Khung cấu trúc nội dung

I. Giới thiệu (Thảo)

- **A. Tổng quan:**
  - Tầm quan trọng của việc phát hiện lỗi phần mềm sớm.
  - Giới thiệu về đồ thị lời gọi (Call graph) trong phát hiện cấu trúc và hành vi chương trình.
  - Giới thiệu về suy luận bất biến - invariant trong khả năng phát hiện lỗi.
  - Mục tiêu và phạm vi của báo cáo/phân tích.
- **B. Vấn đề và động lực:**
  - Thách thức trong việc tìm kiếm và sửa lỗi, đặc biệt là các lỗi không gây crash (non-crashing bugs).
  - Sự cần thiết của các kỹ thuật tự động hoặc bán tự động để hỗ trợ quá trình gỡ lỗi.
  - Ưu điểm của việc sử dụng suy luận bất biến lên đồ thị lời gọi so với các phương pháp truyền thống.

II. Cơ sở lý thuyết (Huy)

- **A. Call Graph (Đồ thị lời gọi):**
  - Định nghĩa Call Graph tĩnh và động.
  - Vai trò của Call Graph động trong việc biểu diễn hành vi thực thi của chương trình.
  - Các thành phần của Call Graph: node (phương thức), edge (lời gọi phương thức).
  - Phân tích nội thủ tục (intraprocedural analysis) và phân tích liên thủ tục (interprocedural analysis) trong xây dựng và phân tích đồ thị lời gọi.
- **B. Các loại lỗi (Bug):**
  - Phân loại lỗi: Crashing vs. Non-Crashing, Occasional vs. Non-Occasional.
  - Structure-affecting bugs (lỗi ảnh hưởng cấu trúc Call Graph).
  - Call Frequency-affecting bugs (lỗi ảnh hưởng tần suất gọi).
- **C. Bất biến (Invariant) và suy luận bất biến (Invariant inference):**
  - Định nghĩa các bất biến khả năng (Likely invariants): Các quy tắc hoặc mẫu hành vi thường xuyên xuất hiện trong đồ thị lời gọi.
  - Các phương pháp suy luận bất biến khả năng từ lời gọi (ví dụ: dựa trên tần suất xuất hiện của các mẫu đồ thị con).

III. Các phương pháp tối giản đồ thị lời gọi (Thảo)

- **A. Tổng quan:**
  - Sự cần thiết của việc giảm đồ thị lời gọi để cải thiện hiệu suất của các thuật toán phân tích và suy luận bất biến (invariant).
  - Các phương pháp tối giản đồ thị lời gọi khác nhau và ưu nhược điểm của chúng.
- **B. Các kỹ thuật giảm đồ thị lời gọi:**
  - Total Reduction: Gộp tất cả các node đại diện cho cùng một phương thức.
  - Total Reduction with Edge Weights: Thêm trọng số vào các cạnh để biểu diễn tần suất gọi.
  - Zero-One-Many Reduction: Giảm các cấu trúc con lặp đi lặp lại.
- **C. So sánh các kỹ thuật:**
  - So sánh hiệu quả giảm kích thước của các phương pháp khác nhau.
  - Ảnh hưởng của các kỹ thuật giảm khác nhau đến khả năng phát hiện lỗi.

IV. Quy trình phát hiện lỗi dựa trên đồ thị lời gọi và suy luận bất biến: (Sang)

- 1. Thu thập và tiền xử lý đồ thị lời gọi.

- 2. Tối giản đồ thị lời gọi.
- 3. Suy luận các bất biến khả năng.
- 4. Phân tích thống kê để xác định các bất biến vi phạm.

V. Đánh giá thống kê (Sang)

- **A. Thiết lập thử nghiệm:**
  - Mô tả chi tiết về bộ dữ liệu thử nghiệm (ví dụ: chương trình C++, số lượng phương thức, số lượng lỗi).
  - Các loại lỗi được sử dụng trong thử nghiệm.
- **B. Kết quả thống kê:**
  - So sánh khả năng phát hiện lỗi khi các tham số thay đổi tuyến tính.

VI. Các nghiên cứu liên quan (Huy)

- **A. Phân tích tĩnh:**
  - Các phương pháp phân tích mã nguồn, lịch sử phiên bản.
- **B. Phân tích động:**
  - Các phương pháp tập trung vào luồng dữ liệu.
  - Các phương pháp tập trung vào luồng điều khiển.
- **C. Kết hợp:**
  - Các phương pháp kết hợp cả phân tích tĩnh và phân tích động.
- **D. Sử dụng PDG:**
- Phát hiện ra các vấn đề đã bị bỏ qua

VII. Kết luận và hướng phát triển (Thảo)

- **A. Tóm tắt:**
  - Tóm tắt các kết quả chính và kết luận của phân tích.
- **B. Hướng phát triển:**
  - Các hướng nghiên cứu tiềm năng để cải thiện các kỹ thuật phát hiện lỗi bằng đồ thị lời gọi và suy luận invariant.
  - Thách thức và cơ hội trong việc áp dụng cho các dự án phần mềm lớn và phức tạp.
  - Phân tích sâu hơn đồ thị lời gọi, bao gồm việc kết hợp các thông tin về edge weights.

VIII. Tham khảo (Sang)

- Liệt kê tất cả các tài liệu tham khảo được sử dụng trong báo cáo/phân tích.

Nội dung của Huy

II. Cơ sở lý thuyết

A. Call Graph (Đồ thị lời gọi)

Đồ thị lời gọi là một cách biểu diễn những lời gọi nào có thể xảy ra khi thực thi chương trình

1. Định nghĩa call graph tĩnh và động.

1.1 Call graph tĩnh

- Đồ thị lời gọi tĩnh là sự trừu tượng hóa của một chương trình máy tính biểu diễn tất cả lời gọi hàm hay phương thức có thể xảy ra. Nó được xây dựng bằng cách phân tích mã nguồn mà không cần phải chạy chương trình và không thể hiện thứ tự thực thi chương trình (flow-insensitive)

- Vì xem xét tất cả các lời gọi có thể xảy ra nên đồ thị lời gọi tĩnh có thể bao gồm những lời gọi không bao giờ xảy ra như lời gọi phương thức đa hình, thực tế chỉ có 1 phương thức được gọi nhưng trong đồ thị biểu diễn tất cả lời gọi từ các lớp con ghi đè lại phương thức đa hình đó.

1.2 Call graph động

- Đồ thị lời gọi động biểu diễn luồng điều khiển của một chương trình khi nó được thực thi. Nó hiển thị trình tự các lệnh gọi hàm hay phương thức được thực hiện trong quá trình thực thi chương trình, cùng với các tham số được truyền cho từng hàm.

- Vì nó thực sự thực thi chương trình nên sẽ không có lời gọi không bao giờ xảy ra nào nhưng đồ thị cũng chỉ chứa những lời gọi với những đầu vào cụ thể, những lời gọi chưa được thực thi sẽ không được ghi lại. Ngoài ra đồ thị lời gọi động cũng chiếm một phần bộ nhớ cũng như tài nguyên của bộ xử lý để ghi lại các lời gọi.

2. Vai trò của Call Graph động trong việc biểu diễn hành vi thực thi của chương trình

- Đồ thị lời gọi động là một cách biểu diễn chính xác, cụ thể về cách các hàm hay phương thức gọi nhau khi chạy chương trình, giúp hiểu sâu hơn về các hành vi của chương trình khi chạy.

-Thống kê hiệu suất: đồ thị lời gọi động ghi lại các số liệu về thời gian, số lượng lệnh hoặc mức sử dụng bộ nhớ, giúp ta xác định được các chuỗi lời gọi được thực thi nhiều nhất và xác định nút thắt cổ chai một cách chính xác.

- Debug và theo dõi chương trình: đồ thị lời gọi động theo dõi trình tự các lời gọi dẫn tới lỗi chương trình một cách chính xác, giúp dễ dàng hơn trong việc tìm ra và xác định lỗi trong một hệ thống phức tạp.

- Phân tích kiểm thử: bằng cách tạo ra một đồ thị lời gọi động với test case có sẵn rồi so sánh với đồ thị lời gọi có thể có, ta có thể chỉnh sửa các thông số của test case để bảo đảm quá trình kiểm thử không bỏ sót các lời gọi có thể sinh ra lỗi.

- Các lợi ích và hạn chế

Khía cạnh	Lợi ích	Hạn chế
Độ chính xác	Không có lời gọi giả nào. Đảm bảo tính đúng đắn cho lần chạy được ghi lại	Có thể bỏ sót những đoạn không được thực thi, độ bao phủ dựa vào test case.
Phân tích sâu vào chương trình	Thống kê chi tiết số lần gọi, các mốc thời gian,	Khối lượng dữ liệu lớn có thể làm quá tải quá trình phân tích
Chi phí	-	Làm tăng thời gian thực thi chương trình và dung lượng bộ nhớ sử dụng
Tính thực tế	Các hành vi thực tế được ghi lại trong môi trường thực tế	Tốn công sức cài đặt công cụ ghi lại hành vi

3. Các thành phần của Call Graph

- Đồ thị lời gọi là một đồ thị có hướng G = (V, E) trong đó các đỉnh v € V biểu diễn các chương trình con là các thủ tục(hàm hay phương thức) và các cạnh (u, v) € E biểu thị chương trình con u có thể gọi chương trình con v trong quá trình thực thi. Các đỉnh và các cạnh có thể bao gồm các thuộc tính như số lần gọi, thông tin về thời gian hoặc thông tin về nơi gọi.

3.1 Đỉnh

- Mỗi đỉnh biểu diễn tương ứng cho một hàm hay phương thức có định danh duy nhất.

- Các thuộc tính:

+Định danh: tên hàm, đặc điểm cụ thể của phương thức, vị trí chương trình.

+Mô-đun, lớp: khi là một phương thức của một lớp sẽ bao gồm thêm thông tin về lớp đó.

+ Số liệu thống kê: Các số liệu thống kê tổng hợp như tổng thời gian tự động, số lần gọi hoặc siêu dữ liệu vị trí nguồn có thể được lưu trữ trên các đỉnh.

3.2 Cạnh

- Các cạnh có hướng từ nơi gọi đến nơi được gọi nhằm biểu diễn luồng điều khiển.

-Tính đa dạng: Một cạnh đơn có thể biểu diễn nhiều lệnh gọi, có thể chú thích bằng:

+Số lần gọi xảy ra trong quá trình chạy.

+Tổng thời gian: Tổng thời gian dành cho các lời gọi .

+Danh sách nơi gọi: Các con trỏ lệnh hoặc số thứ tự của lệnh nơi lời gọi được thực hiện.

- Các thuộc tính:

+Trọng số: Thường được sử dụng như một phương pháp chung để mã hóa tần suất gọi, chi phí hoặc xác suất.

+Loại: nhãn cho lời gọi đệ quy hoặc các lời gọi từ thư viện bên ngoài.

4. Phân tích nội thủ tục và phân tích liên thủ tục trong xây dựng và phân tích đồ thị lời gọi.

4.1 Phân tích nội thủ tục

Phân tích nội thủ tục chỉ xem xét trong phạm vi một hàm đơn lẻ, sử dụng thông tin cục bộ để tính toán luồng điều khiển và dữ liệu bên trong hàm đó. Nó không xem xét các hàm khác hoặc cách các hàm tương tác với nhau.

Vì chỉ tập trung vào một hàm, phân tích nội thủ tục thường đơn giản và chi phí tính toán thấp vì không cần theo dõi lời gọi ngoài hàm

Do không xem xét các lời gọi hàm, nó có thể bỏ lỡ thông tin quan trọng về luồng điều khiển và sự phụ thuộc dữ liệu giữa các hàm.

Ứng dụng trong xây dựng đồ thị lời gọi:

+ Xác định các lời gọi hàm trực tiếp: Phân tích nội thủ tục có thể dễ dàng xác định các lời gọi hàm trực tiếp trong một hàm. Ví dụ nếu hàm foo() gọi hàm bar(), phân tích nội thủ tục trong hàm foo() sẽ phát hiện ra lời gọi này.

+ Xây dựng một phần của đồ thị lời gọi: Nó có thể xây dựng một phần của đồ thị lời gọi, chỉ bao gồm các cạnh là các lời gọi nối các hàm được gọi trực tiếp từ các hàm cụ thể.

4.2. Phân tích liên thủ tục

- Phân tích liên thủ tục so với phân tích nội thủ tục mở rộng phạm vi ra toàn bộ chương trình, xem xét các mối quan hệ giữa các hàm, bao gồm lời gọi, trả về và truyền tham số, giúp xây dựng và tinh chỉnh đồ thị lời gọi chính xác hơn nhưng đồng thời phức tạp và tốn kém về mặt tính toán hơn.

-Phân tích liên thủ tục có độ chính xác cao hơn vì nó có thể thu thập thông tin chính xác hơn về luồng điều khiển và sự phụ thuộc dữ liệu, đặc biệt là trong các chương trình phức tạp với nhiều lời gọi hàm. Nhưng việc triển khai thường phức tạp hơn so với phân tích nội thủ tục.

- Ứng dụng trong xây dựng đồ thị lời gọi:

+ Xác định các lời gọi hàm gián tiếp (dynamic calls): Phân tích liên thủ tục có thể giúp xác định các lời gọi hàm gián tiếp, chẳng hạn như các lời gọi thông qua con trỏ hàm hoặc các lời gọi đa hình trong ngôn ngữ hướng đối tượng.

+ Xây dựng đồ thị lời gọi hoàn chỉnh hơn: Nó có thể xây dựng một đồ thị lời gọi hoàn chỉnh hơn, bao gồm cả các cạnh nối các hàm được gọi trực tiếp và gián tiếp.

+ Phân tích luồng dữ liệu liên thủ tục: Nó cho phép phân tích luồng dữ liệu giữa các hàm, giúp phát hiện các lỗi như truyền tham số không hợp lệ hoặc sử dụng biến chưa được khởi tạo.

B. Các loại lỗi (Bug)

1. Crashing và Non-crashing

- Crashing bug:

+ Là lỗi khiến ứng dụng hoặc hệ thống thoát đột ngột hoặc treo cứng khi chạy thường kèm theo thông báo lỗi hoặc không có thông báo nào.

+ Crashing bug thường được coi là nghiêm trọng vì nó làm gián đoạn trải nghiệm người dùng và có thể gây ra các vấn đề lớn hơn như mất dữ liệu hoặc hỏng hệ điều hành.

+ Nguyên nhân phổ biến: truy cập vùng nhớ không hợp lệ, stack overflow, lỗi con trỏ, ngoại lệ không được xử lý, lỗi chia cho 0, race condition,…

- Non-crashing bug:

+ Non-crashing bug là loại lỗi không khiến chương trình dừng hoạt động hoàn toàn, nhưng chương trình có thể thực hiện một chức năng không chính xác hoặc hiển thị kết quả sai, hay gặp các vấn đề về giao diện người dùng, chẳng hạn như hiển thị sai lệch, nút không hoạt động hoặc bố cục bị hỏng, hay chương trình có thể chạy chậm hoặc tiêu tốn nhiều tài nguyên hơn mức cần thiết.

+ Nguyên nhân phổ biến: Lỗi trong logic của chương trình, lỗi trong các phép tính toán dẫn đến kết quả không chính xác, lỗi khi đọc hoặc ghi dữ liệu từ/đến tệp hoặc thiết bị ngoại vi hoặc cấu hình không chính xác của chương trình.

+ Non-crashing bug thường được coi là ít nghiêm trọng hơn crashing bug, nhưng chúng vẫn có thể gây ra phiền toái và ảnh hưởng đến trải nghiệm người dùng.

2. Occasional và Non-occasional

- Occasional bug:

+ Occasional bug là loại lỗi chỉ xảy ra trong một số trường hợp cụ thể, khó tái tạo mỗi khi thực thi và thường do nhiều yếu tố ngẫu nhiên hoặc môi trường tác động. Chúng thường khó gỡ lỗi vì không dễ dàng để quan sát và phân tích chúng.

+ Lỗi có thể phụ thuộc vào sự kết hợp phức tạp của các yếu tố, chẳng hạn như trạng thái hệ thống, thời gian, dữ liệu đầu vào cụ thể, hoặc thậm chí là sự tương tác với các phần mềm khác.

+ Nguyên nhân phổ biến: vấn đề vùng tranh chấp trong lập trình đa luồng, lỗi bộ nhớ, lỗi trong xử lý ngoại lệ, lỗi trong tương tác với hệ điều hành hoặc phần cứng, lỗi trong quản lý tài nguyên

Ví dụ: Một trang web hiển thị sai thông tin nhưng khi tải lại trang thì thông tin hiển thị đúng trở lại

- Non-occasional bug:

+ Là loại lỗi luôn xuất hiện dưới cùng một thao tác hay điều kiện nhất định, giúp việc phát hiện và sửa lỗi trở nên dễ dàng hơn.

+ Dễ gỡ lỗi hơn do tính nhất quán, dễ dàng kiểm tra do có thể viết các test case để kiểm tra và xác nhận việc sửa lỗi và thường liên quan đến lỗi logic hoặc lỗi cú pháp.

+ Nguyên nhân phổ biến: lỗi logic, lỗi cú pháp, lỗi tính toán, lỗi nhập/xuất, lỗi cấu hình, thiếu kiểm tra đầu vào,…

Ví dụ: Một trang web luôn hiển thị sai thông tin sản phẩm khi người dùng truy cập một trang sản phẩm cụ thể.

3. Structure-affecting bugs

- Lỗi phần mềm không chỉ gây ra hành vi sai hoặc sập ứng dụng, mà còn có thể làm suy thoái cấu trúc chương trình: tăng coupling, giảm cohesion, làm phình to độ phức tạp và phá vỡ các ranh giới kiến trúc. Những "lỗi cấu trúc" này dẫn đến khó bảo trì, dễ xuất hiện lỗi lan truyền và tăng chi phí phát triển dài hạn.

- Lỗi làm tăng coupling: Coupling là mức độ phụ thuộc giữa các module hoặc lớp. Lỗi như sử dụng biến toàn cục, truyền tham số không cần thiết hay vi phạm nguyên tắc phụ thuộc ngược (Dependency Inversion) làm tăng coupling, khiến các module "biết" quá nhiều về nhau.

- Cohesion đo độ gắn kết nội tại của một module. Lỗi đưa nhiều chức năng không liên quan vào cùng một module hoặc lớp xử lý có quá nhiều trách nhiệm dẫn đến cohesion thấp, khó hiểu.

- Lỗi phá vỡ kiến trúc: Vi phạm ranh giới lớp, bỏ qua các pattern kiến trúc (như Hexagonal, Layered) hoặc tạo dependency cycles sẽ phá vỡ tính mô-đun và kiểm soát phạm vi ảnh hưởng khi có lỗi mới.

4. Call Frequency-Affecting Bugs

- Lỗi ảnh hưởng tần suất gọi không thay đổi cấu trúc đồ thị lời gọi (không thêm/bớt cạnh), nhưng làm thay đổi số lần một thủ tục (hàm) được gọi so với kỳ vọng. Những lỗi này thường khó phát hiện bằng phân tích cấu trúc đơn thuần vì đường đi lời gọi hàm vẫn giống nhau, chỉ có tần suất gọi thay đổi.

- Lỗi này ảnh hưởng đến hiệu năng khi tăng tần suất gọi có thể gây giảm hiệu năng nghiêm trọng, hoặc ngược lại bỏ sót xử lý khi gọi quá ít. Làm giảm độ tin cậy khi dữ liệu không được xử lý đúng số lần, dẫn đến mất mát hoặc trùng lặp kết quả.

- Lỗi ảnh hưởng tần suất không làm thay đổi cấu trúc đồ thị nên lập trình viên dễ dàng bỏ qua nguyên nhân gây lỗi nằm ở tần suất gọi hàm chứ không phải ở cấu trúc.

C. Bất biến (Invariant) và suy luận bất biến (Invariant Inference)

- Bất biến (invariant) là một điều kiện hoặc mệnh đề luôn đúng tại một vị trí cụ thể trong chương trình, bất kể con đường thực thi nào dẫn đến nó. Chúng giúp lập trình viên xác định các giả định mà chương trình phụ thuộc vào, từ đó bảo vệ họ khỏi việc vô tình vi phạm các giả định này khi thực hiện thay đổi.

- Suy luận bất biến là quá trình tự động xác định các bất biến của chương trình. Kỹ thuật này nhằm tìm ra các bất biến có khả năng đúng từ chương trình đó.

1. Các bất biến khả năng

- Bất biến khả năng là những tính chất của chương trình mà được suy luận là đúng dựa trên quan sát hành vi của chương trình trong quá trình thực thi. Chúng phản ánh những mối quan hệ hoặc điều kiện "gần như luôn đúng" trong thực tế thực thi, và thường được dùng để hỗ trợ kiểm thử, gỡ lỗi, hay tăng cường tính an toàn của chương trình.

- Các quy tắc hoặc mẫu hành vi thường xuyên xuất hiện trong đồ thị lời gọi (Likely invariants in Call Graph):

+ Chuỗi lời gọi cố định: Mẫu A() -> B() -> C() xuất hiện hầu hết trong các lần chạy, dùng để suy ra luồng nghiệp vụ

+ Bất biến thứ tự: Nếu B được gọi, thì A phải được gọi trước đó trong cùng ngữ cảnh.

+ Bất biến tần suất: Tỉ lệ số lần gọi hàm X/n số lần gọi hàm Y nằm trong một ngưỡng ổn định (ví dụ $X \approx 2Y$)

+ Bất biến vắng mặt: Hàm Z không bao giờ có cạnh hướng tới từ các nút error‑handler.

+ Quan hệ tham số-lời gọi: Giá trị trả về của getKey(a) luôn là tham số đầu vào của updateRecord(…) trong cùng một chuỗi lời gọi.

2. Các phương pháp suy luận bất biến khả năng từ đồ thị lời gọi

2.1 Thống kê và kinh nghiệm

- Sử dụng các đặc điểm thống kê của đồ thị lời gọi để suy luận các bất biến.

- Phương pháp:

    + Nếu một hàm A gọi hàm B rất thường xuyên, có khả năng cao là một số điều kiện nhất định luôn đúng khi A gọi B. Ví dụ, một tham số của A có thể luôn có một giá trị cụ thể khi gọi B.

    + Nếu một hàm A luôn được gọi ở một độ sâu nhất định trong đồ thị lời gọi, có khả năng cao là một số bất biến liên quan đến ngăn xếp (stack) là đúng.

    + Các mẫu cấu trúc trong đồ thị lời gọi (ví dụ: các thành phần liên thông mạnh, các cây con) có thể gợi ý các bất biến. Ví dụ, nếu một nhóm hàm luôn được gọi cùng nhau, có khả năng cao là chúng chia sẻ một số trạng thái hoặc điều kiện chung.

Ưu điểm: Đơn giản, nhanh chóng, và có thể áp dụng cho các chương trình lớn.

Nhược điểm: Không đảm bảo tính chính xác, chỉ cung cấp các gợi ý về các bất biến có khả năng.

2.2 Kết hợp với phân tích dữ liệu thống kê

- Thu thập dữ liệu về hành vi của chương trình trong quá trình thực thi (ví dụ: giá trị của các biến, tần suất các nhánh được thực hiện) và sử dụng thông tin này để suy luận các bất biến.

- Phương pháp:

    + Dynamic Invariant Detection (DID): Chạy chương trình với nhiều bộ dữ liệu đầu vào khác nhau và theo dõi giá trị của các biến. Sử dụng các thuật toán thống kê để tìm ra các mối quan hệ giữa các biến mà luôn đúng (hoặc gần như luôn đúng) trong quá trình thực thi.

    + Kết hợp DID với Đồ thị Lời gọi: Sử dụng đồ thị lời gọi để hướng dẫn quá trình DID. Ví dụ, chỉ theo dõi các biến trong các hàm mà được gọi thường xuyên từ một hàm cụ thể.

Ưu điểm: Cung cấp bằng chứng thực nghiệm để hỗ trợ các bất biến được suy luận.

Nhược điểm: Phụ thuộc vào chất lượng và độ bao phủ của bộ dữ liệu đầu vào.

2.3 Sử dụng Học Máy (Machine Learning)

- Huấn luyện một mô hình học máy để dự đoán các bất biến dựa trên các đặc điểm của đồ thị lời gọi và các thông tin khác.

- Phương pháp:

    + Feature Engineering: Chọn các đặc điểm phù hợp để biểu diễn đồ thị lời gọi và các thông tin khác (ví dụ: tần suất gọi, độ sâu gọi, kiểu dữ liệu, tên biến).

    + Model Training: Huấn luyện một mô hình học máy (ví dụ: cây quyết định, mạng nơ-ron) để dự đoán các bất biến.

    + Model Evaluation: Đánh giá hiệu suất của mô hình trên một bộ dữ liệu kiểm tra.

Ưu điểm: Có thể học các mối quan hệ phức tạp giữa các đặc điểm và các bất biến.

Nhược điểm: Yêu cầu một lượng lớn dữ liệu để huấn luyện, và hiệu suất của mô hình phụ thuộc vào chất lượng của dữ liệu dùng để huấn luyện và các đặc điểm được chọn.

2.4 Phân tích dựa trên mẫu đồ thị con

Tìm kiếm các mẫu đồ thị con phổ biến trong đồ thị lời gọi và các thông tin khác, và sử dụng các mẫu đồ thị con này để suy luận các bất biến.

Phương pháp:

    + Tìm kiếm mẫu đồ thị con: Sử dụng các thuật toán tìm kiếm mẫu đồ thị con để tìm kiếm các mẫu phổ biến trong đồ thị lời gọi và các thông tin khác.

    + Ánh xạ mẫu đồ thị con sang bất biến khả năng: Định nghĩa các quy tắc ánh xạ để ánh xạ các mẫu sang các bất biến khả năng.

Ưu điểm: Có thể tận dụng kiến thức về các mẫu đồ thị con phổ biến để suy luận các bất biến.

Nhược điểm: Phụ thuộc vào việc định nghĩa các mẫu đồ thị con và quy tắc ánh xạ phù hợp.

Nội dung của Thảo

I. Giới thiệu (Thảo)

A. Tổng quan:

1. **Tầm quan trọng của việc phát hiện lỗi phần mềm sớm.**

- Tiết kiệm chi phí:
+ Chi phí sửa lỗi tăng theo thời gian: Nếu lỗi được phát hiện ở giai đoạn thiết kế hoặc lập kế hoạch, chi phí sửa chữa thường rất thấp. Tuy nhiên, nếu lỗi chỉ được phát hiện sau khi phần mềm đã triển khai hoặc đến tay người dùng, chi phí khắc phục có thể tăng gấp 10 - 100 lần.
+ Ví dụ: Một lỗi sai ban đầu có thể gây ra hàng loạt lỗi trong thiết kế, code và kiểm thử sau này.
- Tiết kiệm thời gian phát triển:
+ Khi lỗi được phát hiện sớm, việc sửa chữa sẽ nhanh và ít ảnh hưởng hơn đến các phần khác của hệ thống.
+ Tránh việc phải quay lại sửa lại toàn bộ hệ thống khi lỗi phát hiện muộn.
- Nâng cao chất lượng phần mềm:
+ Phát hiện sớm lỗi giúp đảm bảo phần mềm đúng yêu cầu, đúng chức năng, hoạt động ổn định và ít bug hơn khi đến tay người dùng.
+ Giảm rủi ro lỗi nghiêm trọng xảy ra sau khi phần mềm được triển khai.
- Tăng sự hài lòng của khách hàng:
+ Phần mềm ổn định, ít lỗi ngay từ đầu giúp khách hàng tin tưởng và hài lòng hơn.
+ Tránh được các trường hợp mất uy tin hoặc mất khách hàng vì phần mềm lỗi hoặc sập hệ thống.

2. **Giới thiệu về đồ thị lời gọi (Call graph) trong phát hiện cấu trúc và hành vi chương trình.**

- Đồ thị lời gọi (Call graph) là một mô hình đồ thị biểu diễn mối quan hệ giữa các hàm hoặc thủ tục trong một chương trình. Trong đồ thị này:
    + Mỗi đỉnh (node) biểu diễn một hàm hoặc thủ tục.
    + Mỗi cạnh (edge) biểu diễn một lời gọi từ một hàm này đến một hàm khác.
- Vai trò đồ thị lời gọi trong phát hiện cấu trúc và hành vi chương trình:
    + Hiểu cấu trúc chương trình:
        - Call graph giúp ta hình dung cấu trúc điều khiển tổng thể của chương trình.
        - Biết được hàm nào gọi hàm nào, từ đó xác định mối quan hệ phụ thuộc giữa các thành phần.
        - Giúp xác định hàm gốc (entry point) và các hàm phụ thuộc theo chiều sâu.
    + Phân tích hành vi chương trình:
        - Giúp xác định luồng thực thi, theo dõi dòng dữ liệu và xác định các điểm ảnh hưởng qua lại giữa các hàm.
        - Phục vụ cho việc phân tích hành vi tĩnh (static analysis) mà không cần thực thi chương trình.
    + Phát hiện lỗi và lỗ hổng bảo mật:
        - Dễ phát hiện các hàm không bao giờ được gọi (dead code).
        - Xác định vùng mã nguy hiểm (ví dụ: hàm gọi đến API hệ thống hoặc hàm thao tác với dữ liệu nhạy cảm)
        - Phát hiện vòng lặp đệ quy không kiểm soát hoặc lời gọi hàm bất thường.
    + Hỗ trợ tối ưu hóa và bảo trì:
        - Giúp lập trình viên dễ dàng theo dõi tác động khi chỉnh sửa một hàm (gọi là impact analysis).
        - Tối ưu hóa bộ nhớ và tốc độ bằng cách xác định các hàm thường xuyên được gọi hoặc gọi không cần thiết.

3. **Giới thiệu về suy luận bất biến - invariant trong khả năng phát hiện lỗi.**

- Bất biến (invariant) là một điều kiện hoặc mệnh đề luôn đúng tại một vị trí cụ thể trong chương trình, bất kể con đường thực thi nào dẫn đến nó.

- Suy luận bất biến là quá trình tự động hoặc bán tự động để tìm ra các bất biến tiềm năng trong chương trình bằng cách phân tích mã nguồn hoặc theo dõi hành vi thực thi.
- Vai trò của suy luận bất biến trong phát hiện lỗi:
    + Phát hiện lỗi logic và vi phạm ràng buộc:
        - Nếu trong quá trình thực thi, một bất biến không còn đúng, điều đó cho thấy có lỗi sai lệch về kiểm soát luồng hoạt động.
        - Ví dụ: Nếu bất biến yêu cầu 1 cặp hàm phải đi cùng nhau như malloc() và free(), nhưng thực tế chương trình lại có trường hợp không thỏa mãn cả 2 cùng xuất hiện, có thể gây lỗi rò rỉ bộ nhớ.
    + Hỗ trợ kiểm chứng chương trình (program verification):
        - Trong các hệ thống kiểm chứng hình thức, bất biến là công cụ cốt lõi để chứng minh tính đúng đắn của chương trình.
        - Giúp chứng minh điều kiện trước (precondition) và điều kiện sau (postcondition) luôn được đảm bảo.
    + Phòng chống lỗi bảo mật:
        - Suy luận bất biến có thể chỉ ra các vùng mã có hành vi bất thường hoặc nguy hiểm, như thay đổi giá trị biến không mong muốn, vi phạm giới hạn quyền truy cập bộ nhớ,...
    + Phân tích động (Dynamic Analysis):
        - Dựa vào dữ liệu thu thập khi chạy chương trình, các công cụ có thể suy ra bất biến động, hỗ trợ phát hiện lỗi mà kiểm thử thông thường có thể bỏ sót.

4. Mục tiêu và phạm vi của báo cáo/phân tích:

- Mục tiêu của báo cáo:
    + Làm rõ tầm quan trọng của việc phát hiện lỗi phần mềm sớm, đặc biệt là các lỗi khó phát hiện như lỗi không gây crash (non-crashing bugs), nhằm nâng cao độ tin cậy và chất lượng phần mềm.
    + Tìm hiểu cấu trúc và vai trò của đồ thị lời gọi (Call graph) và vai trò của đồ thị lời gọi trong biểu diễn hành vi chương trình, qua đó xác định mối liên hệ giữa cấu trúc đồ thị và các lỗi phần mềm tiềm ẩn.
    + Khảo sát và trình bày các kỹ thuật suy luận bất biến khả năng (invariants) trên đồ thị lời gọi (Call graph), như các mẫu hành vi thường xuyên xuất hiện hoặc các ràng buộc cấu trúc và tần suất.
    + Phân tích các phương pháp tối giản đồ thị lời gọi (Call graph) nhằm cải thiện hiệu quả tính toán, đồng thời bảo toàn các đặc trưng quan trọng phục vụ việc suy luận và phát hiện lỗi.
    + Mô tả quy trình phát hiện lỗi phần mềm dựa trên kết hợp giữa đồ thị và bất biến, bao gồm xây dựng mô hình, lựa chọn đặc trưng, và sử dụng các kỹ thuật học máy để phân loại lỗi.
    + Đánh giá thực nghiệm hiệu quả của các kỹ thuật đề xuất thông qua bộ dữ liệu chương trình thực tế, dựa trên các tiêu chí: độ chính xác, độ bao phủ, thời gian thực thi và khả năng tổng quát hóa.
- Phạm vi của báo cáo:
    + Phân tích ở mức độ hàm (function-level), tập trung vào quan hệ lời gọi giữa các phương thức trong chương trình.
    + Sử dụng kết hợp phân tích tĩnh và động để xây dựng đồ thị lời gọi, với trọng tâm là Call Graph động nhằm phản ánh chính xác hành vi thực thi thực tế.
    + Tập trung vào các lỗi hành vi (behavioral bugs), đặc biệt là những lỗi ảnh hưởng đến cấu trúc hoặc tần suất lời gọi, thay vì lỗi cú pháp hay lỗi biên dịch.
    + Giới hạn trong các phương pháp mô hình hóa và phân tích đồ thị, không mở rộng sang kiểm thử tự động.
    + Không đề xuất công cụ hoàn chỉnh, nhưng có thể đưa ra gợi ý về hướng phát triển hoặc cải tiến khung phân tích hiện có.

B. Vấn đề và động lực:

1. **Thách thức trong việc tìm kiếm và sửa lỗi, đặc biệt là các lỗi không gây crash (non-crashing bugs).**

- Không có dấu hiệu rõ ràng: Không giống như các lỗi gây crash thường dẫn đến lỗi hệ thống, dừng chương trình hoặc sinh thông báo lỗi, các lỗi không gây crash thường âm thầm gây ra hành vi sai lệch (như tính toán sai, trạng thái sai, kết quả đầu ra không đúng), khiến chúng khó bị phát hiện trong quá trình kiểm thử thông thường.
- Khó tái hiện trong môi trường thử nghiệm: Các lỗi này có thể chỉ xảy ra trong một số điều kiện đầu vào hiếm gặp, hoặc chỉ xuất hiện sau một chuỗi thao tác cụ thể, khiến việc tái tạo lỗi trở nên khó khăn và tốn thời gian.
- Không có stack trace hoặc thông tin gỡ lỗi hữu ích: Do chương trình vẫn tiếp tục chạy bình thường, nên không có nhật ký lỗi (crash log) hoặc dấu vết ngăn xếp (stack trace) để hỗ trợ quá trình truy vết và sửa lỗi.
- Tăng độ phức tạp trong phân tích nguyên nhân gốc (root cause analysis)
- Thường bị bỏ sót trong kiểm thử tự động.

2. **Sự cần thiết của các kỹ thuật tự động hoặc bán tự động để hỗ trợ quá trình gỡ lỗi.**

- Tăng quy mô và độ phức tạp của hệ thống phần mềm.
- Hạn chế của năng lực con người
- Khó phát hiện các lỗi hành vi phi trực quan.
- Giảm chi phí và thời gian phát triển phần mềm.
- Tăng độ tin cậy và khả năng tái hiện.
- Tạo nền tảng cho học máy và phát hiện bất thường nâng cao

3. **Ưu điểm của việc sử dụng suy luận bất biến lên đồ thị lời gọi so với các phương pháp truyền thống.**

- Khả năng học được hành vi "bình thường" của chương trình:
    + Các bất biến khả năng (likely invariants) được suy luận từ nhiều lần chạy của chương trình (nhiều input khác nhau) cho phép mô hình hóa mẫu hình vi phổ biến hoặc kỳ vọng của chương trình.
    + Khi một bất biến bị vi phạm, điều này có thể báo hiệu sự bất thường tiềm ẩn - ngay cả khi chương trình không crash.
- Phát hiện được lỗi không biểu hiện qua đầu ra sai:
    + Không giống các kỹ thuật kiểm thử dựa trên kiểm tra kết quả, phương pháp này phát hiện lỗi dựa trên sự sai lệch hành vi nội tại, ví dụ như 1 lời gọi hàm bất thường, một tần suất gọi thay đổi đáng kể, hay một mẫu con trong Call Graph bị thiếu.
    + Điều này rất hữu ích trong việc phát hiện các lỗi ngầm không ảnh hưởng trực tiếp đến output nhưng làm suy giảm tính đúng đắn về mặt logic của hệ thống.
- Phân tích dựa trên cấu trúc và thống kê hành vi:
    + Bằng cách biểu diễn chương trình dưới dạng Call graph và gắn trọng số (call frequency), ta có thể áp dụng các kỹ thuật phân tích đồ thị và khai phá mẫu.
    + Từ đó dễ dàng phát hiện các bất thường về cấu trúc (như thiếu lời gọi) và tần suất (như lời gọi đột biến), điều mà các kỹ thuật phân tích tĩnh thông thường không phát hiện được.
- Khả năng mở rộng và tự động hóa cao.
    + Sau khi thu thập dữ liệu thực thi và xây dựng Call Graph, quá trình suy luận bất biến có thể được tự động hóa hoàn toàn với các thuật toán thống kê, học máy hoặc khai phá mẫu đồ thị.
    + Đây là lợi thế lớn khi áp dụng cho các hệ thống lớn, có hàng nghìn hàm và hàng triệu lời gọi.

C. Đóng góp: (Nếu có) Đề xuất kỹ thuật mới, cải tiến hiện có hoặc khung phân tích mới.

III. Các phương pháp tối giản đồ thị lời gọi (Thảo)

A. Tổng quan:

1. **Sự cần thiết của việc giảm đồ thị lời gọi để cải thiện hiệu suất của các thuật toán phân tích và suy luận bất biến (invariant).**

- Tối ưu hóa hiệu suất xử lý:
    + Kích thước đồ thị (số lượng node và cạnh) tỉ lệ thuận với chi phí tính toán của các thuật toán phân tích.
    + Việc giữ nguyên toàn bộ chi tiết sẽ dẫn đến bùng nổ độ phức tạp, ảnh hưởng đến cả thời gian chạy, bộ nhớ tiêu thụ, và độ trễ phản hồi của hệ thống.
    + Giảm đồ thị giúp giảm đáng kể chi phí phân tích, đặc biệt khi cần xử lý nhiều tập dữ liệu hoặc nhiều phiên thực thi chương trình.
- Loại bỏ nhiễu và thông tin dư thừa:
    + Trong thực tế, nhiều node hoặc cạnh trong Call Graph chỉ đóng vai trò phụ trợ, lặp đi lặp lại hoặc không mang thông tin phân biệt quan trọng cho suy luận bất biến.
    + Việc giữ lại các thành phần này không những không tăng thông tin giá trị, mà còn gây nhiễu thống kê, dẫn đến việc học sai hoặc đưa ra bất biến không chính xác.
- Làm nổi bật các mẫu hành vi chính:
    + Việc rút gọn đồ thị giúp trừu tượng hóa hành vi chương trình theo cách dễ phát hiện các pattern bất thường (vi phạm bất biến).
    + Ví dụ, khi tổng hợp nhiều lời gọi giống nhau về một node duy nhất và thêm trọng số cạnh, các bất thường về tần suất trở nên dễ quan sát hơn.
    + Từ đó, việc khai phá các mẫu bất biến khả năng (likely invariants) trở nên rõ ràng và đáng tin cậy hơn.
- Tăng khả năng tổng quát và tái sử dụng:
    + Đồ thị sau khi được giảm có tính khái quát cao hơn, từ đó các biến suy luận được ít phụ thuộc từng dữ liệu cụ thể, dễ áp dụng cho các input khác hoặc hệ thống tương tự.

2. **Các phương pháp tối giản đồ thị lời gọi khác nhau và ưu nhược điểm của chúng.**

- Phương pháp Total Reduction:
    + Mô tả: Gộp tất cả các node (đỉnh) đại diện cho cùng một hàm/phương thức trong toàn bộ đồ thị thành 1 node duy nhất. Mỗi cạnh chỉ biểu diễn sự tồn tại của mối quan hệ gọi hàm, không mang thông tin về tần suất.
    + Ưu điểm:
        ● Rất hiệu quả trong giảm số lượng node và cạnh.
        ● Đơn giản, dễ thực hiện.
        ● Phù hợp cho các phân tích cấu trúc cơ bản.
    + Nhược điểm:
        ● Mất thông tin về tần suất gọi - không thể phát hiện các lỗi liên quan đến hành vi bất thường về call frequency.
        ● Có thể làm mờ nhạt hành vi thực thi thực tế, gây khó khăn cho các bất biến liên quan đến tần suất
- Phương pháp Total Reduction with Edge Weights:
    + Mô tả: Giống Total Reduction, nhưng mỗi cạnh trong đồ thị được gắn thêm trọng số đại diện cho tần số gọi giữa hai hàm trong nhiều lần chạy.
    + Ưu điểm:
        ● Giữ lại được thông tin hành vi động (frequency), cho phép phát hiện lỗi ảnh hưởng tần suất gọi.
        ● Cân bằng tốt giữa giảm độ phức tạp và giữ lại thông tin quan trọng.
    + Nhược điểm:
        ● Vẫn có thể làm mất thông tin vị trí cụ thể trong flow gọi hàm (do gộp node).

        - Phân tích phức tạp hơn so với Total Reduction vì phải xử lý các số liệu thống kê (trọng số).
- Phương pháp Zero - One - Many (ZOM) Reduction:
    + Mô tả: Thay vì lưu toàn bộ số lần gọi hoặc gộp hoàn toàn, phương pháp này trừu tượng hóa tần suất thành 3 mức:
        - 0: không gọi.
        - 1: gọi một lần duy nhất.
        - Many: gọi nhiều hơn một lần.
    + Ưu điểm:
        - Đơn giản hóa biểu diễn nhưng vẫn giữ được ý nghĩa hành vi tổng quát.
        - Giảm nhiễu do dao động nhỏ trong tần suất gọi.
        - Tốt cho việc suy luận bất biến dạng định tính (qualitative invariants).
    + Nhược điểm:
        - Mất đi thông tin định lượng chi tiết.
        - Có thể bỏ sót các bất thường nhẹ về tần suất.

B. Các kỹ thuật giảm đồ thị lời gọi:

1. **Total Reduction: Gộp tất cả các node đại diện cho cùng một phương thức.**

- **Total Reduction** là một phương pháp giảm đồ thị lời gọi với mục tiêu rút gọn kích thước đồ thị bằng cách:
    - Mỗi hàm (function/method) chỉ xuất hiện một lần duy nhất trong đồ thị.
    - Một cạnh (edge) sẽ được nối giữa hai node nếu có lời gọi giữa hai hàm tương ứng.
- Đặc điểm chính của **Total Reduction:**
    - Không phân biệt ngữ cảnh hoặc số lần gọi hàm.
    - Tập trung vào cấu trúc quan hệ gọi giữa các hàm, không lưu thông tin động như tần suất hay vị trí gọi.
- Cách thực hiện của **Total Reduction:**
    - Tất cả các node đại diện cho cùng một hàm trong các lần thực thi khác nhau sẽ được gộp lại thành một node duy nhất.
    - Nếu f () được gọi từ nhiều vị trí khác nhau thì trong đồ thị rút gọn, chỉ còn một node f và một số cạnh tương ứng nối đến hoặc từ f.
- Ưu điểm của **Total Reduction:**
    - Giảm đáng kể kích thước đồ thị: Như trong tài liệu D-24, đồ thị gốc có 22 node và 21 cạnh, sau khi áp dụng **Total Reduction** thì còn 6 node và 6 cạnh.
    - Tăng hiệu quả lưu trữ và phân tích: Kích thước nhỏ hơn dẫn đến tiết kiệm bộ nhớ và thời gian tính toán.
- Hạn chế nghiêm trọng của **Total Reduction:**

Vấn đề	Mô tả
Làm mất thông tin thực thi	Không giữ được tần suất gọi, đường dẫn cụ thể, hay các mẫu cấu trúc chi tiết của lời gọi.
Làm thay đổi cấu trúc gốc	Việc hợp nhất node khiến đồ thị mất đi hình dạng ban đầu, khó truy vết hành vi thực tế.
Khó truy vấn hoặc khai thác mẫu	Không thể áp dụng các kỹ thuật khai phá mẫu (pattern mining) vì thông tin đã bị trừu tượng hóa quá mức

- Kết luận chung về **Total Reduction:**

  Total Reduction là **một kỹ thuật đơn giản, hiệu quả cao về mặt giảm kích thước, nhưng có tính trừu tượng hóa quá mức.** Phương pháp này chỉ phù hợp khi:

  - Cần xử lý lượng lớn dữ liệu và không yêu cầu thông tin chi tiết.
  - Mục tiêu là phân tích cấu trúc tổng quát giữa các hàm, không phân tích hành vi động.

  Tuy nhiên, trong bối cảnh cần phát hiện lỗi phần mềm dựa trên hành vi thực thi, tần suất gọi, hoặc suy luận bất biến, Total Reduction thiếu khả năng giữ lại thông tin quan trọng và không phù hợp để làm cơ sở lý luận sâu.

2. **Total Reduction with Edge Weights: Thêm trọng số vào các cạnh để biểu diễn tần suất gọi.**

- **Total Reduction with Edge Weight** là một kỹ thuật tối giản đồ thị lời gọi (Call graph) mở rộng từ Total Reduction, nhằm giữ lại thông tin về tần suất gọi hàm thông qua việc gán trọng số (weights) cho các cạnh của đồ thị.
- Cách hoạt động của **Total Reduction with Edge Weights:** Trong kỹ thuật này, mỗi hàm vẫn chỉ được đại diện bởi một node duy nhất, giống như Total Reduction, nhưng các cạnh giữa các node sẽ mang trọng số thể hiện số lần lời gọi xảy ra giữa hai hàm trong quá trình thực thi..

3. **Zero-One-Many Reduction: Giảm các cấu trúc con lặp đi lặp lại.**

- **Zero - One - Many Reduction (ZOM) là một** kỹ thuật tối giản đồ thị lời gọi nhằm trừu tượng hóa tần suất lời gọi hàm bằng cách phân loại mỗi cạnh trong đồ thị, nhưng thay vì lưu trữ chính xác số lần gọi như trong Edge Weights, ZOM chỉ quan tâm đến mức độ xuất hiện của lời gọi theo hướng định tính.
- Đây là một phương pháp **định tính**, hữu ích khi xử lý **đồ thị lớn** và **phát hiện lỗi hành vi mang tính mẫu hình**.

C. So sánh các kỹ thuật:

1. **So sánh hiệu quả giảm kích thước của các phương pháp khác nhau.**

Phương pháp rút gọn	Số node	Số cạnh	Ảnh hưởng đến cấu trúc
Mã nguồn ban đầu	22	21	
Total Reduction	6	6	Mất thông tin và thay đổi cấu trúc
Zero - One - Many Reduction	15	14	Mất thông tin nhưng giữ nguyên cấu trúc
Thuật toán được đề xuất trong bài báo	10	0	Không mất thông tin và giữ nguyên cấu trúc

2. **Ảnh hưởng của các kỹ thuật giảm khác nhau đến khả năng phát hiện lỗi.**

   Các kỹ thuật giảm đồ thị lời gọi ảnh hưởng trực tiếp đến khả năng phát hiện lỗi trong chương trình. Total Reduction giúp giảm mạnh kích thước đồ thị nhưng làm mất thông tin tần suất và ngữ cảnh, dẫn đến bỏ sót các lỗi hành vi tinh vi. Zero-One-Many Reduction giữ lại cấu trúc và phân

loại tần suất một cách định tính (0, 1, nhiều), giúp giảm nhiễu nhưng vẫn thiếu độ chính xác trong việc phát hiện các thay đổi bất thường về số lần gọi. Trong khi đó, Total Reduction with Edge Weights là phương pháp cân bằng, vừa giảm được kích thước đồ thị, vừa giữ lại thông tin định lượng về tần suất gọi, giúp phát hiện tốt hơn các lỗi liên quan đến hành vi bất thường. Do đó, tùy mục tiêu phân tích mà lựa chọn kỹ thuật phù hợp để tối ưu giữa hiệu suất và độ chính xác.

VII. Kết luận và hướng phát triển (Thảo)

A. Tóm tắt:

Qua quá trình tìm hiểu và phân tích, báo cáo đã làm rõ vai trò của đồ thị lời gọi (Call Graph) trong việc mô hình hóa cấu trúc và hành vi thực thi của chương trình, từ đó trở thành nền tảng hữu ích cho các kỹ thuật phát hiện lỗi phần mềm. Đặc biệt, khi kết hợp với phương pháp suy luận bất biến (invariant inference), call graph có thể giúp phát hiện hiệu quả nhiều loại lỗi hành vi — bao gồm cả lỗi không gây crash, vốn rất khó phát hiện bằng các phương pháp truyền thống.

Báo cáo đã phân tích ba kỹ thuật giảm đồ thị lời gọi phổ biến: Total Reduction, Total Reduction with Edge Weights, và Zero-One-Many Reduction. Mỗi kỹ thuật đều có ưu và nhược điểm riêng, ảnh hưởng trực tiếp đến khả năng giữ lại thông tin thực thi, mức độ trừu tượng hóa hành vi và hiệu quả phát hiện lỗi. Kết quả cho thấy, Total Reduction with Edge Weights là lựa chọn cân bằng tốt giữa hiệu suất xử lý và khả năng phân tích hành vi động, trong khi Zero-One-Many giúp giảm nhiễu ở mức đơn giản, và Total Reduction chỉ phù hợp trong các phân tích cấu trúc sơ bộ.

Qua so sánh thực nghiệm, phương pháp được đề xuất trong tài liệu tham khảo cũng cho thấy hiệu quả rõ rệt khi vừa giảm kích thước đồ thị, vừa giữ được thông tin cấu trúc và tần suất, từ đó hỗ trợ tốt hơn cho việc khai thác mẫu và phát hiện bất thường.

B. Hướng phát triển:

1. **Các hướng nghiên cứu tiềm năng để cải thiện các kỹ thuật phát hiện lỗi bằng đồ thị lời gọi và suy luận invariant.**

- **Kết hợp giữa phân tích tĩnh và động trên đồ thị lời gọi:**
    - Phân tích tĩnh (từ mã nguồn) giúp xây dựng call graph đầy đủ, còn phân tích động (từ runtime logs) phản ánh hành vi thực thi thực tế.
    - Nghiên cứu hướng **kết hợp hai loại đồ thị** để vừa đảm bảo độ bao phủ, vừa chính xác theo ngữ cảnh thực thi.
    - Điều này giúp phát hiện cả lỗi tiềm ẩn (chưa xảy ra) và lỗi thực tế (đã xảy ra trong runtime).
- **Suy luận bất biến với học máy và thống kê nâng cao:**
    - Thay vì dùng rule - based hoặc thống kê đơn giản, có thể ứng dụng:
        + **Machine learning** để học các mẫu hành vi bình thường trên call graph.
        + **Graph Neural Networks (GNNs)** để khai thác cấu trúc đồ thị tốt hơn.
        + **Bayesian inference hoặc HMMs** để mô hình hóa trình tự lời gọi phức tạp.
    - Giúp nâng cao khả năng phát hiện bất thường, đặc biệt là các lỗi hành vi khó định nghĩa.
- **Ngữ cảnh hóa bất biến (Context-sensitive Invariants):**
    - Các bất biến hiện tại thường không phân biệt ngữ cảnh gọi hàm.
    - Nghiên cứu hướng bất biến có ngữ cảnh sẽ giúp phát hiện các lỗi xảy ra chỉ khi một hàm được gọi từ một ngữ cảnh cụ thể.
- **Tự động hóa quá trình phát hiện và cập nhật bất biến:**
    - Phát triển hệ thống có thể học bất biến tự động từ nhiều lần chạy và cập nhật liên tục khi phần mềm thay đổi.

    - Hữu ích cho các dự án lớn, thường xuyên cập nhật.
    - Kết hợp cùng hệ thống kiểm thử để tự động cảnh báo khi có vi phạm bất biến mới.
- **Tăng cường biểu diễn đồ thị lời gọi:**
    - Đề xuất mở rộng biểu diễn cạnh trong đồ thị:
        + Gắn thêm thông tin như thời gian thực thi, tài nguyên sử dụng, loại lời gọi (đệ quy, bất đồng bộ,...)
    - Giúp phát hiện các lỗi hiệu suất, lỗi rò rỉ tài nguyên hoặc lỗi do cạnh tranh luồng.
- **Áp dụng cho phần mềm lớn và phức tạp:**
    - Nghiên cứu các thuật toán phát hiện lỗi phân tán, song song hoặc theo module để đảm bảo khả năng mở rộng.
    - Áp dụng trong môi trường thực tế như hệ thống microservices, phần mềm hệ thống, phần mềm nhúng.

2. Thách thức và cơ hội trong việc áp dụng cho các dự án phần mềm lớn và phức tạp.

- Cơ hội:

Phát hiện lỗi khó tái hiện và lỗi không gây crash

Trong các hệ thống lớn, nhiều lỗi không gây crash nhưng ảnh hưởng nghiêm trọng đến logic hoặc dữ liệu. Việc phân tích call graph kết hợp với bất biến giúp phát hiện các lỗi hành vi tinh vi mà kiểm thử truyền thống dễ bỏ sót.

Tự động hóa kiểm lỗi trong quy trình CI/CD

Khi tích hợp vào quy trình phát triển phần mềm liên tục (Continuous Integration/Continuous Deployment), kỹ thuật này có thể tự động phân tích, phát hiện vi phạm bất biến, và đưa cảnh báo sớm mà không cần viết thêm test case.

Hỗ trợ bảo trì và phân tích tác động (impact analysis)

Đồ thị lời gọi giúp hiểu rõ quan hệ giữa các hàm → hỗ trợ lập trình viên khi sửa lỗi, nâng cấp mã mà không gây ảnh hưởng ngoài ý muốn.

Mở rộng khả năng phát hiện lỗi vượt ngoài kiểm thử truyền thống

Call Graph + Invariant Inference có thể hoạt động tốt ngay cả khi thiếu tài liệu, thiếu test case hoặc không rõ đầu ra đúng sai — rất hữu ích trong các hệ thống kế thừa (legacy systems).

- Thách thức:

Quy mô đồ thị cực lớn

Các dự án lớn có thể chứa hàng chục ngàn hàm và hàng triệu cạnh lời gọi. Việc xây dựng, lưu trữ và phân tích đồ thị đòi hỏi **tài nguyên tính toán lớn** và kỹ thuật tối ưu dữ liệu.

Đồ thị nhiễu và dư thừa thông tin

Trong môi trường thực tế, call graph có thể chứa nhiều lời gọi phụ trợ (helper), lời gọi thư viện hoặc các lời gọi không quan trọng → gây **nhiễu cho quá trình suy luận bất biến** và phát hiện lỗi.

Khó xác định bất biến đúng

Trong hệ thống phức tạp, hành vi "bình thường" rất đa dạng → khó suy luận được bất biến chính xác, dễ

dẫn đến **false positive** (báo sai) hoặc **false negative** (bỏ sót lỗi).

Tính mở rộng (scalability) và tốc độ phân tích

Việc áp dụng các thuật toán khai thác mẫu, học máy hoặc phân tích hành vi trên đồ thị lớn có thể **mất nhiều thời gian**, khó áp dụng trong các pipeline CI/CD thời gian thực.

Thiếu dữ liệu chạy thực tế (runtime logs)

Các hệ thống mới phát triển hoặc hệ thống thiếu ghi log sẽ **khó xây dựng call graph động đầy đủ**, ảnh hưởng đến chất lượng phân tích.

Nội dung của Sang

Nguồn tài liệu tìm hiểu:

- IE105-D-1-Improved-Software-Fault-Detection-with-Graph-Mining.
- IE105-D-2-An-automatically-created-novel-bug-dataset-and-its-validation-in-bug-prediction.
- IE105-D-3-Tracking-Down-Software-Bugs-Using-Automatic-Anomaly-Detection.
- IE105-D-4-Automatic-Fault-Detection-for-Deep-Learning-Programs-Using-Graph-Transformations.
- IE105-D-5-Mining-Edge-Weighted-Call-Graphs-to-Localise-Software-Bugs.
- IE105-D-6-An-Analysis-of-C-CPP-Datasets-for-Machine-Learning-Assisted-Software-Vulnerability-Detection.
- IE105-D-7-Scalable-and-Incremental-Software-Bug-Detection.

Ý tưởng khái quát:

Việc phát hiện lỗi bằng đồ thị lời gọi - bug detection using call graph về cơ bản có thể xác định như sau:

1. Xác định các lỗi thường gặp trong chương trình (có thể hiểu như tìm kiếm các ngoại lệ - exception).

1.1. Nếu phát hiện được các lỗi này thì có thể kết luận thẳng

2. Xác định các phương thức, lời gọi trong chương trình

3. Hình thành đồ thị lời gọi đầy đủ của chương trình

4. Thực hiện đơn giản hóa đồ thị lời gọi

5. Đánh giá tần suất các lời gọi

6. Trả về danh sách các lời gọi có khả năng xuất hiện lỗi nhất

Khung nội dung làm việc cụ thể:

IV. Quy trình phát hiện lỗi dựa trên đồ thị lời gọi và suy luận bất biến: (Sang)

- **A. Thu thập và tiền xử lý đồ thị lời gọi:**
- **B. Tối giản đồ thị lời gọi.**
- **C. Suy luận các bất biến khả năng**
- **D. Phân tích thống kê để xác định các bất biến vi phạm.**

V. Đánh giá thống kê (Sang):

- **A. Thiết lập thử nghiệm:**
  - Mô tả về bộ dữ liệu thử nghiệm (ví dụ: chương trình C++, số lượng phương thức).
  - Các loại lỗi được sử dụng trong thử nghiệm.

    - Tiêu chí đánh giá (ví dụ: khả năng phát hiện lỗi).
- **B. Kết quả:**
    - So sánh khả năng phát hiện lỗi khi các tham số ngưỡng thay đổi tuyến tính.
- **C. Thảo luận:**
    - Giải thích ý nghĩa của kết quả.
    - So sánh với các nghiên cứu liên quan khác.
    - Hạn chế của phương pháp thử nghiệm.

Chi tiết nội dung:

IV. Quy trình phát hiện lỗi dựa trên đồ thị lời gọi và suy luận bất biến:

Chương này sẽ trình bày chi tiết về quy trình xây dựng và phát hiện lỗi dựa trên đồ thị và suy luận bất biến, tổng quan toàn bộ quy trình phát triển và xây dựng thí nghiệm có thể tìm thấy tại Github repository IE105-P21-Software-bug-detection-using-graph.

1. Thu thập và tiền xử lý đồ thị lời gọi:

Đây là quá trình đầu tiên trong việc phát hiện lỗi, ở bước này, tùy thuộc vào ngôn ngữ lập trình được lựa chọn sẽ có nhiều phương pháp thực hiện khác nhau. Nhóm tác giả thực hiện lựa chọn thí nghiệm trên hai ngôn ngữ C và C++ vì tính đơn giản và được xây dựng sẵn bộ công cụ hỗ trợ opt từ LLVM. Sau khi xác định được ngôn ngữ và bộ công cụ đi kèm, lúc này ta có thể lựa chọn thêm bash script để thực hiện tự động hóa quy trình xây dựng.

2. Tối giản đồ thị lời gọi:

Sau khi đã thực hiện tiền xử lý đồ thị lời gọi, ta sẽ tiếp tục thực hiện tối giản đồ thị lời gọi dựa theo các phương pháp đã đề cập ở phần lý thuyết. Ở đây, nhóm tác giả thực hiện lựa chọn theo phương pháp Total Reduction để đảm bảo tính đơn giản, dễ thực hiện đối với thí nghiệm.

3. Suy luận các bất biến khả năng:

Thực hiện suy luận dựa trên việc xác định các cặp hàm (x,y) và bản thân các hàm x, y bất kỳ để tính độ hỗ trợ - *support* và mức tự tin - *confidence*. Sau khi tính toán các giá trị cần thiết, thực hiện so sánh và giữ lại các cặp hàm thỏa mãn giá trị ngưỡng - *threshold* của độ hỗ trợ và mức tự tin.

4. Phân tích thống kê để xác định các bất biến vi phạm:

Dựa vào các cặp hàm (x,y) thỏa mãn yêu cầu, ta thực hiện duyệt lại đồ thị lời gọi gốc ban đầu và kiểm tra nếu có xuất hiện trường hợp một hàm thành phần y trong chương trình hoặc khu vực thực thi được gọi nhưng lại không đi kèm với hàm thành phần x. Nếu xuất hiện trường hợp này thì trả về lỗi có thể xảy ra tại cặp hàm và vị trí lỗi đó.

V. Đánh giá thống kê (Sang):

Chương này sẽ thực hiện mô tả việc áp dụng quy trình phát hiện lỗi lên bộ dữ liệu mẫu và thực hiện thống kê kết quả trả về.

1. Thiết lập thử nghiệm:

Thử nghiệm này được thực hiện trên cấu hình AMD Ryzen™ 3 4300U, 8GB RAM và nền tảng Fedora Linux 40 với các công cụ hỗ trợ như Github, VS Code. Các ngôn ngữ lập trình được đưa vào sử dụng bao gồm Python, C, C++ và Shell.

1.1. Mô tả về bộ dữ liệu thử nghiệm:

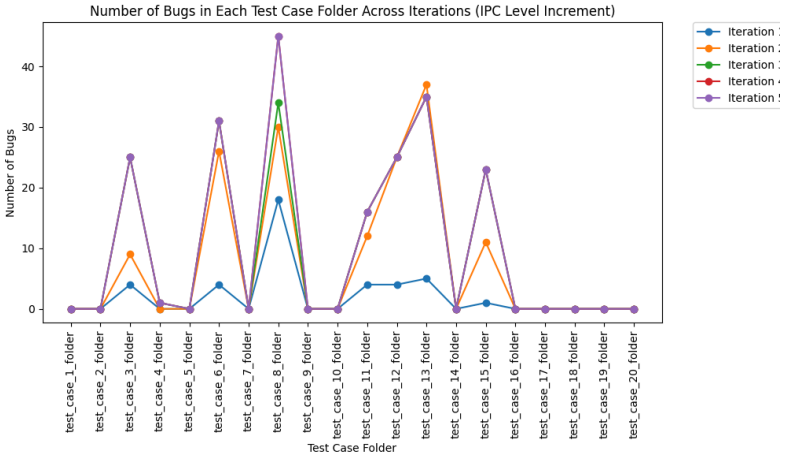
Bộ dữ liệu thử nghiệm - dataset được sử dụng cho thí nghiệm này bao gồm 20 chương trình C++ với số lượng phương thức, lời gọi khác nhau. Bộ dữ liệu này là một phần rất nhỏ được trích xuất từ bộ dữ liệu của Alexandru Jercan về mã nguồn C++ của các thí sinh tham gia lập trình thi đấu và đồng thời cũng được thêm vào bổ sung một số mẫu mã nguồn C++ từ bài giảng về LLVM của Edmund Wong, Irene Huang, Taiyue Liu và giáo sư Lin Tan. Ngoài một số nguồn dữ liệu này, nhóm tác giả cũng lấy ý tưởng thêm từ các mã nguồn thực tế thường sử dụng cho khóa học nhập môn lập trình để hoàn thiện. Các bộ dữ liệu được đề cập để trích xuất sử dụng cho thí nghiệm này không được kiểm chứng và phân loại cần thiết để đánh giá mức độ hiệu quả của quy trình phát hiện cục bộ lỗi, do đó không thể xác định hoàn toàn đúng và đủ các tiêu chí đánh giá cần thiết để xem xét kết quả. Đồng thời kết quả này cũng không hoàn toàn phản ánh đúng các lỗi quy trình phát hiện được do vấn đề liên quan đến trường hợp *false-positive* (nghĩa là trường hợp các khả năng lỗi được xác định bởi quy trình nhưng thực hiện kiểm tra trên bề mặt mã nguồn lại không cho rằng là khả năng lỗi).

1.2. Các loại lỗi được sử dụng trong thử nghiệm.

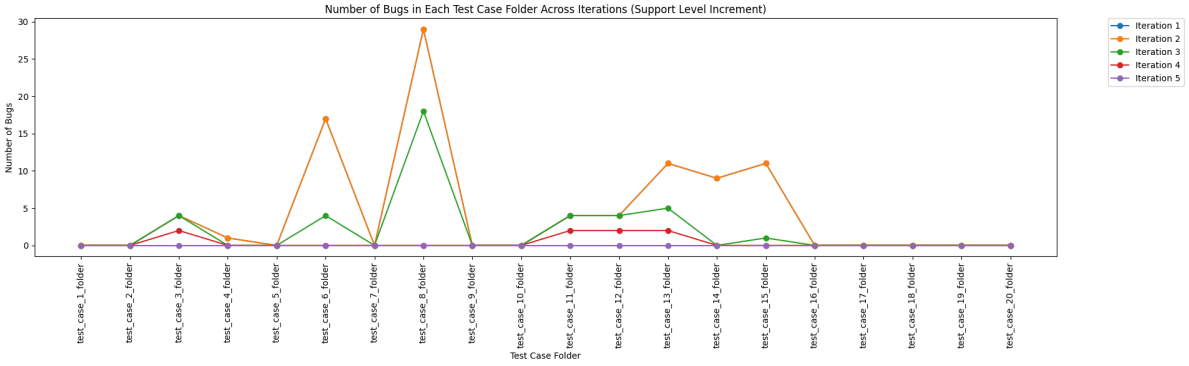
Trong thử nghiệm này sẽ chỉ bao gồm các lỗi liên quan đến sai sót cặp hàm, các loại lỗi liên quan đến cú pháp, crashing, các cảnh báo sử dụng sẽ không được đem vào thử nghiệm do đã có các trình biên dịch hỗ trợ xác định tại thời điểm biên dịch. Đồng thời, thông qua việc thực hiện phân tích tĩnh - *static analysis* cũng giúp hỗ trợ xác định phần lớn các lỗi về khai báo hàm, khai báo dữ liệu, …

2. Kết quả thống kê:

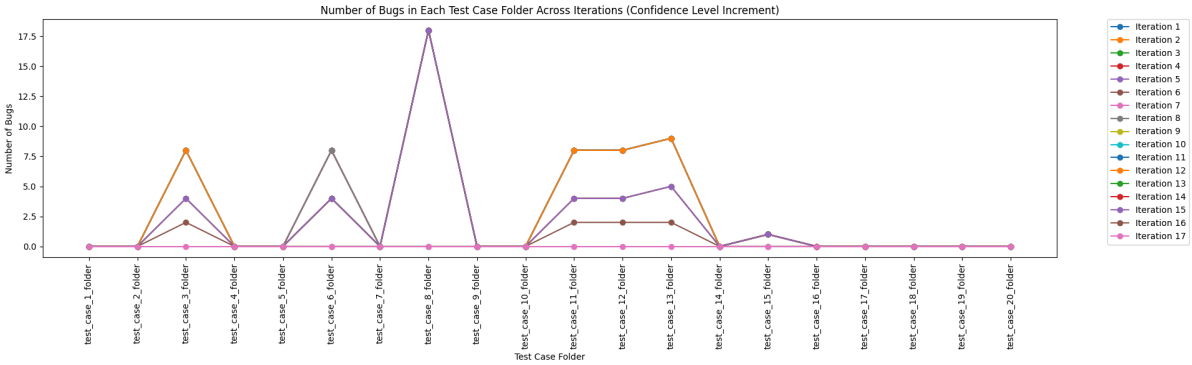
Với việc các tham số ngưỡng như mức phân tích liên thủ tục, độ hỗ trợ, mức tự tin có thể thay đổi tùy chỉnh theo ý thích, nhóm tác giả đã thử nghiệm quy trình theo bộ giá trị mẫu (1,3,65) nghĩa là mức độ phân tích liên thủ tục bằng 1, độ hỗ trợ ngưỡng là 3 và ngưỡng tự tin là 65%. Với việc tăng dần giá trị của một tham số trong khi giữ nguyên các tham số khác và cho ra được kết quả thống kê khả năng lỗi phát hiện như sau:



Hình 1. Kết quả thống kê số lượng khả năng lỗi khi gia tăng mức độ phân tích thủ tục.



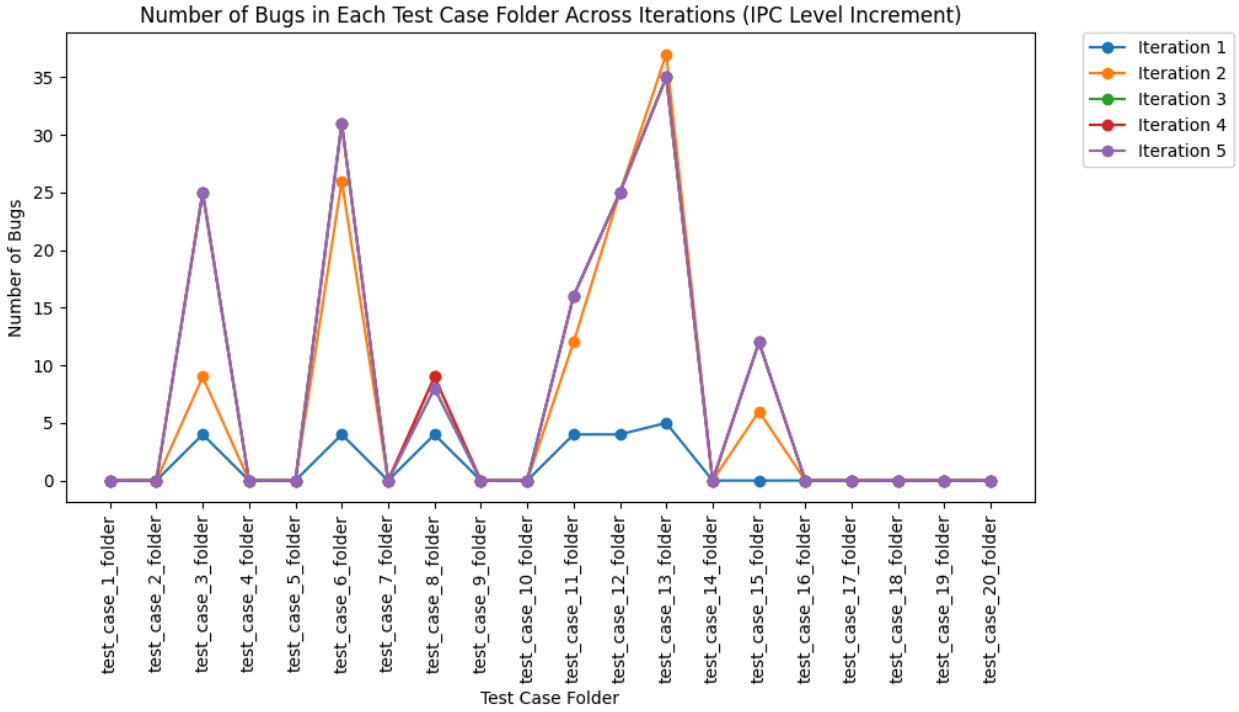
Hình 2. Kết quả thống kê số lượng khả năng lỗi khi gia tăng ngưỡng hỗ trợ.



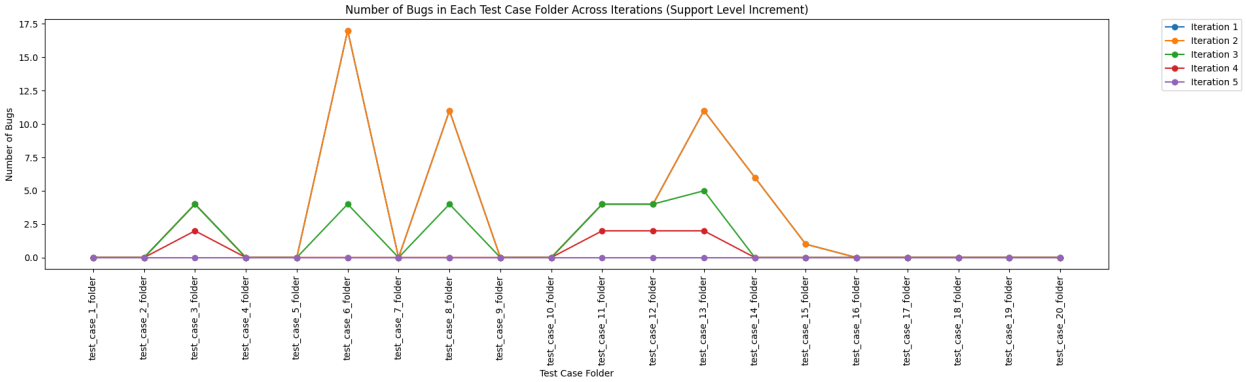
Hình 3. Kết quả thống kê số lượng khả năng lỗi khi gia tăng ngưỡng tự tin.

Dựa vào kết quả thu được, nhóm tác giả phát hiện các mức gia tăng khả năng lỗi bất thường ở một số mẫu kiểm thử khi thực hiện thử nghiệm, điều này có thể được giải thích do cơ chế xác định khả năng lỗi của quy trình đã đào sâu vào việc phân tích cả các hàm được gọi xen lẫn các hàm ẩn (nghĩa là các hàm không thể xác định khi xem xét chương trình nhưng lại được gọi đến khi thực hiện bởi bộ biên dịch và thư viện hỗ trợ) của chương trình. Điều này dẫn đến việc các cặp hàm ẩn này tuy được sử dụng cho các triển khai hiện hữu ở chương trình gốc và không thể được xem là lỗi nhưng khi thực hiện quy trình phát hiện lại vô

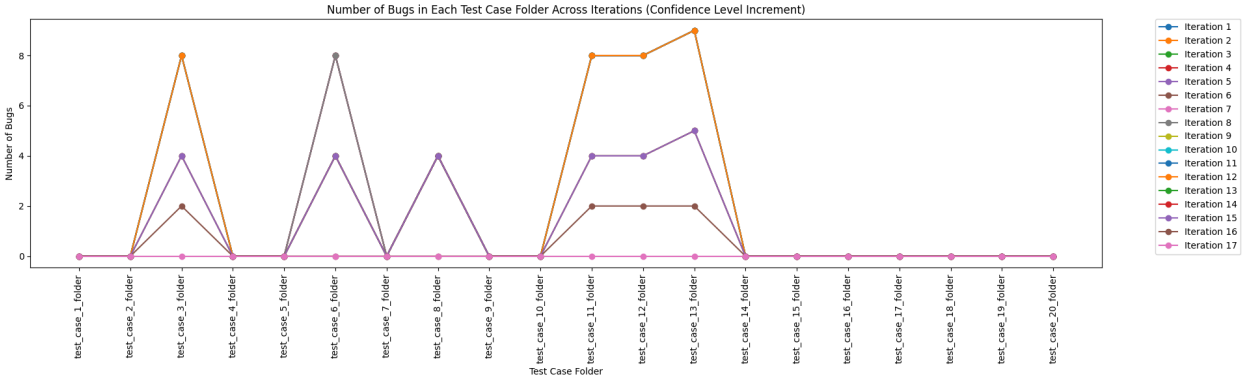
tình bị xem là lỗi. Do đó, cũng cần phải bổ sung các mẫu nhận diện hàm ẩn cho quy trình (ví dụ đối với C++, các mẫu nhận diện hàm ẩn có thể kể đến như toán tử "::", dấu "__" trong tên hàm ẩn).
Kết quả sau khi thực hiện sẽ có sự thay đổi như sau:



Hình 4: Kết quả thống kê số lượng khả năng lỗi khi gia tăng mức phân tích thủ tục và loại bỏ các khả năng lỗi false-positive.



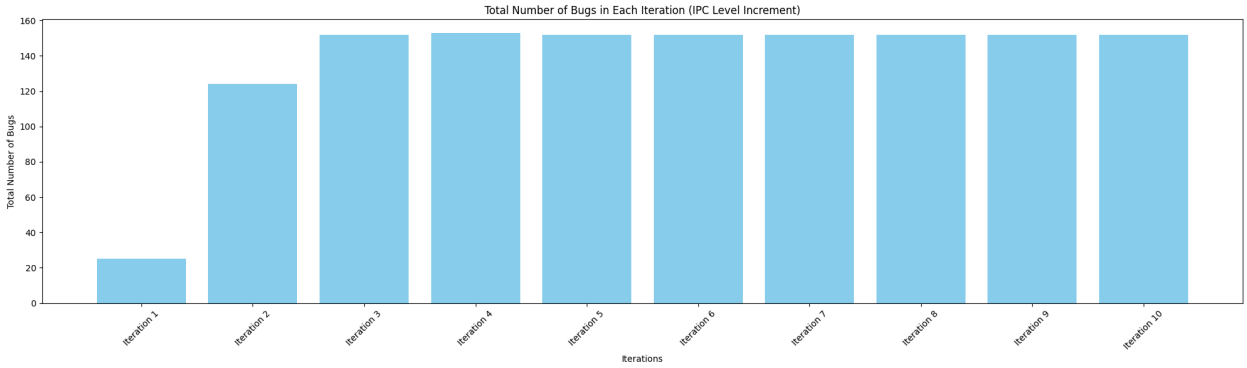
Hình 5: Kết quả thống kê số lượng khả năng lỗi khi gia tăng ngưỡng hỗ trợ và loại bỏ các khả năng lỗi false-positive.



Hình 6. Kết quả thống kê số lượng khả năng lỗi khi gia tăng ngưỡng tự tin và loại bỏ các khả năng lỗi false-positive.

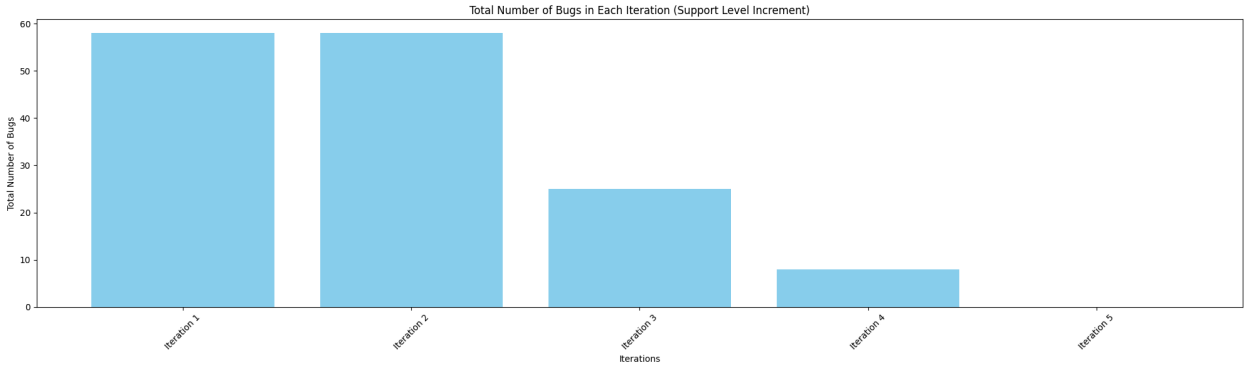
2.1. So sánh khả năng phát hiện lỗi khi các tham số thay đổi tuyến tính:

Đối với gia tăng mức độ phân tích liên thủ tục một cách tuyến tính, số lượng các khả năng lỗi chỉ có sự biến động ở các mức phân tích từ 1 đến 3, trong khi thực hiện ở mức phân tích từ 4 trở đi sẽ chỉ giữ nguyên số lượng khả năng lỗi có thể tìm thấy. Điều này có thể được biểu diễn thành biểu đồ phân tích trong 10 lần thực thi với mức độ phân tích tăng dần như sau (lưu ý, biểu đồ này được kiểm nghiệm ở ngưỡng hỗ trợ là 3 và ngưỡng tự tin là 65%):

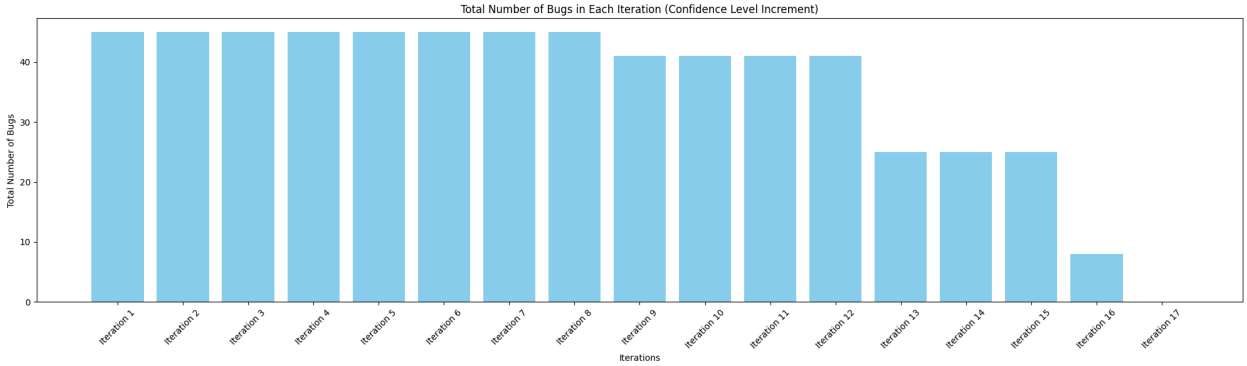


Hình 7: Biểu đồ cột phân tích sự tương quan giữa số lượng khả năng lỗi phát hiện với số lần tăng tuyến tính mức độ phân tích liên thủ tục.

Đối với gia tăng ngưỡng hỗ trợ hoặc ngưỡng tự tin tuyến tính, việc thực hiện phân tích thống kê cho thấy sự ngược lại, khi tăng dần giá trị, số lượng khả năng lỗi tìm thấy lại càng giảm đi cho đến khi không còn tìm thấy. Biểu đồ bên dưới giúp phản ánh nhận xét trên (ở đây mức độ phân tích liên thủ tục được giữ cố định ở mức 1, ngưỡng hỗ trợ bắt đầu tăng dần từ 1 và ngưỡng tự tin bắt đầu tăng dần từ 5%):



Hình 8: Biểu đồ cột phân tích sự tương quan giữa số lượng khả năng lỗi phát hiện với số lần tăng tuyến tính ngưỡng hỗ trợ.



Hình 9. Biểu đồ cột phân tích sự tương quan giữa số lượng khả năng lỗi phát hiện với số lần tăng tuyến tính ngưỡng hỗ trợ.

Trình bày báo cáo

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC VÀ KỸ THUẬT THÔNG TIN

HUỲNH THANH SANG

LÊ VŨ KHÁNH THẢO

NGUYỄN ĐÌNH HUY

BÁO CÁO ĐỒ ÁN HỌC PHẦN

NHẬP MÔN BẢO ĐẢM VÀ AN NINH THÔNG TIN

TÌM HIỂU PHÁT HIỆN LỖI PHẦN MỀM BẰNG ĐỒ THỊ

RESEARCH ON SOFTWARE BUG DETECTION USING GRAPH

TP. HỒ CHÍ MINH, 2025

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC VÀ KỸ THUẬT THÔNG TIN

HUỲNH THANH SANG – 23521341

LÊ VŨ KHÁNH THẢO – 23521469

NGUYỄN ĐÌNH HUY – 23520626

BÁO CÁO ĐỒ ÁN HỌC PHẦN

NHẬP MÔN BẢO ĐẢM VÀ AN NINH THÔNG TIN

TÌM HIỂU PHÁT HIỆN LỖI PHẦN MỀM BẰNG ĐỒ THỊ

RESEARCH ON SOFTWARE BUG DETECTION USING GRAPH

GIẢNG VIÊN HƯỚNG DẪN

NGUYỄN TẤN CẦM

TP. HỒ CHÍ MINH, 2025

THÔNG TIN BÁO CÁO ĐỒ ÁN HỌC PHẦN

Ngày báo cáo đồ án học phần được xác định là ngày 20 tháng 5 năm 2025 theo quyết định của giảng viên học phần IE105 - Nhập môn bảo đảm và an ninh thông tin lớp IE105.P21.

LỜI CẢM ƠN

Nhóm chúng em xin bày tỏ lòng biết ơn sâu sắc đến thầy Nguyễn Tấn Cầm, giảng viên học phần IE105 - "Nhập môn bảo đảm và an ninh thông tin", lớp IE105.P21 thuộc khoa Khoa học và Kỹ thuật thông tin. Nhờ sự hướng dẫn của thầy, chúng em đã có được những kiến thức và kỹ năng nền tảng để thực hiện đề tài "Tìm hiểu phát hiện lỗi phần mềm bằng đồ thị". Mặc dù vậy, do còn hạn chế về kinh nghiệm và kiến thức chuyên môn, nhóm em nhận thấy vẫn còn nhiều thiếu sót trong quá trình nghiên cứu, đánh giá và trình bày đề tài. Chúng em rất mong nhận được sự góp ý, chỉ dẫn từ quý thầy cô bộ môn để đề tài này được hoàn thiện hơn nữa. Xin chân thành cảm ơn.

MỤC LỤC

DANH MỤC HÌNH

DANH MỤC BẢNG

DANH MỤC TỪ VIẾT TẮT

TÓM TẮT ĐỀ TÀI ĐỒ ÁN

Trong quá trình phát triển phần mềm, lập trình viên thường phải đối mặt với các lỗi - *bug* khác nhau. Các lỗi này đa dạng từ logic phát triển đơn giản đến xung đột thư viện sử dụng, hay phức tạp hơn là các lỗi ẩn bên trong quá trình hoạt động nhưng khó phát hiện. Một trong số các lỗi ẩn này có thể kể đến như các trường hợp dữ liệu đặc biệt mà logic hoạt động không thể nắm bắt - *handle* được, lỗi liên quan đến tắc nghẽn luồng hoạt động - *deadlock*, lỗi xử lý biểu thức toán học, … .

Đa dạng phương pháp có thể được sử dụng như phân tích mã nguồn tĩnh, phân tích mã nguồn động, suy luận bất biến, ...Trong đó, quá trình phân tích mã nguồn tĩnh là phương pháp thông dụng được cả lập trình viên và kiểm thử viên, điều này có thể được lý giải bởi vì việc phân tích mã nguồn tĩnh chỉ cần thực hiện kiểm tra logic mã nguồn thủ công mà không cần kiểm chứng thực thi. Điều này giúp cho lập trình viên và kiểm thử viên có thể kiểm tra được phần lớn lỗi thông dụng như NullPointerException, ArrayIndexOutOfBoundsException, FileNotFoundException, ... Ngoài phân tích tĩnh, việc phân tích mã nguồn động bằng thực thi trình biên dịch cũng giúp tiếp tục loại bỏ các lỗi khác. Tuy nhiên, có một số lỗi lại không thể xác định bằng hai phương pháp kể trên, đặc biệt chính là các lỗi liên quan đến lời gọi cặp hàm, lỗi rò rỉ bộ nhớ, lỗi liên quan đến môi trường, ...

Do đó, về tổng quan nhìn nhận bối cảnh kỹ thuật hiện tại, lỗi phần mềm không chỉ xuất hiện đa dạng mà cũng có số lượng công cụ phong phú để phát hiện, điều này dẫn tới việc không thể phán định một phương pháp bất kỳ có khả năng phát hiện toàn bộ lỗi trong một mã nguồn chương trình. Vì vậy, nhóm tác giả đối với đề tài này quyết định lựa chọn áp dụng thử nghiệm các công cụ hỗ trợ kết hợp phương pháp phân tích đồ thị lời gọi kèm suy luận bất biến nhằm xác định và cục bộ hóa phạm vi của các lỗi bao gồm lỗi phát hiện bởi trình biên dịch, lỗi bộ nhớ, lỗi xử lý đa luồng, lỗi tranh chấp dữ liệu và lỗi vi phạm bất biến cặp hàm.

Chương 1. GIỚI THIỆU

1.1. Tổng quan

Chương này sẽ giới thiệu các phần thông tin cơ bản của đề tài, làm cơ sở để xây dựng bối cảnh, vấn đề và động lực của phương pháp.

1.1.1. Tầm quan trọng của việc phát hiện lỗi

Trong bối cảnh xây dựng và phát triển phần mềm, việc phát hiện các lỗi kỹ thuật của phần mềm là chuỗi các bước cần thiết để nhanh chóng khắc phục các gián đoạn hoạt động của phần mềm. Điều này rõ ràng giúp tiết kiệm chi phí triển khai khi phát hiện và sửa lỗi ở giai đoạn đầu của quá trình thiết kế xây dựng. Nếu một lỗi sai ban đầu không được phát hiện và sửa đổi, điều kiện này có thể là một nguy cơ tiềm ẩn của chuỗi lỗi - *chain of bugs* ở giai đoạn kiểm thử, vận hành sau này. Do đó, phát hiện các yếu tố bất ổn và lỗi thao tác kỹ thuật sẽ giảm đi thời gian phát triển dự kiến và nâng cao chất lượng phần mềm.

1.1.2. Đồ thị lời gọi trong phân tích cấu trúc và hành vi chương trình

Đồ thị lời gọi - *call graph* là một mô hình đồ thị có thể biểu diễn mối quan hệ giữa các hàm hoặc thủ tục trong chương trình. Mối quan hệ này được hiểu chính là lời gọi - *call* hoặc *invocation*. Bên trong đồ thị sẽ có hình thức biểu diễn (mô hình hóa) dưới dạng một đồ thị toán học chứa các đỉnh - *node* hoặc *vertice* để biểu diễn một hàm và các cạnh - *edge* để biểu diễn một lời gọi. (1)

Trong thực thi chương trình - *program execution*, đồ thị lời gọi giúp lập trình viên hình dung được cấu trúc điều khiển tổng thể của chương trình và mối quan hệ phụ thuộc giữa các thành phần. Dựa vào thông tin thực thi của chương trình mà đồ thị lời gọi sẽ xác định luồng thực thi, theo dõi dòng dữ liệu và xác định các điểm ảnh hưởng qua lại giữa các hàm. Do đó, việc sử dụng đồ thị lời gọi để thực hiện phân tích hành vi

tĩnh - *static analysis* mà không cần thực thi chương trình cũng là một trong số các phương pháp dò tìm lỗi phổ biến.

1.1.3. Suy luận bất biến trong khả năng phát hiện lỗi

Bất biến - *invariant* là một khái niệm có thể được phát biểu ở nhiều lĩnh vực trải dài từ toán học, vật lý, âm nhạc, ... với nhiều cách khác nhau. Tuy nhiên, các phát biểu ở nhiều lĩnh vực đều đảm bảo một tính chất cốt lõi của bất biến, đó chính là tính đúng đắn không đổi ở bất kể tác động. Nhìn nhận ở góc độ về phạm trù khoa học máy tính, điều này đồng nghĩa với các điều kiện, mệnh đề, khai báo hay lời gọi hàm được xem là các bất biến tại vị trí cụ thể của chính nó trong chương trình mặc kệ con đường thực thi nào dẫn đến nó.

Từ nhìn nhận bên trên, việc thực hiện suy luận bất biến - *invariant inference* chính là quá trình tự động hoặc bán tự động để tìm ra các bất biến khả năng - *likely invariants* trong chương trình bằng cách phân tích mã nguồn hoặc theo dõi hành vi thực thi (2). Điều này có tác dụng gì đối với việc phân tích lỗi? Việc suy luận bất biến sẽ đảm bảo tìm kiếm được các bất biến chung của chương trình, ví dụ như tần suất một lời gọi hàm thỏa mãn ở một giá trị số nào đó trong thực thi chương trình, hoặc một cặp hàm phải xuất hiện cùng nhau trong thực thi chương trình như `malloc()` và `free()` trong ngôn ngữ C. Từ các bất biến này, lập trình viên có thể xác định các ràng buộc rõ ràng của chương trình để tìm kiếm các thực thi vi phạm các bất biến đề ra.

1.2. Vấn đề và động lực

Phần này sẽ trình bày các vấn đề cản trở của bối cảnh hiện tại và động lực để thử nghiệm các phương pháp được trình bày ở phần trước đó.

1.2.1. Thách thức trong việc tìm kiếm và sửa các lỗi không gây crash

Về mặt lý thuyết, các lỗi phần mềm trong quá trình xây dựng thiết kế chương trình được chia thành hai phạm trù hoạt động và tần suất (3, 4). Ở phạm trù hoạt động, lỗi có thể xảy ra với chương trình sẽ khiến chương trình rơi vào một trong hai trạng thái là bình thường - *normal* hay *non-crashing và* tê liệt - *crashing*. Ở phạm trù tần suất, các lỗi có thể xuất hiện ở mức độ từ hiếm gặp đến thường xuyên. Đặc biệt, ở các lỗi không gây ra *crashing*, chúng thường khó phát hiện hơn do điều kiện để xảy ra lỗi rất hiếm dựa trên miền dữ liệu đầu vào. Đồng thời, không giống như các lỗi gây crash - *crashing bugs* thường dẫn đến lỗi hệ thống, dừng chương trình hoặc sinh thông báo lỗi, các lỗi không gây crash - *non-crashing bugs* thường âm thầm gây ra hành vi sai lệch (như tính toán sai, trạng thái sai, kết quả đầu ra không đúng), khiến chúng khó bị phát hiện trong quá trình kiểm thử thông thường.

1.2.2. Sự cần thiết của các kỹ thuật phân tích động và tĩnh để hỗ trợ tìm kiếm lỗi

Dựa trên các thách thức phát hiện lỗi, rất nhiều kỹ thuật được áp dụng trong xử lý. Có thể kể đến kỹ thuật phân tích tĩnh - *static analysis* (5,6,7), kỹ thuật phân tích động - *dynamic analysis* (8,9,10), kỹ thuật khai thác đồ thị lời gọi - *call graph mining* (1,3,4,11,12), suy luận bất biến chung - *general invariant inference* (13,14,15,16,17,18,19), học máy - *machine learning* (20,21,22,23), ...

Trong số các kỹ thuật được nêu tên, kỹ thuật phân tích tĩnh và kỹ thuật phân tích động là hai kỹ thuật phổ biến nhất để tìm kiếm lỗi. Đối với kỹ thuật phân tích tĩnh, kỹ thuật này phân tích chương trình mà không thực sự thực thi các chương trình (9), lập trình viên không cần phải thực thi chương trình mà chỉ cần thực hiện quét thông tin trên đoạn mã nguồn - *code* hoặc các thông tin liên quan để thực hiện tìm kiếm lỗi. Bên cạnh kỹ thuật phân tích tĩnh, kỹ thuật phân tích động cũng có nhiều ưu điểm đáng chú ý. Kỹ thuật phân tích động được thể hiện thông qua việc thực thi chương trình để trực tiếp xem xét hoạt động của chương trình nhằm phát hiện lỗi (9). Ngoài việc phân

tích thực thi hành vi chương trình, kỹ thuật phân tích động cũng tập trung vào các yếu tố như dấu vết thực thi - *execution trace*, lời gọi ngăn xếp - *call stack* tại thời điểm thực thi để đánh giá và tìm kiếm lỗi.

Nhờ vào các kỹ thuật hỗ trợ ở trên, quy mô và độ phức tạp chương trình ngày càng mở rộng nhờ các công cụ áp dụng. Chúng giúp giảm thiểu chi phí và thời gian phát triển phần mềm, đồng thời cũng là nền tảng cho học máy và các phương pháp mới hơn xuất hiện.

1.2.3. Ưu điểm của việc sử dụng suy luận bất biến lên đồ thị lời gọi để phát hiện lỗi so với các phương pháp khác

Mặc dù việc áp dụng các kỹ thuật phân tích tĩnh, kỹ thuật phân tích động, ... cũng đủ đáp ứng cho tuyệt đại đa số nhu cầu tìm lỗi của lập trình viên, nhưng bản thân các kỹ thuật cũng có hạn chế bên trong.

Đối với kỹ thuật phân tích tĩnh, kỹ thuật này có nét tương tự với việc tìm kiếm lỗi thủ công và tốn rất nhiều thời gian. Do đó, rất nhiều kỹ thuật khác được sử dụng kèm với phân tích tĩnh để thực hiện như tích hợp LLM (6), suy luận linh hoạt - dynamic reasoning (9), ...

Còn đối với kỹ thuật phân tích động, chúng cũng có hạn chế khi thiếu đi độ nhạy bối cảnh thực thi - *execution context sensitivity* do chi phí trình bày ở thời gian chạy - *runtime*. Nghĩa là lúc này, lập trình viên phải thực hiện tìm kiếm thủ công với các thông tin cung cấp khi thực thi hoặc sử dụng các công cụ như Breadcrumbs để phân tích. Đối với công cụ Breadcrumbs, nó có thể giúp tăng độ nhạy bối cảnh so với mức bình thường khoảng 10% đến 20% để giảm thiểu chi phí thu thập thông tin ở thời gian thực, nhưng đổi lại cũng có trường hợp dẫn đến phạm vi tìm kiếm vượt ngoài khoảng cho phép (8).

Trong một số trường hợp lỗi đặc biệt như lỗi vi phạm bất biến, lỗi không đúng yêu cầu thiết kế thì cả kỹ thuật phân tích động và kỹ thuật phân tích tĩnh cũng không xác định được các lỗi cụ thể này mà chỉ dừng lại ở các lỗi thông dụng như cú pháp, hành vi, ... Đối với phương pháp suy luận bất biến lên đồ thị lời gọi, kỹ thuật này được đề cập ở (14,15) chính là sự kết hợp giữa việc phân tích tĩnh đồ thị lời gọi và suy luận bất biến để khai thác lỗi. Kỹ thuật này cho phép mở ra khả năng "học và ghi nhớ" hành vi "bình thường" của chương trình thực thi. Ngoài ra, các bất biến này có thể được thay đổi tùy thuộc vào ngưỡng thỏa mãn hành vi - *threshold* mà lập trình viên đề ra, do đó với việc thay đổi *threshold* của kỹ thuật, sẽ cho ra số lượng lỗi khác nhau (14). Khi một bất biến có xảy ra vi phạm, điều này có thể được kỹ thuật phát hiện là sự bất thường tiềm ẩn ngay trước khi chương trình thực thi. Không giống các kỹ thuật kiểm thử dựa trên kiểm tra kết quả, phương pháp này phát hiện lỗi dựa trên sự sai lệch hành vi nội tại, ví dụ như một lời gọi hàm bất thường, một tần suất gọi thay đổi đáng kể.

Ngoài cách vận dụng suy luận bất biến lên đồ thị lời gọi, còn có một số kỹ thuật khác áp dụng lên đồ thị lời gọi chương trình để thực hiện tìm kiếm các lỗi tương tự như khai thác đồ thị - *graph mining* (1,3,4,11,12) , mạng lưới neuron đồ thị - *graph neural network* (24).

1.3. Mục tiêu đề ra của báo cáo:

Dựa vào vấn đề rào cản của việc tìm kiếm lỗi trong không gian phần mềm, các kỹ thuật đã được nghiên cứu sử dụng và ưu điểm của phương pháp được đề cập, nhóm tác giả nhìn nhận rằng các mô hình phát hiện lỗi hiện tại vẫn còn sự thiếu sót và chưa liền mạch để giúp nhận diện đầy đủ các lỗi có thể xảy ra của một chương trình thực thi. Do đó, thông qua việc làm rõ và xây dựng một mô hình con thực thi tuần tự đối với phương pháp áp dụng suy luận bất biến lên đồ thị lời gọi, nhóm tác giả muốn đặt mục tiêu xây dựng thử nghiệm mô hình với bộ biên dịch Clang, bộ công cụ phân tích thực thi Valgrind và phương pháp được đề cập để lắp ráp thành một đường ống - *pipeline* tự

động hóa hoàn chỉnh hỗ trợ phát hiện lỗi trước và sau khi thực thi, đặc biệt sẽ tập trung vào các lỗi có thể phát hiện bởi trình biên dịch, các lỗi về bộ nhớ, đa luồng, tranh chấp dữ liệu và lỗi thiếu sót nội bộ cặp hàm. Đồng thời, nhóm cũng mong muốn trình bày một cách chi tiết các khái niệm, kỹ thuật liên quan để hỗ trợ quá trình xây dựng thử nghiệm.

1.4. Phạm vi tìm hiểu của báo cáo:

Báo cáo này tập trung vào việc giải quyết xem xét phương pháp ở mức độ hàm, cụ thể hơn là tập trung vào lời gọi giữa các hàm trong chương trình. Đồng thời cũng xem xét kết hợp với việc sử dụng bộ biên dịch Clang làm cơ sở cho kỹ thuật phân tích tĩnh và sử dụng Valgrind là cơ sở cho kỹ thuật phân tích động để xây dựng đồ thị lời gọi với bộ máy ảo bậc thấp LLVM nhằm phản ánh chính xác hành vi thực thi thực tế. Các thử nghiệm của báo cáo này tập trung vào tính đơn giản, dễ thực hiện với bộ dữ liệu thử nghiệm mẫu đơn giản nhằm kiểm tra hiệu quả của mô hình phát hiện lỗi, do đó bản thân việc thử nghiệm chỉ giới hạn trong quá trình kiểm thử bán tự động mà không mở rộng sang kiểm thử tự động.

Chương 2. CƠ SỞ LÝ THUYẾT

Chương này sẽ trình bày cơ sở lý thuyết cho nội dung các phương pháp ở các chương sau.

2.1. Đồ thị lời gọi - Call graph

Đồ thị lời gọi là một trong những cách biểu diễn những lời gọi nào có thể xảy ra khi thực thi chương trình.

2.1.1. Định nghĩa đồ thị lời gọi

Đồ thị lời gọi là hình thức biểu diễn dữ liệu hữu ích cho các chương trình điều khiển và luồng dữ liệu nghiên cứu giao tiếp giữa các hàm (tức là cách các hàm trao đổi thông tin). Nó chứa tất cả các mối quan hệ giữa các hàm trong một chương trình và có thể chứa thông tin bổ trợ liên quan đến dữ liệu trong mỗi hàm và dữ liệu toàn cục được chia sẻ giữa các hàm.

Về mặt định nghĩa toán học, đồ thị lời gọi có thể biểu diễn thành đồ thị có hướng với ký hiệu G = (V,E) với V là tập các đỉnh đại diện cho hàm và E là tập các lời gọi từ đỉnh hàm nguồn đến đỉnh hàm đích. Với mỗi hàm P_i thuộc tập hàm V và P, mỗi đỉnh V_i là tương ứng một-một với mỗi hàm P_i và tập vector lời gọi có đỉnh hàm tương ứng. (26)

2.1.2. Đồ thị lời gọi tĩnh - Static call graph

Đồ thị lời gọi tĩnh là sự trừu tượng hóa của một chương trình máy tính biểu diễn tất cả lời gọi hàm hay phương thức có thể xảy ra. Nó được xây dựng bằng cách phân tích mã nguồn mà không cần phải chạy chương trình và không thể hiện thứ tự thực thi chương trình - *flow-insensitive*.

Vì xem xét tất cả các lời gọi có thể xảy ra nên đồ thị lời gọi tĩnh có thể bao gồm những lời gọi không bao giờ xảy ra như lời gọi phương thức đa hình, thực tế chỉ có một

phương thức được gọi nhưng trong đồ thị biểu diễn tất cả lời gọi từ các lớp con ghi đè lại phương thức đa hình đó.

2.1.3. Đồ thị lời gọi động - Dynamic call graph

Đồ thị lời gọi động biểu diễn luồng điều khiển của một chương trình khi nó được thực thi. Nó hiển thị trình tự các lệnh gọi hàm hay phương thức được thực hiện trong quá trình thực thi chương trình, cùng với tùy chọn biểu thị các tham số được truyền cho từng hàm.

Vì nó thực sự thực thi chương trình nên sẽ không có lời gọi không bao giờ xảy ra nào nhưng đồ thị cũng chỉ chứa những lời gọi với những đầu vào cụ thể, những lời gọi chưa được thực thi sẽ không được ghi lại. Ngoài ra, đồ thị lời gọi động cũng chiếm một phần bộ nhớ cũng như tài nguyên của bộ xử lý để ghi lại các lời gọi.

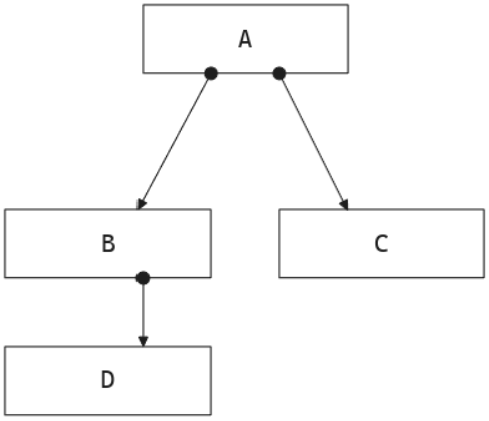
2.1.4. Phân tích nội thủ tục - Intraprocedural analysis

Phân tích nội thủ tục là loại phân tích chỉ xem xét trong phạm vi một hàm đơn lẻ, sử dụng thông tin cục bộ để tính toán luồng điều khiển và dữ liệu bên trong hàm đó. Nó không xem xét các hàm khác hoặc cách các hàm tương tác với nhau. Vì chỉ tập trung vào một hàm, phân tích nội thủ tục thường đơn giản và chi phí tính toán thấp vì không cần theo dõi lời gọi ngoài hàm Do không xem xét các lời gọi hàm ngoài - *external calling*, nó có thể bỏ lỡ thông tin quan trọng về luồng điều khiển và sự phụ thuộc dữ liệu giữa các hàm. Trong việc phân tích đồ thị lời gọi, phân tích nội thủ tục có thể giúp xác định các lời gọi hàm trực tiếp trong một hàm và hỗ trợ xây dựng đồ thị lời gọi.

2.1.5. Phân tích liên thủ tục - Interprocedural analysis

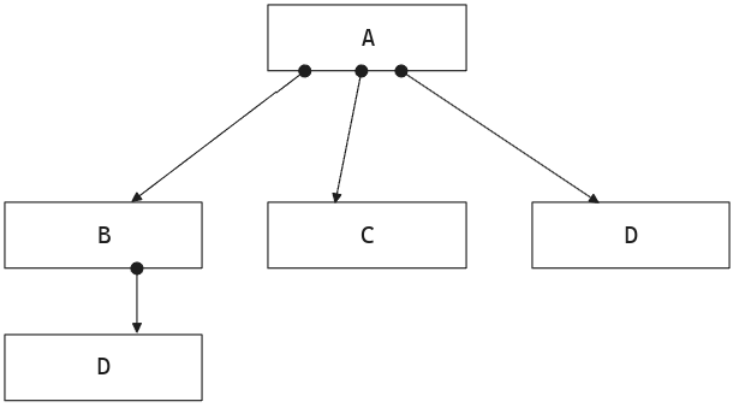
Phân tích liên thủ tục so với phân tích nội thủ tục mở rộng phạm vi ra toàn bộ chương trình, xem xét các mối quan hệ giữa các hàm, bao gồm lời gọi, trả về và truyền tham số, giúp xây dựng và tinh chỉnh đồ thị lời gọi chính xác hơn nhưng đồng thời phức tạp và tốn kém về mặt tính toán hơn.

Phân tích liên thủ tục có độ chính xác cao hơn vì nó có thể thu thập thông tin chính xác hơn về luồng điều khiển và sự phụ thuộc dữ liệu, đặc biệt là trong các chương trình phức tạp với nhiều lời gọi hàm. Nhưng việc triển khai thường phức tạp hơn so với phân tích nội thủ tục. Trong phân tích đồ thị lời gọi, phân tích liên thủ tập có thể xác định các lời gọi hàm gián tiếp. Ví dụ, nếu ta có hàm A gọi đến hàm B và C, hàm B thì gọi đến D. Với phân tích liên thủ tục ở mức một, ta sẽ thu được đồ thị lời gọi phân tích như sau:



Hình 2.1: Đồ thị lời gọi ở mức phân tích liên thủ tục một.

Với phân tích liên thủ tục ở mức hai, ta sẽ thu được đồ thị lời gọi phân tích như sau:



Hình 2.2: Đồ thị lời gọi ở mức phân tích liên thủ tục hai.

2.2. Lỗi - Bug

Phần này trình bày về các loại lỗi trong quá trình phân tích thiết kế phần mềm.

2.2.1. Định nghĩa lỗi

Lỗi là các sai sót, khuyết điểm hoặc lỗi trong phần mềm máy tính dẫn đến kết quả không mong muốn hoặc không lường trước được. Chúng có thể xuất hiện theo nhiều cách khác nhau, bao gồm hành vi không mong muốn, hệ thống bị sập hoặc đóng băng hoặc đầu ra không đúng và không đủ. (27)

Trong tài liệu (4), lỗi trong quá trình thực thi chương trình biểu hiện bằng cách tạo ra một số kết quả khác với kết quả đã chỉ định hoặc dẫn đến một số hành vi thời gian chạy không mong muốn như sự cố hoặc chạy không kết thúc.

2.2.2. Phân loại lỗi

Theo tài liệu (4), lỗi có thể được chia ra thành ba loại và sáu kiểu lỗi:

Lỗi loại hành vi

Loại lỗi này gồm hai lỗi chính là lỗi gây tê liệt - *crashing bug*, lỗi không gây tê liệt - *non-crashing bug*.

Lỗi gây tê liệt là loại lỗi dẫn tới sự kết thúc của chương trình. Thông thường các ngoại lệ của một ngôn ngữ lập trình sẽ thuộc về lỗi này, ví dụ như NullPointerException, ArithmeticException, IndexOutOfBoundException, ... Các lỗi này có thể được tìm thấy thông qua dấu vết ngăn xếp - *stack trace*.

Lỗi không gây tê liệt trái lại với lỗi gây tê liệt sẽ khó tìm thấy hơn và thường sẽ được tập trung vào các phương pháp tìm kiếm hơn.

Lỗi loại tần suất

Loại lỗi này gồm hai lỗi là lỗi thỉnh thoảng - *occasional bug* và lỗi thường xuyên - *non-occasional bug*.

Lỗi thường xuyên là loại lỗi dễ xảy ra, dễ thực hiện lại - *reproduce*. Các lỗi này có thể kể đến như lỗi cú pháp, lỗi logic đơn giản, … dễ dàng kiểm tra do có thể viết các test case để kiểm tra và xác nhận việc sửa lỗi và thường liên quan đến lỗi logic hoặc lỗi cú pháp.

Lỗi thỉnh thoảng thường khó phát hiện hơn so với lỗi thường xuyên, có thể phụ thuộc vào sự kết hợp phức tạp của các yếu tố, chẳng hạn như trạng thái hệ thống, thời gian, dữ liệu đầu vào cụ thể, hoặc thậm chí là sự tương tác với các phần mềm khác.

Lỗi loại tác động

Loại lỗi này gồm lỗi tác động cấu trúc - *structure-affecting bug* và lỗi tác động tần suất gọi - *call frequency-affecting bug*.

Lỗi tác động cấu trúc không chỉ gây ra hành vi sai hoặc sập ứng dụng, mà còn có thể làm suy thoái cấu trúc chương trình. Những lỗi cấu trúc này dẫn đến khó bảo trì, dễ xuất hiện lỗi lan truyền và tăng chi phí phát triển dài hạn. Một ví dụ có thể kể đến như lỗi tăng phụ thuộc liên kết cặp - *coupling*, *coupling* là mức độ phụ thuộc giữa các lớp, hàm hoặc thành phần con - *module*, các thể hiện của lỗi này có thể kể đến như sử dụng biến toàn cục, truyền tham số không cần thiết làm tăng *coupling* khiến các *module* liên quan quá nhiều đến nhau.

Lỗi tác động tần suất gọi sẽ không làm thay đổi cấu trúc đồ thị lời gọi nhưng lại thay đổi đi số lần thực thi hàm so với kỳ vọng. Những lỗi này thường khó phát hiện bằng phân tích cấu trúc đơn thuần vì đường đi lời gọi hàm vẫn giống nhau, chỉ có tần suất

gọi thay đổi. Lỗi này ảnh hưởng đến hiệu năng khi tăng tần suất gọi có thể gây giảm hiệu năng nghiêm trọng, hoặc ngược lại bỏ sót xử lý khi gọi quá ít. Làm giảm độ tin cậy khi dữ liệu không được xử lý đúng số lần, dẫn đến mất mát hoặc trùng lặp kết quả.

2.3. Bất biến - Invariant

Phần này sẽ trình bày lý thuyết về bất biến và suy luận với bất biến.

2.3.1. Định nghĩa bất biến

Bất biến - *invariant* là một điều kiện hoặc mệnh đề luôn đúng tại một vị trí cụ thể trong chương trình, bất kể con đường thực thi nào dẫn đến nó. Chúng giúp lập trình viên xác định các giả định mà chương trình phụ thuộc vào, từ đó bảo vệ họ khỏi việc vô tình vi phạm các giả định này khi thực hiện thay đổi. (2, 18)

2.3.2.Bất biến khả năng

Bất biến khả năng là những tính chất của chương trình mà được suy luận là đúng dựa trên quan sát hành vi của chương trình trong quá trình thực thi. Chúng phản ánh những mối quan hệ hoặc điều kiện "gần như luôn đúng" trong thực tế thực thi, và thường được dùng để hỗ trợ kiểm thử, gỡ lỗi, hay tăng cường tính an toàn của chương trình.

2.3.3. Suy luận bất biến trong chương trình và đồ thị lời gọi

Suy luận bất biến là quá trình tự động xác định các bất biến của chương trình. Kỹ thuật này nhằm tìm ra các bất biến có khả năng đúng từ chương trình đó. Các quy tắc hoặc mẫu hành vi thường xuyên xuất hiện trong đồ thị lời gọi như sau:

Chuỗi lời gọi cố định

Mẫu đường đi thực thi hàm A -> B -> C xuất hiện hầu hết trong các lần chạy, dùng để suy ra luồng nghiệp vụ.

Bất biến thứ tự

Nếu B được gọi, thì A phải được gọi trước đó trong cùng ngữ cảnh.

Bất biến tần suất

Tỉ lệ số lần gọi hàm X/n số lần gọi hàm Y nằm trong một ngưỡng ổn định mẫu (ví dụ $X \approx 2Y$).

Bất biến vắng mặt

Hàm Z không bao giờ có cạnh - lời gọi đến một hàm Y hoặc hàm X mẫu.

Quan hệ tham số-lời gọi

Giá trị trả về của một hàm X luôn là tham số đầu vào của hàm Y trong cùng một chuỗi lời gọi.

Ngoài việc sử dụng các quy tắc hoặc mẫu hành vi để xác định, một số phương pháp khác có thể như sử dụng phân tích thống kê, sử dụng học máy, phân tích mẫu đồ thị con.

Chương 3. PHƯƠNG PHÁP TỐI GIẢN ĐỒ THỊ LỜI GỌI

3.1. Tổng quan

Chương này tập trung vào tìm hiểu các phương pháp tối giản đồ thị lời gọi nhằm hỗ trợ quá trình suy luận trên đồ thị.

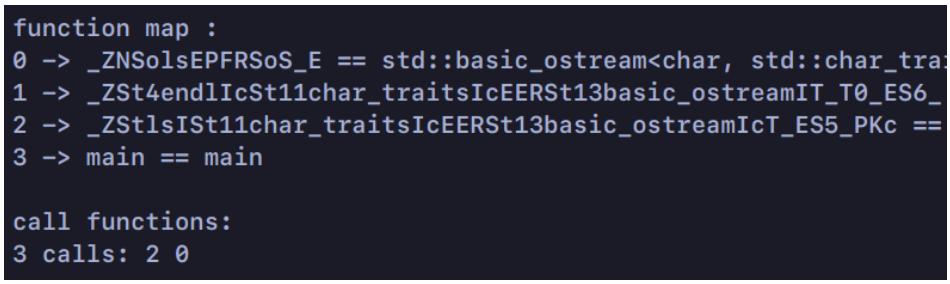
3.1.1. Sự cần thiết của việc tối giản đồ thị lời gọi

Khi thực hiện xây dựng đồ thị lời gọi từ ngôn ngữ lập trình và công cụ, đa phần chỉ hỗ trợ việc xuất ra hoàn chỉnh đồ thị mà không quan tâm tới độ phức tạp thông tin. Nếu không thực hiện tối giản cả về cấu trúc thông tin lẫn kích thước thông tin, chi phí để thực hiện phân tích sẽ vô cùng lớn, ảnh hưởng đến cả thời gian chạy, bộ nhớ tiêu thụ và độ trễ phản hồi của thông tin hệ thống. Ví dụ đối với công cụ dòng lệnh `opt` của LLVM, khi thực hiện trích xuất tệp bitcode (tệp bitcode là tệp trung gian của công cụ Clang dùng để lưu trữ thông tin chương trình không phụ thuộc vào một nền tảng) của một chương trình C hoặc C++, ta sẽ thu được thông tin về đồ thị lời gọi mẫu như sau:

```
Call graph node <<null function>><<0x55a86934df90>>  #uses=0
  CS<None> calls function 'main'
  CS<None> calls function '_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc'
  CS<None> calls function '_ZNSolsEPFRSoS_E'
  CS<None> calls function '_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_'

Call graph node for function: '_ZNSolsEPFRSoS_E'<<0x55a86934a430>>  #uses=2
  CS<None> calls external node

Call graph node for function: '_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_'<<0x55a86934a4b0>>  #uses=1
  CS<None> calls external node

Call graph node for function: '_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc'<<0x55a86934a380>>  #uses=2
  CS<None> calls external node

Call graph node for function: 'main'<<0x55a8693487d0>>  #uses=1
  CS<0x55a86934e540> calls function '_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc'
  CS<0x55a86934e600> calls function '_ZNSolsEPFRSoS_E
```

Hình 3.1: Thông tin đồ thị lời gọi trích xuất từ tệp bitcode của lệnh opt.

Từ Hình 3.1 có thể thấy thông tin về đồ thị lời gọi được trình bày đầy đủ nhưng lại gây khó khăn cho việc đọc hiểu, vì vậy cần thực hiện một số thao tác như ánh xạ và lưu trữ nhằm tối giản thông tin. Từ Hình 3.1, nhóm tác giả thực hiện tối giản thông tin từ đồ thị lời gọi thành mẫu thông tin như sau:



Hình 3.2: Thông tin đồ thị lời gọi được tối giản từ Hình 3.1.

Sau khi thực hiện tối giản thông tin, đồ thị sẽ trở nên gọn gàng và dễ thao tác hơn với kiểu dữ liệu số và cấu trúc dữ liệu của C/C++. Các phần thông tin bị cắt đi sau dấu "==" chỉ là phần biến đổi - *demangle* các tên bị mã hóa trong tệp bitcode, do đó có thể tùy ý thêm vào hoặc không tùy theo yêu cầu.

3.1.2. Các phương pháp tối giản và ưu nhược điểm

Phương pháp Total Reduction

Gộp tất cả các đỉnh đại diện cho cùng một hàm trong toàn bộ đồ thị thành 1 đỉnh duy nhất. Mỗi cạnh chỉ biểu diễn sự tồn tại của mối quan hệ gọi hàm, không mang thông tin về tần suất. Phương pháp này đơn giản dễ thực hiện, phù hợp đối với các phân tích cấu trúc cơ bản nhưng đổi lại làm mất thông tin về tần suất gọi, có thể không phát hiện các lỗi liên quan đến tần suất gọi - *call frequency affecting bugs*.

Phương pháp Total reduction with edge weight

Thực hiện tương tự phương pháp Total Reduction nhưng mỗi cạnh trong đồ thị được gắn thêm trọng số đại diện cho tần số gọi giữa hai hàm trong nhiều lần chạy. Điều này giúp giữ lại thông tin của hành vi, mở rộng hỗ trợ của Total Reduction đối với lỗi liên quan đến tần suất gọi, cân bằng giữa giảm thiểu độ phức tạp và giữ lại thông tin quan trọng. Ngược lại, phương pháp này sẽ tăng xử lý số liệu thống kê, điều này có thể không cần thiết nếu yêu cầu phát hiện lỗi không quá chú trọng vào tần suất.

Phương pháp Zero-one-many Reduction

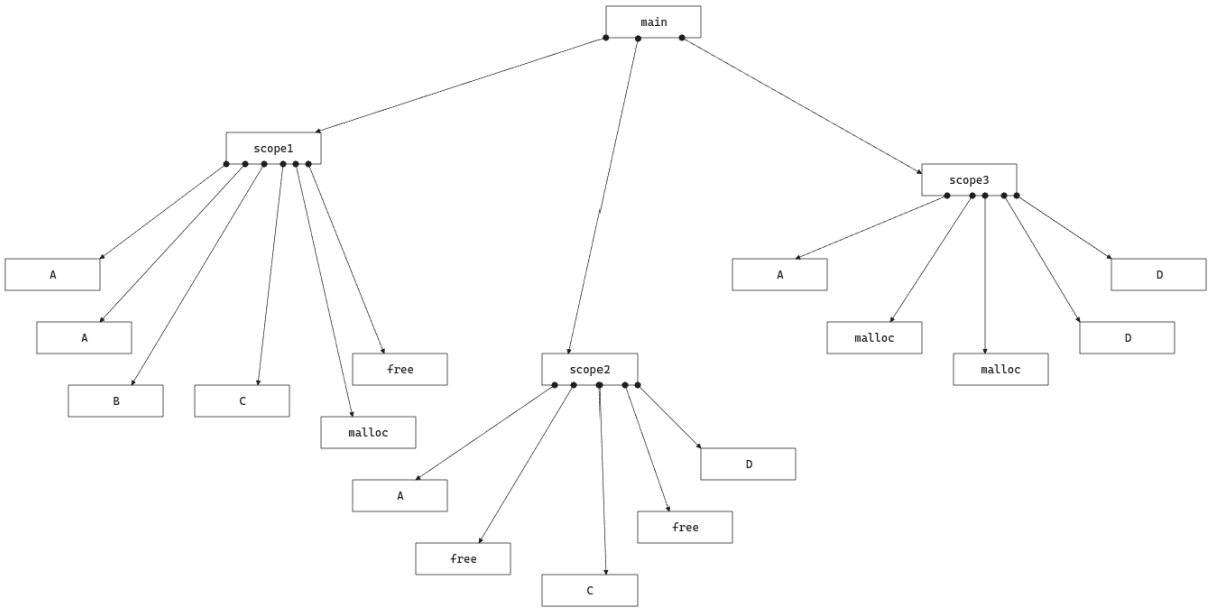
Thực hiện việc trừu tượng hóa tần suất thành ba mức: 0 đối với không gọi, 1 đối với gọi 1 lần, N đối với gọi nhiều hơn 1 lần. Phương pháp này có thể thay thế cho phương pháp Total reduction with edge weight do đơn giản hóa biểu diễn nhưng vẫn giữ được ý nghĩa hành vi tổng quát, hỗ trợ đối với suy luận bất biến định tính - *qualitative invariant*. Tuy nhiên, cách trừu tượng này lại không phù hợp trong trường hợp cần sự chi tiết trong thông tin và đồng thời cũng cho ra đồ thị có sự tối giản hoàn toàn.

3.2. Các kỹ thuật tối giản đồ thị lời gọi

Phần này sẽ tập trung vào ba phương pháp được sử dụng phổ biến là tối giản toàn cục - *total reduction*, tối giản toàn cục với trọng số lời gọi - *total reduction with edge weight* và tối giản 0-1-N - *zero-one-many reduction*. Mỗi phương pháp sẽ có ưu nhược điểm khác nhau tùy thuộc vào mục đích sử dụng.

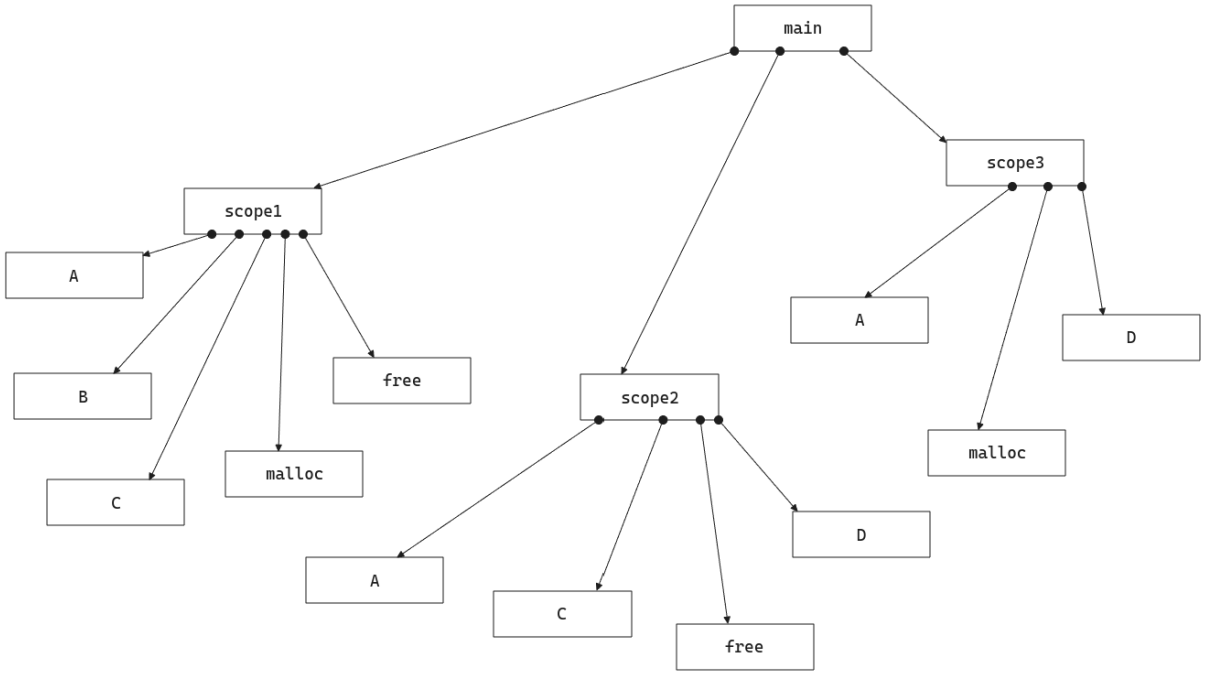
3.2.1. Tối giản toàn cục - Total reduction

Tối giản toàn cục tập trung vào việc tối giản thông tin đồ thị ở mức tối đa mà không quan tâm đến cấu trúc nội tại của đồ thị. Khi này, mỗi hàm sẽ chỉ xuất hiện một lần duy nhất trong đồ thị và được biểu thị bằng một đỉnh, một cạnh sẽ được nối giữa hai đỉnh nếu có lời gọi giữa hai hàm tương ứng. Lấy ví dụ, ta có một đồ thị lời gọi mẫu như sau:



Hình 3.3: Đồ thị lời gọi mẫu.

Nếu thực hiện phương pháp tối giản toàn cục, đồ thị lời gọi sẽ có hình thái như sau:



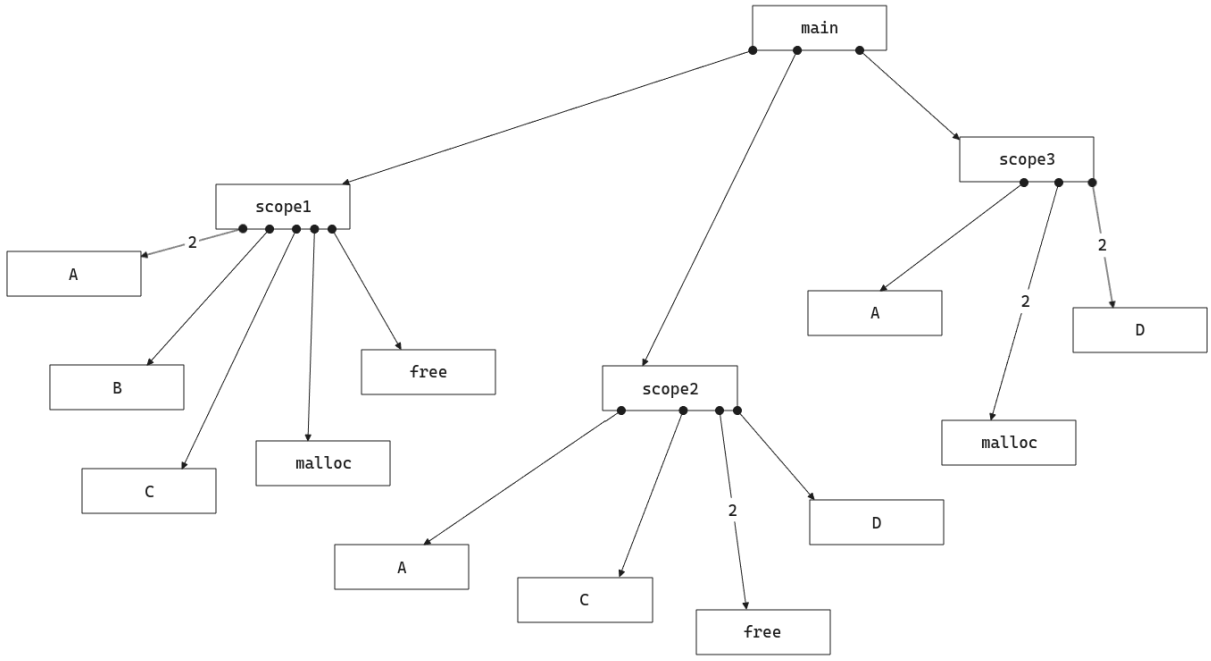
Hình 3.4: Đồ thị lời gọi sau khi áp dụng phương pháp tối giản toàn cục.

Đối với phương pháp tối giản toàn cục, chúng có khả năng giảm đáng kể kích thước đồ thị. Trong thí nghiệm mẫu của (25), nhóm tác giả này đã áp dụng thực hiện phương pháp này lên đồ thị lời gọi mẫu có 22 đỉnh 21 cạnh và cho ra kết quả ấn tượng khi đồ thị lời gọi chỉ còn lại 6 đỉnh 6 cạnh. Tuy nhiên, đánh đổi lại, cấu trúc sẽ bị mất đi thông tin và thay đổi. Điều này lại gây khó khăn trong việc truy vấn hoặc khai thác mẫu đồ thị con - *subgraph*.

Do đó, kỹ thuật tối giản toàn cục chỉ phù hợp khi cần xử lý lượng lớn dữ liệu và không yêu cầu thông tin chi tiết, phân tích cấu trúc tổng quát giữa các hàm, không phân tích hành vi chi tiết.

3.2.2. Tối giản toàn cục với trọng số lời gọi - Total reduction with edge weight

Kỹ thuật này là sự mở rộng từ kỹ thuật tối giản toàn cục khi bổ sung thêm tần suất gọi - ở đây chính là trọng số vào các cạnh. Lấy lại ví dụ ở trên, đối với kỹ thuật này ta sẽ thu được kết quả như sau:



Hình 3.5: Đồ thị lời gọi sau khi áp dụng tối giản toàn cục với trọng số lời gọi.

Kỹ thuật này giảm thiểu các rủi ro về vấn đề không phát hiện các lỗi tần suất mà vẫn đảm bảo cấu trúc thông tin. Tuy nhiên, kỹ thuật này lại không được sử dụng phổ biến do phải bổ sung thêm chi phí thống kê.

3.2.3. Tối giản 0-1-N - Zero-One-Many reduction

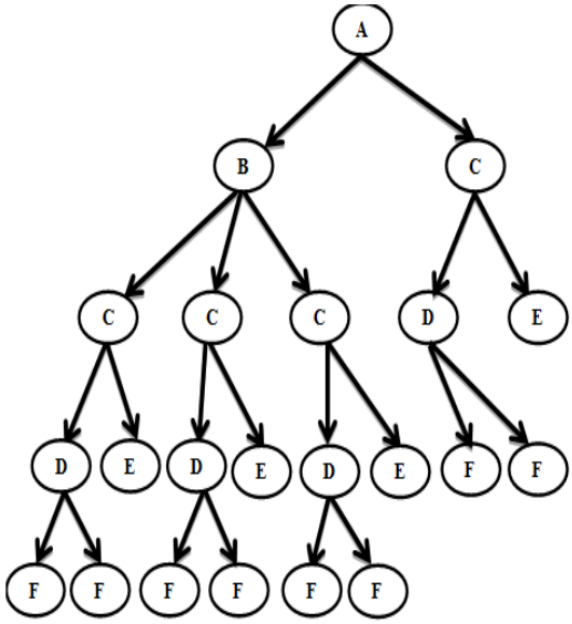
Kỹ thuật tối giản 0-1-N là một kỹ thuật có nét tương tự với kỹ thuật tối giản toàn cục, tuy nhiên trái với việc loại bỏ toàn bộ để giữ lại duy nhất thông tin, kỹ thuật tối giản 0-1-N thực hiện việc trừu tượng hóa tần suất thành ba mức. Trong đó đối với mức N sẽ phụ thuộc vào yêu cầu tối giản. Lấy ví dụ của (12), kỹ thuật này sẽ định sẵn mức N = 2, nghĩa là các lời gọi nào có tần suất vượt quá N sẽ bị loại bỏ và chỉ giữ lại đúng N lời gọi trên đồ thị. Điều này giúp giảm tải việc phân tích thống kê nhưng lại không hoàn toàn tối giản đồ thị, phù hợp hơn đối với việc sử dụng kèm phương pháp khai thác đồ thị - *graph mining*.

3.3. So sánh các kỹ thuật

Phần này thực hiện so sánh giữa kỹ thuật tối giản toàn cục - *total reduction* và kỹ thuật tối giản 0-1-N - *Zero-one-many reduction*. Kỹ thuật tối giản toàn cục với trọng số lời gọi chỉ là phần mở rộng của kỹ thuật tối giản toàn cục, do đó không cần thiết phải đưa vào so sánh.

3.3.1. Hiệu quả tối giản đồ thị

Theo thí nghiệm mô phỏng của (25) với mẫu đồ thị 22 đỉnh 21 cạnh như sau:



Hình 3.6: Đồ thị lời gọi mẫu từ thử nghiệm của (25).

Kết quả được nhóm tác giả đưa ra như sau:

Phương pháp tối giản	**Số lượng đỉnh**	**Số lượng cạnh**	**Ảnh hưởng lên đồ thị**
Đồ thị mẫu	22	21	
Tối giản toàn cục (.)	6	6	Mất thông tin và thay đổi cấu trúc
Tối giản 0-1-N (.)	15	14	Không tối giản hoàn toàn Mất thông tin nhưng đảm bảo cấu trúc

Bảng 3.1: So sánh giữa hai kỹ thuật tối giản đồ thị lời gọi.

Kết quả so sánh cho thấy tính hiệu quả cao của phương pháp tối giản toàn cục lên đồ thị mẫu, nhưng đồng thời cũng tỉ lệ thuận với mức độ ảnh hưởng lên đồ thị. Ngược lại, phương pháp tối giản 0-1-N chỉ cho ra kết quả tối giản ở mức vừa phải nhưng đủ đáp ứng thông tin và cấu trúc để thực hiện phân tích sâu hơn.

3.3.2. Ảnh hưởng đến khả năng phát hiện lỗi

Hiện tại, các nghiên cứu liên quan đến lĩnh vực này vẫn chưa hoàn toàn đầy đủ để thu thập và phân tích sâu ảnh hưởng của các kỹ thuật lên khả năng phát hiện lỗi. Đồng thời, các kỹ thuật này chỉ được xem là bước trung gian đầu vào cho việc phân tích bằng các kỹ thuật khác như suy luận bất biến, khai thác đồ thị, học máy, LLM, ... Do đó, ở thời điểm hiện tại nhóm tác giả nhận định rằng ảnh hưởng của các kỹ thuật này lên khả năng phát hiện chỉ dừng lại xem xét ở khả năng khai thác thu thập đầy đủ thông tin mà vẫn giữ nguyên được cấu trúc về đồ thị lời gọi. Ở tiêu chí, kỹ thuật tối giản 0-1-N có phần nhỉnh hơn so với kỹ thuật tối giản toàn cục và tối giản toàn cục với trọng số lời gọi do có thể bảo toàn được cấu trúc đồ thị lời gọi.

Chương 4. PHƯƠNG PHÁP PHÁT HIỆN LỖI CẶP HÀM DỰA TRÊN ĐỒ THỊ LỜI GỌI VÀ SUY LUẬN BẤT BIẾN

4.1. Tổng quan

Chương này sẽ trình bày chi tiết về phương pháp phát hiện lỗi cặp hàm dựa trên đồ thị lời gọi đã tối giản và suy luận bất biến.

4.2. Nguyên tắc hoạt động tuần tự

Phần này sẽ trình bày đầy đủ các bước của phương pháp.

4.2.1. Thu thập và tiền xử lý đồ thị lời gọi

Đây là quá trình đầu tiên trong việc phát hiện lỗi, ở bước này, tùy thuộc vào ngôn ngữ lập trình được lựa chọn sẽ có nhiều phương pháp thực hiện khác nhau. Nhóm tác giả thực hiện lựa chọn thử nghiệm trên hai ngôn ngữ C và C++ vì tính đơn giản và được xây dựng sẵn bộ công cụ hỗ trợ opt từ LLVM. Sau khi xác định được ngôn ngữ và bộ công cụ đi kèm, lúc này ta có thể lựa chọn thêm bash script để thực hiện tự động hóa quy trình xây dựng. Với đồ thị lời gọi thu thập được, việc tiền xử lý đồ thị lời gọi sẽ bao gồm một số công đoạn như ánh xạ, loại bỏ thông tin thừa, thay đổi thông tin nhằm phù hợp với nhu cầu ở các giai đoạn sau.

4.2.2. Tối giản đồ thị lời gọi

Sau khi đã thực hiện tiền xử lý đồ thị lời gọi, ta sẽ tiếp tục thực hiện tối giản đồ thị lời gọi dựa theo các phương pháp đã đề cập ở phần lý thuyết. Ở đây, nhóm tác giả thực hiện lựa chọn theo phương pháp tối giản 0-1-N với N = 1 để đảm bảo tính đơn giản, dễ thực hiện đối với thử nghiệm. Việc sử dụng N = 1 cho phép tối giản thông tin tối đa với các lời gọi có tần suất cao mà vẫn đảm bảo toàn bộ cấu trúc thông tin của lời gọi.

4.2.3. Suy luận các bất biến khả năng

Ở bước này, thực hiện suy luận dựa trên việc xác định các cặp hàm (x,y) và bản thân các hàm x, y bất kỳ để tính số lần một cặp hàm hoặc hàm xuất hiện - *support* và độ tin cậy của cặp hàm đã xác định - *confidence* (*confidence* có thể hiểu đơn giản chính là xác suất mà hàm y xuất hiện khi hàm x xuất hiện). Công thức tính *support* và *confidence* được tính như sau:

$$\text{support}\,(a) = \sum(\text{function a appears})$$

$$\text{support}\,(a, b) = \sum(\text{function a and b appear together})$$

$$\text{confidence}\,(\{a, b\}, \{a\}) = \frac{\text{support}\,(a, b)}{\text{support}\,(a)}$$

Sau khi tính toán các giá trị cần thiết, thực hiện so sánh và giữ lại các cặp hàm thỏa mãn giá trị ngưỡng - *threshold* của *support* và *confidence*. Hai giá trị ngưỡng này có thể tùy ý thay đổi phụ thuộc vào yêu cầu tần suất lỗi xuất hiện là bao nhiêu.

4.2.4. Phân tích thống kê các bất biến có vi phạm

Dựa vào các cặp hàm (x,y) thỏa mãn yêu cầu, ta thực hiện duyệt lại đồ thị lời gọi gốc ban đầu và kiểm tra nếu có xuất hiện trường hợp một hàm thành phần y trong chương trình hoặc khu vực thực thi được gọi nhưng lại không đi kèm với hàm thành phần x. Nếu xuất hiện trường hợp này thì trả về lỗi có thể xảy ra tại cặp hàm và vị trí có khả năng lỗi đó.

Chương 5. ĐÁNH GIÁ THỐNG KÊ THỬ NGHIỆM

Chương này sẽ thực hiện mô tả việc áp dụng quy trình phát hiện lỗi với hai công cụ Clang, Valgrind và phương pháp được đề cập lên bộ dữ liệu mẫu và thực hiện thống kê kết quả trả về.

5.1. Thiết lập thử nghiệm

Thử nghiệm này được thực hiện trên cấu hình AMD Ryzen™ 3 4300U, 8GB RAM và nền tảng Fedora Linux 40 với các công cụ hỗ trợ như Github, VS Code, Clang, Valgrind, LLVM. Các ngôn ngữ lập trình được đưa vào sử dụng bao gồm Python, C, C++ và Bash Shell.

5.1.1. Bộ dữ liệu thử nghiệm

Bộ dữ liệu thử nghiệm - *dataset* được sử dụng bao gồm 30 chương trình C++ với số lượng phương phức, lời gọi khác nhau không được gắn nhãn lỗi. Đây là tập hợp từ nhiều nguồn khác nhau bao gồm bộ dữ liệu C++ của Alexandru Jercan về bài nộp lập trình thi đấu. mẫu chương trình từ bài giảng LLVM của Edmund Wong, Irene Huang, Taiyue Liu và giáo sư Lin Tan. Ngoài một số nguồn dữ liệu này, nhóm tác giả cũng lấy ý tưởng thêm từ các mã nguồn thực tế thường sử dụng cho khóa học nhập môn lập trình và hệ điều hành để hoàn thiện.

5.1.2. Lỗi sử dụng trong thử nghiệm

Trong thử nghiệm này sẽ sử dụng đa dạng lỗi từ các lỗi cú pháp đơn giản đến các lỗi nghiêm trọng như lỗi liên quan đến tranh chấp dữ liệu, lỗi rò rỉ bộ nhớ. Ngoài ra lỗi liên quan đến cặp hàm cũng được đem vào thử nghiệm để đánh giá hiệu quả khi thực hiện phân tích liên thủ tục kèm theo phương pháp suy luận bất biến lên đồ thị lời gọi. Phân bố lỗi trong thử nghiệm sẽ là ngẫu nhiên và việc áp dụng quy trình vào kiểm tra lỗi với các tham số khác nhau sẽ cho ra số lượng lỗi khác nhau. Điều này cũng dẫn tới

việc nên thực hiện kiểm tra các hành vi không xác định - *undefined behaviour*, các trường hợp dương tính giả - *false-positive* (nghĩa là các trường hợp bị xem là lỗi nhưng lại không có lỗi trên thực tế).

5.1.3. Quy trình thử nghiệm

Phần này sẽ trình bày đầy đủ các bước thực hiện để sử dụng phối hợp các công cụ trong việc tìm kiếm lỗi.

5.1.3.1. Kiểm tra lỗi từ biên dịch thực thi

Ở bước này, ta thực hiện truyền tệp chương trình C++ cần kiểm tra vào công cụ giao diện dòng lệnh Clang. Nếu có lỗi, Clang sẽ trả về trực tiếp trên dòng lệnh hoặc có thể điều hướng vào một tệp đầu ra và không thực hiện biên dịch. Ở bước này sẽ là quyết định quá trình tiếp theo, nếu phát hiện ra lỗi từ lúc biên dịch thì toàn bộ quy trình dừng lại. Cú pháp như sau:

```
clang++ -g <testfilename>.cpp -o <testfilename>
```

Nếu muốn chuyển hướng lỗi vào tệp đầu ra, ta thực hiện bổ sung thêm `&>> output.out` vào đuôi cú pháp.

5.1.3.2. Kiểm tra lỗi bộ nhớ bằng Valgrind

Sau khi vượt qua kiểm tra từ biên dịch thực thi, lúc này ta sẽ có tệp biên dịch chương trình, sử dụng công cụ memcheck của Valgrind, ta có thể thực hiện kiểm tra các lỗi liên quan đến bộ nhớ (lưu ý rằng, đối với một số sai sót trong thao tác bộ nhớ như cấp phát, thu hồi sẽ bị không bị memcheck chú trọng mà chỉ để thông báo xem như một cảnh báo - *warning*). Cú pháp thực thi như sau:

```
valgrind --tool=memcheck <testfilename> <arguments>
```

Việc thực hiện chuyển hướng đầu ra tương tự như phần 5.1.3.1. Nếu muốn kiểm tra kỹ hơn, Valgrind hỗ trợ kiểm tra lỗi rò rỉ chi tiết bằng tùy chọn trước tệp biên dịch thực thi `--leak-check=full`.

5.1.3.3. Kiểm tra lỗi xử lý đa luồng bằng Valgrind

Từ tệp biên dịch thực thi của chương trình, ta tiếp tục thực thi kiểm tra đa luồng với Valgrind bằng cú pháp sau:

```
valgrind --tool=helgrind <testfilename> <arguments>
```

Nếu có xảy ra các khả năng tranh chấp thì Valgrind sẽ thông báo và báo lỗi.

5.1.3.4. Kiểm tra lỗi tranh chấp dữ liệu bằng Valgrind

Sau khi kiểm tra lỗi bằng công cụ helgrind, Valgrind sẽ hỗ trợ tiếp tục kiểm tra lỗi tranh chấp dữ liệu từ chương trình với cú pháp sau:

```
valgrind --tool=drd <testfilename> <arguments>
```

5.1.3.5. Kiểm tra lỗi bằng đồ thị lời gọi và suy luận bất biến

Sau khi đã thực hiện kiểm tra lỗi với các công cụ kể trên, ta sẽ sử dụng công cụ Clang để sinh ra tệp bitcode chương trình như sau:

```
clang++ -emit-llvm -c <testfilename>.cpp -o <testfilename>.bc
```

Từ tệp bitcode thu được, ta sẽ xây dựng chương trình C++ để hỗ trợ cho việc phân tích và xác định các lỗi, trong chương trình C++ cũng hỗ trợ việc phân tích liên thủ tục nếu có thiết lập. Sau khi hoàn tất, công cụ opt của LLVM sẽ giúp xây dựng đồ thị lời gọi và truyền vào chương trình C++ để phân tích với cú pháp như sau:

```
opt -passes=print-callgraph <testfilename>.bc 2>&1 /dev/null
| ./<analyzer> <arguments>
```

Cú pháp này sử dụng cơ chế đường ống - *pipeline* của Bash Shell để thực hiện và hỗ trợ điều hướng lỗi với `/dev/null`.

5.2. Kết quả thống kê

Nội dung ……

5.3. So sánh khả năng phát hiện lỗi khi tham số thay đổi tuyến tính

Nội dung ……

Chương 6. KẾT LUẬN VÀ ĐỊNH HƯỚNG PHÁT TRIỂN

6.1. Kết luận

Qua quá trình tìm hiểu và phân tích, báo cáo đã làm rõ vai trò của đồ thị lời gọi trong việc mô hình hóa cấu trúc và hành vi thực thi của chương trình, từ đó trở thành nền tảng hữu ích cho các kỹ thuật phát hiện lỗi phần mềm. Đặc biệt, khi kết hợp với phương pháp suy luận bất biến và các công cụ hỗ trợ khác, đồ thị lời gọi có thể giúp phát hiện hiệu quả nhiều loại lỗi hành vi rất khó phát hiện bằng các phương pháp truyền thống.

Báo cáo đã phân tích ba kỹ thuật giảm đồ thị lời gọi phổ biến: Total Reduction, Total Reduction with Edge Weights, và Zero-One-Many Reduction. Mỗi kỹ thuật đều có ưu và nhược điểm riêng, ảnh hưởng trực tiếp đến khả năng giữ lại thông tin thực thi, mức độ trừu tượng hóa hành vi và hiệu quả phát hiện lỗi.

Thông qua thử nghiệm, phương pháp kết hợp của báo cáo cũng cho thấy hiệu quả rõ rệt khi vừa hỗ trợ tiền phân tích và phát hiện các lỗi, vừa áp dụng phân tích và giảm kích thước đồ thị, vừa giữ được thông tin cấu trúc và tần suất, từ đó hỗ trợ tốt hơn cho việc phát hiện bất thường khác trong chương trình.

6.1.1. Thách thức và cơ hội trong việc áp dụng thực tế

Thách thức

Trong các dự án thực tế, quy mô của đồ thị lời gọi có thể nảy sinh đến hàng chục ngàn đỉnh hàm và hàng triệu cạnh lời gọi. Do đó việc xây dựng, lưu trữ và phân tích đồ thị lời gọi đòi hỏi tài nguyên tính toán lớn và kỹ thuật tối ưu dữ liệu. Bên cạnh đó, việc phát triển liên tục không ngừng nghỉ của dự án cũng phát sinh ra nhiều lời gọi thư viện hoặc các lời gọi dư thừa làm nhiễu loạn quá trình suy luận bất biến và phát hiện lỗi. Đồng thời, các hành vi được xem là "bình thường" cũng đa dạng tùy thuộc vào yêu cầu

thiết kế, điều này làm cho việc suy luận có thể dẫn đến các trường hợp báo sai lỗi - *false-positive* và bỏ sót lỗi - *false-negative*.

Cơ hội

Mặc cho các thách thức kể trên, phương pháp vẫn có không ít ưu điểm đáng để áp dụng. Nó có thể tự tích hợp như một tính năng tự động hóa kiểm lỗi trong quy trình phát triển phát hành liên tục - *Continuous Integration/Continuous Deployment* - CI/CD, giúp đưa ra các cảnh báo lỗi sớm mà không cần viết thêm test case. Ngoài ra, phương pháp cũng góp phần giúp hỗ trợ bảo trì và phân tích tác động trong quá trình nâng cấp, làm sạch mã nguồn.

6.2. Định hướng phát triển

Suy luận bất biến với học máy và thống kê nâng cao

Thay thế việc sử dụng dựa trên nền tảng các điều kiện - *rule-based* hoặc thu thập thống kê đơn giản, việc suy luận bất biến có thể sử dụng tích hợp học máy để học các mẫu hành vi bình thường trên đồ thị lời gọi, hay sử dụng mạng neural đồ thị - *graph neural network* - GNN để khai thác cấu trúc đồ thị tốt hơn hoặc thay thế / kết hợp suy luận bất biến với suy luận Bayes - *Bayesian inference* để mô hình hóa trình tự lời gọi phức tạp.

Ngữ cảnh hóa bất biến - *Context-sensitive invariants*

Trong bối cảnh các phương pháp sử dụng bất biến hiện tại, chưa có các nghiên cứu để bổ sung ngữ cảnh cho các bất biến.

TÀI LIỆU THAM KHẢO

1) Improved Software Fault Detection with Graph Mining
2) Dynamically Discovering Likely Program Invariants to Support Program Evolution
3) Mining Behavior Graphs for Backtrace of Non-crashing Bugs
4) Software bug localization with graph mining
5) Experiences Using Static Analysis to Find Bugs
6) Enhancing Static Analysis for Practical Bug Detection An LLM Integrated Approach
7) Indirection Bounded Call Graph Analysis
8) Breadcrumbs Efficient Context Sensitivity for Dynamic Bug Detection Analyses
9) Combining Static and Dynamic Reasoning for Bug Detection
10) The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs
11) Mining Edge Weighted Call Graphs to Localise Software Bugs
12) Discriminative Pattern Mining in Software Fault Detection
13) Using Likely Invariants for Automated Software Fault Localization
14) ECE453 CS447 ECE653 CS647 SE465 Software Testing Quality Assurance and Maintenance Project Version 1
15) ECE453 ECE653 SE465 CS447 CS647 Tutorial 2 LLVM
16) Dynamically Discovering Likely Program Invariants to Support Program Evolution
17) Automatic Software Fault Localization using Generic Program Invariants
18) Detecting Bugs of Concurrent Programs With Program Invariants

19) Tracking Down Software Bugs Using Automatic Anomaly Detection

20) An Analysis of C CPP Datasets for Machine Learning Assisted Software Vulnerability Detection

21) Software Bug Prediction Using Supervised Machine Learning Algorithms

22) DeepBugs A Learning Approach to Name-Based Bug Detection

23) Automatic Fault Detection for Deep Learning Programs Using Graph Transformations

24) Detecting Condition Related Bugs with Control Flow Graph Neural Network

25) An Improved Technique for Storage and Reduction of Call Graph in Computer Memory

26) Constructing the Call Graph of a Program

27) https://www.sonarsource.com/learn/software-bugs/

The GNU C++ Library Manual

Bugs as Deviant Behavior A General Approach to Inferring Errors in Systems Code

Program Analysis Call Graphs

Data Flow Based Test Adequacy Analysis for Languages with Pointers

Complexity and Information in Invariant Inference

Experiments on the Effectiveness of Dataflow and Control-flow Based Test Adequacy Criteria

An Enhanced Approach for Software Bug localization using Map Reduce Technique based Apriori - MRTBA Algorithm

Model Checking and Control of Self Assembly

A cost effective software testing strategy employing online feedback information

An automatically created novel bug dataset and its validation in bug prediction

Presentation Data-flow Testing

Methodology for Controlling the Size of a Test Suite

Bug Characteristics in Open Source Software

Valgrind Manual

Introduction to Software Testing

Where the Bugs Are

Scalable and Incremental Software Bug Detection

The GNU C Library Reference Manual

Bảng copy tạm

Điều này có nét tương tự với việc tìm kiếm lỗi thủ công và tốn rất nhiều thời gian. Do đó, rất nhiều kỹ thuật khác được sử dụng kèm với phân tích tĩnh để thực hiện như tích hợp LLM (Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach), suy luận linh hoạt - *dynamic reasoning* (Combining Static and Dynamic Reasoning for Bug Detection), ...

—

. Tuy nhiên, kỹ thuật phân tích động cũng có hạn chế khi thiếu đi độ nhạy bối cảnh thực thi - *execution context sensitivity* do chi phí trình bày ở thời gian chạy - *runtime*. Nghĩa là lúc này, lập trình viên phải thực hiện tìm kiếm thủ công với các thông tin cung cấp khi thực thi hoặc sử dụng các công cụ như Breadcrumbs để phân tích. Đối với công cụ Breadcrumbs, nó có thể giúp tăng độ nhạy bối cảnh so với mức bình thường khoảng 10% đến 20% để giảm thiểu chi phí thu thập thông tin ở thời gian thực, nhưng đổi lại cũng có trường hợp dẫn đến phạm vi tìm kiếm vượt ngoài khoảng cho phép

Khung báo cáo nội dung sao chép

Chương 1. **TÊN CHƯƠNG 1**

1.1. Chủ đề cấp độ 2

Nội dung ……

Nội dung…….

1.1.1. Chủ đề cấp độ 3

Nội dung

1.1.2. Chủ đề cấp độ 3

1.1.2.1. Chủ đề cấp độ 4

Nội dung

Hình TÊN CHƯƠNG 2.1: Tên hình 1

Bảng TÊN CHƯƠNG 2.1: Tên bảng 1

Chương 2.TÊN CHƯƠNG 2

2.1. Chủ đề cấp độ 2

2.1.1. Chủ đề cấp độ 3

2.1.1.1. Chủ đề cấp độ 4

Bảng TÊN CHƯƠNG 2.2: Tên bảng 1

2.2. Chủ đề cấp độ 2

2.2.1. Chủ đề cấp độ 3

Chương 3.TÊN CHƯƠNG 3

3.1. Chủ đề cấp độ 2

Nội dung ……

Nội dung…….

3.1.1. Chủ đề cấp độ 3

3.1.1.1. Chủ đề cấp độ 4

3.2. Chủ đề cấp độ 2

TÀI LIỆU THAM KHẢO

Theo chuẩn IEEE