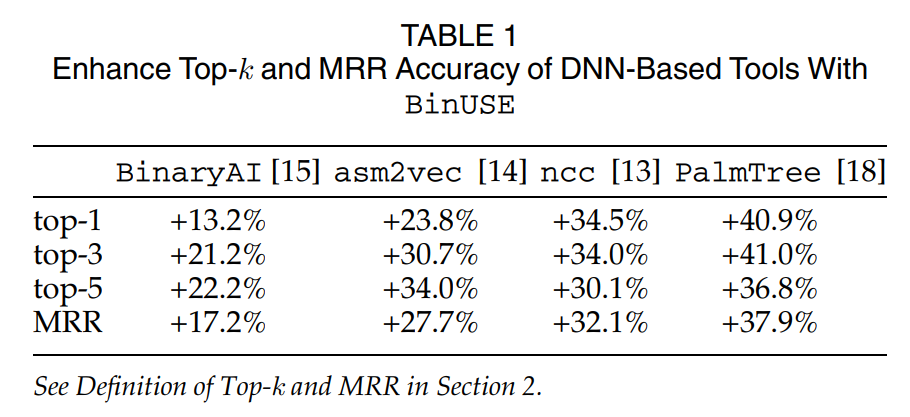
Enhancing DNN-Based Binary Code Function Search With Low-Cost Equivalence Checking.

Nâng cao Tìm kiếm Hàm Mã nhị phân dựa trên DNN với Kiểm tra Tương đương Chi phí thấp.

* Tóm tắt — Tìm kiếm hàm mã nhị phân đã được sử dụng như là cơ sở chính của các ứng dụng an ninh và kỹ thuật phần mềm khác nhau, bao gồm nhóm malware, phát hiện sao chép mã và kiểm tra lỗ hổng. Tuy nhiên, nhận dạng các hàm mã lệnh có logic tương tự vẫn là một thách thức. Hầu hết các công cụ tìm kiếm mã nhị phân dựa vào thông tin cấu trúc chương trình, chẳng hạn như luồng điều khiển và luồng dữ liệu, được trích xuất bằng các kỹ thuật phân tích chương trình hoặc các mạng neural sâu (DNNs). Tuy nhiên, các kỹ thuật dựa trên DNNs chỉ bắt được thông tin mức độ từ vựng, cấu trúc điều khiển hoặc luồng dữ liệu của mã nhị phân để học biểu diễn, điều này thường quá thô sơ và không chính xác đánh dấu chức năng của chương trình. Ngoài ra, chúng có thể thể hiện tính ổn định thấp trong nhiều tình huống phức tạp khác nhau, chẳng hạn như tối ưu hóa trình biên dịch và làm tròn mã. Bài báo này đề xuất một giải pháp tổng quát để cải thiện các ứng cử viên đạt hạng cao nhất trong tìm kiếm hàm mã nhị phân dựa trên DNN. Ý tưởng chính là thiết kế một kiểm tra tương đương toàn diện và giá rẻ, giúp nhanh chóng phát hiện những sai lệch chức năng giữa hàm mục tiêu và các hàm phù hợp hàng đầu. Các hàm không vượt qua kiểm tra tương đương này có thể được loại bỏ khỏi danh sách hàng đầu k, và các hàm vượt qua kiểm tra có thể được xem xét lại để tiến lên trong danh sách hàng đầu k, một cách cân nhắc. Chúng tôi thiết kế một kiểm tra tương đương thực tế và hiệu quả, được đặt tên là BinUSE, sử dụng phép thực thi biểu tượng không bắt buộc (USE). USE, một biến thể của phép thực thi biểu tượng, cải thiện tính mở rộng bằng cách bắt đầu thực thi biểu tượng trực tiếp từ các điểm vào của hàm và nới lỏng ràng buộc về các tham số của hàm. Nó loại bỏ các chi phí được tạo ra bởi việc lan truyền các luồng thực thi và ràng buộc đắt đỏ. BinUSE được thiết kế đặc biệt để cung cấp một kiểm tra tương đương mức độ hàm lắp ráp, cải thiện tìm kiếm mã nhị phân dựa trên DNN bằng cách giảm thiểu các cảnh báo sai với chi phí thấp. Đánh giá của chúng tôi cho thấy rằng BinUSE có thể kích hoạt việc cải tiến tổng quát và hiệu quả cho bốn công cụ tìm kiếm mã nhị phân dựa trên DNN hàng đầu khi đối mặt với các thách thức do các trình biên dịch, tối ưu hóa, làm tròn mã và kiến trúc khác nhau gây ra.

1 GIỚI THIỆU

* Tìm kiếm mã nhị phân xác định mức độ tương đồng giữa hai đoạn mã lắp ráp, và nó là một yếu tố quan trọng trong nhiều ứng dụng an ninh mã nhị phân. Ví dụ, phân tích mã độc nhận dạng các mẫu mã độc có cùng hành vi để tìm ra các họ mã độc và tránh việc phải phân tích lại các mẫu mã độc đã biết [1], [2]. Kiểm tra sự hiện diện các bản vá trích xuất các chữ ký bản vá bảo mật và thực hiện tìm kiếm mã nhị phân để xác định xem các bản vá quan trọng đã được triển khai đúng cách trong một chương trình thực thi [3]. Tìm kiếm mã nhị phân cũng là cơ sở của nhiều nhiệm vụ hiểu mã nhị phân, chẳng hạn như nhận dạng mã lặp lại và phát hiện vi phạm bản quyền [4], [5].
* Khi các kỹ thuật học máy phát triển mạnh mẽ và được áp dụng rộng rãi trong các nhiệm vụ hậu nghiệm như nhúng phần mềm [6], [7], đa số các công cụ tìm kiếm mã nhị phân đương đại nhằm huấn luyện mô hình học máy để xác định tương đồng mã nhị phân [8], [9], [10], [11]. Đặc biệt, những tiến bộ gần đây trong các mạng neural sâu (DNN) và học biểu diễn đã tạo ra phương pháp triển vọng để huấn luyện các mô hình DNN để học các biểu diễn mã lệnh tối ưu có khả năng phân biệt giữa các hàm lắp ráp tương tự nhau [12], [13], [14], [15], [16], [17].
* Để học các biểu diễn mã lệnh, mô hình DNN được huấn luyện với các đặc trưng cấp từ vựng, cấu trúc điều khiển hoặc luồng dữ liệu (nhẹ). Mặc dù việc trích xuất các biểu diễn như vậy dễ dàng, chúng có thể không bảo tồn ngữ nghĩa chương trình một cách đáng kể. Ngoài ra, các đặc trưng nhẹ thường không chịu được thử thách như tối ưu hóa trình biên dịch hay làm tròn mã, làm cho mã lắp ráp có cùng ý nghĩa dường như khác nhau đáng kể. Do đó, các mô hình DNN có thể có tính phân biệt thấp và tính ổn định thấp, dẫn đến một số lượng lớn cảnh báo sai trong các ứng viên hàng đầu lấy ra.
* Bài báo này nhằm cải thiện tìm kiếm hàm mã nhị phân bằng cách áp dụng một phương pháp cơ bản và hiệu quả. Với một hàm mục tiêu ft và một kho lưu trữ các hàm RP, ý tưởng chính của chúng tôi là sử dụng một kiểm tra tương đương giá rẻ để nhanh chóng xác định các hàm trong RP mà không tuân theo mặt nghĩa so với ft và do đó nên được loại bỏ khỏi danh sách ứng cử viên hàng đầu k được lấy ra. Nhờ đó, các hàm vượt qua kiểm tra có thể được xem xét lại để bao gồm vào danh sách ứng cử viên hàng đầu k được lấy ra. Kết quả chính của chúng tôi trong việc cải thiện các công cụ tìm kiếm hàm nhị phân dựa trên DNN được thể hiện trong Bảng 1. Tóm tắt là công việc này cung cấp một cải tiến nhất quán và đáng khích lệ cho bốn công cụ hàng đầu dựa trên DNN, mặc dù các công cụ này dựa trên các mô hình neural và kỹ thuật học khác nhau. Theo quan sát của chúng tôi, BinaryAI đạt độ chính xác cao nhất so với các mô hình còn lại. Tuy nhiên, công việc của chúng tôi vẫn xác định một không gian đáng kể cho việc cải thiện. Cụ thể, độ chính xác hàng đầu 1 của BinaryAI được cải thiện đáng kể lên đến 13,3%. Tương tự, điểm số thứ hạng đối ngược trung bình (MRR) của BinaryAI cũng được cải thiện lớn đến 17,2%. Chúng tôi trình bày đánh giá chi tiết và thảo luận trong Phần 8.
* Để thiết kế một kiểm tra tương đương giá rẻ và thực tế, chúng tôi xây dựng và kiểm tra các mối quan hệ đầu vào-ra của các hàm lắp ráp bằng cách sử dụng kỹ thuật giải ràng buộc và thực thi biểu tượng không bắt buộc (USE) [19]. So với thực thi biểu tượng tiêu chuẩn, USE được thực hiện để thực hiện luận lý biểu tượng linh hoạt và nhanh chóng trực tiếp từ các điểm vào của hàm, bỏ qua việc xử lý chi phí từ hàm main đến hàm mục tiêu. Chúng tôi tối ưu hóa lược đồ USE tiêu chuẩn như một công cụ thực tế, gọi là BinUSE, đặc biệt cho việc kiểm tra tương đương của các hàm lắp ráp. BinUSE khởi động duyệt USE từ điểm vào của hàm và duyệt qua mỗi đường dẫn cho đến khi đạt đến điểm gọi hàm bên ngoài đầu tiên, đánh dấu một nút thông tin và quan trọng trên CFG. Sau đó, BinUSE sử dụng các công thức biểu tượng của đầu vào điểm gọi hàm bên ngoài để hình thành các ràng buộc biểu tượng của mỗi đường dẫn, và để khớp hai hàm, BinUSE khám phá các ràng buộc biểu tượng phù hợp được thu thập từ mỗi đường dẫn trong mỗi hàm.



* BinUSE có những hạn chế không chính xác do ước tính quá cao không gian đầu vào hợp lệ của hàm, mô hình bộ nhớ không chính xác của việc thực thi biểu tượng được sử dụng và một số thách thức kỹ thuật như khớp kiến trúc giao nhau (xem Phần 4). Tuy nhiên, khác với việc kiểm tra tương đương mã nhị phân trước đó chỉ phân tích các khối cơ bản hoặc một lần chạy duy nhất [3], [4], [20], [21], BinUSE có khả năng mở rộng đủ lớn để kiểm tra giao nhau tất cả các hàm lắp ráp trong một cặp chương trình thực thi coreutils trong vòng 56,6 phút CPU (chỉ 25,0 giây để kiểm tra hai hàm lắp ráp). Tương tự, BinUSE mất 4.137,6 phút CPU (khoảng 2,1 giờ thực trạng đồng hồ trên máy chủ của chúng tôi với 32 lõi CPU) để kiểm tra giao nhau các hàm trong một cặp chương trình thực thi binutils. Trung bình, BinUSE mất khoảng 10,9 giây để kiểm tra hai hàm lắp ráp; lưu ý rằng các chương trình binutils có nhiều hàm hơn nhiều so với chương trình coreutils. Các cài đặt so sánh khác nhau và khó khăn được xem xét, bao gồm các trình biên dịch, tối ưu hóa, kiến trúc và việc làm tròn mã khác nhau. BinUSE đạt tỷ lệ tồi tính giả trung bình là 25,0% và tỷ lệ phủ sót giả là 4,2%. Đánh giá của chúng tôi cho thấy rằng BinUSE có thể thành công cải tiến bốn công cụ tìm kiếm mã nhị phân dựa trên DNN hàng đầu trong các cài đặt khác nhau, xác nhận tính hiệu quả và tổng quát của phương pháp. Chúng tôi tiếp tục trình bày các tối ưu hóa giúp BinUSE xử lý các chương trình thực thi binutils lớn hơn nhiều so với chương trình coreutils và cách BinUSE giúp cải thiện độ chính xác của việc tìm kiếm lỗ hổng dựa trên DNN. Tóm tắt, chúng tôi đóng góp như sau:

A diagram of a function

Description automatically generated

* Tại mức độ khái niệm, chúng tôi đề xuất một quan điểm mới để cải thiện tìm kiếm hàm mã nhị phân dựa trên DNN, hiện đang có độ chính xác thấp. Thay vì thiết kế các DNN mới (mà trong nguyên tắc khó để chính xác bắt được ngữ nghĩa), chúng tôi thiết kế một kiểm tra tương đương giá rẻ để phát hiện và loại bỏ các hàm lắp ráp mà không tuân theo mặt nghĩa so với hàm mục tiêu.
* Tại mức kỹ thuật, chúng tôi trình bày một kiểm tra tương đương bằng cách tối ưu hóa lược đồ USE tiêu chuẩn để giảm chi phí thêm nữa. Kiểm tra tương đương này được thiết kế đặc biệt cho các hàm lắp ráp, đồng thời xem xét nhiều thách thức kỹ thuật và cơ hội tối ưu hóa, ví dụ như thu thập các ràng buộc biểu tượng qua các điểm gọi hàm bên ngoài có thể đạt được từ các điểm vào của hàm để giảm độ phức tạp.
* Tại mức thực nghiệm, đánh giá của chúng tôi cho thấy rằng kiểm tra tương đương được thiết kế là tổng quát và hiệu quả để cải thiện các công cụ tìm kiếm hàm nhị phân dựa trên DNN với chi phí thấp. Kiểm tra tương đương cho thấy hiệu suất xuất sắc trong nhiều cài đặt thách thức, bao gồm việc khớp hàm tương đương tổng quát và tìm kiếm CVE. Chúng tôi đã công bố mã nguồn của BinUSE và dữ liệu đánh giá công khai để tái sản xuất tại [22].
* Chúng tôi sẽ tiếp tục duy trì BinUSE để hỗ trợ nghiên cứu trong tương lai.

2 KIẾN THỨC SƠ BẠCH

2.1 Công thức và Đánh giá

* Các công trình gần đây thực hiện tìm kiếm hàm lắp ráp có ý thức về ngữ nghĩa [6], [8], [12], [13], [14], [15], [18], [23]. Điều này có nghĩa là chúng nhằm xác định sự tương đồng ngữ nghĩa của hai hàm lắp ráp trong mã nhị phân, mặc dù hai hàm này có thể có cấu trúc cú pháp khác nhau, chẳng hạn như kết quả của tối ưu hóa trình biên dịch. Nhiệm vụ tìm kiếm hàm tương tự tương đương với việc truy xuất thông tin, khi cho trước một hàm lắp ráp mục tiêu ft và một kho lưu trữ các hàm lắp ráp RP, các công cụ tìm kiếm mã nhị phân trả về các hàm ft 2 RP hàng đầu k được xếp hạng theo độ tương đồng ngữ nghĩa với ft.
* Hình 1 minh họa một thiết lập đánh giá thường xuyên được sử dụng và khó khăn trong lĩnh vực này [6], [8], [12], [13], [14], [15], [23], trong đó chúng ta chuẩn bị hai chương trình thực thi Bin1 và Bin2 được biên dịch từ cùng một chương trình sử dụng các cài đặt biên dịch khác nhau. So sánh từng cặp hàm cho ra một điểm tương đồng, và độ chính xác hàng đầu k được định nghĩa như sau: (theo công thức trong bài báo)
* A black and white math symbols

  Description automatically generated
* Trong đó N là tổng số hàm trong chương trình. Để hiểu công thức tiêu chuẩn về độ chính xác hàng đầu k: chúng ta lần lượt lấy mỗi hàm fi trong Bin1 làm hàm mục tiêu để truy vấn RP được hình thành bởi tất cả các hàm trong Bin2. Để tính toán độ chính xác hàng đầu k, cho phép khớp chính xác (tức là đúng) của fi là f0\_i. Trong công thức 1, pk(fi) cho ra giá trị 1 nếu f0\_i nằm trong danh sách hàng đầu k ứng cử viên được lấy ra, trong khi pk(fi) = 0 nếu f0\_i không nằm trong đó. Lưu ý rằng sự khớp chính xác f0\_i (đúng) thể hiện các hàm chia sẻ chức năng giống nhau với hàm mục tiêu fi. Trong việc đánh giá trong Hình 1, thiết lập tiêu chuẩn và thông thường là coi một hàm f0\_i chia sẻ cùng tên hàm với fi là sự khớp chính xác. Một phương pháp đo thông thường khác được gọi là điểm số trung bình hạng đảo nghịch (MRR), có thể tính toán theo cách sau: (theo công thức trong bài báo):
* A mathematical equation with numbers and symbols

  Description automatically generated
* trong đó |Q| biểu thị tổng số lượng truy vấn được thực hiện đối với RP, và rank\_i đại diện cho thứ hạng của sự khớp chính xác cho truy vấn thứ i được trả về bởi công cụ tìm kiếm hàm. Điểm số MRR càng lớn thì tìm kiếm hàm càng chính xác hơn. Ví dụ, cho ba truy vấn đối với RP, trong đó các sự khớp chính xác cho mỗi truy vấn được đặt ở vị trí thứ 4, thứ 2 và thứ 6 trong danh sách ứng cử viên được lấy ra. Sau đó, MRR được tính toán là 1/3 \* (1/4 + 1/2 + 1/6) = 0.31.
* Klarafikasi tentang Cách Xây dựng Bộ dữ liệu "Mã Tương đương". Những độc giả cẩn thận có thể tự hỏi rằng theo quan điểm, các công cụ phân tích binary dựa trên DNN được thiết kế để tìm mã tương tự. Chúng tôi xác định rõ rằng thiết lập tìm kiếm hàm qua các hàm tương đương thực sự là một thiết lập thông thường được áp dụng bởi hầu hết, nếu không phải tất cả, các công trình (dựa trên DNN) trong lĩnh vực này [6], [8], [12], [13], [14], [15], [23]. Lựa chọn thiết kế này có thể được giải thích từ ba khía cạnh: 1) trong thực tế, việc tạo ra một tập lớn "mã nhị phân tương tự" rất khó khăn, trong khi mã nhị phân tương đương có thể được chuẩn bị tốt bằng cách sử dụng các cài đặt biên dịch khác nhau, 2) thử nghiệm mã tương đương có thể phản ánh hiệu suất của các công cụ trên cùng một cơ sở, và do đó, thiết lập này được sử dụng rộng rãi trong lĩnh vực này, và 3) ngoài phân tích mã độc, một số ứng dụng quan trọng và liên quan khác (ví dụ: phân tích bản vá [3]; tìm kiếm CVE) yêu cầu phù hợp chính xác các đoạn mã tương đương. Ví dụ, khi tìm kiếm lỗ hổng Heartbleed nổi tiếng, phiên bản OpenSSL 1.0.1h, mặc dù có chức năng tương tự như phiên bản 1.0.1f (vulnerable), đã được vá và không chứa Heartbleed nữa.

2.2 Kiểm tra Tương đương Mã Nhị phân

* Ngoài việc học biểu diễn dựa trên DNN phổ biến, một dòng nghiên cứu khác là thực hiện kiểm tra tương đương mã, sử dụng các mối quan hệ đầu vào-ra của chương trình được thu thập thông qua thực thi biểu tượng (SE) [4], [20], [24], [25], [26], [27]. Sau khi có các công thức biểu diễn biểu tượng đại diện cho các mối quan hệ đầu vào-ra của mã nhị phân, bộ giải ràng buộc được sử dụng để kiểm tra tính tương đương của các công thức biểu diễn biểu tượng. Kiểm tra tính tương đương đem lại sự kiên định mạnh mẽ đối với các cài đặt khó khăn như tối ưu hóa trình biên dịch và làm tròn mã, vì những cài đặt này không thay đổi mối quan hệ đầu vào-ra của chương trình.
* Ví dụ Thực hành. Bây giờ chúng tôi sẽ đưa ra một ví dụ thực hành đơn giản để minh họa cách kiểm tra tính tương đương của chương trình sử dụng SE. Xem đoạn mã dưới đây: (theo đoạn mã trong bài báo)
* A white rectangular object with black letters

  Description automatically generated
* Ở đây, a và m là hai giá trị đầu vào của hai đoạn mã lập trình tương ứng. SE đại diện cho các đầu vào dưới dạng biểu tượng tự do và diễn giải mỗi câu lệnh trong miền biểu tượng. Trong trường hợp của chúng tôi, các công thức đầu ra được hiển thị như sau: (theo đoạn mã trong bài báo)
* 
* Và tính tương đương của các đoạn mã trên có thể được kiểm tra bằng cách xây dựng ràng buộc sau đây: (theo đoạn mã trong bài báo)
* 
* trong đó bộ giải ràng buộc tìm kiếm không gian đầu vào của a và m để kiểm tra xem có hai đầu vào a và m làm cho các công thức đầu ra không tương đương. Trong trường hợp bộ giải ràng buộc không tìm thấy các giải pháp hợp lệ (tức là bộ giải ràng buộc trả về unsat), hai đoạn mã này sẽ được kiểm tra một cách nghiêm ngặt là tương đương.
* Hạn chế. Kỹ thuật được đề xuất cung cấp bằng chứng nghiêm ngặt về tính tương đương của chương trình. Tuy nhiên, SE và giải quyết ràng buộc gặp khó khăn về tính mở rộng thấp, do hiện tượng nổ đường đi, suy luận ràng buộc phức tạp và thách thức cụ thể cho phân tích nhị phân [28]. Cho đến nay, các phương pháp dựa trên kiểm tra tính tương đương thường được sử dụng cho việc so sánh các khối cơ bản hoặc các dấu vết thực thi [3], [4], [20].
* Tác vụ Phân loại Nhị phân. Chúng tôi nhận thấy rằng kiểm tra tính tương đương được kích hoạt bởi SE đại diện cho một tác vụ phân loại nhị phân. Điều này có nghĩa là thay vì tính toán một số thực biểu thị điểm tương đồng (như cách các công cụ tìm kiếm mã nhị phân dựa trên DNN thực hiện; Phần 2.1), kiểm tra tính tương đương chủ yếu trả lời một câu hỏi "có/không". Do đó, các lỗi phân loại tiêu chuẩn có thể được định nghĩa như sau:
* sai dương (FP), ám chỉ rằng các hàm lắp ráp với chức năng khác nhau được coi là tương đương.
* sai âm (FN), chỉ ra rằng các hàm lắp ráp với chức năng giống nhau được coi là không tương đương.
* Mặc dù kiểm tra tính tương đương không thể được sử dụng để tính toán trực tiếp độ chính xác hàng đầu k, chúng tôi sử dụng kiểm tra tính tương đương kích hoạt bởi USE để loại bỏ các cảnh báo sai được tạo ra bởi các mô hình DNN, như sẽ được giới thiệu trong Phần 4.
  1. Thực thi Biểu tượng Bất định hướng
* Để giải quyết vấn đề về tính mở rộng của SE, USE [19] được đề xuất để kiểm tra trực tiếp các thành phần mã tùy ý (ví dụ: một hàm) thay vì toàn bộ chương trình, từ đó giảm thiểu độ phức tạp của SE một cách có nguyên tắc và hệ thống. Ngoài ra, vì một số ràng buộc phức tạp về đầu vào không được mô hình hóa, việc giải các ràng buộc có thể trở nên dễ dàng hơn. Nghiên cứu hiện tại đã chỉ ra rằng USE cho phép kiểm tra trực tiếp các đoạn mã ẩn sâu trong chuỗi cuộc gọi [19].
* Để minh họa sự khác biệt kỹ thuật ở cấp độ cao (về mặt độ bao phủ đường đi) giữa SE và USE, Hình 2a trình bày một trường hợp phân tích một chương trình giải mã thông điệp và xác định một lỗi trong hàm decoding\_msg. Hàm chính nhận thông điệp đã được mã hóa với hàm receive\_msg và thực hiện quá trình giải mã trong một câu lệnh lặp. Thông điệp giải mã được chuyển đến hàm decoding\_msg, nơi một lỗi (đánh dấu là bug tại dòng 15 trong Hình 2a) được ẩn trong nhánh if. SE có thể bị trở ngại trong việc phân tích trường hợp đơn giản này do sử dụng tài nguyên tính toán cao và các ràng buộc dài dòng. Như được thể hiện trong Hình 2b, SE bắt đầu từ điểm nhập cửa chính để giải thích các câu lệnh của chương trình với các biến biểu tượng. Khi phân tích câu lệnh lặp, các biến biểu tượng trong bộ nhớ được tạo ra ngày càng nhiều, và mỗi giá trị biểu tượng được xử lý bởi decoding có thể tiếp tục tăng lên. Do đó, việc sử dụng bộ nhớ có thể trở nên không thực tế và các ràng buộc biểu diễn được tạo ra có thể quá phức tạp để giải quyết.
* USE giảm độ phức tạp theo một cách có nguyên tắc. Để đạt được bug, USE trực tiếp phân tích decoding\_msg. Kết quả đường đi thu được, như được hiển thị trong Hình 2c, không đặt ra ràng buộc phức tạp nào cho thông điệp giải mã msg22 và có thể tạo ra ràng buộc dễ dàng hơn nhiều để giải quyết. Phân tích toàn bộ chương trình đắt tiền hơn có thể được trì hoãn cho đến khi cần thiết.

A screenshot of a computer program

Description automatically generated

* Hạn chế. Bằng cách phân tích trực tiếp các đoạn mã mã tùy ý, USE giảm bớt ràng buộc đối với đầu vào của các đoạn mã và do đó có thể tạo ra các ràng buộc đơn giản hơn nhiều. Tuy nhiên, khi sử dụng USE để kiểm tra tính tương đương, nó có thể gây ra các kết quả sai âm bằng cách xem xét hai đoạn mã tương đương như không bằng nhau. Phần 2.2 đã giới thiệu kiểm tra tính tương đương bằng cách xây dựng ràng buộc để kiểm tra sự tồn tại của các đầu vào dẫn đến sai khác đầu ra. Hai đoạn mã có thể vượt qua kiểm tra tính tương đương trong trường hợp không có đầu vào nào gây ra sai khác đầu ra. Tuy nhiên, USE, bằng cách giảm bớt ràng buộc đối với đầu vào, có thể tìm ra các giải pháp hợp lệ mà thực tế là không hợp lệ khi xem xét đoạn đường dẫn từ chương trình chính đến các đoạn mã mục tiêu. Một lần nữa, việc tìm ra thành công một giải pháp hợp lệ có nghĩa là hai đoạn mã không vượt qua kiểm tra tính tương đương. Tổng thể, USE, trong nguyên tắc, nên cung cấp một kiểm tra tính tương đương hoàn chỉnh, hiệu quả nhưng không chính xác, tiềm năng gây ra các kết quả sai âm, điều này thường là không mong muốn. Chúng tôi sẽ tiếp tục phân tích tính hoàn thiện của ứng dụng USE tùy chỉnh của chúng tôi, được gọi là BinUSE, trong Phần 4.

2.4 Phạm vi Ứng dụng

* Độ chi tiết tìm kiếm của kỹ thuật đề xuất của chúng tôi, BinUSE, là ở mức của các hàm lập trình. Như đã được làm rõ trong Phần 2.1, quyết định thiết kế này phù hợp với hầu hết các công trình trong lĩnh vực này [6], [8], [12], [13], [14], [15], [18], [23]. Tuy nhiên, kỹ thuật cốt lõi của BinUSE, tức là USE, có thể được áp dụng chung cho các thành phần mã có độ chi tiết khác nhau [19].
* Như đã báo cáo trong Bảng 1, kiểm tra tính tương đương dựa trên USE đã được chứng minh là cung cấp một cải tiến tổng quát của tìm kiếm mã nhị phân dựa trên DNN tiên tiến. Hơn nữa, BinUSE có thể áp dụng để nâng cao hiệu suất của các công cụ tìm kiếm mã khác, miễn là chúng không sử dụng thông tin ý nghĩa chặt chẽ (ví dụ: mối quan hệ đầu vào-ra) để phù hợp. Chúng tôi tập trung vào tìm kiếm mã nhị phân dựa trên DNN, vì tìm kiếm mã nhị phân dựa trên DNN đã được chứng minh là vượt trội hơn hầu hết các phương pháp thông thường [14], [15], [16], [18]. Trong Phần 8.5, chúng tôi trình bày tính khả thi của BinUSE bằng cách cải tiến các công cụ tìm kiếm mã nhị phân theo phương pháp so sánh cấu trúc thông thường.

3. ĐỘNG LỰC NGHIÊN CỨU

* Tìm kiếm mã nhị phân dựa trên DNN phổ biến học các biểu diễn mã (tức là "nhúng mã") có thể phân biệt các thành phần mã nhị phân tương tự với các thành phần còn lại [6], [8], [9], [10], [12], [14], [17], [29]. Các phương pháp dựa trên DNN học các biểu diễn mã từ các thông tin về ngữ pháp, cấu trúc điều khiển hoặc luồng dữ liệu, chẳng hạn như hai hướng dẫn truy cập vào cùng một vị trí bộ nhớ [13]. Một mô hình DNN được đào tạo tốt sẽ chuyển đổi mẫu nhị phân đầu vào (hoặc các hướng dẫn máy) thành vector số, trong đó hai chương trình tương tự nhau nên có khoảng cách cosin gần hơn. Bằng cách chủ yếu học từ các đặc điểm "mờ" và nhẹ nhàng về dữ liệu và điều khiển, các phương pháp dựa trên DNN thể hiện tính linh hoạt và khả năng mở rộng cao, thúc đẩy phân tích các mẫu nhị phân quy mô lớn. Tuy nhiên, các biểu diễn nhúng đã học thông tin ngữ pháp, điều khiển hoặc dữ liệu không nhất thiết và chính xác đặc trưng chức năng. Tóm lại, chúng tôi cho rằng các biểu diễn nhúng đã học chủ yếu chịu hai hạn chế sau đây:
* Độ phân biệt thấp. Hạn chế này cho biết rằng các mô hình DNN có thể xem xét các hàm logic khác nhau như là tương tự nhau. Do đó, độ phân biệt thấp dẫn đến việc báo cáo nhiều kết quả phù hợp sai dương (FP).
* Độ bền thấp. Tổng thể, độ bền đề cập đến khả năng chống chịu đối với nhiều điều kiện không hoàn hảo khi chạy phần mềm hoặc thuật toán. Do đó, độ bền thấp ngụ ý rằng các mô hình DNN có thể gặp khó khăn khi phù hợp các hàm chia sẻ cùng một logic nhưng có vẻ khác nhau về mặt cú pháp. Nói chung, độ bền thấp dẫn đến việc báo cáo nhiều kết quả phù hợp sai âm (FN).
* Ví dụ động lực. Chúng tôi cho rằng độ phân biệt thấp và độ bền thấp là quan sát chung của các mô hình dựa trên DNN hiện có, điều này cũng đã được đề cập trong một số công cụ tìm kiếm nhị phân dựa trên DNN gần đây như asm2vec [14]. Tuy nhiên, các công cụ dựa trên DNN hiện có, bao gồm các phương pháp "dựa trên ý nghĩa" mới như asm2vec, mặc dù có khả năng học một cách linh hoạt các biểu diễn mã, nhưng chúng chưa thực sự hiểu rõ ý nghĩa. Phần còn lại của phần này trình bày một ví dụ động lực để minh họa độ phân biệt và độ bền tương đối thấp của các công cụ dựa trên DNN phổ biến. Bộ chương trình Linux coreutils có 105 chương trình, trong đó mỗi chương trình có trung bình 103,7 hàm. Mà không mất tính tổng quát, chúng tôi ngẫu nhiên chọn mười chương trình và sử dụng một hàm từ mỗi chương trình để khởi chạy một nhiệm vụ phù hợp hàm. Thiết lập chi tiết đã được đưa ra trong Phần 2.1, và chúng tôi đo đạc độ chính xác top-50. Chúng tôi kiểm tra bốn công cụ phổ biến dựa trên DNN, BinaryAI [15], asm2vec [14], PalmTree [18], và ncc [13]; chúng tôi thảo luận chi tiết về các mô hình này trong Phần 8.1. Đối với nhiệm vụ này, chúng tôi sử dụng gcc -O0 để biên dịch hàm mục tiêu ft và gcc -O3 để biên dịch RP.
* Hình 3 trình bày kết quả so sánh, trong đó các chấm đỏ biểu thị điểm tương đồng của việc so sánh các hàm lập trình được biên dịch từ cùng một hàm nguồn, trong khi các chấm xanh biểu thị điểm tương đồng của việc so sánh các hàm khác. Lý tưởng nhất, điểm tương đồng cao cho các chấm đỏ cho thấy tính bền của các mô hình DNN đối với tối ưu hóa trình biên dịch: các hàm lập trình được biên dịch từ cùng một hàm nguồn, mặc dù có vẻ khác nhau về cú pháp, nhưng có cùng ý nghĩa. Tương ứng, tính phân biệt được phản ánh khi các chấm xanh được phân tách tốt, có nghĩa là các hàm lập trình được biên dịch từ các hàm nguồn khác nhau có sự khác biệt đáng kể trong tầm nhìn của các mô hình DNN.

A screenshot of a graph

Description automatically generated

* Chúng tôi báo cáo rằng trong tổng số 4x10 trường hợp kiểm tra, có 20 trường hợp có các chấm đỏ nằm trong top 5 sự tương đồng cao nhất, và có 13 trường hợp có các chấm đỏ nằm ở vị trí tương đồng cao nhất (ví dụ: f9 và f10 trong đánh giá BinaryAI). Các trường hợp như f7 trong Hình 3c và f5 trong Hình 3b thậm chí nằm ở vị trí thấp nhất trong top 50 (tức là điểm tương đồng rất thấp). Tóm lại, đối với cài đặt tối ưu hóa chéo được đánh giá (gcc -O0 so với -O3), chúng tôi thấy rằng các mô hình DNN gặp khó khăn trong việc phù hợp với hàm mục tiêu với đối tượng phù hợp trong RP, cho thấy độ bền thấp đối với tối ưu hóa trình biên dịch. Mặt khác, cũng rõ ràng rằng các chấm không được phân tách tốt với nhau trong nhiều trường hợp: ví dụ, các chấm trong f7 trong Hình 3b và f6 trong Hình 3c rất gần nhau với nhau, cho thấy các mô hình DNN gặp khó khăn trong việc phân biệt các hàm lập trình với ngữ nghĩa khác nhau.
* Hình 4 trình bày một dự đoán sai: hai hàm lập trình được biên dịch từ mã nguồn khác nhau bị sai là "tương đồng". Điều này có thể do sự tương đồng của "đồ thị dòng ngữ cảnh" được trích xuất bởi ncc (xem phần 8.1 để biết chi tiết về mô hình này), mà không thể phản ánh sự khác biệt chức năng. Nghiên cứu của chúng tôi cũng cho thấy hàm rpl\_fflush và phiên bản tối ưu hóa của nó (được trình bày trong Hình 11 khi thảo luận về kết quả đánh giá ở Phần 8.5), được gán với một điểm tương đồng rất thấp. Phiên bản tối ưu hóa rpl\_fflush có một CFG đã thay đổi lớn, xuất hiện khác nhau về cấu trúc nhưng thực sự giữ nguyên cùng logic.
* Chúng tôi làm rõ rằng cài đặt biên dịch được sử dụng trong ví dụ động lực của chúng tôi đại diện cho một nhiệm vụ dễ dàng. Ngay sau đó trong phần đánh giá của chúng tôi (Phần 8.5, 8.6 và 8.7), chúng tôi cho thấy các công cụ dựa trên DNN, bao gồm BinaryAI và PalmTree, cho thấy độ chính xác tương đối tốt hơn trong ví dụ động lực này, tổng thể gặp khó khăn trong việc phù hợp với các hàm lập trình từ bộ kiểm tra Linux coreutils và binutils, và phần mềm phức tạp thế giới thực như OpenSSL và Wireshark, đặc biệt trong các cài đặt tối ưu hóa trên kiến trúc, trình biên dịch và sự che giấu chéo. Điều này cho thấy độ phân biệt và độ bền thấp là những mối quan tâm chung đối với các công cụ tìm kiếm hàm dựa trên DNN ngày nay.
* Fig. 3 đã minh họa rằng tìm kiếm mã nguồn nhị phân dựa trên DNN hiện đại học cách biểu diễn mã từ các đặc trưng thô và thể hiện tính phân biệt thấp và tính bền thấp. Do đó, thông thường các hàm phù hợp top-k có điểm tương đồng gần nhau, trong khi các hàm phù hợp thực tế có thể không có điểm tương đồng đủ cao do các cài đặt tối ưu trình biên dịch hoặc các kỹ thuật che giấu.
* Nghiên cứu sơ bộ của chúng tôi đã kiểm tra thủ công kết quả phù hợp top-k của các mô hình DNN này, và chúng tôi nghi ngờ rằng một kiểm tra tương đương đơn giản có thể hiệu quả giảm số lượng FPs (sự phù hợp sai). Ví dụ, chúng ta có thể đưa hàm mục tiêu ft và một hàm trong danh sách phù hợp top-k với cùng một đầu vào, và so sánh kết quả đầu ra của chúng để xem liệu chúng có khác nhau hay không. Những quan hệ đầu vào-đầu ra này có thể hoạt động như một bộ chỉ thị, nhanh chóng phát hiện và loại bỏ những hàm có tính năng khác nhau logic, trong khi những hàm phù hợp thực sự gradually bubble out trong quá trình tìm kiếm top-k.
* Sơ đồ tổng quan của quy trình nghiên cứu của chúng tôi được thể hiện trong Hình 5. Tóm lại, nghiên cứu này nhằm cung cấp một kiểm tra tương đương có giá thấp, đưa ra quyết định nhị phân (đúng/sai) về việc hai hàm lập trình có giống nhau hay không. Điều này giúp điều chỉnh kết quả tìm kiếm của các công cụ dựa trên DNN, giảm số lượng FPs trong quá trình tìm kiếm top-k. Ví dụ, nhiều "chấm xanh" trong ví dụ động lực của chúng tôi (Hình 3) có thể được quyết định là không giống với hàm mục tiêu bằng cách đơn giản là thực thi cả hai hàm với một đầu vào và so sánh tính tương đương của kết quả đầu ra của chúng. Với kiểm tra tương đương giá thấp như vậy, chúng ta có thể dễ dàng loại bỏ các FPs khỏi quá trình tìm kiếm top-k, từ đó tối ưu hóa các trường hợp như f7 trong Hình 3b và f9 trong Hình 3c.

A screenshot of a diagram

Description automatically generated

A diagram of a graph

Description automatically generated

* Kiểm tra tính tương đương của phần mềm nói chung là một quá trình đòi hỏi chi phí cao. Vì vậy, như một giải pháp thực tiễn, chúng tôi chấp nhận một kiểm tra tính tương đương có độ chính xác tương đối thấp, có nghĩa là hai hàm khi qua kiểm tra vẫn có thể khác nhau. Như vậy, bằng cách kết hợp DNN (Deep Neural Network) với kiểm tra tính tương đương có chi phí thấp, chúng tôi đạt được hiệu quả kết hợp thực tiễn, giúp cải thiện tốc độ dịch vụ với độ chính xác cao hơn. Trong phần tiếp theo, chúng tôi sẽ thảo luận về ưu điểm và nhược điểm của mỗi phương án kiểm tra tính tương đương dựa trên Fig. 6 và minh họa phương án kiểm tra tính tương đương đã triển khai của chúng tôi (gọi là BinUSE) trong Fig. 6e.
* Tùy chọn kiểm tra tính tương đương hàm. Hình 6 phân tích các tùy chọn tiềm năng để thiết kế kiểm tra tính tương đương của các hàm gọi là assembly. Chúng tôi bây giờ sẽ thảo luận về từng tùy chọn một cách cụ thể.
* Hình 6a: Kiểm tra tính tương đương dựa trên thực thi cụ thể. Một tùy chọn là sử dụng các giá trị được chọn ngẫu nhiên làm đầu vào và so sánh kết quả thực thi cụ thể [21], [29], [30], [31], [32]. Tuy nhiên, như được minh họa trong Hình 6a, phương pháp này có thể chỉ bao phủ một không gian đầu vào nhỏ, dẫn đến số lượng FP (kết quả dương giả) cao khi coi các hàm khác nhau là tương đương. Chúng tôi cũng làm rõ rằng việc thực thi trực tiếp các hàm gọi là assembly mà không thiết lập ngữ cảnh thực thi đúng có thể rất khó khăn [31].
* Hình 6b: Kiểm tra tính tương đương dựa trên SE. Ở phía đối diện của phổ biến, SE mô hình chính xác ràng buộc đầu vào và xây dựng kiểm tra tính tương đương trong không gian đầu vào hợp lệ. Do đó, nó nên chính xác. Tuy nhiên, SE gặp khó khăn về khả năng mở rộng, với việc thực thi của nó khó có thể đạt đến các hàm ẩn sâu trong chuỗi gọi. Trong phần 8.3, chúng tôi cho thấy tốc độ chậm của việc thực hiện SE cho toàn bộ chương trình sử dụng các phiên bản gần đây của KLEE và MOKLEE [33], [34].

A close-up of a white sheet

Description automatically generated

* Hình 6c: Kiểm tra tính tương đương dựa trên USE. USE có thể thực hiện lý luận biểu tượng tại các điểm chương trình tùy ý và nâng cao SE tiêu chuẩn bằng cách bỏ qua các tiền tố đường dẫn đắt đỏ. Tuy nhiên, việc bỏ qua tiền tố đường dẫn, như đã đề cập trong Phần 2.3, cho thấy USE không thể mô hình các ràng buộc về đầu vào của các đoạn mã mục tiêu. Do đó, USE khám phá quá nhiều không gian đầu vào. Phần 2.3 đã làm rõ rằng USE cho phép kiểm tra tính tương đương hoàn chỉnh mặc dù không chính xác, vì nó có thể tìm ra các trường hợp ngoài không gian đầu vào hợp lệ, do đó vi phạm ràng buộc kiểm tra tính tương đương được đưa ra trong Phần 2.2.
* Hình 6d: Kiểm tra tính tương đương dựa trên USE của hàm. Mặc dù khó khăn chung về xử lý không chính xác, quan sát của chúng tôi về phần mềm thế giới thực dẫn đến một giả định chính trong nghiên cứu này:

A close-up of a text

Description automatically generated

* Tổng quát, các hàm trong các chương trình thế giới thực tuân theo nguyên tắc lập trình phòng thủ [35], [36], [37], nguyên tắc này khẳng định rằng không có hàm cụ thể nào nên giả định về đầu vào của nó (ví dụ: một con trỏ được truyền bởi người gọi có thể không hợp lệ). Điều này có nghĩa là các đầu vào cho các hàm có thể là bất kỳ giá trị nào trong không gian đầu vào.
* Như được minh họa trong hình 6d, giả định này cung cấp một cơ hội duy nhất để cung cấp một "kiểm tra tương đương đáng tin cậy", đặc biệt đối với các hàm, vì khi phân tích các hàm trong phần mềm thế giới thực, không gian đầu vào hợp pháp sẽ phù hợp với không gian đầu vào hoàn chỉnh mà USE có thể khám phá. Khả năng thực hiện kiểm tra tương đương không đáng tin cậy trong thực tế nên rất nhỏ. Mặt khác, chúng tôi làm rõ rằng trong trường hợp người dùng muốn sử dụng USE để so sánh các đoạn mã tùy ý, ví dụ: một khối mã đã được vá gần đây với các phiên bản chưa được vá [38], giả định quan trọng đã đề cập ở trên sẽ không được áp dụng. Do đó, người dùng có thể phải dùng USE (với sự không đáng tin cậy) hoặc SE tiêu chuẩn (với khả năng mở rộng thấp).
* Thực tế, đánh giá của chúng tôi (Bảng 7) cho thấy rằng đối với USE được tùy chỉnh của chúng tôi (gọi là BinUSE; xem bên dưới), tỷ lệ FN trung bình rất thấp (4,2%), ngay cả khi đánh giá các thiết lập rất khó khăn như tối ưu hóa liên quan, cross-compiler, cross-architecture và obfuscation. Quan trọng hơn, chúng tôi đã phân tích thủ công các lỗi của BinUSE; chúng tôi báo cáo rằng chúng không phải do vi phạm nguyên tắc lập trình phòng thủ được đề cập ở trên. Chúng tôi liệt kê các kết quả có thể gây ra FP và FN cho BinUSE trong Phần 9. Tóm lại, nguyên tắc này được xem như được tuân theo bởi các nhà lập trình của các bộ phần mềm phổ biến như coreutils, binutils và phần mềm thế giới thực phức tạp như OpenSSL và Wireshark. Cũng thú vị để thực hiện một nghiên cứu kinh nghiệm, là một phần công việc tương lai của bài báo này, để xác định liệu phần mềm thông thường tuân theo các khái niệm lập trình phòng thủ. Với điều đó, nghiên cứu ban đầu của chúng tôi cho thấy rằng thiết kế USE tiêu chuẩn tiếp tục gặp khó khăn về tốc độ phân tích (xem đoạn Baseline ở Phần 5). Do đó, chúng tôi yêu cầu tối ưu hóa thêm về tốc độ, như đã thảo luận dưới đây.
* Fig. 6e mô tả phương pháp BinUSE, được sử dụng trong nghiên cứu này, là một phương pháp sử dụng USE, đặt tên là BinUSE, nơi chúng tôi tối ưu hóa thêm cho tốc độ, đặc biệt là trong việc kiểm tra tương đương của các hàm thủ tục. BinUSE khởi chạy việc duyệt USE từ điểm vào của mỗi hàm thủ tục để khám phá từng đường đi. Khi duyệt qua một đường đi, nó dừng lại khi gặp điểm gọi hàm bên ngoài đầu tiên. Trong nghiên cứu này, chúng tôi giả định rằng các điểm gọi hàm bên ngoài (ví dụ: gọi hàm đến các hàm của thư viện libc) đại diện cho các nút cấu trúc quyết định trên CFG. BinUSE tính toán các công thức tượng trưng của đầu vào tại các điểm gọi hàm bên ngoài để tạo thành các ràng buộc tượng trưng của mỗi đường đi. Để so sánh hai hàm, BinUSE khám phá các ràng buộc tượng trưng khớp thu thập từ mọi đường đi trong mỗi hàm. BinUSE không đảm bảo tính chính xác. Tuy nhiên, tỷ lệ FN trung bình rất thấp theo kết quả thực nghiệm của chúng tôi. Hơn nữa, BinUSE cho phép thực hiện các kiểm tra với chi phí thấp trong các tình huống khó khăn khác nhau (ví dụ: kiến trúc chéo). BinUSE có thể hoàn thành kiểm tra hai tệp thực thi coreutils trong vòng 56,6 phút CPU (trung bình 25 giây cho mỗi cặp hàm), bao gồm tất cả các tác vụ thực thi tượng trưng và giải các ràng buộc. Điều này cho thấy tính ứng dụng thực tế của nó trong sản xuất.

5. THIẾT KẾ CỦA BINUSE

* Phần này giới thiệu thiết kế của BinUSE, một công cụ kiểm tra tính tương đương dựa trên USE. Các thủ tục để điều chỉnh việc truy xuất hàng đầu-k của một mô hình DNN bằng BinUSE sẽ được trình bày trong Mục 6. Hình 7 mô tả quy trình làm việc tổng quan của BinUSE.
* Nhìn chung, khi có một tệp thực thi đầu vào, BinUSE đầu tiên thực hiện kỹ thuật khôi phục thông tin hàm hợp ngữ (xem Mục 2.4 để biết các giả định về kỹ thuật khôi phục ngược). Sau đó, nó bắt đầu từ điểm nhập của mỗi hàm hợp ngữ để khởi chạy việc duyệt USE từng đoạn đường (Hình 7a), trong đó mỗi đoạn đường dừng lại khi đạt đến điểm gọi bên ngoài đầu tiên. Kết quả, một đồ thị con sẽ được tạo ra, trong đó mỗi nút lá tương ứng với một điểm gọi bên ngoài. Để so sánh hai hàm hợp ngữ, BinUSE so sánh các đồ thị con được tạo ra từ chúng (Hình 7b), bằng cách khởi chạy giải quyết ràng buộc để kiểm tra tính tương đương về mặt ngữ nghĩa của đầu vào điểm gọi bên ngoài và ràng buộc đoạn đường (xem chi tiết ở các Mục 5.3 và 5.4). Trước khi trình bày về thiết kế của BinUSE, chúng tôi xác định rõ giả định của mình về kỹ thuật khôi phục ngược và cơ sở ở Mục 5.1.
* Giả định về Kỹ thuật khôi phục ngược. Phân tích của chúng tôi tập trung vào cấp độ hàm, và không giả định sự có mặt của ký hiệu chương trình hoặc thông tin gỡ lỗi. Nhị phân đã bị cắt giảm có thể được xử lý mà không gặp khó khăn bổ sung, miễn là các hàm đã được xác định để sử dụng. Phân tích của chúng tôi cũng là độc lập với nền tảng; chúng tôi đã đánh giá ba cài đặt chéo kiến trúc cho kiến trúc x86 64-bit, x86 32-bit và ARM. Chúng tôi cũng đánh giá các trình biên dịch khác nhau (gcc và clang), các cấp độ tối ưu hóa và các phương pháp mờ hóa thông dụng. BinUSE cho thấy kết quả khá tích cực đối với tất cả các thiết lập này. Tuy nhiên, chúng tôi làm rõ rằng nhiệm vụ tìm kiếm hàm hợp ngữ theo bản chất có thể bị trở ngại bởi việc nội suy hàm; nếu một hàm truy vấn được nội suy vào các người gọi của nó, việc tìm kiếm cấp độ hàm có thể bị ảnh hưởng. Chúng tôi làm rõ rằng đây là một hạn chế phổ biến trong dòng công việc này, không chỉ riêng BinUSE.

5.1 Cơ sở (Baseline)

* Để thực hiện việc kiểm tra tính tương đương của hàm, phương pháp cơ sở là thực hiện phân tích trong chính hàm từ điểm nhập và lặp qua mọi đoạn đường thực thi. Các ký hiệu không ràng buộc được tạo ra mỗi khi tải dữ liệu từ dữ liệu không xác định, bao gồm các tham số hàm, dữ liệu toàn cục và các vùng bộ nhớ khác. Sau đó, chúng tôi thu thập các công thức ký hiệu đầu ra của thanh ghi CPU và bộ nhớ tại điểm thoát của một đoạn đường để xây dựng một mối quan hệ đầu vào-ra của hàm. Đây là một thiết lập tiêu chuẩn trong dòng công việc này [20], và được hình dung theo mô hình USE tiêu chuẩn trong Hình 6d.
* Chúng tôi đã thử nghiệm phương pháp này; trong khi thiết kế này cho phép mô hình hóa đầy đủ các mối quan hệ đầu vào-ra, nghiên cứu ban đầu của chúng tôi đã chỉ ra khả năng thấp về tỷ lệ mở rộng. Cây CFG (Control Flow Graph) nội hàm được tạo ra thường rất phức tạp. Ngoài ra, chúng tôi đã triển khai thuật toán CoP tiêu chuẩn [4] để giảm độ phức tạp bằng cách trích xuất dãy con chung dài nhất (LCS) của các khối cơ bản tương đương trên một đoạn đường. Tuy nhiên, một số chi tiết thực thi (chẳng hạn như phân chia và hợp nhất khối) vẫn còn chưa biết. Trong khi phương pháp USE chủ yếu giảm độ phức tạp của toàn bộ phân tích SE, nghiên cứu thử nghiệm của chúng tôi trong việc phân tích các chương trình thực thi trong thế giới thực đã chỉ ra nhu cầu đòi hỏi phải hiệu chỉnh và tối ưu hóa thêm.

5.2 Tạo đồ thị con từ CFG

* Xem xét khó khăn trong việc khám phá toàn bộ đồ thị điều khiển của một hàm, chúng tôi trích xuất một đồ thị con. Đồ thị con được trích xuất nên giữ lại các đặc trưng đại diện của CFG tương ứng và giảm đáng kể độ phức tạp phân tích. Trong nghiên cứu này, chúng tôi coi các điểm gọi hàm bên ngoài là các nút đại diện và quan trọng trên CFG. Từ góc nhìn toàn diện, hầu hết các tối ưu hóa của trình biên dịch và các phép rối tinh vi được thiết kế để thực hiện các biến đổi nội hàm [39]; các điểm gọi hàm bên ngoài, chủ yếu được tạo thành bởi việc gọi thư viện liên kết động, không nên thay đổi. Sắp tới ở Phần 5.5, chúng tôi sẽ giới thiệu một tối ưu hóa khác để xem lệnh trả về (ret) là các nút đại diện khác trong đồ thị con này.
* BinUSE được thiết kế để duyệt qua mọi đoạn đường thực thi từ điểm nhập của mỗi hàm mã lệnh. Khi gặp một vòng lặp, như một giải pháp thường dùng, chúng tôi giãn vòng lặp (với hệ số giãn vòng lặp hiện được đặt là 5). Khi phân tích một đoạn đường thực thi, BinUSE nạp chồng mỗi điểm gọi hàm trên đoạn đường đó mà có triển khai trong mã nhị phân. BinUSE dừng lại khi gặp điểm gọi hàm bên ngoài đầu tiên. Tương thích với SE tiêu chuẩn, chúng tôi tạo ra các ký hiệu không ràng buộc để đại diện cho các giá trị được lưu trữ trong các thanh ghi hoặc ô nhớ khi chúng được truy cập lần đầu tiên và giá trị ký hiệu được lưu trữ không xác định. Khi gặp một điểm gọi hàm bên ngoài, chúng tôi thu thập các công thức ký hiệu cho mỗi đầu vào của cuộc gọi hàm (xem phần sau về thông tin phục hồi hàm nguyên mẫu) để hình thành "đầu ra" của đoạn đường này. Chúng tôi cũng ghi lại các ràng buộc đoạn đường là tiên quyết để đạt được điểm gọi hàm bên ngoài này.

A close-up of a computer code

Description automatically generated

5.3 So sánh hai đồ thị con

* Hình 8 mô tả các quy trình cần thiết để so sánh hai đồ thị con Gt và Gs tương ứng được xuất phát từ hàm mã lệnh ft và fs. Như được thể hiện trong Hình 8a, việc duyệt qua CFG và tiêu chuẩn dừng được đề xuất trong Phần 5.2 tạo thành một đồ thị con với mức độ phức tạp hợp lý từ mỗi CFG, với mỗi lá tượng trưng cho một điểm gọi hàm bên ngoài. Kết quả là việc so sánh hai CFG trở thành việc so sánh hai đồ thị con này: chúng ta so sánh từng điểm gọi hàm theo từng bước cho đến khi tìm ra một hoán vị làm cho các điểm gọi hàm bên ngoài trong Gt và Gs cùng tương đương với nhau. Lưu ý rằng chúng tôi chỉ cho phép một tập con của các điểm gọi hàm trong Gt phù hợp với một tập con khác của các điểm gọi hàm trong Gs. Lý do là các tối ưu hóa trình biên dịch đôi khi có thể loại bỏ các cuộc gọi hàm của thư viện C, do đó cho phép một tập con của các cuộc gọi thư viện phù hợp với một tập con khác của cuộc gọi thư viện không bỏ sót việc phù hợp Gt với Gs được tối ưu hóa cao. Ngược lại, hai hàm được xem là không tương đương nếu không tìm thấy hoán vị nào.
* Trong khi so sánh từng cặp điểm gọi hàm trong Gt và Gs có thể đưa ra nhiều hoán vị, chúng tôi làm rõ rằng chúng tôi chỉ tiến hành kiểm tra tương đương nặng (xem Hình 8d) khi hai điểm gọi hàm tham chiếu đến cùng một hàm bên ngoài. Tên hàm bên ngoài có thể được lấy từ mã nhị phân ELF ngay cả khi đã loại bỏ. Ví dụ, chúng tôi sẽ so sánh hai điểm gọi hàm trong Gt và Gs trong trường hợp chúng đều tham chiếu đến fopen.
* Thay thế cuộc gọi thư viện C. Khi một số tham số đầu vào của cuộc gọi thư viện C là hằng số, trình biên dịch có thể thay thế cuộc gọi thư viện C này bằng các cuộc gọi khác [40]. Ngoài ra, trình biên dịch đôi khi thay thế một cuộc gọi thư viện C thông thường bằng một phiên bản an toàn hơn. Ví dụ, bằng cách thay thế printf bằng \_\_printf\_chk, tràn ngăn xếp được phát hiện trước khi tính kết quả [41].
* Việc tìm các điểm gọi thư viện phù hợp được coi là một bước quan trọng trong quá trình kiểm tra tính tương đương, và để thực hiện điều này, chúng tôi đã thu thập thủ công danh sách sau đây, trong đó mỗi mục nhập cũng là một danh sách L. BinUSE được xây dựng để xem xét các điểm gọi thư viện trong mỗi L như là như nhau. ***Aaaaaaaaaaaaaaaaaaaaaaaaa*** Ví dụ, ngoài việc so sánh hai điểm gọi printf (Phần 5.3), chúng tôi cũng coi một điểm gọi printf và \_\_printf\_chk là như nhau và tiếp tục xây dựng ràng buộc cho quá trình kiểm tra tính tương đương. Theo kiến thức của chúng tôi, danh sách này bao gồm tất cả các thay thế thư viện C có thể có được phát hiện trong các trường hợp kiểm thử của chúng tôi, bao gồm bộ kiểm thử Linux coreutils và binutils, cũng như cơ sở dữ liệu CVE bao gồm các hàm có lỗ hổng từ phần mềm phức tạp như OpenSSL, Wireshark, bash và ffmpeg. Do đó, chúng tôi cho rằng danh sách sau đây là toàn diện và có thể có lợi cho nghiên cứu trong lĩnh vực này trong tương lai.

A screen shot of a computer code

Description automatically generated

* Tính toán các điểm số tương quan định lượng. Giả sử có n đường dẫn trong Gt (mỗi đường dẫn kết thúc tại một điểm gọi bên ngoài), và chúng ta xác định p đường dẫn có các điểm gọi bên ngoài có cùng nghĩa trong Gs (xem Phần 5.4), một điểm số s thể hiện sự tin tưởng để coi hàm lắp ráp ft và fs là tương đương được tính toán là p/n. Điểm số tin tưởng này sẽ được lưu trữ và sử dụng khi hiệu chuẩn kết quả của các mô hình DNN. Cho biết điều này, chúng tôi làm rõ rằng khi phân tích các tệp thực thi coreutils, 87.9% trường hợp có điểm số tin tưởng là 1.0, cho thấy trong phần lớn trường hợp, tất cả các đường dẫn trên Gf được kết hợp với Gs.

A table with numbers and text

Description automatically generated

Phần 5.4 So sánh hai điểm gọi:

* Để so sánh hai điểm gọi trong mã máy, trước tiên chúng ta phục hồi các nguyên mẫu hàm để xác định số lượng tham số cần trích xuất. Để thực hiện điều này, chúng tôi tận dụng một công cụ phân tích mã máy thương mại, IDA-Pro [42], để phục hồi thông tin nguyên mẫu hàm. Mặc dù nghiên cứu trước đây đã chỉ ra hỗ trợ không đủ của IDA-Pro trong việc phục hồi thông tin hàm [43], nhưng chúng tôi phát hiện rằng thông tin về hàm trong thư viện chuẩn C được duy trì tốt trong cơ sở dữ liệu FLIRT [44] của IDA-Pro. Do đó, sử dụng IDA-Pro (phiên bản 7.3) đảm bảo tính chính xác của việc phân tích các điểm gọi bên ngoài đến mức đáng tin cậy. Tuy nhiên, chúng tôi đồng ý rằng nếu các tệp thực thi chứa một số cuộc gọi thư viện được định nghĩa bởi người dùng (hoặc bên thứ ba), FLIRT không thể xử lý chúng. Việc phục hồi thông tin như vậy yêu cầu suy diễn về số lượng tham số hàm; các tiến bộ gần đây trong việc phục hồi thông tin nguyên mẫu hàm có thể được tham khảo [43], [45], [46]. Cho một điểm gọi với N tham số, chúng tôi trích xuất N công thức biểu tượng theo quy ước gọi trên các kiến trúc tương ứng. Ví dụ, chúng tôi tải N ô nhớ từ đỉnh của ngăn xếp cho các kiến trúc x86 32-bit.
* Hoán đổi tham số cuộc gọi. Như được thể hiện trong Hình 8b, để kiểm tra hai điểm gọi, chúng tôi tìm kiếm một hoán vị của các cặp tham số hàm. Chúng tôi lưu ý rằng thay vì đơn giản so sánh tham số thứ i Argi t và Argi s giữa hai điểm gọi, chúng tôi cho phép một thiết kế cẩn trọng hơn bằng cách tìm kiếm sự tồn tại của một hoán vị. Ví dụ, hai điểm gọi của fopen trong Hình 8b sẽ được coi là tương đương, trong trường hợp filenamet khớp với modes và filenames khớp với modet. Lưu ý rằng thiết kế này làm cho quá trình kiểm tra tính tương đương của chúng tôi cẩn thận hơn, đồng thời đối phó tốt với tối ưu hóa trình biên dịch, các kỹ thuật che giấu tiềm năng, nhưng có thể làm tăng khả năng xảy ra các kết quả sai tích cực. Như đã đề cập trước đó, các tối ưu hóa của trình biên dịch có thể thay thế một số cuộc gọi hàm thư viện thành các cuộc gọi hàm khác, trong đó các tham số hàm có thể được di chuyển hoặc loại bỏ. Xem xét hai hàm thư viện C sau đây:
* A close-up of a sign

  Description automatically generated
* Trong trường hợp trình biên dịch tối ưu hóa, có thể thay thế một điểm gọi của hàm dgettext bằng gettext khi domainname là hằng số. Để ghép nối một điểm gọi của hàm dgettext với một điểm gọi của hàm gettext trong mã được tối ưu hóa cao, chúng ta cần phải ghép nối tham số thứ hai trong hàm dgettext với tham số đầu tiên trong hàm gettext.
* Trường hợp trên cũng cho thấy quan sát rằng hai điểm gọi hàm "giống nhau" có thể có số lượng tham số khác nhau. Để giải quyết điều này, hiện tại, chúng tôi đánh giá hai điểm gọi hàm là tương đương bằng cách thiết lập một ngưỡng g, sao cho g tham số trong điểm gọi đầu tiên nên được ghép nối với các tham số tương ứng trong điểm gọi khác. Như vậy, chúng tôi thực tế làm giảm các rào cản gây ra bởi số lượng tham số hàm không nhất quán. Tổng thể, chúng tôi thiết kế BinUSE một cách liên tục và thận trọng để đối phó với các rào cản thực tế khác nhau, chủ yếu loại bỏ khả năng sai âm tích cực. Lưu ý rằng, dựa trên số lượng tham số trong hai điểm gọi, chúng tôi luôn sử dụng số nhỏ hơn làm số chia để tính tỷ lệ r ≥ 1 và so sánh với g. g được xác định dựa trên thực nghiệm là 0.5 trong hiện thực hiện tại của chúng tôi.
* Phương pháp hoán vị tham số gợi ý một thiết kế cẩn trọng hơn và đáng tin cậy đối với các tối ưu hóa trình biên dịch khác nhau có thể thay đổi các điểm gọi (một ví dụ được đưa ra ở trên). Tuy nhiên, nếu hai điểm gọi hàm liên quan đến nhiều tham số, số lượng hoán vị để kiểm tra sẽ tăng theo cấp số mũ.
* Quan sát thực nghiệm của chúng tôi cho thấy phương pháp hoán vị này không tạo ra quá nhiều chi phí phụ, vì hầu hết phần mềm được sử dụng phổ biến có số lượng tham số hàm hạn chế. Để củng cố quyết định thiết kế này, chúng tôi cung cấp một phân phối số lượng tham số cho tất cả các điểm gọi bên ngoài gặp phải trong các trường hợp kiểm thử của chúng tôi trong Bảng 2. Lưu ý rằng dữ liệu phân phối trong Bảng 2 được thu thập từ việc phân tích tất cả các chương trình đã được đánh giá của chúng tôi, bao gồm các bộ chương trình Linux coreutils và binutils, cũng như cơ sở dữ liệu CVE chứa các hàm có lỗ hổng trong phần mềm phức tạp thế giới thực như OpenSSL, ffmpeg, Wireshark (xem chi tiết trong Phần 8). Từ các kết quả thực nghiệm này, có thể quan sát rằng hầu hết các điểm gọi bên ngoài có ít hơn hoặc bằng 2 tham số, và gần như tất cả các điểm gọi bên ngoài có ít hơn hoặc bằng 5 tham số. Do đó, chúng tôi cho rằng quyết định thiết kế của chúng tôi về việc hoán vị các tham số không làm tăng chi phí đáng kể, nhưng có thể giúp làm cho thiết kế tổng thể của BinUSE cẩn trọng và đáng tin cậy hơn.
* Kiểm tra tính tương đương của hai tham số. Tiếp theo, chúng tôi xây dựng ràng buộc cho việc kiểm tra tính tương đương của một cặp tham số (xem Hình 8d). Hãy cho PCt và PCs là các ràng buộc đường dẫn thu thập từ điểm nhập của các hàm lắp ráp đã được phân tích đến hai điểm gọi bên ngoài này. Hãy cho Argi
* t và Argj
* s là tham số thứ i và j của hai điểm gọi bên ngoài này. Một cách hình thức, chúng tôi kiểm tra..
* A black and white math equation

  Description automatically generated
* Trong đó, X là danh sách các ký hiệu được sử dụng bởi Argi\_t hoặc PCt. Tương tự, Y là danh sách các ký hiệu được sử dụng bởi Argj\_s hoặc PCs. Như được hiển thị trong Hình 8c, chúng tôi kiểm tra xem liệu có tồn tại một hoán vị p(Y) mà sẽ ghép cặp phù hợp giữa các phần tử trong X dưới sự kết hợp của ràng buộc đường dẫn PCt(X) ^ PCs(Y) để làm cho ràng buộc trên trở nên không thỏa mãn. Trong trường hợp tìm thấy hoán vị như vậy, Argi\_t và Argj\_s được kiểm tra xem có tương đương nhau hay không. Chúng tôi lưu ý rằng việc hoán vị có thể khiến BinUSE trở nên thận trọng hơn (giảm FNs) và đáng tin cậy hơn đối với các thay đổi do sự kết hợp giữa các kiến trúc, trình biên dịch và kỹ thuật làm mờ. Tuy nhiên, quyết định thiết kế này có thể tiềm ẩn thêm FPs. Chúng tôi xem đây là một sự cân nhắc thiết kế để hiệu chỉnh tỷ lệ FP/FN.
* Ngoài việc sử dụng hoán vị, ở bước này chúng tôi cũng chuẩn hóa hằng số trong các công thức biểu tượng nếu nó đại diện cho một địa chỉ bộ nhớ. Chúng tôi thực hiện một phương pháp đơn giản để xác định hằng số biểu thị cho một địa chỉ bộ nhớ. Đầu tiên, khi thực hiện USE từng bước đường dẫn, chúng tôi đánh dấu một hằng số là địa chỉ bộ nhớ khi chúng tôi nhận thấy nó được sử dụng để xây dựng địa chỉ cơ sở của các con trỏ mã lệnh. Thứ hai, chúng tôi đưa ra một giả định được chia sẻ bởi nhiều công cụ phân tích tĩnh tiên tiến [47], [48], [49] rằng một hằng số sẽ được coi như là một địa chỉ bộ nhớ nếu nó trỏ đến các phần tử dữ liệu hoặc văn bản của một chương trình thực thi định dạng ELF.
* Các ký hiệu (X và Y) được thu thập bằng cách duyệt qua các công thức biểu tượng và ràng buộc đường dẫn đã xây dựng. Mỗi lần kiểm tra ràng buộc trên, chúng tôi đặt thời gian giới hạn cho bộ giải SMT (Satisfiability Modulo Theories), cụ thể là Z3 [50], là N giây. Trong trường hợp bộ giải SMT trả về kết quả "unsat" (không thỏa mãn), hoặc không thể tìm ra một giải pháp "sat" (thỏa mãn) trong N giây, hai đối số của cuộc gọi hàm được coi là tương đương. N được thiết lập là 15 giây cho phiên bản hiện tại của chúng tôi, và quan sát của chúng tôi cho thấy hầu hết các tác vụ có thể hoàn thành trong ngưỡng này. Thiết lập thời gian giới hạn có thể làm cho các đối số không tương đương được coi là tương đương (tức là FPs), vì không tìm thấy các phản ví dụ trong N giây. Tuy nhiên, thiết lập thời gian giới hạn không giới thiệu FN (kết quả sai âm), mà có thể tăng tốc quá trình phân tích các mẫu nhị phân với quy mô lớn.

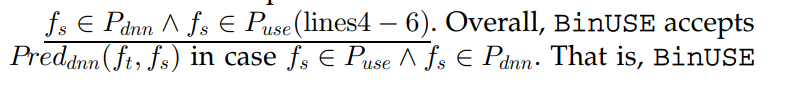
5.5 Tối ưu hóa: Thu thập và So sánh Ràng buộc đường dẫn trên các Đoạn đường không có Điểm gọi bên ngoài

* Mặc dù chiến lược duyệt được giới thiệu trong phần này có thể bao phủ hầu hết các trường hợp thực tế (xem thảo luận trong Phần 8.2), nhưng một số trường hợp góc cạnh có thể tồn tại làm trở ngại đối với phân tích của chúng tôi. Cụ thể, có thể không có điểm gọi bên ngoài nào trên một đoạn đường thực thi. Trong những trường hợp như vậy, thay vì đơn giản bỏ qua phân tích đoạn đường này, chúng tôi thu thập các ràng buộc đường dẫn PC từ điểm vào hàm cho đến khi đạt đến lệnh trả về (ret) ở cuối đoạn đường thực thi. Tuy nhiên, nếu khi duyệt qua đoạn đường này chúng tôi không thể xây dựng bất kỳ điều kiện đường dẫn nào, chúng tôi bỏ qua việc so sánh đoạn đường này. Từ góc nhìn toàn diện, mỗi lệnh trả về ret sẽ được xem như một "điểm gọi bên ngoài" đặc biệt không có tham số, và để quyết định liệu hai đoạn đường không có điểm gọi bên ngoài có thể khớp nhau, chúng tôi sử dụng ràng buộc sau đây để kiểm tra các ràng buộc đường dẫn PCt và PCs liên quan đến chúng: [các ràng buộc PCt ^ PCs sẽ được mô tả ở đây nếu có]
* A black and white text

  Description automatically generated
* Trong đó, X là danh sách các ký hiệu được sử dụng bởi PCt và Y là danh sách các ký hiệu được sử dụng bởi PCs. Cụ thể, chúng ta kiểm tra xem có hoán vị p(Y) nào sao cho phù hợp một cách song song với X để làm cho ràng buộc trên không thỏa mãn. Nếu không, hai ràng buộc đường dẫn sẽ được kiểm tra xem có tương đương nhau hay không.
* Tối ưu hóa này sẽ mở rộng đồ thị con được tạo ra trong quá trình duyệt biểu tượng của chúng tôi (Phần 5.2) với các nút bổ sung, biểu thị các ràng buộc đường dẫn được thu thập từ các đoạn đường thực thi không có điểm gọi bên ngoài. Sau đó, để so sánh hai đồ thị con Gt và Gs được tạo ra từ hàm assembly ft và fs (Phần 5.3), chúng tôi tiếp tục so sánh song song các nút trên Gt và Gs biểu thị các điểm gọi bên ngoài, và cũng so sánh các nút trên Gt và Gs biểu thị các đoạn đường không có điểm gọi bên ngoài bằng cách sử dụng ràng buộc như đã nêu ở trên.
* Theo kết quả thực nghiệm của chúng tôi, tối ưu hóa này có thể tạo điều kiện thuận lợi cho việc so sánh một số đoạn đường không có điểm gọi bên ngoài và cải thiện độ chính xác của các công cụ dựa trên DNN (mạng nơ-ron sâu). Chúng tôi so sánh độ chính xác tốt nhất cải tiến của bốn mô hình DNN được đánh giá (chi tiết về mô hình được xác định rõ trong Phần 8.1) trong Bảng 3. Chúng tôi sử dụng bộ kiểm tra coreutils của Linux để so sánh và độ chính xác được tính bằng cách lấy trung bình của tất cả 12 cài đặt so sánh được đánh giá trong nghiên cứu này (xem Bảng 7 để biết chi tiết các cài đặt này). Tóm lại, sau khi sử dụng tối ưu hóa được đề xuất trong phần này, BinUSE có thể cải thiện độ chính xác của các mô hình DNN khoảng 2,45% trung bình. Chúng tôi cũng thảo luận về một số trường hợp góc cạnh khác mà chúng tôi đã gặp và cách làm việc xung quanh trong Phần 9.1.
* A table with numbers and text

  Description automatically generated
* Cơ hội tối ưu hóa khác. Đáng lưu ý rằng chúng tôi cũng đã thử nghiệm các phương pháp tối ưu hóa khác. Ví dụ, hiện tại, lược đồ USE tùy chỉnh được sử dụng bởi BinUSE không xem xét ảnh hưởng của giá trị trả về từ các điểm gọi bên ngoài. Giả sử rằng các ràng buộc đầu vào/đầu ra của một điểm gọi bên ngoài (ví dụ, một điểm gọi tới hàm chuẩn libc strcpy) đã được mô hình hóa bởi BinUSE, thì việc tiếp tục thực thi biểu tượng với các giá trị trả về đã được mô hình hóa là khả thi. Từ góc nhìn toàn diện, điều này có thể yêu cầu xem xét thêm ràng buộc trên các giá trị trả về của các hàm bên ngoài trong quá trình kiểm tra tương đương, điều này có thể giúp cải thiện tỷ lệ kết quả dương thật. Mặc dù có tiềm năng hứa hẹn của tối ưu hóa này, chúng tôi xác định rõ ràng rằng việc mô hình quan hệ đầu vào/đầu ra của các điểm gọi bên ngoài thường rất khó, và có thể đòi hỏi nỗ lực thủ công đáng kể để mã hóa thông tin cố định như vậy cho BinUSE. Chúng tôi cũng nhận thấy rằng các trình thực thi biểu tượng phổ biến như KLEE đã mã hóa các ràng buộc đầu vào/đầu ra như vậy cho một số hàm chuẩn libc, điều này có thể được tận dụng để cải thiện BinUSE. Tuy nhiên, một vấn đề khó khăn, và có thể là không thể, đó là mã hóa thông tin như vậy cho một số cuộc gọi thư viện do người dùng xác định (hoặc bên thứ ba xác định).

1. CẢI THIỆN CÔNG CỤ DỰA TRÊN DNN

* Bây giờ chúng ta sẽ thảo luận về các chiến lược để cải thiện danh sách các ứng viên xếp hạng từ công cụ tìm kiếm hàm mã nhị phân dựa trên DNN. Hãy nhớ lại định nghĩa vấn đề trong Phần 2.1, trong đó, cho trước một hàm mục tiêu ft, chúng ta so sánh ft với các hàm trong kho dữ liệu fs thuộc RP (tập hàm trong kho dữ liệu), và quyết định sự tương đồng logic giữa từng cặp hàm. Giả sử các mô hình DNN so sánh ft với từng fs thuộc RP và mỗi lần so sánh trả về một điểm tương đồng Preddnn(ft, fs) trong khoảng [0, 1]. Hãy ký hiệu Pdnn là tập các ứng viên được xếp hạng hàng đầu (số lượng phần tử của Pdnn là k) và Ndnn = RP n Pdnn. Tương tự, sau khi kiểm tra ft với từng fs thuộc RP, BinUSE trả về một tập hàm thông thường có số lượng nhỏ là Puse, các cặp hàm này được kèm theo một điểm tin cậy Preduse(ft, fs) trong khoảng [0, 1]. Hãy ký hiệu Nuse là tập dự đoán âm, tức là Nuse = RP n Puse.
* Thuật toán 1 trình bày thuật toán cải thiện việc tìm kiếm hàm mã nhị phân dựa trên DNN với BinUSE, dựa trên các ký hiệu đã nêu ở trên. Để dễ dàng trình bày, giả sử Pdnn là một cấu trúc dữ liệu bản đồ (map), với các khóa là các hàm hàng đầu và các giá trị tương ứng là các điểm tương đồng. Tương tự, Puse ánh xạ mỗi hàm trong kho dữ liệu đến kết quả kiểm tra tương đương (điểm tin cậy) với ft. Với một hàm fs thuộc RP, thuật toán 1 xử lý ba tình huống sau để cải thiện việc truy xuất hàng đầu k:
* 
* Khi các mô hình DNN và công cụ BinUSE đạt được một thỏa thuận và xem xét fs là một trong số các ứng viên hàng đầu, fs sẽ được giữ trong tập Pdnn hàng đầu k. Trong Thuật toán 1, fs được lưu giữ trong một bản đồ tạm thời P cùng với điểm tin cậy s0 và điểm tương đồng s tương ứng trong Pdnn và Puse, tương ứng (dòng 6).

A white paper with black text and black text

Description automatically generated

* s thuộc Pdnn ^ fs thuộc Nuse (dòng 7-8). Trong những trường hợp như vậy, chúng tôi loại bỏ fs khỏi tập Pdnn hàng đầu k. Lý do là BinUSE tạo ra số lượng FNs thấp, và trong trường hợp fs thuộc Nuse, chúng tôi có độ tin cậy cao rằng fs không thuộc Pdnn. Lưu ý rằng fs vẫn có thể được truy xuất trong top-k, trừ khi chúng tôi tìm thấy các trường hợp thỏa mãn điều kiện trong dòng 9-11 và di chuyển chúng từ Ndnn vào Pdnn. Do đó, chúng tôi vẫn đưa fs vào P, mặc dù "điểm tin cậy" để BinUSE ghép fs và hàm mục tiêu là 0.0, như được mô tả ở dòng 8.
* fs thuộc Ndnn ^ fs thuộc Puse (dòng 9-11). Vì Puse có thể gây ra FPs do sự xấp xỉ quá, chúng tôi dựa vào một ngưỡng đã được đào tạo trước, a, để chấp nhận Puse một cách lựa chọn. fs sẽ được thêm vào P chỉ khi điểm tin cậy liên quan Preduse(ft, fs) ≥ a (dòng 10). Chúng tôi quyết định a dựa trên thực nghiệm là 0.41 trong phiên bản hiện tại. Xem các thảo luận liên quan trong Phần 7.
* P ánh xạ các hàm fi vào điểm tin cậy liên quan của chúng s0i (do BinUSE quyết định) và điểm tương đồng si (do các mô hình DNN quyết định). Ở dòng 12, chúng tôi sắp xếp lại các hàm trong P theo thứ tự giảm dần. Điều đó có nghĩa là hàm fi sẽ được đặt trước fj trong P nếu điểm tin cậy s0i > s0j, và nếu s0i = s0j, chúng tôi tiếp tục so sánh điểm tương đồng để xem nếu si > sj. Cuối cùng, chúng tôi trả về các phần tử hàng đầu k trong P ở dòng 13.

1. THỰC HIỆN

* Chúng tôi thực hiện BinUSE với khoảng 5.500 dòng mã Python dựa trên framework phân tích nhị phân angr [51]. Bằng cách liên kết với hệ sinh thái angr phổ biến, mà nâng mã lệnh thành ngôn ngữ trung gian VEX không phụ thuộc nền tảng, BinUSE có thể xử lý các tệp thực thi từ các kiến trúc khác nhau. Quan trọng hơn, angr cung cấp một bộ công cụ phân tích phong phú (ví dụ: thực thi biểu tượng) đã có sẵn, do đó giảm công sức xây dựng lại từ đầu. Khi phân tích từng hàm assembly, chúng tôi đặt ngưỡng thời gian chờ là 10 phút. Chúng tôi chỉ gặp ít trường hợp vượt quá thời gian chờ (xem Phần 8.3).
* Quyết định ngưỡng a. Như đã thảo luận trong Phần 6, chúng tôi xác định a là một ngưỡng để điều chỉnh dự đoán của DNN. Trong phần này, chúng tôi mô tả một thủ tục thực nghiệm để xác định a. Nhìn chung, việc đặt a lớn cho thấy sự quá tin cậy vào kết quả phân tích của BinUSE (tạo ra nhiều FN trong dự đoán cuối cùng đã được điều chỉnh). Một a nhỏ, tuy nhiên, có thể bỏ qua cơ hội để ghép các hàm assembly và gây ra nhiều FPs hơn. Do đó, chúng tôi xác định a thực nghiệm để tối đa hóa tỷ lệ dương thật giữa tất cả các dự đoán tích cực được thực hiện bởi BinUSE.

A graph with a line

Description automatically generated

* Để làm điều này, chúng tôi tạo ra hai tập mã nhị phân bằng cách biên dịch coreutils bằng gcc với tối ưu hóa -O0 và -O3. Cho mỗi chương trình coreutils, giả sử trung bình có n hàm và toàn bộ bộ kiểm tra coreutils có m chương trình, chúng tôi sử dụng BinUSE để so sánh chéo tổng cộng n \* m cặp hàm, theo cài đặt chuẩn giới thiệu trong Hình 1. Ví dụ, hàm get\_global\_opt trong ls biên dịch bằng gcc -O0 sẽ được ghép với hàm tương tự trong ls biên dịch bằng gcc -O3. Với a đã quyết định, chúng tôi giữ các dự đoán tích cực được thực hiện bởi BinUSE có điểm tin cậy s ≥ a. Sau đó, chúng tôi tính toán độ chính xác jTPj/jTj, trong đó jTj đại diện cho số lượng trường hợp tích cực tổng cộng và jTPj đại diện cho số lượng dương thật trong kết quả ghép của BinUSE. Chúng tôi lặp lại giá trị a mỗi bước 0,02 từ 0,01 đến 1,0 và vẽ Hình 9. Theo Hình 9, a = 0,41 làm cho độ chính xác cao nhất.
* Tính chất tổng quát của a. Bộ kiểm tra mà chúng tôi sử dụng, coreutils, chứa hơn 100 chương trình có tính năng đa dạng, được sử dụng phổ biến trên hệ thống Linux. Quan trọng hơn, trong khi a = 0,41 được quyết định trên các trường hợp kiểm tra coreutils được biên dịch bằng gcc với các cài đặt -O0 và -O3, cấu hình này đã được chứng minh là hiệu quả đối với các cài đặt khác nhau khó khăn khác được giới thiệu bởi các trình biên dịch, tối ưu hóa, kiến trúc và làm rối khác nhau. Bảng 4 đánh giá cách thay đổi của a tác động đến việc cải thiện dưới các cài đặt so sánh khác nhau. Có thể thấy rằng a = 0,41 liên tục cho thấy kết quả khích lệ trong tất cả các cài đặt biên dịch. Đánh giá này minh họa tính chất tổng quát của a được lựa chọn trong hầu hết các tình huống sử dụng. Tuy nhiên, chúng tôi thừa nhận rằng tập dữ liệu phần mềm có sự chuyển đổi miền không nhỏ khiến việc lựa chọn tham số đặc biệt này có thể không tổng quát được trên các tập dữ liệu khác nhau. Người dùng có thể thực hiện cùng quy trình để quyết định sự lựa chọn tối ưu của họ với các tập dữ liệu khác. Trên thực tế, khi đánh giá các trường hợp kiểm tra binutils, chúng tôi phát hiện rằng a tối ưu là 0,68, chứ không phải 0,41, mặc dù cấu hình sau cũng đạt độ chính xác gần như như nhau.

A table with numbers and symbols

Description automatically generated

8. ĐÁNH GIÁ

8.1 Thiết lập đánh giá

* Tập dữ liệu và cài đặt biên dịch. Chúng tôi đánh giá BinUSE bằng việc sử dụng tập dữ liệu Linux coreutils (phiên bản 8.28). Tập dữ liệu coreutils chứa 106 chương trình. Chúng tôi biên dịch các chương trình với bảy cài đặt khác nhau (xem Bảng 5). Chúng tôi sử dụng gcc 7.5.0 và clang 4.0.1 để biên dịch các chương trình. Chúng tôi biên dịch các chương trình với không tối ưu hóa (-O0) và tối ưu hóa cao nhất (-O3). Để hỗ trợ so sánh chéo kiến trúc, chúng tôi biên dịch mã nhị phân trên ba kiến trúc khác nhau, 32-bit x86, 64-bit x86 và ARM. Chúng tôi báo cáo rằng mỗi chương trình coreutils được biên dịch với tùy chọn -O3 trung bình có 103,7 hàm. Nói cách khác, cho một cặp mã nhị phân coreutils, BinUSE cần so sánh chéo tổng cộng 103,7 \* 103,7 hàm assembly. Hơn nữa, chúng tôi đo hiệu năng của BinUSE bằng cách sử dụng tập dữ liệu Linux binutils (phiên bản 2.36) trong Phần 8.7. Tập dữ liệu binutils chứa 112 chương trình. Mỗi mã nhị phân binutils trung bình có 1.765,0 hàm. Do đó, cho một cặp mã nhị phân binutils, BinUSE cần thực hiện một số lượng so sánh chéo lớn hơn nhiều; xem Phần 8.7 để biết chi tiết về tối ưu hóa của chúng tôi để giảm số lượng so sánh. Chúng tôi sử dụng cùng bảy cài đặt để biên dịch các chương trình binutils trên kiến trúc x86 32-bit, x86 64-bit và ARM. Chúng tôi cũng đánh giá một ứng dụng thứ cấp phổ biến, tìm kiếm các hàm assembly có lỗ hổng. Chúng tôi sử dụng tập dữ liệu CVE được phát hành bởi [26], cũng được sử dụng bởi asm2vec. Các hàm assembly này đến từ phần mềm phức tạp trong thế giới thực, bao gồm OpenSSL, Wireshark, ffmpeg và ntpd; xem chi tiết ở Phần 8.6.

A table with numbers and symbols

Description automatically generated

* Đối với tất cả các chương trình kiểm tra, chúng tôi tạo mã nhị phân làm mờ với Obfuscator-LLVM [52], bao gồm ba kế hoạch làm mờ. Thay thế lệnh (-sub) thực hiện một số quy tắc ánh xạ để chuyển đổi một số lệnh thành lệnh có cùng ý nghĩa về ngữ nghĩa. Luồng điều khiển vô nghĩa (-bcf) chèn các tiên đề mờ vào các khối mã được chọn ngẫu nhiên. Nó thay đổi cấu trúc luồng điều khiển bằng cách thêm các nút và cạnh bổ sung [39]. Cấu trúc luồng điều khiển phẳng (-fla) thay đổi CFG của một hàm thành một cấu trúc "phẳng". Luồng thực thi ban đầu được bảo tồn bằng một câu lệnh chuyển đổi C được tạo ra có chọn lọc [39].
* Nhìn chung, các cài đặt biên dịch khác nhau được báo cáo trong Bảng 5 dẫn đến tổng cộng 12 cài đặt so sánh chéo (xem Bảng 7). Chúng tôi báo cáo các thống kê về độ phức tạp của các hàm assembly được đánh giá trong Bảng 6. Các kết quả được tính bằng cách lấy trung bình của tất cả các hàm assembly trong coreutils, binutils và cơ sở dữ liệu chứa phần mềm có lỗ hổng thế giới thực dưới tất cả các cài đặt biên dịch, tối ưu hóa và làm mờ. Đặc biệt, đối với mỗi hàm assembly, chúng tôi đo lường số lượng đường đi được bao phủ, tất cả các lệnh được phân tích bởi BinUSE, cũng như số lần gọi đến điểm ngoài gặp phải. Có thể thấy rằng các chương trình trong binutils và phần mềm có lỗ hổng thế giới thực thể hiện tính phức tạp tương đương, trong khi các chương trình coreutils có ít đường đi và ngắn hơn. Chúng tôi cũng báo cáo số đo hướng lướt [53], một số liệu thường được sử dụng để đánh giá độ phức tạp của một CFG: số hướng lướt được định nghĩa là e + n + 2, trong đó e và n lần lượt là số cạnh và khối cơ bản trong CFG. Tóm lại, chúng tôi hiểu rằng tất cả các hàm assembly được đánh giá trong nghiên cứu này có tính phức tạp hợp lý; theo quan sát của chúng tôi, các chương trình assembly này thường chứa các vòng lặp lồng nhau và nhiều lời gọi hàm (được inlined bởi BinUSE trong quá trình phân tích), tạo ra các ràng buộc biểu tượng phức tạp. Tuy nhiên, BinUSE có thể nhanh chóng hoàn thành phân tích mỗi hàm assembly trong vài phút; xem Phần 8.3.
* Các mô hình DNN. Chúng tôi sử dụng BinUSE để cải thiện bốn công cụ tìm kiếm hàm mã nhị phân dựa trên DNN tiên tiến: BinaryAI [15], asm2vec [14], PalmTree [18] và ncc [13].
* BinaryAI, được công bố tại hội nghị AAAI'20, thực hiện nhúng hàm assembly bằng cách tính toán nhúng khối cơ bản với BERT [54] và sau đó thực hiện nhúng đồ thị với mạng lưới nơ-ron đồ thị có cổng (GGNN) [55]. BinaryAI cung cấp các API để truy cập vào mô hình đã được huấn luyện và thực hiện so sánh các hàm mã nhị phân (xem https://github.com/binaryai). Chúng tôi làm rõ rằng dữ liệu huấn luyện của BinaryAI không được tiết lộ. Từ bài báo của nó, được nêu rõ rằng BinaryAI đã được huấn luyện trước trên hàng triệu mẫu mã nhị phân [15] được biên dịch bằng các trình biên dịch, tối ưu hóa và trên các kiến trúc khác nhau. Chúng tôi thấy rằng BinaryAI thể hiện độ chính xác đủ cao, ngang bằng với bài báo của nó [15]. Theo hiểu biết của chúng tôi, BinaryAI đại diện cho công cụ nhúng mã nhị phân tiên tiến nhất, đã được đánh giá [15] vượt trội hơn so với các mô hình nhúng phổ biến khác có thể được mở rộng mượt mà cho mã nhị phân, bao gồm Structure2vec [12], Word2vec [56], BERT [54], MPNN [57] và các mô hình CNN.

A table with numbers and a few words

Description automatically generated

A table with numbers and letters

Description automatically generated

* asm2vec, công bố tại IEEE S&P'20, tạo ra nhúng hàm assembly bằng cách chủ yếu sử dụng mô hình nhúng ngôn ngữ PV-DM mở rộng [58] và mạng neural đồ thị. Chúng tôi cài đặt phiên bản khách chính thức của nó yêu cầu IDA-Pro. Rất tiếc, asm2vec không cung cấp mô hình đã được huấn luyện trước để tái tạo kết quả được báo cáo của nó. Chúng tôi vì vậy làm theo bài báo của nó bằng cách tải lên các tập tin thực thi trong tập dữ liệu của chúng tôi được biên dịch trên kiến trúc x86 32-bit, 64-bit và ARM với không tối ưu hóa lên máy chủ của nó để tạo thành bộ lưu trữ hàm và huấn luyện mô hình. Trong quá trình thử nghiệm, chúng tôi lặp đi lặp lại việc tải lên tệp thực thi và cho mỗi hàm bên trong tệp thực thi tải lên, asm2vec trả về 15 hàm tương tự về mặt ngữ nghĩa nhất trong bộ lưu trữ hàm.
* ncc, công bố tại NeurIPS'18, tạo ra nhúng mã mã nguồn từ mã LLVM IR bằng cách xây dựng một đồ thị dòng ngữ cảnh, bao gồm các đặc trưng luồng dữ liệu và luồng điều khiển. Sau đó, nó sử dụng các mô hình nhúng dựa trên GNN để trích xuất biểu diễn số học. Chúng tôi sử dụng một bộ biên dịch nhị phân tĩnh phổ biến, RetDec [59], để chuyển đổi mã nhị phân thành mã LLVM IR. Để chuẩn bị đầu vào cho ncc, chúng tôi đã thử nghiệm các bộ biên dịch nhị phân khác, bao gồm mcsema [60] và mctoll [61]. Chúng thể hiện kết quả chuyển đổi LLVM IR tồi tệ hơn so với RetDec. ncc được huấn luyện bằng coreutils.
* PalmTree [18], công bố tại CCS'21, cung cấp mô hình ngôn ngữ mới cho việc nhúng các hướng dẫn máy x86. PalmTree có quy trình huấn luyện tự cấu trúc linh hoạt trên mã nhị phân không nhãn, trong đó biểu diễn được tạo ra được cho là hiệu quả trong các nhiệm vụ phụ thuộc hàng dưới phổ biến như phân tích độ tương đồng mã, suy luận nguyên mẫu hàm và phân tích tĩnh. Chúng tôi làm rõ rằng PalmTree tập trung vào tính toán một biểu diễn mới chỉ của các hướng dẫn máy. Theo báo cáo của nó, nó sử dụng pooling trung bình cho việc nhúng cơ bản khối, và Gemini [12] cho việc nhúng đồ thị kiểm soát (hàm). Chúng tôi tuân theo bài báo của nó để trang bị PalmTree với pooling trung bình cho việc nhúng khối cơ bản. Đối với việc nhúng đồ thị, chúng tôi liên hệ tác giả của Gemini để lấy mô hình nhúng đồ thị và chi tiết thiết lập nhưng không nhận được phản hồi vào thời điểm viết báo cáo. Vì vậy, chúng tôi sử dụng mô hình nhúng đồ thị được sử dụng bởi BinaryAI, GGNN, cho việc nhúng đồ thị của PalmTree. Chúng tôi gọi cài đặt này là PalmTree (mean/GGNN) trong quá trình đánh giá. Hơn nữa, vì PalmTree thể hiện độ chính xác tương đối thấp (xem Bảng 8), chúng tôi cũng thay thế pooling trung bình bằng HBMP [62], mô hình mạng nơ-ron hồi quy phổ biến, cho việc nhúng khối cơ bản. Bảng 8 cho thấy rằng HBMP có thể cải thiện độ chính xác của PalmTree một cách hợp lý. Sự cải thiện này của PalmTree được gọi là PalmTree (HBMP/GGNN) trong quá trình đánh giá.
* Chúng tôi sử dụng mô hình nhúng hướng dẫn của PalmTree đã được huấn luyện trước và được phát hành bởi các tác giả [63]. Các tác giả tiết lộ rằng mô hình này được huấn luyện với các tệp nhị phân x64 từ coreutils và binutils. Mô hình đã được huấn luyện trước này có kích thước vectơ nhúng là 128. Do đó, chúng tôi đánh giá PalmTree trên các tệp thực thi x86 64-bit và 32-bit và bỏ qua việc phân tích các tệp thực thi trên ARM. Mô hình nhúng hàm cho PalmTree được huấn luyện với binutils, được biên dịch bởi gcc và clang, với các tùy chọn -O0 và -O3. Đối với PalmTree (mean/GGNN), chúng tôi cấu hình kích thước pooling trung bình và kích thước nhúng đồ thị là 128. Bước truyền tin nhắn, một siêu tham số quan trọng cho GGNN, là 5. Đối với PalmTree (HBMP/GGNN), chúng tôi đặt kích thước HBMP là 128; các cài đặt khác đều giống như PalmTree (mean/GGNN).
* Sự rõ ràng về việc lựa chọn tập dữ liệu huấn luyện. Chúng tôi sử dụng mã nhị phân bình thường để huấn luyện các mô hình DNN, trong khi các mô hình DNN đã được đánh giá về tính ổn định trong các tùy chọn biên dịch khác nhau, tối ưu hóa khác nhau, kiến trúc khác nhau và các tùy chọn che dấu khác nhau. Rõ ràng là các mô hình DNN không được tiếp xúc với các mẫu mã nhị phân phức tạp như vậy, tức là không có các mẫu mã này trong tập dữ liệu huấn luyện. Một mặt, chúng tôi xác nhận rằng thiết lập này là tiêu chuẩn và chia sẻ bởi hầu hết các nghiên cứu hiện tại trong lĩnh vực này [14], [15], [18], [29], [64]. Lý do là chúng tôi không giả định phương pháp che dấu được áp dụng, vì các phương pháp che dấu là đa dạng và chủ yếu không thể dự đoán trong các thiết lập thực tế [39]. Mặt khác, việc kiểm tra các mẫu mã nhị phân không có trong tập dữ liệu huấn luyện có thể làm nảy sinh mối quan ngại rằng tập dữ liệu huấn luyện và tập dữ liệu đánh giá không có cùng một "phân phối tương tự". Nghiên cứu gần đây đã chứng minh khả năng cao của việc sử dụng các mẫu mã nhị phân được tối ưu hóa đa dạng để bổ sung mô hình học sâu và cải thiện tính ổn định của biểu diễn nhúng học được [65]. Chúng tôi để lại việc nghiên cứu trong tương lai để khám phá việc bổ sung các mô hình dựa trên DNN với các mẫu mã nhị phân được biến đổi qua các kỹ thuật tối ưu hóa và che dấu tiêu chuẩn.

A table with numbers and letters

Description automatically generated

8.2 BinUSE Performance

* Bảng 7 báo cáo hiệu suất của BinUSE liên quan đến tổng cộng 12 thiết lập so sánh trên tập dữ liệu coreutils. Hầu hết các so sánh đòi hỏi các cài đặt biên dịch, tối ưu hóa và kiến trúc gắn kết khác nhau, tạo ra các tình huống thách thức. Ví dụ, so sánh cuối cùng trong Bảng 7 đề cập đến một cài đặt rất khó, bao gồm kiến trúc gắn kết (ARM so với x86 64-bit), trình biên dịch gắn kết (gcc so với clang), tối ưu hóa gắn kết (-O0 so với -O3) và cũng áp dụng kỹ thuật che dấu dòng điều khiển (-fla) với cấu trúc luồng điều khiển bị thay đổi rộng rãi.
* Tổng thể, BinUSE được thiết kế để cẩn thận. Ví dụ, chúng tôi cho phép so sánh từng cặp và hoán vị cho các tham số và ký hiệu hàm (Mục 5.3), mặc dù các hoán vị này tạo ra nhiều ràng buộc để giải quyết và tiềm ẩn rủi ro FN. Tỷ lệ FN tổng thể (4,2%) được hiểu là thực tế và hợp lý. Phân tích mã nhị phân thế giới thực tiếp bộc lộ nhiều vấn đề kỹ thuật và tình huống đặc biệt, trong đó một số khó khăn, hoặc không thể giải quyết mà không cần nỗ lực thủ công. Mục 9 cung cấp thảo luận chi tiết hơn. BinUSE có thể không thể phân tích một số hàm không có điểm gọi ngoại vi xác định. Cho mỗi chương trình coreutils, trung bình có khoảng 103,7 hàm, khoảng 18,7 hàm không có điểm gọi ngoại vi, đóng góp chính vào FP. Ngoài ra, để tăng tốc phân tích, BinUSE chỉ phân tích một đồ thị con của mỗi hàm, điều này có thể sai lầm xem các hàm khác nhau là tương đương và gây ra thêm FP.
* Ngoài ra, các công cụ đảo ngược kỹ thuật và thực thi biểu tượng có thể gặp phải các ngoại lệ và chấm dứt phân tích cho 13.8% các trường hợp thử nghiệm (xem cột cuối cùng của Bảng 7). Tổng thể, công cụ thực thi biểu tượng, angr, có thể gây ra lỗi khi suy luận đích đến chuyển tiếp điều khiển của con trỏ mã. Cụ thể, Bảng 7 cho thấy BinUSE thất bại nhiều hơn ở các hàm được biên dịch với kỹ thuật che phủ dòng điều khiển -fla. Như đã đề cập trước đó, kỹ thuật che phủ này chuyển đổi CFG thành một câu lệnh chuyển tiếp điều khiển C và nối các khối cơ bản với một nút điều phối. Con trỏ mã thường được sử dụng trong nút điều phối để hướng dẫn việc chuyển tiếp điều khiển, dẫn đến khả năng thất bại cao khi cụ thể hóa các con trỏ mã biểu tượng. Đọc giả quan tâm có thể tham khảo Phần 9.2 để biết thêm thông tin về các vấn đề liên quan đến công cụ chuỗi ngược này.
* Thảo luận về Kết quả và Sử dụng. Tổng thể, để áp dụng phương pháp USE kiểm tra tính đồng giống của các hàm lắp ráp, chúng tôi đã đối mặt với một số thách thức thiết kế và trường hợp góc. Để cân nhắc cả sự hiểu biết và tính khả thi, BinUSE được thiết kế để chỉ phân tích một phân đồ của toàn bộ hàm. Chúng tôi nhấn mạnh rằng BinUSE không được thiết kế để sử dụng độc lập, mà là để bù đắp cho công cụ tìm kiếm mã nguồn dựa trên DNN. Thực tế, chúng tôi báo cáo rằng trong số tất cả các hàm có thể thông qua kiểm tra đồng giống, 87.9% có điểm tự tin là 1.0. Với hầu hết các ứng cử viên thông qua với điểm tự tin 1.0, việc tính toán độ chính xác top-k là không thực tế. Mặt khác, chúng tôi báo cáo cách các công cụ tìm kiếm hàm nhị phân dựa trên DNN được cải thiện bởi BinUSE trong Phần 8.5, Phần 8.6 và Phần 8.7, tương ứng.
* Những đánh giá trước đó, cũng như các đánh giá tiếp theo, đều giả định rằng RP được tạo thành bằng mã nhị phân được tối ưu hoặc che phủ. Chúng tôi làm rõ rằng khi sử dụng cấu hình ngược (mã nhị phân được tối ưu cao được tìm kiếm trong RP được hình thành bởi mã nhị phân không tối ưu -O0), hiệu suất có thể khác nhau. Tuy nhiên, chúng tôi nhấn mạnh rằng việc sử dụng mã nhị phân được tối ưu và che phủ để tạo RP là một cấu hình phổ biến được chia sẻ bởi hầu hết các công trình trước đó và tất cả các công cụ được đánh giá trong nghiên cứu này. Lưu ý rằng đây là một cấu hình thực tế. Ví dụ, chúng tôi tạo RP bằng các mẫu nhị phân được thu thập tự do và có thể được tối ưu hoá và che phủ cao. Sau đó, chúng tôi truy vấn RP và tìm các lỗ hổng trong các mẫu nhị phân đó bằng cách sử dụng mẫu lỗ hổng đã biết được tạo ra dưới dạng biểu mẫu tiêu chuẩn và không tối ưu hóa (-O0).

8.3 Thời gian xử lý

* Các thí nghiệm của chúng tôi với BinUSE được thực hiện trên một máy Ubuntu 18.04 với CPU Intel Xeon E5-2678 và RAM 256GB. Trung bình, BinUSE mất 56.6 phút CPU để xử lý hai tập tin thực thi coreutils (trung bình 25.0 giây để kiểm tra hai hàm), bao gồm tất cả các nhiệm vụ thực thi biểu tượng và giải quyết ràng buộc. Hãy nhớ rằng chúng tôi đặt ngưỡng thời gian 10 phút làm giới hạn thời gian của BinUSE khi phân tích một hàm: chỉ có ba trường hợp thời gian chạy quá giới hạn trong tất cả các hàm đã phân tích. Chúng tôi liệt kê ba hàm đó ở [22].
* Điều này cho thấy sức mạnh của phương pháp USE và thiết kế USE thực tế của chúng tôi để kiểm tra tính đồng giống. Điều này cũng chỉ ra chất lượng kỹ thuật của angr. Chúng tôi báo cáo rằng khoảng 23.5% thời gian xử lý được dùng cho thực thi biểu tượng. Giải quyết ràng buộc mất ba lần thời gian thực thi (76.5%). BinUSE tiêu tốn mọi đường dẫn bắt đầu từ điểm vào của hàm, trình ký thời mọi lời gọi hàm người dùng gặp phải cho đến khi đạt đến điểm gọi hàm bên ngoài đầu tiên. Điều này đòi hỏi tiêu thụ tài nguyên CPU chính.
* Để so sánh với BinUSE, chúng tôi đã thử nghiệm KLEE [33] (phiên bản 2.1) và phiên bản tối ưu hóa có tên MOKLEE [34] với 10 tập tin thực thi coreutils và một ngưỡng thời gian chạy là 10 giờ. KLEE mất vài giờ để xử lý một tập tin thực thi (3 trường hợp quá thời gian) trong khi MOKLEE có 8 trường hợp quá thời gian và 2 ngoại lệ. Xem kết quả của chúng tôi tại [66], [67].
* BinUSE cũng có lượng sử dụng bộ nhớ hợp lý: các thí nghiệm BinUSE của chúng tôi đã được thực hiện bằng cách mở 30 quy trình angr cùng một lúc trên máy chủ. Tổng lượng sử dụng bộ nhớ cao điểm của 30 quy trình đã thấp hơn 40GB. Tổng thể, BinUSE phân tích một phân đồ của từng hàm riêng lẻ và tất cả các hàm được gọi dưới dạng thể trên phân đồ đó. Do đó, trong sự so sánh với SE tiêu chuẩn (ví dụ: [34]), việc sử dụng bộ nhớ không phải là mối quan tâm hàng đầu cho BinUSE.
* Hình 10 trình bày chi tiết thời gian xử lý cho so sánh gcc -O0 so với gcc -O3. Chúng tôi báo cáo thời gian xử lý của thực thi biểu tượng và giải quyết ràng buộc trong Hình 10a và 10b tương ứng. Tổng thể, chúng tôi nhận thấy thời gian xử lý tăng tỷ lệ gần đúng tuyến tính với kích thước của các tệp thực thi. Điều này dễ hiểu: các tệp thực thi lớn có nhiều hàm hơn, do đó kéo dài thời gian thực hiện biểu tượng của BinUSE. Tương tự, các tệp thực thi lớn có thể chứa nhiều ràng buộc biểu mẫu phức tạp hơn, do đó kéo dài thời gian giải quyết các ràng buộc biểu mẫu. Tuy nhiên, có thể thấy rằng hầu hết các mẫu mã nhị phân có thể được phân tích trong 2.000 giây CPU cho thực thi biểu tượng và 4000 giây CPU cho giải quyết ràng buộc. Chúng tôi do đó đánh giá chi phí này là hợp lý.

A graph of a graph of a graph

Description automatically generated

8.4 So sánh mô hình DNN

* Chúng tôi trước tiên chạy bốn công cụ tìm kiếm hàm nhị phân dựa trên mô hình DNN là BinaryAI, asm2vec, ncc và PalmTree trên 12 cài đặt so sánh (xem cột đầu tiên của Bảng 8). PalmTree không thể xử lý tập tin thực thi trên nền tảng ARM; do đó, chúng tôi bỏ qua phần đánh giá tương ứng. Bảng 8 tóm tắt kết quả hiệu suất. BinaryAI được coi là vượt trội hơn tất cả các mô hình trong tất cả các cài đặt khác nhau. Trong khi các cài đặt giao kiến đòi hỏi các thách thức chính, BinaryAI dường như mạnh mẽ hơn đối với những thay đổi giao kiến, vì nó học từ mã nhúng giao diện nền tảng bởi IDA-Pro [15]. Việc làm rối, đặc biệt là làm rối luồng điều khiển (-fla), chủ yếu và liên tục làm giảm độ chính xác hàng đầu. Khi chuyển mã nhị phân thành mã LLVM IR để làm đầu vào cho ncc, chúng tôi đã gặp nhiều lỗi kỹ thuật đảo mã hóa. Công cụ đảo mã nhị phân, RetDec, gây ra ngoại lệ khi xử lý một số mã nhị phân. Đối với các trường hợp như vậy, chúng tôi chỉ đo lường độ chính xác hàng đầu đối với mã nhị phân được xử lý thành công (khoảng 40% trường hợp còn lại cho mã nhị phân được biên dịch bởi clang -O3). Các hàm còn lại đối đối đối khá đơn giản, điều này giải thích sự chính xác đáng kinh ngạc cao cho một số cài đặt so sánh của ncc.
* BinaryAI được duy trì bởi một ông lớn trong công nghiệp (Tencent), cho thấy có nhiều nguồn lực được dành cho việc đào tạo mô hình và chất lượng kỹ thuật tốt hơn. Chúng tôi không thể khôi phục lại độ chính xác cao được báo cáo trong bài báo asm2vec: chúng tôi nhấn mạnh rằng cả cộng đồng kỹ thuật phần mềm và an ninh đã chỉ ra các vấn đề tương tự [16], [68], [69]. Đánh giá của chúng tôi cho thấy asm2vec có độ chính xác hàng đầu 38,3%, mặc dù thấp hơn so với độ chính xác được báo cáo trong bài báo của nó, nhưng rất nhất quán với kết quả của nghiên cứu gần đây [16], [68], [69]. Tuy nhiên, asm2vec vẫn là một trong những khung nâng cao hàng đầu trong lĩnh vực này (vượt qua việc so khớp dựa trên CFG truyền thống [14]). PalmTree (mean/GGNN) cho thấy độ chính xác hợp lý. Nó đáng chú ý hơn asm2vec đối với cài đặt làm rối -sub trên mã nhị phân 64-bit x86. Tuy nhiên, PalmTree (mean/GGNN) trở nên ít chính xác đối với tập tin thực thi 32-bit x86. Nguyên nhân có thể là mô hình đã được huấn luyện sẵn cung cấp bởi PalmTree chủ yếu sử dụng các tệp thực thi 64-bit x86 làm dữ liệu huấn luyện. Chúng tôi làm rõ rằng PalmTree không cung cấp các chi tiết đầy đủ về việc huấn luyện lại mô hình, do đó, chúng tôi không thể rõ ràng cách nào để huấn luyện lại mô hình được phát hành của nó bằng cách sử dụng các tệp thực thi 32-bit x86. PalmTree (HBMP/GGNN) cho thấy sự cải thiện hứa hẹn trên các tệp thực thi 64-bit; độ chính xác của nó gần như bằng với BinaryAI. Tuy nhiên, BinUSE vẫn cung cấp một sự cải thiện cao cho PalmTree (mean/GGNN), như sẽ được báo cáo trong Phần 8.5.

8.5 Cải tiến mô hình DNN

* Để đánh giá sự cải tiến của các phương pháp tìm kiếm hàm mã nhị phân dựa trên DNN bằng cách sử dụng BinUSE, chúng tôi cố gắng trả lời hai câu hỏi: 1) RQ1: liệu BinUSE có thể cải tiến các công cụ tìm kiếm hàm mã nhị phân dựa trên DNN khác nhau? và 2) RQ2: liệu BinUSE có thể cải tiến BinaryAI với các cài đặt khác nhau? Đối với RQ2, chúng tôi lấy BinaryAI làm mục tiêu vì nó đáng chú ý vượt trội hơn ba mô hình còn lại. Ngoài việc cải tiến các phương pháp dựa trên DNN, chúng tôi cũng khám phá RQ3: liệu BinUSE có đủ tổng quát để cải tiến các công cụ diffing nhị phân thông thường dựa trên thông tin cấu trúc chương trình? Đối với RQ3, chúng tôi đánh giá một công cụ diffing nhị phân phổ biến, FuncSimSearch [70], được phát triển và duy trì bởi Google Project Zero.
* RQ1. Bảng 9 trình bày kết quả đánh giá theo các cài đặt khác nhau. Một cách nhất quán, chúng tôi đo lường độ cải tiến top-1, top-3 và top-5. Bảng 9 cho thấy rằng tất cả các phương pháp dựa trên DNN đều có thể được cải thiện đáng kể bằng cách sử dụng BinUSE. Điều này là dễ hiểu và nhất quán với động lực của chúng tôi: các mô hình DNN thường học từ các đặc trưng mã hóa thô, không đảm bảo độ bền đối với các cài đặt khác nhau, do đó gây ra rất nhiều báo động giả. BinUSE được thiết kế để giải quyết hạn chế chính của chúng một cách nhất quán. Chúng tôi nhận thấy rằng BinUSE tìm ra nhiều cơ hội để cải tiến các công cụ dựa trên DNN khác hơn là BinaryAI. Như đã đề cập trước đó, BinaryAI có độ chính xác cao hơn so với các công cụ khác: BinaryAI nên có ít dự đoán sai hơn để được điều chỉnh. Cũng thú vị để so sánh BinaryAI và PalmTree (HBMP/GGNN): trong khi hai mô hình này cho thấy độ chính xác gần như nhau (mô hình sau ít tốt hơn hơn một chút) trong Bảng 8, BinUSE mang lại cải tiến cao hơn đáng kể đối với PalmTree (HBMP/GGNN). Lý do chính là PalmTree (HBMP/GGNN) được nhận thấy hoạt động kỳ lạ trên các hàm mã nhị phân nhỏ. Do đó, khi đánh giá PalmTree (HBMP/GGNN), chúng tôi chỉ sử dụng các hàm có hơn ba khối cơ bản, dẫn đến fởr tỉ lệ cải tiến cao hơn, vì mẫu số (tức là tổng số hàm được phân tích; "N" trong Công thức 1) của tỉ lệ này là nhỏ.

A table with numbers and text

Description automatically generated

* Đánh giá này bao gồm bốn công cụ tìm kiếm mã nhị phân dựa trên DNN gần đây đã được chỉ ra gặp phải các vấn đề tương tự và có thể được cải tiến một cách nhất quán. Chúng tôi nhận thấy có cơ hội chủ yếu để cải thiện hiệu suất của các nghiên cứu khác chia sẻ phương pháp tương tự và giới hạn tiềm năng. Chúng tôi để lại điều này là công việc tương lai để cải thiện các mô hình DNN khác bằng BinUSE.
* Nghiên cứu trường hợp. Hình 11 trình bày một nghiên cứu trường hợp bằng cách so sánh chương trình coreutils shuf được biên dịch với gcc -O0 so với biên dịch với clang -O3. Nhớ lại rằng ncc, như đã báo cáo ở phần 3, tạo ra các lỗi tương tự. Chúng tôi thường chọn trường hợp này vì nó nhỏ hơn để dễ dàng trình bày. Hình 11a trình bày CFG của hàm rpl\_fflush khi biên dịch với gcc -O0. Hình 11b trình bày CFG của rpl\_fflush khi sử dụng clang -O3.
* Khi bật tối ưu hóa đầy đủ, CFG tương đối "bẹp phẳng" của rpl\_fflush trong Hình 11a được chuyển đổi thành biểu diễn tuyến tính trực quan. Thay vào đó, BinaryAI xem xét hàm khác, xrealloc (Hình 11c), làm phù hợp hàng đầu với rpl\_fflush trong Hình 11a. Chúng tôi hiểu kết quả này là hợp lý; xrealloc được coi là chia sẻ nhiều đặc trưng cấu trúc với rpl\_fflush, và do đó làm cho BinaryAI bị nhầm lẫn. Trái lại, BinUSE xây dựng các công thức biểu diễn đầu vào của hai điểm gọi hàm ngoài trong rpl\_fflush, và sử dụng các công thức này để xác định tính tương tự về mặt ngữ nghĩa giữa rpl\_fflush trong Hình 11a và rpl\_fflush trong Hình 11b.
* RQ2. Để trả lời RQ2, chúng tôi đã nghiên cứu ba tham số siêu của học biểu diễn liên quan đến kích thước vector nhúng. Nói chung, các kích thước khác nhau chủ yếu ảnh hưởng đến độ chính xác của mô hình: vector nhúng dài có thể chứa thông tin tinh vi về dữ liệu đầu vào, trong khi kích thước nhỏ hơn có thể không biểu thị ngữ nghĩa đủ tốt. Tuy nhiên, vector dài đòi hỏi nhiều thách thức cho việc huấn luyện mô hình và có thể làm suy yếu tính ổn định của mô hình. BinaryAI bao gồm ba tham số liên quan đến kích thước vector nhúng, bao gồm kích thước nhúng mã thông báo, kích thước nhúng hằng số và kích thước nhúng đồ thị. Nhớ lại BinaryAI trước tiên thực hiện nhúng cấp đoạn cơ bản và sau đó nhúng cấp đồ thị: kích thước nhúng mã thông báo và kích thước nhúng hằng số là các tham số siêu được sử dụng trong bước đầu tiên, trong khi kích thước nhúng đồ thị chủ yếu điều chỉnh chất lượng nhúng đồ thị. Bảng 10 báo cáo rằng đối với tất cả các tham số siêu, mặc dù kích thước nhúng khác nhau, BinUSE vẫn liên tục cải thiện độ chính xác. Tổng cộng, đánh giá này tiết lộ một quan sát dễ hiểu: kiểm tra tính tương đương liên tục giải quyết các báo động giả cao dù có thay đổi trong cài đặt mô hình, cho thấy tính tổng quát của BinUSE từ một khía cạnh quan trọng khác.
* RQ3. Nghiên cứu này chủ yếu tập trung vào tìm kiếm mã nhị phân dựa trên DNN, vì các phương pháp dựa trên DNN thể hiện độ chính xác hứa hẹn cao và đã vượt trội so với các thuật toán dựa trên cấu trúc chương trình truyền thống như đồ thị đồng cấu [71]. Tuy nhiên, rõ ràng thấy rằng BinUSE không bị giới hạn chỉ làm tăng cường cho các phương pháp dựa trên DNN. Về nguyên tắc, chúng tôi đề xuất rằng so sánh nhị phân dựa trên cấu trúc chương trình thông thường đối mặt với việc đưa ra các dự đoán không phân biệt và không đáng tin cậy. RQ3 chứng minh thực nghiệm các luận điểm của chúng tôi, bằng cách sử dụng BinUSE để cải thiện một công cụ so sánh nhị phân tiên tiến, FuncSimSearch, mà tính toán điểm Simhash trên đồ thị luồng điều khiển để quyết định hiệu suất của các hàm mã.
* Bảng 11 báo cáo kết quả đánh giá. Chúng tôi thấy rằng FuncSimSearch cho kết quả tệ hơn đáng kể so với các phương pháp DNN hiện đại. Do đó, BinUSE có thể cải thiện đáng kể độ chính xác của nó cho tất cả các cài đặt được đánh giá. Độ chính xác tương đối thấp của FuncSimSearch cũng được nhấn mạnh trong bài báo asm2vec. Điều này là hợp lý, vì các mô hình DNN hiện đại đã thể hiện khả năng đáng kể trong việc hiểu các cấu trúc phức tạp (mờ). Hơn nữa, các mô hình DNN có thể học từ các tập dữ liệu lịch sử toàn diện.
* Trên mặt khác, BinUSE thể hiện kết quả đáng khích lệ trong việc nâng cao FuncSimSearch. Tóm lại, khám phá của chúng tôi đối với RQ3 cho thấy các công cụ so sánh nhị phân dựa trên cấu trúc chia sẻ giới hạn chung về việc đưa ra các dự đoán không phân biệt và không đáng tin cậy, đặc biệt là đối với những thay đổi được giới thiệu bởi tối ưu hóa, đánh lạc hướng và nền tảng. Chúng tôi cho thấy kiểm tra tính tương đương được kích hoạt bởi BinUSE có thể giảm các cảnh báo giả cao của các phương pháp dựa trên cấu trúc một cách tổng quát.
* Tổng thể, đánh giá trong phần này liên tục chứng minh rằng các cảnh báo giả cao được tạo ra bởi các công cụ dựa trên cấu trúc (DNN) phổ biến có thể được cải thiện bởi BinUSE một cách tổng quát, hiệu quả và độc đáo. Do đó, chúng tôi khuyến nghị kết hợp tìm kiếm mã nhị phân với BinUSE và đạt được hiệu quả tương tác trong việc sử dụng sản xuất.

A diagram of a computer program

Description automatically generated

A table with numbers and symbols

Description automatically generated

8.6 Tìm kiếm hàm có lỗi bảo mật

* Chúng tôi tiến hành một nghiên cứu trường hợp bằng cách áp dụng BinUSE để bổ sung một nhiệm vụ tìm kiếm lỗ hổng đối với một tập dữ liệu lỗ hổng công khai. Ứng dụng này mô phỏng một kịch bản sử dụng bảo mật phổ biến: cho một hàm mã hóa f từ một tệp thực thi đáng ngờ, chúng tôi tìm kiếm trong cơ sở dữ liệu D các hàm đã biết có lỗ hổng và quyết định liệu f có khớp với bất kỳ hàm nào trong D hay không.
* Tương tự như asm2vec, chúng tôi sử dụng bộ dữ liệu D được công bố bởi [26]. Cơ sở dữ liệu này chứa các mẫu chương trình của tám lỗ hổng CVE. Chúng tôi đánh giá bảy lỗ hổng CVE, vì lỗ hổng CVE còn lại, venom, yêu cầu xây dựng lại qemu (phiên bản 2.4.0), không thể được xử lý bởi Obfuscator-LLVM. D bao gồm 12 hàm của bảy lỗ hổng CVE, bao gồm cả Heartbleed nổi tiếng sử dụng thư viện mã hóa OpenSSL và Shellshock cho phép kẻ tấn công từ xa thực thi các lệnh tùy ý trên máy nạn nhân. Để tăng độ khó, D cũng chứa 1.225 "mẫu tiêu cực", đại diện cho các hàm không có lỗ hổng. Một công cụ tìm kiếm lỗ hổng phải khớp các đầu vào có lỗ hổng với các mẫu lỗ hổng chính xác trong D ở vị trí hàng đầu, mà không bị ảnh hưởng từ các hàm còn lại (đáng tin cậy).
* Ở bước này, chúng tôi biên dịch các mẫu trong D thành cơ sở dữ liệu Dasm của các hàm mã hóa bằng bốn cài đặt là -sub, -bcf, -fla và -hybrid. Lưu ý rằng cài đặt hybrid, được gọi là -hybrid, kết hợp tất cả ba phương pháp lẩn trốn cùng nhau trong quá trình biên dịch. Chúng tôi cũng kích hoạt tối ưu hóa đầy đủ -O3 khi biên dịch mỗi chương trình mẫu và hàm mục tiêu. Tóm lại, cho một hàm mã hóa đã được tối ưu hóa mạnh (-O3) với các lỗ hổng đã biết, chúng tôi lấy các hàm khớp của nó từ Dasm và kiểm tra xem các hàm khớp chính xác, có cùng các lỗ hổng, có tồn tại trong các ứng cử viên được xếp hạng hàng đầu hay không.
* A table with numbers and symbols

  Description automatically generated
* Ở bước này, chúng tôi đánh giá asm2vec, BinaryAI và hai phiên bản của PalmTree. Chúng tôi bỏ qua việc đánh giá ncc vì chúng tôi phát hiện rằng công cụ binary lifter mà ncc sử dụng gặp quá nhiều lỗi khi xử lý các phần mềm phức tạp trong thế giới thực. Chúng tôi báo cáo kết quả đánh giá cho mỗi cài đặt trong các Bảng 12, 13, 14 và 15. asm2vec dường như gặp khó khăn với OpenSSL và Wireshark, vì cả hai chương trình này đều phức tạp về mặt kỹ thuật. Đối với ba phiên bản của OpenSSL và Wireshark, asm2vec xếp hạng khớp chính xác thấp hơn rất nhiều. Ví dụ, asm2vec xếp hạng khớp chính xác của lỗ hổng Heartbleed trong OpenSSL (phiên bản 1.0.1f) ở vị trí thứ 17, cho thấy người dùng có thể phải so sánh thủ công ít nhất 17 bản sao của chương trình trong Dasm để xác nhận rằng lỗ hổng Heartbleed tồn tại trong đầu vào đáng ngờ. Ngược lại, BinUSE có thể khớp đầu vào đáng ngờ với lỗ hổng Heartbleed trong Dasm ở vị trí hàng đầu. Khi phân tích một lỗ hổng CVE nổi tiếng khác, ws-snmp, asm2vec cũng đạt được độ chính xác thấp hơn rất nhiều. Chúng tôi thấy rằng lỗ hổng này chứa một CFG lớn, gây trở ngại cho việc tính toán graph-level embedding của asm2vec dựa trên random walk. Với sự hỗ trợ của BinUSE, asm2vec có thể đặt hàm có lỗ hổng chính xác vào vị trí hàng đầu.
* Các quan sát tương tự được thu được từ đánh giá BinaryAI và PalmTree. BinaryAI nói chung cho thấy độ chính xác hứa hẹn so với asm2vec. Cuộc điều tra thủ công của chúng tôi cho thấy BinaryAI trích xuất các chuỗi hằng biểu đạt từ những tệp thực thi này; phụ thuộc vào những chuỗi hằng này giúp khớp chính xác giữa các hàm có lỗ hổng. Trong khi BinaryAI vẫn có thể mắc một số lỗi, BinUSE có thể cải thiện đáng kể độ chính xác của nó bằng cách khớp chính xác tất cả các hàm có lỗ hổng ở vị trí hàng đầu. So với BinaryAI và asm2vec, PalmTree cho thấy độ chính xác kém hơn trong đánh giá này. Chúng tôi đánh giá hai cấu hình của PalmTree trong Bảng 14 và Bảng 15, tương ứng. Mặc dù việc cải thiện của chúng tôi, bằng cách thay thế mean pooling mặc định của nó bằng HBMP, tăng đáng kể độ chính xác trong nhiệm vụ này, PalmTree vẫn mắc một số lỗi khá lớn, đặc biệt là đối với mã nhị phân bị lẩn trốn. Ví dụ, khi bật điều khiển flow flattening, ký hiệu là -fla và -hybrid trong Bảng 14 và Bảng 15, việc khớp chính xác là thấp hơn top-1000. Người dùng khó có thể xác định được lỗ hổng CVE từ đầu vào đáng ngờ, vì khớp chính xác của họ trong Dasm được xếp hạng ở vị trí rất thấp. Chúng tôi cũng muốn làm rõ rằng có một số trường hợp trong Bảng 15 không nằm ở vị trí hàng đầu sau khi cải thiện của BinUSE (ví dụ, trường hợp wget dưới cài đặt biên dịch -hybrid). Chúng tôi làm rõ rằng tất cả các trường hợp này không nằm ở vị trí hàng đầu là do nhiều trường hợp, bao gồm cả trường hợp tích cực thực sự, đều cùng xếp hạng hàng đầu. Mặc dù điều này làm tăng độ khó cho người dùng xác nhận lỗ hổng, BinUSE vẫn giảm đáng kể công sức. Ví dụ, theo Bảng 15, trong khi người dùng có thể cần kiểm tra 944 trường hợp để xác nhận rằng đầu vào đáng ngờ thực sự chứa lỗ hổng CVE 2014-4877 (nếu có thể), người dùng chỉ cần kiểm tra 14 trường hợp sau khi sử dụng BinUSE.

A table with numbers and text

Description automatically generated

* Discussion về cải thiện của BinUSE: Mặc dù các công cụ hiện có dựa trên DNN thường mắc sai lầm trong nhiệm vụ này, BinUSE vẫn có thể xác định đúng các khớp chính xác nhất với điểm tự tin cao nhất. Ngoài việc nắm bắt chính xác các ràng buộc cấp độ ý nghĩa, cuộc điều tra thủ công của chúng tôi cho thấy BinUSE, bằng cách khớp tên của các callsite ngoài, thực sự tiết lộ một "phím tắt" để tìm kiếm các hàm có lỗ hổng. Cụ thể, chúng tôi phát hiện rằng trong những chương trình thực tế như OpenSSL, mỗi hàm assembly, bao gồm cả các hàm có lỗ hổng CVE, thường có một mẫu callsite riêng biệt. Nói cách khác, việc sử dụng tên của callsite bên ngoài đã giúp khớp một số lượng khá lớn các hàm có lỗ hổng thực tế một cách hiệu quả. Ngược lại, nhiều chương trình trong coreutils và binutils chứa các hàm có ý nghĩa gần như liên quan nhau. Ví dụ, các hàm quote\_mem, quote\_n\_mem, quoteargs\_n\_mem trong các chương trình coreutils tiêu biểu chia sẻ mẫu callsite gần như giống nhau. Điều này tạo ra một thách thức bổ sung cho việc khớp hàm của BinUSE. Trong khi đánh giá tìm kiếm hàm có lỗ hổng trong phần này đã tiết lộ rằng BinUSE có thể thể hiện kết quả tốt hơn khi xử lý phần mềm thực tế.

A table with numbers and text

Description automatically generated

* Tóm lại, đánh giá trong phần này đã cho thấy kết quả đáng khích lệ khi sử dụng BinUSE trong việc phân tích ứng dụng thực tế cho mục đích bảo mật. Chúng tôi cũng giải thích rằng đánh giá trong phần này chứng minh sự cần thiết của việc xem xét thông tin callsite chi tiết khi thực hiện khớp hàm.

A table with numbers and symbols

Description automatically generated

8.7: Mở rộng của BinUSE

* Như được định nghĩa trong Phần 2.1, phương pháp mà chúng tôi đề xuất so sánh một hàm mục tiêu ft với mọi hàm trong kho chứa RP của các hàm assembly. Mặc dù USE được tùy chỉnh mà BinUSE sử dụng được chứng minh là hiệu quả (Phần 8.3), nhưng nó vẫn đòi hỏi chi phí lớn nếu áp dụng tổng quát. Do đó, chúng tôi xem xét một mở rộng có thể của BinUSE; chúng tôi nhằm giảm chi phí bằng cách so sánh ft với top-k hàm RPk RP được trả về đầu tiên bởi các công cụ so sánh nhị phân DNN.
* Cài đặt: Tại bước này, chúng tôi đánh giá một tập dữ liệu quy mô lớn và đầy thử thách, Linux binutils. Mỗi chương trình trong binutils chứa khoảng 1.765.0 hàm. Điều đó có nghĩa là khi phân tích mỗi chương trình trong coreutils, RP có khoảng một trăm hàm, trong khi phân tích mỗi binutils đối diện với một RP có khoảng 15 lần nhiều hàm hơn. Bảng 16 báo cáo độ chính xác top-1 qua các trường hợp kiểm tra binutils bằng hai công cụ dựa trên DNN. Ở bước này, asm2vec không được bao gồm, vì client cục bộ của nó bị lỗi khi xử lý các trường hợp kiểm tra binutils [72]. Đối với ncc, trình nhấn của nó không hoạt động hoặc tạo mã LLVM IR bị lỗi trên các chương trình binutils; chúng tôi do đó bỏ qua việc đánh giá ncc.

A table with numbers and text

Description automatically generated

* In general, Bảng 16 cho thấy cả hai mô hình DNN, đặc biệt là PalmTree, có thể được cải thiện đối với các trường hợp kiểm tra binutils. Tuy nhiên, vì BinUSE mặc định cần so sánh hàm mục tiêu ft với mọi hàm trong RP, chúng tôi coi đó là cần thiết để tối ưu hóa việc sử dụng BinUSE bằng cách hạn chế phân tích của BinUSE đối với top-K hàm được xếp hạng bởi các mô hình DNN. Lưu ý rằng mở rộng này chỉ yêu cầu thay đổi một cách nhỏ thuật toán tái sắp xếp đã được định nghĩa trong Phần 6: trong các thí nghiệm hiện tại của chúng tôi, chúng tôi chọn K là 100. Khi mô hình DNN đã quyết định các hàm trùng khớp top 100 với hàm mục tiêu ft, BinUSE được sử dụng để so sánh các hàm được xếp hạng này với ft và điều chỉnh thứ hạng của chúng, sử dụng chiến lược tái sắp xếp đã được trình bày trong Phần 6. Như vậy, việc so sánh của BinUSE được giảm từ kích thước RP (jRPj) xuống còn 100, giảm chi phí khi phân tích các chương trình binutils. Tuy nhiên, vì BinUSE chỉ truy cập và tái sắp xếp 100 hàm assembly hàng đầu được xếp hạng bởi mô hình DNN, độ chính xác top-k đã được cải thiện (trong đó k 100) bị giới hạn bởi độ chính xác top-100 của mô hình DNN. Nói cách khác, nếu mô hình DNN mục tiêu có độ chính xác thấp ngay cả đối với top 100, khả năng cải thiện nó là không cao.

A table with numbers and letters

Description automatically generated

* Kết quả. Chúng tôi báo cáo kết quả cải thiện so với BinaryAI và PalmTree trong Bảng 17. Lưu ý rằng trong bảng này, chúng tôi đánh giá 12 cài đặt so sánh phù hợp với các cài đặt đánh giá trước đó của chúng tôi. Tuy nhiên, chúng tôi xác nhận rằng, theo thông tin được tiết lộ bởi các tác giả BinaryAI, các chương trình binutils được bao gồm trong tập dữ liệu huấn luyện của BinaryAI theo ba cài đặt so sánh: gcc -O0 so với gcc -O3, gcc -m32 -O0 so với gcc -O3, và arm -O0 so với gcc -O3. Điều này giải thích tương đối thấp cải thiện của BinUSE trong ba cài đặt này: ví dụ, đối với cài đặt so sánh gcc -m32 -O0 so với gcc -O3, BinUSE dẫn đến cảm giác cải thiện âm. Chúng tôi đã xem xét các trường hợp này một cách thủ công và xác nhận rằng chúng xuất phát từ các dự đoán sai của BinUSE. Tương tự, BinUSE mang lại cải thiện thấp đối với BinaryAI trong ba cài đặt so sánh sử dụng việc làm mờ -sub. Nghiên cứu thủ công của chúng tôi cho thấy làm mờ -sub chủ yếu không ảnh hưởng đến cấu trúc luồng điều khiển và tạo ra ít thách thức đối với BinaryAI. Kết quả là, ba cài đặt so sánh -sub này chủ yếu giống với các cài đặt so sánh chưa được làm mờ trong tập dữ liệu huấn luyện của BinaryAI, do đó khó có thể được cải thiện bởi BinUSE. Tuy nhiên, đối với hai cài đặt làm mờ khác (-bcf và -fla), BinUSE mang lại cải thiện tổng quát cao hơn, mặc dù hai phương pháp làm mờ này đã thay đổi đáng kể cấu trúc luồng điều khiển.
* BinUSE đạt được cải thiện cao hơn đối với PalmTree so với BinaryAI. Điều này chủ yếu do độ chính xác tương đối thấp của PalmTree trên các trường hợp kiểm tra binutils, để lại nhiều cơ hội cho cải thiện. Trong khi đó, so với các đánh giá trên tập dữ liệu coreutils, BinUSE đạt được cải thiện thấp hơn. Ngoài khó khăn chung trong việc phân tích các hàm binutils, chúng tôi xác nhận rằng trong đánh giá này, BinUSE chỉ phân tích 100 hàm hàng đầu được trả về bởi các công cụ dựa trên DNN. Theo quan sát của chúng tôi, một số kết hợp đúng không nằm trong 100 hàm hàng đầu. Để khám phá một mức độ cải thiện độ chính xác cao hơn, người dùng có thể xem xét sử dụng 150 hoặc 200 hàm hàng đầu được trả về bởi các công cụ dựa trên DNN.
* Các Phần Mở Rộng Khả Thi Khác. Bên cạnh phần mở rộng được đề xuất và đánh giá trong phần này, chúng tôi ưu tiên những cách tiếp cận khác để mở rộng việc sử dụng BinUSE. Tóm lại, những tiến bộ gần đây trong các kỹ thuật Trí tuệ Nhân tạo Giải thích (XAI) đã cho phép xác định những thành phần mã nguồn có tác động lớn nhất đối với quyết định của mô hình DNN. Do đó, chúng tôi dự kiến mở rộng BinUSE và tạo ra một quy trình "xác nhận sau cùng" trong đó chúng tôi sử dụng các kỹ thuật XAI để đánh dấu những đoạn mã nguồn ảnh hưởng, được gọi là c1 và c2, chịu trách nhiệm chính cho quyết định của mô hình DNN để ghép nối các hàm mã nguồn. Sau đó, chúng tôi có thể sử dụng BinUSE đặc biệt cho những đoạn mã c1 và c2 đã được đánh dấu này để kiểm tra sự tương đương về mặt ngữ nghĩa. Cần lưu ý rằng những đoạn mã quan trọng này, c1 và c2, thường nhỏ hơn đáng kể so với toàn bộ các hàm mã nguồn f1 và f2, giúp giảm đáng kể chi phí sử dụng BinUSE. Chúng tôi để công việc này cho khám phá trong tương lai; khó khăn chính sẽ là xác định chính xác ranh giới mã nguồn của c1 và c2, vì sự chệch lệch nhỏ trong các ràng buộc tượng trưng đã hình thành có thể làm thay đổi quyết định từ tương đối thỏa mãn thành không thỏa mãn, hoặc ngược lại.

9 THẢO LUẬN

* Độ chính xác và Đầy đủ. Điểm mạnh của BinUSE là khả năng thực hiện kiểm tra tương đương cấp độ hàm nhị phân thực tế với tỷ lệ báo động sai thấp và tốc độ cao. Tuy nhiên, trong quá trình đánh giá của chúng tôi, khi sử dụng BinUSE để phân tích các tập hợp biến đổi của mã nguồn nhị phân thế giới thực, có thể xảy ra tỷ lệ FP trung bình là 25,0% và tỷ lệ FN trung bình là 4,2% (xem Bảng 7).
* Ngoài việc xấp xỉ quá mức không gian đầu vào hợp lệ, sự triển khai USE của chúng tôi, BinUSE, không đảm bảo do một số thách thức kỹ thuật (ví dụ: so sánh giữa các kiến trúc). Ngoài ra, động cơ SE cơ bản, angr [51], có một mô hình bộ nhớ nhẹ nhàng nhưng không đảm bảo. Ngoài ra, một số hàm không chứa điểm gọi bên ngoài và do đó không thể phân tích được bởi BinUSE. Chúng cũng đóng góp vào các lỗi của BinUSE.

A table with numbers and symbols

Description automatically generated

* Tuy nhiên, khác với các kỹ thuật phát hiện bản sao nhị phân dựa trên SE trước đó mà phân tích các khối cơ bản hoặc dấu vết thực thi [4], [20], BinUSE có tính mở rộng và hiệu quả để phân tích toàn bộ chương trình coreutils. Để cung cấp một thiết kế USE thực tế cho việc kiểm tra tính tương đương của hàm, thiết kế của BinUSE đánh đổi độ đầy đủ để có tốc độ nhanh. Lưu ý rằng vấn đề tìm kiếm hàm là khó khăn vì tỷ lệ dương thực tế trong dân số tất cả các hàm khá thấp (khoảng 1/103.7 tức là khoảng 1,0% với mỗi chương trình coreutils trong tập dữ liệu của chúng tôi chứa khoảng 103.7 hàm). Bên cạnh FPs và FNs, Bảng 7 cũng báo cáo rằng việc phân tích 13,8% hàm coreutils bị chấm dứt do các ngoại lệ của động cơ thực thi biểu tượng hoặc nền tảng kỹ thuật đảo ngược.
* Trong phần còn lại của phần này, chúng tôi thảo luận về tất cả các vấn đề có thể gây ra báo động sai của BinUSE trong các phần 9.1 và 9.2. Phần 9.3 liệt kê các vấn đề có thể dẫn đến việc thất bại của BinUSE. Chúng tôi làm rõ rằng nhiều vấn đề cũng đã được chỉ ra bởi các công cụ SE trước đó [75]. BinUSE sẽ trở nên chính xác sau khi sửa các vấn đề này (cấp độ thực hiện), điều này rất thách thức và nằm ngoài sự cân nhắc của công việc này và nhiều "động cơ" SE khác (ví dụ: do mô hình bộ nhớ không đảm bảo; xem phần thảo luận 9.2).

9.1. Không nhất quán về các điểm gọi hàm bên ngoài

* Để so sánh một hàm mã hóa mục tiêu ft với một hàm khác fs, BinUSE được thiết kế để trích xuất các chữ ký ý nghĩa từ đầu vào của cuộc gọi hàm bên ngoài để kiểm tra tính tương đương. Nói cách khác, chúng tôi giả định rằng biên dịch mã nguồn của một hàm C không nên tạo ra hai hàm mã hóa không nhất quán với các điểm gọi hàm bên ngoài. Mặc dù những giả định này thường đúng ngay cả đối với các thiết lập khó khăn (ví dụ: đối kiến kiến trúc hoặc làm khó), chúng tôi liệt kê tất cả các trường hợp góc nhìn vi phạm giả định của chúng tôi trong phần này.
* Đổi tên điểm gọi hàm bên ngoài. Để xây dựng các ràng buộc và kiểm tra tính tương đương của hai điểm gọi hàm, trước tiên chúng tôi quyết định xem chúng có đề cập đến cùng một hàm bên ngoài hay không. Rõ ràng, một điểm gọi hàm bên ngoài của fopen không nên được khớp với một điểm gọi hàm fwrite.
* Tuy nhiên, tối ưu hóa trình biên dịch có thể đổi tên một số hàm thư viện C. Ví dụ, chúng tôi thấy rằng khi biên dịch các chương trình coreutils với tối ưu hóa được kích hoạt, các cuộc gọi hàm đến hàm thư viện C dcgettext có thể được tối ưu hóa thành gettext nhưng không làm thay đổi ý nghĩa. Với hiện thực hiện tại, chúng tôi thủ công ánh xạ một hàm thư viện thành các hàm có thể thay thế của nó do tối ưu hóa. Điều này giúp chúng tôi xem xét dcgettext, dgettext và gettext là các hàm thư viện tương đương. Tuy nhiên, chúng tôi thừa nhận rằng ánh xạ của chúng tôi không hoàn chỉnh, điều này có thể dẫn đến không phù hợp với một số điểm gọi tham chiếu đến các cuộc gọi thư viện C khác nhau. Chúng tôi đã thảo luận về vấn đề này và liệt kê tất cả các trường hợp đổi tên chúng tôi đã xây dựng trong phần 5.3. Theo kiến thức của chúng tôi, nó bao gồm tất cả các sự thay thế hàm thư viện có thể được tìm thấy trong các trường hợp thử nghiệm của chúng tôi.
* Loại bỏ điểm gọi hàm bên ngoài do tối ưu hóa. Tối ưu hóa trình biên dịch có thể thay thế các cuộc gọi thư viện C chuẩn bằng các cuộc gọi hàm xây dựng sẵn và tiếp tục lồng ghép hàm xây dựng sẵn. Ví dụ, chúng tôi thấy rằng gcc, với tối ưu hóa -O3 được kích hoạt, có thể lồng ghép một phiên bản xây dựng sẵn của các hàm thư viện C như memset và memcpy. Điều này giúp tiết kiệm chi phí thêm của cuộc gọi và trả về hàm. Trái ngược, clang dường như giữ lại những cuộc gọi thư viện này ngay cả khi bật tối ưu hóa đầy đủ.
* Điều này có thể gây ra không nhất quán và FNs khi so sánh các hàm mã hóa do gcc và clang biên dịch, vì các cuộc gọi thư viện nhất định (ví dụ: memset) trong các hàm mã hóa biên dịch bởi clang sẽ không bao giờ tìm thấy điểm gọi thích hợp của nó trong hàm mã hóa biên dịch bởi gcc.
* Không có điểm gọi hàm bên ngoài trong một hàm. Khi so sánh hai hàm ft và fs, có thể xảy ra trường hợp không một trong hai hàm đã mở rộng CFG chứa bất kỳ cuộc gọi hàm bên ngoài nào. Đối với các trường hợp như vậy, hiện tại, hiện thực của chúng tôi phải bỏ qua việc so sánh. Hai hàm có rất ít khả năng có chức năng giống nhau trong trường hợp một hàm đã mở rộng CFG chứa các cuộc gọi hàm bên ngoài và hàm còn lại không. Tuy nhiên, FNs vẫn có thể được giới thiệu, trong trường hợp tất cả các cuộc gọi hàm bên ngoài được loại bỏ bởi tối ưu hóa của trình biên dịch từ CFG mở rộng, mặc dù khả năng xảy ra là rất thấp. Hơn nữa, nếu không có điểm gọi hàm bên ngoài nào được tìm thấy trong cả hai hàm, chúng tôi dựa vào kết quả tương tự được sinh ra bởi các công cụ dựa trên DNN để so sánh.
* Không có tham số trong điểm gọi hàm bên ngoài. Một số cuộc gọi thư viện C không có tham số (ví dụ: time). Đối với các trường hợp như vậy, hiện tại, hiện thực của chúng tôi sẽ trích xuất các điều kiện đường dẫn của hàm này. Nói cách khác, giải pháp của chúng tôi cho các điểm gọi hàm bên ngoài như vậy tương ứng với việc tối ưu hóa được giới thiệu trong phần 5.5. Như đã đề cập trong phần 5, các ràng buộc đường dẫn được thu thập cùng với việc thực hiện ký hiệu và được sử dụng để hình thành ràng buộc ký hiệu. Tuy nhiên, nếu không có điều kiện đường dẫn nào có thể được xây dựng, chúng tôi bỏ qua việc so sánh điểm gọi này.

9.2 Những vấn đề khác

* Ngoài việc không nhất quán về các điểm gọi hàm bên ngoài, chúng tôi cũng liệt kê một số vấn đề khác có thể làm suy yếu việc phân tích của BinUSE. Chúng chủ yếu xuất phát từ mô hình bộ nhớ không đáng tin cậy của angr và một số trường hợp cụ thể được đưa ra bởi việc so sánh giữa các kiến trúc khác nhau.
* Mô hình bộ nhớ không đáng tin cậy của angr. angr kết hợp thực thi cụ thể với thực thi tượng trưng để giảm thiểu chi phí [76]. Trước khi giải tham chiếu con trỏ tượng trưng, angr sẽ thay thế địa chỉ bộ nhớ tượng trưng bằng địa chỉ cụ thể. Thiết kế này giúp giảm gánh nặng của phân tích điểm-tới phức tạp. Tuy nhiên, việc áp dụng mô hình bộ nhớ không đáng tin cậy làm cho việc thực hiện một phiên bản BinUSE đáng tin cậy trở nên không khả thi.
* Thay đổi khu vực bộ nhớ toàn cầu bằng angr. Chúng tôi cũng phát hiện rằng khi bắt đầu thực hiện BinUSE tại điểm nhập của một hàm, một số vị trí bộ nhớ có thể đã được khởi tạo với các giá trị cụ thể. Những giá trị cụ thể này, khi được sử dụng để tính toán các điều kiện nhánh, có thể buộc angr chỉ đi vào một con đường thay vì khám phá tất cả các con đường khả thi. Kết quả là, angr có thể bị cản trở khỏi tìm các con đường tới các điểm gọi hàm bên ngoài cụ thể. Bỏ qua các điểm gọi hàm bên ngoài có thể dẫn đến việc gặp phải FN.
* Lỗi của IDA-Pro. Để so sánh một cặp hàm viết bằng ngôn ngữ hợp ngữ f và f0, chúng tôi tiến hành quá trình USE và giải các ràng buộc, điều này thường có chi phí cao. Hiện thực của BinUSE thực sự áp dụng tiêu chí dừng sớm hợp lý: f và f0 sẽ không được so sánh một cách chặt chẽ, trong trường hợp chúng không chia sẻ ít nhất một điểm gọi hàm nào đó. Thay vào đó, thay vì sử dụng angr để thu thập các điểm gọi hàm bên ngoài trong quá trình USE, hiện thực hiện tại của chúng tôi sử dụng IDA-Pro để disassemble tĩnh mã máy, tái thiết lập CFG và thu thập đệ quy các điểm gọi hàm bên ngoài của f và f0. Tuy nhiên, chúng tôi báo cáo rằng IDA-Pro có thể đôi khi gặp lỗi phân tích ngược và dừng việc phân tích các hàm, đặc biệt là đối với các mẫu mã nhị phân đã tối ưu hóa và đã bị làm rối (gọi lại rằng chúng tôi biên dịch các bộ kiểm tra của chúng tôi với tối ưu hóa đầy đủ và các phương pháp làm rối khác nhau). IDA-Pro đã không thể phân tích khoảng 1,0% mẫu mã nhị phân trong tất cả các bộ kiểm tra của chúng tôi; đối với những trường hợp này, chúng tôi phải dựa vào dự đoán của các công cụ dựa trên DNN.
* Những thách thức của các kiến trúc khác nhau. Đánh giá của chúng tôi bao gồm một số cài đặt kiến trúc khác nhau, bằng cách so sánh các hàm được biên dịch trên một kiến trúc với các hàm được biên dịch trên các kiến trúc khác nhau. angr thực hiện thực thi tượng trưng bằng cách chuyển mã nhị phân thành VEX, một ngôn ngữ trung gian giống RISC. Mặc dù VEX được xem là "độc lập với nền tảng," chúng tôi vẫn gặp một số không nhất quán về kiến trúc gây trở ngại cho việc kiểm tra tương đương chắc chắn.
* Đăng ký 64-bit so với Đăng ký 32-bit. Chúng tôi tìm thấy rất nhiều trường hợp, trong đó một đăng ký 64-bit trên các kiến trúc 64-bit x86 được biểu diễn bằng hai đăng ký 32-bit trên các kiến trúc 32-bit x86. Khi hình thành ràng buộc cho việc kiểm tra tương đương, việc so khớp đăng ký 64-bit với hai đăng ký 32-bit tương ứng trở nên rất không rõ ràng, nếu có thể.
* Hằng 64-bit so với Hằng 32-bit. Chúng tôi cũng thấy rằng một số hằng số bị thay đổi trong mã máy 32-bit và 64-bit. Ví dụ, một hằng số 0x55555554 trong mã máy 32-bit x86 được mở rộng thành 0x5555555555555554 trong mã máy 64-bit. Mặc dù điều này không ảnh hưởng đến việc thực thi trên CPU, chúng tôi lưu ý rằng những không nhất quán này gây ra kết quả sai cho việc thực thi tượng trưng và kiểm tra tương đương.

9.3 Những lỗi của BinUSE:

* Phần này liệt kê các lỗi được gây ra bởi công cụ phân tích tượng trưng cũng như các công cụ đảo ngược mã nguồn. Như đã đề cập ở Phần 9.2, những vấn đề này đóng góp vào việc gây ra các lỗi khi phân tích mã nguồn nhị phân của coreutils (trung bình 13.8% trường hợp thất bại; như đã báo cáo trong Bảng 7).
* Con trỏ mã tượng trưng. Như đã đề cập ở phần 9.2, angr thực hiện mô hình bộ nhớ không chính xác. Khi giải tham chiếu con trỏ tượng trưng, angr thay thế nó bằng một địa chỉ cụ thể. Khi thực hiện thực thi tượng trưng, chúng tôi thực sự đã quan sát thấy một số lỗi tham chiếu con trỏ được ném ra bởi angr. Sau khi tiến hành điều tra thêm, chúng tôi nhận thấy rằng trước khi thực hiện tham chiếu con trỏ, một số con trỏ mã tượng trưng không thể được biến chuyển thành các địa chỉ cụ thể. Chúng tôi đã báo cáo vấn đề này cho các nhà phát triển angr. Như đã ghi chú bởi các nhà phát triển angr, vấn đề này đòi hỏi một chiến lược xác định cụ thể. Hiện tại, khi gặp lỗi tham chiếu con trỏ này, chúng tôi phải bỏ qua việc phân tích hàm này.
* Các lệnh không được hỗ trợ. Khi phân tích một số mã máy cụ thể, angr có thể ném ra "lệnh không được hỗ trợ", dẫn đến việc thất bại trong việc phân tích hàm này. Ví dụ, angr không thể xử lý lệnh x86 pavgusb, là một lệnh phổ biến được sử dụng để tối ưu hóa phần mềm xử lý hình ảnh và video như FFmpeg.

10 CÔNG TRÌNH LIÊN QUAN.

* Phần 3 đã đánh giá các nghiên cứu về tương đồng mã nguồn nhị phân dựa trên mạng nơ-ron và khám phá giới hạn của chúng. Trong khi các phương pháp dựa trên mạng nơ-ron gần đây đã cho thấy hỗ trợ tốt cho các cài đặt khó như tối ưu hóa và kiến trúc chéo [14], [15], [16], [23], [77], nghiên cứu của chúng tôi đề xuất sử dụng kiểm tra chức năng chi phí thấp để bù đắp hạn chế bẩm sinh của các phương pháp dựa trên mạng nơ-ron hơn nữa. Chúng tôi cũng đã xem xét việc chứng minh lý thuyết về việc kiểm tra tính tương đồng mã nguồn nhị phân trong Phần 2.2, và so sánh BinUSE với phương pháp tương đồng mã nguồn dựa trên trình biên dịch tượng trưng và trình chuyển đổi mẫu ngẫu nhiên trong Hình 6.
* Tracy [78] phân rã các hàm mã nguồn thành các tracelet để so sánh. Các đặc trưng trích xuất, chủ yếu ở mức cú pháp, có thể gặp khó khăn khi so sánh cài đặt tối ưu hoặc trình biên dịch chéo. Hai công trình theo sau, Esh [26] và GitZ [79], trích xuất strands (tức là các phần tử dòng dữ liệu của các khối cơ bản) để so sánh. Cả hai phương pháp hoạt động tại các ranh giới của một khối cơ bản. Một số cài đặt khó khăn làm gián đoạn tính toàn vẹn của các khối cơ bản và có thể giảm độ chính xác của hai phương pháp này. GitZ nâng mã nguồn nhị phân lên thành VEX IR, sau đó chuyển đổi VEX IR đã nâng lên thành LLVM IR để phân tích. Tuy nhiên, trình chuyển đổi VEX thành LLVM của họ không có sẵn để sử dụng hoặc so sánh.
* Một số phương pháp truyền thống sử dụng các đặc trưng cú pháp chương trình để phân tích tính tương đồng, chẳng hạn như phân phối các chỉ thị, opcode, lệnh hệ thống hoặc một số đặc trưng cấu trúc đồ thị điều khiển [9], [71], [80], [81], [82]. BinDiff, là công cụ tiêu chuẩn công nghiệp, xác định các thành phần mã giống nhau thông qua so sánh đồ thị đồng cấu [71]. Đã được chỉ ra rằng BinDiff không đáng tin cậy đối với các phương pháp làm mờ như làm phẳng dòng điều khiển làm thay đổi cấu trúc dòng điều khiển mạnh mẽ [14], [31].
* Một số công trình sử dụng phân tích động hoặc mẫu ngẫu nhiên đầu vào để phân tích tính tương đồng [83], [84], [85], [86], [87], [88], [89]. Tuy nhiên, như đã so sánh trong Hình 6, một vấn đề nổi bật là sự phủ sóng thấp của các mục tiêu kiểm tra (chẳng hạn như các hàm), làm cho nó không phù hợp với các tình huống thực tế. Phần 4 đã làm rõ rằng trong khi mẫu ngẫu nhiên có thể được sử dụng để nhanh chóng tạo ra mối quan hệ đầu vào-ra đầu ra của hai chương trình thực thi để so sánh, chúng có thể thể hiện tính toàn vẹn tiềm năng thấp và dẫn đến FP [21], [27], [30], [31], [32]. Công trình gần đây [90], bằng cách cải tiến các đặc trưng I/O với các đặc trưng cấu trúc và đặc trưng ý nghĩa cấp cao, thể hiện kết quả thực nghiệm tốt hơn và vượt qua các công cụ tĩnh như BinDiff và Tracy.

11 KẾT LUẬN VÀ CÔNG TRÌNH TƯƠNG LAI.

* Bài báo này xác định các giới hạn chung trong tìm kiếm mã nhị phân dựa trên mạng nơ-ron và đề xuất cải tiến mô hình mạng nơ-ron với việc kiểm tra tương đồng chi phí thấp. Cụ thể, chúng tôi thiết kế BinUSE, một khung phân tích tĩnh thực tế dựa trên việc thực thi tượng trưng không xác định (USE) để kiểm tra tính tương đồng của các hàm mã lệnh. BinUSE tích hợp nhiều tối ưu hóa để giảm thiểu chi phí cho việc tạo ra đường dẫn và ràng buộc đắt giá. Nhờ vậy, BinUSE có thể được sử dụng để đánh dấu và loại bỏ các hàm mã lệnh có sự khác biệt về ngữ nghĩa so với hàm mục tiêu, từ đó cải thiện hiệu suất tìm kiếm mã nhị phân dựa trên mạng nơ-ron. Kết quả thực nghiệm của chúng tôi cho thấy phương pháp được đề xuất cho phép cải tiến chung hiệu quả của các mô hình mạng nơ-ron tiên tiến trong lĩnh vực này, từ đó khiến tìm kiếm hàm mã lệnh trở nên thực tế hơn trong sản xuất. Chúng tôi đã thể hiện khả năng của BinUSE bằng cách so khớp các chương trình từ bộ kiểm tra coreutils và binutils của Linux, cũng như khả năng gia tăng tìm kiếm hàm dễ tổn thương trên phần mềm thế giới thực phức tạp như OpenSSL, Wireshark và FFmpeg.
* Như đã đề cập ở trên, chúng tôi đã phát hành mã nguồn của BinUSE và dữ liệu đánh giá để tái tạo kết quả tại [22]. Trong tương lai, chúng tôi sẽ duy trì BinUSE để hỗ trợ so sánh và mở rộng nghiên cứu. Trên cơ sở của BinUSE, chúng tôi hình dung có thể khám phá và phát triển nhiều ứng dụng phụ thuộc hạ lưu khác. Ví dụ, Phần 8.7 đã làm rõ rằng ngoài việc sử dụng BinUSE để sắp xếp lại chỉ những hàm phù hợp hàng đầu, BinUSE có thể được sử dụng để xác minh từng quyết định cá nhân được thực hiện bởi mạng nơ-ron: chúng tôi sử dụng tiến bộ gần đây trong các kỹ thuật XAI (ví dụ: chú ý của hệ thống thần kinh) để xác định phạm vi các đoạn mã quan trọng góp phần vào quyết định của việc so khớp mã nhị phân dựa trên mạng nơ-ron, sau đó sử dụng BinUSE để xác minh tính tương đồng ngữ nghĩa của các đoạn mã phạm vi đó (thường nhỏ hơn nhiều so với toàn bộ hàm mã lệnh).
* Bên cạnh việc tìm kiếm hàm dễ tổn thương được thể hiện trong bài báo này, chúng tôi dự định tích hợp BinUSE với phương pháp so khớp mã nhị phân dựa trên mạng nơ-ron thực tế để nhanh chóng tìm kiếm các lỗ hổng không xác định trong phần mềm thực thế (thương mại). BinUSE chứa một nỗ lực kỹ thuật cao nhằm phát triển giải pháp xuyên kiến trúc có khả năng phân tích các mã thực thi nhị phân trên cả nền tảng x86 và ARM (aarch64). Với sự phát triển ngày càng nhiều của phần mềm cũ và các thư viện của bên thứ ba trên các thiết bị nhúng và Internet of Things (IoT), dự kiến khả năng tìm kiếm mã nhị phân của BinUSE trên nhiều kiến trúc, trình biên dịch và mã nhị phân được làm mờ sẽ tăng tốc quá trình so khớp các tệp thực thi dễ tổn thương.