Trường Đại học Công nghệ Thông tin – ĐHQG Tp. HCM

Khoa Mạng máy tính và truyền thông

BÁO CÁO TỔNG KẾT ĐỒ ÁN MÔN HỌC

Môn học: Lập trình an toàn và khai thác lỗ hổng phần mềm

Tên chủ đề: FLAG: Finding Line Anomalies (in code) with Generative AI

Mã nhóm: 5 Mã đề tài: CK28

Lớp: NT521.O11.ANTT

1. THÔNG TIN THÀNH VIÊN NHÓM:

(Sinh viên liệt kê tất cả các thành viên trong nhóm)

STT	Họ và tên	MSSV	Email
1	Nguyễn Đạo Ga Đô	21521955	21521955@gm.uit.edu.vn
2	Hoàng Anh Khoa	21522220	21522220@gm.uit.edu.vn
3	Đào Võ Hữu Hiệp	21522065	21522065@gm.uit.edu.vn
4	Phạm Ngọc Thiện	21522627	21522627@gm.uit.edu.vn
5			

2. TÓM TẮT NỘI DUNG THỰC HIỆN:[1]

Phần này tóm tắt nội dung của đồ án, sinh viên báo cáo nội dung chi tiết ở phần BÁO CÁO CHI TIẾT

2.1. Chủ đề nghiên cứu trong lĩnh vực An toàn phần mềm:

☒ Phát hiện lỗ hổng bảo mật phần mềm

☐ Khai thác lỗ hổng bảo mật phần mềm

☐ Sửa lỗi bảo mật phần mềm tự động

☐ Lập trình an toàn

☐ Khác: ……..

2.2. Tên bài báo tham khảo chính:

Tên tiếng Anh: **FLAG: Finding Line Anomalies (in code) with Generative AI**
Tên tiếng Việt (dịch): **FLAG: Tìm dòng mã bất thường với Generative AI**

[1] Ghi nội dung tương ứng theo mô tả

2.3. Tên bài báo tham khảo khác (nếu có):

[1] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt and Ramesh Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," *2022 IEEE Symposium on Security,* p. 754–768, 2022.

[2] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri and Brendan Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," *2023 IEEE Symposium on Security and Privacy (SP),* p. 2339–2356, 2023.

[3] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo and Eng Lieh Ouh, "BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,* p. 1556–1560, 2020.

[4] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan and Abhik Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,* p. 740–751, 2017.

[5] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri and Hammond Pearce, "Fixing Hardware Security Bugs with Large Language Models," *arXiv:2302.01215,* 2023.

[6] Hammad Ahmad, Yu Huang and Westley Weimer, "CirFix: automatically repairing defects in hardware design code," *In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems,* p. 990–1003, 2022.

2.4. Tóm tắt nội dung chính:

"FLAG: Finding Line Anomalies (in code) with Generative AI" tập trung vào việc sử dụng trí tuệ nhân tạo trong sinh ngẫu nhiên ngôn ngữ (LLMs) để tìm kiếm lỗi trong mã nguồn. Nội dung bài báo trình bày một phương pháp mới gọi là FLAG, dựa trên khả năng về ngôn ngữ của LLMs, nhằm hỗ trợ các nhà phát triển trong công việc tìm lỗi và sửa lỗi. Phương pháp này cho phép so sánh mã nguồn gốc với mã nguồn được tạo sinh bởi LLMs và xác định những khác biệt đáng chú ý như các lỗi tiềm ẩn hoặc lỗi chức năng để tiến hành kiểm tra chi tiết hơn. Các đặc điểm như khoảng cách đến các dòng ghi chú và độ tin cậy của LLMs cũng được sử dụng để phân loại các lỗi. Phương pháp

FLAG không phụ thuộc vào ngôn ngữ cụ thể và có thể hoạt động trên mã nguồn không hoàn chỉnh hoặc không biên dịch. Bài báo tiến hành thực nghiệm trên 121 bộ kiểm tra bao gồm mã nguồn C, Python và Verilog, và sử dụng hai LLMs tiên tiến của OpenAI là code-davinci-002 và gpt-3.5-turbo. Kết quả thực nghiệm cho thấy FLAG có thể xác định 101 lỗi và giảm không gian tìm kiếm mã nguồn xuống còn 12-17% so với mã nguồn ban đầu.

2.5. Công việc/tính năng/kỹ thuật mà nhóm thực hiện lập trình và triển khai cho demo:

Đọc và phân tích kĩ ý tưởng, nội dung, phương pháp và kết quả đánh giá của bài báo. Sau khi tìm hiểu và phân tích thì nhóm em hiểu rõ được ý tưởng và cơ chế hoạt động của mô hình trong bài báo. Tìm hiểu rõ hơn về các feature mà bài báo sử dụng để đánh giá và cách phân loại dữ liệu dựa trên feature.

Tìm source code và đọc hiểu các nguyên lý hoạt động của nó. Từ đó có thể dễ dàng cho việc phân tích, sử dụng lại source code.

Tiến hành chạy lại và so sánh kết quả. Kết quả cho thấy sự tương đồng trong bài báo và kết quả chạy lại. Có tìm hiểu và giải thích sự sai số nhưng chưa phân tích được 2 giá trị để đánh giá kĩ kết quả là FPR và TPR trong code.

Xây dựng bộ dataset mới để tiến hành kiểm tra và đánh giá tính chính xác của mô hình ban đầu. Nhóm đã tạo được 12 dữ liệu mẫu để sử dụng cho quá trình kiểm tra.

Chạy và đánh giá bộ dataset vừa tìm được. Tìm được khá nhiều lỗi, kết quả khá tích cực.

3. TỰ ĐÁNH GIÁ MỨC ĐỘ HOÀN THÀNH SO VỚI KẾ HOẠCH THỰC HIỆN:

90%

4. NHẬT KÝ PHÂN CÔNG NHIỆM VỤ:

STT	Công việc	Phân công nhiệm vụ
1	Tìm hiểu, phân tích bài báo	Phạm Ngọc Thiện
2	Tìm hiểu và chạy thử nghiệm mô hình đã được đề cập trong bài báo	Cả nhóm
3	Tìm thêm bộ dữ liệu mới + xử lý dữ liệu	Đào Võ Hữu Hiệp, Nguyễn Đạo Ga Đô

4	Chạy với bộ dataset mới thêm vào	Cả nhóm
5	Đánh giá kết quả	Hoàng Anh Khoa

BÁO CÁO TỔNG KẾT CHI TIẾT

Phần bên dưới của báo cáo này là tài liệu báo cáo tổng kết - chi tiết của nhóm thực hiện cho đề tài này.

Qui định: Mô tả các bước thực hiện/ Phương pháp thực hiện/Nội dung tìm hiểu (Ảnh chụp màn hình, số liệu thống kê trong bảng biểu, có giải thích)

A. GIỚI THIỆU TỔNG QUAN

Lỗi xuất hiện trong source code (mã nguồn) khi nó không hoạt động đúng so với với mục đích ban đầu mà các nhà phát triển dự định triển khai. Những lỗi này có thể gây ra các lỗ hổng bảo mật hoặc dẫn đến các chức năng của chương trình hoạt động không chính xác. Việc tìm kiếm các lỗi này là một quá trình tốn nhiều tài nguyên và công sức, mặc dù hiện nay đã có các công cụ hỗ trợ nhưng chúng thường chỉ hoạt động khi các chương trình đã hoàn thiện (hoặc ít nhất là có thể biên dịch được) và sẽ chỉ tập trung vào một số ít các ngôn ngữ lập trình cụ thể hoặc các lớp (class) bị lỗi. Do đó trong quá trình phát triển, đội ngũ nhà phát triển nên thường xuyên kiểm tra công việc của mình. Các phương pháp thường được sử dụng đó là kết hợp giữa kiểm tra thủ công với các phương pháp như automated testing, static analysis hay fuzzing.

Các tác giả trong bài báo giả định rằng ý định của nhà phát triển (developer) chủ yếu được ghi lại trong source code (mã nguồn) và các comment (nhận xét), và trong môi trường thực tế hai yếu tố này có thể sai sót cần được phát hiện và xử lý. Giả sử một nhà phát triển muốn xem lại code của họ theo cách thủ công: nếu chương trình chỉ bao gồm một vài dòng code và comment thì họ có thể dễ dàng kiểm tra. Tuy nhiên với các dự án lớn, khó có thể kiểm tra theo cách này bởi vì quy mô code lúc này trở nên quá phức tạp. Tìm cách thu hẹp các đoạn mã có thể có rủi ro để kiểm tra là cách xử lý tối ưu. Để giải quyết vấn đề này, các tác giả của bài báo đã đề xuất sử dụng Generative AI, cụ thể ở đây là các mô hình ngôn ngữ lớn (LLMs) để đánh dấu các đoạn mã bất thường thông qua source code và comment.

Các LLM như GPT-3 và Codex hiện nay có khả năng viết code rất tốt. Chúng tạo ra output (kết quả) dựa trên input (dữ liệu vào) mà ta cung cấp. Những input này có thể là source code và comment, lúc này ta có thê yêu cầu các mô hình hoàn thiện code dựa trên input mà ta đã cung cấp. Điều này đặt ra một khả năng thú vị: nếu các dòng code có lỗi chỉ là một phần nhỏ trong source code và phần lớn code không có lỗi, thì việc sử dụng LLM để phát hiện lỗi có hợp lý không? Nếu code do các LLM sinh ra khác so với code ban đầu thì chúng có thể có khả năng bị lỗi.

Từ đó, các tác giả của bài báo đã đề xuất ra FLAG - một mô hình sử dụng LLM để tạo các dòng mã thay thế dựa trên mã nguồn và nhận xét hiện có, những đoạn mã thay thế này được so sánh với mã nguồn của nhà phát triển để xác định các điểm khác biệt. Nhờ vào các LLM, mô hình của các tác giả giúp xác định các lỗi ở cả dạng lỗ hổng bảo mật và lỗi về chức năng.

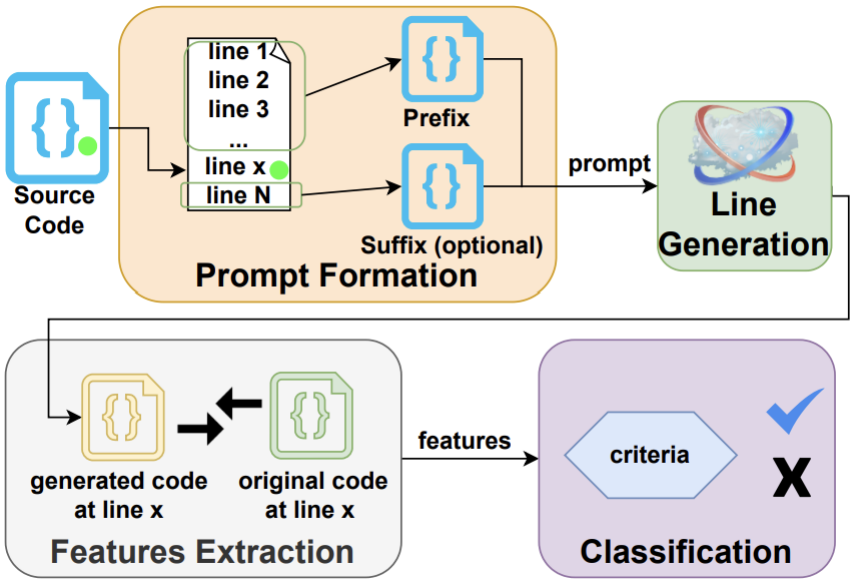
B. PHƯƠNG PHÁP THỰC HIỆN

B.1. Phương pháp được đề xuất

B.1.1. Tổng quan phương pháp

Giới hạn của các công cụ phân tích tĩnh (static detector) và phân tích học máy (machine leanring detector) đòi hỏi sự ra đời của các công cụ không cần phải thiết lập nhiều quy tắc kiểm tra bảo mật và có thể sử dụng trên nhiều ngôn ngữ lập trình. Thông qua các thử nghiệm khác nhau, các tác giả đã chứng minh rằng phương pháp kiểm tra FLAG không phụ thuộc vào các ngôn ngữ lập trình. Bên cạnh đó, FLAG cũng không yêu cầu mã nguồn phải được biên dịch hoặc soạn thảo đúng cú pháp, từ đó cho phép nó hoạt động trên mã nguồn không đầy đủ. Cuối cùng, FLAG không yêu cầu phải tạo các quy tắc kiểm tra bảo mật.

FLAG dựa vào khả năng của LLM để sinh ra mã. Đối với mỗi dòng xác định trong mã nguồn, FLAG tạo ra một *promt* bao gồm phần mã nguồn trước dòng này (gọi là *prefix*) và phần mã sau dòng này (gọi là *suffix*) nếu cần. Prompt bao gồm mã nguồn và nhận xét là input của LLM, output sẽ là một dòng mã hoặc nhận xét. Dòng mã được tạo này được so sánh với dòng mã ban đầu để tạo ra các *thuộc tính* (feature) được sử dụng trong việc phân loại dòng mã gốc có lỗi hay không. Các thuộc tính này cho phép tính toán sự khác nhau giữa hai dòng mã hoặc độ tin cậy của dòng mã do LLM tạo ra. Các dòng mã được phân loại là có khả năng có lỗi sẽ được gắn cờ (flag) để đánh dấu. Mô hình FLAG gồm 4 bước như sau: Prompt Formation, Line Generation, Feature Extraction và Classification.



Hình 1 Mô hình hoạt động của FLAG

Để FLAG tiến hành kiểm tra một file mã nguồn, ta cần phải cung cấp cho FLAG vị trí dòng mã đầu tiên cần kiểm tra. Điều này sẽ cung cấp cho LLM đủ ngữ cảnh để bắt đầu tạo mã và các nhận xét. Thông thường, dòng bắt đầu sẽ ở các vị trí như định nghĩa của một hàm, nhận xét, các khai báo hoặc các phép tính toán hay xử lý. Trong một số trường hợp đặc biệt, ta sẽ cần phải kiểm tra từ đầu file. Ngoài ra, các file cần được xử lý trước để loại bỏ các dòng trống và xác định xem có nhận xét nào trước dòng cần kiểm tra hay không.

B.1.2. Các bước hoạt động của FLAG

B.1.2.1. Prompt Formation

Input của bước này là mã nguồn, cụ thể ở đây là prefix và suffix (nếu cần) và một dòng cụ thể trong mã nguồn cần phân loại (ta sẽ gọi là target line), output sẽ là một prompt. Prompt được tạo ra sẽ được gởi đến LLM để thực hiện bước 2 – Line Generation. Để xác định được target line, FLAG sẽ lặp lại bước này cho tất cả các dòng code trong file mã nguồn, tương ứng với prefix sẽ tăng và suffix sẽ giảm cho mỗi lần lặp.

FLAG giới hạn độ dài tối đa của prefix và suffix là 50 dòng do giới hạn về token của các LLM. Prompt cũng được điều chỉnh để cho ra phản hồi tốt hơn từ LLM, ví dụ: nếu LLM tạo ra nhận xét khi ta cần tạo mã, ta sẽ cần thêm một vài ký tự đầu tiên của dòng mã gốc vào prefix. Quá trình này có thể được lặp lại nhiều lần cho đến khi nhận được prompt hợp lệ.

B.1.2.2. Line Generation

Input của bước này sẽ là prompt từ output của bước 1, output sẽ là dòng code hoặc nhận xét được sinh ra bởi LLM. Output của LLM có thể khác so với target line ban đầu, trong trường hợp này, nó sẽ được gắn cờ để kiểm tra xem có lỗi hay không.

Bên cạnh đó, LLM cần được hướng dẫn để tạo ra output hợp lý vì đôi khi LLM có thể trả về một dòng trống hoặc trả về một nhận xét thay vì mã mà ta cần. Từ đó, thuật toán chi tiết của bước này được mô tả như sau:

```
procedure GENERATE(orig_lines, loc, num_lines )
If loc > max_pre_len
    prefix ← orig_lines[loc-max_pre_len:loc]
Else prefix ← orig_lines[:loc]
If num_lines – loc > max_suf_len
    suffix ← orig_lines[loc+1:loc+max_suf_len]
Else suffix ← orig_lines[loc+1:]
max_attempts ←3, attempts ←0, generated ←False
while generated==False
    Try
        If attempts > 0
            Append orig_lines[loc][:4] to prefix
            response ← Generation for given prefix and
                       suffix (optional) by LLM
    Except Note error
    Else
        If response==‘’ and attempts < max_attempts:
            Increment attempts
        Elif orig_lines[loc] is code and response is not
            Increment attempts
    Else generated ←True
end while
end procedure
```

Hình 2 Thuật toán Line Generation

Trong đó:

- *orig_lines* là danh sách chứa các dòng trong file mã nguồn ban đầu. Đánh thứ tự từ 1.

- *loc* là vị trí của dòng mà ta cần LLM sinh ra mã để kiểm tra, nói cách khác, *loc* là vị trí của target line.

- *num_lines* là tổng số dòng trong file mã nguồn ban đầu sau khi đã được xử lý.

- *max_pre_len* và *max_suf_len* để giới hạn độ dài tối đa của prefix và suffix, ở đây là 50.

B.1.2.3. Feature Extraction

B.1.2.3.1 Features

- Levenshtein Distance - ld: là một đo lường khoảng cách chỉnh sửa giữa hai chuỗi. Nó bao gồm ba phép toán: thêm, xóa và thay thế. Tổng số phép toán cần thiết để chuyển đổi một chuỗi thành chuỗi khác chính là khoảng cách Levenshtein. Khi hai chuỗi hoàn toàn giống nhau, kết quả là 0. Trong nghiên cứu này, khoảng cách Levenshtein được sử dụng để so sánh các dòng mã. Vì mục tiêu là phát hiện các lỗi trong mã, khoảng cách Levenshtein được sử dụng như tiêu chí chính để xác định lỗi.

- BLEU (Bilingual Evaluation Understudy Score) là một phép đo được sử dụng để đánh giá sự tương tự giữa một câu cần đánh giá và một câu tham chiếu. Khi hai câu hoàn toàn giống nhau, kết quả là 1.0, khi khác nhau honaf toàn sẽ cho kết quả là 0.0. Trong nghiên cứu này, BLEU được sử dụng để so sánh các bình luận (comment) vì chúng tương tự với ngôn ngữ tự nhiên. Các giá trị BLEU-1 đến BLEU-4 được thu thập, nhưng chỉ BLEU-1 mang ý nghĩa. BLEU-2, BLEU-3 và BLEU-4 có giá trị rất nhỏ và không hữu ích. Do đó, khi nói đến BLEU chính là đề cập đến BLEU-1.

- Distance from comment - dfc: chỉ ra một dòng mã cách xa bình luận gần nhất trước nó bao xa. Nếu dòng mã cũng chứa một bình luận, giá trị dfc sẽ là 0. Nếu không có bình luận trước dòng mã, dfc sẽ không có giá trị. Chỉ có các bình luận trước dòng mã được xem là liên quan đến mã vì đó là phong cách thông thường mà mã được viết.

- logprob (logarithm of probability) trong ngữ cảnh sinh mã bởi mô hình ngôn ngữ lớn (LLM) là logarit của xác suất của token(đề cập đến các đơn vị nhỏ nhất trong một ngôn ngữ, ví dụ như từ, ký tự hoặc subword) được sinh ra. Nếu một token có khả năng được sinh ra cao hơn, nó sẽ có giá trị logprob cao, với giá trị tối đa là 0. Nếu một token có khả năng được sinh ra thấp hơn, nó sẽ có giá trị logprob thấp hơn. Một LLM có xu hướng "chọn" một token có xác suất cao hơn. Đối với một dòng mã được sinh ra, chúng ta lấy trung bình của các giá trị logprob của các token đã được sinh ra. Giá trị logprob gần 0 cho thấy sự tin tưởng hơn trong quá trình sinh mã, trong khi giá trị âm lớn hơn cho thấy sự tin tưởng ít hơn.

B.1.2.3.2 Extraction

Để có được các giá trị đặc trưng trên, các dòng mã ban đầu và dòng mã được tạo ra được loại bỏ các khoảng trắng cuối dòng. Nếu một trong hai dòng mã là sự kết hợp của mã và bình luận, mã và bình luận được tách riêng để so sánh. Mã của dòng mã ban đầu được so sánh với mã của dòng mã được tạo ra để tính toán khoảng cách Levenshtein. Bình luận của dòng mã ban đầu được so sánh với bình luận của dòng mã được tạo ra để tính toán chỉ số BLEU. Ngoài ra, nếu dòng mã ban đầu là một bình luận, vị trí của bình luận trước đó gần nhất được cập nhật thành dòng hiện tại và dfc được tính toán.

B.1.2.4. Classification

Việc phân loại cho một tệp xác định có đầu vào là các đặc tính(features) nói trên và các dòng để gắn cờ dựa trên một số điều kiện. Các điều kiện này được gọi là "tiêu chí," và các dòng được đánh dấu là "reported_lines." Các tiêu chí có thể là bao gồm hoặc loại trừ. Các tiêu chí bao gồm được thiết kế để đưa các dòng vào tập reported_lines. Các tiêu chí loại trừ được sử dụng để loại bỏ các dòng khỏi reported_lines. Các tiêu chí bao gồm sử dụng hai ngưỡng trong khung công cụ FLAG, đó là ngưỡng giới hạn trên Levenshtein distance (ld_limit) và giới hạn distance from comment (dfc_limit).

Tại sao sử dụng ld_limit? ld chỉ ra sự khác biệt giữa hai đoạn mã. Nếu ld bằng 0, hai đoạn mã giống nhau. Điều này có nghĩa là mô hình LLM không có đề xuất thay thế nào, vì vậy không có lí do để đánh dấu mã này. Nếu ld lớn hơn 0, hai đoạn mã khác nhau, cho thấy mô hình LLM đang đề xuất một lựa chọn thay thế cho dòng mã gốc. Điều này có thể là đáng để đánh dấu. Tuy nhiên, nếu ld quá cao, có thể cho thấy mô hình LLM đang tạo ra một cái gì đó hoàn toàn khác biệt. Dựa trên nhận thức rằng phiên bản lỗi của mã thường rất tương tự với phiên bản đã sửa, chúng ta sử dụng một giới hạn trên ld để nhắm vào mã được tạo ra khác biệt nhưng không quá khác biệt.

Tại sao sử dụng dfc_limit? dfc chỉ ra khoảng cách của một dòng mã đến bình luận gần nhất trước đó. Nếu có một bình luận gần mã, tức là dfc có giá trị thấp, chúng ta giả thuyết rằng mô hình LLM sẽ tạo ra mã với thông tin liên quan, do đó chúng ta tin tưởng nó hơn. Nếu dfc lớn hơn dfc_limit, bình luận(comment) có thể không liên quan đến dòng mã, và chúng ta loại bỏ nó. Sử dụng dfc giúp nới lỏng tiêu chí của chúng ta trong việc lựa chọn dòng mã để đánh dấu, vì một dòng mã có ld lớn hơn ld_limit vẫn có thể được đánh dấu nếu dfc nhỏ hơn dfc_limit.

Tiêu chí đơn giản C0 sử dụng ld_limit làm ngưỡng của nó:

C0(ld_limit): 0 < ld <= ld_limit

Tiêu chí phức tạp hơn C1 sử dụng cả hai ngưỡng:

C1(ld_limit, dfc_limit): 0 < ld AND (ld <= ld_limit OR 0 < dfc < dfc_limit)

Tiêu chí cuối cùng C2 cũng sử dụng cả hai ngưỡng và áp dụng hàm reduce_fp() để loại bỏ các kết quả dương tính sai.

C2(ld_limit, dfc_limit): 0 < ld AND (ld <= ld_limit OR 0 < dfc < dfc_limit) AND reduce_fp()

Một tác dụng phụ của việc phát hiện các lỗi là số lượng lớn kết quả dương tính sai(false positive) trong reported_lines. Để giải quyết vấn đề này, chúng ta thực hiện một số biện pháp trong hàm reduce_fp(). Quá trình này kiểm tra reported_lines để loại bỏ một số dòng được đánh dấu có khả năng là kết quả dương tính sai.

- Biện pháp đầu tiên là tính lại ld sau khi loại bỏ khoảng trắng trong các dòng mã được tạo ra và gốc. Điều này loại bỏ các kết quả dương tính sai mà ld tính cả khoảng trắng, ví dụ như always(@posedge clk) và always (@posedge clk).

- Biện pháp thứ hai là kiểm tra xem dòng gốc có chỉ là một "từ khóa" không. Trong trường hợp này, dòng đó được loại bỏ khỏi reported_lines vì một từ khóa đơn giản không thể gây lỗi.

- Biện pháp thứ ba sử dụng giá trị logprob như ngưỡng để loại trừ. Nếu mô hình ngôn ngữ có giá trị logprob âm lớn cho dòng mã được tạo ra, nó sẽ được loại bỏ khỏi reported_lines vì điều này cho thấy mô hình ngôn ngữ không tự tin trong đề xuất của mình.

B.2. Nội dung cải tiến/phát triển thêm (nếu có)

C. CHI TIẾT HIỆN THỰC VÀ THỰC NGHIỆM PHƯƠNG PHÁP

C.1. Hiện thực phương pháp

C.1.1. Cài đặt

Nhóm sử dụng bộ mã nguồn open source của FLAG framework được viết bằng ngôn ngữ Python do các tác giả cung cấp trên bài báo để tiến hành hiện thực phương pháp. Bộ mã nguồn được cung cấp tại: https://zenodo.org/records/8012211.

Để tiến hành thực nghiệm, các tác giả đề xuất sử dụng 2 LLMs của OpenAI. LLM đầu tiên là code-davinci-002, đây là một LLM được tối ưu hóa cho các tác vụ tự động hoàn thành mã nguồn. LLM thứ hai là gpt-3.5-turbo, đây là phiên bản cải tiến của GPT-3, nó có thể hiểu, sinh văn bản hoặc mã nguồn.

C.1.2. Code thực thi

Theo bộ mã nguồn có 2 file thực thi python là: generation.py để tạo đoạn mã và evaluation.py để đánh giá kết quả thu được. Tuy nhiên, với file mã nguồn cần kiểm tra, người dùng cần phải xác định trước vị trí của dòng có lỗi cần kiểm tra để thực hiện sinh mã.

Trong đoạn mã generation.py:

- Có 2 tham số là llm (gpt-3.5-turbo và code-davinci-002) và mode (tùy theo mô hình llm mà có những mode cụ thể: instructed-complete, instructed-insertion của

gpt-3.5-turbo và insertion của code-davinci-002). Ngoài ta nếu trống Arg[2] thì chương trình sẽ thực hiện mode mặc định là auto-complete.

- Chương trình sẽ nhận dữ liệu input từ file inputs.csv. File này bao gồm các trường: benchmark, path, defect-line và start-line.
- Chương trình sẽ preprocessing lại đoạn mã nhận vào (input) và lưu lại nó vào ngay vị trí path của đoạn mã ban đầu. Cụ thể như sau: sẽ lấy start-line mới sau khi bỏ hết những dòng trống hoặc các comment token nhưng không có nội dung. Sau đó tạo 1 map liên kết giữa dòng mới và dòng cũ. Sau cùng ghi lại code đã preprocessing vào file.
- Tìm dòng comment gần nhất.
- Hình thành file output với các trường sau: Original Line Number, Last line of prompt, Original line, Generated line, Comment?, Distance from comment, Levenshtein, Distance, BLEU1, BLEU2, BLEU3, BLEU4, logprobs và avg_logprobs. Đây là các future để đánh giá.
- Thực hiện khởi tạo các trường: Original Line Number, Last line of prompt và Original line. Sau đó kiểm tra xem nó phải là Comment không. Nếu phải thì cờ Comment sẽ được bật và đặt lại biến previous_comment_line_no. Sau dùng tính khoảng cách từ dòng đó đến dòng comment gần nhất (nếu nó không phải là comment) và gán vào trường Distance from comment.
- Thực hiện việc sinh mã bằng hàm llm_line_generator (Thuật toán tương tự Hình 2) với các tham số tương ứng. Đầu ra sẽ là 3 tham số Generated line, logprobs và avg_logprobs.
- Sau cùng thực hiện tính toán các giá trị BLEU (nếu dòng đó là comment) theo các thông số sau:

```
if(result['c'] == '1'):
    result['b1'] = '%s' % float('%.3g' % sentence_bleu(ref_comment, cand_comment, weights=(1, 0, 0, 0)) )
    result['b2'] = '%s' % float('%.3g' % sentence_bleu(ref_comment, cand_comment, weights=(0.5, 0.5, 0, 0)) )
    result['b3'] = '%s' % float('%.3g' % sentence_bleu(ref_comment, cand_comment, weights=(0.33, 0.33, 0.33, 0)) )
    result['b4'] = '%s' % float('%.3g' % sentence_bleu(ref_comment, cand_comment, weights=(0.25, 0.25, 0.25, 0.25)) )
else:
    result['b1'] = ''
    result['b2'] = ''
    result['b3'] = ''
    result['b4'] = ''
```

Hình 3: Tính toán BLEU

Trong đoạn mã Evaluation.py: Ngoài các bước đánh giá như trong bài báo thì đoạn mã này còn có các mode khác nhau như sau: “l.d.”, “d.f.c.”, “both”, “scatter-plot”, “roc” và default:

- L.d. : là chỉ số Levenshtein Distance. Ở mode này sẽ chạy các trường hợp là ld_limits sẽ các trường hợp: 1.0, 5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0,

50.0, … , 100.0 và dfc_limit = 10. Sau mỗi vòng lập sẽ in ra kết quả tương ứng với ld đã chọn. Và ta có thể thấy được ld càng cao thì độ chính xác càng cao. Nhưng độ chính xác khi đã đạt ngưỡng thì ta có tăng ld lên bao nhiêu lần nữa thì nó vẫn sẽ không thay đổi giá trị. Và chỉ số FPR tăng theo giá trị nên cũng không khả quan nếu tăng l.d lên cao quá.

```
ld limit: 35.0
dfc limit: 10
100 defects found out of 121
average f.p. ratio: 0.2354243282699789

performance: 76.4575671730021


ld limit: 40.0
dfc limit: 10
100 defects found out of 121
average f.p. ratio: 0.24492113081323l7

performance: 75.50788691867683


ld limit: 45.0
dfc limit: 10
104 defects found out of 121
average f.p. ratio: 0.25192016486556706

performance: 78.80798351344329


ld limit: 50.0
dfc limit: 10
106 defects found out of 121
average f.p. ratio: 0.25804351404130893

performance: 80.1956485958691


ld limit: 55.0
dfc limit: 10
106 defects found out of 121
average f.p. ratio: 0.25804351404130893

performance: 80.1956485958691
```

```
ld limit: 100.0
dfc limit: 10
106 defects found out of 121
average f.p. ratio: 0.25804351404130893

performance: 80.1956485958691
```

Hình 4: Kết quả mode l.d.

- D.f.c. : là chỉ số Distance from comment. Tương tự với mode trên thì ở đây chỉ số dfc cũng sẽ tăng như vậy và ld_limits=30. Và kết quả cũng giống như trên nhưng ngưỡng max của dfc nhỏ hơn ld.

```
ld limit: 30
dfc limit: 100.0
102 defects found out of 121
average f.p. ratio: 0.24131397864017 54

performance: 77.86860213598246
```

Hình 5: Kết quả mode d.f.c.

- Both : còn ở mode này là cả 2 trường hợp đề biến thiên như mode ld và dfc. Kết quả thu được ngưỡng của các giá trị sẽ bằng ngưỡng của ld

```
ld limit: 100.0
dfc limit: 100.0
106 defects found out of 121
average f.p. ratio: 0.25804351404130893

performance: 80.1956485958691
```

Hình 6: Kết quả mode both

- scatter-plot : ở chế độ này ld và dfc được chỉnh lần lượt là 30 và 10. Kết quả của nó sẽ trả về kết quả của True Positive (tp) và False Positive (fp) bao gồm số lượng và cặp giá trị [ld, dfc] của chúng.

```
ment-matching\FLAG> python .\evaluation.py scatter-plot
115
7595
[['14', '11'], ['7', '12'], ['1', '6'], ['8', '0'], ['28', '
0'], ['9', '0'], ['17', '0'], ['9', '0'], ['8', '1'], ['3',
'2'], ['6', '1'], ['16', '1'], ['5', '1'], ['24', '0'], ['3'
, '2'], ['28', '0'], ['7', '1'], ['9', '0'], ['9', '0'], ['1
5', '0'], ['12', '1'], ['3', '2'], ['13', '3'], ['16', '1'],
 ['5', '1'], ['17', '0'], ['7', '1'], ['9', '0'], ['9', '0']
, ['7', '1'], ['14', '1'], ['6', '1'], ['13', '3'], ['6', 'N
/A'], ['11', 'N/A'], ['4', 'N/A'], ['1', 'N/A'], ['7', 'N/A'
], ['3', 'N/A'], ['26', '13'], ['1', '16'], ['4', '24'], ['1
5', '48'], ['21', '49'], ['3', '50'], ['16', '66'], ['10', '
```

Hình 7: Kết quả mode scatter-plot

- roc : là chế độ dùng để tìm cụ thể False Positive Rate (FPR) and True Positive Rate (TPR) dựa trên tỉ lệ giữa tp hoặc fp trên tổng có nó và giá trị False Nagative (fn). Mode này sẽ chạy từ ld_start = 0 cho đến ld_end là 1 trong các giá trị của list np.concatenate((np.array([-1,0]),np.logspace(0,3,num=50))).

```
PS E:\Documents\HocKi5\NT521.O11.ANTT - LTAT&KTLHPM\code-com
ment-matching\FLAG> python .\evaluation.py roc
[{'ld_start': 0, 'ld_end': -1.0, 'tpr': 0.0, 'fpr': 0.0}, {'
ld_start': 0, 'ld_end': 0.0, 'tpr': 0.0, 'fpr': 0.0}, {'ld_s
tart': 0, 'ld_end': 1.0, 'tpr': 0.08982035928143713, 'fpr':
0.019060956159800833}, {'ld_start': 0, 'ld_end': 1.151395399
3264474, 'tpr': 0.08982035928143713, 'fpr': 0.01906095615980
0833}, {'ld_start': 0, 'ld_end': 1.3257113655901092, 'tpr':
0.08982035928143713, 'fpr': 0.019060956159800833}, {'ld_star
t': 0, 'ld_end': 1.5264179671752334, 'tpr': 0.089820359281437
13, 'fpr': 0.019060956159800833}, {'ld_start': 0, 'ld_end':
 1.7575106248547918, 'tpr': 0.08982035928143713, 'fpr': 0.01
9060956159800833}, {'ld_start': 0, 'ld_end': 2.0235896477251
57, 'tpr': 0.1497005988023952, 'fpr': 0.03582681759831952},
{'ld_start': 0, 'ld_end': 2.329951810515372, 'tpr': 0.149700
5988023952, 'fpr': 0.03582681759831952}, {'ld_start': 0, 'ld
_end': 2.6826957952797255, 'tpr': 0.1497005988023952, 'fpr':
 0.03582681759831952}, {'ld_start': 0, 'ld_end': 3.088843596
477481, 'tpr': 0.16167664670658682, 'fpr': 0.045046096393978
```

Hình 8: Kết quả mode roc

- default : chế độ mặc định này là cài đặt ld_limit = 20 và dfc_limit = 10

```
PS E:\Documents\HocKi5\NT521.O11.ANTT - LTAT&KTLHPM\code-comment-matching\FLAG> python .\evaluation.py

ld limit: 20
dfc limit: 10
91 defects found out of 121
average f.p. ratio: 0.19822596827500832

performance: 71.17740317249917
```

Hình 9: Kết quả default

C.1.3. Cấu hình

Đối với gpt-3.5-turbo thì được cấu hình như sau:

- Auto complete

```
if mode=='auto-complete':
    response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "user", "content": prompt}
    ],
    # logprobs=1,
    stop=['\n'],
    temperature=0,
    top_p= 1,
    max_tokens=150,
    n=1
    )
```

Hình 10: Cấu hình Auto complete - gpt-3.5-turbo

- Instructed Complete

```
elif mode=='instructed-complete':
    SYSTEM_PROMPT = """\
    You are a skilled AI programming assistant. \
    Complete the next line of code.
    """
    response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages = [
        {"role": "system", "content": SYSTEM_PROMPT},
        {"role": "user", "content": prompt},
    ],
    stop=['\n'],
    # logprobs=1,
    max_tokens=150,
    n=1
    )
```

Hình 11: Cấu hình Instructed Complete - gpt-3.5-turbo

- Instructed Insertion

```
elif mode == 'instructed-insertion':
    response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "user", "content": prompt}
    ],
    stop=['\n'],
    # logprobs=1,
    max_tokens=150,
    n=1
    )
```

Hình 12: Cấu hình Instructed Insertion - gpt-3.5-turbo

Đối với code-davinci-002 thì được cấu hình như sau:

- Auto complete

```
if mode=='auto-complete':
    response = openai.Completion.create( model="code-davinci-002", prompt=prompt, suffix=suffix,
        temperature=0,
        max_tokens=150,
        top_p=1,
        n=1,
        frequency_penalty=0,
        presence_penalty=0.6,
        logprobs=1,
        stop=['\n']
    )
```

Hình 13: Cấu hình Auto complete - code-davinci-002

- Insertion

```
elif mode=='insertion':
    response = openai.Completion.create( model="code-davinci-002", prompt=prompt,
        temperature=0,
        max_tokens=150,
        top_p=1,
        n=1,
        frequency_penalty=0,
        presence_penalty=0.6,
        logprobs=1,
        stop=['\n']
    )
```

Hình 14: Cấu hình Insertion - code-davinci-002

C.1.4. Bộ dữ liệu

Để đánh giá tính khả thi của việc sử dụng LLM trong việc tìm ra lỗi trong mã nguồn và nhận xét, các tác giả đã thử nghiệm trên một tập hợp các chương trình mẫu có lỗi về bảo mật và lỗi về chức năng trong các ngôn ngữ như C, Python và Verilog đến từ nhiều nguồn khác nhau. Các tập hợp C1, P1 và V1 chứa các lỗi về bảo mật, các tập hợp C2, P2 và V2 chứa các lỗi về chức năng. Các lỗi này được mô tả trong danh sách 25 lỗi CWE hàng đầu của MITRE. Với mỗi lỗi sẽ gồm một file mã nguồn và vị trí các dòng tương ứng chứa lỗi trong file đó. Các tác giả sử dụng tổng cộng 121 lỗi: 43 cho C, 34 cho Python và 44 cho Verilog.

Source	Lang.	Type	What we used as defects/bugs	#Bugs
C1	C	Sec	Insecure code instances from CVEs and CWEs that authors used to engineer repairs.	13
C2	C	Func	Defects in students' submissions for introductory programming assignments.	30
P1	Python	Sec	Insecure code instances from CVEs and CWEs that authors used to engineer repairs.	12
P2	Python	Func	Bugs extrapolated from patch files for a subset of the bugs in this database.	22
V1	Verilog	Sec	Defects introduced in modules from various sources to evaluate their repair tool.	10
V2	Verilog	Func	Defects introduced in common Verilog modules by authors to evaluate their repair tool.	34

Các lỗi trong C được lấy từ các nguồn [1, 2, 4].

Các lỗi trong python được lấy từ các nguồn [1, 2, 3].

Các lỗi trong Verilog được lấy từ các nguồn [5, 6].

Các lỗi được các tác giả lấy từ những nguồn trên đã được chỉnh sửa, tinh chỉnh cấu trúc và cách biểu diễn cho phù hợp để sử dụng trên source code của tác giả.

C.2. Thực nghiệm phương pháp

C.2.1. Kịch bản thực nghiệm

C.2.1.1 Kịch bản 1

Kịch bản thực nghiệm được triển khai đó là mô phỏng lại kịch bản mà các tác giả đã sử dụng: sử dụng LLM gpt-3.5-turbo với chế độ auto-complete và instructed-complete. Bộ dữ liệu được sử dụng là 121 lỗi trong bộ mã nguồn mà các tác giả cung cấp. Vì nhóm không tìm được API key cho LLM code-davinci-002, nên kịch bản thử nghiệm cho LLM này sẽ không được tiến hành. API được sử dụng ở kịch bản này là gpt-3.5-turbo TIER 5.

C.2.1.2 Kịch bản 2

Ở kịch bản này, nhóm đề xuất sẽ kiểm tra lại tính chính xác của mô hình ban đầu bằng cách kiểm tra thêm 11 bộ mã nguồn C và 1 mã nguồn Python. API được sử dụng ở kịch bản này là gpt-3.5-turbo TIER free.

123	123-cwe_iterative6	../C-bugs/cwe-repair/cwe_iterative/cwe-787/codeql-eg-PotentialBufferOverflow/scenario.c	[16 17 18]	8
124	124-cwe_iterative7	../C-bugs/cwe-repair/cwe_iterative/cwe-119/codeql-eg-OverflowBuffer/scenario.c	[10 12]	5
125	125-cwe_iterative8	../C-bugs/cwe-repair/cwe_iterative/cwe-119/my-eg-1/scenario.c	[12 14]	4
126	126-cwe_iterative9	../C-bugs/cwe-repair/cwe_iterative/cwe-190/mitre-eg-4/scenario.c	[8 11 14 17	4
127	127-cwe_iterative10	../C-bugs/cwe-repair/cwe_iterative/cwe-190/codeql-eg-ArithmeticTainted/scenario.c	[13 14]	4
128	128-cwe_iterative11	../C-bugs/cwe-repair/cwe_iterative/cwe-190/codeql-eg-ArithmeticUncontrolled/scenario.c	[11]	5
129	129-cwe_iterative12	../C-bugs/cwe-repair/cwe_iterative/cwe-787/mitre-eg-2/scenario.c	[40]	4
130	130-cwe_iterative13	../C-bugs/cwe-repair/cwe_iterative/cwe-787/mitre-eg-5/scenario.c	[22]	5
131	131-cwe_iterative14	../C-bugs/cwe-repair/cwe_iterative/cwe-22/codeql-eg-C-TaintedPath/scenario.c	[17 30]	7
132	132-cwe_iterative15	../C-bugs/cwe-repair/cwe_iterative/cwe-78/codeql-eg-c-ExecTainted/scenario.c	[15 17]	7
133	133-cwe_iterative16	../C-bugs/cwe-repair/cwe_iterative/cwe-79/codeql-eg-ExternalAPISinkExample/scenario.c	[23]	7
134	P2-CWE13	../Python-bugs/CWEs/cwe-79/codeql-eg-Jinja2WithoutEscaping/scenario.py	[8]	4

Hình 15: Bộ dữ liệu thêm cho kịch bản 2

C.2.2. Kết quả thực nghiệm

C.2.2.1 Kịch bản 1

Sau gần 32h liên tục chạy các file thực nghiệm trên mã nguồn và bộ dữ liệu ban đầu, trong đó phần lớn thời gian là cho việc sinh mã để tạo các thuộc tính cho việc đánh giá. Nhóm đã thu được các số liệu thống kê cần thiết.

Cấu hình máy tính sẽ không ảnh hưởng nhiều đến thời gian training dữ liệu do quá trình tốn thời gian nhất trong mô hình này đó là yêu cầu sinh code từ LLM. Quá trình này phụ thuộc hoàn toàn vào tốc độ mà nhà cung cấp LLM đó giới hạn. Bên cạnh đó, chi phí trong việc sử dụng các API LLM cũng là một vấn đề đáng quan tâm. Để cho ra kết quả với bộ dữ liệu của tác giả với API gpt-3.5-turbo, nhóm đã tốn khoảng 40$.

Để đánh giá kết quả thực nghiệm, các tác giả sử dụng 3 tiêu chí đó là Number of defects detected (DD), False Positive Rate (FPR) và True Positive Rate (TPR). Nhưng vì 1 số lí do nên nhóm em chưa in ra được FPR và TPR nên nhóm em sẽ so sánh dựa trên DD:

Model	Mode of Completion	Language	Source	# Defects	Criterion 0			Criterion 1			Criterion 2		
					DD	FPR	TPR	DD	FPR	TPR	DD	FPR	TPR
davinci-002	auto complete	C	C1- CVEs	8	1	0.071	0.125	2	0.113	0.250	2	0.098	0.250
			C1-CWEs	5	1	0.101	0.200	2	0.163	0.400	2	0.124	0.400
			C2	30	19	0.088	0.633	25	0.111	0.833	24	0.093	0.800
		Verilog	V1	10	1	0.081	0.100	7	0.216	0.700	7	0.206	0.700
			V2	34	20	0.089	0.588	26	0.172	0.765	25	0.149	0.735
		Python	P1	12	5	0.063	0.417	10	0.165	0.833	7	0.092	0.583
			P2	22	5	0.064	0.227	9	0.163	0.409	9	0.147	0.409
				121	**52**	**0.075**	**0.430**	**81**	**0.138**	**0.669**	**76**	**0.121**	**0.628**
	insertion	C	C1- CVEs	8	2	0.143	0.250	2	0.144	0.250	2	0.126	0.250
			C1-CWEs	5	1	0.147	0.200	2	0.163	0.400	2	0.124	0.400
			C2	30	24	0.108	0.800	25	0.111	0.833	24	0.094	0.800
		Verilog	V1	10	4	0.155	0.400	7	0.215	0.700	7	0.204	0.700
			V2	34	25	0.148	0.735	26	0.174	0.765	25	0.150	0.735
		Python	P1	12	7	0.209	0.583	9	0.296	0.750	7	0.175	0.583
			P2	22	9	0.145	0.409	11	0.196	0.500	11	0.168	0.500
				121	**72**	**0.145**	**0.595**	**82**	**0.162**	**0.678**	**78**	**0.141**	**0.645**
gpt-3.5-turbo	auto complete	C	C1- CVEs	8	0	0.089	0.000	4	0.165	0.500	4	0.137	0.500
			C1-CWEs	5	0	0.147	0.000	3	0.302	0.600	3	0.256	0.600
			C2	30	19	0.162	0.633	27	0.245	0.900	25	0.219	0.833
		Verilog	V1	10	0	0.116	0.000	8	0.352	0.800	8	0.318	0.800
			V2	34	18	0.135	0.529	30	0.294	0.882	29	0.251	0.853
		Python	P1	12	1	0.092	0.083	11	0.282	0.917	11	0.189	0.917
			P2	22	4	0.060	0.182	10	0.193	0.455	10	0.176	0.455
				121	**42**	**0.099**	**0.347**	**93**	**0.210**	**0.769**	**90**	**0.181**	**0.744**
	instructed complete	C	C1- CVEs	8	1	0.096	0.125	3	0.168	0.375	3	0.131	0.375
			C1-CWEs	5	0	0.124	0.000	3	0.279	0.600	3	0.256	0.600
			C2	30	19	0.133	0.633	24	0.224	0.800	22	0.193	0.733
		Verilog	V1	10	0	0.127	0.000	6	0.372	0.600	6	0.318	0.600
			V2	34	14	0.159	0.412	26	0.312	0.765	25	0.257	0.735
		Python	P1	12	2	0.102	0.167	10	0.282	0.833	10	0.194	0.833
			P2	22	3	0.055	0.136	8	0.160	0.364	8	0.144	0.364
				121	**39**	**0.105**	**0.322**	**80**	**0.210**	**0.661**	**77**	**0.172**	**0.636**

Hình 16: Kết quả trong bài báo

Đây là bảng kết quả khi chạy lại từ dataset gốc và đánh giá lại ở 2 mode của gpt-3.5-turbo:

Model	Mode of Completion	Language	Source	# Defects	Criterion 0	Criterion 1	Criterion 2
					DD	DD	DD
gpt-3.5-turbo	auto complete	C	C1- CVEs	8	1	3	3
			C1-CWEs	5	1	2	1
			C2	30	19	26	25
		Verilog	V1	10	0	9	9
			V2	34	19	29	29
		Python	P1	12	0	12	10
			P2	22	3	11	11
				121	**43**	**92**	**88**
	instructed complete	C	C1- CVEs	8	0	1	1
			C1-CWEs	5	0	3	3
			C2	30	13	22	21
		Verilog	V1	10	1	5	5
			V2	34	18	25	23
		Python	P1	12	1	11	10
			P2	22	3	7	7
				121	**36**	**74**	**70**

Nhận xét: Kết quả gần như tương đồng với kết quả trên bài báo nhưng không giống hoàn toàn. Kết quả bị thay đổi so với đánh giá ban đầu của bài báo có thể do lúc thực hiện generation ra các đoạn mã thay thế để kiểm tra sẽ khác nhau do mỗi request đến các model LLM sẽ trả về 1 response khác nhau, từ đó dẫn đến mỗi lần thực hiện generation code và đánh giá kết quả sẽ có sự khác nhau.

C.2.2.2 Kịch bản 2

Kết quả thực nghiệm như sau:

Model	Mode of Completion	Language	Source	# Defects	Criterion 0 DD	Criterion 1 DD	Criterion 2 DD
gpt-3.5-turbo	auto complete	C + python	Demo	12	2	8	8
	instructed complete	C + python	Demo	12	2	9	9

Nhận xét: Mô hình này nhận biết được nhiều bug trong bộ dữ liệu của nhóm phát triển thêm. Tuy nhiên, hiện tại nhóm vẫn chưa trích xuất được các giá trị FPR và TPR cụ thể cho các trường hợp trên để đánh giá chi tiết hơn. Nhóm sẽ tìm hiểu rõ hơn về vấn đề này sau. Bên cạnh đó, Bộ dữ liệu tuy nhỏ nhưng thời gian thực hiện vẫn lâu do giới hạn của API.

D. HƯỚNG PHÁT TRIỂN

Đề tài này rất tiềm năng trong tương lai vì công nghệ LLM hiện nay đang phát triển rất nhanh chóng và việc tìm lỗi dựa trên nó không cần thiết phải thực thi code. Tuy nhiên, hiện tại đề tài này vẫn làm còn khá sơ sài, chưa hoàn chỉnh.

Đầu tiên, bộ dữ liệu của mô hình hiện tại còn được xử lý một cách thủ công, yêu cầu người kiểm tra cần phải xác định tổng quát vị trí lỗi của chương trình cần kiểm tra để đưa vào mô hình (phải xác định target line cần kiểm tra), vì vậy lượng dữ liệu ban đầu để kiểm tra trong bài báo chưa đủ lớn để có thể xác định được chính xác rằng với bộ dữ liệu kiểm tra lớn hơn thì tỉ lệ chính xác có giảm hay không, vì vậy cần phải phát triển thêm cơ chế xác định dòng code cần kiểm tra một cách tự động hoặc không cần phải xác định dòng code này khi đưa vào mô hình.

Thứ hai, các tiêu chí để xác định mã nguồn có lỗi hay không hiện nay trong bài báo còn khá hạn chế, dẫn đến tỉ tệ false positive rate có thể tăng lên với bộ dữ liệu lớn hơn hoặc mã nguồn phức tạp hơn. Vì vậy cần áp dụng thêm nhiều tiêu chí phân loại và đánh giá cho mô hình này để tăng độ chính xác cho mô hình.

Tiếp theo, hiện tại mô hình chỉ có thể kiểm tra lỗi theo các dòng cố định, vì vậy có thể ta phải cần kiểm tra một bộ mã nguồn nhiều lần nếu có nhiều dòng nghi vấn, do đó có thể không phát hiện được những lỗi liên quan đến cấu trúc hay luồng của chương trình. Từ

đó, ta có thể phát triển hệ thống từ việc kiểm tra theo dòng thành kiểm tra theo từng đoạn code xác định.

Cuối cùng, mô hình hiện nay với mỗi lần chạy chỉ sử dụng một LLM cụ thể, vì vậy sẽ gặp phải vấn đề giới hạn tốc độ trả lời truy vấn của các LLM, từ đó dẫn đến thời gian kiểm tra sẽ lâu hơn, vì vậy ta có thể phát triển các cơ chế sử dụng đồng thời nhiều LLM trong một lần chạy để cải thiện tốc độ của hệ thống.

Về tính ứng dụng, các mô hình LLM hiện nay rất mạnh mẽ và tính chính xác được cải thiện liên tục, tốc độ trả lời các truy vấn cũng ngày càng nhanh. Bên cạnh đó, việc sử dụng LLM giúp phát hiện bug là một giải pháp còn khá mới lạ hiện nay với chúng ta nên sẽ còn nhiều khía cạnh có thể khai thác thêm như gợi ý sửa lỗi được phát hiện cho an toàn hơn.

Sinh viên đọc kỹ yêu cầu trình bày bên dưới trang này

YÊU CẦU CHUNG

- Sinh viên tìm hiểu và thực hiện bài tập theo yêu cầu, hướng dẫn.
- Nộp báo cáo kết quả chi tiết những việc (**Report**) bạn đã thực hiện, quan sát thấy và kèm ảnh chụp màn hình kết quả (nếu có); giải thích cho quan sát (nếu có).
- Sinh viên báo cáo kết quả thực hiện và nộp bài.

Báo cáo:

- File .PDF. Tập trung vào nội dung, không mô tả lý thuyết.
- Đặt tên theo định dạng: [Mã lớp]-Project_Final_NhomX_Madetai. (trong đó X và Madetai là mã số thứ tự nhóm và Mã đề tài trong danh sách đăng ký nhóm đồ án).

  Ví dụ: [NT521.O11.ANTT]-Project_Final_Nhom03_CK01.
- Báo cáo tổng kết được nộp cùng slide trình bày và source code trên moodle, nhóm sinh viên tạo thư mục theo quy ước cho trước trên moodle khi nộp các file.
- Nộp file báo cáo trên theo thời gian đã thống nhất tại courses.uit.edu.vn.

Đánh giá:

- Hoàn thành tốt yêu cầu được giao.
- Có nội dung mở rộng, ứng dụng.

Bài sao chép, trễ, … sẽ được xử lý tùy mức độ vi phạm.

HẾT