Modeling and Discovering Vulnerabilities with Code Property Graphs

Fabian Yamaguchi∗, Nico Golde†, Daniel Arp∗ and Konrad Rieck∗ ∗University of Gottingen, Germany ¨
†Qualcomm Research Germany

ABSTRACT

Đại đa số các cuộc tấn công an ninh gặp phải ngày nay đều là kết quả trực tiếp của mã nguồn không an toàn. Do đó, bảo vệ hệ thống máy tính quan trọng phụ thuộc nghiêm trọng vào việc xác định nghiêm túc các lỗ hổng trong phần mềm, một quá trình tẻ nhạt và dễ mắc lỗi đòi hỏi sự chuyên môn đáng kể. Thật không may, chỉ cần một lỗ hổng duy nhất cũng đủ để làm suy yếu bảo mật của hệ thống và do đó, lượng mã nguồn cần phải kiểm tra đóng vai trò quan trọng trong tay của kẻ tấn công. Trong bài báo này, chúng tôi giới thiệu một phương pháp để khai thác hiệu quả lượng lớn mã nguồn để tìm ra các lỗ hổng. Để đạt được điều này, chúng tôi giới thiệu một biểu diễn mới của mã nguồn gọi là đồ thị thuộc tính mã nguồn (code property graph) kết hợp các khái niệm của phân tích chương trình cổ điển, bao gồm cây cú pháp trừu tượng (abstract syntax trees), đồ thị luồng điều khiển (control flow graphs) và đồ thị phụ thuộc chương trình (program dependence graphs) vào một cấu trúc dữ liệu chung. Biểu diễn toàn diện này cho phép chúng tôi mô hình hóa một cách tinh tế các mẫu lỗ hổng phổ biến bằng cách duyệt qua đồ thị, ví dụ như nhận diện tràn bộ đệm (buffer overflows), tràn số nguyên (integer overflows), lỗ hổng chuỗi định dạng (format string vulnerabilities) hoặc tiết lộ bộ nhớ (memory disclosures). Chúng tôi triển khai phương pháp của mình bằng cách sử dụng cơ sở dữ liệu đồ thị phổ biến và chứng minh hiệu quả của nó bằng cách phát hiện 18 lỗ hổng trước đó chưa được biết đến trong mã nguồn của hạt nhân Linux.

I. INTRODUCTION

Bảo mật của các hệ thống máy tính hoàn toàn phụ thuộc vào chất lượng của phần mềm cơ bản. Mặc dù đã có một chuỗi nghiên cứu dài trong cả học thuật và công nghiệp, những lỗ hổng bảo mật thường xuyên xuất hiện trong mã chương trình, ví dụ như việc không xử lý đúng ranh giới bộ đệm (buffer boundaries) hoặc không xác thực đầy đủ dữ liệu đầu vào. Do đó, lỗ hổng trong phần mềm vẫn là một trong những nguyên nhân chính dẫn đến các vụ vi phạm bảo mật ngày nay. Ví dụ, vào năm 2013, một lỗi tràn bộ đệm duy nhất trong thư viện plug-and-play đa năng đã làm cho hơn 23 triệu router trở nên dễ bị tấn công từ Internet [26]. Tương tự, hàng ngàn người dùng hiện nay trở thành nạn nhân của phần mềm độc hại dựa trên web, tận dụng các lỗ hổng khác trong môi trường thực thi Java [29].

Phát hiện các lỗ hổng phần mềm là một vấn đề bảo mật cổ điển nhưng đầy thách thức. Do khả năng của chương trình không thể nhận diện được các thuộc tính phức tạp của một chương trình khác, vấn đề chung của việc tìm kiếm lỗ hổng phần mềm là không thể giải quyết được [33]. Do đó, các phương pháp hiện tại để phát hiện lỗ hổng bảo mật thường chỉ giới hạn trong các loại lỗ hổng cụ thể hoặc dựa vào việc kiểm tra thủ công mệt nhọc. Đặc biệt, bảo vệ các dự án phần mềm lớn như hệ điều hành kernel là một công việc đáng sợ, vì một lỗi duy nhất có thể đe dọa bảo mật

của toàn bộ mã nguồn. Mặc dù một số lớp lỗ hổng tái xuất hiện khắp cảnh cảnh phần mềm trong một thời gian dài, chẳng hạn như lỗ hổng tràn bộ đệm và lỗ hổng chuỗi định dạng, việc phát hiện chúng một cách tự động là một thách thức.

Do tình hình này, nghiên cứu bảo mật ban đầu tập trung vào việc tìm ra tĩnh các loại lỗ hổng cụ thể, chẳng hạn như lỗi do các hàm thư viện không an toàn gây ra [6], tràn bộ đệm [45], tràn số nguyên [40] hoặc xác thực không đủ cho dữ liệu đầu vào [18]. Dựa trên các khái niệm từ kiểm thử phần mềm, phát hiện lỗ hổng rộng hơn đã được đạt được bằng phân tích chương trình động, từ kiểm tra mờ [ví dụ như 38, 42] đến theo dõi vết bẩn và thực thi biểu tượng tiên tiến [ví dụ như 2, 35]. Mặc dù các phương pháp này có thể phát hiện ra các loại lỗi khác nhau, nhưng chúng khó có thể hoạt động hiệu quả trong thực tế và thường không cung cấp kết quả phù hợp do thời gian chạy cấm hoặc sự gia tăng mũi đường thực thi [16, 21]. Như một biện pháp khắc phục, nghiên cứu bảo mật gần đây đã bắt đầu khám phá các phương pháp hỗ trợ người phân tích trong quá trình kiểm tra hơn là thay thế họ. Các phương pháp đề xuất tăng tốc quá trình kiểm tra bằng cách bổ sung phân tích chương trình tĩnh với kiến thức chuyên gia và do đó có thể hướng dẫn tìm kiếm lỗ hổng [ví dụ như 39, 43, 44].

Trong bài báo này, chúng tôi tiếp tục hướng nghiên cứu này và trình bày một phương pháp mới để khai thác lượng lớn mã nguồn để tìm lỗ hổng. Phương pháp của chúng tôi kết hợp các khái niệm cổ điển của phân tích chương trình với các phát triển mới nhất trong lĩnh vực khai thác đồ thị. Cái nhìn chính đằng sau phương pháp của chúng tôi là nhiều lỗ hổng chỉ có thể được phát hiện một cách thích hợp bằng cách kết hợp cấu trúc, luồng điều khiển và phụ thuộc của mã chương trình. Chúng tôi giải quyết yêu cầu này bằng cách giới thiệu một biểu diễn mới của mã nguồn được gọi là đồ thị thuộc tính mã nguồn. Đồ thị này kết hợp các thuộc tính của cây cú pháp trừu tượng, đồ thị luồng điều khiển và đồ thị phụ thuộc chương trình trong một cấu trúc dữ liệu chung. Quan điểm toàn diện này về mã cho phép chúng tôi mô hình hóa mẫu cho các lỗ hổng phổ biến bằng cách sử dụng các duyệt đồ thị. Tương tự như truy vấn trong cơ sở dữ liệu, một duyệt đồ thị đi qua đồ thị thuộc tính mã nguồn và kiểm tra cấu trúc mã, luồng điều khiển và các phụ thuộc dữ liệu liên quan với mỗi nút. Sự truy cập chung này vào các thuộc tính mã khác nhau cho phép tạo ra các mẫu rõ ràng cho một số loại lỗi và do đó giúp kiểm tra lượng lớn mã nguồn để tìm lỗ hổng.

Chúng tôi triển khai phương pháp của mình bằng cách sử dụng một cơ sở dữ liệu đồ thị phổ biến và minh họa các lợi ích thực tiễn bằng cách thiết kế các duyệt đồ thị cho một số loại lỗ hổng nổi tiếng, như tràn bộ đệm, tràn số nguyên, lỗ hổng chuỗi định dạng hoặc tiết lộ bộ nhớ. Như một ví dụ, chúng tôi phân tích mã nguồn của hạt nhân Linux - một mã nguồn lớn và đã được kiểm tra. Chúng tôi phát hiện rằng gần như tất cả các lỗ hổng được báo cáo cho nhân Linux vào năm 2012 có thể được mô tả bằng các duyệt đồ thị thuộc tính mã. Mặc dù cộng đồng mã nguồn mở đã nỗ lực đáng kể để cải thiện tính bảo mật của hạt nhân, những duyệt đồ thị này cho phép chúng tôi phát hiện ra 18 lỗ hổng trước đây chưa biết đến trong nhân Linux, từ đó chứng minh khả năng của đồ thị thuộc tính mã trong thực tế.

Tóm lại, chúng tôi đóng góp vào vấn đề phát hiện lỗ hổng bảo mật như sau:

Code property graph (đồ thị thuộc tính mã): Chúng tôi giới thiệu một biểu diễn mới của mã nguồn kết hợp các thuộc tính của cây cú pháp trừu tượng, đồ thị luồng điều khiển và đồ thị phụ thuộc chương trình trong một cấu trúc dữ liệu chung.

- Traversals for vulnerability types (duyệt đồ thị cho các loại lỗ hổng): Chúng tôi chỉ ra rằng các loại lỗ hổng phổ biến có thể được mô hình hóa một cách tinh tế thông qua duyệt đồ thị thuộc tính mã và tạo ra các mẫu phát hiện hiệu quả.
- Efficient implementation (triển khai hiệu quả): Chúng tôi chứng minh rằng bằng cách nhập đồ thị thuộc tính mã vào một cơ sở dữ liệu đồ thị, các duyệt đồ thị có thể được thực hiện một cách hiệu quả trên các dự án mã nguồn lớn như nhân Linux.

II. REPRESENTATIONS OF CODE

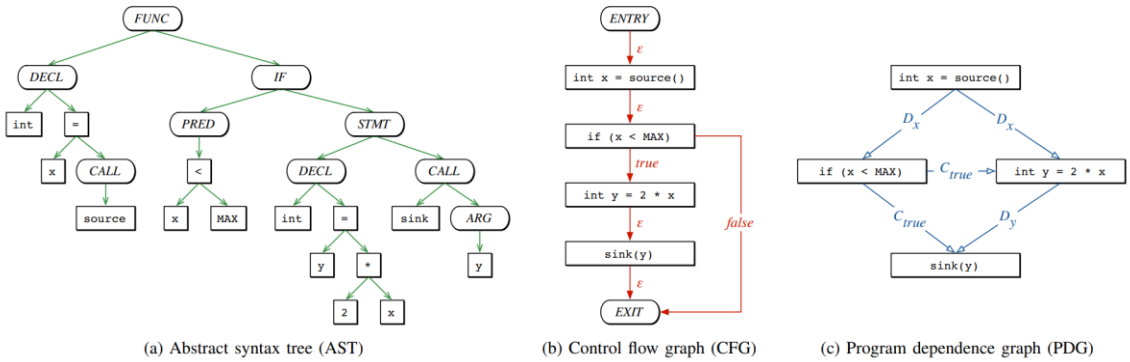
- Các biểu diễn khác nhau của mã đã được phát triển trong lĩnh vực phân tích chương trình và thiết kế trình biên dịch để xem xét các thuộc tính của các chương trình. Mặc dù những biểu diễn này được thiết kế chủ yếu để phân tích và tối ưu hóa mã, chúng cũng được sử dụng để mô tả mã như nghiên cứu trong công việc này. Đặc biệt, chúng tôi tập trung vào ba biểu diễn cổ điển, bao gồm cây cú pháp trừu tượng (abstract syntax trees), đồ thị luồng điều khiển (control flow graphs) và đồ thị phụ thuộc chương trình (program dependence graphs), là cơ sở cho phương pháp của chúng tôi trong phát hiện lỗ hổng. Một thảo luận chi tiết về các biểu diễn mã có thể được tìm thấy trong sách của Aho và cộng sự [1].

-

```
void foo()                          1
{                                   2
    int x = source();               3
    if (x < MAX)                    4
    {                               5
        int y = 2 * x;              6
        sink(y);                    7
    }                               8
}                                   9
```

Ảnh 1: Mẫu mã ví dụ.

Như một ví dụ đơn giản để minh họa các biểu diễn khác nhau và chạy qua phần này, chúng ta xem xét mẫu mã được hiển thị trong Hình 1.



(a) Abstract syntax tree (AST) (b) Control flow graph (CFG) (c) Program dependence graph (PDG)

Hình 2: Các biểu diễn mã cho ví dụ trong Hình 1. Các phụ thuộc điều khiển và dữ liệu được biểu thị bởi C và D trong đồ thị phụ thuộc chương trình.

A. Cây cú pháp trừu tượng (AST)

Cây cú pháp trừu tượng thường là một trong những biểu diễn trung gian đầu tiên được tạo ra bởi các trình phân tích mã của trình biên dịch và do đó là nền tảng cho việc tạo ra nhiều biểu diễn mã khác. Các cây này trung thành mã hóa cách câu lệnh và biểu thức được lồng vào nhau để tạo ra các chương trình. Tuy nhiên, khác với cây phân tích cú pháp, cây cú pháp trừu tượng không biểu thị cú pháp cụ thể được chọn để biểu diễn chương trình. Ví dụ, trong ngôn ngữ C, một danh sách các khai báo được phân cách bằng dấu phẩy thường sẽ tạo ra cùng một cây cú pháp trừu tượng như hai khai báo liên tiếp.

Cây cú pháp trừu tượng là các cây có thứ tự trong đó các nút bên trong biểu diễn các toán tử (ví dụ: phép cộng hay gán) và các nút lá tương ứng với các toán hạng (ví dụ: hằng số hoặc định danh). Ví dụ, hình 2a cho thấy một cây cú pháp trừu tượng cho mẫu mã được cho trong Hình 1. Mặc dù cây cú pháp trừu tượng rất thích hợp cho các biến đổi mã đơn giản và đã được sử dụng để nhận diện mã tương tự về mặt ngữ nghĩa [3, 43], nhưng chúng không áp dụng được cho phân tích mã phức tạp hơn, chẳng hạn như phát hiện mã chết hoặc biến chưa khởi tạo. Lý do cho sự thiếu sót này là biểu diễn này của mã không làm rõ luồng điều khiển hoặc phụ thuộc dữ liệu.

B. Đồ thị luồng điều khiển (CFG)

Đồ thị luồng điều khiển mô tả rõ ràng thứ tự thực hiện các câu lệnh mã cũng như các điều kiện cần phải đáp ứng để một đường đi thực thi cụ thể được chọn. Để làm được điều này, các câu lệnh và biểu thức điều kiện được biểu diễn bằng các nút, được kết nối bởi các cạnh có hướng để chỉ ra sự chuyển giao điều khiển. Mặc dù các cạnh này không cần phải được sắp xếp như trong trường hợp của cây cú pháp trừu tượng, nhưng cần phải gán một nhãn true, false hoặc ϵ cho mỗi cạnh. Cụ thể, một nút câu lệnh có một cạnh đi ra được gán nhãn ϵ, trong khi một nút điều kiện có hai cạnh đi ra tương ứng với một đánh giá true hoặc false của điều kiện. Đồ thị luồng điều khiển có thể được xây dựng từ cây cú pháp trừu tượng trong hai bước:

- Đầu tiên, các câu lệnh điều khiển có cấu trúc (ví dụ: if, while, for) được xem xét để xây dựng một đồ thị luồng điều khiển sơ bộ.
- Thứ hai, đồ thị luồng điều khiển sơ bộ được chỉnh sửa bằng cách xem xét thêm các câu lệnh điều khiển không có cấu trúc như goto, break và continue. Hình 2b cho thấy CFG cho mẫu mã được cho trong Hình 1.

Đồ thị luồng điều khiển đã được sử dụng cho nhiều ứng dụng trong bối cảnh bảo mật, ví dụ như để phát hiện các biến thể của các ứng dụng độc hại đã biết [11] và để hướng dẫn các công cụ kiểm tra lỗi fuzz [37]. Hơn nữa, chúng đã trở thành một biểu diễn mã tiêu chuẩn trong kỹ thuật đảo ngược để hỗ trợ trong việc hiểu chương trình. Tuy nhiên, mặc dù đồ thị luồng điều khiển tiết lộ luồng điều khiển của ứng dụng, chúng không cung cấp thông tin về luồng dữ liệu. Đối với phân tích lỗ hổng đặc biệt, điều này có nghĩa là đồ thị luồng điều khiển không thể dễ dàng được sử dụng để xác định các câu lệnh xử lý dữ liệu bị ảnh hưởng bởi một kẻ tấn công.

C. Đồ thị phụ thuộc chương trình (PDG)

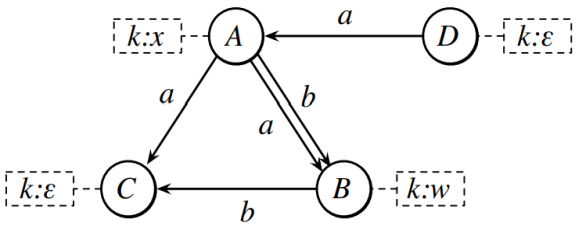
Đồ thị phụ thuộc chương trình, được giới thiệu bởi Ferrante và đồng nghiệp [9], ban đầu đã được phát triển để thực hiện phân đoạn chương trình [41], tức là xác định tất cả các câu lệnh và điều kiện của chương trình ảnh hưởng đến giá trị của một biến tại một câu lệnh cụ thể. Đồ thị phụ thuộc chương trình rõ ràng biểu diễn các phụ thuộc giữa các câu lệnh và điều kiện. Cụ thể, đồ thị được xây dựng bằng hai loại cạnh: các cạnh phụ thuộc dữ liệu phản ánh sự ảnh hưởng của một biến lên biến khác và các cạnh phụ thuộc điều khiển tương ứng với sự ảnh hưởng của các điều kiện lên các giá trị của biến. Các cạnh của đồ thị phụ thuộc chương trình có thể được tính từ đồ thị luồng điều khiển bằng cách đầu tiên xác định tập hợp các biến được định nghĩa và tập hợp các biến được sử dụng bởi mỗi câu lệnh, và tính toán các định nghĩa đạt được cho mỗi câu lệnh và điều kiện, là một vấn đề tiêu chuẩn trong thiết kế trình biên dịch [xem 1]. Như một ví dụ, Hình 2c cho thấy đồ thị phụ thuộc chương trình cho mẫu mã mã được cho trong Hình 1. Lưu ý rằng các cạnh phụ thuộc điều khiển không đơn giản là các cạnh luồng điều khiển và đặc biệt là thứ tự thực thi các câu lệnh không còn có thể được xác định từ đồ thị, trong khi các phụ thuộc giữa các câu lệnh và điều kiện được thể hiện rõ ràng.

III. PROPERTY GRAPHS AND TRAVERSAL

Mỗi biểu diễn đã trình bày cung cấp một góc nhìn độc đáo về mã nguồn, nhấn mạnh các khía cạnh khác nhau của chương trình cơ bản. Để kết hợp những góc nhìn này vào một biểu diễn chung cho việc phát hiện lỗ hổng bảo mật, chúng tôi sử dụng khái niệm đồ thị thuộc tính (property graph) [34], là một biểu diễn cơ bản của dữ liệu có cấu trúc trong nhiều cơ sở dữ liệu đồ thị, chẳng hạn như ArangoDB, Neo4J và OrientDB. Về mặt hình thức, một đồ thị thuộc tính được định nghĩa như sau.

Định nghĩa 1: Một đồ thị thuộc tính $G = (V, E, \lambda, \mu)$ là một đồ thị có hướng, gán nhãn cạnh, có thuộc tính và là đồ thị đa hướng, trong đó V là tập hợp các nút, $E \subseteq (V \times V)$ là tập hợp các cạnh có hướng, và $\lambda: E \rightarrow \Sigma$ là một hàm gán nhãn cạnh, gán một nhãn từ bảng chữ cái Σ cho mỗi cạnh. Các thuộc tính có thể được gán cho các cạnh và nút bởi $\mu: (V \cup E) \times K \rightarrow S$, trong đó K là tập hợp các khóa thuộc tính và S là tập hợp các giá trị thuộc tính.

Hình 3 minh họa một đồ thị thuộc tính đơn giản với bốn nút. Lưu ý rằng đồ thị thuộc tính là đồ thị đa hướng, do đó, hai nút có thể được kết nối bởi nhiều cạnh, chẳng hạn như các nút A và B trong Hình 3. Hơn nữa, trong ví dụ này, một thuộc tính với khóa $k \in K$ được gán cho mỗi nút, trong đó chỉ các nút A và B lưu trữ các giá trị thuộc tính từ tập hợp $S = \{x, w\}$.



Hình 3: Ví dụ về một đồ thị thuộc tính. Các thuộc tính được gán cho các nút được biểu thị bằng các đường nét đứt.

Công cụ chính để khai thác thông tin trong các đồ thị thuộc tính là các phép duyệt đồ thị (graph traversals), hay ngắn gọn là duyệt (traversals), được sử dụng để di chuyển dọc theo các cạnh của đồ thị tùy thuộc vào các nhãn và thuộc tính. Về mặt hình thức, một phép duyệt đồ thị được định nghĩa như sau.

Định nghĩa 2: Một phép duyệt là một hàm $T : P(V) \rightarrow P(V)$ ánh xạ một tập hợp các nút thành một tập hợp các nút khác theo một đồ thị thuộc tính G, trong đó P là tập hợp con của V.

Định nghĩa chung này cho phép kết hợp nhiều phép duyệt với nhau. Ví dụ, hai phép duyệt đồ thị T_0 và T_1 có thể được kết hợp với nhau thành $T_0 \circ T_1$ bằng cách sử dụng phép hợp hàm $\circ$. Dựa trên sự kết hợp này, chúng ta có thể định nghĩa một số phép duyệt cơ bản, những phép duyệt này làm cơ sở cho việc xây dựng tất cả các phép duyệt khác được thảo luận trong bài báo này. Chúng ta bắt đầu bằng cách định nghĩa một phép duyệt lọc đơn giản:

$$FILTER_p(X) = \{v \in X : p(v)\}$$

Phép duyệt này sẽ trả về tất cả các nút trong tập hợp X khớp với điều kiện Boolean p(v), ví dụ, bằng cách kiểm tra một thuộc tính nhất định.

Để di chuyển dọc theo các cạnh của đồ thị thuộc tính, chúng ta định nghĩa các phép duyệt các cạnh đi ra sau đây:

$$\mathrm{OUT}(X) = \bigcup_{v \in X} \{u : (v, u) \in E\}$$

$$\mathrm{OUT}_l(X) = \bigcup_{v \in X} \{u : (v, u) \in E \text{ and } \lambda((v, u)) = l\}$$

$$\mathrm{OUT}_{l}^{k,s}(X) = \bigcup_{v \in X}\{u: (v,u) \in E \text{ and } \lambda((v,u)) = l \text{ and } \mu((v,u),k) = s\}$$

Để di chuyển dọc theo các cạnh của đồ thị thuộc tính, chúng ta định nghĩa các phép duyệt các cạnh đi vào sau đây:

$$\mathrm{IN}(X) = \bigcup_{u \in X}\{u: (v,u) \in E\}$$

$$\mathrm{IN}_{l}(X) = \bigcup_{u \in X}\{u: (v,u) \in E \text{ and } \lambda((v,u)) = l\}$$

$$\mathrm{IN}_{l}^{k,s}(X) = \bigcup_{u \in X}\{u: (v,u) \in E \text{ and } \lambda((v,u)) = l \text{ and } \mu((v,u),k) = s\}$$

Cuối cùng, chúng ta định nghĩa hai phép duyệt OR và AND để tổng hợp kết quả của các phép duyệt khác nhau như sau.

$$\mathrm{OR}(T_1, \ldots, T_N)(X) = T_1(X) \cup T_2(X) \cup \ldots. \cup T_n(X)$$

$$\mathrm{AND}(T_1, \ldots, T_N)(X) = T_1(X) \cap T_2(X) \cap \ldots \cap T_n(X)$$

Mặc dù các định nghĩa kỹ thuật hơi phức tạp, nhiều cơ sở dữ liệu đồ thị cung cấp các triển khai hiệu quả cho các phép duyệt cơ bản này. Đặc biệt, các phép duyệt FILTER, OUT và IN là các hàm cơ bản trong ngôn ngữ đồ thị Gremlin được hỗ trợ bởi các cơ sở dữ liệu phổ biến như Neo4J và InfiniteGraph. Chúng tôi sẽ thảo luận chi tiết về việc triển khai các phép duyệt đồ thị cùng với đánh giá của phương pháp của chúng tôi trong Phần VI.

IV. CODE PROPERTY GRAPHS

Mỗi biểu diễn được giới thiệu trong Phần II nắm bắt các thuộc tính cụ thể của phần mềm cơ sở. Tuy nhiên, một biểu diễn đơn lẻ không đủ để miêu tả một loại lỗ hổng trong hầu hết các trường hợp. Do đó, chúng tôi kết hợp ba biểu diễn này thành một cấu trúc dữ liệu chung bằng cách sử dụng khái niệm đồ thị thuộc tính được giới thiệu trong Phần III. Cụ thể, chúng tôi đầu tiên mô hình hóa ASTs, CFGs và PDGs như các đồ thị thuộc tính và sau đó tiến hành hợp nhất chúng thành một đồ thị duy nhất cung cấp tất cả các lợi ích của các biểu diễn cá nhân.

A. Transforming the Abstract Syntax Tree

Đại diện duy nhất cung cấp một phân rã chi tiết của mã nguồn thành các cấu trúc ngôn ngữ là cây cú pháp trừu tượng (AST). Do đó, chúng ta bắt đầu xây dựng một biểu diễn chung bằng cách biểu diễn AST như một đồ thị thuộc tính $G_A = (V_A, E_A, \lambda_A, \mu_A)$,trong đó các nút V_A được xác định bởi các nút của cây và các cạnh E_A là các cạnh của cây tương ứng được gán nhãn là các cạnh AST bởi hàm nhãn λ_A. Ngoài ra, chúng ta gán một thuộc tính mã (code) cho mỗi nút bằng cách sử dụng μ_A, sao cho giá trị thuộc tính tương ứng với toán tử hoặc toán hạng mà nút đó đại diện.

Cuối cùng, chúng ta gán một thuộc tính thứ tự (order) cho mỗi nút để phản ánh cấu trúc có thứ tự của cây. Như vậy, các khóa thuộc tính của đồ thị là $K_A = \{code, order\}$, trong đó tập hợp các giá trị thuộc tính S_A bao gồm tất cả các toán tử và toán hạng cùng với các số tự nhiên.

B. Transforming the Control Flow Graph

Bước tiếp theo, chúng ta chuẩn bị CFG để tích hợp vào biểu diễn chung. Để làm điều này, chúng ta biểu diễn CFG như một đồ thị thuộc tính $G_C = (V_C, E_C, \lambda_C, .)$,trong đó các nút V_C đơn giản tương ứng với các câu lệnh và điều kiện trong AST, nghĩa là tất cả các nút V_A với các giá trị thuộc tính STMT và PRED cho khóa code. Hơn nữa, chúng ta định nghĩa một hàm gán nhãn cạnh λ_C gán nhãn từ tập $\Sigma_C = \{true, false, \epsilon\}$ cho tất cả các cạnh trong đồ thị thuộc tính.

C. Transforming the Program Dependence Graph

PDG biểu diễn các phụ thuộc dữ liệu và điều khiển giữa các câu lệnh và điều kiện. Các nút của đồ thị này do đó giống như của CFG và chỉ có sự khác biệt về các cạnh của hai đồ thị. Do đó, PDG có thể được biểu diễn như một đồ thị thuộc tính $G_P = (V_C, E_P, \lambda_P, \mu_P)$ đơn giản bằng cách định nghĩa một tập hợp mới các cạnh E_P và một hàm gán nhãn cạnh tương ứng $\lambda_P : E_P \rightarrow \Sigma_P$ với $\Sigma_P = \{C, D\}$ tương ứng với phụ thuộc điều khiển và dữ liệu. Ngoài ra, chúng ta gán một thuộc tính biểu tượng cho mỗi phụ thuộc dữ liệu để chỉ ra biểu tượng tương ứng và một điều kiện thuộc tính cho mỗi phụ thuộc điều khiển để chỉ ra trạng thái của điều kiện gốc là true hoặc false.

D. Combining the Representations

Và cuối cùng, chúng ta kết hợp ba đồ thị thuộc tính thành một cấu trúc dữ liệu chung được gọi là đồ thị thuộc tính mã nguồn. Cái nhìn quan trọng cần thiết để xây dựng đồ thị này là trong mỗi trong ba đồ thị, tồn tại một nút cho mỗi câu lệnh và điều kiện trong mã nguồn. Thực tế, AST là duy nhất trong ba biểu diễn này, mà giới thiệu các nút bổ sung. Các nút câu lệnh và điều kiện do đó tự nhiên kết nối các biểu diễn và phục vụ như các điểm chuyển tiếp từ một biểu diễn sang một biểu diễn khác.

Một đồ thị thuộc tính mã nguồn là một đồ thị thuộc tính G = (V, E, λ, μ) được xây dựng từ AST, CFG và PDG của mã nguồn với các thành phần sau:

- V là tập hợp các nút V_A của cây cú pháp trừu tượng (AST).
- E là tập hợp các cạnh $E_A \cup E_C \cup E_P$, gồm các cạnh từ cây cú pháp trừu tượng (AST), đồ thị luồng điều khiển (CFG) và đồ thị phụ thuộc chương trình (PDG).
- λ là hàm nhãn cạnh, kết hợp từ các hàm nhãn λ_A của AST, λ_C của CFG và λ_P của PDG.

- μ là hàm thuộc tính, kết hợp từ các hàm thuộc tính μ_A của AST và μ_E của các thành phần khác.

Ở đây, chúng ta kết hợp các hàm nhãn và thuộc tính với một chút lạm dụng biểu thức để tạo ra một đồ thị bao quát cho mã nguồn.

Ví dụ về một biểu đồ thuộc tính mã nguồn được minh họa trong Hình 4 cho mẫu mã được cung cấp trong Hình 1. Để đơn giản, các khóa và giá trị thuộc tính cũng như nhãn trên các cạnh AST

không được hiển thị. Các nút của đồ thị chủ yếu phù hợp với AST trong Hình 2a (trừ các nút FUNC và IF không liên quan), trong khi các CFG (Control Flow Graph) và PDG (Program Dependency Graph) được biến đổi được chỉ định bằng các cạnh màu sắc.

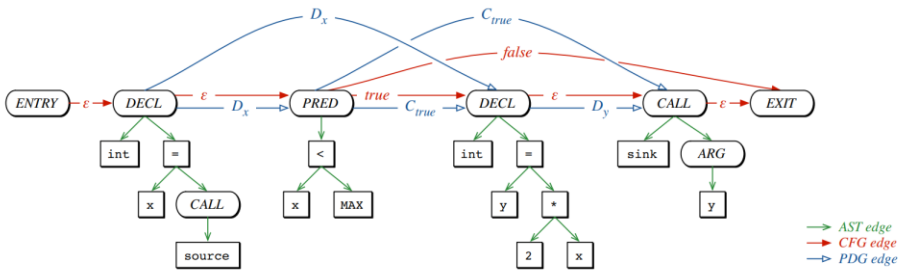


Fig. 4: Code property graph for the code sample given in Figure 1.