VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary.

VulSeeker: Một Công cụ Tìm lỗ hổng dựa trên Học ngữ nghĩa cho Binary Đa Nền tảng.

TÓM TẮT

* Việc tái sử dụng mã cải thiện hiệu suất phát triển phần mềm, tuy nhiên, lỗ hổng có thể được đưa vào một cách không cố ý. Nhiều công trình hiện có tính toán sự tương đồng mã dựa trên đồ thị dòng điều khiển (CFGs) để xác định xem một hàm nhị phân chứa một lỗ hổng đã biết hay không. Thật không may, hiệu suất của họ trong việc tìm kiếm mã nhị phân đa nền tảng bị thách thức. Bài báo này giới thiệu VulSeeker, một công cụ tìm lỗ hổng dựa trên học ngữ nghĩa cho mã nhị phân đa nền tảng. Cho một hàm mục tiêu và một hàm có lỗ hổng, VulSeeker trước tiên xây dựng các đồ thị luồng ngữ nghĩa có nhãn và trích xuất các đặc trưng khối cơ bản dưới dạng vectơ số học cho cả hai hàm. Sau đó, vectơ nhúng của toàn bộ hàm nhị phân được tạo ra bằng cách đưa các vectơ số học của các khối cơ bản vào mô hình DNN (Mạng nơ-ron sâu) có kiến thức ngữ nghĩa tùy chỉnh. Cuối cùng, sự tương đồng của hai hàm nhị phân được đo bằng cách dựa trên khoảng cách Cosine. Kết quả thực nghiệm cho thấy VulSeeker vượt trội hơn các phương pháp tiếp cận hàng đầu về độ chính xác. Ví dụ, so với công trình gần đây nhất và có liên quan là Gemini, VulSeeker tìm thấy 50,00% lỗ hổng nhiều hơn trong 10 ứng viên hàng đầu và 13,89% nhiều hơn trong 50 ứng viên hàng đầu, và cải thiện các giá trị AUC và ACC lần lượt lần lượt 8,23% và 12,14%. Video được giới thiệu tại <https://youtu.be/Mw0mr84gpI8>.

1 GIỚI THIỆU

* Nhằm cải thiện hiệu suất của quá trình phát triển phần mềm, việc sao chép và tái sử dụng mã code của bên thứ ba là phổ biến. Nếu quá trình phát triển này được quản lý kém, mã code có lỗ hổng chưa được vá có thể dễ dàng được lan truyền đến các chương trình phần mềm khác. Với sự phổ biến của các thiết bị di động, các chương trình phần mềm trên kiến trúc truyền thống x86 dần được biên dịch và chuyển sang các kiến trúc khác (ví dụ: ARM, MIPS). Do đó, ngày càng có nhiều chương trình nhị phân chứa một số lượng lớn mã code có lỗ hổng tương tự hoặc giống nhau. Ví dụ, đã xác nhận 145 lỗ hổng trùng lặp chưa được vá trong hệ thống Debian [7].
* Nhiều công trình đã được trình bày để thực hiện tìm kiếm lỗ hổng cho các tệp nhị phân chạy trên nhiều nền tảng dựa trên các kỹ thuật sao chép mã [3, 5, 6, 9, 14] hoặc kiểm thử mờ mờ [8, 13]. Hầu hết các kỹ thuật dựa trên sao chép mã phân tích đồ thị luồng điều khiển (CFG) để xác định liệu một tệp nhị phân có chứa lỗ hổng hay không [3, 5, 6, 14]. Chúng hoạt động tốt trên cài đặt của họ, nhưng độ chính xác và hiệu suất có thể giảm đi khi áp dụng cho các tệp nhị phân lớn.
* Trong bài báo này, chúng tôi giới thiệu VulSeeker1, một công cụ tìm kiếm lỗ hổng dựa trên học ngữ nghĩa cho các tệp nhị phân chạy trên nhiều nền tảng. VulSeeker đạt được độ chính xác và hiệu suất cao hơn thông qua việc xây dựng đồ thị luồng điều khiển ngữ nghĩa được gán nhãn (LSFG) và quá trình tạo ra các ngữ nghĩa chức năng dựa trên mạng nơ-ron sâu nhận thức về ngữ nghĩa. LSFG bao gồm cả CFG và DFG (đồ thị luồng dữ liệu), do đó nắm bắt được nhiều thông tin ngữ nghĩa của một chức năng nhị phân hơn so với việc sử dụng chỉ CFG một mình. Mô hình DNN nhận thức ngữ nghĩa biến đổi các đặc trưng số học của các khối cơ bản trong chức năng thành ngữ nghĩa chức năng (hoặc vectơ nhúng). Lỗ hổng được xác định bằng cách đo sự tương đồng của hai chức năng nhị phân dựa trên khoảng cách Cosine của các vectơ nhúng của chúng.
* Để đánh giá, chúng tôi so sánh VulSeeker với phương pháp tìm kiếm lỗ hổng sao chép nhị phân chạy trên nhiều nền tảng hàng đầu trên một số bộ kiểm tra bên thứ ba phổ biến bao gồm các ứng dụng thực tế. Kết quả thử nghiệm cho thấy VulSeeker vượt trội so với công trình mới nhất và liên quan Gemini [14]. Trung bình, về phát hiện sao chép, AUC và ACC của VulSeeker lần lượt là 88,49% và 81,3%, cao hơn 8,23% và 12,14% so với Gemini. Hơn nữa, chúng tôi sử dụng lỗ hổng CVE-2015-1791 để đánh giá khả năng tìm kiếm lỗ hổng trong 4643 hình ảnh firmware. Trong 10 kết quả giống nhau nhất và 50 kết quả giống nhau nhất, VulSeeker đã tìm thấy 50,00% và 13,89% lỗ hổng thực tế hơn so với Gemini.

A computer screen shot of a diagram

Description automatically generated

2 CÔNG TRÌNH LIÊN QUAN

* Phát Hiện Sao Chép Nhị Phân. COP [9] là một công cụ phát hiện đạo văn sử dụng cú pháp chương trình kết hợp với so khớp mờ dựa trên dãy con chung dài nhất. BinGold [2] trích xuất ngữ nghĩa của mã nhị phân dưới dạng cả DFG và CFG, và tổng hợp chúng thành một biểu diễn mới gọi là đồ thị luồng ngữ nghĩa. Tuy nhiên, nó không hỗ trợ việc phát hiện sao chép giữa các kiến trúc khác nhau. BinSim [11] tính toán các tương đương của các cuộc gọi hệ thống được căn chỉnh để xử lý tối ưu mã. Nó kết hợp phân rã động với tính toán điều kiện yếu nhất để xác định các tương tự ngữ nghĩa tinh vi giữa hai lần chạy thực thi.
* Tìm Kiếm Lỗ Hổng. Bingo [3] sử dụng việc ghép thể hiện lựa chọn và theo dõi một phần dấu vết biến đổi chiều dài để tính toán ngữ nghĩa chức năng, tạo thành các mô hình chức năng để thực hiện so sánh tương đồng và tìm kiếm lỗ hổng. Genius [6] sử dụng phân cụm phổ để tạo ra một bộ từ điển mã nguồn và tính toán sự tương đồng giữa một ACFG cụ thể và mỗi ACFG đại diện trong bộ từ điển dựa trên thuật toán khớp đồ thị hai bên. Gemini [14] trích xuất các đặc trưng nhẹ cùng mạng nơ-ron sâu đối với Genius và chỉ dựa vào CFG để tạo ra vectơ nhúng của chức năng. Sau đó, đo lường sự tương đồng của hai vectơ nhúng để có kết quả dự đoán.
* Khác Biệt Chính. Khác với những công trình trên, cho đến hiện tại, VulSeeker là công cụ đầu tiên kết hợp CFG và DFG để tạo thành LSFG, và sử dụng học sâu để tìm kiếm lỗ hổng trong tệp nhị phân chạy trên nhiều nền tảng. Nó trích xuất 8 loại đặc trưng chỉ thị nhẹ cho mỗi khối cơ bản trong LSFG. Dựa vào đồ thị và mô hình DNN nhận thức ngữ nghĩa đã được điều chỉnh, chúng tôi áp dụng 6 lần lặp lại lên LSFG để thu được biểu diễn ngữ nghĩa của toàn bộ chức năng nhị phân và đạt được độ chính xác cao hơn.

3 THIẾT KẾ VULSEEKER

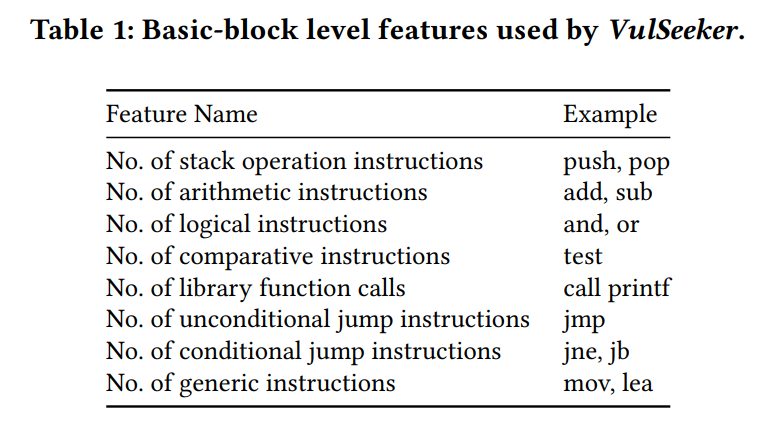
* Công việc tổng thể của VulSeeker được thể hiện trong Hình 1(a). Nó bao gồm bốn thành phần chính: xây dựng LSFG, trích xuất đặc trưng khối, tạo ra ngữ nghĩa chức năng và tính toán sự tương đồng. Mục tiêu của VulSeeker là xác định xem tệp nhị phân mục tiêu có chứa các chức năng tương tự với các lỗ hổng đã biết hay không. Do đó, đầu vào của nó là hai chức năng nhị phân từ tệp nhị phân mục tiêu và cơ sở dữ liệu lỗ hổng. Đầu tiên, VulSeeker xây dựng LSFG cho hai chức năng nhị phân. Sau đó, nó trích xuất 8 loại đặc trưng chỉ thị nhẹ và mã hóa chúng thành một vectơ số học cho mỗi khối cơ bản của LSFG. Ngữ nghĩa chức năng được tạo ra bằng cách đưa các vectơ số học của các khối cơ bản trong LSFG vào mô hình mạng nơ-ron sâu nhận thức ngữ nghĩa (DNN). Cuối cùng, VulSeeker xuất kết quả xem chức năng nhị phân mục tiêu có chứa lỗ hổng đã biết hay không dựa trên sự tương đồng của các vectơ nhúng của hai chức năng đầu vào.

3.1 Xây Dựng LSFG

* Labeled semantic flow graph chứa cả CFG và đồ thị luồng dữ liệu (DFG), và các cạnh của nó được đánh dấu là 0 và 1 tương ứng. Mục đích của LSFG là cải thiện độ chính xác của việc tạo ra ngữ nghĩa chức năng, vì nó kết hợp cả cấu trúc điều khiển và truyền dữ liệu trong một chức năng, từ đó hiệu quả giảm thiểu sự can thiệp cấu trúc được đưa ra bởi các CFG khác nhau dưới các nền tảng khác nhau. Hình 1(b) mô tả một ví dụ về LSFG.
* Chúng tôi sử dụng IDAPython được cung cấp bởi IDA Pro [12] để tạo ra CFG cho các khối cơ bản của mỗi chức năng nhị phân. Dựa vào CFG, chúng tôi suy ra xem có nên có một cạnh phụ thuộc dữ liệu giữa hai khối cơ bản bằng cách sử dụng plugin LLVM IR [10] trên IDA Pro. Đối với hai chỉ thị i và j từ hai khối cơ bản khác nhau đáp ứng cấu trúc CFG, nếu chỉ thị i ghi một vị trí bộ nhớ và chỉ thị j đọc địa chỉ bộ nhớ cùng một, chúng tôi tạo ra một cạnh phụ thuộc dữ liệu cho hai khối này. Bên cạnh đó, chỉ các phụ thuộc dữ liệu giữa các khối khác nhau được lưu giữ, và có tối đa một cạnh phụ thuộc dữ liệu giữa hai khối cơ bản. VulSeeker lưu trữ các cạnh điều khiển và cạnh dữ liệu của mỗi chức năng trong hai tệp riêng biệt.

3.2 Trích xuất Đặc trưng Khối

* Bằng cách tham khảo các đặc trưng được sử dụng trong các công trình trước [2, 6] và thực hiện một loạt thí nghiệm sao chép mã cho các bộ đặc trưng khác nhau, chúng tôi đã xác định cuối cùng sử dụng 8 loại đặc trưng được hiển thị trong Bảng 1 làm biểu diễn ngữ nghĩa ban đầu của mỗi khối cơ bản. Các đặc trưng được lựa chọn này nhẹ và đáng tin cậy, có thể được trích xuất dễ dàng và thay đổi ít dưới các nền tảng thực hiện khác nhau với các kiến trúc vi xử lý và các cấu hình tối ưu hóa biên dịch khác nhau. Chúng tôi sử dụng IDAPython để trích xuất đặc trưng cho mỗi khối cơ bản. Sau đó, chúng tôi mã hóa 8 đặc trưng của mỗi khối cơ bản thành một vectơ số học. Hình 1(c) là các vectơ số học của mỗi khối cơ bản tương ứng với chức năng trong Hình 1(b). Đối với mỗi chức năng nhị phân, các vectơ số học của tất cả các khối cơ bản trong chức năng được lưu trữ trong một tệp riêng biệt.



3.3 Tạo ra Ngữ nghĩa Chức năng

* Đầu vào của thành phần này là các vectơ số học ban đầu có kích thước d của tất cả các khối cơ bản trong chức năng, và đầu ra là vectơ nhúng có kích thước p biểu diễn ngữ nghĩa chức năng. Để chính xác nắm bắt ngữ nghĩa chức năng, cần xem xét các phụ thuộc dữ liệu và kiểm soát giữa các khối cơ bản theo đồ thị LSFG. Tham khảo mạng nơ-ron cấu trúc [4], chúng tôi đề xuất một mô hình mạng nơ-ron sâu nhận thức ngữ nghĩa được hiển thị trong Hình 2 chuyên xử lý biểu diễn LSFG có cấu trúc.

A diagram of a diagram of a structure

Description automatically generated

Hình 2(a) là một LSFG được ký hiệu là д = ⟨V, E⟩, chứa ba đỉnh với các vectơ số học ban đầu: x1, x2, x3, trong đó V và E biểu thị tập hợp các đỉnh và tập hợp các cạnh, tương ứng. Các cạnh được đánh dấu là 1 và 0 cho biết sự phụ thuộc dữ liệu và sự phụ thuộc kiểm soát, tương ứng. Mô hình DNN chứa tổng cộng T lớp lặp lại, và mỗi lần lặp lại biến đổi vectơ số học ban đầu xi của mỗi đỉnh i thành vectơ nhúng của nó eµ (t) i . Sau khi có được các vectơ nhúng của tất cả các đỉnh khối cơ bản trong chức năng, chúng tôi tổng hợp chúng thành vectơ nhúng có kích thước p của chức năng với công thức eµ = W2(Σi∈V eµ (T) i ), trong đó W2 là ma trận tham số có kích thước p × p.

Hình 2(b) minh họa sơ đồ phát sinh vectơ nhúng eµ (t) i của mỗi đỉnh i trong suốt lượt lặp t lớp. Đầu vào của quá trình biến đổi bao gồm ba phần khác nhau: vectơ số học ban đầu xi của đỉnh tương ứng i (mũi tên chấm trong Hình 2(a-b)), tổng các vectơ nhúng trước đó của các đỉnh trỏ tới đỉnh i qua sự phụ thuộc kiểm soát (được ký hiệu là C(i)), và tổng các vectơ nhúng trước đó của các đỉnh trỏ tới đỉnh i qua sự phụ thuộc dữ liệu (được ký hiệu là D(i)). Vectơ nhúng của đỉnh i được tính thông qua công thức eµ (t) i = tanh(W1xi + σc (Σj∈C(i)eµ (t−1) j ) + σd (Σj∈D(i)eµ (t−1) j )), trong đó W1 là ma trận tham số có kích thước d × p. σc và σd là hai mạng nơ-ron kết nối đầy đủ n lớp chịu trách nhiệm tính toán vectơ nhúng với khả năng biểu diễn mạnh mẽ hơn, chúng được biểu diễn như sau:

A group of black letters

Description automatically generated

* Trong đó, n là độ sâu nhúng của mỗi đỉnh, Pi và Qi là các ma trận tham số có kích thước p × p. Thông qua T lớp lặp lại, đặc trưng của mỗi đỉnh được truyền cho các đỉnh khác khi tiến hành lần lặp theo đồ thị LSFG, đảm bảo rằng mỗi khối cơ bản của chức năng có ngữ nghĩa bối cảnh tương ứng.
* Sau khi có được vectơ nhúng eµ cho chức năng mục tiêu và vectơ nhúng eν cho chức năng có lỗ hổng, VulSeeker tính toán sự tương đồng giữa chúng bằng hàm Cosine yb = cos(eµ, eν) = (eµ · eν) / (||eµ|| ||eν||), trong đó yb là điểm tương đồng, nằm trong khoảng từ -1 đến 1. Nếu điểm tương đồng yb lớn hơn một ngưỡng được xác định trước, chức năng nhị phân mục tiêu được coi là tương tự với lỗ hổng. Chúng tôi sử dụng TensorFlow [1] để triển khai mô hình mạng nơ-ron nhận thức ngữ nghĩa và áp dụng thuật toán gradient descent ngẫu nhiên để tự động học các tham số mô hình, chẳng hạn như W1, W2, P1 và Q1.

4. Kết quả thử nghiệm

* VulSeeker chủ yếu bao gồm 3 tập tin thực thi có thể được sử dụng dựa trên các bước sau: 1) chỉnh sửa tệp config.py để cấu hình các chương trình mục tiêu để tìm lỗ hổng; 2) thực thi tệp command.py để tạo các LSFGs và trích xuất các đặc trưng số học ban đầu cho các khối cơ bản; 3) thực thi tệp search\_by\_list\_vulseeker.py để có được các vector nhúng của các hàm và nhận danh sách các hàm theo thứ tự giảm dần của điểm tương đồng.
* VulSeeker được đánh giá trên hai tập dữ liệu. Tập dữ liệu I chứa 735.540 hàm với 9.345K khối cơ bản. Chúng tôi biên dịch BusyBox (v1.21.0), OpenSSL (v1.0.1f và v1.0.1u) và Coreutils (v6.5 và v6.7) trên các kiến trúc X86, X64, MIPS32, MIPS64, ARM32, ARM64, sử dụng GCC (v4.9 và v5.5) với các cấp độ tối ưu hóa O0-O3. Tập dữ liệu II bao gồm 4643 hình ảnh firmware cho các kiến trúc khác nhau từ [6]. Tất cả các thử nghiệm được thực hiện trên cấu hình mặc định của mô hình DNN như sau: độ sâu nhúng n là 2, kích thước nhúng p là 64, số lần lặp T cho mỗi khối cơ bản là 6, số lượng epoch huấn luyện là 100.
* Độ chính xác của Code Clone. Chúng tôi xem xét hai phiên bản biên dịch khác nhau của cùng một hàm nguồn như một cặp hàm tương tự và ngược lại. Chúng tôi ngẫu nhiên chọn 2500 cặp hàm tương tự và 2500 cặp hàm không tương tự từ tập dữ liệu I để thực hiện thử nghiệm so sánh với Gemini được cấu hình với các tham số tối ưu [14]. Hình 3 là đồ thị ROC (đặc trưng hoạt động của người nhận) của mã sao chép. Chúng tôi quan sát rằng đường cong ROC của VulSeeker nằm trên Gemini, điều này có nghĩa là VulSeeker có thể đạt được tỷ lệ true positive cao hơn ở cùng tỷ lệ false positive. Giá trị AUC và giá trị ACC của VulSeeker lần lượt là 88,49% và 81,3%, cao hơn 8,23% và 12,14% so với Gemini. Tóm lại, VulSeeker vượt trội hơn Gemini, bởi vì ngoài CFG, chúng tôi còn xây dựng DFG để theo dõi việc sử dụng các biến giữa các khối cơ bản. Trong quá trình tạo ngữ nghĩa của hàm, chúng tôi thu được thông tin ngữ nghĩa mạnh mẽ hơn với một cấu trúc DNN đã được sửa đổi, điều này có lợi cho việc nhận dạng hiệu quả của các hàm trùng lặp.

A graph of a number of lines

Description automatically generated

* Kết quả thử nghiệm của Gemini ở đây thấp hơn so với mô tả trong tài liệu [14]. Hai lý do dẫn đến tình trạng này. Thứ nhất, tập dữ liệu I của chúng tôi chứa 5 chương trình, nhưng Gemini chỉ chứa 2 trong số đó. Thứ hai, chúng tôi biên dịch các chương trình này thành sáu kiến trúc, bao gồm ba kiến trúc 64-bit, trong khi Gemini chỉ biên dịch chúng thành ba kiến trúc 32-bit. Số lượng thanh ghi tổng quát được sử dụng trong các kiến trúc 32-bit và 64-bit khác nhau, điều này ảnh hưởng đến các vector đặc trưng của các khối cơ bản, và kết quả của nó giảm đi tương ứng. Trong cài đặt tương đối đơn giản của họ, họ có thể đạt được giá trị AUC là 97%, và chúng tôi có thể cải thiện giá trị này lên gần 99%. Chúng tôi nhận thấy rằng càng phức tạp hơn tập dữ liệu, VulSeeker sẽ đạt được nhiều cải tiến hơn và hiệu suất của Gemini giảm nhanh hơn so với VulSeeker đối với các cài đặt phức tạp.
* Độ chính xác của Tìm kiếm Lỗ hổng. Chúng tôi sử dụng tập dữ liệu II để đánh giá hiệu quả của VulSeeker và Gemini trong tìm kiếm lỗ hổng. Chúng tôi lấy CVE-2015-1791 với 48 phiên bản biên dịch làm hàm có lỗ hổng. Đối với mỗi phiên bản của lỗ hổng, chúng tôi sử dụng hai công cụ để thực hiện nhiệm vụ tìm kiếm từ các hình ảnh firmware đã biết có lỗ hổng. Đối với mỗi hình ảnh firmware, chúng tôi sắp xếp các hàm theo thứ tự giảm dần của điểm tương đồng trung bình cho 48 lần tìm kiếm. VulSeeker trung bình xếp hạng hàm có lỗ hổng là thứ 8, trong khi Gemini trung bình xếp hạng là thứ 99. Nếu chúng tôi lấy xếp hạng cao nhất trong 48 lần tìm kiếm cho mỗi hình ảnh firmware, VulSeeker có 100% cơ hội tìm thấy hàm có lỗ hổng trong top 3 ứng viên, cao hơn 11,76% so với Gemini.

A table with numbers and text

Description automatically generated

* Kết quả tìm kiếm lỗ hổng phiên bản MIPS32, chúng tôi sắp xếp các hàm trong tất cả các hình ảnh firmware theo thứ tự giảm dần của điểm tương đồng. Bảng 2 thể hiện hiệu quả của tìm kiếm lỗ hổng trên các kết quả tương đồng hàng đầu (top-K) trong tất cả các hàm. Cột 1 là các giá trị K khác nhau. Các cột 2 và 4 là số lượng lỗ hổng thực tế trong các kết quả top-K, và các cột 3 và 5 là phần trăm của các lỗ hổng thực tế tương ứng. Chúng ta có thể thấy rằng VulSeeker có sự cải thiện đáng kể về độ chính xác tìm kiếm so với Gemini. Trong top-10 kết quả, VulSeeker tìm thấy 50% lỗ hổng nhiều hơn so với Gemini. Tóm lại, VulSeeker vượt trội hơn Gemini trong việc tìm kiếm lỗ hổng liên quan đến CVE-2015-1791. Chúng tôi cũng tiến hành đánh giá trên một số CVE khác, và hiệu suất cải tiến vẫn được duy trì.
* Chi phí thời gian. VulSeeker chủ yếu bao gồm bốn thành phần, và đối với kích thước của cơ sở dữ liệu thử nghiệm, nó có thể xác định xem chức năng nhị phân đã cho có chứa lỗ hổng đã biết trong trung bình 0,20 giây, trong khi Gemini mất khoảng 0,15 giây. Việc tạo ngữ nghĩa của hàm chiếm gần một nửa thời gian, và thời gian tạo ngữ nghĩa tăng tuyến tính theo số lượng khối cơ bản trong hàm. Mặc dù thời gian thực hiện của VulSeeker lâu hơn 0,05 giây so với Gemini, chúng ta có thể đạt được độ chính xác tìm kiếm cao hơn trong một thời gian hợp lý.

5 KẾT LUẬN:

* Trong bài báo này, chúng tôi giới thiệu VulSeeker, một công cụ tìm kiếm lỗ hổng nhị phân đa nền tảng dựa trên học ngữ nghĩa. Bằng cách tích hợp CFG và DFG của hàm nhị phân, chúng tôi thu thập nhiều ngữ nghĩa hàm hơn. Kết quả thử nghiệm cho thấy VulSeeker đạt được giá trị AUC là 88,49% và giá trị ACC là 81,3% cho mã sao chép, cải thiện lần lượt 8,23% và 12,14% so với Gemini. Trong nghiên cứu trường hợp về tìm kiếm lỗ hổng CVE-2015-1791, VulSeeker tìm thấy 50,00% lỗ hổng nhiều hơn trong 10 ứng viên hàng đầu và 13,89% lỗ hổng nhiều hơn trong 50 ứng viên hàng đầu. Đối với thời gian tìm kiếm lỗ hổng, VulSeeker cần 0,20 giây để xác định xem một hàm có chứa lỗ hổng đã biết hay không trong cơ sở dữ liệu tương đối lớn. Các kết quả này cho thấy VulSeeker phù hợp cho việc tìm kiếm lỗ hổng trong mã lệnh quy mô lớn.