VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.

VulHawk: Phát hiện lỗ hổng chéo kiến trúc với Tìm kiếm mã nhị phân dựa trên entropy.

* Tóm tắt—Việc tái sử dụng mã nguồn là phổ biến trong phát triển phần mềm. Tuy nhiên, nó mang đến sự lan truyền mạnh mẽ của lỗ hổng, đe dọa an ninh phần mềm. Đáng tiếc, với sự phát triển và triển khai của Internet of Things (IoT), hậu quả của việc tái sử dụng mã nguồn được khuếch đại. Tìm kiếm mã nhị phân là một cách khả thi để tìm ra những lỗ hổng ẩn này. Đối diện với các hình ảnh firmware IoT được biên dịch bởi các trình biên dịch khác nhau với các mức tối ưu hóa khác nhau từ các kiến trúc khác nhau, các phương pháp hiện có khó phù hợp với các tình huống phức tạp này. Trong bài báo này, chúng tôi đề xuất một mô hình hàm biểu diễn trung gian mới, đó là một mô hình không phụ thuộc kiến trúc cho tìm kiếm mã nhị phân chéo kiến trúc. Chúng tôi nâng mã nhị phân thành mã micro và bảo toàn các ý nghĩa chính của các hàm nhị phân thông qua việc bổ sung các toán hạng ngầm định và cắt giảm các chỉ thị dư thừa. Sau đó, chúng tôi sử dụng các kỹ thuật xử lý ngôn ngữ tự nhiên và mạng ngữ cảnh đồ thị để tạo ra các nhúng hàm. Chúng tôi gọi sự kết hợp giữa trình biên dịch, kiến trúc và mức tối ưu hóa là môi trường tập tin, và áp dụng chiến lược chia để trị để chia một vấn đề tính đa dạng giữa C2N các tình huống môi trường tập tin thành N-1 bài toán chuyển nhúng. Chúng tôi đề xuất một bộ chuyển đổi dựa trên entropy để chuyển các nhúng hàm từ các môi trường tập tin khác nhau vào cùng một môi trường tập tin để giảm sự khác biệt gây ra bởi các môi trường tập tin khác nhau. Để xác định chính xác các hàm có lỗ hổng, chúng tôi đề xuất một chiến lược tìm kiếm tiến triển để bổ sung các đặc trưng chi tiết của hàm để giảm thiểu các kết quả sai do các hàm được vá lỗi. Chúng tôi triển khai một nguyên mẫu có tên là VulHawk và tiến hành các thí nghiệm trong bảy tác vụ khác nhau để đánh giá hiệu suất và tính ổn định của nó. Kết quả thí nghiệm cho thấy VulHawk vượt trội hơn Asm2Vec, Asteria, BinDiff, GMN, PalmTree, SAFE và Trex.

I. GIỚI THIỆU

* Việc tái sử dụng mã nguồn là phổ biến trong phát triển phần mềm. Tuy nhiên, một lượng lớn mã và thư viện được tái sử dụng vào nhiều mã nhị phân kiến trúc mà không có kiểm tra bảo mật, dẫn đến nhiều lỗ hổng ẩn trong các dự án phần mềm. Synopsys đã kiểm tra 2.409 dự án vào năm 2021 và báo cáo rằng 97% trong số đó chứa mã nguồn của bên thứ ba và 81% trong số này chứa các lỗ hổng đã biết [49]. Một lỗ hổng đơn lẻ trong mã nguồn mở có thể lan truyền qua hàng ngàn phần mềm, đưa hàng triệu người vào nguy cơ bị đe dọa bởi các mối đe dọa an ninh phần mềm nghiêm trọng.
* Đáng tiếc, với sự phát triển và triển khai của Internet of Things (IoT), hậu quả của việc tái sử dụng mã nguồn được khuếch đại. Thiết bị IoT được sử dụng rộng rãi trong các kịch bản khác nhau. Do các yêu cầu sử dụng khác nhau, các hình ảnh firmware IoT này từ các bộ kiến trúc hướng dẫn (ISAs) khác nhau được tạo ra bởi các trình biên dịch khác nhau với các mức tối ưu hóa khác nhau. Tuy nhiên, nhiều hình ảnh firmware IoT chỉ cung cấp các tệp nhị phân, không có mã nguồn sẵn để phân tích bảo mật. Thông tin về các ký hiệu của chúng, chẳng hạn như tên hàm, thường bị lược bỏ. Do đó, tìm kiếm mã nhị phân trở thành một trọng tâm nghiên cứu tích cực để tìm kiếm các lỗ hổng ẩn trong các thiết bị IoT.
* Tìm kiếm mã nhị phân được áp dụng để tìm các hàm nhị phân tương tự hoặc đồng dạng trong một kho chứa hàm lớn. Nó được sử dụng rộng rãi trong phát hiện lỗ hổng [5]–[7], [16], [29], [41], [43], [48], [58]. Ví dụ, cho một tệp nhị phân, tìm kiếm mã nhị phân so sánh các hàm của nó với tất cả các hàm trong kho chứa lỗ hổng dựa trên sự tương đồng của hàm để tìm các hàm có lỗ hổng trong tệp nhị phân. Ngoài ra, nó cũng được sử dụng trong phân tích phần mềm độc hại [2], [4], [13], [18], [20] và phân tích vá nhị phân [19], [22], [56]. Vì hình ảnh firmware IoT đến từ các trình biên dịch, mức tối ưu hóa và ISAs khác nhau, điều này mang đến những thách thức khó khăn cho tìm kiếm mã nhị phân, đòi hỏi sự ổn định cao cho các phương pháp tìm kiếm.
* P1: Tìm kiếm các lỗ hổng trong firmware IoT đòi hỏi các phương pháp tìm kiếm mã nhị phân mạnh mẽ qua các bộ kiến trúc hướng dẫn (ISAs). Trong tìm kiếm mã nhị phân đơn kiến trúc, các phương pháp Asm2Vec [10], DeepBinDiff [11], và PalmTree [27] sử dụng các kỹ thuật xử lý ngôn ngữ tự nhiên (NLP) đã đạt được kết quả khích lệ. Tuy nhiên, chúng chỉ có thể tìm kiếm mã nhị phân trên cùng một ISA và không hỗ trợ các tác vụ chéo kiến trúc. InnerEye [60] xem xét các mã nhị phân từ các ISA khác nhau như là các ngôn ngữ tự nhiên khác nhau và sử dụng máy dịch nơ-ron để tính toán sự tương đồng mã nhị phân. SAFE [35] huấn luyện mô hình ngôn ngữ của mình bằng cách sử dụng mã nhị phân từ nhiều ISA để tìm kiếm mã nhị phân qua các kiến trúc. Tuy nhiên, những phương pháp này dựa nặng vào dữ liệu huấn luyện và khó triển khai cho nhiều ISA. Việc nâng mã nhị phân cụ thể kiến trúc lên một biểu diễn trung gian (IR) không phụ thuộc kiến trúc là một cách hiệu quả để giải quyết thách thức chéo kiến trúc trong firmware IoT. Tuy nhiên, ngôn ngữ tự nhiên và IR có những khác biệt cơ bản [42]. Khác với ngôn ngữ tự nhiên, IR chứa EFLAGS như là các toán hạng ngầm định (ví dụ: ZF). Các cờ này điều khiển các lộ trình thực thi của hàm và có những ý nghĩa quan trọng cho cú pháp hàm. Ngoài ra, nhiều chỉ thị dư thừa trong IR làm giảm trọng số của cú pháp chính, ảnh hưởng đến độ chính xác trong việc trích xuất cú pháp chính.
* P2: Hình ảnh firmware IoT được biên dịch bởi các trình biên dịch khác nhau với các mức tối ưu hóa khác nhau do yêu cầu sử dụng. Những mã nhị phân từ cùng một mã nguồn biên dịch bởi các trình biên dịch khác nhau với các mức tối ưu hóa khác nhau có cú pháp tương tự nhưng có cấu trúc khác nhau, điều này mang đến những thách thức lớn cho tìm kiếm mã nhị phân. Trong bài báo này, chúng tôi xem xét 3 kiến trúc (x86, arm và mips), 2 kích thước từ (32-bit và 64-bit), 2 trình biên dịch (Clang và GCC) và 6 mức tối ưu hóa (O0, O1, O2, O3, Os và Ofast), tổng cộng 72 kết hợp (3 × 2 × 2 × 6). Nếu các mã nhị phân được chọn từ bất kỳ hai kết hợp trên, tổng số các tình huống kết hợp là 2.556 (C2 72). Các phương pháp hiện có [9], [10], [27], [41] dựa vào học sâu để giảm bớt những sự khác biệt này và xây dựng một mô hình mạnh mẽ chống lại các tình huống này. Có thể xây dựng một mô hình mạnh mẽ cho một hoặc một số tình huống cụ thể. Tuy nhiên, việc xây dựng một mô hình mạnh mẽ chống lại 2.556 tình huống này là phức tạp. Ngoài ra, không có thông tin trực tiếp chỉ ra trình biên dịch và mức tối ưu hóa trong các tệp nhị phân.
* Để giải quyết những vấn đề đã đề cập, trong bài báo này, chúng tôi đề xuất một phương pháp tìm kiếm mã nhị phân chéo kiến trúc mới có tên là VulHawk. Nó bao gồm một mô hình hàm biểu diễn trung gian (IRFM) mới để tạo ra các nhúng hàm mạnh mẽ. Trong IRFM, chúng tôi trước tiên nâng mã nhị phân lên thành mã micro. Sau đó, chúng tôi coi các chuỗi mã micro như là ngôn ngữ và sử dụng một biến thể của mô hình RoBERTa [31] để xây dựng các nhúng khối cơ bản. Chúng tôi sử dụng mạng ngữ cảnh đồ thị (GCNs) để tích hợp các nhúng khối cơ bản và các đồ thị luồng điều khiển (CFGs) để tạo ra các nhúng hàm. Đối với thách thức chéo kiến trúc ở P1, mã micro là một ngôn ngữ không phụ thuộc kiến trúc, cho phép mô hình của chúng tôi được huấn luyện từ một ISA và tìm kiếm các hàm trong nhiều ISA. Đối với các chỉ thị dư thừa và toán hạng ngầm định trong P1, chúng tôi thực hiện một quá trình đơn giản hóa chỉ thị trong IRFM. Chúng tôi xem xét việc gán các toán hạng ngầm định (EFLAGS) là các chỉ thị gán thực sự, giúp IRFM bổ sung ý nghĩa toán hạng ngầm định. Đối với các chỉ thị dư thừa, quá trình đơn giản hóa chỉ thị đơn giản hóa mã micro dựa trên quan hệ def-use, loại bỏ các chỉ thị dư thừa và bảo toàn ý nghĩa chính của các hàm nhị phân. Điều này giúp IRFM trích xuất ý nghĩa hàm một cách chính xác hơn. Chúng tôi cũng đề xuất các nhiệm vụ tiền huấn luyện dự đoán toán hạng gốc (ROP) và dự đoán khối liền kề (ABP) để giúp mô hình hiểu quan hệ giữa các toán hạng và quan hệ dòng dữ liệu giữa các khối cơ bản.
* Đối với các thách thức ở P2, chúng tôi áp dụng một chiến lược chia để trị để chia bài toán tính đa dạng tương tự với 2.556 tình huống thành 71 vấn đề chuyển nhúng. Chúng tôi gọi sự kết hợp giữa trình biên dịch, kiến trúc và mức tối ưu hóa là môi trường tập tin. Đối mặt với 72 môi trường tập tin, chúng tôi chọn một môi trường tập tin trung gian và chuyển các nhúng hàm từ các môi trường tập tin khác nhau vào các môi trường tập tin cùng một để giảm đi sự khác biệt. Trước tiên, chúng tôi giới thiệu entropy của Shannon [47] từ một quan điểm lý thuyết thông tin để biểu diễn lượng thông tin trong các tệp nhị phân. Trong thực tế, chúng tôi thấy rằng các tệp nhị phân từ cùng một môi trường tập tin có phân phối entropy tương tự nhau. Do đó, chúng tôi sử dụng các luồng entropy để xác định môi trường tập tin. Với kiến thức về môi trường tập tin của các hàm, chúng tôi triển khai một bộ chuyển đổi dựa trên entropy để chuyển các nhúng hàm này vào môi trường tập tin trung gian để giảm bớt sự khác biệt do các môi trường khác nhau gây ra. Hơn nữa, chúng tôi đề xuất một chiến lược tìm kiếm tiến triển để tìm kiếm các hàm ứng viên nhằm duy trì hiệu suất và độ chính xác truy xuất cao. Đầu tiên, nó sử dụng các nhúng hàm để truy xuất các hàm ứng viên hàng đầu dựa trên khoảng cách Euclidean. Sau đó, chúng tôi đề xuất phương pháp hiệu chuẩn tương đồng, bổ sung các nhúng hàm bằng các đặc trưng chi tiết để giảm thiểu các kết quả sai.
* Tóm lại, chúng tôi đã đóng góp những điều sau đây:
* Chúng tôi đề xuất một IRFM để tạo ra các nhúng hàm mạnh mẽ qua các kiến trúc. IRFM nâng mã nhị phân lên thành mã micro và bảo toàn cú pháp chính của các hàm nhị phân thông qua việc đơn giản hóa chỉ thị. Hai nhiệm vụ tiền huấn luyện được đề xuất để giúp mô hình của chúng tôi học cú pháp gốc của các toán hạng và nắm vững các quan hệ dòng dữ liệu của các khối cơ bản. Chúng tôi sử dụng mạng ngữ cảnh đồ thị (GCNs) để tích hợp các nhúng khối cơ bản dựa trên các đồ thị luồng điều khiển (CFGs) để tạo ra các nhúng hàm. IRFM có thể hoạt động qua các kiến trúc khác nhau, giải quyết vấn đề P1.
* Theo chiến lược chia để trị, chúng tôi sử dụng luồng entropy để xác định môi trường tập tin của các tệp nhị phân từ quan điểm lý thuyết thông tin. Chúng tôi đề xuất một bộ chuyển đổi dựa trên entropy để chuyển các nhúng hàm vào cùng một môi trường tập tin để giảm đi sự khác biệt do các môi trường tập tin khác nhau gây ra. Điều này làm cho mô hình của chúng tôi mạnh mẽ chống lại các trình biên dịch và mức tối ưu hóa khác nhau, giải quyết vấn đề P2.
* Chúng tôi đề xuất một chiến lược tìm kiếm tiến triển, thực hiện hiệu chuẩn tương đồng bằng cách sử dụng các đặc trưng chi tiết để tăng hiệu suất và giảm số kết quả sai do các hàm đã được vá.
* Chúng tôi triển khai và đánh giá VulHawk trong ba kịch bản khác nhau: so sánh một-một, tìm kiếm một-nhiều và ghép cặp nhiều-nhiều, qua các trình biên dịch, mức tối ưu hóa và kiến trúc. Các thử nghiệm cho thấy VulHawk vượt trội hơn so với các phương pháp tiên tiến.
* Chúng tôi công bố chương trình và mô hình đã được tiền huấn luyện của VulHawk (https://github.com/RazorMegrez/VulHawk) để hỗ trợ nghiên cứu tiếp theo.

II. NỀN TẢNG

A. Định nghĩa vấn đề

* Tìm kiếm mã nhị phân chéo kiến trúc nhằm truy xuất các hàm ứng cử viên tương tự hàng đầu cho hàng loạt các hàm nhị phân được trích xuất từ các thiết bị IoT khác nhau [55]. Được truyền cảm hứng từ các công trình hiện có [10], [55], [57], chúng tôi xác định rằng hai hàm nhị phân tương tự ý nghĩa nếu chúng được biên dịch từ cùng mã nguồn hoặc mã nguồn tương tự logic. Tương tự như phát hiện tương đồng mã nhị phân, trọng tâm của tìm kiếm mã nhị phân là thiết kế một mô hình mạnh mẽ để xác định xem các hàm đã cho có tương tự nhau hay không. Thay vì so khớp một-một, tìm kiếm mã nhị phân xem xét tìm kiếm một-nhiều, yêu cầu các phương pháp truy xuất các ứng cử viên tương tự ngữ nghĩa nhanh chóng và chính xác hơn. Trong thực tế, firmware IoT có thể được biên dịch bởi các trình biên dịch khác nhau (ví dụ: GCC và Clang) với các cấp độ tối ưu hóa khác nhau (ví dụ: O0-O3, Os, và Ofast), dẫn đến các hàm nhị phân biên dịch có cùng ý nghĩa nhưng có cấu trúc khác nhau. Do đó, một tìm kiếm mã nhị phân chéo kiến trúc hiệu quả cần đạt được các mục tiêu sau đây:
* Hỗ trợ chéo kiến trúc. Vì các thiết bị IoT có thể đến từ các kiến trúc khác nhau, nó yêu cầu các phương pháp có tính chống chịu với những sự khác biệt được giới thiệu bởi các kiến trúc khác nhau.
* Hỗ trợ chéo trình biên dịch. Do môi trường phát triển khác nhau, các trình biên dịch và phiên bản trình biên dịch thường khác nhau. Điều này đòi hỏi các phương pháp chịu được những biến thể cú pháp được giới thiệu bởi các phần sau của trình biên dịch khác nhau.
* Hỗ trợ chéo tối ưu hóa. Trong thế giới thực, cùng mã nguồn có thể được biên dịch với các cấp độ tối ưu hóa khác nhau do các yêu cầu khác nhau (ví dụ: nhanh hơn hoặc tinh tế hơn). Điều này đòi hỏi các phương pháp chịu được sự thay đổi cấu trúc luồng điều khiển do các cấp độ tối ưu hóa khác nhau.
* Độ chính xác và hiệu suất cao. Tìm kiếm mã nhị phân nên truy xuất chính xác các hàm tương tự nhất từ hàng triệu hàm và phân biệt các hàm không liên quan. Đối với một cơ sở dữ liệu hàm lớn, nó yêu cầu các phương pháp tìm kiếm mã nhị phân tìm kiếm các hàm tương tự nhanh hơn.

A diagram of a mathematical equation

Description automatically generated

B. Lý thuyết về entropy:

* Trong quan điểm lý thuyết thông tin, entropy của Shannon [47] là một đo lường thông tin cổ điển. Nó đo lường sự ngẫu nhiên và lượng thông tin trung bình trong một hệ thống. Hàm entropy được định nghĩa như sau:
* A black and white math equation

  Description automatically generated
* Trong đó, S là tập các phần tử, và p(x) đại diện cho xác suất của phần tử x. Hình 1 minh họa các ví dụ về việc tính toán entropy của Shannon. Mỗi hình mẫu đại diện cho một phần tử khác nhau. Ví dụ, Hộp 1 được lấp đầy các hình tròn có entropy H bằng 0; Hộp 2 chứa cả hình tròn và hình tam giác, phức tạp hơn Hộp 1, và entropy của nó cao hơn Hộp 1; Hộp 3 là hệ thống phức tạp nhất trong ba, có entropy H cao nhất. Thông qua phân tích entropy, chúng ta có thể hiểu trước lượng thông tin trung bình trong một hệ thống trước khi tiếp tục nghiên cứu chi tiết. Trong nhiệm vụ tìm kiếm mã nhị phân, chúng ta có thể lấy được phân phối thông tin của các tệp nhị phân thông qua entropy của tệp nhị phân, từ đó có thể suy luận thông tin như trình biên dịch và mức tối ưu hóa. Điều này giúp cho mô hình của chúng tôi lựa chọn các tham số phù hợp cho các tệp nhị phân đầu vào khác nhau.

III. THIẾT KẾ

* Trong phần này, chúng tôi mô tả thiết kế của VulHawk. Nó bao gồm ba thành phần: mô hình hàm biểu diễn trung gian, bộ chuyển đổi dựa trên entropy và chiến lược tìm kiếm tiến triển. Hình 2 cho thấy tổng quan về VulHawk.
* Mô hình hàm biểu diễn trung gian (IRFM) được sử dụng để tạo ra nhúng khối cơ bản và nhúng hàm. Chúng tôi đầu tiên nâng mã nhị phân lên thành mã micro. Sau đó, việc đơn giản hóa chỉ thị bổ sung cú pháp toán hạng ngầm và loại bỏ các chỉ thị dư thừa, giúp bảo tồn cú pháp chính của các hàm và cải thiện tính ổn định của IRFM. Tiếp theo, chúng tôi sử dụng mô hình ngôn ngữ dựa trên RoBERTa [31] để xây dựng nhúng khối cơ bản. Trong quá trình huấn luyện mô hình, chúng tôi đề xuất các nhiệm vụ tiền huấn luyện dự đoán cú pháp toán hạng gốc và dự đoán khối cơ bản kế cận để cho phép IRFM hiểu các mối quan hệ giữa các toán hạng và mối quan hệ dòng dữ liệu giữa các khối cơ bản. Cuối cùng, chúng tôi sử dụng mạng ngữ cảnh đồ thị (GCNs) để tổng hợp các nhúng khối cơ bản hàng xóm để nắm bắt các mối quan hệ luồng điều khiển và tạo ra các nhúng hàm.
* Bộ chuyển đổi dựa trên entropy xác định môi trường tập tin của các tệp nhị phân đầu vào và áp dụng chiến lược chia để trị để làm cho các nhúng hàm từ các môi trường tập tin khác nhau giống nhau hơn. Ở đây, chúng tôi giới thiệu entropy từ góc độ lý thuyết thông tin. Đầu tiên, chúng tôi sử dụng entropy để dự đoán các môi trường tập tin, sau đó sử dụng bộ chuyển đổi dựa trên entropy để chuyển các nhúng hàm vào môi trường tập tin trung gian tùy theo môi trường tập tin của chúng để giảm bớt sự khác biệt do các môi trường tập tin gây ra.
* Chiến lược tìm kiếm tiến triển được sử dụng để phát hiện chính xác các ứng cử viên cho các truy vấn hàm. Chúng tôi đề xuất một chiến lược hai bước, bao gồm tìm kiếm toàn diện và hiệu chuẩn tương đồng. Với sự hỗ trợ của hiệu chuẩn tương đồng, chúng tôi lọc các kết quả sai (ví dụ: các hàm đã được vá lỗi) khi phát hiện các hàm có lỗ hổng, làm cho mô hình của chúng tôi chính xác hơn.

A screenshot of a computer

Description automatically generated

A. Mô hình hàm biểu diễn trung gian

* Để giải quyết thách thức liên kiến kiến trúc, chúng tôi nâng mã nhị phân lên thành IR và đề xuất mô hình Hàm biểu diễn Trung gian (IRFM) sử dụng mô hình RoBERTa [31] để xây dựng nhúng IR. IRFM nhúng các hàm IR vào không gian nhúng chiều cao, giúp các nhúng của các hàm có ý nghĩa tương tự gần nhau trong không gian số học.

1. Biểu diễn Trung gian:

* Đối với các tệp nhị phân từ các kiến trúc khác nhau, chúng tôi tháo dỡ chúng và nâng mã nhị phân lên thành một biểu diễn trung gian (IR) không phụ thuộc vào kiến trúc. Chúng tôi sử dụng IDA Pro [45] và IR của nó, được gọi là mã vi mô, trong việc triển khai của chúng tôi, nhưng các trình giải mã khác và các IR cũng có thể sử dụng (ví dụ: McSema [40]). Như Bảng I hiển thị, các nhóm mã vi mô phân loại các chỉ thị khác nhau từ các kiến trúc khác nhau thành 73 mã opcode và 16 loại toán tử. Ví dụ, mop\_z đại diện cho không toán tử, mop\_r đại diện cho thanh ghi, và mop\_str đại diện cho hằng chuỗi1. Mã vi mô, một biểu diễn trung gian đã được thiết lập, có thể giảm thiểu tác động của sự khác biệt loại chỉ thị đối với tìm kiếm mã nhị phân bằng mã vi mô chéo kiến trúc.
* Xử lý token. Trong mã vi mô, một chỉ thị bao gồm một mã opcode và một bộ tam giác toán tử bao gồm toán tử bên trái, bên phải và đích. Khác với PalmTree [27] chia một chỉ thị thành các yếu tố cơ bản tinh vi (ví dụ: "mov", "[", "+ "và" qword "), chúng tôi chia một chỉ thị thành một mã opcode và ba toán tử (tức là bên trái, bên phải và đích) dựa trên đặc tính của mã vi mô. Địa chỉ cơ sở và độ lệch khác nhau trong các tệp nhị phân khác nhau, gây ra nhiễu và làm cho mô hình ít ổn định hơn. Chúng tôi chuẩn hóa các địa chỉ này (ví dụ: 0x4040E0 và 0x4150D0) bằng một token đặc biệt [addr]. Để giảm thiểu vấn đề OOV (ngoài từ vựng), chúng tôi giới thiệu 16 mã thụ cao cấp tùy theo các loại toán tử mã vi mô trong Bảng I. Đối với các toán tử có trong từ vựng, chúng tôi sử dụng các token riêng của chúng, và đối với các toán tử OOV, chúng tôi sử dụng mã thụ cao cấp để đại diện cho cú pháp cơ bản của chúng. Trong các giai đoạn tiền huấn luyện, chúng tôi thay thế các token, có tần suất ít hơn 100 lần, bằng mã thụ cao cấp của chúng để xây dựng nhúng mã thụ cao cấp.

A diagram of a function model

Description automatically generated

2. Mô hình ngôn ngữ:

* IRFM dựa trên RoBERTa [31], mô hình NLP tiên tiến và rộng rãi được sử dụng. Mô hình sử dụng bộ mã hóa biến đổi đa tầng song hướng để xây dựng nhúng. Vì sự khác biệt cơ bản giữa mã vi mô và ngôn ngữ tự nhiên, chúng tôi thực hiện những cải tiến cần thiết sau đây cho mô hình ngôn ngữ IR.
* Lớp Kiểu Token. Khác với ngôn ngữ tự nhiên, mã vi mô bao gồm các mã opcode và các toán tử, không chỉ là từ đơn. Các mã opcode đại diện cho các hoạt động cần thực hiện (ví dụ: ldx và goto), và các toán tử chỉ ra dữ liệu hoặc vị trí bộ nhớ được sử dụng cho các hoạt động. Xem xét những khác biệt này, chúng tôi sử dụng một lớp kiểu token để giúp IRFM phân biệt mã opcode và các toán tử. Chúng tôi chia các token thành ba loại: mã opcode, toán tử và các loại khác. Loại khác bao gồm các token đặc biệt (ví dụ: [pad]) không có ý nghĩa thực tế.
* Đơn giản hóa Chỉ thị. Mã nhị phân chứa EFLAGS (tức là thanh ghi cờ) như là các toán tử ngầm định. Các EFLAGS này được gán một cách ngầm định bởi các chỉ thị và được sử dụng như đầu vào cho các nhảy điều kiện. Chúng kiểm soát các lộ trình thực thi hàm và có ý nghĩa quan trọng cho cú pháp của hàm. Rất tiếc, hầu hết các phương pháp BCSD tĩnh bỏ qua các EFLAGS này [10], [11], [41], [53], [59], [60]. Mặc dù PalmTree [27] đề cập đến EFLAGS, nhưng không xử lý các EFLAGS liên quan đến các dãy chỉ thị. Các phương pháp phân tích mã nhị phân động, ví dụ: VEX [39], để mô phỏng hành vi chương trình thực tế, phải xem xét tất cả các EFLAGS. Trong các phương pháp BCSD tĩnh, các ý nghĩa quan trọng sẽ bị mất nếu không xem xét các EFLAGS; khi xem xét tất cả các EFLAGS, các EFLAGS dư thừa không chỉ tạo ra chi phí thêm, mà còn có thể che giấu ý nghĩa chính của mã nhị phân. Do đó, chúng tôi chỉ xem xét các EFLAGS đã sử dụng. Chúng tôi chuyển đổi việc gán của mỗi chỉ thị cho các toán tử ngầm định thành một chỉ thị gán thực tế thông qua mã vi mô (ví dụ: Dòng 12 trong Hình 3(a)), và bảo tồn các EFLAGS đã sử dụng của chúng vào các dãy chỉ thị.
* Hình 3(a) cho thấy một chuỗi mã vi mô, bao gồm các biến toàn cục, giá trị trả về, đối số cho các hàm con và các chỉ thị dư thừa. Các chỉ thị dư thừa (ví dụ: Dòng 12-15 trong Hình 3(a)) đem đến các chi phí thêm để xử lý và giảm trọng số của ý nghĩa chính của hàm, ảnh hưởng đến hiệu suất sinh nhúng. Khi cắt giảm các chỉ thị dư thừa, chúng tôi ngăn không xóa các biến toàn cục, đối số cho các hàm con và giá trị trả về, vì các chỉ thị này đại diện cho hành vi của hàm và bao gồm các ý nghĩa cần thiết.

A diagram of a computer

Description automatically generated

* Ở đây, chúng tôi đề xuất một phương pháp đơn giản hóa chỉ thị dựa trên mối quan hệ def-use để cắt giảm các chỉ thị dư thừa và bảo tồn các ý nghĩa quan trọng. Đầu tiên, chúng tôi đánh dấu các chỉ thị "quan trọng" sau đây để tránh xóa: (1) Biến toàn cục và biến cục bộ được lưu trữ trong bộ nhớ thay vì trong thanh ghi, do đó chúng tôi đánh dấu các chỉ thị gán mà toán tử đích là một địa chỉ bộ nhớ, ví dụ: Dòng 18 trong Hình 3(a). (2) Giá trị trả về thường được lưu trữ trong các thanh ghi cụ thể, ví dụ: rax (x86) và x0-x1 (arm). Do đó, chúng tôi đánh dấu các thanh ghi cụ thể dựa trên quy tắc gọi hàm gần chỉ thị trả về trên tất cả các lối, ví dụ: Dòng 19 trong Hình 3(a). (3) Các đối số cho các hàm con xuất hiện trước cuộc gọi hàm con, và chúng không bị ghi đè bởi các chỉ thị khác trước khi được truyền vào hàm con, ví dụ: Dòng 2 trong Hình 3(a). Chúng tôi sử dụng các quy tắc lỏng lẻo để xác định các chỉ thị "quan trọng" để đảm bảo không có ý nghĩa chính bị xóa nhầm. Chúng tôi coi các chỉ thị mà thanh ghi hoặc EFLAGS được xác định trong các chỉ thị sau không được sử dụng là các chỉ thị không được sử dụng (ví dụ: Dòng 5, 12-15 trong Hình 3(a)) và cắt giảm chúng. Sau khi cắt giảm các chỉ thị không được sử dụng, chúng tôi tối ưu hóa các chỉ thị dư thừa (ví dụ: Dòng 5-8 trong Hình 3). Chúng tôi tập trung vào các chỉ thị xác định một thanh ghi được gán trực tiếp cho một biến khác, được gọi là các chỉ thị truyền giá trị. Thông qua đơn giản hóa chỉ thị, 20 chỉ thị của Hình 3(a) được đơn giản hóa thành 9 chỉ thị, bảo tồn ý nghĩa chính của Hình 3(a) và tương tự như mã giả của nó. Điều này giúp IRFM trích xuất ý nghĩa hàm chính xác hơn. Trong thực tế, mô hình RoBERTa chấp nhận độ dài đầu vào hạn chế, và việc đơn giản hóa chỉ thị cho phép đầu vào của RoBERTa bảo tồn nhiều chỉ thị hợp lệ hơn.

1. Các Nhiệm vụ Tiền huấn luyện:

* Trong giai đoạn huấn luyện, chúng tôi sử dụng Masked Language Model (MLM), Root Operand Prediction (ROP) và Adjacent Block Prediction (ABP) để tiền huấn luyện.
* Masked Language Model. Chúng tôi giới thiệu mô hình MLM để hiểu các mối quan hệ giữa mã vi mô và xây dựng các nhúng từ phù hợp. MLM ban đầu được giới thiệu bởi BERT [8], sử dụng các từ ngữ ngữ cảnh xung quanh một mã thông báo lớp để dự đoán mã thông báo đã che mặt nhằm tối ưu hóa các tham số của mô hình. Trong lớp che mặt của MLM, nó ngẫu nhiên chọn 15% mã thông báo để thay thế. Đối với các mã thông báo được chọn, 80% trong số chúng được thay thế bằng mã thông báo [mask], 10% được thay thế bằng các mã thông báo gốc của chúng (mã gốc của opcode vẫn là chính nó), và 10% không thay đổi. Hình 4 thể hiện một ví dụ, trong đó các hộp màu vàng, đỏ và xanh lá cây lần lượt đại diện cho các mã thông báo đã che mặt, mã thông báo đã thay thế và kết quả dự đoán. Trong Hình 4, opcode setz và thanh ghi r0 được che mặt thành [mask], và số hằng số ngay lập tức #0 được thay thế bằng mã thông báo gốc của nó là mop\_n.
* A diagram of a model

  Description automatically generated
* Trong quá trình huấn luyện, chúng tôi đưa các trạng thái ẩn cuối cùng tương ứng với các mã thông báo đã che mặt/thay thế vào một lớp softmax đầu ra qua bộ từ vựng để dự đoán xác suất của các mã thông báo này. Hàm mất mát của MLM sử dụng mất mát Cross-Entropy như sau:
* A math equation with numbers

  Description automatically generated
* Trong đó y là kết quả thực tế, yˆ là dự đoán, |V | là kích thước của từ vựng, và M đại diện cho tập hợp các mã thông báo đã che mặt. θ1 và θ2 là các tham số của IRFM và đầu ra MLM, tương ứng.
* Root Operand Prediction. Chúng tôi đề xuất nhiệm vụ tiền huấn luyện ROP để liên kết ý nghĩa của mã thông báo với ý nghĩa của mã thông báo gốc của chúng, từ đó giúp mô hình của chúng tôi tạo ra ý nghĩa mã thông báo gốc đáng tin cậy hơn cho các từ không có trong từ vựng (OOV). Trong mã vi mô, các toán hạng được chia thành 16 loại (xem Bảng I). Chúng tôi sử dụng chúng làm mã thông báo gốc. Điều này thân thiện với các toán hạng OOV vì chúng ta có thể chuyển đổi các toán hạng OOV thành các mã thông báo gốc của chúng để biểu thị ý nghĩa gốc của chúng. Ví dụ, với một địa chỉ cụ thể là 0xdeadbeef, giả sử đó là một toán hạng OOV, mô hình của chúng tôi gán nó với ý nghĩa của mã thông báo gốc mop\_a, đại diện cho các toán hạng địa chỉ, trong khi mô hình truyền thống không thể phân biệt được ý nghĩa của nó [10], [35]. Vì các mã thông báo gốc của opcode là chính nó, nhiệm vụ ROP sẽ không dự đoán các mã thông báo gốc của opcode. Chúng tôi thực hiện một phần ROP để dự đoán các mã thông báo gốc của chúng. Trong giai đoạn huấn luyện, chúng tôi đưa các trạng thái ẩn cuối cùng của các mã thông báo vào một biến đổi tuyến tính. Chúng tôi sử dụng một đầu ra softmax qua các loại toán hạng để dự đoán xác suất của các mã thông báo gốc này. Hàm mất mát của ROP sử dụng mất mát Cross-Entropy như sau:
* A math equation with black text

  Description automatically generated
* trong đó |VR| là kích thước của từ vựng toán hạng gốc, và S đại diện cho tập hợp các mã thông báo toán hạng. θ1 và θ3 là các tham số của IRFM và đầu ra ROP, tương ứng.
* Nhiệm vụ dự đoán Khối Lân cận. Trong các hàm nhị phân, có mối quan hệ dòng chảy dữ liệu giữa các khối cơ bản. Khác với ngôn ngữ tự nhiên, các biến trong mã nhị phân yêu cầu được định nghĩa trước khi được sử dụng. Các khối cơ bản có mối quan hệ dòng chảy dữ liệu với nhau là có thứ tự, điều này không được trực tiếp nắm bắt bởi IRFM. Để huấn luyện một mô hình hiểu các mối quan hệ dòng chảy dữ liệu giữa các khối kề nhau, chúng tôi đề xuất một nhiệm vụ tiền huấn luyện ABP (Adjacent Block Prediction). Cụ thể, cho hai khối cơ bản A và B, trong đó B là khối kế tiếp của A, biến x được định nghĩa trong khối A và được sử dụng trong khối B. Chúng tôi đánh dấu thứ tự của A-B là tích cực và thứ tự của B-A là tiêu cực. Lưu ý rằng A và B không phải là các khối giống nhau, và A không phải là khối kế tiếp của B. Ngoài ra, chúng tôi không xem xét các trường hợp mà A và B chỉ có mối quan hệ dòng điều khiển mà không có mối quan hệ dòng chảy dữ liệu. Bởi vì nếu không có hỗ trợ từ các mối quan hệ dòng chảy dữ liệu, thứ tự đảo ngược của các khối cũng có thể xảy ra. Chúng tôi đưa trạng thái ẩn cuối cùng của mã thông báo [cls] trong IRFM vào phần ABP, một biến đổi tuyến tính, để xác định liệu hai chuỗi mã vi mô đầu vào có thứ tự tích cực hay không. Hàm mất mát của ABP sử dụng mất mát Cross-Entropy như sau:
* A black text on a white background

  Description automatically generated
* trong đó y là nhãn thứ tự (tích cực hoặc tiêu cực), yˆ là dự đoán và D đại diện cho tập huấn luyện. θ1 và θ4 là các tham số của IRFM và ABP head, tương ứng.
* Hàm mất mát tổng cộng của mô hình ngôn ngữ là sự kết hợp của ba hàm mất mát trên:
* 
* Việc tạo ra nhúng chức năng (function embeddings) là nhiệm vụ của IRFM trong VulHawk. Đầu tiên, chúng tôi tạo ra các nhúng khối cơ bản (basic block embeddings). Đối với các khối mã lệnh đầu vào, bộ mã hóa biến đổi của IRFM xuất ra các chuỗi trạng thái ẩn. Ở đây, chúng tôi áp dụng một lớp mean pooling để tích hợp các nhúng hướng dẫn mã lệnh. Theo các kết quả mô hình được huấn luyện trước [54], các trạng thái ẩn của lớp cuối cùng quá gần với các nhiệm vụ mục tiêu (ví dụ: MLM) trong quá trình huấn luyện trước, điều này có thể tạo sai lệch đến các nhiệm vụ huấn luyện trước này. Các trạng thái ẩn của lớp kế cuối cung cấp sự tổng quát hơn so với các trạng thái ẩn của lớp cuối cùng. Do đó, chúng tôi sử dụng mean pooling trên các trạng thái ẩn của lớp kế cuối để tạo ra các nhúng khối cơ bản.
* Các nghiên cứu hiện có [34], [55] đã chứng minh rằng các giải pháp dựa trên các đồ thị kiểm soát (CFGs) có ưu thế trong các kịch bản băng thông khác nhau. Ở đây, chúng tôi tích hợp các nhúng khối cơ bản và CFGs để tạo ra các nhúng chức năng. Xem xét cấu trúc đa chi nhánh của các chức năng nhị phân, chúng tôi sử dụng Graph Convolutional Networks (GCNs) [24] để nắm bắt cấu trúc CFG và tổng hợp ngữ nghĩa của các khối cơ bản đến các khối cơ bản lân cận của chúng. Chúng tôi coi các chức năng nhị phân là các đồ thị có thuộc tính, trong đó các khối cơ bản là các nút trong đồ thị và các nhúng của chúng là thuộc tính của các nút. Chúng tôi đưa đồ thị điều khiển dựa trên thuộc tính (ACFGs) vào lớp GCN. X(l) đại diện cho các tính năng của các nút lớp thứ l của đồ thị, và hàm tổng hợp được thể hiện như sau:
* A mathematical equation with numbers

  Description automatically generated
* Ở đây, A˜ là ma trận kề có kết nối tự thân (self-connections). IN là ma trận đơn vị (identity matrix). D˜ii là ma trận bậc của các nút, và W(l) là ma trận trọng số có thể điều chỉnh của từng lớp. ReLU là một hàm kích hoạt. Sau một lớp tổng hợp, các khối học cách ngữ nghĩa bối cảnh từ các khối lân cận của chúng.
* Để toàn diện học cách ngữ nghĩa của các khối và cấu trúc của đồ thị kiểm soát (CFGs), chúng tôi sử dụng một mạng GCN gồm 16 lớp để tổng hợp các nhúng ngữ nghĩa của các nút lân cận. Cuối cùng, chúng tôi sử dụng mean pooling trên đầu ra của lớp GCN để tạo ra các nhúng chức năng.

A diagram of a diagram

Description automatically generated

* Đối với việc huấn luyện, cho hai hàm nhị phân, chúng tôi tạo ra giá trị đúng y, tức là không tương đồng (0) và tương đồng (1), dựa trên tên hàm và tệp nguồn. Chúng tôi sử dụng khoảng cách Euclidean để tính toán sự tương đồng s của hai hàm như sau:
* A black text with a white background

  Description automatically generated
* Mục tiêu huấn luyện là làm cho sự tương đồng của các hàm tương đồng tiến tới 1 và sự tương đồng của các hàm không tương đồng tiến tới 0. Chúng tôi sử dụng hàm mất mát Cross-Entropy như sau:
* A close up of a word

  Description automatically generated
* Chính xác, trong đó y là giá trị thực, sij là sự tương đồng của hàm i và j. Chúng tôi sử dụng bộ tối ưu hóa Adam để tối ưu các tham số của GCN và giảm thiểu hàm mất mát LF unction.

B. Entropy-based Adapter:

* Trong thế giới thực, các hàm nhị phân được biên dịch bởi nhiều trình biên dịch với các cấp độ tối ưu hóa khác nhau từ các kiến trúc khác nhau. Trong bài báo này, chúng ta gọi các sự kết hợp của các trình biên dịch, kiến trúc và cấp độ tối ưu hóa là "môi trường tập tin". Các hàm từ các môi trường tập tin khác nhau, ngay cả từ cùng mã nguồn, có thể khác nhau về các chỉ thị và cấu trúc.
* Chúng tôi đề xuất một chiến lược chia để trị. Đầu tiên, chúng tôi chia không gian nhúng của môi trường tập tin hỗn hợp thành nhiều không gian nhúng con. Tiếp theo, chúng tôi chọn một trong các môi trường tập tin V làm môi trường tập tin trung gian và chia bài toán tương tự thành N - 1 bài toán con của việc chuyển nhúng hàm. Cuối cùng, chúng tôi sử dụng các bộ chuyển mà chúng tôi đã huấn luyện để chuyển nhúng hàm từ các môi trường tập tin khác nhau vào cùng một môi trường tập tin V để tính toán sự tương đồng, điều này có thể giảm thiểu sự khác biệt gây ra bởi các môi trường tập tin khác nhau. Với sự giúp đỡ của chiến lược chia để trị, mô hình của chúng tôi tạo ra nhúng hàm mạnh mẽ đối với các môi trường tập tin khác nhau.
* Như được thể hiện trong Hình 5, vấn đề tính toán độ tương đồng với 2.556 kịch bản đã được chia thành 71 vấn đề con, giúp giảm đáng kể sự phức tạp của vấn đề. Phân bố nhúng chức năng khác nhau trong các môi trường tập tin khác nhau. Các bộ chuyển đổi (adapters) chuyển các nhúng chức năng vào cùng một môi trường tập tin và giữ cho các chức năng tương tự được nhóm lại cùng nhau. Như vậy, độ tương đồng chức năng có thể được xác định nhanh chóng dựa trên khoảng cách trong không gian nhúng. Trong bài báo này, chúng tôi xem xét 3 kiến trúc bao gồm x86, arm và mips, 2 kích cỡ từ bao gồm 32-bit và 64-bit, 2 trình biên dịch bao gồm Clang và GCC, và 6 mức tối ưu hóa bao gồm O0, O1, O2, O3, Os và Ofast, tổng cộng có 72 môi trường tập tin (3 × 2 × 2 × 6). Trong việc tìm kiếm mã nhị phân, yêu cầu là nhúng chức năng tương tự từ hai môi trường tập tin bất kỳ nào đó nên gần nhau, điều này là một vấn đề phức tạp với 2.556 kịch bản (C2 72). Chiến lược chia và chinh phục của chúng tôi chuyển đổi vấn đề này với 2.556 kịch bản thành 71 vấn đề chuyển đổi nhúng. Xem xét rằng số O1 nằm ở giữa tất cả các mức tối ưu hóa, chúng tôi chọn môi trường tập tin O1-GCC-x86-64 làm môi trường tập tin trung gian. Lưu ý rằng cũng có thể chọn môi trường tập tin khác làm môi trường tập tin trung gian.
* Phân tích nhị phân dựa trên thông số thông tin: Đối với chiến lược chia và chinh phục, một bước quan trọng là xác định môi trường tập tin. Các kiến trúc và kích cỡ từ của các chức năng nhị phân có thể được xác định thông qua các hướng dẫn của chúng. Tuy nhiên, vấn đề là không có thông tin để chỉ rõ các trình biên dịch và mức tối ưu trong các tập tin nhị phân.
* Để giải quyết vấn đề này, chúng tôi sử dụng quan điểm thông tin học để hiểu các tập tin nhị phân và giới thiệu thông số thông tin (entropy) để xác định trình biên dịch và mức tối ưu của các tập tin nhị phân. Nói chung, các đoạn mã đã được nén hoặc mã hóa có thông số thông tin cao hơn mã nguồn tự nhiên. Nguyên tắc này cũng có thể được áp dụng để phân biệt các trình biên dịch và tối ưu hóa khác nhau.
* Từ quan điểm thông tin học, hai đoạn mã nhị phân, biên dịch từ cùng mã nguồn với cấp độ tối ưu hóa O0 và O3 tương ứng, mang cùng một ngữ nghĩa mã nguồn. Tuy nhiên, do tối ưu hóa bởi trình biên dịch, đoạn mã nhị phân với O3 thường được tinh chế hơn so với O0. Do đó, entropy của đoạn mã nhị phân với O3 khác với các đoạn mã với O0. Hình 6 hiển thị các dãy entropy từ 12 đoạn mã nhị phân khác nhau từ 3 môi trường tập tin. Dãy entropy của một tập tin nhị phân được tính bằng cách chia các byte thô thành biểu diễn thập lục phân (0x00-0xFF). Có thể quan sát thấy rằng các dãy entropy từ cùng môi trường tập tin có vẻ giống nhau trong khi khác biệt so với các dãy thuộc các môi trường tập tin khác nhau. Bằng cách sử dụng dãy entropy và lý thuyết entropy, chúng ta có thể xác định các trình biên dịch và tối ưu hóa khác nhau.

A screenshot of a diagram

Description automatically generated

* Để tránh vấn đề va chạm mà dãy entropy đơn lẻ có thể gây ra, chúng tôi sử dụng các đặc trưng sau đây:
* Dãy entropy của phân đoạn .text, bao gồm 256 xác suất tương ứng của byte thô (0x00-0xFF).
* Entropy của phân đoạn .text, tổng của dãy entropy trên phân đoạn .text. Điều này tập trung vào phần thực thi trong các tập tin nhị phân và tránh tác động của các thay đổi trong phần dữ liệu.
* Entropy của tập tin, tổng của dãy entropy trên toàn bộ tập tin, cung cấp thông tin toàn cục tại cấp độ tập tin.
* Chúng tôi sử dụng một Mạng Nơ-ron Residual (ResNet) [50] làm bộ phân loại, và các đặc trưng nêu trên làm đầu vào để xác định môi trường tập tin. Hình 7 cho thấy cấu trúc của khối nơ-ron Residual cơ bản. Nó bao gồm chuẩn hóa dữ liệu theo lô và biến đổi tuyến tính, và hàm kích hoạt là ReLU. Một kết nối nhảy sử dụng bản đồ danh dự từ đầu vào được thêm vào đầu ra của khối nơ-ron Residual cơ bản, điều này giữ nguyên ngữ nghĩa của hàm và giúp giải quyết vấn đề biến mất gradient [50]. Chúng tôi lấy mẫu các đầu vào bằng một lớp tích chập biến đổi 258 chiều đầu vào thành các đặc trưng 64 chiều. Sau 16 lớp khối nơ-ron Residual cơ bản, chúng tôi sử dụng dự đoán tuyến tính và hàm softmax để thực hiện phân loại đa lớp. Vì Os và Ofast là các tối ưu hóa tăng tiến trên O2 và O3 tương ứng, chúng tôi bao gồm Os và Ofast vào O2 và O3 tương ứng. Ở đây, mục tiêu của softmax là 8 lớp (2 trình biên dịch × 4 cấp độ tối ưu hóa). Phân tích entropy giúp chúng tôi xác định môi trường tập tin (trình biên dịch và cấp độ tối ưu hóa) để chuẩn bị cho việc chuyển đổi nhúng hàm tiếp theo.

1. Lớp chuyển đổi dựa trên entropy: Để tính toán độ tương tự của các hàm nhị phân từ các môi trường tập tin khác nhau, chúng tôi đề xuất một lớp chuyển đổi dựa trên entropy sau IRFM. Lớp chuyển đổi dựa trên entropy hoạt động như một ánh xạ F để chuyển đổi các nhúng hàm từ các môi trường tập tin khác nhau vào cùng một môi trường tập tin trung gian V để giảm thiểu sự khác biệt do các môi trường tập tin khác nhau gây ra. Ánh xạ F cần đồng thời giữ nguyên ngữ nghĩa của hàm và giảm nhẹ sự lệch do các môi trường tập tin khác nhau.

* Giả sử có hai nhúng hàm x và x0 từ cùng một mã nguồn nhưng khác nhau về môi trường tập tin V và V0, chúng ta viết lại x dưới dạng x0 − α trong đó α đại diện cho sự lệch giữa x và x0. Độ tương tự của chúng được viết dưới dạng s =
* 1
* 1+|(x0−α)−x0
* |2
* . Khi sự lệch α tiến tới 0, độ tương tự s tiến tới 1. Do đó, ánh xạ F là sự kết hợp giữa nhúng hàm chính nó và sự lệch, trong đó chúng ta tìm kiếm sự lệch với hàm f(x, V). Chúng tôi sử dụng ResNets [50] để xây dựng ánh xạ F. Trọng số của bộ chuyển đổi khác nhau tùy theo môi trường tập tin đầu vào V.
* Huấn luyện. Để giảm độ phức tạp huấn luyện, chúng tôi đóng băng các tham số của IRFM. Chúng tôi sử dụng độ tương tự của hàm làm giá trị thực, tức là không tương tự (0) và tương tự (1). Mục tiêu của việc huấn luyện là làm cho độ tương tự của các hàm tương tự tiến tới 1 và độ tương tự của các hàm không tương tự tiến tới 0. Chúng tôi sử dụng hàm mất mát Cross-Entropy làm hàm mất mát:
* A number and a plus one

  Description automatically generated
* Trong hàm mất mát, y đại diện cho sự thật cơ bản, và sij đại diện cho độ tương đồng giữa các hàm i và j. Chúng tôi sử dụng bộ tối ưu Adam để tối ưu hóa các tham số để giảm thiểu hàm mất mát L. Đáng chú ý rằng khi có sự xuất hiện của các trình biên dịch và cấp độ tối ưu mới, chúng ta chỉ cần huấn luyện các bản đồ tương ứng cho các môi trường tập tin mới, điều này làm cho phương pháp của chúng tôi trở nên linh hoạt và thực tế hơn.

C. Chiến lược tìm kiếm tiến tiến

* Các phương pháp hiện có [10], [27], [35] sử dụng nhúng hàm để tìm kiếm các hàm tương tự. Đây là một phương pháp phát hiện thô thiểu thông tin chi tiết (ví dụ: các tính năng cấp block), giúp giảm thiểu công sức tìm kiếm nhưng dẫn đến nhiều kết quả dương giả, đặc biệt là đối với các hàm đã bị thay đổi nhỏ. Trong khi đó, Marcelli và cộng sự [34] sử dụng Mạng Khớp Đồ Thị [28] và các phương pháp khác [58], [60] sử dụng mạng Siamese [3] để tính toán sự tương đồng của từng cặp hàm ở mức chi tiết, điều này có thể đạt được hiệu suất cao nhưng tốn kém tính toán.
* Đối mặt với kịch bản phát hiện lỗ hổng phức tạp, chúng tôi đề xuất một chiến lược tìm kiếm tiến tiến mới, có tên là Chiến lược Tìm kiếm Tiến Tiến, để giảm thiểu gánh nặng tính toán trong khi duy trì hiệu suất tốt và giảm số kết quả dương giả do các hàm đã bị thay đổi trong việc phát hiện lỗ hổng. Chiến lược này kết hợp hai phân chiến lược. Trước tiên, chúng tôi sử dụng nhúng hàm như tổng quan toàn cầu để tìm kiếm theo cấu trúc thô. Tiếp theo, chúng tôi thiết kế việc hiệu chỉnh sự tương đồng theo cặp cho các hàm ứng viên để bổ sung thông tin cấp block cho nhúng hàm nhằm duy trì độ chính xác cao trong việc phát hiện lỗ hổng.
* Tìm kiếm theo Nhúng Hàm: Với sự hỗ trợ của bộ chuyển đổi dựa trên entropy, nhúng hàm được tạo ra bởi IRFM cho các hàm tương tự sẽ gần nhau trong không gian nhúng. Để tìm hiểu hiệu quả các hàm tương tự trong một kho chứa hàm lớn, chúng tôi sử dụng sự tương đồng khoảng cách Euclidean của nhúng hàm để lấy các ứng viên theo cấu trúc thô, điều này sẽ giảm thiểu phạm vi của phát hiện cấu trúc chi tiết và giảm thiểu công sức tính toán.
* Trong thuật toán 1, dòng 2, chúng tôi thực hiện các tính toán ma trận trên các nhúng hàm đã tạo trước đó để thu được sự tương đồng khoảng cách Euclidean. Cho một tập truy vấn chứa n hàm và một kho chứa hàm chứa m hàm, chúng tôi sử dụng VulHawk để tạo ra các ma trận nhúng hàm của chúng, trong đó N ∈ R
* n×d
* và M ∈ R
* m×d
* . Hàm tính toán ma trận như sau:
* A group of black letters

  Description automatically generated
* Trong đó 1m T ∈ R 1×m được sử dụng để mở rộng một vector cột thành một ma trận. Kết quả Y ∈ R n×m biểu thị các khoảng cách giữa tập truy vấn và kho chứa hàm, trong đó yij biểu thị khoảng cách giữa hàm thứ i trong tập truy vấn và hàm thứ j trong kho chứa hàm. Sau đó, chúng ta chuyển ma trận khoảng cách Euclidean Y thành ma trận tương đồng S (sij ∈ [0, 1]) bằng Công thức 7. Trong thuật toán 1 dòng 4, với ngưỡng mặc định h, chúng tôi lấy top-K ứng viên làm kết quả. Lưu ý rằng có thể sử dụng các thư viện như Faiss [21] để tăng tốc tìm kiếm tiếp theo.

1. Hiệu chuẩn độ tương đồng:

* Tương tự như các phương pháp khác, nhúng hàm của chúng tôi được thiết kế để đối phó với các tình huống giao cắt giữa các môi trường tệp khác nhau. Chúng chịu được những thay đổi nhỏ. Đồng thời, chúng cũng có thể nhạy cảm với các bản vá lỗi nhỏ do thiếu thông tin tinh vi. Điều này sẽ gây ra những kết quả dương tính không thể chấp nhận được và tăng gánh nặng phân tích cho các nhà nghiên cứu. Để tìm kiếm mã nhị phân chính xác cao, chúng tôi đề xuất một phương pháp hiệu chuẩn độ tương đồng để phát hiện chi tiết cụ thể. Nó kết hợp thông tin của các khối cơ bản, các hằng số chuỗi và các hàm được nhập vào để tính toán điểm tương đồng cặp, từ đó trích xuất các vector và kết hợp với thông tin cấp hàm để cải thiện hiệu suất của việc phát hiện lỗ hổng.
* Đặc trưng cấp khối. Các đặc trưng cấp khối, như phân phối nhúng khối và kích thước hàm, có thể bị mất đi bởi nhúng cấp hàm. Trong nhiều trường hợp, sự khác biệt giữa các hàm nằm trong các cấu trúc con nhỏ và khó có thể được phản ánh bằng nhúng hàm. Một sự tương đồng là, trong việc phù hợp đồ thị, hiệu suất phù hợp đồ thị dựa trên nhúng đồ thị có thể được cải thiện thông qua thông tin cấp nút tinh vi. Trong thuật toán 1 dòng 6, chúng tôi tính toán sự tương đồng của tập khối cơ bản giữa hàm được truy vấn và các hàm ứng cử viên của nó để bổ sung thông tin cấp khối. Cho hai hàm f1 và f2, chúng tôi đầu tiên sử dụng Công thức 10 để tính toán sự tương đồng của tập khối cơ bản của chúng, sau đó đếm sự tương đồng tối đa trong các khoảng [0, 0.2), [0.2, 0.4), [0.4, 0.6), [0.6, 0.8) và [0.8, 1.0]. Bằng cách này, chúng ta tạo ra một vector 5 chiều Vblk làm bổ sung cấp khối cơ bản.

A screenshot of a computer program

Description automatically generated

* Mô tả tính năng của các chuỗi, các hàm đã được nhập, và các đặc điểm cơ bản (basic block) giúp chúng ta xác định độ tương đồng giữa các cặp hàm tương tự. Ở Đoạn mã 1 dòng 7, chúng tôi tính độ tương đồng của các chuỗi. Chúng tôi sử dụng mô hình đã được huấn luyện trước Sentence-BERT để tạo ra các nhúng cho các chuỗi đã nối lại với nhau. Theo cài đặt của Sentence-BERT, chúng tôi sử dụng độ tương đồng cosine của các nhúng chuỗi. Ở đây, chúng tôi tạo ra một vector 2 chiều Vstr: độ tương đồng chuỗi và tổng độ dài của các chuỗi.
* Các hàm đã nhập. Ở Đoạn mã 1 dòng 8, chúng tôi sử dụng Chỉ số Jaccard để tính toán độ tương đồng si của hai tập hàm đã nhập I1 và I2: si = 1−
* |I1∩I2|
* |I1∪I2|
* , trong đó |I1∩I2| đại diện cho số lượng hàm đã nhập xuất hiện trong cả hai tập, và |I1∪I2| đại diện cho tổng số lượng hàm đã nhập. Ở đây, chúng tôi tạo ra một vector 2 chiều Vimp: độ tương đồng của các hàm đã nhập và tổng số lượng hàm đã nhập.
* Sau khi tính toán các vector trên, chúng tôi kết hợp chúng với độ tương đồng s từ quá trình tìm kiếm các hàm bằng nhúng chức năng thành vector V. Sau đó, chúng tôi đưa vector V vào mạng lan truyền thuận (feed-forward network) để học các trọng số và dự đoán độ tương đồng cuối cùng của các hàm s
* 0
* . Chúng tôi sử dụng hàm mất mát Cross-Entropy để tối ưu trọng số mạng. Cuối cùng, chúng tôi sử dụng ngưỡng mặc định h để lọc ra các hàm tương tự như kết quả.

IV. EVALUATION

* Trong phần này, chúng ta sẽ đánh giá VulHawk bằng cách so sánh với các phương pháp tìm kiếm mã nhị phân và phát hiện độ tương đồng mã nhị phân hiện có trạng thái tiên tiến khác trong ba kịch bản khác nhau: chéo biên dịch viên, chéo cấp độ tối ưu hóa và chéo kiến trúc.
* Các thử nghiệm nhằm trả lời các câu hỏi nghiên cứu sau:
* RQ.1: Cho hai hàm nhị phân, liệu VulHawk có thể xác định chúng có tương đồng hay không (tức là so sánh một-một)?
* RQ.2: VulHawk có thể được sử dụng để tìm kiếm một hàm trong kho hàm lớn không (tức là tìm kiếm một-nhiều)?
* RQ.3: VulHawk có thể xác định có bao nhiêu hàm tương đồng từ hai mã nhị phân không (tức là nhiều-nhiều)?
* RQ.4: Hiệu quả của VulHawk trong việc tìm kiếm trong kho hàm lớn như thế nào (tức là hiệu quả thời gian chạy)?
* RQ.5: Đóng góp của từng thành phần trong VulHawk như thế nào (tức là nghiên cứu loại bỏ thành phần)?
* RQ.6: VulHawk có thể phát hiện các lỗ hổng 1-ngày trong thế giới thực hay không?

A white text with black text

Description automatically generated

A. Thực hiện và Cài đặt Thí nghiệm

* Chúng tôi triển khai VulHawk bằng cách sử dụng các thư viện Transformers [52], NetworkX [17] và PyG [51], dựa trên Python 3.8.5. Trình disassembler mà chúng tôi sử dụng là IDA Pro 7.5 [45]. Các thí nghiệm của chúng tôi được thực hiện trên một máy tính để bàn chạy hệ điều hành Windows 10, với bộ vi xử lý Intel Core i9-10920X, RAM DDR4 64GB và một GPU NVIDIA RTX3090.
* Để hỗ trợ nghiên cứu tiếp theo, chúng tôi đã đưa các chương trình và tập dữ liệu lên GitHub để công khai.
* Các siêu tham số của chúng tôi như sau: trong mô hình RoBERTa, layer=6, head=8 và hidden dimension=256; trong mô hình GCN, hidden dimension=256 và layer=16; trong mô hình nhận diện môi trường tập tin, layer=8, input dimension=258, hidden dimension=64; trong mô hình entropy-based adapter, layer=2 và hidden dimension=256; trong các mạng hiệu chuẩn, input dimension=10 và layer=3. Các thiết lập này là sự cân đối giữa hiệu suất và hiệu năng.

1. Tập dữ liệu: Tương tự như công trước [10], [11], [27], [30], [34], [35], [55], chúng tôi sử dụng các dự án sau: Coreutils-8.30, Curl-7.70, Diffutils-3.6, Findutils-4.7.0, Libmicrohttpd-0.9.75, mtools-4.0.36, OpenSSL-1.1.1l, putty-0.74, wget2-2.0.0 và sqlite-3.37.1. Tập dữ liệu này được sử dụng rộng rãi trong thực tế và các công việc liên quan. Chúng tôi biên dịch các dự án này bằng 2 trình biên dịch (GCC-10 và Clang-10), 6 cấp độ tối ưu hóa (O0, O1, O2, O3, Os và Ofast), 3 kiến trúc (x86, arm và mips) và 2 kích thước từ (32-bit và 64-bit). Tổng cộng, chúng tôi thu được 3.393 tệp chứa 596.099 hàm nhị phân và 13.398.845 khối cơ bản. Để đánh giá khả năng tổng quát hóa của mô hình đã được huấn luyện trên các nhị phân chưa được biết trước, chúng tôi ngẫu nhiên chọn một nửa các nhị phân trong tập dữ liệu của chúng tôi để huấn luyện mô hình. Bất kỳ nhị phân nào trong tập huấn luyện không xuất hiện trong tập đánh giá, điều này đảm bảo tập đánh giá là không biết đối với VulHawk. Các thiết lập này đảm bảo các nhị phân được tạo ra gần giống với trường hợp thực tế, khiến việc đánh giá trở nên thiết thực và đáng tin cậy. Như đã hiển thị trong bảng II, chúng tôi xác định bảy nhiệm vụ khác nhau để đánh giá: (1) XO, (2) XC, (3) XA, (4) XO+XC, (5) XO+XA, (6) XC+XA và (7) XO+XC+XA, trong đó ×/ đại diện cho các cặp hàm có cùng/khác nhau cho cài đặt này. Ví dụ, nhiệm vụ XO+XC đại diện cho các cặp hàm có tối ưu hóa và trình biên dịch khác nhau nhưng kiến trúc giống nhau.
2. Các cơ sở đối sánh: Trong các thí nghiệm so sánh, chúng tôi chọn các phương pháp tiên tiến nhất hiện có làm cơ sở để so sánh:

* PalmTree [27], phương pháp tiên tiến nhất cho BCSD, sử dụng các mô hình được huấn luyện trước để tạo ra nhúng hướng dẫn có thể được sử dụng để tính toán độ tương tự của các hàm.
* SAFE [35] sử dụng mô hình word2vec để tạo ra các nhúng hướng dẫn và đề xuất một mạng neural recurrent để tạo ra các nhúng hàm.
* Trex [41], phương pháp tiên tiến nhất cho BCSD, sử dụng các mô hình dựa trên micro-traces để tạo ra các nhúng hàm để so khớp các hàm tương tự.
* Asteria [58] sử dụng mạng Tree-LSTM dựa trên cây cú pháp trừu tượng (ASTs) và sử dụng mạng Siamese [3] để tính toán độ tương tự của các hàm, trong đó các thí nghiệm của nó cho thấy Asteria vượt trội hơn Gemini [55].
* Asm2Vec [10] sử dụng mô hình học không giám sát để tạo ra các nhúng hàm sử dụng mô hình PV-DM.
* BinDiff [61], một công cụ BCSD thương mại tiên tiến, sử dụng nhiều đặc trưng để thực hiện phát hiện hàm tương tự. Chúng tôi đo nó với phiên bản 7 mới nhất với các thiết lập mặc định.
* Graph Matching Networks (GMN) [28]. Các nghiên cứu hiện có [34] cho thấy GMN dựa trên CFG có những ưu điểm tự nhiên trong các tình huống chéo kiến trúc. GMN được cài đặt như nghiên cứu [34].
* Đối với các cơ sở trên, chúng tôi sử dụng các cài đặt gốc và các thiết lập mặc định của chúng. Để đánh giá đóng góp của adapter dựa trên entropy và chiến lược hiệu chuẩn độ tương tự, chúng tôi thiết lập ba cấu hình:
* VulHawk: VulHawk gốc.
* VulHawk-ES: VulHawk thay thế adapter dựa trên entropy bằng mạng neural và không sử dụng hiệu chuẩn độ tương tự.
* VulHawk-S: VulHawk không có hiệu chuẩn độ tương tự.

B. So sánh một-sang-một

* Chúng tôi kiểm tra hiệu suất của VulHawk và các cơ sở đối sánh với việc phát hiện độ tương tự của hàm một-sang-một, một phương pháp rộng rãi được thực hiện trong các phương pháp trước đây [27], [34], [41], [55]. Như trong cài đặt thí nghiệm của họ, chúng tôi xây dựng một tập đánh giá cân bằng gồm 50k cặp hàm dương và 50k cặp hàm âm, và một tập đánh giá không cân bằng gồm 1.400 cặp hàm dương và 140k cặp hàm âm cho mỗi nhiệm vụ. Chúng tôi sử dụng diện tích dưới đường cong (AUC) của đường cong ROC làm đo lường. AUC là một đo lường hiệu suất toàn diện của một mô hình tích hợp tất cả các ngưỡng phân loại có thể có. Bảng III cho thấy các kết quả so sánh của VulHawk và các cơ sở đối sánh khác.
* Như đã thấy, VulHawk vượt trội hơn SAFE, Asteria, GMN, PalmTree, Asm2Vec và Trex về điểm AUC trên cả tập dữ liệu cân bằng và không cân bằng trong tất cả các thiết lập thí nghiệm. Ví dụ, trong thí nghiệm giao kiến trúc (XA), VulHawk đạt được AUC là 0,998, trong khi Trex đạt AUC là 0,947, Asteria đạt AUC là 0,951, SAFE chỉ đạt AUC là 0,509 và PalmTree và Asm2Vec thất bại trong thí nghiệm giao kiến trúc. PalmTree và Asm2Vec tập trung vào một tập hướng dẫn duy nhất (x86), không thể xử lý các hàm từ các kiến trúc khác nhau. Mặc dù SAFE huấn luyện mô hình của nó trên các tập hướng dẫn khác nhau, nhưng vẫn khó để xây dựng các mối quan hệ ngữ nghĩa giữa các hướng dẫn từ các ISA khác nhau và nhúng các hàm tương tự từ các kiến trúc khác nhau thành các vector tương tự. Như đã nêu trong các vấn đề trên Github của nó, SAFE hiện tại khó có thể hỗ trợ các nhiệm vụ giao kiến trúc. Mặc dù GMN đạt AUC là 0,865 trong nhiệm vụ XA, nhưng nó thực hiện kém trong các nhiệm vụ khác. Vì CFGs đều chắc chắn đối với các kiến trúc nhưng thay đổi ở các trình biên dịch và mức tối ưu khác nhau. VulHawk chuyển đổi mã nhị phân thành mã vi nhỏ để giảm thiểu vấn đề của các ISA khác nhau và sử dụng bộ chuyển đổi dựa trên entropy để chuyển các nhúng vào môi trường tập tin trung gian, điều này khiến VulHawk vượt trội hơn các cơ sở đối sánh khác. Hình 8 trình bày chi tiết các đường cong ROC của VulHawk và các cơ sở đối sánh khác, trong đó càng gần đường cong ROC với góc trên bên trái càng tốt hiệu suất. Như đã thấy trong Hình 8, càng nhiều sự khác biệt về môi trường tập tin (trình biên dịch, kiến trúc và mức tối ưu hóa) giữa các hàm, hiệu suất so sánh hàm càng kém. May mắn thay, nhờ sự hỗ trợ của adapter dựa trên entropy và hiệu chuẩn độ tương tự, VulHawk có khả năng chống lại những sự khác biệt này và đạt được hiệu suất cao.
* Trả lời RQ.1: VulHawk xếp hạng đầu tiên trong tất cả các nhiệm vụ so sánh một-sang-một, chứng minh tính ổn định của VulHawk sử dụng chiến lược chia để trị đối với các nhiệm vụ giao kiến trúc, giao mức tối ưu hóa và giao trình biên dịch.

A table with numbers and letters

Description automatically generated

A group of graphs with different colored lines

Description automatically generated

C. Tìm kiếm một-sang-nhiều

* Trong phần này, chúng tôi đánh giá hiệu suất của tìm kiếm một-sang-nhiều. Giống như trong nghiên cứu [34], chúng tôi sử dụng các đo lường xếp hạng để đánh giá hiệu suất mô hình trong các ứng dụng tìm kiếm, ví dụ như tìm kiếm lỗ hổng, trong đó mô hình phải lấy các hàm ứng cử viên từ cơ sở dữ liệu lớn. Chúng tôi sử dụng độ nhớ lại (Recall@K) ở các ngưỡng K khác nhau làm đo lường, đây là một đo lường xếp hạng được sử dụng rộng rãi. Chúng tôi sử dụng tập đánh giá không cân bằng (trong phần IV-B) làm tập đánh giá. Trong quá trình đánh giá, các mô hình tính toán và xếp hạng độ tương tự giữa từng hàm được truy vấn và các mẫu tích cực/âm tích cực của nó.
* Chúng tôi thu thập độ nhớ lại ở các kết quả top-K khác nhau và vẽ đồ thị độ nhớ lại so với k trong Hình 9. Kết quả cho thấy VulHawk vượt trội hơn các phương pháp hàng đầu và đạt được giá trị recall@1 tốt nhất là 0,935 trong nhiệm vụ XO và 0,879 trong nhiệm vụ XC+XO+XA. Trong nhiệm vụ XC+XO+XA, khi số lượng kết quả được thu thập lên đến 30, độ nhớ lại của mỗi phương pháp có xu hướng ổn định, trong đó VulHawk đạt được độ nhớ lại @30 xung quanh 0,994, VulHawk-ES đạt được độ nhớ lại @30 xung quanh 0,968, VulHawk-S đạt được độ nhớ lại @30 xung quanh 0,988, Trex đạt được độ nhớ lại @30 xung quanh 0,888 và SAFE đạt được độ nhớ lại @30 xung quanh 0,310. Độ nhớ lại @K của SAFE trong nhiệm vụ XC+XO+XA gần với xác suất ngẫu nhiên ( K/100 ), vì SAFE không ổn định trên các nhiệm vụ giao kiến trúc do vấn đề OOV nặng, điều này đã được thể hiện trong so sánh một-sang-một. Trong tìm kiếm một-sang-nhiều trong một cơ sở dữ liệu hàm lớn, điểm yếu của SAFE được phóng đại, vì vậy nó chỉ đạt được độ nhớ lại @1 là 0,007.
* Trả lời RQ.2: VulHawk có thể lấy chính xác các ứng cử viên tốt nhất trong một cơ sở dữ liệu hàm lớn.

A graph of different values

Description automatically generated

D. Nhiều-sang-nhiều Khớp

* Chúng tôi tiến hành thực nghiệm để đo đạc hiệu suất trong nhiệm vụ nhiều-sang-nhiều khớp, mà đã được thực hiện trong các nghiên cứu trước [10], [61]. Nhiệm vụ nhiều-sang-nhiều khớp được sử dụng để đo lường sự tương đồng của hai tập tin nhị phân đã cho ở mức hàm. Tương tự như cài đặt thử nghiệm của họ, chúng tôi xây dựng một tập đánh giá các cặp tập tin nhị phân cho mỗi nhiệm vụ, và mỗi công cụ sinh ra độ tương đồng hàm khớp tốt nhất cho mỗi cặp tập tin nhị phân. Chúng tôi báo cáo kết quả độ nhớ lại và độ chính xác trung bình của mỗi công cụ trong các nhiệm vụ khác nhau trong Bảng IV.
* Trong số đó, các phương pháp cơ sở có kết quả thấp nhất trong nhiệm vụ XO với O0-O3. VulHawk đạt độ nhớ lại là 0,876 trong nhiệm vụ này, cải thiện độ nhớ lại lần lượt là 385,9%, 292,9%, 208,6%, 240,0%, 211,6%, 621,3% và 202,8% so với SAFE, Asteria, Asm2Vec, BinDiff, PalmTree, GMN và Trex. Thú vị là, kết quả tồi nhất của VulHawk (0,805) là trong nhiệm vụ XC, không phải trong các nhiệm vụ O0-O3. Tuy nhiên, VulHawk vẫn vượt trội hơn các phương pháp hàng đầu trong các nhiệm vụ XC. Trong tùy chọn O3, để giảm kích thước và tăng tốc hiệu suất của các hàm nhị phân, nó nén các hướng dẫn trùng lặp trong tùy chọn O0 thành hướng dẫn ngắn gọn, làm mạnh hơn các ý nghĩa chính và loại bỏ các ý nghĩa dư thừa của các hàm, dẫn đến sự khác biệt về ý nghĩa. VulHawk sử dụng việc đơn giản hóa hướng dẫn để lọc bỏ các hàm nhị phân và giữ lại các ý nghĩa chính, giúp VulHawk giảm thiểu tác động của các cấp độ tối ưu hóa và làm cho nó chống lại các thí nghiệm giao cắt các cấp độ tối ưu hóa.

A screenshot of a computer screen

Description automatically generated

A screenshot of a graph

Description automatically generated

* Hình 10 cho thấy phân phối của độ nhớ lại (recall) và độ chính xác (precision) trong các nhiệm vụ XC, XA và XO bằng các biểu đồ violin, trong đó chúng tôi chú thích kết quả trung bình của mỗi phương pháp trên đỉnh của hình. So với SAFE, BinDiff, Asteria, Asm2vec, PalmTree, GMN và Trex, phân phối xác suất độ nhớ lại và độ chính xác của VulHawk gần với 1 và tập trung hơn, trong khi phân phối kết quả của các phương pháp khác phân tán và không ổn định trong các tình huống khác nhau. Điều này cho thấy hiệu suất của VulHawk tốt hơn và ổn định hơn so với các phương pháp cơ sở khác.
* So với việc so sánh một-sang-một, tỷ lệ độ nhớ lại và độ chính xác trong việc khớp nhiều-sang-nhiều thấp hơn, bởi vì có nhiều mẫu tiêu cực hơn trong việc khớp nhiều-sang-nhiều, và các cặp không khớp sai ảnh hưởng đến kết quả tiếp theo của thuật toán khớp (ví dụ: thuật toán Hungary [25]). Trả lời RQ.3: VulHawk có thể được sử dụng để khớp các hàm tương tự giữa hai tập tin nhị phân, và nó vượt trội hơn các phương pháp hàng đầu trong việc khớp nhiều-sang-nhiều.

E. Hiệu suất thời gian chạy

* Chúng tôi đánh giá hiệu suất thời gian chạy của VulHawk với các cài đặt khác nhau. Cho một hàm, chúng tôi sử dụng mô hình cần đánh giá để trích xuất các đặc trưng và tạo ra hàm nhúng của nó, sau đó lấy ra 10 hàm ứng cử viên từ kho lưu trữ. Thời gian ghi chép là từ việc trích xuất đặc trưng đến trả về độ tương tự của các hàm ứng cử viên, và mỗi thử nghiệm được đo mười lần để giảm thiểu các yếu tố tình cờ. Kích thước kho lưu trữ được đặt thành 10^3, 10^4, 10^5 và 10^6.
* Bảng V cho thấy chi phí thời gian của việc tìm kiếm một hàm trong kho lưu trữ có kích thước khác nhau và khả năng thông qua của chúng. Khả năng thông qua đại diện cho số lượng các hàm trong kho lưu trữ có thể được lấy ra trong một giây, không tính đến chi phí thêm của việc tạo ra nhúng. Kết quả cho thấy VulHawk chậm hơn VulHawk-ES và VulHawk-S, bởi vì VulHawk sử dụng bộ chuyển đổi dựa trên entropy trong quá trình tạo ra nhúng và sử dụng hiệu chỉnh độ tương tự trong quá trình tìm kiếm. Tuy nhiên, việc tăng một chút overhead để đạt được độ chính xác và độ nhớ lại cao là chấp nhận được. Với sự hỗ trợ của GPU, VulHawk có thể tìm kiếm một hàm từ 10^6 hàm trong khoảng 2 giây.
* Trả lời RQ.4: VulHawk có thể duy trì hiệu suất cao trong kho lưu trữ hàm lớn (10^6).

A table with numbers and a black text

Description automatically generated

F. Nghiên cứu rút gọn

* Chúng tôi phân tích đóng góp của bộ chuyển đổi dựa trên entropy và chiến lược hiệu chuẩn độ tương tự trong VulHawk.
* Bộ chuyển đổi dựa trên entropy. Như đã thấy trong Bảng III, VulHawk-S có một AUC cao hơn so với VulHawk-ES trong bảy tác vụ so sánh một-nhiều hàm. Trong Hình 9, VulHawk-S đạt được các giá trị nhớ@K cao hơn VulHawk-ES trong tìm kiếm một-nhiều hàm. Điều này cho thấy rằng VulHawk-S vượt trội hơn VulHawk-ES, và đó chính là đóng góp của bộ chuyển đổi dựa trên entropy, không phải là do các mạng neural bổ sung thêm. Bộ chuyển đổi dựa trên entropy chuyển các nhúng hàm từ môi trường tập tin khác nhau thành cùng một môi trường tập tin, giảm thiểu sự khác biệt do trình biên dịch và cấp độ tối ưu hóa. Ví dụ, trong các tình huống ghép nhiều-nhiều, VulHawk-ES đạt mức nhớ là 0,793 trong thử nghiệm O0-O3, trong khi VulHawk-S, nhờ sự hỗ trợ của bộ chuyển đổi dựa trên entropy, giảm thiểu sự khác biệt do môi trường tập tin khác nhau, đạt mức nhớ là 0,873.
* Hiệu chuẩn độ tương tự. Bảng III và Hình 9 cho thấy VulHawk đạt được kết quả tốt hơn so với VulHawk-S trong tác vụ so sánh một-nhiều hàm và tìm kiếm một-nhiều hàm. Bảng IV cũng cho thấy VulHawk chính xác hơn VulHawk-S trong các tình huống ghép nhiều-nhiều. Độ chính xác của VulHawk tăng từ 0,593 lên 0,818 trong nhiệm vụ O0-O3 của ghép nhiều-nhiều, bởi vì VulHawk sử dụng chiến lược hiệu chuẩn tương tự để bổ sung độ tương tự hàm từ các đặc trưng cấp đoạn, đặc trưng chuỗi và các hàm nhập khẩu, làm cho thông tin được xem xét trong việc tính toán độ tương tự phong phú hơn và cải thiện độ chính xác phát hiện.
* Nhiệm vụ huấn luyện. Để đánh giá đóng góp của ba nhiệm vụ huấn luyện, chúng tôi cũng đánh giá VulHawk với các cài đặt huấn luyện khác nhau. Để đo lường rõ ràng hơn đóng góp của các nhiệm vụ huấn luyện, các mô hình đánh giá không sử dụng các bộ chuyển đổi dựa trên entropy và hiệu chuẩn độ tương tự, chỉ khác nhau về nhiệm vụ huấn luyện. Trong tác vụ so sánh một-nhiều XC+XO+XA, mô hình được huấn luyện bởi nhiệm vụ MLM đạt AUC là 0.833, mô hình được huấn luyện bởi nhiệm vụ MLM+ROP đạt AUC là 0.934 và mô hình được huấn luyện bởi nhiệm vụ MLM+ROP+ABP đạt AUC là 0.966. Nhiệm vụ huấn luyện ROP giúp mô hình học cú pháp của token gốc, giúp sử dụng cú pháp của token gốc để thay thế các toán hạng OOV để giảm thiểu các vấn đề OOV. Nhiệm vụ huấn luyện ABP giúp mô hình học các mối quan hệ thứ tự của mã nhúng. Cả hai đều cải thiện hiệu suất của mô hình để phân biệt độ tương tự hàm. Trả lời RQ.5: Bộ chuyển đổi dựa trên entropy, hiệu chuẩn độ tương tự và các nhiệm vụ huấn luyện ROP và ABP đóng góp tích cực vào hiệu suất của VulHawk.

G. Xác định Môi trường File

* Chúng tôi cũng đánh giá độ chính xác của việc xác định môi trường file dựa trên thông tin entropy. Để thực hiện đánh giá này, chúng tôi sử dụng phương pháp chia dữ liệu thành 10 phần bằng kỹ thuật kiểm tra chéo (10-fold cross-validation), giống như trong các thiết lập học máy truyền thống. Các tập tin nhị phân được sử dụng trong đánh giá này đến từ các kiến trúc khác nhau (x86, arm và mips) và các trình biên dịch khác nhau (GCC và Clang). Pizzolotto et al. [44] đã sử dụng các mô hình CNN và LSTM trên dữ liệu byte của hàm để xác định môi trường file.
* Để minh chứng cho hiệu suất của phương pháp của chúng tôi, chúng tôi đã tải xuống các mô hình đã được huấn luyện trước từ nghiên cứu của Pizzolotto et al. và sử dụng chúng (CNN và LSTM) như các mô hình so sánh. Lưu ý rằng, trình biên dịch và cấp độ tối ưu hóa của các tập tin nhị phân cụ thể là không xác định và có thể thay đổi trong thực tế, vì vậy chúng tôi không cố định các thông số khác (ví dụ như kiến trúc và cấp độ tối ưu hóa) khi đánh giá một thông số cụ thể (ví dụ như trình biên dịch) để đảm bảo tính thực tế và liên quan đến các tình huống thực tế.

A graph of a graph of a number of numbers and a number of numbers

Description automatically generated

* Hình 11 trình bày xu hướng độ chính xác của VulHawk và các phương pháp khác theo thời gian. Có các quan sát sau từ kết quả: (1) VulHawk vượt trội hơn CNN và LSTM trong việc xác định các trình biên dịch và cấp độ tối ưu hóa. (2) Tốc độ xác định của VulHawk nhanh hơn so với CNN và LSTM. (3) Đường cong độ chính xác của VulHawk ổn định, trong khi đó các đường cong của CNN và LSTM dao động. Điều này cho thấy VulHawk tổng quát hơn trong nhiều tình huống khác nhau, trong khi CNN và LSTM bị hạn chế trong một số trường hợp. Chúng tôi tiến hành phân tích chi tiết như sau. So với CNN và LSTM, VulHawk sử dụng luồng entropy bao gồm thông tin toàn cục của các tập tin nhị phân, thay vì chỉ dữ liệu byte gốc của một hàm duy nhất, điều này làm cho sự sai lệch tình cờ của các hàm không ảnh hưởng nhiều đến VulHawk. Entropy giữ cho sự phân biệt rõ ràng giữa các môi trường file khác nhau, trong khi sự khác biệt trong dữ liệu byte gốc không rõ ràng. Điều này giúp VulHawk duy trì khả năng tổng quát cao và vượt trội hơn CNN và LSTM. Ngoài ra, VulHawk không chỉ vượt trội hơn CNN và LSTM về độ chính xác mà còn về hiệu suất. Bởi vì VulHawk sử dụng một đặc trưng mạnh mẽ (entropy) và mô hình của chúng tôi nhẹ hơn nhiều so với CNN và LSTM, ví dụ như kích thước tham số trong mô hình của chúng tôi chỉ chiếm 1.3% so với CNN.
* Để đánh giá toàn diện việc nhận dạng môi trường file dựa trên entropy, chúng tôi tiến hành thực nghiệm trong các tình huống khác nhau với các kiến trúc khác nhau (x86, mips và arm), kích thước file khác nhau (kích thước nhỏ và kích thước lớn) và các loại file khác nhau (thư viện và các tập tin thực thi). Ở đây, với phân phối kích thước file, chúng tôi xem xét các kích thước file nhỏ hơn 1024 KB là các tập tin nhỏ và ngược lại là các tập tin lớn. Hình 12 trình bày các kết quả, và mỗi khối được chú thích với tỷ lệ phân loại. Ví dụ, trong dự đoán arm-O3, có 95.4% được phân loại đúng là O3 và 4.6% được phân loại là O2, tỷ lệ chính xác tổng thể là 96.2%. Tùy chọn O0 đạt được độ chính xác cao, bởi vì O0 sử dụng các tối ưu hóa mặc định để giảm thời gian biên dịch, điều này khác biệt đáng kể so với các tùy chọn khác. Nhận dạng của O2 và O3 trong các tập tin mips và thư viện đạt được kết quả tồi nhất, trong đó hầu hết sự nhận dạng sai lệch tập trung vào sự phân biệt giữa O2 và O3. So với O0, tùy chọn O3 thêm hơn 285 tối ưu hóa, trong khi O3 chỉ có 3 tối ưu hóa hơn O2. Do đó, các tập tin từ O2 và O3 có cấu trúc rất tương tự nhau. Vì các tập tin từ O2 và O3 có cấu trúc rất tương tự nhau, các sự nhận dạng sai lệch này là chấp nhận được và không ảnh hưởng nhiều đến việc tìm kiếm hàm nhị phân. Khi xem xét O2 và O3 cùng nhau, độ chính xác của việc phân biệt chúng so với O0 và O1 là trên 90% trong tất cả các tình huống. Kết quả cho thấy việc nhận dạng môi trường file dựa trên entropy của chúng tôi mạnh mẽ trong các tình huống khác nhau với các kiến trúc khác nhau, kích thước file và loại file.
* A screenshot of a computer screen

  Description automatically generated

A table with numbers and a number on it

Description automatically generated

H. Phát hiện lỗ hổng trong 1 ngày từ Firmware

* Trong thí nghiệm này, chúng tôi thu thập 20 hình ảnh firmware IoT mới nhất từ ba nhà cung cấp (D-Link, TP-Link và NetGear) và thực hiện VulHawk và các phương pháp cơ sở khác trong nhiệm vụ phát hiện lỗ hổng trong 1 ngày. Các dự án OpenSSL và Curl được sử dụng rộng rãi trong firmware IoT, vì vậy chúng tôi chọn chúng làm mục tiêu và xây dựng một kho lỗ hổng dựa trên Cơ sở dữ liệu lỗ hổng và Tiết lộ (CVE). Kho lưu trữ chứa các hàm có lỗ hổng và hàm đã được vá lỗi của 12 CVE liên quan, trong đó chi tiết và sự thật được hiển thị trong Bảng VI. Tổng cộng, có 53.739 hàm, bao gồm 93 hàm có lỗ hổng liên quan và 119 hàm đã được vá lỗi liên quan.
* Đối với mỗi hàm có lỗ hổng/đã được vá lỗi, chúng tôi sử dụng VulHawk để tạo ra các nhúng hàm của chúng và ghi lại các đặc trưng tinh vi của chúng cho việc hiệu chỉnh tương đồng. Trong giai đoạn phát hiện lỗ hổng, chúng tôi sử dụng tất cả các hàm trong các thư viện firmware làm truy vấn hàm và thực hiện một tìm kiếm một-nhiều trong kho lưu trữ lỗ hổng đã xây dựng.

A graph of a function

Description automatically generated

* Bảng VI hiển thị kết quả của VulHawk và các giá trị cơ sở khác với ngưỡng tốt nhất của chúng theo hình 13. Đối với 12 CVEs, Trex không có sai sót nhưng chỉ đạt được 52.7% nhận dạng; GMN đạt được 86.0% nhận dạng nhưng lại gây ra 36,342 sai sót; Asteria đạt được độ chính xác 64.1% với không có sai sót âm thầm; trong khi VulHawk đạt được hiệu suất tốt nhất với không có sai sót dương tính và 100% nhận dạng. Hơn nữa, chúng tôi thấy được các quan sát sau đây. (1) VulHawk vượt trội hơn các giá trị cơ sở trong việc phát hiện lỗ hổng trong 1 ngày với sự nhận dạng và chính xác cao, điều này giúp giảm gánh nặng cho nhà nghiên cứu khi phát hiện các lỗ hổng đã biết trong các tập tin nhị phân chưa biết. (2) Tác động của lỗ hổng thư viện bên thứ ba trong firmware IoT vẫn còn nghiêm trọng, và các lỗ hổng đã biết trong hình ảnh firmware vẫn chưa được khắc phục kịp thời. Ngay cả firmware mới nhất vẫn chứa các lỗ hổng từ 7 năm trước. (3) Hình ảnh firmware từ các mô hình liền kề của cùng một nhà cung cấp thông thường có các chức năng có lỗ hổng tương tự nhau (xem Phụ lục B). (4) Việc phát hiện lỗ hổng chỉ thông qua các phiên bản thư viện bên thứ ba sẽ dẫn đến các sai sót dương tính, vì thư viện bên thứ ba có thể không bao gồm các chức năng có lỗ hổng cụ thể do thiếu các mô-đun tương ứng. Ví dụ, TP-LINK Deco M4 sử dụng OpenSSL 1.0.2d, đó là một phiên bản bị ảnh hưởng của CVE-2016-6303, nhưng nó không bị ảnh hưởng bởi lỗ hổng này vì nó không có mô-đun tương ứng.
* Phân tích Sai Positve. Chúng tôi thay đổi ngưỡng để phân tích tác động của các ngưỡng khác nhau đối với việc phát hiện lỗ hổng trong 1 ngày. Hình 13 cho thấy các đường cong của độ nhớ lại/độ chính xác với các ngưỡng khác nhau. Khi ngưỡng tăng lên, độ nhớ lại của VulHawk giảm nhẹ, trong khi độ chính xác tăng đáng kể. Để hỗ trợ các nhiệm vụ XC, XO và XA, SAFE, GMN và Asteria thiết kế mô hình của họ để chống lại các thay đổi nhỏ của các chức năng, điều này dẫn đến khó khăn trong việc phân biệt các chức năng khác nhau và gây ra các sai positve. Trex sử dụng các đặc trưng tinh vi dưới dạng các micro-traces để phát hiện các chức năng có lỗ hổng, điều này đạt được độ chính xác cao nhưng chịu thiệt hại do các sai negatves. Chúng tôi đề xuất một chiến lược tìm kiếm tiến bộ kết hợp tìm kiếm cấu trúc và hiệu chuẩn độ tương đồng tinh vi, cho phép VulHawk đạt được độ nhớ lại cao trong khi vẫn duy trì độ chính xác cao. Ví dụ, khi đặt ngưỡng là 0.9, VulHawk đạt 100% độ nhớ lại với không có sai positve, vượt qua các tiêu chuẩn cơ sở khác. Chúng tôi giới thiệu VulHawk-S, không sử dụng hiệu chuẩn độ tương đồng. So sánh với VulHawk-S, VulHawk thể hiện độ chính xác cao hơn ở cùng ngưỡng trong việc phát hiện lỗ hổng trong 1 ngày. Vì nhúng có độ dài cố định được thiết kế để chống lại các thay đổi nhỏ do các môi trường tệp khác nhau, làm cho việc phát hiện lỗ hổng chính xác khá thô sơ. Chúng tôi đề xuất hiệu chuẩn độ tương đồng để bổ sung các nhúng chức năng bằng các đặc trưng tinh vi để tăng độ chính xác phát hiện. Các chức năng đã được vá với các thay đổi nhỏ có thể gây ra các sai positve khi ngưỡng được đặt nhỏ. Ví dụ, chức năng sub\_20cc8 trong libcurl.so của NetGear RBR20 là một chức năng đã được vá của CVE-2016-8618. Vì bản vá này là một thay đổi nhỏ và không sửa đổi bất kỳ giá trị nào, VulHawk sẽ nhận nhầm sub\_20cc8 là một chức năng có lỗ hổng khi ngưỡng nhỏ hơn 0.85. May mắn thay, khi đặt ngưỡng của VulHawk là 0.9, các sai positve này giảm đáng kể với độ nhớ lại 100%.
* Hơn nữa, chúng tôi sử dụng VulHawk để thực hiện phát hiện lỗ hổng trong các dự án mới nhất và phát hiện một lỗ hổng nghi ngờ về tràn ngăn xếp trong OpenSSL-3.1.0/ssl/ssl\_lib.c, đã được vá lại nhờ báo cáo của chúng tôi. Điều này cho thấy khả năng của VulHawk trong phát hiện các lỗ hổng mới. Các chi tiết được hiển thị trong Phụ lục B. Cuối cùng, chúng tôi đã an toàn báo cáo các lỗ hổng đã phát hiện trong các thí nghiệm của chúng tôi cho các nhà cung cấp và nhà phát triển tương ứng. Trả lời cho RQ.6: VulHawk có thể phân biệt các chức năng có lỗ hổng và các chức năng đã được vá và phát hiện các lỗ hổng 1 ngày với hiệu suất cao hơn so với các phương pháp hiện đại trong thế giới thực. Ở phần này, chúng tôi thảo luận về chiến lược chia và trị, tính chính xác của IRFM và nghiên cứu tương lai.

5. Thảo luận:

* Trong phần này, chúng tôi thảo luận về chiến lược chia và trị, tính chính xác của IRFM và nghiên cứu tương lai.
* Từ kết quả, các phương pháp cơ sở này khó đạt được hiệu suất cao trong tất cả ba nhiệm vụ của trình biên dịch chéo, cấp tối ưu hóa chéo và kiến trúc chéo. Điều này bởi vì các khác biệt được giới thiệu bởi các trình biên dịch, kiến trúc và cấp tối ưu hóa là khác nhau. Do đó, việc xây dựng một mô hình mạnh mẽ chống lại các kiến trúc, trình biên dịch và cấp tối ưu hóa cùng một lúc là một thách thức. Trong khi VulHawk sử dụng bộ chuyển đổi dựa trên entropy để thực hiện chiến lược chia và trị, nó chia bài toán tính toán độ tương đồng của các kịch bản môi trường tệp C2N chéo thành N - 1 bài toán con chuyển nhúng. Bằng cách chuyển nhúng các nhúng chức năng từ các môi trường tệp khác nhau đến cùng một môi trường tệp, VulHawk có thể giảm bớt sự khác biệt do các trình biên dịch, kiến trúc và tối ưu hóa gây ra. Có thể thấy rằng, sự khác biệt lớn hơn do các môi trường tệp gây ra, thì cải thiện do bộ chuyển đổi dựa trên entropy càng rõ rệt hơn (ví dụ, nhiệm vụ XC trong Bảng IV). Do đó, chiến lược chia và trị rất hiệu quả trong kịch bản tìm kiếm firmware IoT phức tạp.
* IRFM (Instruction-level Robust Function Matching) nhằm tạo ra các nhúng cơ bản và nhúng chức năng mạnh mẽ, là quan trọng cho việc phát hiện và tìm kiếm tiếp theo. Ở đây, chúng tôi thảo luận về tính chính xác của nó:
* (1) Trong VulHawk, chúng tôi sử dụng microcode, một biểu diễn trung gian đã được thiết lập tốt, để giảm thiểu sự khác biệt do các kiến trúc bộ lệnh gây ra.
* (2) Để tăng trọng số cho các ngữ nghĩa chính, chúng tôi đơn giản hóa các hướng dẫn dư thừa và mờ hóa, trong đó các hướng dẫn cần được loại bỏ được lọc một cách nghiêm ngặt để tránh xóa tình cờ các giá trị trả về, biến toàn cục và đối số cho các hàm phụ (xem Phần III-A2). Kết quả thí nghiệm cho thấy việc đơn giản hóa hướng dẫn làm cho khoảng cách giữa các khối tương tự gần hơn và đưa các khối không tương tự xa hơn (Phụ lục A). Ví dụ, khoảng cách trung bình giữa các khối tương tự giảm đi 44,4%, và khoảng cách trung bình giữa các khối không tương tự tăng lên 2,6% trong nhiệm vụ XC.
* (3) Áp dụng các kỹ thuật xử lý ngôn ngữ tự nhiên vào tìm kiếm mã nhị phân có thể tự động tạo ra các nhúng ngữ nghĩa của chuỗi hướng dẫn. Mặc dù chúng tôi không phải là người đầu tiên áp dụng NLP vào tìm kiếm mã nhị phân [10], [35], [60], nhưng chúng tôi là người đầu tiên tinh chỉnh các chuỗi hướng dẫn để bảo tồn ngữ nghĩa chính trong các phương pháp tiếp cận BCSD tĩnh. Với các đặc điểm của các chức năng nhị phân, chúng tôi tùy chỉnh mô hình ngôn ngữ và đề xuất hai nhiệm vụ tiền huấn luyện dựa trên các đặc điểm mã nhị phân, làm cho mô hình của chúng tôi mạnh mẽ hơn.
* (4) Công trước đó [34] đã chứng minh rằng GNN (Graph Neural Network) dựa trên CFG (Control Flow Graph) là cách hiệu quả để giải quyết vấn đề tương tự chức năng nhị phân. Để tạo ra các nhúng chức năng, chúng tôi sử dụng GCN (Graph Convolutional Network) để tổng hợp ngữ nghĩa của các khối cơ bản và tích hợp tất cả các nhúng khối cơ bản dựa trên CFGs, làm cho các nhúng chức năng được tạo ra bao gồm các đặc trưng cấu trúc CFG và ngữ nghĩa của các khối cơ bản.
* Trong phần IV-H, chúng tôi phát hiện các thư viện đã lỗi thời được sử dụng trong hình ảnh firmware. Cập nhật trực tiếp những thư viện đã lỗi thời này lên phiên bản mới nhất có thể làm hỏng các phụ thuộc tệp trong firmware. VulHawk có thể phát hiện các chức năng có lỗ hổng đã biết trước. Hơn nữa, chúng tôi có thể tạo các bản vá nhanh ở mức nhị phân để ngăn chặn những lỗ hổng đã biết trước này. Chúng tôi để lại công việc này cho tương lai.

6. RELATED WORK:

* Trong phần này, chúng tôi tóm tắt một số nghiên cứu liên quan bổ sung. Chúng tôi tập trung vào các phương pháp sử dụng tương đồng mã để tìm kiếm mã và phát hiện các lỗ hổng đã biết trước mà không cần mã nguồn.
* Phương pháp Mono-architecture. Tìm kiếm mã nhị phân cho các nhị phân mono-architecture đã đạt được nhiều thành công. TEDEM [43] giới thiệu khoảng cách chỉnh sửa cây để đo độ tương đồng mã ở cấp độ các khối cơ bản. BinHunt [15] và CoP [32] so sánh sự tương đồng mã nhị phân bằng cách sử dụng thực thi biểu tượng và bằng chứng toán học. Tuy nhiên, những phương pháp này tốn nhiều tài nguyên tính toán và do đó không áp dụng được cho kho lưu trữ chức năng lớn. Tracelet [7] phân tách các chức năng nhị phân thành các dấu vết liên tục và sử dụng khoảng cách chỉnh sửa giữa hai dấu vết để đo độ tương đồng của chúng. Các phương pháp khác [20], [23], [37], [38] sử dụng các đặc trưng thống kê, đặc trưng cú pháp và đặc trưng cấu trúc để phù hợp với các chức năng nhị phân tương tự. Cảm hứng từ các kỹ thuật NLP, nhiều nhà nghiên cứu [10], [11], [27] giới thiệu các mô hình ngôn ngữ để trích xuất ngữ nghĩa của mã nhị phân. DeepBinDiff [11] xem các hướng dẫn hợp ngữ như là các từ và sử dụng mô hình Word2Vec CBOW [36] để tạo ra các nhúng ngữ nghĩa cho mã nhị phân. Asm2Vec xem các chức năng như là các tài liệu, và các mã thông báo (mã opcode hoặc toán hạng) như là các từ trong tài liệu, và sử dụng mô hình Distributed Memory Model of Paragraph Vectors (PV-DM) [26] để tạo ra các nhúng chức năng. Chúng biểu diễn các chức năng nhị phân dưới dạng các vectơ số học chiều cao, giúp tìm kiếm các chức năng ứng viên tương tự trong kho lưu trữ chức năng lớn. Tuy nhiên, đối với các hình ảnh firmware IoT từ các kiến trúc khác nhau, các phương pháp tìm kiếm mã nhị phân cần hỗ trợ tìm kiếm các chức năng tương tự qua các kiến trúc.
* Các phương pháp Cross-architecture. Gần đây, các nhà nghiên cứu đã bắt đầu đối mặt với thách thức tìm kiếm mã nhị phân chéo kiến trúc. Các phương pháp truyền thống chọn các đặc trưng bền vững kiến trúc từ các đặc trưng thống kê, đặc trưng cú pháp và đặc trưng cấu trúc để tính toán độ tương đồng của mã nhị phân. BinDiff [61], là công cụ phát hiện sự tương đồng mã nhị phân thương mại hiện đại, trích xuất số lượng các khối cơ bản, tham chiếu chuỗi và cấu trúc của đồ thị gọi để tính toán độ tương đồng giữa các chức năng nhị phân. DiscovRE [12] sử dụng phù hợp dựa trên CFG để tìm các chức năng tương tự, nhưng phù hợp đồ thị mang lại tính toán đắt đỏ. Esh [5] sử dụng bộ giải SMT dựa trên các lát cắt luồng dữ liệu của các khối cơ bản để xác minh tính tương tự chức năng, điều này không hỗ trợ kho chức năng lớn. Genius [14], giới thiệu học máy, xem các đặc trưng thống kê như các thuộc tính của CFG để nhúng đồ thị cho tìm kiếm mã nhị phân. Gemini [55] cải tiến Genius bằng cách sử dụng mạng neural để tạo ra các nhúng chức năng để tìm các chức năng tương tự. Cả Gemini và Genius đều sử dụng các đặc trưng được tạo bằng tay, yêu cầu kinh nghiệm phong phú và kiến thức chuyên gia để phù hợp với các chức năng tương tự. InnerEye [60] xem các nhị phân từ các kiến trúc khác nhau như các ngôn ngữ khác nhau và sử dụng mô hình dựa trên dịch máy neural Word2Vec [36] để tính toán độ tương đồng mã nhị phân qua các kiến trúc cụ thể (x86 và arm). SAFE [35] đào tạo mô hình ngôn ngữ của mình bằng cách sử dụng các nhị phân từ nhiều kiến trúc để tìm kiếm mã nhị phân qua các kiến trúc. Tuy nhiên, SAFE gặp vấn đề ngoài vựng đáng kể trong thực tế, do đó hiệu suất của nó trong các nhiệm vụ chéo kiến trúc kém. Trex [41] đề xuất các micro-traces bao gồm hướng dẫn và giá trị động, và học ngữ nghĩa thực thi chức năng từ các micro-traces của chúng để phù hợp với chức năng tương tự chéo kiến trúc.
* Chức năng của các hình ảnh firmware IoT từ các kiến trúc khác nhau được tạo ra bởi các trình biên dịch khác nhau với các cấp tối ưu hóa khác nhau tùy theo yêu cầu sử dụng khác nhau. Có nhiều sự khác biệt giữa các mã nhị phân này do các trình biên dịch và cấp tối ưu hóa khác nhau gây ra. Các phương pháp hiện có sử dụng một mô hình được đào tạo cố định có thể hoạt động cho các cấp tối ưu hóa và trình biên dịch cụ thể, nhưng vẫn khó đạt được hiệu suất tốt đối với các trình biên dịch và cấp tối ưu hóa khác nhau.

VII. KẾT LUẬN

* Trong bài báo này, chúng tôi đề xuất một phương pháp tìm kiếm mã nhị phân chéo kiến trúc mang tên VulHawk. Nó bao gồm một mô hình chức năng biểu diễn trung gian dựa trên RoBERTa và GCNs để tạo ra các nhúng chức năng. Để đảm bảo tính bền vững trước các môi trường tệp khác nhau, chúng tôi đề xuất một chiến lược chia và trị và giới thiệu thông tin entropy từ quan điểm thông tin học để xác định các môi trường tệp của mã nhị phân. Chúng tôi đề xuất một bộ chuyển đổi dựa trên entropy để chuyển nhúng chức năng vào cùng một môi trường tệp trung gian để giảm bớt sự khác biệt do các trình biên dịch, tối ưu hóa và kiến trúc khác nhau gây ra. Trong chiến lược tìm kiếm tiến hóa, việc hiệu chuẩn tương đồng bổ sung phát hiện lỗ hổng bằng cách sử dụng các đặc trưng cấp độ tinh vi để giảm thiểu các dương tính sai do các chức năng đã được vá lỗi. Các đánh giá rộng rãi cho thấy VulHawk vượt trội hơn so với các phương pháp tiên tiến trong bảy nhiệm vụ. Chiến lược chia và trị cải thiện hiệu quả tính bền vững của VulHawk đối mặt với các trình biên dịch, cấp tối ưu hóa và kiến trúc. Thử nghiệm phát hiện lỗ hổng trong 1 ngày cho thấy hiệu suất cao của VulHawk trong việc phát hiện các lỗ hổng.