-Coder đạt độ chính xác tổng thể (*Accuracy*) lên đến 96.83%, cùng F1-score 0.9175, dẫn đầu toàn bộ các phương án thử nghiệm. Điều này khẳng định tầm quan trọng của việc sử dụng các mô hình ngôn ngữ lớn đã được huấn luyện trước, sau đó được điều chỉnh trên tập dữ liệu đặc thù về lỗ hổng bảo mật. Mô hình CodeBERT cũng cho kết quả cải thiện đáng kể sau fine-tune với Accuracy 88.76%

và F1-score 0.7152, chứng minh giá trị của embedding ngữ nghĩa trong bối cảnh phát hiện các pattern độc hại tiềm ẩn trong code.

Về phía phương pháp graph-based, mặc dù không vượt qua các mô hình semantic-based về F1-score (0.6602), kết quả Accuracy 82.55% của mô hình vẫn cho thấy khả năng khai thác hữu ích mối quan hệ điều khiển và dữ liệu thông qua Program Dependence Graph (PDG). Đây là minh chứng rõ ràng cho việc biểu diễn cấu trúc chương trình có thể bổ trợ cho việc phát hiện các lỗ hổng mà các mô hình chỉ dựa trên chuỗi token khó nắm bắt.

Hai mô hình fusion-based (CBERT+Graph và QWEN+Graph) cũng thể hiện hiệu năng khả quan khi kết hợp đồng thời embedding ngữ nghĩa và biểu diễn đồ thị, đạt F1-score lần lượt 0.7568 và 0.8078. Mặc dù chưa vượt qua Qwen2.5-Coder fine-tuned đơn lẻ, chúng đã chứng minh được tính ổn định và khả năng tổng quát hóa tốt hơn trên nhiều dạng lỗ hổng khác nhau, đồng thời cân bằng hai khía cạnh precision và recall một cách hiệu quả.

Việc so sánh trực tiếp các mô hình trên cùng một tập dữ liệu đã mang lại những nhận định có giá trị đối với cộng đồng nghiên cứu và thực tiễn an ninh phần mềm. Cụ thể, chúng tôi đã rút ra một số điểm cốt lõi:

- **Tác động của fine-tune: Việc điều chỉnh mô hình ngôn ngữ lớn trên dữ liệu lỗ hổng chuyên biệt làm tăng đáng kể khả năng phát hiện, minh chứng qua bước nhảy vọt về precision và recall sau fine-tune.**
- **Vai trò của biểu diễn cấu trúc: Mô hình graph-based dù đơn lẻ không dẫn đầu nhưng cung cấp góc nhìn bổ sung về mối liên kết nội tại trong code, rất hữu ích khi kết hợp với embedding ngữ nghĩa.**

- **Tiềm năng của fusion-based: Sự kết hợp giữa hai dạng biểu diễn giúp cải thiện độ ổn định, giảm thiểu trường hợp lỗi thiên về một chiều và góp phần gia tăng khả năng tổng quát trên những pattern lỗ hổng phức tạp.**

Nhìn chung, nghiên cứu đã chứng minh rằng một hệ thống phát hiện lỗ hổng bảo mật hiệu quả nhất nên tích hợp linh hoạt giữa mô hình ngôn ngữ lớn sau fine-tune và phân tích cấu trúc chương trình. Mô hình Qwen2.5-Coder fine-tuned hiện là lựa chọn tiềm năng nhất về hiệu năng thuần túy, trong khi các kiến trúc kết hợp fusion-based sẽ là nền tảng cho các giải pháp toàn diện, có khả năng mở rộng và tùy biến cao.

Với kết quả và phân tích trên, báo cáo cung cấp một lộ trình tham chiếu rõ ràng cho các nghiên cứu và triển khai thực tế trong lĩnh vực DevSecOps, giúp tự động hóa bước kiểm thử bảo mật ngay từ khâu phát triển và tích hợp liên tục. Những đóng góp này không chỉ nâng cao hiệu quả phát hiện lỗ hổng mà còn góp phần bảo đảm tính ổn định, an toàn và uy tín cho phần mềm trong các môi trường công nghiệp và thương mại ngày càng phức tạp.

5.2. Hạn chế

Mặc dù nghiên cứu đã thiết kế và thực hiện kỹ lưỡng, một số hạn chế sau đây cần được lưu ý:

5.2.1. Tập dữ liệu chưa có nhãn lỗ hổng gốc

- **Bộ data `codeparrot/github-code-clean` chỉ bao gồm các đoạn mã sạch (clean code) mà không có nhãn trực tiếp về lỗ hổng.**

- **Việc gán nhãn lỗ hổng hoàn toàn phụ thuộc vào công cụ Bandit, dẫn đến việc thiếu một "ground-truth" ban đầu do con người xác thực toàn diện.**

5.2.2. Giới hạn của công cụ Bandit

- **Bandit dựa trên một rule-set cố định (B101–B603), chỉ phát hiện được các anti-pattern có trong quy tắc, không bao quát hết mọi kiểu lỗ hổng (ví dụ race-condition, logic bugs).**
- **Tỷ lệ *false positive* và *false negative* có thể cao, đặc biệt với code đã qua sanitize hoặc những đoạn mã phức tạp vượt ra ngoài signature của Bandit.**
- **Thiếu khả năng mở rộng rule-set động hoặc gán mức độ nghiêm trọng tùy biến theo ngữ cảnh dự án.**

5.2.3. QQuy trình tiền xử lý dữ liệu

- **Loại bỏ toàn bộ comment, docstring và dòng trống giúp tập trung vào logic nhưng đồng thời làm mất ngữ cảnh quan trọng, ví dụ chú thích về bảo mật hoặc hướng dẫn cấu hình.**
- **Chuẩn hóa indent, xóa tab và loại bỏ khoảng trắng thừa đôi khi làm thay đổi cấu trúc ban đầu của mã, ảnh hưởng đến việc sinh PDG và biểu diễn graph-based.**

5.2.4. Giới hạn tài nguyên phần cứng

- **Thí nghiệm chạy trên Kaggle Notebook (GPU P100, 16GB VRAM) khiến batch size và mô hình phải kìm hãm để phù hợp với tài nguyên.**

- **Không thể mở rộng fine-tune các mô hình lớn hơn (ví dụ Qwen2.5-XL) hoặc tăng bộ đệm đồ thị (graph cache) do giới hạn bộ nhớ GPU, gây hạn chế về tốc độ huấn luyện và khả năng thử nghiệm các kiến trúc phức tạp.**
- **Khả năng tái lập kết quả trên hệ thống khác (GPU V100/A100, CPU đa lõi hơn) có thể khác biệt về hiệu năng và thời gian, ảnh hưởng đến reproducibility.**

Những hạn chế này gợi ý các hướng cải tiến tiếp theo, bao gồm xây dựng bộ dữ liệu có nhãn thủ công, mở rộng rule-set hoặc kết hợp nhiều công cụ quét, hoàn thiện pipeline tiền xử lý giữ lại ngữ cảnh cần thiết, và sử dụng hạ tầng phần cứng mạnh mẽ hơn để thử nghiệm các mô hình quy mô lớn.

5.3. Hướng phát triển

Dựa trên kết quả hiện tại và những hạn chế đã nêu, đồ án có thể được mở rộng và cải tiến theo các hướng sau:

5.3.1. Mở rộng quy mô và đa dạng của tập dữ liệu

- **Thu thập và gán nhãn từ nhiều nguồn thực tế như Cơ sở dữ liệu CVE, kho GitHub public lớn hơn, và kết quả phân tích CodeQL.**
- **Áp dụng kỹ thuật data augmentation (ví dụ: biến đổi code về dạng AST, thêm/bớt biến) để tăng tính đa dạng và giảm lệ thuộc vào rule-set cố định.**
- **Xây dựng một bộ dữ liệu benchmark riêng, có ground-truth do chuyên gia đánh giá, phục vụ cho so sánh công bằng giữa các nghiên cứu tương lai.**

5.3.2. Thử nghiệm với các mô hình ngôn ngữ lớn (LLM) mới

- **Fine-tune và so sánh hiệu năng của CodeT5+, StarCoder, Claude, GPT-4 code và các LLM khác để tìm ra kiến trúc tối ưu cho nhiệm vụ phát hiện lỗ hổng.**
- **Khảo sát các kỹ thuật parameter-efficient fine-tuning (adapter, LoRA) để tối ưu chi phí tính toán và bộ nhớ.**

5.3.3. Mở rộng sang đa ngôn ngữ lập trình

- **Áp dụng quy trình tiền xử lý, gán nhãn và huấn luyện cho các ngôn ngữ khác như JavaScript, C/C++, Java, nhằm đánh giá khả năng khái quát và tái sử dụng pipeline.**
- **So sánh đặc thù của từng ngôn ngữ (ví dụ: cơ chế quản lý bộ nhớ của C/C++, event-driven trong JavaScript) để điều chỉnh rule-set và kiến trúc mô hình phù hợp.**

5.3.4. Phát triển công cụ kiểm thử thực tế

- **Xây dựng plugin IDE (VSCode, PyCharm) hoặc web-app cho phép lập trình viên kiểm tra tự động ngay khi viết code.**
- **Tích hợp vào quy trình CI/CD dưới dạng bước `security scan` để phát hiện sớm lỗ hổng trước khi merge hoặc deploy.**

5.3.5. Mở rộng khả năng giải thích và định vị lỗi

- **Huấn luyện mô hình để không chỉ phân loại có/không lỗ hổng, mà còn nêu chính xác vị trí (token, dòng lệnh) và loại lỗ hổng.**
- **Kết hợp kỹ thuật Explainable AI (XAI) như attention visualization, gradient-based saliency maps hoặc counterfactual explanations để tăng tính minh bạch cho người dùng.**

5.3.6. Tối ưu hóa hiệu năng và khả năng triển khai

- **Nghiên cứu các phương pháp giảm độ trễ (quantization, pruning) để đưa mô hình lên thiết bị edge hoặc tích hợp vào các hệ thống có giới hạn tài nguyên.**
- **Thiết lập benchmark về thời gian phát hiện và mức tiêu thụ tài nguyên, so sánh giữa các kiến trúc và cấu hình phần cứng khác nhau.**

5.3.7. Mở rộng sang học liên tục và active learning

- **Triển khai cơ chế active learning để mô hình chủ động chọn các đoạn code khó gán nhãn hoặc mới lạ, gửi cho chuyên gia đánh giá nhằm cải thiện chất lượng nhãn qua thời gian.**
- **Xây dựng pipeline học liên tục (continual learning) giúp mô hình cập nhật nhanh khi xuất hiện loại lỗ hổng mới.**

TÀI LIỆU THAM KHẢO

1. **L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python," *Information and Software Technology*, vol. 144, p. 106809, Apr. 2022, doi: 10.1016/j.infsof.2021.106809.**
2. **M. Ehrenberg, S. Sarkani, and T. Mazzuchi, "Python source code vulnerability detection with named entity recognition," *Computers & Security*, vol. 140, p. 103802, Mar. 2024, doi: 10.1016/j.cose.2024.103802.**
3. **H.-C. Tran, A.-D. Tran, and K.-H. Le, "DetectVul: A statement-level code vulnerability detection for Python," *Future Generation Computer Systems*, vol. 163, p. 107504, Feb. 2025, doi: 10.1016/j.future.2024.107504.**
4. **A. Mechri, M. A. Ferrag, and M. Debbah, "SecureQwen: Leveraging LLMs for vulnerability detection in python codebases," *Computers & Security*, vol. 148, p. 104151, Jan. 2025, doi: 10.1016/j.cose.2024.104151.**