Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned.

Tái khám phá Phân tích Tương tự Mã Nhị phân bằng cách Sử dụng Kỹ thuật Kỹ thuật Đặc trưng Hiểu được và Bài học đã học.

Tóm tắt:

* Phân tích tương tự mã nhị phân (BCSA) được sử dụng rộng rãi cho các ứng dụng bảo mật đa dạng, bao gồm phát hiện sao chép, phát hiện vi phạm giấy phép phần mềm và khám phá lỗ hổng. Mặc dù có nhiều nghiên cứu tăng cường về BCSA, nhưng việc thực hiện nghiên cứu mới trong lĩnh vực này vẫn đối diện nhiều thách thức. Một số lý do gồm việc các phương pháp hiện có tập trung chỉ vào kết quả cuối cùng, tức là tăng tỷ lệ thành công của BCSA, bằng cách áp dụng máy học không thể giải thích. Ngoài ra, họ sử dụng bộ kiểm tra của riêng mình mà không chia sẻ mã nguồn hoặc toàn bộ dữ liệu. Cuối cùng, các nhà nghiên cứu thường sử dụng thuật ngữ khác nhau hoặc thậm chí sử dụng cùng một kỹ thuật mà không trích dẫn đúng đắn tài liệu nghiên cứu trước đó, điều này làm cho việc tái tạo hoặc mở rộng công việc trước đó trở nên khó khăn. Để giải quyết các vấn đề này, chúng tôi tiến hành một bước lùi so với trường phái chính và suy ngẫm về các câu hỏi nghiên cứu cơ bản cho BCSA. Tại sao một kỹ thuật hoặc đặc trưng cụ thể cho thấy kết quả tốt hơn các yếu tố khác? Cụ thể, chúng tôi tiến hành nghiên cứu hệ thống đầu tiên về các đặc trưng cơ bản được sử dụng trong BCSA bằng cách sử dụng kỹ thuật đặc trưng giải thích trên một bộ kiểm tra quy mô lớn. Nghiên cứu của chúng tôi đã tiết lộ nhiều thông tin hữu ích về BCSA. Ví dụ, chúng tôi chỉ ra rằng một mô hình giải thích đơn giản với một số đặc trưng cơ bản có thể đạt được kết quả so sánh với các phương pháp dựa trên học sâu gần đây. Hơn nữa, chúng tôi chỉ ra rằng cách chúng ta biên dịch mã nhị phân hoặc tính chính xác của các công cụ phân tích nhị phân cơ bản có thể ảnh hưởng đáng kể đến hiệu suất của BCSA. Cuối cùng, chúng tôi công bố tất cả các mã nguồn và bộ kiểm tra của mình và đề xuất hướng phát triển trong lĩnh vực này để hỗ trợ nghiên cứu tiếp theo.

1 GIỚI THIỆU

* Các nhà lập trình viên thường tái sử dụng mã nguồn hiện có để xây dựng phần mềm mới. Thậm chí việc tìm mã nguồn từ một dự án khác và tái sử dụng mã đó cho mục đích riêng của họ là một thói quen phổ biến [1]. Người phát triển không có kinh nghiệm thậm chí sao chép và dán các mẫu mã từ Internet để đơn giản hóa quá trình phát triển.
* Xu hướng này có ảnh hưởng sâu sắc đến bảo mật và riêng tư phần mềm. Khi một lập trình viên sao chép một chức năng có lỗi từ dự án hiện có, lỗi đó vẫn tồn tại nguyên vẹn ngay cả sau khi nhà phát triển ban đầu đã sửa chữa nó. Hơn nữa, nếu một nhà phát triển trong một công ty phần mềm thương mại vô tình sử dụng mã thư viện từ một dự án mã nguồn mở, công ty có thể bị buộc tội vi phạm giấy phép mã nguồn mở như Giấy phép Công cộng Chung GNU (GPL) [2].
* Tuy nhiên, việc phát hiện các vấn đề này từ mã nhị phân bằng cách sử dụng phân tích tương tự không đơn giản, đặc biệt khi mã nguồn không có sẵn. Điều này là do mã nhị phân thiếu các trừu tượng cấp cao, chẳng hạn như kiểu dữ liệu và chức năng. Ví dụ, không rõ từ mã nhị phân xác định một ô nhớ có đại diện cho một số nguyên, một chuỗi hay kiểu dữ liệu khác. Hơn nữa, xác định các ranh giới chính xác của các chức năng là một thách thức đối với mã nhị phân [3], [4].
* Vì vậy, đo độ tương đồng giữa các mã nhị phân đã trở thành một chủ đề nghiên cứu quan trọng trong nhiều lĩnh vực, chẳng hạn như phát hiện phần mềm độc hại [5], [6], phát hiện sao chép [7], [8], xác định tác giả [9], và khám phá lỗ hổng [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21].
* Tuy nhiên, mặc dù sự quan tâm nghiên cứu đối với phân tích tương đồng mã nhị phân (BCSA) đang tăng cao, chúng tôi nhận thấy rằng việc tiến hành nghiên cứu mới trong lĩnh vực này vẫn đòi hỏi nhiều thách thức đáng kể vì một số lý do.
* Thứ nhất, hầu hết các phương pháp tập trung chỉ vào kết quả cuối cùng mà không xem xét lý do cụ thể đằng sau các phương pháp của họ. Ví dụ, trong quá trình nghiên cứu văn kiện trong lĩnh vực này, chúng tôi quan sát thấy có một xu hướng nghiên cứu nổi bật là áp dụng các kỹ thuật BCSA cho các mã nhị phân chéo kiến trúc và chéo trình biên dịch của cùng một chương trình [11], [12], [13], [15], [16], [19], [22]. Những phương pháp này nhằm đo độ tương đồng giữa hai hoặc nhiều mã nhị phân có vẻ khác biệt được tạo ra từ các trình biên dịch khác nhau nhắm vào các tập lệnh khác nhau. Để đạt được điều này, nhiều phương pháp đã tạo ra các phân tích phức tạp dựa trên học máy để trích xuất ngữ nghĩa của mã nhị phân, giả định rằng ngữ nghĩa của chúng không thay đổi qua các trình biên dịch hoặc kiến trúc mục tiêu. Tuy nhiên, không có phương pháp hiện có nào thực sự chứng minh sự cần thiết của các phân tích ngữ nghĩa phức tạp như vậy. Người ta có thể tưởng tượng rằng một trình biên dịch có thể tạo ra các mã nhị phân có cấu trúc tương tự cho các kiến trúc khác nhau, mặc dù chúng có cú pháp khác nhau. Liệu các trình biên dịch và kiến trúc có thực sự quan trọng đối với BCSA trong trường hợp này? Rõ ràng, câu trả lời cho câu hỏi này khó khăn bởi vì hầu hết các phương pháp hiện có sử dụng các kỹ thuật học máy không thể giải thích được. Hơn nữa, ngay cả việc giải thích tại sao một thuật toán BCSA hoạt động tốt trên một số bài kiểm tra nhưng không trên các bài kiểm tra khác cũng không rõ ràng.
* Thứ hai, mỗi bài báo hiện có về BCSA mà chúng tôi đã nghiên cứu đều sử dụng bộ kiểm tra riêng của họ để đánh giá kỹ thuật được đề xuất, điều này làm cho việc so sánh các phương pháp trở nên khó khăn. Hơn nữa, việc tái tạo kết quả trước đó thường là không khả thi vì hầu hết các nhà nghiên cứu không tiết lộ mã nguồn hoặc tập dữ liệu của họ. Chỉ có 10 trong 43 bài báo mà chúng tôi đã nghiên cứu đã công bố toàn bộ mã nguồn của họ, và chỉ có hai trong số đó đã mở tập dữ liệu hoàn chỉnh của họ.
* Cuối cùng, các nhà nghiên cứu trong lĩnh vực này không sử dụng thuật ngữ thống nhất và thường bỏ sót các tài liệu tham khảo quan trọng đã xuất hiện trong các hội nghị hàng đầu của các lĩnh vực khác. Một số trong số họ thậm chí sử dụng sai kỹ thuật giống nhau mà không trích dẫn tài liệu trước đó đúng cách. Những quan sát này khuyến khích mục tiêu nghiên cứu của chúng tôi, đó là tổng hợp và xem xét các kỹ thuật phổ biến được áp dụng trong lĩnh vực này, đặc biệt là liên quan đến việc tạo ra các đặc trưng.
* Để giải quyết những vấn đề này, chúng tôi lùi bước lại khỏi xu hướng chính và suy ngẫm về các câu hỏi nghiên cứu cơ bản cho BCSA. Như bước đầu tiên, chúng tôi xác định đúng thuật ngữ và phân loại các đặc trưng đã được sử dụng trong các tài liệu trước đó để thống nhất thuật ngữ và xây dựng cơ sở kiến thức cho BCSA. Sau đó, chúng tôi xây dựng một bộ kiểm tra toàn diện và có thể tái tạo được cho BCSA để giúp các nhà nghiên cứu mở rộng và đánh giá các phương pháp của họ một cách dễ dàng. Cuối cùng, chúng tôi thiết kế mô hình kỹ thuật tạo đặc trưng có thể hiểu được và tiến hành một loạt thí nghiệm để điều tra tác động của trình biên dịch, các tùy chọn của chúng và các kiến trúc mục tiêu lên các đặc trưng cú pháp và cấu trúc của các tệp nhị phân kết quả.
* BINKIT là bộ kiểm tra của chúng tôi, bao gồm nhiều bộ kiểm tra hiện có. Nó được tạo ra bằng cách sử dụng các tùy chọn và mục tiêu trình biên dịch chính, bao gồm 8 kiến trúc, 9 trình biên dịch khác nhau, 5 cấp độ tối ưu hóa và nhiều cờ trình biên dịch khác. BINKIT chứa 243,128 tệp nhị phân khác nhau và 36,256,322 hàm được xây dựng cho 1,352 kết hợp khác nhau của các tùy chọn trình biên dịch, trên 51 gói phần mềm thực tế. Chúng tôi cũng cung cấp một kịch bản tự động giúp mở rộng BINKIT để xử lý các kiến trúc hoặc phiên bản trình biên dịch khác nhau. Chúng tôi tin rằng điều này là điều quan trọng vì việc sửa đổi hoặc mở rộng các bộ kiểm tra trước đó không dễ dàng, mặc dù chúng tôi có mã nguồn của chúng. Việc biên dịch chéo các gói phần mềm bằng các tùy chọn trình biên dịch khác nhau gặp nhiều vấn đề về môi trường. Theo hiểu biết của chúng tôi, BINKIT là bộ kiểm tra có thể tái tạo và mở rộng đầu tiên cho BCSA.
* Với bộ kiểm tra của chúng tôi, chúng tôi tiến hành một loạt nghiên cứu nghiêm ngặt về cách cách thức biên dịch có thể ảnh hưởng đến các tệp nhị phân kết quả liên quan đến cấu trúc và hình dạng cú pháp của chúng. Để làm được điều này, chúng tôi thiết kế một mô hình BCSA có thể hiểu được đơn giản, thực chất tính các sự khác biệt tương đối giữa các giá trị đặc trưng BCSA. Sau đó, chúng tôi xây dựng một công cụ BCSA mà chúng tôi gọi là TIKNIB, sử dụng mô hình có thể hiểu của chúng tôi. Với TIKNIB, chúng tôi đã tìm thấy một số hiểu lầm trong lĩnh vực BCSA cũng như những cái nhìn mới cho nghiên cứu tương lai như sau:
* Đầu tiên, xu hướng nghiên cứu hiện tại về BCSA được xây dựng trên một giả định hơi phóng đại: các tệp nhị phân khác nhau một cách tuyệt đối giữa các kiến trúc, loại trình biên dịch hoặc phiên bản trình biên dịch. Tuy nhiên, nghiên cứu của chúng tôi cho thấy điều này không nhất thiết là đúng. Ví dụ, chúng tôi chứng minh rằng các đặc trưng số học đơn giản, chẳng hạn như số cuộc gọi đến/đi từ một hàm, tương tự nhau đáng kể trong các tệp nhị phân được biên dịch cho các kiến trúc khác nhau. Chúng tôi cũng trình bày các đặc trưng cơ bản khác mà đều mạnh mẽ đối với các loại trình biên dịch, phiên bản trình biên dịch và thậm chí các biện pháp che giấu trong nội-bộ. Với những kết quả này, chúng tôi chỉ ra rằng TIKNIB với những đặc trưng đơn giản này có thể đạt được độ chính xác tương đương với các công cụ BCSA hàng đầu, chẳng hạn như VulSeeker, dựa trên một mô hình phức tạp dựa trên học sâu.
* Thứ hai, hầu hết các nhà nghiên cứu tập trung vào việc vector hóa các đặc trưng từ tệp nhị phân, nhưng không tập trung vào việc khôi phục thông tin bị mất trong quá trình biên dịch, chẳng hạn như các kiểu biến. Tuy nhiên, kết quả thực nghiệm của chúng tôi cho thấy tập trung vào việc này có thể rất hiệu quả đối với BCSA. Cụ thể, chúng tôi chỉ ra rằng TIKNIB với thông tin loại được khôi phục đạt được độ chính xác trên 99% trên tất cả các bộ kiểm tra của chúng tôi, đây thực sự là kết quả tốt nhất so với tất cả các công cụ hiện có mà chúng tôi nghiên cứu. Kết quả này nhấn mạnh rằng việc khôi phục thông tin loại từ tệp nhị phân có thể quan trọng như việc phát triển một thuật toán học máy mới cho BCSA.
* Cuối cùng, tính khả giải thích của mô hình giúp nâng cao lĩnh vực bằng cách hiểu sâu hơn về kết quả của BCSA. Ví dụ, chúng tôi trình bày một số vấn đề thực tiễn trong công cụ phân tích nhị phân cơ bản, tức là IDA Pro, mà TIKNIB sử dụng, và thảo luận về cách những lỗi này có thể ảnh hưởng đến hiệu suất của BCSA. Vì bộ kiểm tra của chúng tôi có kết quả thực tế và công cụ của chúng tôi sử dụng mô hình có khả năng giải thích, chúng tôi đã dễ dàng xác định những vấn đề cơ bản này, điều này cuối cùng sẽ hỗ trợ cho các công cụ phân tích nhị phân và toàn bộ lĩnh vực phân tích nhị phân.
* Đóng góp. Tóm lại, đóng góp của chúng tôi là như sau:
* Chúng tôi nghiên cứu các đặc trưng và bộ kiểm tra được sử dụng trong văn bản trước đây về BCSA và làm rõ các câu hỏi nghiên cứu ít được khám phá trong lĩnh vực này.
* Chúng tôi đề xuất BINKIT1, bộ kiểm tra BCSA tái sản xuất và mở rộng đầu tiên. Nó chứa 243,128 tệp nhị phân và 36,256,322 hàm biên dịch cho 1,352 kết hợp khác nhau của trình biên dịch, tùy chọn trình biên dịch và kiến trúc mục tiêu.
* Chúng tôi phát triển một công cụ BCSA, TIKNIB2, sử dụng một mô hình đơn giản có khả năng giải thích. Chúng tôi chứng minh rằng TIKNIB có thể đạt được độ chính xác tương đương với công cụ học sâu hàng đầu. Chúng tôi tin rằng điều này sẽ là một cơ sở để đánh giá nghiên cứu tương lai trong lĩnh vực này.
* Chúng tôi điều tra tính hiệu quả của các đặc trưng cơ bản trong BCSA với TIKNIB trên bộ kiểm tra của chúng tôi và tiết lộ một số quan niệm sai lầm và hiểu biết mới.
* Chúng tôi cung cấp mã nguồn, bộ kiểm tra và dữ liệu thực nghiệm của mình cho cộng đồng để hỗ trợ khoa học mở.

2 PHÂN TÍCH TƯƠNG ĐỒNG MÃ NHỊ PHÂN

* Phân tích tương đồng mã nhị phân (BCSA) là quá trình xác định xem hai đoạn mã đã cho có cùng ý nghĩa tương đương hay không. Thông thường, nó nhận hai đoạn mã làm đầu vào và trả về một điểm tương đồng trong khoảng từ 0 đến 1, trong đó 0 chỉ hai đoạn mã hoàn toàn khác nhau và 1 đồng nghĩa với chúng tương đương. Đoạn mã đầu vào có thể là một hàm [11], [16], [19], [21], [24], [30], [31], [32], hoặc thậm chí là một hình ảnh mã nhị phân toàn bộ [7], [8]. Ngoài ra, so sánh thực tế có thể dựa trên các hàm, ngay cả khi đầu vào là toàn bộ hình ảnh mã nhị phân [12], [13], [15], [23], [33], [34], [35].

A diagram of a diagram

Description automatically generated

Hình 1: Luồng công việc thông thường của phân tích mã nhị phân (phía trên) và so sánh tương đồng (phía dưới) trong phân tích tương đồng mã nhị phân. Các công cụ có thể bỏ qua một số bước.

* Trên mức cao, BCSA thực hiện bốn bước chính như được mô tả dưới đây:
* (S1) Phân tích cú pháp. Đưa vào một đoạn mã nhị phân, ta phân tích mã để thu được dạng mã hóa hoặc Cây cú pháp trừu tượng (AST) của mã, thường được gọi là Biểu diễn trung gian (IR). Bước này tương ứng với phân tích cú pháp trong lý thuyết biên dịch truyền thống, trong đó mã nguồn được phân tích cú pháp thành AST. Nếu mã đầu vào là một tệp nhị phân hoàn chỉnh, ta đầu tiên phân tích tệp dựa trên định dạng tệp của nó và chia nó thành các phần.
* (S2) Phân tích cấu trúc. Bước này phân tích và khôi phục các cấu trúc điều khiển tồn tại trong mã nhị phân đã cho, những cấu trúc này không thể dễ dàng thu được từ giai đoạn phân tích cú pháp (S1). Cụ thể, bước này bao gồm khôi phục các biểu đồ luồng điều khiển (CFGs) và biểu đồ gọi (CGs) trong mã nhị phân. Khi có thông tin về cấu trúc điều khiển, ta có thể sử dụng bất kỳ thuộc tính nào của các cấu trúc điều khiển này làm đặc trưng. Ta phân biệt bước này với phân tích ngữ nghĩa (S3) vì các khung phân tích mã nhị phân thường cung cấp CFGs và CGs miễn phí; các nhà phân tích không cần viết bộ phân tích ngữ nghĩa phức tạp.
* (S3) Phân tích ngữ nghĩa. Sử dụng thông tin về cấu trúc điều khiển thu được từ S2, ta có thể thực hiện các phân tích chương trình truyền thống, chẳng hạn như phân tích luồng dữ liệu và phân tích biểu tượng, trên mã nhị phân để tìm ra ngữ nghĩa cơ bản. Trong bước này, ta có thể tạo ra các đặc trưng biểu diễn ngữ nghĩa chương trình phức tạp, chẳng hạn như cách giá trị đăng ký chảy vào các điểm chương trình khác nhau. Ta cũng có thể nâng cao các đặc trưng thu thập từ S1-S2 cùng với thông tin ngữ nghĩa.
* (S4) Vectorization và So sánh. Bước cuối cùng là vector hóa tất cả thông tin được thu thập từ S1-S3 để tính toán độ tương đồng giữa các mã nhị phân. Bước này về cơ bản dẫn đến một điểm tương đồng giữa 0 và 1.
* Hình 1 mô tả quy trình bốn bước. Ba bước đầu tiên xác định đầu vào cho bước so sánh (S4), thường được gọi là đặc trưng. Một số bước đầu tiên có thể được bỏ qua tùy thuộc vào các đặc trưng được sử dụng. Phương pháp so sánh thực tế trong S4 cũng có thể thay đổi tùy thuộc vào kỹ thuật BCSA. Ví dụ, ta có thể tính khoảng cách Jaccard [39] giữa tập hợp các đặc trưng, tính toán khoảng cách biên tập đồ thị [40] giữa CFGs, hoặc sử dụng các thuật toán học sâu [41], [42]. Tuy nhiên, thành công của bất kỳ thuật toán so sánh nào đều phụ thuộc đáng kể vào các đặc trưng được chọn, vì vậy bài báo này tập trung vào các đặc trưng được sử dụng trong các nghiên cứu trước đó thay vì phương pháp so sánh.
* Trong phần này, trước tiên chúng tôi mô tả các đặc trưng được sử dụng trong các bài báo trước đó và giả định cơ bản của chúng (§2.1). Sau đó, chúng tôi thảo luận về các bài kiểm tra được sử dụng trong những bài báo đó và chỉ ra các vấn đề của chúng (§2.2). Cuối cùng, chúng tôi trình bày một số câu hỏi nghiên cứu được xác định trong quá trình nghiên cứu của chúng tôi (§2.3).
* Phạm vi. Nghiên cứu của chúng tôi tập trung vào 43 bài báo gần đây về BCSA (từ năm 2014 đến 2020) xuất hiện trong 27 tạp chí hàng đầu của các lĩnh vực khác nhau trong ngành khoa học máy tính, chẳng hạn như bảo mật máy tính, kỹ thuật phần mềm, ngôn ngữ lập trình và học máy. Tất nhiên, có rất nhiều bài báo nghiên cứu trong lĩnh vực này, tất cả đều rất quý giá. Tuy nhiên, tập trung của chúng tôi ở đây không phải để thực hiện một cuộc khảo sát hoàn chỉnh về chúng mà là để giới thiệu một xu hướng đáng chú ý và các câu hỏi nghiên cứu cơ bản trong lĩnh vực này, cũng như để trả lời những câu hỏi đó. Chúng tôi đặc biệt tập trung vào các đặc trưng và tập dữ liệu được sử dụng trong những nghiên cứu đó, dẫn đến bốn câu hỏi nghiên cứu chưa được khai thác đầy đủ sẽ được thảo luận trong phần §2.3; mục tiêu của chúng tôi là điều tra những câu hỏi nghiên cứu này bằng cách tiến hành một loạt thí nghiệm nghiêm ngặt.
* Do hạn chế về không gian, chúng tôi loại bỏ các bài báo [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53] được xuất bản trước năm 2014 và những bài báo không đề cập đến các tạp chí hàng đầu, cũng như các công cụ so sánh mã nhị phân [54], [55], [56] được sử dụng trong ngành công nghiệp. Ngoài ra, chúng tôi cũng loại bỏ các bài báo nhằm giải quyết một vấn đề nghiên cứu cụ thể như phát hiện malware, xác định hàm thư viện hoặc xác định các bản vá. Mặc dù nghiên cứu của chúng tôi tập trung chỉ vào các bài báo gần đây, nhưng chúng tôi thấy rằng các đặc trưng chúng tôi nghiên cứu trong bài báo này thực sự rất tổng quát; chúng bao gồm hầu hết các đặc trưng được sử dụng trong các bài báo cũ hơn.

2.1 Các Đặc Trưng Được Sử Dụng Trong Các Nghiên Cứu Trước Đó

* Chúng tôi phân loại các đặc trưng thành hai nhóm dựa trên thời điểm chúng được tạo ra trong quá trình BCSA. Cụ thể, chúng tôi gọi các đặc trưng thu được trước và sau bước phân tích ngữ nghĩa (S3) là các đặc trưng tiền ngữ và đặc trưng ngữ nghĩa, tương ứng. Các đặc trưng tiền ngữ có thể được tạo ra từ S1 hoặc S2, trong khi các đặc trưng ngữ nghĩa có thể được tạo ra từ S3. Chúng tôi tóm tắt cả hai loại đặc trưng được sử dụng trong những nghiên cứu gần đây trong Bảng 1.

2.1.1 Các Đặc Trưng Tiền Ngữ

* Các đặc trưng tiền ngữ đề cập đến các kết quả trực tiếp hoặc gián tiếp của phân tích cú pháp (S1) và phân tích cấu trúc (S2). Do đó, chúng tôi gọi bất kỳ thuộc tính nào của mã nhị phân, có thể được tạo ra mà không cần phân tích ngữ nghĩa, là đặc trưng tiền ngữ. Chúng tôi có thể tiếp tục phân loại các đặc trưng tiền ngữ được sử dụng trong nghiên cứu trước đó dựa trên việc liệu đặc trưng có biểu diễn một số hay không. Chúng tôi gọi các đặc trưng biểu diễn một số là đặc trưng tiền ngữ số học, và các đặc trưng khác là đặc trưng tiền ngữ phi số học. Nửa đầu Bảng 1 tóm tắt chúng.

A screenshot of a graph

Description automatically generated

* Các đặc trưng tiền ngữ số học. Đếm số lần xuất hiện của một thuộc tính cụ thể của một chương trình là phổ biến trong BCSA vì các số liệu này có thể được sử dụng trực tiếp như một vector số học trong bước so sánh sự tương đồng (S4). Chúng tôi phân loại các đặc trưng tiền ngữ số học thành ba nhóm dựa trên mức độ chi tiết của thông tin được yêu cầu để trích xuất chúng.
* Thứ nhất, nhiều nhà nghiên cứu trích xuất các đặc trưng số học từ mỗi khối cơ bản của một đoạn mã mục tiêu [11], [12], [13], [17], [23], [28], [71]. Một người có thể đo lường tần số của các mã opcode (nhãn) [17], [71] hoặc các chỉ thị được nhóm lại dựa trên chức năng của chúng (ví dụ: toán học, logic hoặc chuyển giao điều khiển) [11], [28]. Dạng số học này cũng có thể được xử lý sau bằng máy học [12], [13], [23], [28], như chúng tôi tiếp tục thảo luận trong §2.1.2
* Tương tự như vậy, các đặc trưng số học cũng có thể được trích xuất từ một CFG. Các đặc trưng số học cấp CFG cũng có thể phản ánh thông tin cấu trúc nằm dưới CFG. Ví dụ, một hàm có thể được mã hóa thành một vector số học, bao gồm số lượng các nút (tức các khối cơ bản) và các cạnh (tức dòng điều khiển), cũng như các chỉ thị được nhóm lại trong CFG của nó [11], [28], [61]. Người ta có thể mở rộng các vector số học này bằng cách thêm các đặc trưng bổ sung như số lượng nút liền kề hoặc trung tâm giữa hai CFG [12], [23], [28]. Có thể sử dụng cả khái niệm 3D-CFG [72], trong đó mỗi nút trong một CFG được đặt vào không gian 3D. Ở đây, khoảng cách giữa trọng tâm của hai 3D-CFG có thể biểu thị điểm tương đồng của chúng [18]. Các đặc trưng số học khác có thể là năng lượng đồ thị, độ lệch chuẩn hoặc phức tạp cyclomatic của một CFG [17], [28], [71]. Ngay cả vòng lặp trong một CFG cũng có thể được chuyển thành các đặc trưng số học bằng cách đếm số lượng tiêu đề và đuôi vòng lặp, cũng như số lượng các cạnh tiến và lùi [65].
* Cuối cùng, các phương pháp trước đây sử dụng các đặc trưng số học thu được từ CGs. Chúng tôi gọi chúng là các đặc trưng số học cấp CG. Hầu hết các phương pháp này đo lường số lượng người gọi và người được gọi trong một CG [11], [17], [19], [23], [28], [65], [71], [73]. Khi trích xuất các đặc trưng này, người ta có thể áp dụng một phân tích liên kết giữa các thủ tục bằng cách sử dụng tỷ lệ của bậc vào/ra của các thủ tục gọi nội bộ trong cùng một binary và các thủ tục gọi ngoại của các thư viện nhập khẩu [15], [18], [20], [28]. Điều này tương tự như khái niệm ghép nối [74], trong đó phân tích sự phụ thuộc lẫn nhau giữa các mô-đun phần mềm. Các đặc trưng trích xuất cũng có thể được xử lý sau bằng cách sử dụng máy học [19].
* Đặc trưng trước ngữ nghĩa phi số học. Các thuộc tính của chương trình cũng có thể được sử dụng trực tiếp làm đặc trưng. Phương pháp trực tiếp nhất là so sánh trực tiếp các byte của các tệp nhị phân [6], [53], [75]. Tuy nhiên, người ta thường không xem xét phương pháp này vì so khớp cấp byte không bền bỉ bằng so sánh các thay đổi mã đơn giản. Ví dụ, các ứng dụng chống malware thường sử dụng các chữ ký được viết thủ công bằng cách sử dụng biểu thức chính quy để bắt các phiên bản malware tương tự nhưng cú pháp khác nhau [76]. Các phương pháp gần đây đã cố gắng trích xuất ý nghĩa ngữ nghĩa từ mã nhị phân gốc bằng cách sử dụng mạng neural sâu (DNN) để xây dựng một biểu diễn vector đặc trưng [19], [25].
* Một phương pháp trực tiếp khác liên quan đến việc xem xét các opcode và operand của các lệnh lắp ráp hoặc biểu diễn trung gian của chúng [18], [77]. Các nhà nghiên cứu thường chuẩn hóa các operand [32], [34], [57] vì giá trị thực tế của chúng có thể thay đổi đáng kể giữa các tùy chọn trình biên dịch khác nhau. Các phương pháp gần đây [62], [70] cũng đã áp dụng các kỹ thuật tái tối ưu hóa [78] với cùng một lý do. Để tính toán điểm tương đồng, người ta có thể đo số lượng các yếu tố khớp hoặc khoảng cách Jaccard [15] giữa các nhóm khớp, trong một đơn vị so sánh như cửa sổ trượt [58], khối cơ bản [34], hoặc tracelet [57]. Ở đây, tracelet chỉ định một loạt các khối cơ bản. Mặc dù các phương pháp này sử dụng các đơn vị so sánh khác nhau, người ta có thể điều chỉnh kết quả của chúng để so sánh hai thủ tục hoặc tìm dãy con chung dài nhất [32], [34] trong các thủ tục. Nếu chuyển đổi các lệnh lắp ráp thành một biểu diễn đơn viết tĩnh (SSA), người ta có thể tính toán khoảng cách chỉnh sửa cây giữa các cây biểu diễn SSA như một điểm tương đồng [10]. Các phương pháp gần đây đã đề xuất áp dụng các kỹ thuật phổ biến trong xử lý ngôn ngữ tự nhiên (NLP) để biểu diễn một lệnh lắp ráp hoặc một khối cơ bản dưới dạng một vectơ nhúng, phản ánh ý nghĩa cơ bản của chúng [20], [21], [24], [26], [27], [29]. Cuối cùng, một số đặc trưng có thể được trích xuất trực tiếp từ các hàm. Những đặc trưng này có thể bao gồm tên của các hàm đã nhập và giao của hai đầu vào có thể thể hiện tính tương đồng của chúng [19], [61]. Lưu ý rằng những đặc trưng này cũng có thể hợp tác với các đặc trưng khác.
* Các đặc trưng mà chúng ta có thể thu được từ giai đoạn phân tích ngữ nghĩa (S3) được gọi là đặc trưng ngữ nghĩa. Để thu được các đặc trưng ngữ nghĩa, một phân tích phức tạp như thực hiện biểu tượng [7], [8], [15], [18], [63], đánh giá động của các đoạn mã lệnh [8], [30], [31], [33], [35], [63], [64], [66], [67] hoặc nhúng dựa trên máy học [12], [13], [19], [20], [21], [23], [24], [25], [26], [27], [28], [29] là cần thiết. Có chín đặc trưng ngữ nghĩa chính được sử dụng trong các nghiên cứu trước đây, như được liệt kê trong Bảng 1. Thông thường, việc sử dụng nhiều đặc trưng ngữ nghĩa cùng nhau hoặc kết hợp chúng với các đặc trưng trước ngữ nghĩa là phổ biến.
* Thứ nhất, một phương pháp trực tiếp để biểu diễn ngữ nghĩa của một đoạn mã lệnh cụ thể là sử dụng ràng buộc biểu tượng. Các ràng buộc biểu tượng có thể biểu thị các biến đầu ra hoặc trạng thái của một khối cơ bản [7], một đoạn chương trình [16], [59], [63] hoặc một đường đi [8], [79], [80]. Do đó, sau khi trích xuất các ràng buộc biểu tượng từ một đơn vị so sánh mục tiêu, người ta có thể so sánh chúng bằng cách sử dụng một trình giải SMT.
* Thứ hai, một cách để biểu diễn ngữ nghĩa của mã lệnh là sử dụng các mẫu I/O (Input/Output) [8], [15], [18], [22]. Ý tưởng chính ở đây là hai đoạn mã lệnh giống nhau sẽ tạo ra các mẫu I/O nhất quán và so sánh trực tiếp chúng sẽ hiệu quả về thời gian. Người ta có thể tạo ra các mẫu I/O bằng cách cung cấp đầu vào ngẫu nhiên [8], [22] cho một đoạn mã lệnh, hoặc sử dụng một trình giải SMT để xử lý các ràng buộc biểu tượng của đoạn mã lệnh [15], [18]. Nếu mã lệnh mục tiêu bao gồm cuộc gọi hàm, người ta cũng có thể sử dụng phân tích chéo thủ tục để mô hình chính xác các mẫu I/O [15], [18].
* Thứ ba, hành vi chạy thời gian của một đoạn mã lệnh có thể trực tiếp thể hiện ngữ nghĩa của nó, như đã được trình bày bởi phân tích mã độc truyền thống [81]. Bằng cách thực thi hai hàm mục tiêu với cùng một môi trường thực thi, người ta có thể trực tiếp so sánh các chuỗi lệnh được thực thi [64] hoặc các cạnh CFG được thăm của các hàm mục tiêu [66]. Để so sánh, người ta có thể tập trung vào các hành vi cụ thể quan sát được trong quá trình thực thi [18], [28], [30], [31], [35], [67], [82]: các giá trị đọc/ghi của bộ nhớ stack và heap, các giá trị trả về từ cuộc gọi hàm và các cuộc gọi hàm thư viện/hệ thống được kích hoạt trong quá trình thực thi. Để trích xuất các đặc trưng như vậy, người ta có thể áp dụng kỹ thuật fuzzing [31], [83] hoặc một phương pháp dựa trên giả lập [67]. Hơn nữa, người ta cũng có thể kiểm tra tên cuộc gọi, tham số hoặc chuỗi cuộc gọi cho các cuộc gọi hệ thống [18], [33], [35], [63], [67].
* Danh mục tiếp theo là gán nhãn thủ công cho ngữ nghĩa cấp cao của một chương trình hoặc hàm. Người ta có thể phân loại các hàm thư viện dựa trên chức năng cấp cao của chúng, chẳng hạn như liệu hàm có thao tác trên chuỗi hay xử lý bộ nhớ heap không [15], [18], [61]. Gán nhãn các hàm mật mã trong một đoạn mã lệnh mục tiêu [84] cũng hữu ích vì các phép toán phức tạp của chúng gây cản trở cho việc phân tích các ràng buộc biểu tượng hoặc hành vi của mã lệnh.
* Thứ năm, danh mục tiếp theo là trích xuất các đặc trưng từ một phần chương trình (program slice) [85], vì chúng có thể đại diện cho ngữ nghĩa dòng dữ liệu của nó dưới dạng trừu tượng. Cụ thể, người ta có thể cắt chương trình thành một tập hợp các dây [14], [62]. Ở đây, một dây là một chuỗi lệnh trong cùng một luồng dữ liệu, có thể được thu thập từ cắt ngược. Tiếp theo, các dây này có thể được chuẩn hóa, chuẩn hóa hoặc tối ưu lại để so sánh chính xác hơn [14], [62]. Ngoài ra, người ta có thể băm các dây để so sánh nhanh chóng [68] hoặc trích xuất ràng buộc biểu tượng từ các dây [59]. Người ta cũng có thể trích xuất các đặc trưng từ đồ thị phụ thuộc chương trình (PDG) [86], mà thực chất là một sự kết hợp của đồ thị dòng dữ liệu và CFG, để đại diện cho ngữ nghĩa phức tạp của mã lệnh mục tiêu, bao gồm thông tin về cấu trúc của nó [13].
* Các biến chương trình đã được phục hồi cũng có thể là các đặc trưng ngữ nghĩa. Ví dụ, người ta có thể so sánh sự tương tự của các chuỗi hằng được tham chiếu trong đoạn mã lệnh [11], [12], [17], [23], [28], [61], [65], [71]. Người ta cũng có thể sử dụng kích thước các biến cục bộ, tham số hàm hoặc kiểu trả về của các hàm [11], [28], [61], [69]. Người ta có thể kiểm tra các thanh ghi hoặc biến cục bộ lưu trữ giá trị trả về của các hàm [18].
* Gần đây, một số phương pháp đã sử dụng các vector nhúng, áp dụng các kỹ thuật học máy khác nhau. Sau khi xây dựng một đồ thị luồng điều khiển được đánh dấu (ACFG) [23], đó là một CFG chứa các đặc trưng số học trước ngữ nghĩa trong các khối cơ bản của nó, người ta có thể áp dụng phân cụm phổ [87] để nhóm nhiều ACFG lại hoặc các phương pháp mã hóa phổ biến [88], [89], [90] để nhúng chúng thành một vector [12]. Cùng kỹ thuật có thể được áp dụng cho các đồ thị phụ thuộc chương trình (PDGs) [13]. Trong khi đó, các kỹ thuật NLP gần đây, như Word2Vec [91] hoặc các mô hình mạng neural tích chập [92], có thể được sử dụng để nhúng các byte hoặc lệnh ghi nhớ thành các vector số học [19], [20], [21], [24], [25], [26], [27], [29]. Đối với việc nhúng này, người ta cũng có thể xem xét độ chi tiết cao hơn [20], [24] bằng cách áp dụng các kỹ thuật NLP khác, như nhúng câu [93] hoặc nhúng đoạn văn bản [94]. Lưu ý rằng người ta cũng có thể áp dụng học máy để so sánh các vector nhúng thay vì tạo chúng [60], [68], và Bảng 1 không đánh dấu chúng để sử dụng các vector đã nhúng.

2.1.3 Key Assumptions from Past Research:

* Trong quá trình nghiên cứu tài liệu, chúng tôi phát hiện rằng hầu hết các phương pháp đều dựa nhiều vào các đặc trưng ngữ nghĩa được trích xuất từ giai đoạn (S3), và cho rằng chúng không nên thay đổi theo các trình biên dịch hoặc kiến trúc mục tiêu. Tuy nhiên, không có một phương pháp nào mà rõ ràng giải thích sự cần thiết của các phân tích phức tạp dựa trên ngữ nghĩa như vậy. Họ chỉ tập trung vào kết quả cuối cùng mà không xem xét cẩn thận cách tiếp cận cụ thể trong nghiên cứu của họ.
* Điều này chính là động lực chính cho nghiên cứu của chúng tôi. Mặc dù hầu hết các phương pháp hiện có tập trung vào các phân tích phức tạp, nhưng có thể tồn tại các đặc trưng cơ bản mà chúng tôi đã bỏ qua. Ví dụ, có thể tồn tại các đặc trưng tiền ngữ hiệu quả, có thể vượt trội so với các đặc trưng ngữ nghĩa bất kể trình biên dịch hoặc kiến trúc mục tiêu. Có thể trường hợp các đặc trưng đã biết đã không được đánh giá cẩn thận trên bộ thử nghiệm thích hợp vì chưa có nghiên cứu toàn diện về chúng.
* Ngoài ra, nghiên cứu hiện có giả định về tính chính xác của framework phân tích nhị phân cơ bản, chẳng hạn như IDA Pro [95], là công cụ phổ biến nhất được sử dụng, như được thể hiện trong cột phải nhất của Bảng 2. Tuy nhiên, các CFG tạo ra từ các công cụ này có thể bị sai lệch. Chúng có thể bỏ sót một số khối cơ bản quan trọng, ví dụ như điều này có thể ảnh hưởng trực tiếp đến độ chính xác của các đặc trưng BCSA.
* Thật vậy, cả (S1) và (S2) đều là những vấn đề nghiên cứu khó khăn: có nhiều nỗ lực nghiên cứu để cải tiến độ chính xác của cả hai phân tích. Ví dụ, tự giải mã mã nhị phân chính là một vấn đề không thể giải quyết [96], và việc viết một công cụ nâng cấp mã nhị phân hiệu quả và chính xác là rất thách thức trong thực tế [36], [97]. Xác định các hàm từ mã nhị phân [3], [4], [96], [98], [99], [100], [101] và khôi phục các cạnh điều khiển luồng [102] cho nhánh gián tiếp vẫn là các lĩnh vực nghiên cứu đang hoạt động. Tất cả những quan sát này dẫn đến các câu hỏi nghiên cứu trong §2.3.

2.2 Bộ thử nghiệm đã được sử dụng trong các nghiên cứu trước đây

* Việc sử dụng bộ thử nghiệm chính xác là cần thiết để đánh giá một kỹ thuật BCSA. Do đó, chúng tôi đã nghiên cứu các bộ thử nghiệm được sử dụng trong tài liệu trước đây, như được hiển thị trong Bảng 2. Tuy nhiên, trong quá trình nghiên cứu, chúng tôi đã phát hiện rằng việc đánh giá đúng đắn một kỹ thuật BCSA mới bằng cách sử dụng các bộ thử nghiệm trước đây là vô cùng khó khăn.

A table of information

Description automatically generated

* Đầu tiên, chúng tôi không tìm thấy bất kỳ cặp bài báo nào sử dụng cùng một bộ thử nghiệm. Một số trong số chúng sử dụng các gói như GNU coreutils [15], [30], [31], nhưng các tệp nhị phân, phiên bản và tùy chọn trình biên dịch không giống nhau. Mặc dù không có tiêu chuẩn được biết đến cho việc đánh giá BCSA, việc không có bài báo nào sử dụng cùng một bộ dữ liệu thật đáng ngạc nhiên. Chúng tôi tin rằng điều này một phần là do khó khăn trong việc chuẩn bị cùng một bộ thử nghiệm. Ví dụ, ngay cả khi chúng ta có thể tải về cùng một phiên bản mã nguồn được sử dụng trong một bài báo, việc cross-compile chương trình cho các kiến trúc mục tiêu khác nhau với các tùy chọn trình biên dịch khác nhau là vô cùng khó khăn; nó đòi hỏi nỗ lực đáng kể để thiết lập môi trường. Tuy nhiên, chỉ có hai trong tổng số 43 bài báo mà chúng tôi nghiên cứu hoàn toàn công khai bộ dữ liệu của họ. Ngay cả trong trường hợp đó, việc xây dựng lại hoặc mở rộng bộ thử nghiệm cũng khó khăn do thiếu tập lệnh biên dịch công khai cho bộ thử nghiệm.
* Thứ hai, số lượng các tệp nhị phân được sử dụng trong mỗi bài báo bị giới hạn và có thể không đủ cho việc phân tích. Cột #Binaries trong Bảng 2 tóm tắt số lượng các tệp nhị phân chương trình thu được từ hai nguồn khác nhau: gói ứng dụng và hình ảnh firmware. Vì một gói duy nhất có thể chứa nhiều tệp nhị phân, chúng tôi đã thủ công trích xuất các gói ứng dụng được sử dụng trong mỗi bài báo và đếm số lượng tệp nhị phân trong mỗi gói. Chúng tôi chỉ đếm các tệp nhị phân sau quá trình biên dịch thành công, sao cho các tệp đối tượng được tạo ra trong quá trình biên dịch không được tính vào. Nếu một bài báo không đề cập rõ đến phiên bản gói, chúng tôi đã sử dụng phiên bản gói mới nhất vào thời điểm viết báo và đánh dấu chúng bằng dấu ngoặc đơn. Lưu ý rằng chỉ có 6 trong số 43 bài báo có hơn 10.000 tệp nhị phân, và không có bài báo nào đạt đến 100.000 tệp nhị phân. Firmware có thể chứa nhiều tệp nhị phân, nhưng nó không thể được sử dụng trực tiếp cho BCSA vì không thể tạo ra ground truth mà không có mã nguồn.
* Cuối cùng, các bản kiểm tra trước đó chỉ bao gồm một số ít các trình biên dịch, tùy chọn trình biên dịch và kiến trúc mục tiêu. Một số bài báo thậm chí không mô tả các tùy chọn trình biên dịch hoặc phiên bản gói đã được kiểm tra. Cột Compiler trong bảng thể hiện số lượng phiên bản nhỏ được sử dụng cho mỗi phiên bản trình biên dịch chính. Đáng chú ý, tất cả các bản kiểm tra trừ một bản đều chỉ xem xét ít hơn năm phiên bản trình biên dịch chính khác nhau. Cột Extra trong bảng hiển thị việc sử dụng các tùy chọn trình biên dịch bổ sung cho mỗi bài kiểm tra. Chỉ có một số ít bài kiểm tra xem xét việc ghép chèn hàm (function inlining) và tối ưu hóa thời gian liên kết (Link-Time Optimization, LTO). Không có bài kiểm tra nào xử lý tùy chọn Position Independent Executable (PIE), mặc dù hiện nay nó được sử dụng rộng rãi [103].
* Những quan sát này dẫn đến các câu hỏi nghiên cứu được trình bày trong phần tiếp theo (§2.3) và cuối cùng thúc đẩy chúng tôi tạo ra bộ kiểm tra của riêng mình mà chúng tôi gọi là BINKIT, được hiển thị trong dòng cuối cùng của Bảng 2.

2.3 Các Vấn Đề Nghiên Cứu và Câu Hỏi

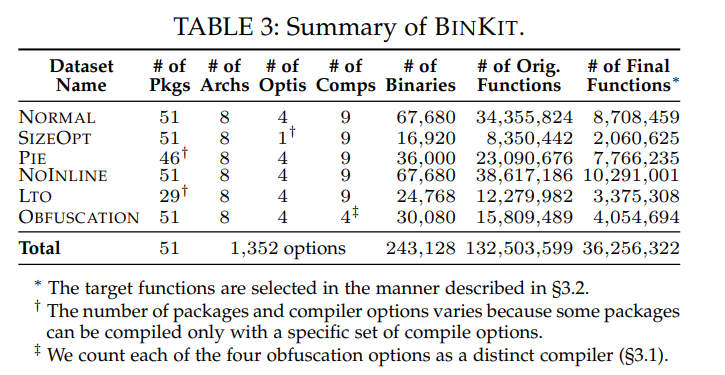
* Chúng tôi tóm tắt một số vấn đề quan trọng quan sát được từ các tài liệu trước và giới thiệu các câu hỏi nghiên cứu được phát sinh từ những vấn đề này. Đầu tiên, không có bài báo nào sử dụng cùng một bộ kiểm tra để đánh giá kỹ thuật của họ, và cách họ đánh giá các kỹ thuật này khác nhau đáng kể. Thứ hai, chỉ có một số ít nghiên cứu phát hành mã nguồn và dữ liệu của họ, làm cho việc tái tạo hoặc cải tiến các công trình hiện có trở nên cực kỳ khó khăn. Hơn nữa, hầu hết các bài báo sử dụng dữ liệu thực tế do con người chọn làm dữ liệu tham chiếu cho đánh giá của họ, điều này có thể dễ dàng gây lỗi. Cuối cùng, các phương pháp tiên tiến nhất hiện tại trong BCSA tập trung vào việc trích xuất các đặc trưng ngữ nghĩa bằng các kỹ thuật phân tích phức tạp (từ §2.1.1 và §2.1.2).
* Những quan sát này tự nhiên dẫn đến các câu hỏi nghiên cứu dưới đây. Lưu ý rằng một số câu hỏi là không thể kết thúc, và chúng tôi chỉ giải quyết một phần trong số đó.
* Câu hỏi nghiên cứu 1 (RQ1): Làm thế nào chúng ta nên thiết lập một bộ kiểm tra quy mô lớn và dữ liệu tham chiếu?
* Một cách để xây dựng các bộ kiểm tra là bằng cách biên dịch thủ công mã nguồn ứng dụng. Tuy nhiên, có quá nhiều phiên bản trình biên dịch, cấp độ tối ưu hóa và tùy chọn cần xem xét khi xây dựng các tệp nhị phân. Do đó, tạo quy trình tự động hóa để xây dựng một bộ kiểm tra quy mô lớn cho BCSA là mong muốn. Cần lưu ý rằng nhiều nghiên cứu hiện tại cũng đã cố gắng xây dựng dữ liệu thực tế từ mã nguồn. Tuy nhiên, số lượng tệp nhị phân và tùy chọn trình biên dịch được sử dụng trong những nghiên cứu đó hạn chế và không đủ cho nghiên cứu dựa trên dữ liệu. Hơn nữa, những nghiên cứu đó không công bố mã nguồn và dữ liệu (§2.2). Trái lại, chúng tôi giới thiệu một kịch bản có thể tự động xây dựng dữ liệu thực tế quy mô lớn từ một tập hợp các gói mã nguồn với các mô tả rõ ràng (§3).
* Câu hỏi nghiên cứu 2 (RQ2): Hiệu quả của các đặc trưng trước cú pháp có giới hạn theo các kiến trúc mục tiêu và tùy chọn trình biên dịch sử dụng không?
* Chúng tôi lưu ý rằng hầu hết các nghiên cứu trước đây cho rằng các đặc trưng trước cú pháp ít hiệu quả hơn đáng kể so với các đặc trưng ngữ nghĩa, vì chúng có thể thay đổi lớn tùy thuộc vào các kiến trúc và tối ưu hóa trình biên dịch cơ bản. Ví dụ, các trình biên dịch có thể thực hiện các kỹ thuật tối ưu hóa dành riêng cho một kiến trúc cụ thể. Thực tế là, 36 trong tổng số 43 bài báo (≈ 84%) chúng tôi nghiên cứu tập trung vào các đặc trưng ngữ nghĩa mới trong phân tích của họ, như được thể hiện trong Bảng 1. Để xác định xem giả định này có hợp lệ hay không, chúng tôi điều tra nó thông qua một loạt các nghiên cứu thí nghiệm nghiêm ngặt. Mặc dù thông tin cấp độ byte có sự khác biệt đáng kể tùy thuộc vào mục tiêu và các kỹ thuật tối ưu hóa, chúng tôi phát hiện rằng một số đặc trưng trước cú pháp, như thông tin cấu trúc thu được từ CFGs, có tính chất rộng rãi tương tự trên các tệp nhị phân khác nhau của cùng một chương trình. Ngoài ra, chúng tôi đã chứng minh rằng việc sử dụng các đặc trưng trước cú pháp như vậy mà không cần phân tích ngữ nghĩa phức tạp có thể đạt được độ chính xác tương đương với phương pháp gần đây dựa trên học sâu với phân tích ngữ nghĩa (§5).
* Câu hỏi nghiên cứu 3 (RQ3): Thông tin gỡ lỗi có thể giúp BCSA đạt được tỷ lệ chính xác cao không?
* Chúng tôi không có thông tin cụ thể về việc bất kỳ nghiên cứu định lượng nào về mức độ ảnh hưởng của thông tin gỡ lỗi đối với độ chính xác của BCSA. Hầu hết các công trình trước đây đơn giản là giả định rằng thông tin gỡ lỗi không có sẵn, nhưng nó giúp đỡ đến mức nào? Các kỹ thuật giải mã có ảnh hưởng như thế nào đến độ chính xác của BCSA? Để trả lời câu hỏi này, chúng tôi đã trích xuất một danh sách các loại hàm từ bộ kiểm tra của chúng tôi và sử dụng chúng để thực hiện BCSA trên bộ dữ liệu của chúng tôi. Bất ngờ, chúng tôi đã đạt được tỷ lệ chính xác cao hơn bất kỳ công trình hiện có nào về BCSA mà không sử dụng bất kỳ phương pháp phức tạp nào (§6).
* Câu hỏi nghiên cứu 4 (RQ4): Chúng ta có thể hưởng lợi từ việc phân tích các trường hợp thất bại của BCSA không?
* Hầu hết các công trình hiện có không phân tích các trường hợp thất bại của họ vì họ phụ thuộc vào các kỹ thuật học máy không thể giải thích được. Tuy nhiên, mục tiêu của chúng tôi là sử dụng một mô hình đơn giản và dễ hiểu để học từ các trường hợp thất bại và đạt được cái nhìn sâu hơn cho nghiên cứu tương lai. Do đó, chúng tôi đã thủ công kiểm tra các trường hợp thất bại bằng cách sử dụng phương pháp có thể giải thích được của chúng tôi và quan sát ba nguyên nhân chung dẫn đến sự thất bại, điều này đã bị bỏ qua trong hầu hết các nghiên cứu trước đó. Đầu tiên, các công cụ phân tích nhị phân COTS thực sự cho kết quả sai. Thứ hai, các trình biên dịch khác nhau cho cùng một kiến trúc có thể khác nhau đáng kể lẫn nhau. Thứ ba, có các đoạn mã cụ thể cho kiến trúc cùng một chức năng. Chúng tôi tin rằng tất cả những quan sát này sẽ giúp định hướng cho các nghiên cứu tương lai (§7).
* Phạm vi phân tích: Trong bài báo này, chúng tôi tập trung vào phân tích tương đồng cấp hàm vì các hàm là đơn vị cơ bản của phân tích nhị phân, và phân tích tương đồng cấp hàm được sử dụng rộng rãi trong các công trình trước đó [11], [16], [19], [21], [24], [30], [31], [32]. Chúng tôi tin rằng người ta có thể dễ dàng mở rộng công việc của chúng tôi để hỗ trợ phân tích tương đồng cấp nhị phân toàn bộ như trong các công trình trước đó [7], [8].

3 XÂY DỰNG BỘ THAM CHIẾU QUY MÔ LỚN VÀ ĐẠT THỰC CHO BCSA (RQ1)

* Xây dựng một bộ tham chiếu quy mô lớn cho BCSA và xác định tham chiếu của nó là một thách thức. Một phương pháp tiềm năng để tạo dữ liệu tham chiếu là xác định thủ công các hàm tương đồng từ các nhị phân hoặc hình ảnh firmware hiện có [10], [57], [59]. Tuy nhiên, điều này yêu cầu kiến thức chuyên môn và thường gặp phải sai sót và tốn thời gian.
* Một phương pháp khác để có được dữ liệu tham chiếu là biên dịch các nhị phân từ mã nguồn hiện có với các tùy chọn biên dịch và kiến trúc mục tiêu khác nhau [13], [15], [16], [23]. Nếu chúng ta biên dịch nhiều nhị phân (với các tùy chọn biên dịch khác nhau) từ cùng mã nguồn, ta có thể xác định hàm nào tương ứng với dòng mã nguồn nào. Thật không may, hầu hết các phương pháp hiện có không công khai bộ tham chiếu của họ hoặc các kịch bản biên dịch được sử dụng để tạo ra chúng (Bảng 2).
* Do đó, chúng tôi trình bày BINKIT, là một bộ tham chiếu toàn diện cho BCSA, cùng với các tập lệnh biên dịch tự động giúp tái tạo và mở rộng nó cho các mục đích nghiên cứu khác nhau. Các phần còn lại của phần này sẽ trình bày chi tiết về BINKIT và thảo luận về cách xác định tham chiếu (RQ1).

3.1 BINKIT: Bộ tham chiếu quy mô lớn cho BCSA

* BINKIT là một bộ tham chiếu toàn diện cho BCSA, bao gồm tổng cộng 243,128 nhị phân được biên dịch từ 51 gói mã nguồn với 1,352 sự kết hợp riêng biệt của trình biên dịch, tùy chọn biên dịch và kiến trúc mục tiêu. Do đó, BINKIT bao phủ hầu hết các bộ tham chiếu được sử dụng trong các phương pháp hiện có, như được thể hiện trong Bảng 2. BINKIT bao gồm các nhị phân được biên dịch cho 8 kiến trúc khác nhau. Ví dụ, chúng tôi sử dụng cả nhị phân little-endian và big-endian cho MIPS để điều tra tác động của endianness. Nó sử dụng 9 phiên bản trình biên dịch khác nhau: GCC v{4.9.4, 5.5.0, 6.4.0, 7.3.0, 8.2.0} và Clang v{4.0, 5.0, 6.0, 7.0}. Chúng tôi cũng xem xét 5 cấp độ tối ưu hóa từ O0 đến O3 cũng như Os, tối ưu hóa kích thước mã nguồn. Cuối cùng, chúng tôi xem xét các tùy chọn PIE, LTO và obfuscation, những tùy chọn ít được khám phá trong BCSA.
* Chúng tôi chọn các gói phần mềm GNU [104] làm mục tiêu biên dịch của chúng tôi vì tính phổ biến và sự tiếp cận: chúng là các ứng dụng thực tế được sử dụng rộng rãi trên hệ thống Linux và mã nguồn của chúng được công khai. Chúng tôi đã thành công trong việc biên dịch 51 gói GNU cho tất cả các kiến trúc mục tiêu và tùy chọn trình biên dịch của chúng tôi.
* Để hỗ trợ so sánh mục tiêu một cách tốt hơn, chúng tôi chia BINKIT thành sáu bộ dữ liệu: NORMAL, SIZEOPT, NOINLINE, PIE, LTO và OBFUSCATION. Tóm tắt mỗi bộ dữ liệu được hiển thị trong Bảng 3. Mỗi bộ dữ liệu chứa các nhị phân được thu thập bằng cách biên dịch các gói GNU với các kết hợp khác nhau của trình biên dịch và các mục tiêu. Không có sự giao nhau giữa các bộ dữ liệu.
* NORMAL bao gồm các nhị phân được biên dịch cho 8 kiến trúc khác nhau với các trình biên dịch và cấp độ tối ưu hóa khác nhau. Chúng tôi không sử dụng các tùy chọn bổ sung khác như PIE, LTO và không sử dụng inline cho bộ dữ liệu này.
* SIZEOPT tương tự như NORMAL ngoại trừ việc sử dụng chỉ tùy chọn tối ưu hóa Os thay vì O0-O3.
* Tương tự, PIE, NOINLINE, LTO và OBFUSCATION không khác biệt so với NORMAL ngoại trừ việc được tạo ra bằng cách sử dụng một cờ bổ sung để bật PIE, tắt tối ưu hóa inline, bật LTO và bật obfuscation tại thời điểm biên dịch, tương ứng.
* PIE làm cho các tham chiếu bộ nhớ trong nhị phân trở nên tương đối với hỗ trợ ASLR. Trên một số kiến trúc, ví dụ như x86, trình biên dịch tiêm các đoạn mã bổ sung để đạt được địa chỉ tương đối. Kết quả, đầu ra biên dịch có thể khác nhau một cách nghiêm trọng. Mặc dù PIE trở thành mặc định trên hầu hết các hệ thống Linux [103], nó chưa được nghiên cứu kỹ về BCSA. Lưu ý rằng chúng tôi không thể biên dịch tất cả 51 gói với tùy chọn PIE được kích hoạt. Do đó, chúng tôi có ít nhị phân hơn trong PIE so với NORMAL.



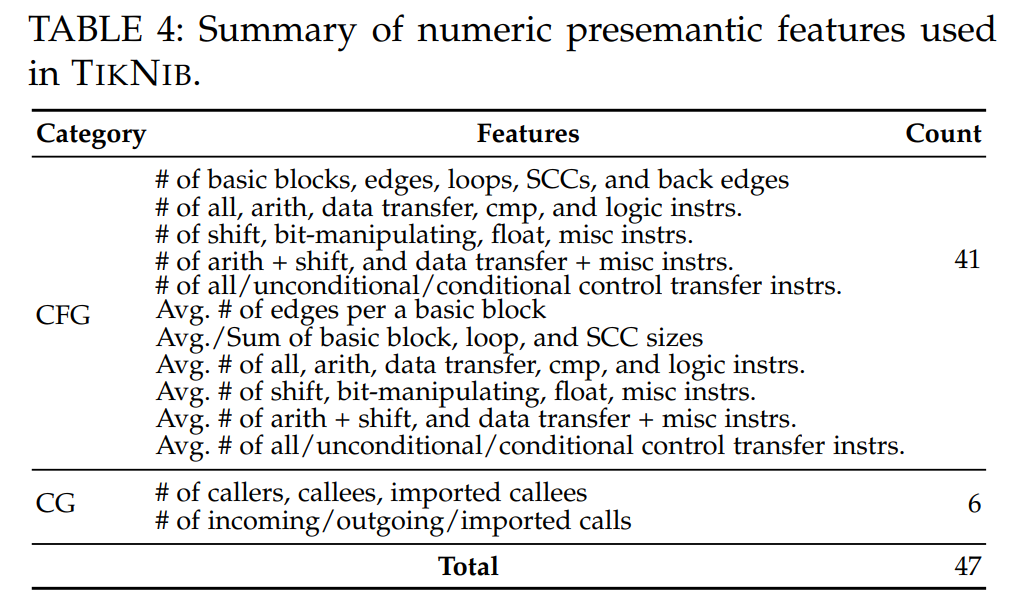
* Function inlining là một kỹ thuật tối ưu hóa nhúng các hàm được gọi vào thân của hàm gọi. Điều này làm cho các tính năng trước ngữ nghĩa biến đổi một cách đáng kể. Do đó, chúng tôi nghiên cứu tác động của việc nhúng hàm vào BCSA bằng cách tắt tối ưu hóa inline một cách rõ ràng với tùy chọn fno-inline.
* LTO là một kỹ thuật tối ưu hóa hoạt động trong quá trình liên kết. Nó loại bỏ các khối mã không cần thiết, giảm số lượng tính năng trước ngữ nghĩa. Tuy nhiên, LTO cũng ít được nghiên cứu trong BCSA. Chúng tôi chỉ có thể biên dịch thành công 29 gói khi tùy chọn LTO được kích hoạt.
* Cuối cùng, bộ dữ liệu OBFUSCATION sử dụng công cụ Obfuscator-LLVM [105] để làm mờ các nhị phân mục tiêu. Chúng tôi chọn Obfuscator-LLVM vì nó là công cụ phổ biến nhất được sử dụng trong việc làm mờ nhị phân và cho phép chúng tôi so sánh trực tiếp hiệu quả của làm mờ bằng trình biên dịch LLVM cơ bản. Chúng tôi sử dụng phiên bản mới nhất của Obfuscator-LLVM với bốn tùy chọn làm mờ: thay thế chỉ thị (SUB), kiểm soát luồng chảy giả (BCF), làm phẳng luồng kiểm soát (FLA) và sự kết hợp của tất cả các tùy chọn này. Chúng tôi coi mỗi tùy chọn là một trình biên dịch riêng biệt, như được thể hiện trong cột Comp của Bảng 3. Một nhị phân duy nhất có thể bị làm mờ nhiều lần. Tuy nhiên, chúng tôi chỉ áp dụng nó một lần. Điều này là do làm mờ một nhị phân nhiều lần có thể tạo ra một nhị phân lớn đáng kể, làm cho việc tiền xử lý cho IDA Pro trở nên tốn thời gian. Ví dụ, khi chúng tôi làm mờ a2ps hai lần với tất cả ba tùy chọn, kích thước của nhị phân biên dịch đạt hơn 30 MB, gấp 30 lần so với bình thường.
* Số lượng gói và số lượng tùy chọn trình biên dịch được sử dụng trong việc biên dịch mỗi bộ dữ liệu khác nhau vì một số gói chỉ có thể được biên dịch với một tập hợp cụ thể của các tùy chọn biên dịch và các mục tiêu. Một số gói không thể biên dịch được vì chúng có mã cụ thể cho từng kiến trúc, chẳng hạn như lồng ghép bằng mã lệnh, hoặc do sử dụng ngữ pháp cụ thể của trình biên dịch. Ví dụ, Clang không hỗ trợ việc bật cả hai tùy chọn LTO và Os cùng một lúc. Có các trường hợp khác nơi các gói có các phụ thuộc mâu thuẫn. Chúng tôi cũng loại bỏ các gói không biên dịch thành công trong vòng 30 phút vì một số gói yêu cầu một lượng thời gian đáng kể để biên dịch. Ví dụ, việc biên dịch gói smalltalk mất hơn 10 giờ với tùy chọn obfuscation được kích hoạt.
* Tóm lại, BINKIT chứa 243.128 nhị phân và tổng cộng 36.256.322 hàm, điều này thực sự nhiều hàng đơn vị lớn hơn nhiều so với các bộ kiểm tra khác xuất hiện trong các nghiên cứu trước đây. Cột Source trong Bảng 2 cho thấy sự khác biệt rõ ràng. BINKIT không bao gồm hình ảnh firmware vì mục tiêu của chúng tôi là xây dựng tự động một bộ kiểm tra với định nghĩa rõ ràng. Bạn có thể mở rộng bộ kiểm tra của chúng tôi với hình ảnh firmware. Tuy nhiên, sẽ mất công sức đáng kể để xác định định nghĩa rõ ràng cho chúng. Để biết thêm chi tiết về từng gói, vui lòng tham khảo Bảng 12 trong Phụ lục.
* Bộ kiểm tra của chúng tôi và các tập lệnh biên dịch có sẵn trên GitHub. Môi trường biên dịch của chúng tôi dựa trên Crosstool-NG [111], GNU Autoconf [112] và Linux Parallels [113]. Qua môi trường này, chúng tôi đã biên dịch toàn bộ bộ dữ liệu của BINKIT trong khoảng 30 giờ trên máy chủ của chúng tôi với 144 nhân Xeon E7-8867v4 của Intel.

3.2 Xây dựng Dữ liệu Thật Sự (Ground Truth)

* Tiếp theo, chúng ta xác định dữ liệu thật sự (ground truth) cho bộ dữ liệu của chúng ta. Đầu tiên, chúng ta xác định các tiêu chí để xác định tính đồng đẳng của hai hàm. Cụ thể, chúng ta kiểm tra xem hai hàm cùng tên có xuất phát từ cùng các tập tin nguồn không và có cùng số dòng mã. Ngoài ra, chúng ta xác minh rằng cả hai hàm đều xuất phát từ cùng một gói và có cùng tên trong các tệp nhị phân của chúng để đảm bảo tính đồng đẳng của chúng. Dựa trên các tiêu chí này, chúng ta đã xây dựng dữ liệu thật sự bằng cách thực hiện các bước sau đây. Trước tiên, chúng ta biên dịch tất cả các tệp nhị phân với thông tin gỡ lỗi bằng cách sử dụng tùy chọn -g. Sau đó, chúng ta sử dụng IDA Pro [95] để xác định các hàm trong các tệp nhị phân đã biên dịch. Tiếp theo, chúng ta đánh nhãn cho từng hàm đã xác định với tên, tên gói, tên nhị phân, cũng như tên của tệp nguồn tương ứng và số dòng mã. Để làm được điều này, chúng ta viết một kịch bản (script) để phân tích thông tin gỡ lỗi từ mỗi tệp nhị phân.
* Dựa vào thông tin này, chúng tôi tiến hành làm sạch bộ dữ liệu của mình để tránh kết quả không chính xác hoặc thiên vị. Trong số các hàm đã xác định, chúng tôi chỉ lựa chọn những hàm nằm trong các phần mã (.text), vì các hàm trong các phần khác có thể không bao gồm mã nhị phân hợp lệ. Ví dụ, chúng tôi bỏ qua các hàm trong các phần Procedure Linkage Table (.plt) vì những hàm này chỉ là các bao gọi đến các hàm bên ngoài và không chứa các phần thân hàm thực sự. Trong bộ dữ liệu của chúng tôi, chúng tôi đã loại bỏ 40% số hàm đã xác định trong bước này.
* Chúng tôi cũng bỏ qua khoảng 4% các hàm được tạo ra bởi trình biên dịch, nhưng không phải bởi các nhà phát triển ứng dụng. Chúng tôi có thể dễ dàng xác định các hàm nội tại của trình biên dịch như vậy bằng cách kiểm tra các tệp nguồn và số dòng tương ứng. Ví dụ, GCC sử dụng các hàm nội tại như \_\_udivdi3 trong libgcc2.c hoặc \_\_aeabi\_uldivmod trong bpabi.S để tạo mã tối ưu hóa cao.
* Ngoài ra, chúng tôi đã loại bỏ các hàm trùng lặp trong cùng một dự án/gói. Hai tệp nhị phân khác nhau thường chia sẻ cùng mã nguồn, đặc biệt là khi chúng thuộc cùng một dự án/gói. Ví dụ, gói GNU coreutils chứa 105 tệp thực thi khác nhau mà chia sẻ 80% số hàm chung. Chúng tôi đã loại bỏ các hàm trùng lặp trong từng gói bằng cách kiểm tra tên tệp nguồn và số dòng tương ứng. Ngoài ra, các trình biên dịch cũng có thể tạo ra nhiều bản sao của cùng một hàm trong một tệp nhị phân duy nhất do tối ưu hóa. Những hàm này chia sẻ cùng mã nguồn nhưng có sự khác biệt trong các dạng nhị phân của chúng. Ví dụ, một số phần của mã nhị phân được loại bỏ hoặc được sắp xếp lại cho mục đích tối ưu hóa. Bởi vì những hàm này chia sẻ một phần lớn mã, nếu xem xét tất cả chúng, sẽ dẫn đến kết quả thiên vị. Để tránh điều này, chúng tôi chỉ lựa chọn một bản sao cho mỗi hàm trong các thử nghiệm của chúng tôi. Bước này đã loại bỏ khoảng 54% số hàm còn lại. Cột cuối cùng của Bảng 3 báo cáo kết quả đếm cuối cùng, đó là số hàm duy nhất.
* Bằng cách thực hiện tất cả các bước trên, chúng tôi có thể tự động xây dựng dữ liệu thật sự (ground truth) quy mô lớn. Tổng thời gian mất để xây dựng dữ liệu thật sự cho tất cả các bộ dữ liệu của chúng tôi là 13,300 giây. Bằng cách tận dụng dữ liệu thật sự này, chúng tôi tiếp tục điều tra các câu hỏi nghiên cứu còn lại (tức là RQ2-RQ4) trong các phần tiếp theo. Để khuyến khích nghiên cứu tiếp theo, chúng tôi đã công bố toàn bộ các bộ dữ liệu và mã nguồn của mình.

4 BUILDING AN INTERPRETABLE MODEL:

* Trước đó, các kỹ thuật BCSA (Binary Code Similarity Analysis) tập trung vào đạt được độ chính xác cao hơn bằng cách tận dụng những tiến bộ gần đây trong các kỹ thuật học sâu [12], [13], [19], [25]. Điều này thường đòi hỏi xây dựng một mô hình phức tạp, không dễ hiểu và gây trở ngại cho các nhà nghiên cứu khi suy luận về kết quả BCSA và trả lời các câu hỏi cơ bản liên quan đến BCSA. Do đó, chúng tôi thiết kế một mô hình có thể hiểu được cho BCSA để trả lời các câu hỏi nghiên cứu và triển khai TIKNIB, một công cụ BCSA sử dụng mô hình này. Phần này minh họa cách chúng tôi thu được một mô hình như vậy và cách chúng tôi thiết lập môi trường thử nghiệm của chúng tôi.



4.1 Tổng quan về TIKNIB

* Một cách tổng quan, TIKNIB sử dụng một tập hợp các đặc trưng tiền từ ngữ được sử dụng rộng rãi trong các nghiên cứu trước đây để đánh giá lại hiệu quả của các đặc trưng tiền từ ngữ (RQ2). Nó đánh giá mỗi đặc trưng trong hai hàm đầu vào, dựa trên độ đo điểm tương đồng của chúng (§4.3), mà đo trực tiếp sự khác biệt giữa giá trị của mỗi đặc trưng. Nói cách khác, nó bắt giữ cách mà mỗi đặc trưng khác nhau giữa các tùy chọn biên dịch khác nhau. Lưu ý rằng TIKNIB được thiết kế có chủ ý để đơn giản để chúng ta có thể trả lời các câu hỏi nghiên cứu được trình bày trong §2.3. Mặc dù phương pháp của chúng tôi đơn giản, TIKNIB vẫn đem lại tỷ lệ chính xác cao so với các công cụ tiên tiến hiện tại (§5.2). Chúng tôi không bàn luận ở đây rằng TIKNIB là thuật toán BCSA tốt nhất.

4.2 Các đặc trưng được sử dụng trong TIKNIB

* Nhớ lại từ RQ2, một trong những mục tiêu của chúng tôi là tái xem xét khả năng của các đặc trưng tiền từ ngữ. Do đó, chúng tôi tập trung vào việc lựa chọn các đặc trưng tiền từ ngữ khác nhau đã được sử dụng trong các nghiên cứu BCSA trước đó thay vì tạo ra các đặc trưng mới.
* Tuy nhiên, việc tạo ra một tập hợp đầy đủ các đặc trưng không phải là điều dễ dàng vì hai lý do sau đây. Thứ nhất, có rất nhiều đặc trưng hiện có có tính tương đồng với nhau, như đã thảo luận trong §2. Thứ hai, một số đặc trưng yêu cầu kiến thức cụ thể về lĩnh vực, nhưng kiến thức này không được công khai. Ví dụ, một số bài báo hiện có [11], [12], [13], [17], [18], [23], [61], [65] phân loại các chỉ thị thành các nhóm ngữ nghĩa. Tuy nhiên, việc nhóm các chỉ thị là một nhiệm vụ chủ quan, và không có tiêu chuẩn nào được biết đến cho nó. Hơn nữa, hầu hết các công trình hiện có không công bố thuật toán phân nhóm của họ.
* Chúng tôi đối mặt với những thách thức này bằng cách (1) trích xuất thủ công các đặc trưng tiền từ ngữ đại diện và (2) công khai mã nguồn của quá trình trích xuất đặc trưng của chúng tôi. Cụ thể, chúng tôi tập trung vào các đặc trưng tiền từ ngữ dạng số. Vì những đặc trưng này được biểu diễn dưới dạng số, mối quan hệ giữa giá trị của chúng qua các tùy chọn biên dịch khác nhau có thể dễ dàng quan sát được.
* Bảng 4 tóm tắt các đặc trưng đã được chọn. Tập đặc trưng của chúng tôi bao gồm các đặc trưng dạng số ở mức đồ thị điều khiển (Control Flow Graph - CFG) và đồ thị gọi (Call Graph - CG) vì chúng có thể hiệu quả tiết lộ các thay đổi cấu trúc trong mã nguồn mục tiêu. Cụ thể, chúng tôi sử dụng các đặc trưng liên quan đến các khối cơ bản (basic blocks), các cạnh CFG, các vòng lặp tự nhiên và các thành phần kết nối mạnh (Strongly Connected Components - SCCs) từ CFG bằng cách sử dụng NetworkX [114]. Chúng tôi cũng phân loại các chỉ thị thành một số nhóm ngữ nghĩa dựa trên sự nhận định cẩn thận của chúng tôi bằng cách tham khảo các hướng dẫn tham chiếu [115], [116], [117] và sử dụng phân loại nội bộ của Capstone [118]. Tiếp theo, chúng tôi đếm số lượng chỉ thị trong mỗi nhóm ngữ nghĩa cho từng hàm (tức là CFG).
* Ngoài ra, chúng tôi sử dụng sáu đặc trưng từ CG. Số lượng các hàm gọi và các hàm được gọi đại diện cho số lượng các cạnh đi và đến từ CG.
* Để trích xuất các đặc trưng này, chúng tôi thực hiện các bước sau đây. Trước tiên, chúng tôi tiền xử lý các tệp nhị phân trong BINKIT bằng IDA Pro [95]. Sau đó, chúng tôi tạo dữ liệu thật sự (ground truth) cho các tệp nhị phân này như chúng tôi đã mô tả trong §3.2. Đối với những hàm mà chúng tôi có dữ liệu thật sự, chúng tôi trích xuất các đặc trưng đã đề cập ở trên. Bảng 5 hiển thị thời gian dành cho mỗi bước trong các bước này. Quá trình tiền xử lý bằng IDA chiếm phần lớn thời gian, vì IDA thực hiện các phân tích nội bộ khác nhau. Trong khi đó, quá trình trích xuất đặc trưng chỉ tốn ít thời gian hơn vì nó chỉ hoạt động trên kết quả đã được tính toán từ bước tiền xử lý trước đó.

4.3 Scoring Metric:

* Độ đo tính điểm của chúng tôi dựa trên tính toán sự khác biệt tương đối [119] giữa các giá trị đặc trưng. Cho hai hàm A và B, chúng ta ký hiệu giá trị của đặc trưng f cho mỗi hàm là Af và Bf, tương ứng. Hãy nhớ rằng bất kỳ đặc trưng nào trong TIKNIB đều có thể được biểu diễn dưới dạng một số. Chúng ta có thể tính toán sự khác biệt tương đối δ giữa hai giá trị đặc trưng như sau:
* A black and white image of a mathematical equation

  Description automatically generated
* Giả sử chúng ta có N đặc trưng riêng biệt (f1, f2, · · · , fN) trong tập đặc trưng của chúng ta. Chúng ta có thể định nghĩa điểm tương đồng s giữa hai hàm A và B bằng cách lấy trung bình của các sự khác biệt tương đối cho tất cả các đặc trưng như sau:
* A math equation with a line and a plus and a plus

  Description automatically generated
* Mặc dù mỗi đặc trưng số học có thể có một phạm vi giá trị khác nhau, TIKNIB có thể xử lý chúng hiệu quả bằng cách sử dụng sự khác biệt tương đối, bằng cách biểu diễn sự khác biệt của mỗi đặc trưng với một giá trị từ 0 đến 1. Do đó, điểm tương đồng s luôn nằm trong khoảng từ 0 đến 1.
* Hơn nữa, chúng ta có thể hiểu và giải thích một cách trực quan kết quả của BCSA bằng cách sử dụng độ đo tính điểm của chúng tôi. Ví dụ, giả sử có hai hàm A và B được tạo ra từ cùng mã nguồn, một hàm có sử dụng tùy chọn biên dịch X và một hàm không sử dụng tùy chọn biên dịch X. Nếu sự khác biệt tương đối của giá trị đặc trưng f giữa hai hàm là nhỏ, điều đó ngụ ý rằng f là một đặc trưng mạnh mẽ chống lại tùy chọn biên dịch X.
* Trong bài báo này, chúng tôi tập trung chỉ vào sự khác biệt tương đối đơn giản, thay vì khám phá các mối quan hệ phức tạp giữa các đặc trưng để có tính giải thích. Tuy nhiên, chúng tôi tin rằng phương pháp của chúng tôi có thể là một bước đệm để xây dựng các mô hình có tính giải thích cải tiến hơn để hiểu những mối quan hệ phức tạp như vậy.

A table with numbers and text

Description automatically generated

4.4 Lựa chọn Đặc trưng

* Dựa trên độ đo tính điểm của chúng tôi, chúng tôi thực hiện tiền xử lý nhẹ để chọn các đặc trưng hữu ích cho BCSA vì một số đặc trưng có thể không giúp phân biệt giữa các hàm. Để đo chất lượng của một tập đặc trưng đã cho, chúng tôi tính diện tích dưới đường cong đặc trưng hoạt động của thuật toán (Receiver Operating Characteristic - ROC) (tức là ROC AUC) của các mô hình được tạo ra.
* Giả sử chúng ta được cung cấp một bộ dữ liệu trong BINKIT, được tạo ra từ mã nguồn chứa N hàm duy nhất. Tổng cộng, chúng ta có tối đa N · M hàm trong bộ dữ liệu của chúng ta, trong đó M là số lượng các tổ hợp tùy chọn biên dịch được sử dụng để tạo bộ dữ liệu. Số hàm thực tế có thể ít hơn N · M do việc ghép hàm (function inlining).
* Đối với mỗi hàm duy nhất λ, chúng tôi ngẫu nhiên chọn hai hàm khác với các điều kiện sau. (1) Một hàm dương tính (true positive - TP), λ\_TP, được tạo ra từ cùng mã nguồn như λ, nhưng với các tùy chọn biên dịch khác, và (2) một hàm tiêu cực tính (true negative - TN), λ\_TN, được tạo ra từ mã nguồn khác với mã được sử dụng để tạo λ, nhưng sử dụng cùng tùy chọn biên dịch như λ\_TP. Chúng tôi tạo ra các cặp như vậy cho mỗi hàm duy nhất, từ đó có được khoảng 2 · N cặp hàm. Sau đó, chúng tôi tính toán các điểm tương đồng cho các hàm trong mỗi cặp và tính AUC của chúng.
* Chúng tôi lưu ý rằng phương pháp này đã được sử dụng trong các công trình trước đây [12], [13]. Chúng tôi đã chọn phương pháp này vì nó cho phép chúng tôi phân tích hiệu quả xu hướng trên một tập dữ liệu quy mô lớn. Một cách khác, người ta có thể xem xét đánh giá theo top-k [12], [13], [14], [20] hoặc precision@k [12], [20], nhưng phương pháp này đòi hỏi quá nhiều tính toán với độ phức tạp: O((N · M)^2).
* Rất tiếc, không có thuật toán hiệu quả nào để lựa chọn tập con đặc trưng tối ưu; đây là một vấn đề thuộc lớp NP-khó hiểu biết [120]. Do đó, chúng tôi sử dụng một thuật toán lựa chọn đặc trưng tham lam [121]. Bắt đầu từ một tập rỗng F, chúng tôi xác định xem chúng ta có thể thêm một đặc trưng vào F để tăng AUC hay không. Đối với mỗi đặc trưng có thể, chúng tôi thực hiện một phép hợp với F và tính AUC tương ứng. Sau đó, chúng tôi chọn đặc trưng nào làm tăng AUC lớn nhất và cập nhật F để bao gồm đặc trưng đã chọn. Chúng tôi lặp lại quá trình này cho đến khi AUC không còn tăng nữa bằng cách thêm một đặc trưng mới. Mặc dù phương pháp của chúng tôi không đảm bảo tìm ra một giải pháp tối ưu, nó vẫn cung cấp kết quả có ý nghĩa từ mặt thực nghiệm, như chúng tôi mô tả trong các phần tiếp theo.

4.5 Thiết lập Thí nghiệm

* Cho tất cả các thí nghiệm trong nghiên cứu này, chúng tôi thực hiện kiểm tra chéo 10-fold trên mỗi bộ kiểm tra. Khi chia bộ dữ liệu kiểm tra, chúng tôi đảm bảo các hàm chia sẻ cùng mã nguồn (tức là tên tệp nguồn và số dòng) sẽ nằm trong một tập huấn luyện hoặc kiểm tra, nhưng không nằm cùng lúc trong cả hai tập. Đối với mỗi lượt kiểm tra, trong giai đoạn học tập, tức là giai đoạn lựa chọn đặc trưng, chúng tôi chọn lên đến 200,000 hàm từ một tập huấn luyện và tiến hành lựa chọn đặc trưng, vì huấn luyện hàng triệu hàm sẽ mất một lượng thời gian đáng kể. Giới hạn số lượng hàm để huấn luyện có thể làm giảm chất lượng kết quả cuối cùng. Tuy nhiên, khi chúng tôi thử nghiệm số lượng hàm từ 100,000 đến 1,000,000, kết quả vẫn khá ổn định. Trong giai đoạn kiểm tra độ chính xác, chúng tôi kiểm tra tất cả các hàm trong tập kiểm tra mà không có bất kỳ mẫu nào. Do đó, sau khi thực hiện kiểm tra chéo 10-fold, tất cả các hàm trong bộ dữ liệu mục tiêu được kiểm tra ít nhất một lần.
* Chúng tôi chạy tất cả các thí nghiệm trên một máy chủ được trang bị bốn CPU Intel Xeon E7-8867v4 2.40 GHz (tổng cộng 144 nhân), 896 GB RAM DDR4 và 8 TB ổ cứng SSD. Chúng tôi cài đặt Ubuntu 18.04.5 LTS cùng với IDA Pro v6.95 [95] trên máy chủ. Đối với việc lựa chọn đặc trưng và so sánh sự tương đồng, chúng tôi sử dụng Python scikit-learn [122], SciPy [123] và NumPy [124].

5 PHÂN TÍCH ĐẶC TRƯNG TRƯỚC NGHĨA (RQ2)

* Bây giờ chúng tôi trình bày kết quả thí nghiệm của chúng tôi sử dụng TIKNIB trên các đặc trưng trước nghĩa (§4.2) để trả lời RQ2 (§2.3). Với việc phân tích toàn diện các đặc trưng này, chúng tôi thu được một số hiểu biết hữu ích cho nghiên cứu tương lai. Trong phần này, chúng tôi thảo luận về những phát hiện và bài học mà chúng tôi đã học được.

5.1 Kết quả phân tích

* Để phân tích tác động của các tùy chọn biên dịch và kiến trúc mục tiêu khác nhau đối với BCSA, chúng tôi thực hiện tổng cộng 72 thử nghiệm bằng cách sử dụng TIKNIB. Chúng tôi thực hiện các thử nghiệm trên bộ kiểm tra của chúng tôi, BINKIT, với ground truth mà chúng tôi xây dựng ở §3. Bảng 6 mô tả các kết quả thực nghiệm trong đó mỗi cột tương ứng với một thử nghiệm chúng tôi thực hiện. Lưu ý rằng chúng tôi chỉ trình bày 26 trong số 72 thử nghiệm vì hạn chế không gian. Trừ khi có ghi chú khác, tất cả các thử nghiệm đều được thực hiện trên bộ dữ liệu NORMAL. Như đã mô tả ở §4.4, chúng tôi đã chuẩn bị các tập huấn luyện 10-fold cho mỗi thử nghiệm.
* Chúng tôi chia các thử nghiệm thành bảy nhóm tùy theo mục đích của chúng, như được hiển thị ở hàng đầu của bảng. Ví dụ, nhóm Arch chứa một tập các thử nghiệm để đánh giá mỗi đặc trưng trên các kiến trúc mục tiêu khác nhau.
* Đối với mỗi thử nghiệm, chúng tôi chọn các cặp hàm để huấn luyện và kiểm tra như đã mô tả ở §4.4. Đó là, đối với một hàm λ, chúng tôi chọn các hàm tương ứng của nó (tức là λ\_TP và λ\_TN). Do đó, N hàm sẽ tạo ra 2·N cặp hàm. Dòng đầu tiên ( 1 ) của Bảng 6 cho thấy số lượng các cặp hàm cho mỗi thử nghiệm. Khi chọn các cặp này, chúng tôi chọn mục tiêu các tùy chọn dựa trên mục tiêu của mỗi thử nghiệm. Ví dụ, chúng tôi kiểm tra ảnh hưởng của việc thay đổi kiến trúc mục tiêu từ x86 sang ARM (cột x86 vs. ARM của Bảng 6). Đối với mỗi hàm λ trong các tệp nhị phân x86 của bộ dữ liệu của chúng tôi, chúng tôi chọn cả λ\_TP và λ\_TN từ các tệp nhị phân ARM được biên dịch với cùng tùy chọn biên dịch như λ. Nói cách khác, chúng tôi giữ nguyên tất cả các tùy chọn khác, ngoại trừ kiến trúc mục tiêu để chọn λ\_TP và λ\_TN để chúng tôi có thể tập trung vào mục tiêu kiểm tra của chúng tôi. Quy tắc tương tự áp dụng cho các cột khác. Đối với các cột Rand., chúng tôi thay đổi tất cả các tùy chọn biên dịch trong nhóm một cách ngẫu nhiên để tạo ra các cặp hàm.
* Dòng thứ hai ( 2 ) của Bảng 6 trình bày thời gian được tiêu tốn cho việc huấn luyện và kiểm tra trong mỗi thử nghiệm, loại bỏ thời gian để tải dữ liệu hàm vào bộ nhớ. Thời gian trung bình tiêu tốn cho một hàm là nhỏ hơn 1 ms.
* Mỗi ô trong dòng thứ ba ( 3 ) của Bảng 6 đại diện cho giá trị trung bình của δ(λf , λTNf) - δ(λf , λTPf) cho đặc trưng f, mà chúng tôi gọi là khoảng cách TP-TN của f. Khoảng cách TP-TN này đo lường sự tương đồng giữa λTP và λ, cũng như sự khác biệt giữa λTN và λ, dưới góc nhìn của đặc trưng mục tiêu. Do đó, khi khoảng cách của một đặc trưng lớn hơn, khả năng phân biệt của nó đối với BCSA cao hơn. Khi chúng tôi thực hiện kiểm tra chéo 10-fold cho mỗi thử nghiệm, chúng tôi đánh dấu các ô màu xám khi đặc trưng tương ứng được chọn trong tất cả mười lần thử nghiệm. Các đặc trưng như vậy cho thấy khoảng cách TP-TN tương đối cao hơn so với các đặc trưng khác trong mỗi thử nghiệm. Chúng tôi cũng trình bày khoảng cách TP-TN trung bình trong dòng thứ tư ( 4 ) của bảng.
* Số lượng đặc trưng được chọn trung bình trong mỗi thử nghiệm được hiển thị ở dòng thứ năm ( 5 ) của Bảng 6. Một số đặc trưng trước nghĩa có thể đạt được AUC và độ chính xác trung bình (AP) cao, như được hiển thị ở dòng thứ sáu ( 6 ) và dòng thứ bảy ( 7 ) của cùng bảng. Bây giờ, chúng tôi tổng kết những quan sát của mình như sau.

5.1.1 Tối ưu hóa có tác động lớn

* Nhiều nhà nghiên cứu đã tập trung vào việc thiết kế một mô hình cho BCSA qua các kiến trúc khác nhau [11], [15], [18], [22], [33]. Tuy nhiên, kết quả thực nghiệm của chúng tôi cho thấy kiến trúc có thể không phải là yếu tố quan trọng nhất đối với BCSA. Thay vào đó, mức tối ưu hóa là yếu tố có tác động lớn nhất đối với sự khác biệt tương đối giữa các đặc trưng trước nghĩa. Đặc biệt, chúng tôi đo lường khoảng cách TP-TN trung bình của tất cả các đặc trưng trước nghĩa cho mỗi thử nghiệm (dòng "Avg. of TP-TN Gap" của bảng) và thấy rằng khoảng cách trung bình của thử nghiệm O0 vs. O3 (0.41) nhỏ hơn so với thử nghiệm x86 vs. ARM (0.46) và thử nghiệm x86 vs. MIPS (0.42). Hơn nữa, thử nghiệm ngẫu nhiên về mức tối ưu hóa (cột "Rand." của nhóm "Opt Level") cho thấy AUC thấp nhất (0.96) so với nhóm kiến trúc và trình biên dịch (0.98). Những kết quả này xác nhận rằng các trình biên dịch có thể tạo ra các nhị phân khác biệt lớn tùy thuộc vào các kỹ thuật tối ưu hóa sử dụng; do đó, sự biến đổi giữa các nhị phân do tối ưu hóa lớn hơn nhiều so với kiến trúc mục tiêu trên bộ dữ liệu của chúng tôi.

5.1.2 Phiên bản trình biên dịch có tác động nhỏ

* Xấp xỉ một phần ba các thử nghiệm trước đây được hiển thị trong Bảng 2 sử dụng nhiều phiên bản của cùng một trình biên dịch. Tuy nhiên, chúng tôi đã tìm thấy rằng ngay cả các phiên bản chính của cùng một trình biên dịch cũng tạo ra các nhị phân tương tự nhau. Nói cách khác, phiên bản trình biên dịch không ảnh hưởng quá nhiều đến các đặc trưng trước nghĩa. Mặc dù Bảng 6 không bao gồm tất cả các thử nghiệm chúng tôi đã thực hiện do hạn chế không gian, nhưng rõ ràng từ cột "Compiler" rằng hai thử nghiệm giữa hai phiên bản khác nhau của cùng một trình biên dịch, tức là GCC v4 vs. GCC v8 và Clang v4 vs. Clang v7, có các khoảng cách TP-TN cao hơn nhiều (0.52) so với các thử nghiệm khác và AUC của chúng gần 1.0.

5.1.3 GCC và Clang có đặc điểm đa dạng

* Ngược lại, thử nghiệm GCC vs. Clang đã dẫn đến khoảng cách TP-TN thấp nhất (0.44) và AUC (0.97) trong nhóm "Compiler". Điều này có thể do mỗi trình biên dịch sử dụng một back-end khác nhau, từ đó tạo ra các nhị phân khác nhau. Một vấn đề tiềm năng khác là các kỹ thuật bên trong từng mức tối ưu hóa có thể thay đổi tùy thuộc vào trình biên dịch. Chúng tôi sẽ chi tiết vấn đề này trong §7.2.

A black and white text with numbers

Description automatically generated

5.1.4 Nhị phân ARM gần với nhị phân x86 hơn là MIPS

* Các thử nghiệm trong nhóm "Arch" đo lường tác động của các kiến trúc mục tiêu với bộ dữ liệu "NORMAL". Tổng thể, kiến trúc mục tiêu không có nhiều ảnh hưởng đến tỷ lệ chính xác. Các AUC đều lớn hơn 0.98 trong tất cả các trường hợp. Đáng ngạc nhiên, thử nghiệm x86 vs. ARM có khoảng cách TP-TN cao nhất (0.46) và AUC (1.0), cho thấy các đặc trưng trước nghĩa của các nhị phân x86 và ARM gần giống nhau, mặc dù là các kiến trúc khác nhau. Thử nghiệm ARM vs. MIPS cho thấy khoảng cách TP-TN thấp hơn (0.43) và AUC (0.98) mặc dù cả hai đều là kiến trúc RISC. Ngoài ra, tác động của kích thước từ (bits) và endianess (thứ tự lưu trữ byte) cũng tương đối nhỏ. Tuy nhiên, chúng tôi không thể loại trừ khả năng rút trích đặc trưng của chúng tôi cho các nhị phân MIPS bị sai sót. Chúng tôi sẽ thảo luận vấn đề này chi tiết hơn trong §7.1.

5.1.5 Các mức tối ưu hóa gần nhau cho kết quả tương tự nhau

* Chúng tôi cũng đo lường tác động của tối ưu hóa kích thước (Os) bằng cách phù hợp với hàm λ trong bộ dữ liệu "NORMAL" với một hàm (λ TP và λ TN) trong bộ dữ liệu "SIZEOPT". Sau đó, các nhị phân được biên dịch với tùy chọn Os gần giống như các nhị phân được biên dịch với các tùy chọn O1 và O2. Điều này không ngạc nhiên bởi vì Os cho phép hầu hết các kỹ thuật O2 trong cả GCC và Clang [125], [126]. Hơn nữa, chúng tôi quan sát rằng các tùy chọn O1 và O2 tạo ra các nhị phân tương tự nhau, mặc dù điều này không được hiển thị trong Bảng 6 do hạn chế không gian.

5.1.6 Các tùy chọn bổ sung có tác động ít hơn

* Để đánh giá tác động của các tùy chọn PIE, no-inline và LTO, chúng tôi so sánh các hàm trong bộ dữ liệu "NORMAL" với các hàm trong các bộ dữ liệu "PIE", "NOINLINE" và "LTO" tương ứng. Đối với thử nghiệm "no-inline", chúng tôi giới hạn mức tối ưu hóa từ O1 đến O3 vì việc gộp hàm được áp dụng từ O1. Kết quả cho thấy tác động của các tùy chọn bổ sung này không đáng kể. Nhị phân với và không có tùy chọn PIE tương tự nhau vì nó chỉ thay đổi các hướng dẫn để sử dụng địa chỉ tương đối; do đó, nó không ảnh hưởng đến các đặc trưng trước nghĩa của chúng tôi. Gộp hàm cũng không ảnh hưởng đến một số đặc trưng, chẳng hạn như số lần gọi đến đến hàm, dẫn đến AUC cao (0.97). LTO cũng không có tác động đáng kể.
* Tuy nhiên, bằng cách phân tích từng trường hợp thử nghiệm, chúng tôi nhận thấy rằng một số tùy chọn ảnh hưởng đến AUC hơn những tùy chọn khác. Ví dụ, trong thử nghiệm "no-inline", AUC giảm mạnh khi mức tối ưu hóa tăng lên: O1 (0.995), O2 (0.981) và O3 (0.967). Điều này bởi vì khi áp dụng nhiều kỹ thuật tối ưu hóa hơn, có nhiều hàm được gộp và biến đổi trong "NORMAL", trong khi các hàm tương ứng trong "NOINLINE" không được gộp. Trong khi đó, trong thử nghiệm "LTO", AUC tăng khi phiên bản Clang tăng lên: v4 (0.956), v5 (0.968), v6 (0.986) và v7 (0.986). Ngược lại, GCC cho thấy AUC ổn định (0.987-0.988) qua tất cả các phiên bản và tất cả các AUC đều cao hơn so với Clang. Kết quả này cho thấy rằng thay đổi nhiều tùy chọn có thể ảnh hưởng đáng kể đến tỷ lệ thành công, chúng tôi sẽ mô tả chi tiết phía dưới.

5.1.7 Obfuscator-LLVM không ảnh hưởng đến các đặc trưng CG (Control-Flow Graph)

* Nhiều nghiên cứu trước đây [20], [31], [61], [67], [70] đã chọn Obfuscator-LLVM [105] để thực hiện các thử nghiệm làm rối mã nhị phân vì nó tạo ra sự biến đổi đáng kể trong mã nhị phân [20]. Tuy nhiên, áp dụng tất cả ba tùy chọn làm rối của nó chỉ cho thấy AUC là 0.95 trên bộ dữ liệu của chúng tôi, tức là tỷ lệ thành công tương đối cao hơn so với các thử nghiệm về mức tối ưu hóa. Làm rối mã nhị phân gây giảm mạnh trong khoảng cách trung bình giữa TP-TN (trung bình của khoảng cách giữa hàm TP và hàm TN) ngoại trừ các đặc trưng của CG (Control-Flow Graph).
* Điều này xảy ra bởi vì Obfuscator-LLVM chỉ áp dụng làm rối trong phạm vi nội-procedural, tức là trong một hàm cụ thể. Đối với các tùy chọn làm rối SUB, nó thay thế các hướng dẫn số học trong khi giữ nguyên ý nghĩa; với tùy chọn làm rối BCF, nó tác động đáng kể đến các đặc trưng của CG bằng cách thêm các luồng điều khiển giả mạo; tùy chọn làm rối FLA thay đổi các bộ định nghĩa của cấu trúc điều khiển [127]. Tuy nhiên, không một tùy chọn nào trong số này thực hiện làm rối giữa các procedural, tức là làm thay đổi mối quan hệ gọi hàm giữa các hàm. Do đó, chúng tôi khuyến khích các nghiên cứu trong tương lai nên sử dụng các công cụ làm rối khác, chẳng hạn như Themida [128] hoặc VMProtect [107], để đánh giá kỹ thuật của họ đối với làm rối giữa các procedural.

5.1.8 Tùy chọn so sánh đối tượng có ảnh hưởng

* Dựa trên các kết quả thực nghiệm cho đến nay, chúng tôi thực hiện các thử nghiệm bổ sung để hiểu rõ hơn ảnh hưởng của việc so sánh nhiều tùy chọn biên dịch bằng cách chọn λTP và λTN từ các mã nhị phân có thể cung cấp khoảng cách TP-TN thấp nhất. Trong nghiên cứu này, chúng tôi chỉ trình bày hai trong số chúng do giới hạn không gian. Cụ thể, cho thử nghiệm đầu tiên, chúng tôi đã chọn các hàm từ các mã nhị phân ARM 32-bit biên dịch bằng GCC v4 với tùy chọn O0 và các hàm λTP và λTN tương ứng từ các mã nhị phân MIPS 64-bit endian lớn biên dịch bằng Clang v7 với tùy chọn O3. Đối với thử nghiệm thứ hai, chúng tôi đã thay đổi trình biên dịch Clang thành Obfuscator-LLVM với tất cả ba tùy chọn làm rối được bật. Cột Bad trong bảng tổng hợp các kết quả. AUC trong cả hai trường hợp đều là khoảng 0.93 và 0.91, tương ứng. Giá trị trung bình của khoảng cách TP-TN cũng thấp đáng kể (0.42 và 0.27) so với các thử nghiệm khác. Điều này chứng tỏ tầm quan trọng của việc lựa chọn các mục tiêu đánh giá để đánh giá các kỹ thuật BCSA.
* Nghiên cứu BCSA hiện tại so sánh các hàm cho tất cả các mục tiêu có thể trong một bộ dữ liệu, như đã thể hiện trong các thử nghiệm Rand. trong nghiên cứu này. Tuy nhiên, kết quả của chúng tôi cho thấy các nhà nghiên cứu nên lựa chọn mục tiêu đánh giá cẩn thận để tránh bỏ qua ảnh hưởng của các trường hợp xấu.

5.2 So sánh với các kỹ thuật tiên tiến hiện tại

* Từ các thử nghiệm trong phần 5.1, chúng tôi đã chỉ ra rằng sử dụng chỉ các tính năng tiền nghĩa với một mô hình tuyến tính đơn giản (tức là TIKNIB) là đủ để đạt được các giá trị AUC cao. Tiếp theo, chúng tôi so sánh TIKNIB với các kỹ thuật tiên tiến hiện tại.
* Để thực hiện điều này, chúng tôi chọn một trong những phương pháp mới nhất, là VulSeeker [13], làm mục tiêu vì nó sử dụng cả tính năng tiền nghĩa và tính năng ngữ nghĩa dưới dạng số học bằng cách sử dụng xử lý sau mạng neural. Do đó, chúng tôi có thể đánh giá trực tiếp mô hình đơn giản của chúng tôi bằng các tính năng tiền nghĩa số học. Lưu ý rằng mục tiêu của chúng tôi không phải là tuyên bố rằng phương pháp của chúng tôi tốt hơn, mà là để chứng minh rằng kỹ thuật hợp lý của tính năng tiền nghĩa có thể đạt được kết quả có thể so sánh với các kỹ thuật tiên tiến hiện tại.

A data sheet with a diagram

Description automatically generated

* Cho thí nghiệm này, chúng tôi đã chuẩn bị các bộ dữ liệu của VulSeeker, cùng với các bộ dữ liệu bổ sung được liệt kê trong Bảng 7. Chúng tôi gọi những bộ dữ liệu này là ASE1 đến ASE4. ASE1 và ASE3 là những bộ dữ liệu được sử dụng trong VulSeeker, và ASE2 và ASE4 là những bộ dữ liệu bổ sung với nhiều gói phần mềm hơn, kiến trúc mục tiêu và trình biên dịch. Lưu ý rằng số lượng gói phần mềm, kiến trúc và tùy chọn trình biên dịch tăng khi chỉ số của bộ dữ liệu tăng lên. Các mức tối ưu hóa cho tất cả các bộ dữ liệu là O0-O3. Chúng tôi cố ý bỏ qua hình ảnh firmware được sử dụng trong bài báo gốc, vì chúng không cung cấp dữ liệu thực tế vững chắc. Đối với mỗi bộ dữ liệu, chúng tôi xác định dữ liệu thực tế theo cùng cách mô tả trong §3.2. Thời gian tiêu tốn cho việc tiền xử lý IDA, xây dựng dữ liệu thực tế và trích xuất tính năng lần lượt là 2197 giây, 889 giây và 239 giây. Sau đó, chúng tôi tiến hành thí nghiệm theo phương pháp được giải thích trong §4; lưu ý rằng cùng phương pháp đã được sử dụng trong bài báo gốc. Hình 2 miêu tả kết quả. Hình 2a cho thấy rằng các AUC của TIKNIB trên ASE1 và ASE3 lần lượt là 0.9724 và 0.9783. Tuy nhiên, các AUC của VulSeeker là 0.99 và 0.8849 như đã báo cáo bởi các tác giả. Hình 2b minh họa rằng AUC của mỗi lượt trong ASE3 nằm trong khoảng từ 0.9777 đến 0.9793, cao hơn so với AUC của VulSeeker (0.8849). Do đó, TIKNIB ổn định hơn VulSeeker về kích thước và tùy chọn biên dịch trong bộ dữ liệu. TIKNIB cũng cho thấy kết quả ổn định, ngay cả với ASE2 và ASE4.
* Từ những kết quả này, chúng tôi kết luận rằng các đặc trưng tiền tương thích kết hợp với kỹ thuật kỹ thuật đặc trưng hợp lý có thể đạt được kết quả tương đương với các kỹ thuật BCSA hiện đại nhất. Mặc dù trọng tâm hiện tại của chúng tôi là so sánh các giá trị đặc trưng, nhưng có thể mở rộng công việc của chúng tôi để phân tích các mối quan hệ phức tạp giữa các đặc trưng bằng cách sử dụng các kỹ thuật máy học tiên tiến [12], [13], [19], [20], [21], [23], [24], [25], [26], [27], [28], [29].

A screenshot of a computer

Description automatically generated

5.3 Nghiên cứu trường hợp: Heartbleed (CVE-2014-0160)

* Để đánh giá hiệu quả của các đặc trưng tiền tương thích, chúng tôi áp dụng TIKNIB vào việc phát hiện lỗ hổng, một ứng dụng thực tế phổ biến của BCSA [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. Chúng tôi xem xét khả năng của TIKNIB trong việc xác định chính xác một hàm có lỗ hổng trên các tùy chọn trình biên dịch và kiến trúc khác nhau.
* Chúng tôi đã chọn hàm tls1\_process\_heartbeat trong gói OpenSSL làm mục tiêu vì nó chứa lỗ hổng Heartbleed nổi tiếng (CVE-2014-0160), đã được sử dụng rộng rãi trong các nghiên cứu trước đây để đánh giá [11], [12], [20], [23]. Chúng tôi sử dụng hai phiên bản của OpenSSL trong tập dữ liệu ASE4 như đã hiển thị trong Bảng 7: phiên bản v1.0.1f chứa hàm có lỗ hổng, trong khi phiên bản v1.0.1u chứa phiên bản đã được vá lỗi. Vì tập dữ liệu được biên dịch với 288 sự kết hợp khác nhau của tùy chọn trình biên dịch và kiến trúc, mỗi hàm có 576 mẫu: 288 (số kết hợp có thể) × 2 (số phiên bản OpenSSL có sẵn) ≈ 576.
* Đáng chú ý, việc kiểm tra tất cả các kết hợp tùy chọn có thể gây một lượng tính toán đáng kể; nó yêu cầu 288 (số lựa chọn cho hàm mục tiêu của chúng tôi) × 287 (số lựa chọn cho một hàm trong OpenSSL) × 2 (số phiên bản OpenSSL) × 5K (số lượng hàm trong OpenSSL) ≈ 826 triệu phép tính. Do đó, chúng tôi tập trung vào các kiến trúc và tùy chọn trình biên dịch phổ biến được sử dụng trong các gói phần mềm. Cụ thể, chúng tôi chọn ba kiến trúc 64-bit (aarch64, x86-64 và mips64el) và hai cấp độ tối ưu hóa (O2-O3). Cấu hình này phản ánh các kịch bản thực tế, vì nhiều gói phần mềm sử dụng O2-O3 theo mặc định: coreutils sử dụng O2, trong khi OpenSSL sử dụng O3. Các nghiên cứu trước đó [20], [59] cũng đã sử dụng cấu hình tương tự (O2-O3) chỉ trừ rằng họ chỉ kiểm tra các tệp nhị phân x86. Ngoài ra, chúng tôi đã chọn bốn trình biên dịch (Clang v4.0, Clang v7.0, GCC v4.9.4 và GCC v8.2.0) để xem xét các trường hợp cực đoan. Do đó, có tổng cộng 24 kết hợp tùy chọn của các kiến trúc này và trình biên dịch.
* Chúng tôi đã thực hiện tổng cộng 552 thử nghiệm trên 24 kết hợp tùy chọn này: 24 (số lựa chọn cho hàm mục tiêu của chúng tôi) × 23 (số lựa chọn cho một hàm trong OpenSSL). Đối với mỗi thử nghiệm, chúng tôi đơn giản tính toán các điểm tương đồng cho tất cả các cặp hàm bằng TIKNIB và kiểm tra thứ hạng của hàm có lỗ hổng. Để phản ánh các kịch bản thực tế, chúng tôi giả định trong tất cả các thử nghiệm rằng chúng tôi không biết chính xác cấp độ tối ưu hóa, loại trình biên dịch hoặc phiên bản trình biên dịch của tệp nhị phân được kiểm tra. Trong khi đó, chúng tôi giả định rằng chúng tôi có thể nhận ra kiến trúc của tệp nhị phân được kiểm tra vì điều này rất dễ dàng. Do đó, khi chúng tôi đào tạo TIKNIB, chúng tôi chọn một bộ đặc trưng đã đạt được hiệu suất tốt nhất qua tất cả các kết hợp có thể của các cấp độ tối ưu hóa, các loại trình biên dịch và phiên bản trình biên dịch, trong khi giữ cố định kiến trúc nguồn và đích. Đối với việc đào tạo, chúng tôi sử dụng tập dữ liệu NORMAL (Bảng 3) vì nó không bao gồm OpenSSL; do đó, tập dữ liệu đào tạo và kiểm tra là hoàn toàn khác nhau.
* Bảng 8 tổng kết kết quả, trong đó mỗi cột tương ứng với các thử nghiệm cho các tùy chọn được chỉ định. Chúng tôi tổ chức kết quả theo nhóm tùy chọn được chỉ định trong mỗi cột sau khi thực hiện tất cả 552 thử nghiệm. Hàng đầu tiên của bảng (# of Option Pairs) chỉ ra tổng số cặp tùy chọn, tương đương với số cặp tích cực thực tế. Các hàng còn lại của bảng hiển thị các giá trị trung bình được thu được từ các thử nghiệm cặp tùy chọn. Ví dụ, cột All to All biểu thị các kết quả trung bình của tất cả các kết hợp có thể (24 × 23). Cột ARM to MIPS, ngược lại, đại diện cho các kết quả trung bình của tất cả các kết hợp với kiến trúc nguồn và đích được đặt là ARM và MIPS, tương ứng. Điều này có nghĩa là chúng tôi truy vấn các hàm có lỗ hổng được biên dịch với kiến trúc ARM và tìm kiếm các hàm tích cực thực tế được biên dịch với kiến trúc MIPS trong khi biến đổi các tùy chọn khác.
* Trong hầu hết các thử nghiệm, TIKNIB đã thành công trong việc xác định hàm có lỗ hổng với một hạng gần 1.0 và độ chính xác tại vị trí đầu tiên gần 1.0, chứng tỏ tính hiệu quả của nó trong việc phát hiện lỗ hổng. Tuy nhiên, nó đã thể hiện kết quả ít tốt hơn trong các thử nghiệm cho kiến trúc MIPS. Kết quả này xác nhận nhận định của chúng tôi ở phần 5.1 rằng việc trích xuất đặc trưng cho các tập tin nhị phân MIPS có thể không chính xác. Chúng tôi sẽ tiếp tục thảo luận về vấn đề này trong phần 7.1. Ngoài ra, ba hàng cuối cùng của Bảng 8 hiển thị thứ hạng của các hàm bổ sung đáng chú ý. Cột "dtls" biểu thị sự triển khai DTLS của hàm mục tiêu của chúng tôi (tức là dtls1\_process\_heartbeat), cũng chứa cùng một lỗ hổng. Do tính tương đồng với hàm mục tiêu của chúng tôi, nó đã xếp hạng cao trong tất cả các thử nghiệm. Hai hàng cuối cùng của bảng hiển thị thứ hạng của các phiên bản đã được vá lỗi của hai hàm này trong OpenSSL v1.0.1u. Đáng chú ý, việc vá lỗi ảnh hưởng đến các đặc trưng tiền ngữ của các hàm này, đặc biệt là số lượng lệnh chuyển đổi điều khiển và lệnh số học. Do đó, các hàm đã được vá lỗi có thứ hạng thấp.

5.4 Analyzing Real-World Vulnerabilities on Firmware Images of IoT Devices:

* Để đánh giá hiệu quả của các đặc trưng tiền ngữ, chúng tôi tiến hành xác định các hàm có lỗ hổng trên các hình ảnh firmware thực tế của các thiết bị IoT bằng cách sử dụng TIKNIB. Chúng tôi sử dụng bộ dữ liệu firmware của FirmAE [129], đó là một trong những khung nhúng firmware hàng đầu trong ngành. Bộ dữ liệu này bao gồm 1.124 hình ảnh firmware của các bộ định tuyến không dây và camera IP từ tám nhà cung cấp hàng đầu.
* Cụ thể, chúng tôi tìm kiếm một lỗ hổng nổi tiếng khác (CVE-2015-1791) từ OpenSSL, lỗ hổng này có lỗi chạy đồng thời trong hàm ssl3\_get\_new\_session\_ticket(). Lỗ hổng này cũng đã được sử dụng rộng rãi trong các nghiên cứu trước đó [12], [13], [23]. Vì có rất nhiều hàm (≈ 52 triệu) trong các hình ảnh firmware, việc xác định hàm có lỗ hổng trong số chúng là đủ để đánh giá tác động của các đặc trưng tiền ngữ.

A black and white text with numbers and percentages

Description automatically generated

* Đặc biệt, chúng tôi tìm kiếm một lỗ hổng nổi tiếng khác (CVE-2015-1791) từ OpenSSL, lỗ hổng này có lỗi chạy đồng thời trong hàm ssl3\_get\_new\_session\_ticket(). Lỗ hổng này cũng đã được sử dụng rộng rãi trong các nghiên cứu trước đó [12], [13], [23]. Vì có rất nhiều hàm (≈ 52 triệu) trong các hình ảnh firmware, việc xác định hàm có lỗ hổng trong số chúng là đủ để đánh giá tác động của các đặc trưng tiền ngữ.
* Chúng tôi đánh giá hệ thống của mình, TIKNIB, so với các kỹ thuật tiên tiến hỗ trợ cả kiến trúc ARM và MIPS [12], [13]. Đáng chú ý, hai kiến trúc này phổ biến trong các thiết bị IoT [129]. Tuy nhiên, khi phân tích các kho lưu trữ của các công cụ này, chúng tôi nhận thấy họ không bao gồm mã nguồn hoàn chỉnh hoặc bộ dữ liệu. Do đó, chúng tôi không thể so sánh trực tiếp hệ thống của mình với họ. Thay vào đó, chúng tôi so sánh kết quả với những gì được nêu trong bài báo [13]. Cụ thể, chúng tôi biên dịch phiên bản có lỗ hổng của OpenSSL (tức là v1.0.1f) bằng nhiều tùy chọn trình biên dịch và kiến trúc, bao gồm sáu kiến trúc (x86, ARM và MIPS, mỗi kiến trúc có 32 và 64 bit), hai trình biên dịch (GCC v4.9.4 và v5.5.0) và hai cấp độ tối ưu hóa (O2–O3). Ở đây, chúng tôi sử dụng hai cấp độ tối ưu hóa (O2–O3) vì nhiều gói phần mềm thực tế sử dụng chúng theo mặc định, như đã mô tả trong §5.3. Kết quả là chúng tôi nhận được 24 mẫu hàm có lỗ hổng. Đáng chú ý, bộ dữ liệu này thực chất là một phần con của bộ dữ liệu ASE3, được giới thiệu trong Bảng 7. Sau đó, chúng tôi truy vấn từng mẫu hàm có lỗ hổng vào tất cả 52 triệu hàm trong 1.124 hình ảnh firmware. Điều này cho kết quả 24 điểm tương đồng cho mỗi trong 52 triệu hàm. Sau đó, chúng tôi tính toán kết quả top-k bằng cách lấy trung bình điểm tương đồng cho mỗi hàm. Cuối cùng, chúng tôi thủ công đếm số lượng hàm thực sự có lỗ hổng trong top 100 kết quả.
* Bảng 9 tổng kết kết quả top-k cho điểm tương đồng trung bình cho tất cả 52 triệu chức năng firmware. Mặc dù bộ dữ liệu của chúng tôi khác biệt so với các nghiên cứu trước đó [12], [13], TIKNIB được trang bị các đặc trưng tiền ngữ đã đạt được mức hiệu suất tương đương với các công cụ tiên tiến nhất. Cần lưu ý rằng mục tiêu của chúng tôi không phải là khẳng định phương pháp của chúng tôi vượt trội hơn các công cụ tiên tiến, mà là chứng minh tính hiệu quả của việc sử dụng các đặc trưng tiền ngữ một cách thích hợp. Ngoài ra, kết quả thử nghiệm của chúng tôi cho thấy rằng các hình ảnh firmware IoT thực tế (ít nhất là các hình ảnh mà chúng tôi đã thử nghiệm) rất có khả năng được biên dịch với O2 hoặc O3.

6 LỢI ÍCH CỦA THÔNG TIN LOẠI (RQ3)

* Để đánh giá ý nghĩa của thông tin gỡ lỗi đối với BCSA, chúng tôi sử dụng thông tin loại như một nghiên cứu trường hợp với giả định rằng chúng không thay đổi trừ khi mã nguồn bị thay đổi. Cụ thể, chúng tôi trích xuất ba loại đặc trưng cho mỗi chức năng: số lượng đối số, loại đối số và loại trả về của một chức năng. Lưu ý rằng suy ra thông tin loại chính xác là một thách thức và đang được nghiên cứu tích cực [130], [131]. Trong ngữ cảnh này, chúng tôi chỉ xem xét các loại cơ bản: char, short, int, float, enum, struct, void và void \* . Để trích xuất thông tin loại, chúng tôi tạo một bản đồ loại để xử lý các loại tùy chỉnh được định nghĩa trong mỗi gói bằng cách theo định nghĩa đệ quy bằng cách sử dụng Ctags [132]. Sau đó, chúng tôi gán một số nguyên tố duy nhất làm nhận dạng cho mỗi loại. Để biểu thị các loại đối số dưới dạng một số nguyên, chúng tôi nhân các nhận dạng loại của chúng.
* Để nghiên cứu lợi ích của các đặc trưng loại này, chúng tôi đã thực hiện các thử nghiệm giống như được mô tả trong §5, và Bảng 10 trình bày kết quả. Ở đây, chúng tôi giải thích kết quả bằng cách so sánh chúng với Bảng 6, mà chúng tôi đã thu được mà không sử dụng các đặc trưng loại. Dòng đầu tiên của Bảng 10 cho thấy số lượng trung bình các đặc trưng được chọn, bao gồm các đặc trưng loại, nhỏ hơn số lượng các đặc trưng được chọn (5) trong Bảng 6. Lưu ý rằng tất cả ba đặc trưng loại luôn được chọn trong tất cả các thử nghiệm. Dòng thứ hai trong Bảng 10 cho thấy sử dụng các đặc trưng loại có thể đạt được một khoảng lớn giữa TP-TN trung bình (hơn 0.50); các giá trị tương ứng trong Bảng 4 là nhỏ hơn nhiều. Do đó, AUC và AP với các đặc trưng loại đạt hơn 0.99 trong tất cả các thử nghiệm, như được thể hiện trong hai dòng cuối cùng của Bảng 10. Ngoài ra, nó cho thấy một kết quả tương tự (tức là một AUC gần 1.0) trên các tập dữ liệu ASE mà chúng tôi đã sử dụng cho so sánh với công nghệ tiên tiến (§5.2). Kết quả này xác nhận rằng thông tin loại thực sự có lợi cho BCSA về tỷ lệ thành công, mặc dù việc phục hồi thông tin như vậy là một công việc khó khăn. Do đó, chúng tôi khuyến khích các nghiên cứu tiếp theo về BCSA cần xem xét việc phục hồi thông tin gỡ lỗi, chẳng hạn như phục hồi hoặc suy diễn thông tin loại từ mã nhị phân [130], [131], [133], [134], [135], [136].

7. Suy luận về các trường hợp thất bại (RQ4)

* Chúng tôi đã cẩn thận phân tích các trường hợp thất bại trong các thử nghiệm của chúng tôi và tìm ra nguyên nhân của chúng. Điều này có thể thực hiện được vì bài thử nghiệm của chúng tôi (tức là BINKIT) có đối chiếu và công cụ của chúng tôi (tức là TIKNIB) sử dụng một mô hình có thể giải thích được. Chúng tôi đã kiểm tra khoảng cách TP-TN của mỗi đặc trưng cho các trường hợp thất bại và tiến hành phân tích chi tiết bằng cách sử dụng IDA Pro. Chúng tôi phát hiện rằng việc tối ưu hóa ảnh hưởng lớn đến hiệu suất BCSA, như đã mô tả trong §5.1. Trong phần này, chúng tôi thảo luận về các nguyên nhân thất bại khác và tổng kết những bài học đã học được; tuy nhiên, nhiều trong số các nguyên nhân này liên quan chặt chẽ đến tối ưu hóa. Chúng tôi phân loại các nguyên nhân thành ba trường hợp: (1) lỗi trong các công cụ phân tích nhị phân (§7.1), (2) sự khác biệt trong các bộ biên dịch (§7.2) và (3) mã cụ thể cho kiến trúc (§7.3).

A close-up of a chart

Description automatically generated

7.1 Lỗi trong các công cụ phân tích nhị phân

* Hầu hết các nghiên cứu BCSA dựa nhiều vào các công cụ phân tích nhị phân thương mại có sẵn (COTS) như IDA Pro [95]. Tuy nhiên, chúng tôi đã phát hiện rằng IDA Pro có thể đưa ra kết quả sai. Đầu tiên, IDA Pro không thể phân tích các nhánh gián tiếp, đặc biệt là khi xử lý các tập tin nhị phân MIPS được biên dịch bằng Clang sử dụng tùy chọn mã code vị trí độc lập (PIC). Tùy chọn PIC đặt trình biên dịch để tạo mã máy có thể được đặt ở bất kỳ địa chỉ nào và chủ yếu được sử dụng để biên dịch các thư viện chia sẻ hoặc các tập tin nhị phân PIE (Position Independent Executables). Cụ thể, các trình biên dịch sử dụng các hướng nhánh gián tiếp thông qua các lệnh, chẳng hạn như jalr, để gọi các hàm theo cách không phụ thuộc vào vị trí. Ví dụ, khi gọi một hàm, GCC lưu trữ địa chỉ cơ sở của Bảng Offset Toàn cục (GOT) trong thanh ghi gp và sử dụng nó để tính toán các địa chỉ hàm tại thời gian chạy. Trái lại, Clang sử dụng thanh ghi s0 hoặc v0 để lưu trữ các địa chỉ cơ sở tương tự. Sự khác biệt tinh tế này gây nhầm lẫn cho IDA Pro và khiến nó không thể lấy được địa chỉ cơ sở của GOT, do đó không thể tính toán các địa chỉ mục tiêu của các nhánh gián tiếp.
* Hơn nữa, IDA Pro đôi khi tạo ra các Control Flow Graph (CFG - Đồ thị dòng điều khiển) không đầy đủ. Khi có một câu lệnh switch, các trình biên dịch thường tạo một bảng lưu trữ danh sách các địa chỉ mục tiêu của các lệnh nhảy. Tuy nhiên, IDA Pro thường không thể xác định chính xác số lượng phần tử trong bảng này, đặc biệt là trên kiến trúc ARM, nơi các bảng switch có thể được đặt trong một đoạn mã. Đôi khi, các bảng switch được đặt giữa các khối cơ bản, và điều này khó phân biệt hơn.
* Vấn đề trở nên tồi tệ hơn khi xử lý các tập tin nhị phân MIPS được biên dịch cho Clang với PIC, vì các bảng switch thường được lưu trữ trong một phần dữ liệu chỉ đọc, có thể được tham chiếu thông qua GOT. Do đó, nếu IDA Pro không thể phân tích đầy đủ địa chỉ cơ sở của GOT, nó cũng không thể xác định các mục tiêu nhảy của các câu lệnh switch.
* Khi chúng tôi phân tích lỗi một cách thủ công, có thể chúng tôi đã bỏ sót một số lỗi. Tìm kiếm các lỗi một cách hệ thống là một công việc khó khăn vì nội tại của nhiều công cụ phân giải mã không được tiết lộ đầy đủ và chúng có sự khác biệt đáng kể. Có thể mở rộng nghiên cứu trước đó [96] để tiếp tục phân tích các lỗi của các công cụ phân giải mã và các đặc trưng đã trích xuất, và chúng tôi để lại điều này cho các nghiên cứu trong tương lai.
* Trong quá trình phân tích, chúng tôi cũng phát hiện rằng IDA Pro cũng thất bại trong việc truy xuất một số tên hàm nếu chúng có tiền tố được định nghĩa trước trong IDA Pro, chẳng hạn như "off\_" hoặc "sub\_". Ví dụ, nó không thể truy xuất tên của hàm "off\_to\_chars" trong gói tar. Chúng tôi đã sử dụng phiên bản IDA Pro v6.95 trong các thí nghiệm của chúng tôi, nhưng chúng tôi đã phát hiện rằng phiên bản mới nhất của nó (v7.5) không gặp vấn đề này.

7.2 Đa dạng của các trình biên dịch

* Từ §5.1, đặc điểm của các tệp nhị phân thay đổi lớn tùy thuộc vào trình biên dịch nền tảng. Nghiên cứu của chúng tôi đã tiết lộ rằng GCC và Clang tạo ra các tệp nhị phân khác nhau đáng kể từ cùng mã nguồn.
* Trước tiên, số khối cơ bản của hai trình biên dịch khác biệt đáng kể. Để quan sát cách số khối thay đổi tùy thuộc vào các tùy chọn biên dịch và kiến trúc mục tiêu khác nhau, chúng tôi đã đếm số lượng trong bộ dữ liệu bình thường (NORMAL dataset). Hình 3 mô tả số lượng hàm và khối cơ bản trong bộ dữ liệu cho các tùy chọn biên dịch và kiến trúc được chọn (xem phụ lục để biết thêm chi tiết). Như được thể hiện trong hình vẽ, số lượng khối cơ bản trong các tệp nhị phân được biên dịch bằng Clang lớn hơn đáng kể so với các tệp nhị phân được biên dịch bằng GCC ở O0. Chúng tôi đã phát hiện ra rằng Clang chèn các khối cơ bản giả mạo cho O0 trên các kiến trúc ARM và MIPS; các khối giả mạo này chỉ có một chỉ thị nhảy đến khối tiếp theo. Những khối giả mạo này sẽ bị loại bỏ khi cấp độ tối ưu hóa tăng lên (O1) vì các kỹ thuật tối ưu hóa trong Clang sẽ gộp những khối cơ bản này vào khối cơ bản tiền nhiệm của chúng.
* Hơn nữa, hai trình biên dịch áp dụng các kỹ thuật nội bộ khác nhau cho cùng một cấp độ tối ưu hóa, trong khi họ cùng diễn tả cấp độ tối ưu hóa bằng các thuật ngữ giống nhau (tức là O0–O3 và Os). Cụ thể, bằng cách phân tích số lượng hàm gọi và hàm được gọi, chúng tôi đã phát hiện ra rằng GCC áp dụng việc nhúng hàm từ O1, trong khi Clang áp dụng nó từ O2. Do đó, số lượng hàm cho mỗi trình biên dịch khác biệt đáng kể (xem số lượng hàm trong O1 cho Clang và số lượng hàm trong O1 cho GCC trong Hình 3).

A graph of different colored lines

Description automatically generated

* Hơn nữa, chúng tôi đã phát hiện ra rằng hai trình biên dịch sử dụng mã cấp độ hàm khác nhau cho các thao tác cụ thể. Ví dụ, GCC có các hàm như \_\_umoddi3 trong libgcc2.c hoặc \_\_aeabi\_dadd trong ieee754-df.S để tối ưu hóa một số thao tác số học cụ thể. Hơn nữa, trên kiến trúc x86, GCC tạo một hàm đặc biệt, chẳng hạn như \_\_x86.get\_pc\_thunk.bx, để tải con trỏ chỉ thị hiện tại vào một thanh ghi, trong khi Clang nhúng thủ tục này vào bên trong hàm mục tiêu. Những hàm này có thể ảnh hưởng lớn đến các đặc trưng liên quan đến cuộc gọi, chẳng hạn như số lượng lệnh điều khiển hoặc số lượng cuộc gọi đi ra. Mặc dù chúng tôi đã loại bỏ những hàm cụ thể cho trình biên dịch này để không bao gồm chúng trong các thí nghiệm của mình (§3.2), chúng có thể đã được nhúng vào các hàm gọi của chúng ở các cấp độ tối ưu hóa cao hơn (O2-O3). Với xấp xỉ 4% số lượng hàm được xác định bởi IDA Pro, chúng có thể đã ảnh hưởng đến các đặc trưng kết quả.
* Tương tự, hai trình biên dịch cũng sử dụng các mã cấp độ lệnh khác nhau. Ví dụ, trong trường hợp lệnh di chuyển cho kiến trúc ARM, GCC sử dụng lệnh điều kiện, chẳng hạn như MOVLE, MOVGT, hoặc MOVNE, trừ khi cấp độ tối ưu hóa là không (O0). Ngược lại, Clang sử dụng các lệnh di chuyển thông thường cùng với các lệnh nhảy. Điều này ảnh hưởng đáng kể đến số lượng lệnh cũng như số lượng khối cơ bản trong các tệp nhị phân kết quả. Do đó, trong những trường hợp đặc biệt như vậy, các hàm được biên dịch bằng GCC có số lượng khối cơ bản tương đối nhỏ hơn so với những hàm được biên dịch bằng Clang.
* Cuối cùng, trình biên dịch đôi khi tạo ra nhiều bản sao của cùng một hàm cho mục đích tối ưu hóa. Ví dụ, chúng thực hiện thay thế skalar liên kết giữa các thành phần, loại bỏ các tham số không sử dụng hoặc tối ưu hóa việc sử dụng bộ nhớ/bộ nhớ cache. Kết quả là, một tệp nhị phân được biên dịch có thể có nhiều hàm chia sẻ cùng mã nguồn nhưng có mã nhị phân khác nhau. Chúng tôi phát hiện ra rằng GCC và Clang hoạt động khác nhau trong việc này. Cụ thể, chúng tôi phát hiện ba kỹ thuật trong GCC tạo ra các bản sao của hàm với hậu tố đặc biệt, chẳng hạn như .part, .cold hoặc .isra. Ví dụ, đối với hàm get\_data của readelf trong binutils (ở O3), GCC cho ra ba bản sao với hậu tố .isra, trong khi Clang không tạo ra bất kỳ hàm nào như vậy. Tương tự, đối với hàm tree\_eval và expr\_eval trong bool (ở O3), GCC tạo ra hai bản sao với hậu tố .cold, nhưng Clang không tạo ra. Mặc dù chúng tôi chỉ chọn một bản sao như vậy trong các thí nghiệm của mình để tránh kết quả thiên vị (§3.2), những bản sao khác vẫn có thể tồn tại trong các hàm gọi của chúng bằng cách nhúng mã.
* Tóm lại, sự đa dạng của các trình biên dịch có thể ảnh hưởng đáng kể đến hiệu suất của BCSA, khi làm cho các tệp nhị phân kết quả khác nhau. Ở đây, chúng tôi đã giới thiệu các vấn đề chính mà chúng tôi đã phát hiện ra. Chúng tôi khuyến khích các nghiên cứu tiếp tục điều tra tác động của các tùy chọn chi tiết tại mỗi cấp độ tối ưu hóa trên các trình biên dịch khác nhau.

7.3 Mã nguồn Dựa trên Kiến trúc cụ thể

* Khi kiểm tra kỹ lưỡng các trường hợp thất bại, chúng tôi phát hiện rằng một số gói có các đoạn mã nguồn cụ thể cho từng kiến trúc, được bảo vệ bằng các macro điều kiện như #if và #ifdef. Ví dụ, các hàm khác nhau trong OpenSSL, chẳng hạn như mul\_add và BN\_UMULT\_HIGH, được viết dưới dạng mã lệnh nội tuyến cụ thể cho từng kiến trúc để tạo ra các tệp nhị phân được tối ưu hóa cao. Điều này có nghĩa là một hàm có thể tương ứng với hai hoặc nhiều dòng mã nguồn khác nhau tùy thuộc vào kiến trúc đích.
* Do đó, các tính năng tiền xử lý cấp lệnh có thể khác nhau đáng kể giữa các kiến trúc khác nhau khi các chương trình đích có các đoạn mã nguồn cụ thể cho từng kiến trúc, và một người nên xem xét mã này khi thiết kế các kỹ thuật BCSA chéo kiến trúc.

8. THẢO LUẬN

* Nghiên cứu của chúng tôi xác định một số hướng nghiên cứu trong BCSA trong tương lai. Trước tiên, nhiều bài báo BCSA đã tập trung vào xây dựng một mô hình tổng quát có thể đạt được các kết quả ổn định với bất kỳ tùy chọn biên dịch nào. Tuy nhiên, một người có thể đào tạo mô hình nhắm vào một tập hợp cụ thể của các tùy chọn biên dịch, như đã thấy trong thí nghiệm của chúng tôi, để cải thiện kỹ thuật BCSA của họ. Rõ ràng từ kết quả thí nghiệm của chúng tôi, một người có thể dễ dàng tăng tỷ lệ thành công của kỹ thuật của họ bằng cách suy ra các tùy chọn biên dịch được sử dụng để biên dịch các tệp nhị phân đích. Có một phương pháp suy ra như vậy [137], và kết hợp nó với các phương pháp BCSA hiện có là một hướng nghiên cứu đầy hứa hẹn.
* Thứ hai, hiện chỉ có một số ít nghiên cứu về việc sử dụng các kỹ thuật giải mã cho BCSA. Tuy nhiên, nghiên cứu của chúng tôi đã tiết lộ tính quan trọng của các kỹ thuật như vậy, và do đó, mời các nghiên cứu tiếp theo về việc tận dụng chúng cho BCSA. Một người cũng có thể thực hiện một phân tích toàn diện về tác động của các tính năng ngữ nghĩa cùng với các kỹ thuật giải mã. Ngoài ra, chúng tôi đã điều tra các tính năng tiền xử lý cơ bản trong nghiên cứu này. Tuy nhiên, hiệu quả của các tính năng ngữ nghĩa vẫn chưa được nghiên cứu kỹ lưỡng trong lĩnh vực này. Do đó, chúng tôi khuyến khích nghiên cứu tiếp theo để điều tra tính hiệu quả của các tính năng ngữ nghĩa cùng với các tính năng tiền xử lý khác không được bao gồm trong nghiên cứu này. Đặc biệt, việc áp dụng các kỹ thuật NLP sẽ là một nghiên cứu quan trọng khác như trong nhiều nghiên cứu gần đây.
* Phạm vi nghiên cứu của chúng tôi giới hạn trong phân tích cấp độ hàm (§4.1). Tuy nhiên, người ta có thể mở rộng phạm vi để xử lý các tình huống BCSA khác như so sánh các tệp nhị phân [20], [27], [54] hoặc một loạt các hướng dẫn [32], [34], [57]. Hơn nữa, người ta có thể mở rộng phương pháp của chúng tôi cho mục đích đa dạng, như phát hiện lỗ hổng [11], [12], [20], [23], [28], [59], [138], phát hiện mã độc [5], [6], [139], [140], [141], [142], [143], nhận dạng chức năng thư viện [71], [84], [144], [145], [146], [147], nhận dạng vi phạm bản quyền/người tác giả [8], [82], [148], hoặc nhận dạng bản vá [149], [150], [151]. Tuy nhiên, việc mở rộng công việc của chúng tôi cho các tác vụ BCSA khác có thể không phù hợp trực tiếp. Điều này bởi vì nó đòi hỏi kiến thức ngành bổ sung để thiết kế một mô hình phù hợp với mục đích và xem xét cẩn thận các sự đánh đổi. Chúng tôi tin rằng những hiểu biết được báo cáo trong nghiên cứu này có thể hỗ trợ trong quá trình này.
* Lưu ý rằng từ §2, chúng tôi không có ý định tiến hành một cuộc khảo sát hoàn chỉnh về các kỹ thuật hiện có, mà thay vào đó, chúng tôi tập trung vào việc hệ thống hóa các tính năng cơ bản đã được sử dụng trong các nghiên cứu trước đó. Hơn nữa, mục tiêu của chúng tôi là điều tra các câu hỏi nghiên cứu chưa được khám phá trong lĩnh vực này thông qua việc tiến hành một loạt các thí nghiệm nghiêm ngặt. Để có một cuộc khảo sát hoàn chỉnh, chúng tôi khuyến nghị đọc giả tham khảo các cuộc khảo sát gần đây về BCSA [152], [153].
* Cuối cùng, vì chúng tôi tập trung vào việc so sánh các tệp nhị phân mà không có mã nguồn, chúng tôi đã cố ý loại bỏ các kỹ thuật so sánh tương tự yêu cầu mã nguồn. Tuy nhiên, nên lưu ý rằng đã có rất nhiều tài liệu nghiên cứu về việc so sánh hai đoạn mã nguồn [75], [154], [155], [156], [157], [158], [159], [160], [161], [162], hoặc so sánh đoạn mã nguồn với đoạn mã nhị phân [163], [164], [165], [166].

9. Kết luận:

* Trong nghiên cứu này, chúng tôi đã khảo sát các tài liệu nghiên cứu trước về BCSA liên quan đến các tính năng và bài kiểm tra được sử dụng. Chúng tôi phát hiện rằng không có nghiên cứu BCSA trước đó nào sử dụng cùng một bài kiểm tra để đánh giá, và một số trong số họ yêu cầu tự tạo ra kết quả thực tế cho bài kiểm tra của họ. Nhìn nhận điều này, chúng tôi đã thiết kế BINKIT, bài kiểm tra lớn đầu tiên về BCSA công khai, cùng với một bộ các tập lệnh xây dựng tự động. Bên cạnh đó, chúng tôi đã phát triển công cụ BCSA, TIKNIB, sử dụng một mô hình có thể giải thích được. Sử dụng bài kiểm tra và công cụ của chúng tôi, chúng tôi đã trả lời những câu hỏi nghiên cứu ít được khám phá về các tính năng cú pháp và cấu trúc của BCSA. Chúng tôi đã phát hiện rằng một số tính năng cơ bản có thể đáng tin cậy trên các kiến trúc, loại trình biên dịch, phiên bản trình biên dịch khác nhau và thậm chí là việc che giấu trong phạm vi nội chức. Hơn nữa, chúng tôi đề xuất các chiến lược tiềm năng để nâng cao BCSA. Chúng tôi kết thúc bằng lời mời các nghiên cứu tiếp theo về BCSA sử dụng các kết quả và bài kiểm tra của chúng tôi.

Lời cảm ơn:

* Chúng tôi trân trọng những nhận xét tận tình từ các nhà phê bình ẩn danh. Công trình này được hỗ trợ bởi Kế hoạch & Đánh giá Công nghệ Thông tin & Truyền thông (IITP) do chính phủ Hàn Quốc tài trợ (MSIT) (Số 2021-0-01332, Phát triển bộ giải mã nhị phân thế hệ kế tiếp).