Asteria-Pro: Enhancing Deep-Learning Based Binary Code Similarity Detection by Incorporating Domain Knowledge.

Asteria-Pro: Nâng cao khả năng phát hiện sự tương đồng mã nhị phân dựa trên học sâu bằng cách tích hợp kiến thức lĩnh vực.

1. Giới thiệu:

* Sự tái sử dụng mã rất phổ biến trong firmware IoT để tăng cường quá trình phát triển [64]. Thật không may, việc tái sử dụng mã cũng mang vào nhiều firmware các lỗ hổng ẩn chứa trong mã gốc [22]. Sự an toàn và riêng tư của cuộc sống của chúng ta đang bị đe dọa nghiêm trọng bởi việc sử dụng rộng rãi các firmware này [65]. Ngay cả khi các lỗ hổng đã được công khai, vẫn còn rất nhiều phiên bản firmware chứa chúng do việc nâng cấp mã bị trì hoãn hoặc gặp vấn đề về tương thích mã [18]. Các lỗ hổng tái diễn, thường được gọi là "lỗ hổng N-day", không thể được phát hiện thông qua thông tin biểu tượng như tên chức năng vì loại thông tin này thường bị gỡ bỏ trong quá trình biên dịch firmware. Hơn nữa, mã nguồn của firmware thường không có sẵn vì các nhà cung cấp IoT chỉ cung cấp phiên bản nhị phân của firmware của họ.
* Vì vậy, việc áp dụng phát hiện sự tương đồng mã nhị phân (BCSD) được thực hiện để nhanh chóng tìm ra các lỗ hổng tương đồng trong một lượng lớn firmware [23]. Kỹ thuật BCSD tập trung vào xác định sự tương đồng giữa hai phần mã nhị phân. Đối với việc tìm kiếm lỗ hổng, BCSD tìm các chức năng có lỗ hổng khác tương tự với một chức năng đã được biết có lỗ hổng. Ngoài việc tìm kiếm lỗ hổng, BCSD đã được sử dụng rộng rãi cho các ứng dụng bảo mật khác như phát hiện sao chép mã [16, 48, 57], phát hiện phần mềm độc hại [41, 42], và phân tích bản vá [28, 34, 63]. Mặc dù có nhiều nỗ lực nghiên cứu đã được thực hiện, sự đa dạng của kiến trúc phần cứng IoT và các nền tảng phần mềm đặt ra những thách thức cho BCSD đối với firmware IoT. Có nhiều kiến trúc bộ chỉ thị khác nhau (ISA) cho firmware IoT, chẳng hạn như ARM, PowerPC, X64 và X86. Các chỉ thị là khác nhau, và các quy tắc, chẳng hạn như giao thức gọi và bố cục ngăn xếp, cũng khác nhau giữa các ISA khác nhau. Việc tìm các chức năng có lỗ hổng tương tự qua các kiến trúc khác nhau là rất phức tạp.
* Các phương pháp BCSD (phát hiện sự tương đồng mã nhị phân) có thể được phân loại chung thành hai loại: i) phương pháp dựa trên phân tích động và ii) phương pháp dựa trên phân tích tĩnh. Các phương pháp dựa trên phân tích động thu thập hành vi chạy thời gian như tính năng ý nghĩa của chức năng bằng cách chạy các chức năng mục tiêu, trong đó các tính năng của chức năng có thể là cặp đầu vào/đầu ra của chức năng [55] hoặc các cuộc gọi hệ thống trong quá trình thực thi chương trình [29], v.v. Tuy nhiên, chúng không thích hợp cho phân tích firmware quy mô lớn vì yêu cầu chạy firmware đòi hỏi các thiết bị cụ thể và mô phỏng firmware cũng khó khăn [20, 35, 73].
* Các phương pháp dựa trên phân tích tĩnh chủ yếu trích xuất các tính năng thống kê từ mã hợp ngữ. Một cách trực quan là tính toán khoảng cách chỉnh sửa giữa các chuỗi mã hợp ngữ [24]. Chúng không thể được áp dụng trực tiếp trên các kiến trúc vì các bộ chỉ thị hoàn toàn khác biệt. Các tính năng thống kê độc lập kiến trúc của các chức năng được đề xuất cho việc phát hiện sự tương đồng [31]. Những tính năng này ít bị ảnh hưởng qua các kiến trúc như số lượng cuộc gọi hàm, chuỗi và hằng số. Hơn nữa, sử dụng đồ thị dòng điều khiển (CFG) ở cấp mã hợp ngữ thông qua việc thực hiện so sánh đồ thị đồng cấu đã được sử dụng để cải thiện việc phát hiện sự tương đồng [31, 33]. Dựa trên các tính năng thống kê và CFG, Gemini [66] sử dụng mạng nhúng đồ thị để mã hóa các chức năng thành các vectơ để phát hiện sự tương đồng. Với việc áp dụng các mô hình học sâu trong phân tích ngôn ngữ lập trình, gần đây đã xuất hiện các phương pháp khác nhau để sử dụng các mô hình này để mã hóa các chức năng nhị phân dưới các hình thức khác nhau và tính toán sự tương đồng chức năng dựa trên mã hóa chức năng [46, 50, 54, 62].
* Các phương pháp dựa trên phân tích tĩnh nhanh hơn và có khả năng mở rộng hơn cho phân tích firmware quy mô lớn, nhưng thường sản sinh kết quả dương giả do thiếu thông tin ý nghĩa. Vì các chức năng có lỗ hổng tương tự nhau trong các kiến trúc khác nhau thường có cùng ý nghĩa, nên BCSD trên các kiến trúc giao cắt nhau phải có khả năng thu thập thông tin ý nghĩa về các chức năng một cách có thể mở rộng.
* Trong công trình trước đó của chúng tôi, Asteria [71], chúng tôi đã sử dụng mạng Tree-LSTM để mã hóa cây cú pháp trừu tượng (AST) để bắt lấy biểu diễn ý nghĩa của nó. Cụ thể, Tree-LSTM được huấn luyện bằng cấu trúc siamese [37] để hiểu biểu diễn ý nghĩa bằng cách đưa các cặp chức năng homologous và non-homologous vào mạng Tree-LSTM. Do đó, mạng Tree-LSTM học biểu diễn ý nghĩa chức năng để phân biệt giữa các chức năng homologous và non-homologous. Để cải thiện độ chính xác, chúng tôi cũng sử dụng đồ thị gọi (call graph) để hiệu chỉnh sự tương đồng của AST. Cụ thể, chúng tôi đếm các chức năng được gọi của các chức năng mục tiêu trong đồ thị gọi để đo sự khác biệt trong các cuộc gọi chức năng. Sự tương đồng chức năng cuối cùng được xác định bằng cách hiệu chỉnh sự tương đồng của AST với sự khác biệt trong các cuộc gọi chức năng. Trong đánh giá trước đây của chúng tôi, Asteria vượt trội hơn so với các phương pháp tiên tiến có sẵn, như Gemini và Diaphora, về mặt độ chính xác. Kết quả đánh giá cho thấy sự ưu việt của việc trích xuất ý nghĩa chức năng bằng cách mã hóa AST với mô hình Tree-LSTM. Tuy nhiên, mã hóa AST đòi hỏi một chi phí thời gian rõ ràng đối với Asteria. Theo nghiên cứu trước đó của chúng tôi [71], quá trình mã hóa toàn bộ AST mất khoảng một giây. Khi áp dụng Asteria cho phát hiện lỗ hổng, nơi có nhiều chức năng để thực hiện tính toán tương đồng cho một chức năng có lỗ hổng, chi phí thời gian trở nên không chấp nhận được. Vì hầu hết các chức năng ứng cử viên không tương tự, có khả năng cải thiện hiệu suất của Asteria. Nói cách khác, các chức năng không tương tự nhau khác biệt với các chức năng có lỗ hổng trong một số đặc điểm mà chúng ta có thể khai thác để bỏ qua phần lớn các chức năng không tương tự một cách hiệu quả hơn. Ngoài ra, các đánh giá không phù hợp với các phương pháp được sử dụng trong phần lớn các nỗ lực phát hiện lỗ hổng thực tế [33, 45, 52, 66, 74], bao gồm nghiên cứu trước đó của chúng tôi Asteria. Phát hiện lỗ hổng bao gồm việc lấy các chức năng homologous (có lỗ hổng) từ một lượng lớn chức năng. Do đó, hiệu suất của Asteria trong việc phát hiện lỗ hổng chưa được miêu tả đầy đủ. Cần thiết để đánh giá hiệu suất của Asteria trên nhiệm vụ tìm kiếm lỗ hổng. Hơn nữa, theo kết quả trong việc phát hiện lỗ hổng thực tế [71], Asteria chịu sự ảnh hưởng của nhiều dương tính sai, điều này ảnh hưởng đến hiệu quả của nó trong thực tế.
* Có hai thách thức chính làm cản trở Asteria trở nên thực tế cho việc phát hiện lỗ hổng quy mô lớn:
* Thách thức 1 (C1): Việc lọc ra phần lớn các chức năng không tương tự trước khi mã hóa AST, đồng thời giữ lại các chức năng homologous để tăng tốc quá trình phát hiện lỗ hổng là một thách thức khó khăn.
* Thách thức 2 (C2): Việc phân biệt các chức năng tương tự nhưng không homologous là một thách thức. Mặc dù Asteria có độ chính xác cao trong việc phân loại homologous và non-homologous, nó vẫn tạo ra các kết quả dương tính sai khi phân biệt các chức năng có AST tương tự nhau.
* Chúng tôi đã thiết kế Asteria-Pro bằng cách giới thiệu kiến thức lĩnh vực thành hai câu trả lời, A1 và A2, để vượt qua hai thách thức này. Khái niệm cơ bản của chúng tôi là giới thiệu kiến thức lĩnh vực giữa các chức năng sẽ giúp Asteria-Pro đạt được độ chính xác cao hơn khi kết hợp với mô hình học sâu đã học được kiến thức ý nghĩa bên trong các chức năng. Asteria-Pro bao gồm ba mô-đun: 1) mô-đun tiền lọc (DK-based pre-filtration) dựa trên kiến thức lĩnh vực, 2) mô-đun phát hiện sự tương đồng dựa trên học sâu (DL-based similarity detection), và 3) mô-đun tái xếp hạng dựa trên kiến thức lĩnh vực (DK-based re-ranking), trong đó mô-đun phát hiện sự tương đồng dựa trên học sâu (DL-based similarity detection) về cơ bản dựa trên Asteria. Kiến thức lĩnh vực được khai thác đầy đủ cho các mục đích khác nhau trong mô-đun tiền lọc và tái xếp hạng dựa trên kiến thức lĩnh vực. Trong mô-đun tiền lọc, Asteria-Pro nhằm bỏ qua càng nhiều chức năng không tương tự càng tốt bằng cách so sánh các tính năng nhẹ nhàng và đáng tin cậy (A1). Đồng thời, việc lọc cần được thực hiện để giữ lại tất cả các chức năng homologous. Để đạt được điều này, chúng tôi đã tiến hành một nghiên cứu sơ bộ về hiệu suất lọc của một số tính năng chức năng nhẹ. Dựa trên kết quả của nghiên cứu, chúng tôi đề xuất một thuật toán mới thành công sử dụng ba tính năng chức năng khác nhau trong quá trình lọc. Trong mô-đun tái xếp hạng, Asteria-Pro xác nhận sự homologous của các chức năng bằng cách so sánh các mối quan hệ gọi (A2), dựa trên giả thiết rằng các chức năng được thiết kế cho các mục đích khác nhau có các mối quan hệ gọi khác nhau.
* Kết quả đánh giá của chúng tôi cho thấy Asteria-Pro vượt trội đáng kể so với các phương pháp tiên tiến hiện có về cả độ chính xác và hiệu quả. So với Asteria, Asteria-Pro đã thành công cắt giảm thời gian phát hiện của Asteria đến 96,90% bằng cách tích hợp mô-đun tiền lọc dựa trên kiến thức lĩnh vực. Trong nhiệm vụ tìm kiếm lỗ hổng, Asteria-Pro có thời gian tìm kiếm trung bình ngắn hơn so với các phương pháp cơ sở khác. Bằng cách tích hợp tái xếp hạng dựa trên kiến thức lĩnh vực, Asteria-Pro đã cải thiện MRR và Recall@Top-1 lần lượt là 23,71% và 36,4%, đạt 90,8% và 89,6%. Chúng tôi cũng đã áp dụng khung nâng cao của chúng tôi để nhúng các phương pháp cơ sở, và kết quả đánh giá cho thấy sự cải thiện đáng kể về độ chính xác của các phương pháp này. Asteria-Pro đã xác định được 1.482 chức năng có lỗ hổng với độ chính xác cao lên đến 91,65% thông qua việc thực hiện phát hiện lỗ hổng firmware thực tế quy mô lớn sử dụng 90 CVEs. Hơn nữa, kết quả phát hiện của CVE-2017-13001 cho thấy rằng Asteria-Pro có khả năng nâng cao trong việc phát hiện mã có lỗ hổng đã được nhúng.
* Các đóng góp của chúng tôi được tóm tắt như sau:
* Chúng tôi tiến hành một nghiên cứu sơ bộ để chứng minh tính hiệu quả của các tính năng chức năng đơn giản khác nhau trong việc xác định các chức năng không tương tự nhau.
* Đây là công trình đầu tiên đề xuất tích hợp kiến thức lĩnh vực trước và sau các mô hình học sâu để tối ưu hóa phát hiện lỗ hổng. Chúng tôi thực hiện các thuật toán tiền lọc dựa trên kiến thức lĩnh vực và tái xếp hạng và trang bị chúng vào Asteria.
* Kết quả đánh giá cho thấy mô-đun tiền lọc giảm đáng kể thời gian phát hiện và mô-đun tái xếp hạng cải thiện đáng kể độ chính xác của phát hiện. Asteria-Pro vượt trội hơn so với các phương pháp tiên tiến hiện có về cả độ chính xác và hiệu suất. Trong đánh giá 8.5, chúng tôi phát hiện rằng hiệu suất của các phương pháp BCSD khác nhau có thể thay đổi rộng rãi trong các tình huống sử dụng khác nhau.
* Chúng tôi chứng minh tính hữu ích của Asteria-Pro bằng cách thực hiện việc phát hiện lỗ hổng firmware quy mô lớn thực tế. Asteria-Pro đã tìm thấy 1.482 chức năng có lỗ hổng với độ chính xác cao lên đến 91,65%. Chúng tôi phân tích phân bố lỗ hổng trong phần mềm được sử dụng rộng rãi từ các nhà cung cấp IoT khác nhau để minh họa những kết quả đáng ngạc nhiên của chúng tôi.

1. Nền tảng:

* Trước tiên, chúng tôi sẽ tóm tắt ngắn gọn cấu trúc AST được sử dụng trong công việc này, tiếp theo là một ví dụ về AST với cấu trúc ổn định hơn CFG (Control Flow Graph) qua các kiến trúc. Sau đó, chúng tôi giới thiệu mô hình Tree-LSTM được sử dụng trong việc mã hóa AST. Cuối cùng, chúng tôi đưa ra định nghĩa vấn đề rộng lớn cho việc áp dụng BCSD vào tìm kiếm lỗi.
  1. Cây Abstract Syntax:

A diagram of a graph

Description automatically generated

Hình 1. Mã nguồn của hàm histsizesetfn và cây cú pháp đã được giải mã tương ứng của kiến trúc x86.

Bảng 1. Các câu lệnh và biểu thức trong cây cú pháp (AST). Chúng tôi đếm số lượng câu lệnh và biểu thức cho các nút trong cây cú pháp sau khi giải mã bằng IDA Pro và liệt kê các câu lệnh và biểu thức phổ biến. Bảng này có thể được mở rộng nếu có thêm câu lệnh hoặc biểu thức mới được giới thiệu.

A close-up of a note

Description automatically generated

2.1.1 Mô tả AST.

* AST (Abstract Syntax Tree) là biểu diễn cây của cấu trúc cú pháp trừu tượng của mã trong quá trình biên dịch và giải mã. Công việc này tập trung vào các AST được trích xuất thông qua quá trình giải mã các chức năng nhị phân. Các cây con khác nhau trong một AST tương ứng với các phạm vi mã khác nhau trong mã nguồn. Hình 1 hiển thị một AST đã được giải mã tương ứng với mã nguồn của hàm histsizesetfn trong zsh v5.6.2. Zsh là một phần mềm shell phổ biến được thiết kế để sử dụng tương tác, và hàm histsizesetfn thiết lập giá trị của một tham số. Các đoạn đường nối giữa mã nguồn và AST trong Hình 1 cho thấy rằng một nút trong AST tương ứng với một biểu thức hoặc một câu lệnh trong mã nguồn. Một biến hoặc một giá trị hằng số được biểu diễn bằng một nút lá trong AST. Chúng tôi nhóm các nút trong một AST thành hai loại: i) nút câu lệnh và ii) nút biểu thức dựa trên chức năng của chúng, như được hiển thị trong Bảng 1. Các nút câu lệnh điều khiển luồng thực thi chức năng trong khi các nút biểu thức thực hiện các phép tính khác nhau. Các nút câu lệnh bao gồm if, for, while, return, break và các câu lệnh khác. Các nút biểu thức bao gồm các phép toán số học phổ biến và các phép toán bit.

2.1.2 Ưu thế cấu trúc AST.

* Cả CFG và AST đều là biểu diễn cấu trúc của một chức năng. CFG của một chức năng chứa các mối quan hệ nhảy giữa các khối cơ bản chứa các chuỗi mã thẳng đứng [38]. Mặc dù CFG đã được sử dụng cho việc đo lường sự tương đồng trong BCSD [31], David và cộng sự [24] đã chứng minh rằng cấu trúc CFG khác nhau đáng kể giữa các kiến trúc khác nhau. Chúng tôi quan sát thấy rằng AST cho thấy sự ổn định kiến trúc tốt hơn qua các kiến trúc so với CFG. Điều này bởi vì AST được tạo ra từ các biểu diễn trung gian không phụ thuộc vào máy tính, được giải mã từ các hướng dẫn lắp ráp trong quá trình giải mã [21]. Hình 2 mô tả sự tiến hóa của AST và CFG cho các kiến trúc x86 và ARM, tương ứng. Đối với CFG từ x86 đến ARM, chúng tôi quan sát thấy số khối cơ bản thay đổi từ 4 thành 1, và số lượng hướng dẫn lắp ráp đã thay đổi nhiều. Tuy nhiên, AST, được dựa trên một biểu diễn trung gian cấp cao hơn, khác rất ít giữa x86 và ARM, với sự khác biệt được đánh dấu bằng các ô màu xanh. Ngoài ra, AST giữ nguyên ý nghĩa của chức năng, làm cho nó trở thành cấu trúc lý tưởng cho việc phát hiện sự tương đồng qua nhiều nền tảng.

A screenshot of a computer

Description automatically generated

Hình 2. Các AST và CFG của hàm histsizesetfn dưới các kiến trúc khác nhau.

* 1. Model cây LSTM:
* Trong xử lý ngôn ngữ tự nhiên, Mạng Nơ-ron Đệ Quy (RNN) được sử dụng rộng rãi và hoạt động tốt hơn so với Mạng Nơ-ron Tích Chập [72]. RNNs nhận dạng các chuỗi có chiều dài tùy ý bởi vì một câu có thể bao gồm bất kỳ số lượng từ nào. Tuy nhiên, RNN tiêu chuẩn không thể xử lý các phụ thuộc dài hạn do vấn đề biến mất gradient và bùng nổ gradient. Là một trong các biến thể của RNN, Long Short-Term Memory (LSTM) [39] đã được đề xuất để giải quyết các vấn đề này. LSTM giới thiệu cơ chế cổng bao gồm cổng nhập, cổng quên và cổng xuất. Các cổng điều khiển việc chuyển thông tin để tránh việc biến mất gradient và bùng nổ gradient (chi tiết tính toán trong phần 6.1). Tuy nhiên, LSTM chỉ có thể xử lý đầu vào chuỗi mà không thể xử lý đầu vào có cấu trúc. Tree-LSTM được đề xuất để xử lý đầu vào có cấu trúc cây [59]. Quá trình tính toán bởi mô hình Tree-LSTM được thực hiện từ dưới lên. Đối với mỗi nút không phải lá trong cây, tất cả thông tin từ các nút con được thu thập và sử dụng để tính toán cho nút hiện tại. Trong các nhiệm vụ phân loại cảm xúc và liên quan ý nghĩa ngữ nghĩa, Tree-LSTM hoạt động tốt hơn so với mạng LSTM cơ bản. Có hai loại Tree-LSTM được đề xuất trong công việc [60]: Child-Sum Tree-LSTM và Binary Tree-LSTM. Các nhà nghiên cứu đã chỉ ra rằng Binary Tree-LSTM hoạt động tốt hơn so với Child-Sum Tree-LSTM [60]. Vì Child-Sum Tree-LSTM không xem xét thứ tự của các nút con, trong khi thứ tự các câu lệnh trong AST phản ánh ý nghĩa của chức năng, chúng tôi sử dụng Binary Tree-LSTM cho việc mã hóa AST của chúng tôi.
  1. Tối ưu hóa biên dịch gọi hàm:
* Có hai loại tối ưu hóa biên dịch gọi hàm chính có thể ảnh hưởng đến phân tích tương tự mã nhị phân: hàm nội tuyến và hàm nội tại (intrinsic functions).
* Hàm nội tuyến (Function inline): Hàm nội tuyến là một kỹ thuật tối ưu hóa trình biên dịch, trong đó mã của một hàm được chèn trực tiếp vào hàm gọi, thay vì tạo một lời gọi hàm riêng biệt. Điều này có thể cải thiện hiệu suất chương trình bằng cách giảm thiểu chi phí của các lời gọi hàm và cải thiện việc sử dụng bộ nhớ cache. Quyết định nội tuyến một hàm thường do trình biên dịch đưa ra dựa trên các yếu tố khác nhau như kích thước hàm, tần suất gọi và không gian đăng ký có sẵn.
* Hàm nội tại (Intrinsic function): Hàm nội tại (còn được gọi là hàm tích hợp) là các hàm đặc biệt được triển khai bởi trình biên dịch chính nó và được ánh xạ thành một lệnh duy nhất hoặc một chuỗi lệnh trong kiến trúc đích. Các hàm này cung cấp truy cập cấp thấp vào phần cứng và được sử dụng để thực hiện các hoạt động cấp thấp, chẳng hạn như phép tính số học, xử lý bit và truy cập bộ nhớ. Hàm nội tại thường được sử dụng trong mã quan trọng về hiệu suất, nơi sử dụng các lệnh cấp thấp có thể dẫn đến tăng tốc đáng kể so với mã tương đương viết bằng một ngôn ngữ cấp cao hơn.

1. Nghiên cứu sơ bộ:

* Nghiên cứu sơ bộ này nhằm đánh giá và khám phá các đặc trưng hàm có sẵn mà hiệu quả trong việc xác định các hàm không đồng thời để hướng dẫn thiết kế bộ lọc trước của chúng tôi. Để đánh giá các đặc trưng, chúng tôi chuẩn bị cơ sở mã và tích hợp một số chỉ số (phần 3.1). Chúng tôi tập trung chủ yếu vào việc đánh giá và so sánh các đặc trưng thông thường phổ biến hiện có trong các nghiên cứu đáng chú ý đã có (phần 3.2).
  1. Đánh giá Benchmark:

3.1.1 Tập dữ liệu

* Để tạo ra các đặc trưng đáng tin cậy, chúng tôi biên dịch một bộ sưu tập lớn các tập tin nhị phân từ 184 phần mềm mã nguồn mở (OSS), bao gồm các phần mềm phổ biến như OpenSSL, FFmpeg, Binutils, v.v. Vì công cụ của chúng tôi nhằm thực hiện phát hiện sự tương đồng trên các kiến trúc khác nhau, chúng tôi biên dịch các phần mềm mã nguồn mở này cho bốn kiến trúc phổ biến: X86, X64, ARM và PowerPC. Ngoài ra, chúng tôi cân nhắc các cài đặt biên dịch mặc định trong quá trình biên dịch với việc sử dụng thực tế. Sau khi biên dịch, nhiều tập tin nhị phân thử nghiệm với tiền tố "test" hoặc hậu tố "buildtest" được tạo ra để kiểm tra chức năng của phần mềm. Các tập tin nhị phân thử nghiệm này được loại bỏ khỏi bộ sưu tập vì 1) chức năng của chúng đơn giản và chỉ bao gồm vài dòng mã. 2) không tham gia vào việc thực thi thực sự của chức năng phần mềm. Sau khi loại bỏ, bộ sưu tập nhị phân còn lại 1.130 tệp, tức là 226 tệp cho mỗi kiến trúc.
* Chúng tôi tạo ra một tập dữ liệu lớn gồm các cặp hàm đồng thời và không đồng thời dựa trên tên hàm của chúng. Tên hàm được giữ lại trong phần mềm sau khi biên dịch, cho phép chúng tôi xây dựng tập dữ liệu. Để tạo các cặp hàm đồng thời, chúng tôi chọn các hàm nhị phân có cùng tên hàm trong cùng một phần mềm. Trong khi đó, các hàm có tên khác nhau được xem là không đồng thời. Ví dụ, nếu hàm A xuất hiện trong mã nguồn, quá trình biên dịch sẽ tạo ra bốn phiên bản hàm nhị phân cho các kiến trúc bộ lệnh khác nhau: X86, X64, ARM và PowerPC. Các biến thể này của các hàm được xem là đồng thời với nhau. Chúng tôi trích xuất tổng cộng 529.096 hàm nhị phân, bao gồm 132.274 hàm duy nhất cho mỗi kiến trúc. Để tránh quá khớp trong đánh giá cuối cùng, chúng tôi ngẫu nhiên chọn 40.111 hàm từ mỗi kiến trúc. Trong số này, chúng tôi ngẫu nhiên chọn � hàm làm hàm nguồn để đánh giá khả năng lọc của các đặc trưng đa dạng. Đối với mỗi hàm nguồn �
* ��, chúng tôi xây dựng một bộ gồm các hàm ứng cử viên bao gồm � hàm nhị phân được chọn ngẫu nhiên và ba hàm đồng thời với �
* �
* �
* . Kết quả là mỗi hàm nguồn �
* �
* �
* hình thành ba cặp hàm đồng thời và � cặp hàm không đồng thời.

3.1.2 Các chỉ số đánh giá

* Chúng tôi sử dụng tỷ lệ true positive (TPR) và tỷ lệ false positive (FPR) để đánh giá khả năng lọc của các đặc trưng khác nhau. TPR thể hiện khả năng giữ lại các hàm đồng thời, trong khi FPR thể hiện khả năng loại trừ các hàm không đồng thời. Trong pha lọc tiếp theo, mục tiêu của chúng tôi là xác định các đặc trưng có thể loại bỏ các hàm không đồng thời một cách hiệu quả nhất (FPR thấp) trong khi vẫn giữ lại tất cả các hàm đồng thời (TPR rất cao).
* Đối với một hàm nguồn � � � , tất cả các cặp hàm trong bộ hàm ứng cử viên được đo lường bằng các điểm số tương tự đặc trưng khác nhau. Các cặp hàm có điểm số tương tự dưới ngưỡng � sẽ được lọc bỏ. Trong các cặp hàm còn lại, các cặp hàm đồng thời được coi là true positive (TP) � � trong khi các cặp hàm không đồng thời được coi là false positive (FP) ��. Các công thức sau đây minh họa cách tính ba chỉ số này cho các đặc trưng khác nhau:

A white background with black dots

Description automatically generated

* 1. Đánh giá Đặc trưng Ứng cử viên:
* Chúng tôi nhằm xác định các đặc trưng lọc hiệu quả nhất và hiệu quả bằng cách đánh giá các đặc trưng hiện có được đề xuất trong các nghiên cứu trước đó và các biến thể của chúng. Dựa vào kết quả đánh giá, chúng tôi lựa chọn và cải tiến các đặc trưng ứng cử viên để đáp ứng yêu cầu lọc, đó là loại bỏ càng nhiều hàm không đồng thời càng tốt trong khi vẫn giữ lại tất cả các hàm đồng thời.
  + 1. Lựa chọn đặc trưng.
* Chúng tôi thu thập các đặc trưng cơ bản từ nghiên cứu trước đó [31, 66, 67, 71] và phân loại chúng thành hai nhóm: đặc trưng họ CFG (Control Flow Graph) và đặc trưng họ AST (Abstract Syntax Tree).
* Các đặc trưng họ CFG (Control Flow Graph) bao gồm bốn loại đặc trưng số: số lệnh (No. Instruction), lệnh số học (No. Arithmetic), lệnh gọi hàm (No. Callee), và lệnh logic (No. Logic), cùng với hai đặc trưng hằng số: hằng số chuỗi (String Constant) và hằng số số (Numeric Constant) [31]. Ngoài ra, chúng tôi giới thiệu một đặc trưng mới được đề xuất gọi là danh sách tên của các hàm được gọi (NCL - named callee list) để thu thập thông tin chuỗi văn bản về các hàm được gọi giữ lại tên hàm của họ do liên kết động. Cụ thể, NCL được thiết kế để là một danh sách các hàm được gọi là các hàm được nhập khẩu hoặc xuất khẩu. Những hàm này giữ nguyên tên gốc của họ vì chúng được sử dụng như các định danh để tham chiếu đến các hàm trong các phần khác của mã.
* Vì AST là cần thiết cho việc tính toán mã hóa mô hình (mục 6), chúng tôi tóm tắt ba đặc trưng cú pháp như các đặc trưng họ AST (Abstract Syntax Tree).
* • No. AST Nodes: The number of AST nodes.
* • AST Node Cluster: The number of diferent node types in the AST. For example, in Figure 1, the AST node
* cluster is denoted as [����� : 3,� � : 1, ������ : 1, ���� : 1, ��� : 3, ����� : 2, ��� : 2, ��� : 4,�� : 1].
* • AST Fuzzy Hash: We irst generate a node sequence by traversing the AST preorder. Then we apply the fuzzy hash algorithm [44] to generate the fuzzy hash of the AST.

3.2.2 Tính toán Độ tương đồng của Đặc trưng.

* Định dạng của các đặc trưng chia chúng thành hai loại với các tính toán tương đồng riêng biệt: loại giá trị (value type) và loại chuỗi (sequence type). Các đặc trưng loại giá trị bao gồm No. Instruction, No. Arithmetic, No. Logic, No. Callee và No. AST nodes. Các đặc trưng loại chuỗi bao gồm Numeric Constant, String Constant, AST Node Cluster, AST Fuzzy Hash và NCL. Đối với các đặc trưng loại giá trị, chúng tôi sử dụng tỷ lệ sự khác biệt tương đối (relative difference ratio) được thể hiện như sau để tính toán độ tương đồng:
* (Ở đây có lẽ có một công thức hoặc thông tin liên quan đến tính toán độ tương đồng của các đặc trưng loại giá trị, nhưng tôi không thể nhìn thấy nó trong câu hỏi của bạn. Nếu bạn có công thức hoặc thông tin thêm, hãy cung cấp để tôi có thể giúp bạn dịch.)

A black and white math equation

Description automatically generated

* Ở đây, �1 và �2 là các giá trị của đặc trưng. Đối với mỗi đặc trưng loại chuỗi, chúng tôi trước tiên sắp xếp các mục của đặc trưng và sau đó nối chúng thành một chuỗi duy nhất. Sau đó, chúng tôi sử dụng tỷ lệ chuỗi chung (CSR - common sequence ratio) dựa trên chuỗi chung dài nhất (LCS - longest common sequence) như sau:

A math equation with numbers and symbols

Description automatically generated

A diagram with colorful lines

Description automatically generated

* Ở đây, �1 và �2 là các chuỗi đặc trưng, và hàm ��� (·, ·) trả về độ dài của chuỗi chung dài nhất giữa �1 và �2. Hai phương trình trên được sử dụng để tính toán độ tương đồng của các đặc trưng khác nhau.

3.2.3 Kết quả đánh giá.

* Trong quá trình đánh giá, các giá trị của � và � trong mục 3.1.2 được đặt là 1000 và 20.000, tương ứng. Như được mô tả trong Hình 3, các tỷ lệ TPR (True Positive Rate) và FPR (False Positive Rate) tính toán cho mỗi đặc trưng dưới nhiều ngưỡng được trình bày dưới dạng đường cong ROC (Receiver Operating Characteristic) [75]. Bên cạnh đó, chúng tôi tính toán diện tích dưới đường cong ROC (AUC), thể hiện khả năng của đặc trưng phân biệt giữa các hàm homolog và không homolog. Các giá trị AUC của các đặc trưng trích xuất từ AST (tức là No. AST Nodes, AST Node Cluster và AST Fuzzy Hash) đều cao, như được trình bày trong Hình 3. Tuy nhiên, khi TPR cao, chúng tạo ra FPR cao. Hình 5 mô tả chi phí thời gian liên quan đến tính toán độ tương đồng cho các đặc trưng khác nhau. Rõ ràng, các đặc trưng loại chuỗi yêu cầu nhiều thời gian hơn so với các đặc trưng loại giá trị. Tuy nhiên, thời gian tiêu thụ của chúng nằm trong phạm vi chấp nhận được. Ít nhất có thể thực hiện ít nhất 105 phép tính chính xác mỗi giây.
* Chúng tôi quan sát từ Hình 3 rằng ở TPR cao (0.996), đặc trưng No. Callee tạo ra một FPR tương đối thấp hơn (0.111). Nhớ lại yêu cầu của giai đoạn lọc, chúng tôi nhằm chọn các đặc trưng có FPR thấp ở TPR rất cao. Các đặc trưng có AUC cao không nhất thiết đáp ứng mục tiêu của chúng tôi. Ví dụ, đặc trưng AST Node Cluster có một FPR cao hơn (0.47) so với đặc trưng No. Callee (FPR = 0.111) ở cùng TPR (0.996), mặc dù đặc trưng AST Node Cluster có một AUC cao hơn (0.978) so với đặc trưng No. Callee (AUC = 0.944). Liên quan đến điều này, chúng tôi đề xuất một độ đo mới, ������ , biểu thị TPR cao và FPR thấp hơn.

A math equation with black text

Description automatically generated

* Figure 4 biểu thị các đường cong ������ của các đặc trưng khác nhau ở các ngưỡng tương đồng khác nhau. Kết quả cho thấy đặc trưng "NCL" có ������ cao nhất là 0.92 trong số tất cả các đặc trưng ứng cử viên. Nó đạt được tỷ lệ true positive cao ở mức false positive rate thấp, với điểm AUC tương đối cao là 0.963. Đặc trưng "No. Callee" có kết quả hơi kém hơn, với điểm AUC là 0.944 và ������ là 0.902. Đặc trưng "String Constant" hiển thị ������ tương đối cao ở ngưỡng rất thấp (ví dụ, 0.01) vì nó quyết định một cách rõ ràng tính đồng loạt của các hàm. Cụ thể, nếu hai hàm có cùng chuỗi, rất có thể chúng là đồng loạt. Mặc dù ������ không tăng khi ngưỡng tăng, điều này là do một số hàm không bao gồm hằng số chuỗi, điều này giới hạn số cặp true positive. Dựa trên hiệu suất lọc của các đặc trưng ứng cử viên, chúng tôi đã quyết định sử dụng NCL kết hợp với No. Callee và String Constant cho thiết kế lọc trước của chúng tôi.

A diagram of a diagram

Description automatically generated

Fig. 6. Quy trình làm việc của Asteria-Pro. DK là viết tắt của Kiến thức lĩnh vực (Domain Knowledge). DL là viết tắt của Học sâu (Deep Learning).

4. TỔNG QUAN VỀ PHƯƠNG PHÁP:

* Asteria-Pro bao gồm ba mô-đun chính: DK-based Preiltration (Tiền xử lý dựa trên Kiến thức lĩnh vực), DL-based Similarity Calculation (Tính toán độ tương đồng dựa trên Học sâu), và DK-based Re-ranking (Sắp xếp lại dựa trên Kiến thức lĩnh vực), như được hiển thị trong Hình 6. Ở đây, DK đại diện cho Kiến thức lĩnh vực (Domain Knowledge) và DL đại diện cho Học sâu (Deep Learning).
* Mô-đun tiền xử lý dựa trên Kiến thức lĩnh vực sử dụng các đặc trưng cú pháp để loại bỏ các hàm không đồng loạt khỏi các hàm ứng cử viên một cách nhẹ nhàng và hiệu quả (xem mục 5). Mô-đun tính toán độ tương đồng dựa trên Học sâu mã hóa AST thành các vectơ biểu diễn bằng mô hình Tree-LSTM và xác định điểm tương đồng giữa hàm mục tiêu và các hàm còn lại bằng mạng Siamese (xem mục 6). Mô-đun sắp xếp lại dựa trên Kiến thức lĩnh vực tái sắp xếp các hàm đồng loạt ứng cử viên do mô-đun tính toán độ tương đồng dựa trên Học sâu tạo ra bằng cách sử dụng các đặc trưng cấu trúc nhẹ nhàng, chẳng hạn như mối quan hệ gọi hàm. Bằng cách tích hợp ba mô-đun này, Asteria-Pro phát hiện các hàm đồng loạt trên các kiến trúc một cách hiệu quả và hiệu quả.

5 DK-BASED PREFILTRATION:

* Ở giai đoạn này, Asteria-Pro nhằm tích hợp một cơ chế lọc hiệu quả và hiệu quả. Để đạt được mục tiêu này, chúng tôi đã tóm tắt các thách thức liên quan đến việc khai thác đầy đủ đặc trưng NCL. Dựa trên những thách thức này, chúng tôi đã phát triển một thuật toán độc đáo vượt qua những rào cản này và cho phép xây dựng cơ chế lọc.

5.1 Những Thách thức Khai thác:

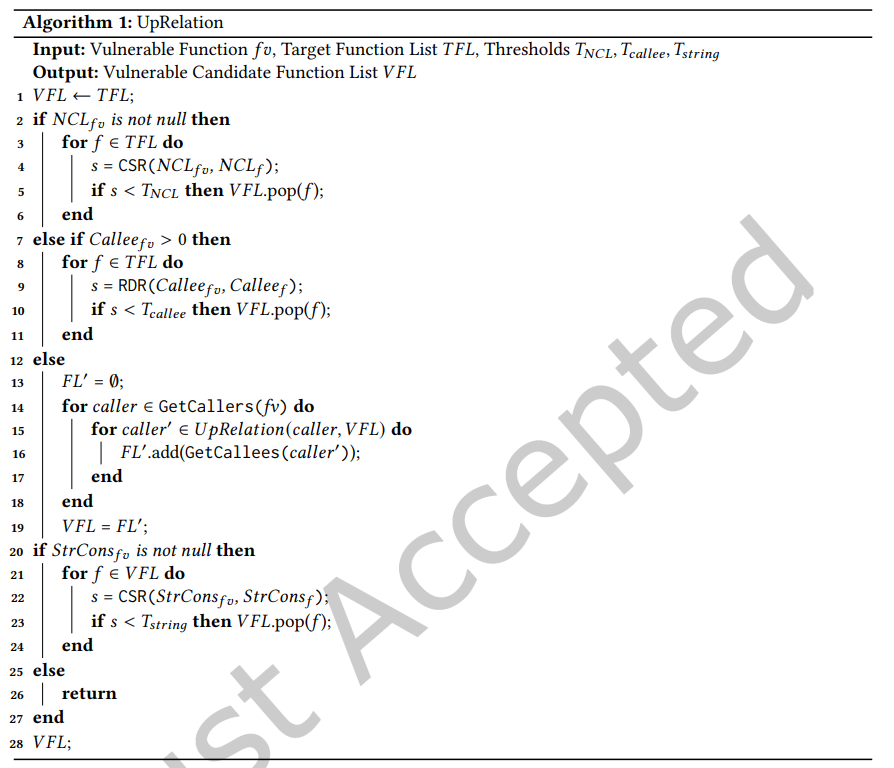
* Chúng tôi đã thủ công kiểm tra các trường hợp âm tính sai trong đó các hàm đồng loạt bị lọc bởi cả hai đặc trưng NCL và No. of callee. Qua việc kiểm tra này, chúng tôi đã xác định những thách thức liên quan đến việc khai thác đúng đắn các đặc trưng NCL và No. of callee để giải quyết các trường hợp âm tính sai này.
* Thách thức Khai thác 1 (EC1). Tên hàm được trang trí (Decorated callee function name). Tên hàm được trang trí là kết quả của việc trang trí tên hàm bởi trình biên dịch bằng cách sử dụng các kỹ thuật khác nhau [8]. Một trong các kỹ thuật như vậy là "name mangling", được sử dụng bởi trình biên dịch C++ để mã hóa tên hàm với thông tin bổ sung về các tham số và kiểu trả về để hỗ trợ chức năng chồng chéo (function overloading). Tên của hàm sau khi được trang trí có thể khác so với tên gốc trong mã nguồn và sẽ khác biệt giữa các kiến trúc khác nhau, đặc biệt là X86 và X64, do kiểu trả về khác nhau.
* Thách thức Khai thác 2 (EC2). Trong một số hàm cụ thể, thường được gọi là các nút lá trong một đồ thị gọi hàm (call graph), không có hàm được gọi. Những hàm này tự chứa và không gọi bất kỳ hàm nào khác trong mã của chúng. Do đó, các nút lá không có các đặc trưng NCL và No. of callee có thể phân biệt.
* Thách thức Khai thác 3 (EC3). Các lệnh gọi hàm trong các hàm mục tiêu nhị phân có thể không luôn đồng nhất với mã nguồn. Các lệnh gọi hàm có thể được thêm hoặc xóa do tối ưu hóa của trình biên dịch. Nguyên nhân của việc thay đổi lệnh gọi hàm là việc lồng ghép hàm, thay thế hàm nội tại (intrinsic function), thay thế lệnh để tối ưu hóa, những thay đổi này có thể hành xử khác nhau trên các kiến trúc khác nhau. Những thách thức này được giới thiệu trong mục 2.3.
* Để vượt qua những thách thức khai thác, chúng tôi cải tiến đặc trưng ECL và đề xuất một thuật toán mới gọi là UpRelation.

5.2 Định nghĩa của NCL:

* Phần này cung cấp định nghĩa chính xác của đặc trưng NCL để tăng tính rõ ràng và chính xác. NCL được xây dựng dựa trên đồ thị gọi của phần mềm. Đồ thị gọi được định nghĩa bằng cách biểu diễn tất cả các hàm như các nút và mối quan hệ gọi giữa chúng là các cạnh: Gọi đồ thị �� có thể được định nghĩa như sau: �� = (V, E), trong đó V = {�|� là một hàm} đại diện cho tập hợp các nút và E = {(�, �)|� gọi � } đại diện cho tập hợp các cạnh. Đối với bất kỳ cạnh (�, �) ∈ E, chúng ta nói rằng hàm � là một hàm được gọi của hàm �. Để thuận tiện cho việc liên kết, tên hàm trong bảng ký tự động ��� (tức là bảng nhập khẩu và xuất khẩu) được bảo tồn [36]. Ví dụ, nếu một hàm mục tiêu gọi một hàm bên ngoài như 'strcpy', tên hàm được gọi 'strcpy' vẫn được giữ trong bảng nhập khẩu, thay vì bị xóa sau khi cắt bỏ nhị phân. NCL của một hàm mục tiêu � được định nghĩa như sau: NCL(�) = {�|� ∈ V, � ∈ ��� và (� , �) ∈ E}, trong đó � được sắp xếp theo địa chỉ hướng dẫn gọi.
* Để giải quyết EC1, chúng tôi sử dụng hai chiến lược để khôi phục lại tên hàm gốc. Đầu tiên, đối với các tên được trang trí bằng C++, chúng tôi sử dụng công cụ khôi phục cxxfilt [5] để khôi phục lại tên hàm. Thứ hai, đối với các hàm được trang trí khác, chúng tôi định nghĩa các quy tắc đánh giá để khôi phục lại tên hàm. Ví dụ, chúng tôi khôi phục cuộc gọi hàm 'gets' bằng cách thay thế nó bằng 'gets', bằng cách loại bỏ gạch dưới ở đầu tiên. Trong trường hợp mà một hàm gọi cùng một hàm nhiều lần, chúng tôi giữ lại nhiều tên hàm giống nhau.

5.3 Thuật toán lọc:

* Để giải quyết hai thách thức bổ sung, chúng tôi đề xuất một thuật toán dựa trên độ tương đồng của hàm được gọi gọi là UpRelation. Thuật toán này sử dụng thông tin ngữ cảnh trong đồ thị gọi để vượt qua các thách thức EC2 và EC3. Cụ thể, thuật toán sử dụng các nút cha của các nút lá trong đồ thị gọi để phù hợp với các nút lá tương tự và giải quyết thách thức EC2. Trong thuật toán, chúng tôi áp dụng chiến lược "drill-down" (đục xuống) kết hợp ba đặc trưng: NCL, No. Callee và String Constants, dựa trên thông tin của chúng. No. Callee của hàm � được ký hiệu là �������, và tập hợp các String Constants của hàm � được ký hiệu là ��������.
* Cho một hàm mục tiêu �� , Thuật toán 1 nhằm loại bỏ hầu hết các hàm không đồng loạt trong khi giữ lại các hàm ứng cử viên không an toàn trong một danh sách (� � �) từ danh sách hàm mục tiêu (� � �). Đoạn mã từ dòng 2 đến 6 thực hiện việc lọc khi đặc trưng ���� � của �� không trống. Cụ thể, thuật toán tính toán tỷ lệ độ tương đồng của các hàm gọi (���) giữa ���� � và ���� của tất cả các hàm ứng cử viên ở dòng 4. Sau đó, nó loại bỏ các hàm có ��� nhỏ hơn một ngưỡng ����. Tương tự, khi số lượng hàm gọi (������� � ) của �� không bằng không, thuật toán loại bỏ các hàm bằng cách tính điểm Relevance Distance Ratio (���) từ dòng 7 đến 11. Phần quan trọng nhất của thuật toán là từ dòng 12 đến 19, nơi nó phù hợp với các hàm lá để giải quyết EC2. Tất cả các hàm gọi của �� được truy cập trước tiên, và thuật toán sử dụng � ��������� để tìm ra tất cả các hàm tương tự với hàm gọi ������ ở dòng 14. Đối với mỗi hàm tương tự ������′, thuật toán xem xét tất cả các hàm được gọi của nó như là các hàm ứng cử viên không an toàn tại dòng 16. Việc phù hợp với cùng các hàm lá bằng cách xác định các hàm gọi cùng nhau giới thiệu một số hàm dư thừa (lá) chia sẻ cùng một hàm gọi nhưng không giống như hàm lá. Để loại bỏ các hàm dư thừa này, thuật toán sử dụng độ tương đồng chuỗi ở dòng 22. Sau khi lọc theo các hàm được gọi và chuỗi, cuối cùng thuật toán thu được danh sách hàm ứng cử viên không an toàn mong đợi � � �.
* Minh họa tính toán các nút lá (Leaf Node). Khi Strings, No. Callee và NCL không trống, tính toán độ tương đồng trong thuật toán của chúng tôi là dễ dàng. Khía cạnh khó khăn của thuật toán nằm ở việc quản lý các hàm lá không gọi các hàm khác. Để cung cấp một minh họa rõ ràng hơn, chúng tôi đã sử dụng một ví dụ được mô tả trong Hình 7 để chứng minh tại sao các hàm đồng loạt của hàm lá �� được bảo tồn sau khi tiền xử lý. Trong ví dụ này, chúng ta giả định rằng hàm lá �� không bao gồm bất kỳ chuỗi nào. Thuật toán tiếp tục tìm kiếm hàm gọi của nó và tổng hợp NCL của nó là [��., ����., ��., ����.], trong đó Ex. Func. là viết tắt của 'Exported Function' và Im. Func. là viết tắt của 'Imported Function'. Tương tự, thuật toán thu thập NCL của hàm gọi của � ′� và cố gắng liên kết giữa hai NCL này. Chúng tôi giả định rằng các hàm đồng loạt từ cùng một phần mềm có cùng người gọi, biểu thị rằng các hàm gọi ���� � ′ � và ���� � ′ � gọi các hàm xuất và nhập tương tự nhau. Do đó, NCL của hai hàm gọi bao gồm cùng các phần tử [��., ����., ��., ����.]. Khi kết hợp thành công NCL của hàm gọi ���� � ′ �, thuật toán giữ lại tất cả các nút con của nó, bao gồm � ′ � ′, Im. Func., và Ex. Func., và loại bỏ tất cả các hàm khác. Kết quả là, hàm đồng loạt � ′� của �� được bảo tồn sau khi tiền xử lý.



6. TÍNH TOÁN ĐỘ TƯƠNG ĐỒNG DỰA TRÊN HỌC SÂU:  
- Mô-đun này tính toán độ tương đồng giữa hai AST của hàm bằng cách mã hóa chúng thành các vector và áp dụng kiến trúc Siamese để tính toán độ tương đồng giữa các vector đã được mã hóa. Hình 8 mô tả quá trình tính toán.

A diagram of a couple of circles with text

Description automatically generated

Hình 7. Hàm lá � ′ � được phù hợp với hàm đồng loạt �� của nó (thay vì bị loại bỏ). Ex. Func. viết tắt của 'Exported Function' (hàm được xuất). Im. Func. viết tắt của 'Imported Function' (hàm được nhập).

A diagram of a algorithm

Description automatically generated

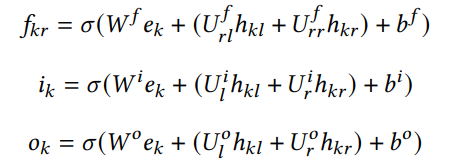
Hình 8. Kiến trúc Siamese và Mã hóa Tree-LSTM.

6.1 Mã hóa Tree-LSTM:

* Khi có một cây AST, mô hình Tree-LSTM mã hóa nó thành một vectơ biểu diễn. Mô hình Tree-LSTM ban đầu được đề xuất để mã hóa cây biểu diễn của một câu và tóm tắt thông tin ngữ nghĩa trong xử lý ngôn ngữ tự nhiên. Mô hình Tree-LSTM có thể giữ lại mọi thuộc tính của cơ chế điều khiển LSTM thông thường khi xử lý đầu vào có cấu trúc cây. Sự khác biệt chính giữa LSTM thông thường và Tree-LSTM là cách xử lý đầu ra của các nút tiền đề. LSTM thông thường sử dụng đầu ra của chỉ một nút tiền đề trong đầu vào dạng chuỗi. Chúng tôi sử dụng Tree-LSTM để tích hợp đầu ra của tất cả các nút con trong AST để tính toán nút hiện tại.
* Để minh họa quá trình mã hóa Tree-LSTM, chúng tôi giả định rằng nút �� có hai nút con �� và ��. Mã hóa Tree-LSTM của nút �� sử dụng ba loại đầu vào: nhúng nút �� của ��, trạng thái ẩn ℎ�� và ℎ��, và trạng thái ô ��� và ��� như minh họa trong Hình 8. Nhúng nút �� được tạo bằng cách sử dụng mô hình đã được huấn luyện trước CodeT5 để nhúng nút �� thành một vectơ biểu diễn chiều cao. ℎ�� , ℎ��, ��� , và ��� là đầu ra từ quá trình mã hóa của các nút con. Trong quá trình mã hóa nút trong Tree-LSTM, có ba cổng và ba trạng thái quan trọng trong tính toán. Ba cổng này được tính toán để lọc thông tin để tránh sự bùng nổ và mất mạch gradient [59]. Chúng là cổng đầu vào, cổng đầu ra và cổng quên. Có hai cổng quên ��� và ���, lọc trạng thái ô từ nút con bên trái và nút con bên phải một cách riêng biệt. Như được thể hiện trong Mã hóa nút trong Hình 8, cổng quên được tính toán bằng cách kết hợp ℎ�� , ℎ�� và �� . Tương tự như cổng quên, cổng đầu vào và cổng đầu ra cũng được tính toán bằng cách kết hợp ℎ�� , ℎ�� và �� . Chi tiết về ba loại cổng này như sau:

A black and orange letters

Description automatically generated



* Trong đó �� và �� là cổng đầu vào và cổng đầu ra, và ký hiệu �� là hàm kích hoạt sigmoid. Ma trận trọng số ��, �� và độ lệch �� là khác nhau tương ứng với các cổng khác nhau. Sau khi tính toán các cổng, có ba trạng thái ��, �� và ℎ� trong Tree-LSTM để lưu trữ các mã hóa trung gian được tính toán dựa trên đầu vào ℎ��, ℎ�� và ��. Trạng thái được lưu trữ �� kết hợp thông tin từ nhúng nút �� và các trạng thái ẩn ℎ�� và ℎ�� (Phương trình 10). Lưu ý rằng trạng thái �� sử dụng hàm kích hoạt tanh thay vì hàm sigmoid để giữ thêm thông tin từ các đầu vào. Trạng thái ô �� kết hợp thông tin từ trạng thái được lưu trữ �� và các trạng thái ô ��� và ��� được lọc qua các cổng quên (Phương trình 11). Trạng thái ẩn ℎ� được tính toán bằng cách kết hợp thông tin từ trạng thái ô �� và cổng đầu ra �� (Phương trình 12). Ba trạng thái được tính toán như sau:

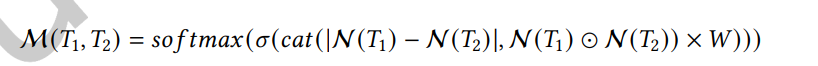
A group of black symbols

Description automatically generated

* Ở đó, ⊙ đại diện cho phép nhân Hadamard [40]. Sau khi tính toán trạng thái ẩn và trạng thái đầu vào, quá trình mã hóa của nút hiện tại �� được hoàn thành. Sau đó, các trạng thái �� và ℎ� sẽ được sử dụng cho việc mã hóa của nút cha của ��. Trong quá trình mã hóa AST, Tree-LSTM mã hóa mỗi nút trong AST từ dưới lên như được thể hiện trong Mã hóa Tree-LSTM trong Hình 8. Sau khi mã hóa tất cả các nút trong AST, trạng thái ẩn của nút gốc được sử dụng làm mã hóa của AST.

6.2 Tính toán Siamese:

* Bước này sử dụng kiến trúc Siamese, tích hợp hai mô hình Tree-LSTM giống hệt nhau để tính toán độ tương đồng giữa các vectơ biểu diễn đã được mã hóa. Chi tiết của kiến trúc Siamese M(θ1, θ2) được thể hiện trong Hình 8. Kiến trúc Siamese bao gồm hai mạng Tree-LSTM giống nhau chia sẻ cùng các tham số. Trong quá trình tính toán độ tương đồng, kiến trúc Siamese đầu tiên sử dụng Tree-LSTM để mã hóa cây AST thành các vectơ. Chúng tôi thiết kế kiến trúc Siamese với các phép trừ và nhân để lấy các quan hệ giữa hai vectơ biểu diễn. Sau các phép toán này, hai vectơ kết quả được ghép vào một vectơ lớn hơn. Sau đó, vectơ kết quả được đưa qua một lớp softmax để tạo ra một vectơ có 2 chiều. Phép tính được định nghĩa như sau:



* Trong đó, Θ là ma trận 2×2, ⊙ biểu thị phép nhân Hadamard [40], | · | biểu thị phép lấy giá trị tuyệt đối, hàm ���(·) biểu thị phép ghép các vectơ. Hàm softmax chuẩn hóa vectơ thành một phân bố xác suất. Vì Θ là ma trận trọng số 2×2, đầu ra của kiến trúc Siamese là một vectơ 2×1. Định dạng của đầu ra là [Dissimilarity Score, Similarity Score], trong đó giá trị đầu tiên biểu thị điểm không tương đồng và giá trị thứ hai biểu thị điểm tương đồng. Trong quá trình huấn luyện mô hình, định dạng đầu vào của kiến trúc Siamese là < �1,�2,����� >. Trong công việc của chúng tôi, vector nhãn [1, 0] biểu thị �1 và �2 đến từ các cặp hàm không đồng loạt và vector [0, 1] biểu thị đồng loạt. Vectơ kết quả và vectơ nhãn được sử dụng để tính toán tổn thất và đạo hàm của mô hình. Trong quá trình suy luận mô hình, giá trị thứ hai trong vectơ đầu ra được lấy làm độ tương đồng của hai cây AST, và độ tương đồng của các cây AST được sử dụng trong giai đoạn sắp xếp lại (re-ranking).

A diagram of a relationship between two different points

Description automatically generated

Hình 9. Ví dụ minh họa về giai đoạn sắp xếp lại (Re-ranking). Trong hộp chữ nhật với đường kẻ đứt là danh sách K hàm ứng cử viên hàng đầu đồng loạt của �1 do quá trình tìm kiếm (tức là phát hiện độ tương đồng dựa trên DL) tạo ra. Mũi tên đường liền chỉ ra mối quan hệ gọi hàm (ví dụ, �1 gọi đến � �1 �). Mũi tên đứt chỉ ra mối tương đồng với các hàm được sắp xếp lại.

7 DK-BASED RE-RANKING:

* Bước này cố gắng xác nhận tính đồng loạt của các hàm ứng cử viên hàng đầu k được đưa ra bởi mạng Tree-LSTM thông qua việc sắp xếp lại chúng. Trong giai đoạn trước đó, mạng Tree-LSTM suy ra thông tin ngữ nghĩa từ cây AST, đó là một đặc trưng trong nội hàm của hàm. Kiến thức thu được từ cây AST là không đủ để xác định tính đồng loạt của các hàm. Trong giai đoạn này, các mối quan hệ gọi hàm được sử dụng làm kiến thức ngữ cảnh để bù đắp cho thiếu sót kiến thức về các đặc trưng giữa các hàm trong mạng Tree-LSTM. Để đạt được điều này, chúng tôi thiết kế một thuật toán được gọi là Relational Structure Match. Khác với ứng dụng của người được gọi trong mô-đun tiền lọc, mô-đun này sử dụng thông tin từ các mối quan hệ gọi hàm rộng hơn để chỉ ra mức độ đồng loạt của các hàm ứng cử viên.

7.1 Motivated Example:

* Thuật toán của chúng tôi dựa trên một quan sát phù hợp với một nguyên tắc hiển nhiên: Nếu hàm **Func1** gọi hàm **Func2**, thì hàm đồng loạt **Func1'** cũng sẽ gọi hàm đồng loạt **Func2'** của **Func2**. Như được miêu tả trong Hình 9, chúng ta có **Func1** gọi **Func2**, và **Func1'** gọi **Func2'**. Giả sử quá trình tìm kiếm cho **Func1** đưa ra các hàm hàng đầu K, bao gồm hàm đồng loạt mục tiêu **Func1'**.
* Chúng tôi sau đó sử dụng mối quan hệ gọi hàm của **Func1** và **Func1'** để tiến hành việc kết hợp chính xác hàm được gọi lại cho việc sắp xếp lại. Các hàm được gọi của **Func1** được chia thành hai nhóm: hàm được đặt tên và hàm không đặt tên. Đối với các hàm được đặt tên, chúng tôi sử dụng tên của họ để so khớp các hàm được gọi lại giữa hàm nguồn **Func1** và các hàm ứng cử viên hàng đầu K. Đối với các hàm không đặt tên, chúng tôi sử dụng phát hiện độ tương đồng dựa trên mạng Tree-LSTM để tính toán độ tương đồng giữa các hàm được gọi lại giữa hàm nguồn **Func1** và các hàm ứng cử viên hàng đầu K. Nhớ lại quan sát, hàm đồng loạt **Func1'** của **Func1** giữ lại số lượng hàm được gọi lại tương đồng nhất.
* Sau khi sắp xếp lại các hàm ứng cử viên dựa trên các hàm được gọi lại đã khớp, hàm đồng loạt **Func1'** được sắp xếp lại ở vị trí đầu tiên.

7.2 Relational Structure Match Algorithm:

* Thuật toán nhằm điểm lại điểm cho mỗi hàm ứng cử viên bằng cách sử dụng mối quan hệ gọi hàm giữa hàm mục tiêu và các hàm ứng cử viên. Cấu trúc mối quan hệ đề cập đến các mối quan hệ gọi hàm giữa hàm mục tiêu và tất cả các hàm được gọi lại của nó, như được miêu tả trong phần ğ 7.1. Để khớp cấu trúc mối quan hệ, thuật toán thực hiện một trong hai phép toán khác nhau (�1 và �2) dựa trên việc hàm nguồn có hàm được gọi lại hay không.
* �1: Khi hàm nguồn �1 có một hoặc nhiều hàm được gọi lại, thuật toán trích xuất tất cả các hàm được gọi lại của �1 để xây dựng một tập hợp hàm được gọi lại kết hợp (MCFS) (chi tiết được miêu tả bên dưới). Sử dụng MCFS, thuật toán tính toán sự tương đồng giữa hàm mục tiêu và các hàm ứng cử viên, tạo ra các điểm khớp mới. Sau đó, nó sắp xếp lại tất cả các hàm ứng cử viên bằng cách kết hợp điểm Asteria ban đầu (Công thức 5) với điểm khớp được tính toán mới. Chi tiết về MCFS và tính toán điểm khớp được miêu tả trong các phần tiếp theo.
* �2: Khi hàm nguồn �1 không có hàm được gọi lại, thuật toán loại bỏ tất cả các hàm ứng cử viên có một hoặc nhiều hàm được gọi lại. Sau đó, các hàm ứng cử viên còn lại được sắp xếp lại dựa trên điểm Asteria ban đầu của chúng.

7.2.1 Mixed Callee Function Set.

* Tập hợp hàm được gọi lại kết hợp (MCFS) của hàm � bao gồm hai loại hàm được gọi lại: hàm được gọi lại có tên và hàm được gọi lại ẩn danh. Hàm được gọi lại có tên đề cập đến các hàm có tên được bảo tồn. Những hàm này thường là các hàm được nhập khẩu hoặc xuất khẩu, và tên hàm của chúng cần thiết cho mục đích liên kết bên ngoài. Trong khi đó, hàm được gọi lại ẩn danh là loại hàm mà tên hàm đã được loại bỏ vì lý do bảo mật. Những hàm này thường được ẩn danh để bảo vệ thông tin nhạy cảm. Chúng ta ký hiệu MCFS của hàm � là ��� = � � �1 , ...,�� ��,�� �1 , ...,�� �� , trong đó � � �� đại diện cho một hàm được gọi lại có tên và � � �� đại diện cho một hàm được gọi lại ẩn danh. Tập hợp ��� bao gồm cả hai loại hàm được gọi lại cho hàm � .