

**ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**



Công Nghĩa Hiếu

**XÂY DỰNG CÔNG CỤ PHÂN TÍCH MÃ NGUỒN CHO
NGÔN NGỮ RUST**

KHOÁ LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công nghệ thông tin

HÀ NỘI - 2024

**ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**



Công Nghĩa Hiếu

**XÂY DỰNG CÔNG CỤ PHÂN TÍCH MÃ NGUỒN CHO
NGÔN NGỮ RUST**

KHOÁ LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công nghệ thông tin

Cán bộ hướng dẫn: PGS. TS. Võ Đình Hiếu

HÀ NỘI - 2024

**VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY**



Cong Nghia Hieu

**DEVELOPING A SOURCE CODE ANALYSIS TOOL FOR THE
RUST PROGRAMMING LANGUAGE**

BACHELOR'S THESIS

Major: Information technology

Supervisor: Assoc. Prof., Dr. Vo Dinh Hieu

HANOI - 2024

Lời cảm ơn

Lời đầu tiên, tôi xin bày tỏ lòng biết ơn sâu sắc tới thầy Võ Đình Hiếu, người đã tận tình chỉ bảo tôi trong suốt quá trình học tập tại trường và đặc biệt là thời gian thực hiện khóa luận tốt nghiệp. Thầy đã không chỉ cung cấp những kiến thức chuyên môn quý báu mà còn động viên và hỗ trợ tôi vượt qua những khó khăn trong quá trình thực hiện khóa luận.

Tôi cũng xin cảm ơn ThS. Trần Mạnh Cường, những người đã luôn giúp đỡ tôi trong suốt thời gian thực tập. Sự hỗ trợ nhiệt tình của thầy đã giúp tôi có thêm tự tin và khả năng áp dụng kiến thức vào thực tiễn.

Ngoài ra, tôi xin gửi lời cảm ơn đến các thầy cô, anh chị, và các bạn trong phòng thí nghiệm bộ môn Công nghệ phần mềm, những người đã giúp đỡ tôi rất nhiều trong việc mở rộng kiến thức và kỹ năng thực hành.

Chúc mọi người luôn luôn vui vẻ và gặt hái được nhiều thành công trong cuộc sống.

Lời cam đoan

Tôi là Công Nghĩa Hiếu, sinh viên lớp QH-2021-I/CQ-I-IT2 khóa K66 theo học ngành Công nghệ thông tin tại trường Đại học Công Nghệ - Đại học Quốc gia Hà Nội. Tôi xin cam đoan khoá luận "Xây dựng công cụ phân tích mã nguồn cho ngôn ngữ Rust" là công trình nghiên cứu do bản thân tôi thực hiện. Các nội dung nghiên cứu, kết quả trong khoá luận là xác thực.

Các thông tin sử dụng trong khoá luận là có cơ sở và không có nội dung nào sao chép từ các tài liệu mà không ghi rõ trích dẫn tham khảo. Tôi xin chịu trách nhiệm về lời cam đoan này.

Hà Nội, ngày 16 tháng 12 năm 2024

Sinh viên

Công Nghĩa Hiếu

Tóm tắt

Tóm tắt:

Rust đang trở thành một ngôn ngữ lập trình vô cùng nổi bật và được sử dụng rộng rãi trong vài năm gần đây. Nhờ vào các tính năng nổi bật về đảm bảo an toàn bộ nhớ và hiệu suất vượt trội, mã nguồn của hàng ngàn dự án trên thế giới đã được viết lại hoặc viết mới bằng Rust, từ các hệ thống nhúng cho đến các ứng dụng web hiện đại. Rust đã được chọn làm ngôn ngữ phát triển của các dự án cấp độ doanh nghiệp như Servo - web engine của trình duyệt Mozilla, 1 phần của hệ điều hành Windows 11, và nhiều dự án khác.

Những dự án này cho thấy Rust không chỉ là một xu hướng mà còn là một ngôn ngữ có tiềm năng phát triển lâu dài. Với sự phổ biến của Rust, nhu cầu về các công cụ phân tích mã nguồn, kiểm thử và bảo mật cũng ngày càng tăng. Khi các dự án lớn chuyển sang sử dụng Rust, việc đảm bảo mã nguồn an toàn và không có lỗi trở thành một thách thức. Tuy nhiên hệ sinh thái và các công cụ cho phân tích mã nguồn Rust hiện tại vẫn chưa đáp ứng đủ. Rust chỉ mới được sử dụng phổ biến trong vài năm gần đây, vì vậy việc phát triển các công cụ phân tích vẫn còn trong giai đoạn đầu. Nhằm nâng cao độ tin cậy của mã nguồn, các phương pháp kiểm thử phần mềm tự động đang nhận được đông đảo sự quan tâm và đang được áp dụng rộng rãi trong cả cộng đồng nghiên cứu lẫn các công ty phần mềm. Một trong những phương pháp sử dụng trong kiểm thử là phân tích mã nguồn hay phân tích tĩnh. Quá trình này là việc phân tích, đánh giá chất lượng mã nguồn và tìm ra các lỗi lập trình, lỗ hổng bảo mật mà không cần phải thực thi chương trình.

Khóa luận này trình bày phương pháp xây dựng công cụ phân tích mã nguồn dành cho ngôn ngữ Rust. Đầu vào của công cụ là các tệp mã nguồn Rust và đầu ra là đồ thị thuộc tính mã nguồn (CPG). Công cụ sử dụng đồ thị thuộc tính mã nguồn tuân theo chuẩn đặc tả và nền tảng có sẵn của Joern. Đầu ra có thể được lưu trữ trong cơ sở dữ liệu đồ thị, trực quan hóa, truy vấn, phân tích thủ công hoặc tự động nhằm phục vụ cho mục đích phân tích mã nguồn khác nhau. Ngoài ra CPG này có thể được áp dụng cho các kĩ thuật học máy để phát hiện lỗ hổng bảo mật như graph neural networks (GNN).

Từ khóa: Phân tích mã nguồn, phân tích tĩnh, ngôn ngữ lập trình Rust

Abstract

Abstract:

Rust has become an incredibly prominent and widely-used programming language in recent years. Thanks to its outstanding features of ensuring memory safety and superior performance, the source code of thousands of projects worldwide has been rewritten or newly developed using Rust, from embedded systems to modern web applications. Rust has been chosen as the development language for enterprise-level projects such as Servo, the web engine of the Mozilla browser, part of the Windows 11 operating system, and many other projects.

These projects demonstrate that Rust is not just a trend but a language with long-term potential for development. As Rust becomes more popular, the demand for source code analysis, testing, and security tools is also increasing. When major projects switch to using Rust, ensuring that the source code is safe and error-free becomes a challenge. However, the current ecosystem and tools for Rust source code analysis are still insufficient. Since Rust has only become widely used in recent years, the development of analysis tools is still in its early stages. To enhance the reliability of source code, automated software testing methods are receiving widespread attention and are being widely applied in both the research community and software companies. One of the methods used in testing is source code analysis or static analysis. This process involves analyzing, evaluating the quality of the source code, and identifying programming errors and security vulnerabilities without needing to execute the program.

This thesis presents a method for building a source code analysis tool for the Rust language. The tool's input is Rust source code files, and the output is a Code Property Graph (CPG). The tool utilizes the Code Property Graph following the specification and existing platform of Joern. The output can be stored in a graph database, visualized, queried, manually or automatically analyzed to serve various source code analysis purposes. Additionally, this CPG can be applied to machine learning techniques to detect security vulnerabilities such as graph neural networks (GNN).

Keywords: Source code analysis, static analysis, Rust programming language

Mục lục

Lời cảm ơn

Lời cam đoan i

Tóm tắt ii

Abstract iii

Mục lục iv

Danh sách hình vẽ vi

Danh sách bảng vii

Danh sách đoạn mã viii

Danh mục các từ viết tắt ix

Chương 1 Đặt vấn đề 1

Chương 2 Kiến thức cơ sở 3

2.1 Ngôn ngữ lập trình Rust 3

2.1.1 Giới thiệu tổng quan 3

2.1.2 Cơ chế an toàn của Rust 4

2.2 Các cách biểu diễn mã nguồn 6

2.2.1 Cây cú pháp trừu tượng 6

2.2.2 Đồ thị dòng điều khiển 8

2.2.3 Đồ thị phụ thuộc chương trình 10

2.2.4 Đồ thị thuộc tính mã nguồn 10

2.3 Công cụ Joern 12

2.3.1 Đặc tả đồ thị thuộc tính mã nguồn của Joern 12

2.3.2 Bộ công cụ của Joern 14

Chương 3 Phân tích mã nguồn Rust	16
3.1 Quy trình tổng quan	16
3.2 Xây dựng cây cú pháp trừu tượng	17
3.3 Xây dựng đồ thị thuộc tính mã nguồn (CPG)	24
3.3.1 Các loại đỉnh và cạnh của đồ thị thuộc tính mã nguồn CPG . .	25
3.3.2 Chuyển hóa cây cú pháp trừu tượng sang đồ thị thuộc tính mã nguồn	27
3.4 Thực nghiệm trên các tính năng đã hỗ trợ	27
3.4.1 if let	27
3.4.2 while let	28
3.4.3 match	29
3.4.4 lifetime	30
3.5 Hạn chế	30
3.5.1 Path	30
3.5.2 Macro	30
3.5.3 Module	30
3.5.4 Type Argument match Type Parameter	30
Chương 4 Cài đặt công cụ và thực nghiệm	31
4.1 Cài đặt công cụ	31
4.1.1 Mô-đun xây dựng cây cú pháp trừu tượng Rust Parser	32
4.1.2 Mô-đun xây dựng đồ thị thuộc tính mã nguồn Joern	32
4.2 Tính áp dụng vào thực tế của đồ thị CPG dành cho Rust	33
4.2.1 RUSTSEC-2021-0086	33
4.2.2 RUSTSEC-2022-0028	34
4.2.3 RUSTSEC-2020-0044	34
4.2.4 RUSTSEC-2021-0130	35
Kết luận	36
Tài liệu tham khảo	37

Danh sách hình vẽ

2.1	Ví dụ về cây cú pháp trừu tượng cho mã nguồn Rust.	8
2.2	Các thành phần cơ bản trong đồ thị dòng điều khiển.	9
2.3	Các cấu trúc điều khiển phổ biến trong các ngôn ngữ lập trình. . . .	9
2.4	Ví dụ về đồ thị phụ thuộc chương trình.	10
2.5	Ví dụ đồ thị thuộc tính mã nguồn.	11
2.6	Cách hoạt động của công cụ Joern.	14
2.7	Các công cụ xung quanh Joern.	14
3.1	Quy trình phân tích mã nguồn Rust.	16
3.2	Quy trình xây dựng cây cú pháp trừu tượng.	17
3.3	Ví dụ đồ thị thuộc tính mã nguồn.	18
3.4	Ví dụ đồ thị thuộc tính mã nguồn.	19
4.1	Kiến trúc công cụ.	31

Danh sách bảng

3.1	Các nút trong cú pháp mã nguồn của Rust.	24
3.2	Các đỉnh trong đồ thị thuộc tính mã nguồn (CPG)	26
3.3	Các cạnh trong đồ thị thuộc tính mã nguồn (CPG).	27

Danh sách đoạn mã

2.1 Ví dụ các khái niệm an toàn trong Rust: (1) ownership, (2) borrowing, and (3) Exclusive mutability.	4
2.2 Mã nguồn đầy đủ cho đồ thị thuộc tính mã nguồn hình 2.5.	11
3.1 Ví dụ mã nguồn cho	27
3.2 Ví dụ mã nguồn cho	28
3.3 Ví dụ mã nguồn cho	29
3.4 Ví dụ mã nguồn cho	30
4.1 Ví dụ mã nguồn cho RUSTSEC-2021-0086	33
4.2 Ví dụ mã nguồn cho RUSTSEC-2022-0028	34
4.3 Ví dụ mã nguồn cho RUSTSEC-2020-0044	34
4.4 Ví dụ mã nguồn cho RUSTSEC-2021-0130	35

Danh mục các từ viết tắt

Từ viết tắt	Cụm từ đầy đủ	Cụm từ tiếng Việt
AST	Abstract Syntax Tree	Cây cú pháp trừu tượng
CFG	Control Flow Graph	Đồ thị dòng điều khiển
PDG	Program Dependence Graph	Đồ thị phụ thuộc chương trình
CDG	Control Dependence Graph	Đồ thị phụ thuộc điều khiển
CPG	Code Property Graph	Đồ thị thuộc tính mã nguồn
IDE	Integrated Development Environment	Môi trường phát triển tích hợp

Chương 1

Đặt vấn đề

Hiện nay, cùng với sự phát triển của công nghệ và độ phủ sóng của internet, ngành công nghiệp phần mềm đang trải qua giai đoạn phát triển mạnh mẽ và đa dạng hóa không ngừng, đi kèm với đó là hàng tỷ người sử dụng. Các hệ thống phần mềm phải luôn đảm bảo tính sẵn sàng và ổn định bởi vì chỉ một sai sót xảy ra cũng sẽ gây ra những hậu quả khôn lường. Trong quá trình xây dựng phần mềm, các đội ngũ phát triển phần mềm thường dùng là sử dụng các công cụ phân tích mã nguồn cho việc phát triển, kiểm thử và đảm bảo chất lượng phần mềm. Các công cụ phân tích mã nguồn sẽ giúp phát hiện các lỗi lập trình, vấn đề về bảo mật hay việc sử dụng tài nguyên kém hiệu quả mà không cần phải thực thi chương trình. Đồng thời, các công cụ này sẽ giúp cải thiện chất lượng mã nguồn, đảm bảo lập trình viên sẽ tuân thủ các quy tắc và tiêu chuẩn khi viết mã nguồn. Điều này sẽ giúp giảm thiểu thời gian và công sức cho quá trình kiểm thử và đánh giá mã nguồn.

Hiện tại, trên thị trường đã có nhiều công cụ cung cấp khả năng phân tích mã nguồn như SonarQube, ReSharper hay CodeClimate. Những công cụ này cung cấp khả năng phân tích mã nguồn cho nhiều ngôn ngữ lập trình khác nhau như C/C++, Java, Javascript, C... Bên cạnh đó, mỗi ngôn ngữ lập trình lại có thêm nhiều công cụ phân tích mã nguồn khác nhau, nếu xét riêng cho ngôn ngữ lập trình Rust, ta có một số công cụ tiêu biểu như gosec1, staticcheck2 hay govulncheck3. Đây là các công cụ cung cấp khả năng phân tích cú pháp và tìm lỗ hổng trong mã nguồn. Điểm chung của các công cụ này là đều sử dụng dạng cây cú pháp trừu tượng được định nghĩa sẵn của Rust nên chúng chỉ dừng lại ở tìm các lỗi dựa trên cú pháp mã nguồn mà không khai thác sâu hơn về luồng điều khiển hay luồng dữ liệu của mã nguồn. Do vậy, những công cụ này thường cung cấp nhiều cảnh báo giả và bỏ qua những lỗi nghiêm trọng khi phân tích mã nguồn.

Khác với các công cụ vừa nêu, trên thị trường hiện nay còn xuất hiện một công cụ là Joern. Joern là một nền tảng mã nguồn mở cung cấp khả năng phân tích mã nguồn, mã bytecode và mã nhị phân. Một công cụ phân tích mã nguồn sẽ gồm ba thành phần chính: phân tích cú pháp mã nguồn (parser), biểu diễn cấu trúc của

mã nguồn và phân tích cấu trúc biểu diễn đó [3]. Joern đóng vai trò là một công cụ giúp xử lý hai bước đầu của quá trình phân tích mã nguồn. Thay vì chỉ sử dụng cây cú pháp trừu tượng, Joern biểu diễn mã nguồn dưới dạng đồ thị thuộc tính mã nguồn và cung cấp chức năng truy vấn khai thác đồ thị này bằng các câu lệnh truy vấn viết bằng ngôn ngữ Scala. Joern được xây dựng với mục tiêu cung cấp chức năng tìm kiếm lỗi hổng trong mã nguồn và cung cấp các công cụ để xây dựng các trình phân tích tĩnh sử dụng dữ liệu phân tích mà Joern cung cấp. Điểm đặc biệt của Joern là nó cung cấp dạng đồ thị biểu diễn duy nhất cho tất cả các ngôn ngữ mà nó hỗ trợ (bao gồm C, C++, Java, Javascript, Python...), điều này đem đến khả năng phân tích đa ngôn ngữ, giúp giảm thiểu việc phải xử lý riêng cho từng ngôn ngữ khi chúng ta tìm kiếm lỗi trong mã nguồn hay khi xây dựng các trình phân tích tĩnh dựa trên Joern.

Mặc dù Joern đã cung cấp khả năng phân tích mã nguồn đối với ngôn ngữ Rust tuy nhiên vẫn còn rất hạn chế do chưa xử lý hết toàn bộ các thành phần mã nguồn có trong Rust và thiếu các thông tin về kiểu dữ liệu. Vì vậy khóa luận này xây dựng một công cụ phân tích mã nguồn dành cho ngôn ngữ lập trình Rust dựa trên mã nguồn của Joern. Công cụ này sẽ xử lý các thành mã nguồn mà Rust cung cấp một cách chi tiết và đầy đủ hơn, đồng thời cũng xử lý triệt để kiểu dữ liệu để cung cấp đồ thị mã nguồn chi tiết nhất có thể.

Khóa luận sẽ được trình bày theo cấu trúc như sau. Đầu tiên, Chương 2 thảo luận một số kiến thức cơ sở liên quan đến phân tích mã nguồn, cụ thể cho ngôn ngữ lập trình Rust. Chương 3 trình bày chi tiết quy trình xây dựng công cụ phân tích. Tiếp theo, Chương 4 sẽ trình bày kiến trúc, cài đặt công cụ và một số kết quả thực nghiệm đánh giá. Cuối cùng, tóm tắt những kết quả và kết luận sau quá trình phát triển công cụ sẽ được trình bày ở Chương 5.

Chương 2

Kiến thức cơ sở

Chương này sẽ trình bày các kiến thức cơ sở về ngôn ngữ lập trình Rust, cơ chế quản lý bộ nhớ hiệu quả tạo nên đặc trưng của Rust và tính hướng hàm được đan xen vào trong cú pháp. Chương cũng sẽ trình bày về các dạng biểu diễn đồ thị của mã nguồn bao gồm cây cú pháp trừu tượng, đồ thị dòng điều khiển, đồ thị phụ thuộc chương trình và đồ thị thuộc tính mã nguồn. Bộ công cụ Joern, 1 bộ công cụ phân tích mã nguồn tĩnh, và những ưu điểm mà nó mang lại sẽ được giới thiệu ở phần cuối của chương.

2.1 Ngôn ngữ lập trình Rust

2.1.1 Giới thiệu tổng quan

Rust lần đầu tiên được giới thiệu vào năm 2006 bởi Mozilla, với phiên bản 1.0 được công bố vào năm 2015 và nhanh chóng được đem vào sử dụng trong nhiều dự án. Với điểm mạnh tập trung vào sự an toàn và hiệu suất cao, có thể so sánh với C/C++, là những lý do chính cho sự thành công của Rust [8, 19]. Rust là ngôn ngữ định kiểu mạnh tương tự như C/C++ hay Java và áp đặt các cơ chế đảm bảo về an toàn bộ nhớ, bổ sung cho sự an toàn của ngôn ngữ này [15]. Một số tính năng an toàn có thể kể đến như đảm bảo chỉ có một tham chiếu có thể ghi (mutable) tới một đối tượng hoặc nhiều tham chiếu chỉ đọc, nhưng không thể cả ghi và đọc cùng 1 lúc. Cơ chế quản lý bộ nhớ *ownership* được áp dụng vào thời điểm biên dịch, do đó loại bỏ được một lớp lớn các lỗi bộ nhớ mà C/C++ gặp phải. Các lỗi về an toàn bộ nhớ cổ điển có thể được tránh bởi việc sử dụng Rust bao gồm tràn bộ đệm (stack-overflow), sử dụng sau khi giải phóng bộ nhớ (use-after-free) và tham chiếu null [5]. Ngoài các cơ chế đảm bảo an toàn được sử dụng mặc định trong ngôn ngữ, Rust vẫn cho phép viết những đoạn mã code có tiềm năng mất an toàn bằng từ khóa *unsafe* [15]. Tổng quan, Rust cung cấp cho sự an toàn và quyền kiểm soát khi cần thiết, làm cho nó rất phù hợp cho lập trình hệ thống [10], lập trình nhúng [18], hay các chương trình du hành ngoài không gian yêu cầu sự an toàn tuyệt đối

[17].

2.1.2 Cơ chế an toàn của Rust

```
1 fn ownership_and_borrowing() -> &u32 {
2     // creates a `Vec`, a heap allocated buffer
3     let vec = vec![1, 2, 3];
4
5     // creates a reference to the first value with borrowing
6     let first_val = &vec[0];
7
8     // Ownership: `Vec` is automatically reclaimed
9     // when its owner `vec` goes out of the scope.
10    //
11    // Borrowing: compile error; Rust prevents `first_val`
12    // to outlive `vec` by tracking variable lifetimes.
13    return first_val;
14 }
15
16 fn aliasing_xor_mutability() {
17     let mut vec = vec![1, 2, 3];
18
19     // exclusive mutable borrowing
20     let mut_ref = &mut vec;
21
22     // shared read-only borrowing
23     let shared_ref1 = &vec;
24     let shared_ref2 = &vec;
25     println!("{}", shared_ref1[0]);
26     println!("{}", shared_ref2[0]);
27
28     // Exclusive mutability: compile error;
29     // Rust invalidates `mut_ref` when `shared_ref1` is
30     // used since they cannot coexist at the same time.
31     mut_ref.push(4);
32 }
```

Đoạn mã 2.1: Ví dụ các khái niệm an toàn trong Rust: (1) ownership, (2) borrowing, and (3) Exclusive mutability.

Rust là một ngôn ngữ an toàn về kiểu, được thiết kế cho phát triển phần mềm

mức hệ thống, mang lại cho lập trình viên quyền kiểm soát tối đa với tài nguyên nhưng đảm bảo an toàn bộ nhớ và đa luồng bằng một tập các cơ chế nghiêm ngặt. Trình biên dịch của Rust sẽ kiểm tra các cơ chế này để loại bỏ các vấn đề nguy hiểm tiềm tàng. Các cơ chế an toàn bao gồm các khái niệm cơ bản:

Ownership: Cơ chế *ownership* giúp Rust có sự điều khiển vừa đủ với bộ nhớ, không cần sử dụng bộ thu gom rác (garbage collector) hoặc để người dùng tự xử lý như C/C++. Theo cơ chế *ownership* của Rust, một giá trị (vị trí bộ nhớ) chỉ có một chủ sở hữu độc quyền (biến). Khi chủ sở hữu của giá trị ra khỏi phạm vi cụ thể, giá trị trong bộ nhớ sẽ bị giải phóng. Gán biến cho 1 biến khác dẫn đến chuyển quyền sở hữu. Khi một biến mất quyền sở hữu một giá trị, biến đó sẽ không còn sử dụng được. Trình biên dịch Rust theo dõi tuổi thọ của mỗi giá trị thông qua cơ chế *ownership* và thực hiện thu hồi bộ nhớ cần thiết. Cơ chế *ownership* tương tự như mẫu Resource-Acquisition-Is -Initialization (RAII) [3] thường được sử dụng trong ngôn ngữ C++.

Borrowing: Rust cho phép mượn giá trị (tức là tạo tham chiếu đến nó) trong suốt thời gian sống của biến chủ sở hữu. Với cơ chế mượn, một giá trị có thể được đọc hoặc cập nhật mà không thay đổi quyền sở hữu của giá trị. Hệ thống kiểu của Rust đảm bảo rằng các vấn đề an toàn bộ nhớ truyền thống như sử dụng sau khi giải phóng (use-after-free) hoặc con trỏ treo không thể xảy ra bằng cách không cho phép các tham chiếu tồn tại lâu hơn biến chủ sở hữu.

Exclusive mutability: Có hai loại mượn: 1) mượn chia sẻ để đọc và 2) mượn độc quyền để ghi. Trình biên dịch Rust đảm bảo rằng cả tham chiếu đọc và tham chiếu ghi không bao giờ xuất hiện cùng một lúc. Điều này có nghĩa là các thao tác đọc và ghi đồng thời là không thể trong Rust, loại bỏ khả năng xảy ra tương tranh với dữ liệu và các lỗi an toàn bộ nhớ như truy cập các tham chiếu không hợp lệ (null-pointer dereferencing).

Lifetime: Lifetime giải thích các phạm vi mà tham chiếu trong chương trình Rust có hiệu lực. Tính năng lifetime trong Rust bao gồm một loạt các generic cho biết cách các tham chiếu liên quan đến nhau. Cụ thể, để xác định khi nào các tham chiếu ra khỏi phạm vi, trình biên dịch liên kết mỗi tham chiếu mượn với một lifetime và theo dõi các ràng buộc giữa các tham chiếu. Lifetime inference đảm bảo rằng thời gian sống của quyền sở hữu mượn sẽ đủ dài để sử dụng.

Thread safe: Lập trình đa luồng trong Rust được đảm bảo an toàn nhờ mô hình

ownership và *borrowing*, ngăn chặn tương tranh dữ liệu ngay từ khi biên dịch.

Send và Sync Traits: Rust sử dụng các đặc điểm Send và Sync để đảm bảo an toàn đa luồng ở mức kiểu. Một kiểu là Send nếu nó có thể được chuyển an toàn giữa các luồng, và Sync nếu nó có thể được chia sẻ an toàn giữa các luồng. Trình biên dịch Rust sử dụng các đặc điểm này để kiểm tra tại thời điểm biên dịch xem dữ liệu có thể được chia sẻ hoặc di chuyển giữa các luồng một cách an toàn hay không.

Mutex và Arc: Đối với trạng thái có thể thay đổi được chia sẻ, Rust cung cấp các nguyên thủy đồng bộ hóa như Mutex (loại trừ lẫn nhau) và Arc (đếm tham chiếu nguyên tử). Mutex đảm bảo rằng chỉ có một luồng có thể truy cập dữ liệu tại một thời điểm, ngăn chặn các điều kiện đua. Arc được sử dụng để đếm tham chiếu an toàn giữa các luồng, cho phép nhiều luồng chia sẻ quyền sở hữu dữ liệu.

Ownership Transfer in Threads: Rust khuyến khích chuyển quyền sở hữu dữ liệu vào các luồng, điều này ngăn chặn trạng thái có thể thay đổi được chia sẻ giữa các luồng. Khi một luồng được tạo ra, dữ liệu có thể được chuyển vào luồng đó, đảm bảo rằng luồng cha không còn quyền truy cập vào nó, do đó tránh được các điều kiện đua tiềm ẩn.

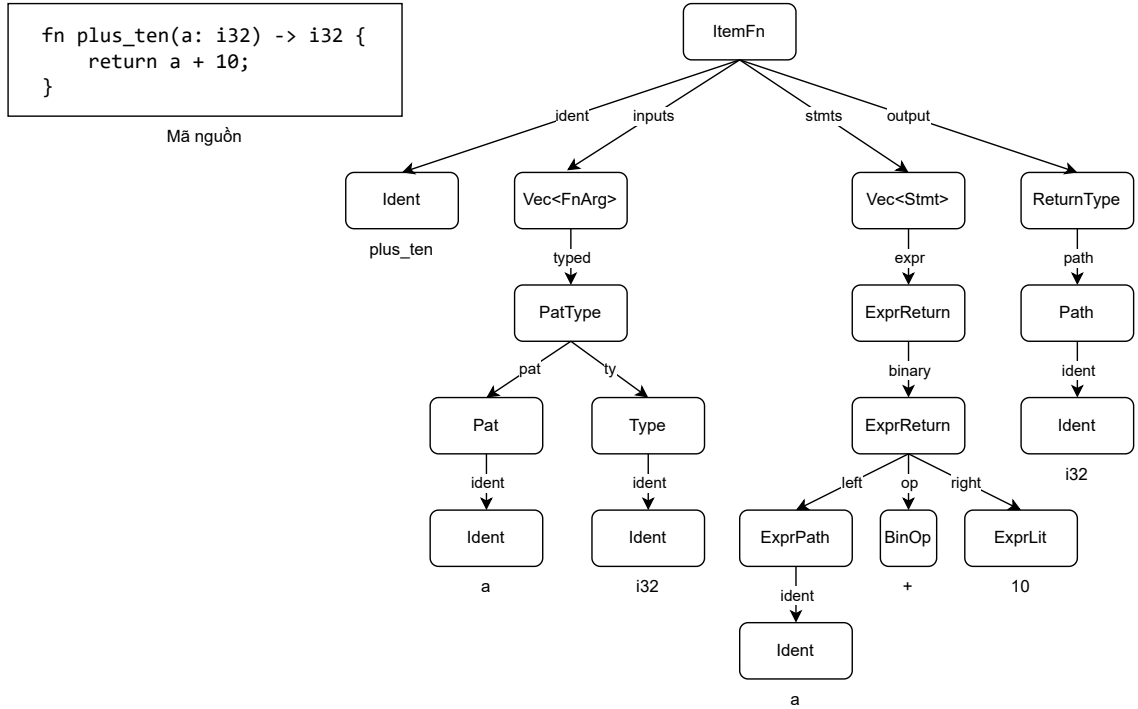
2.2 Các cách biểu diễn mã nguồn

Đối với các kỹ thuật phân tích chương trình, mã nguồn thường được thể hiện dưới dạng cấu trúc dữ liệu cây và đồ thị. Khóa luận sử dụng hai dạng dữ liệu là cây cú pháp trừu tượng và đồ thị thuộc tính mã nguồn, trong đó đồ thị thuộc tính mã nguồn là dạng đồ thị được hợp thành từ cây cú pháp trừu tượng, đồ thị dòng điều khiển và đồ thị phụ thuộc chương trình.

2.2.1 Cây cú pháp trừu tượng

Cây cú pháp trừu tượng (Abstract Syntax Tree) [26] là một cấu trúc dữ liệu quan trọng trong ngôn ngữ lập trình Rust, biểu diễn cấu trúc mã nguồn của chương trình. Khi trình biên dịch Rust phân tích mã nguồn, nó tạo ra một AST để hiểu và xử lý cú pháp của chương trình. AST trong Rust giúp trừu tượng hóa các phần chi tiết của mã nguồn và chỉ giữ lại những thông tin cần thiết để trình biên dịch

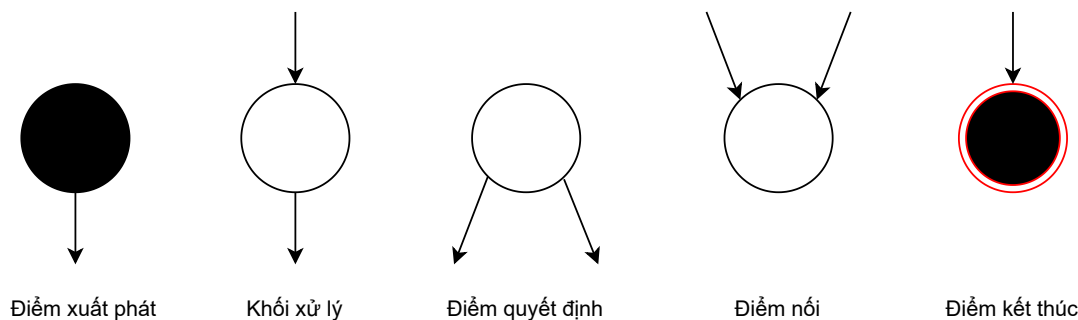
hiểu cấu trúc của chương trình. Điều này bao gồm các thông tin về khai báo biến, hàm, khối lệnh, biểu thức và mệnh đề. AST trong Rust được sử dụng rộng rãi trong nhiều công cụ và ứng dụng khác nhau, bao gồm trình biên dịch (compiler), trình dịch ngược (decompiler), và các công cụ phân tích mã nguồn (Source Code Analysis Tool). Trong Rust, AST không chỉ giúp trình biên dịch hiểu cấu trúc của mã nguồn mà còn hỗ trợ quá trình tối ưu hóa và kiểm tra lỗi. Ví dụ, khi Rust kiểm tra các quy tắc về an toàn bộ nhớ, nó sử dụng AST để xác minh rằng các biến được sử dụng đúng cách và không gây ra các lỗi bộ nhớ như tràn bộ đệm hay truy cập ngoài giới hạn. AST cũng hỗ trợ việc thực hiện các thao tác như refactoring, giúp thay đổi cấu trúc mã nguồn một cách tự động mà không làm thay đổi hành vi của chương trình. Ngoài ra, AST còn đóng vai trò quan trọng trong việc tạo ra các công cụ kiểm thử tự động và phân tích bảo mật, giúp phát hiện sớm các lỗi và lỗ hổng tiềm ẩn trong mã nguồn. Cấu trúc của AST trong Rust có thể khác nhau tùy thuộc vào cách triển khai của trình biên dịch. Mỗi nút trong AST đại diện cho một phần tử cụ thể của mã nguồn, chẳng hạn như một khai báo biến hay một biểu thức điều kiện. Các nút này có thể có các thuộc tính và con trỏ đến các nút con, tạo thành một cây cấu trúc phân cấp. Một trong những ưu điểm lớn của AST trong Rust là khả năng tích hợp với các công cụ xử lý ngôn ngữ tự nhiên (NLP) và các công cụ mã nguồn khác. Bằng cách sử dụng AST, các nhà phát triển có thể tạo ra các công cụ phân tích mã nguồn mạnh mẽ và hiệu quả, giúp cải thiện chất lượng và an toàn của mã nguồn Rust. Hình 2.1 biểu diễn một cây cú pháp trừu tượng tương ứng với mã nguồn trong Rust.



Hình 2.1: Ví dụ về cây cú pháp trừu tượng cho mã nguồn Rust.

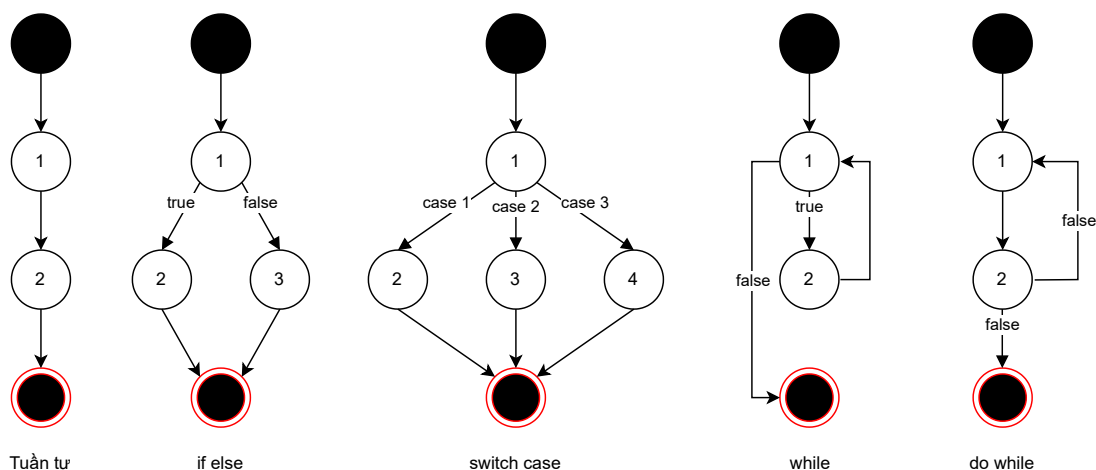
2.2.2 Đồ thị dòng điều khiển

Đồ thị dòng điều khiển (Control Flow Graph) [25] là đồ thị có hướng, biểu diễn kịch bản thực thi của chương trình/đơn vị chương trình. Đồ thị này được xây dựng từ mã nguồn của chương trình, trong đó các nút là các câu lệnh/nhóm câu lệnh và các cạnh là các dòng điều khiển, thể hiện thứ tự thực hiện của câu lệnh/nhóm câu lệnh. Tất cả các đồ thị dòng điều khiển đều có điểm xuất phát và điểm kết thúc đại diện cho trạng thái bắt đầu và trạng thái kết thúc của chương trình. CFG bao gồm các thành phần chính là điểm xuất phát, khối xử lý, điểm quyết định, điểm nối và điểm kết thúc.



Hình 2.2: Các thành phần cơ bản trong đồ thị dòng điều khiển.

Trong hình 2.2, **điểm xuất phát** và **điểm kết thúc** biểu thị điểm bắt đầu và kết thúc của chương trình, lần lượt được thể hiện bằng hình tròn đặc và hình tròn đặc viền. **Khối xử lý** tượng trưng cho các câu lệnh gán, khai báo và khởi tạo, được thể hiện bằng hình tròn rỗng. **Điểm quyết định** biểu thị các câu lệnh điều kiện trong các khối lệnh rẽ nhánh, được thể hiện bằng hình tròn rỗng với nhiều nhánh đi ra. **Điểm nối** biểu thị các câu lệnh thực hiện ngay sau các lệnh rẽ nhánh, có nhiều hơn một điểm trở đến, được thể hiện bằng hình tròn rỗng.

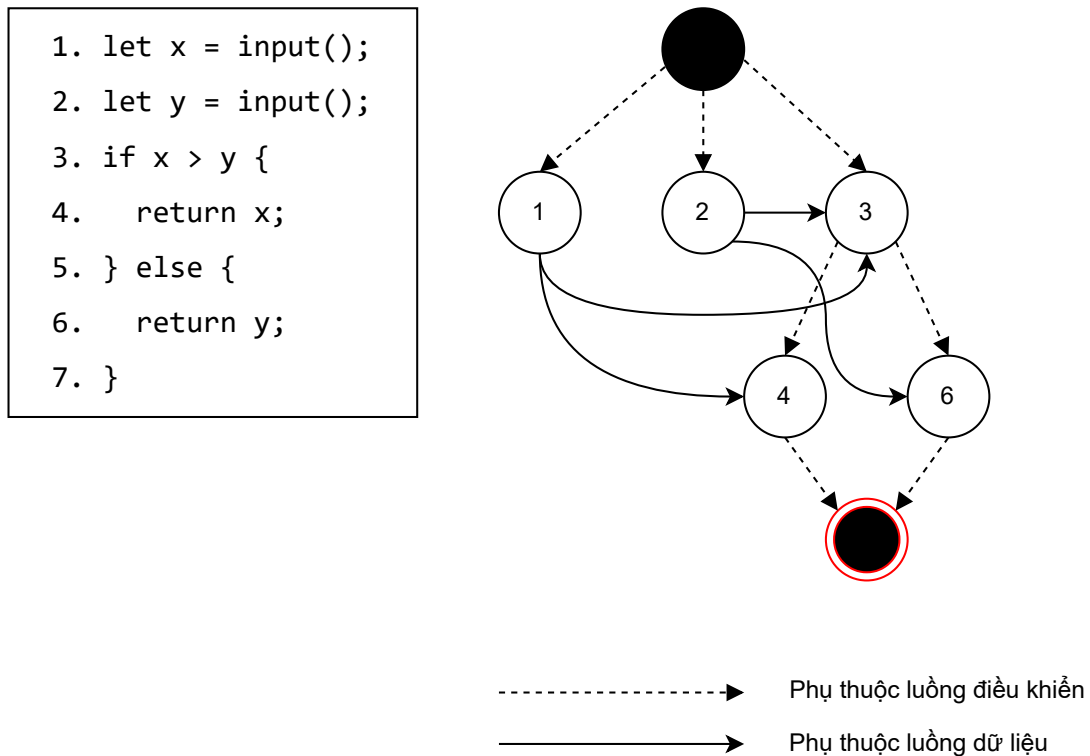


Hình 2.3: Các cấu trúc điều khiển phổ biến trong các ngôn ngữ lập trình.

Hình 2.3 mô tả các cấu trúc điều khiển phổ biến có trong các ngôn ngữ lập trình được biểu diễn dưới dạng đồ thị CFG, bao gồm có cấu trúc điều khiển tuần tự, *if else*, *switch case*, *while* và *do while*.

2.2.3 Đồ thị phụ thuộc chương trình

Đồ thị phụ thuộc chương trình (Program Dependence Graph) [4, 7] là đồ thị có hướng thể hiện cả 2 khía cạnh phụ thuộc điều khiển và phụ thuộc dữ liệu của chương trình. Một nút đại diện cho một câu lệnh / nhóm câu lệnh, một cạnh thể hiện mối quan hệ phụ thuộc điều khiển hoặc phụ thuộc dữ liệu giữa các nút. Câu lệnh / nhóm câu lệnh mà một nút đại diện có được thực hiện hay không phụ thuộc vào các cạnh điều kiện điều khiển và điều kiện dữ liệu trở tới nút đó.



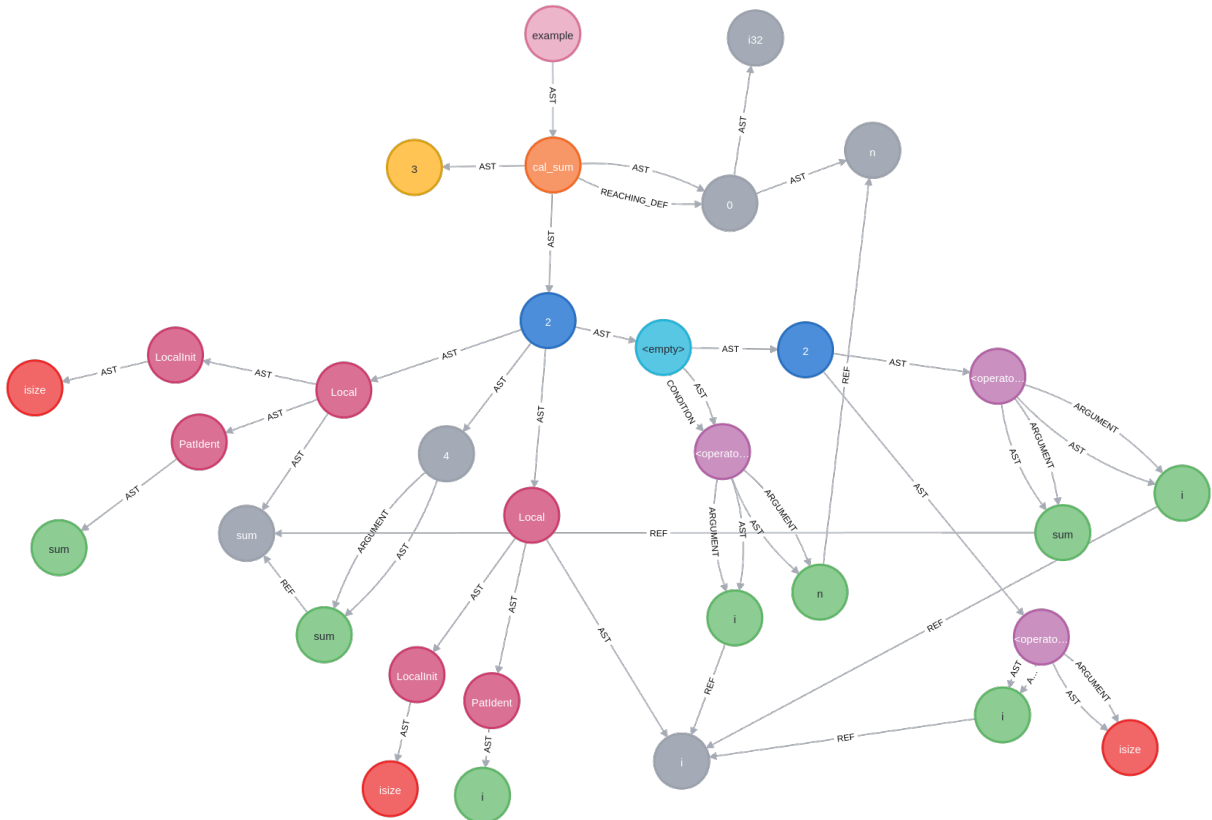
Hình 2.4: Ví dụ về đồ thị phụ thuộc chương trình.

Hình 2.4 biểu diễn ví dụ về một đồ thị phụ thuộc chương trình với cấu trúc điều kiện *if else* trong ngôn ngữ Rust, các cạnh nét đứt biểu diễn phụ thuộc điều khiển và các cạnh nét liền biểu diễn phụ thuộc dữ liệu.

2.2.4 Đồ thị thuộc tính mã nguồn

Đồ thị thuộc tính mã nguồn (Code Property Graph) [24] là một dạng đồ thị biểu diễn mã nguồn mới, được hợp thành cây cú pháp trừu tượng, đồ thị dòng điều

hiển và đồ thị phụ thuộc chương trình. Đồ thị chứa các thông tin về cấu trúc cú pháp, luồng điều khiển và phụ thuộc dữ liệu trong chương trình. Các nút đại diện cho các thành phần như hàm, biến, lớp, gói và các cạnh đại diện cho mối quan hệ giữa chúng như lời gọi hàm, sự gán giá trị, quan hệ cha con hay tham chiếu. CPG được sử dụng để tìm kiếm lỗi hỏng trong mã nguồn, phát hiện sao chép mã nguồn, đo lường khả năng kiểm thử mã nguồn và sinh mã khai thác [6, 12, 23].



Hình 2.5: Ví dụ đồ thị thuộc tính mã nguồn.

```

1 fn cal_sum(n: i32) {
2   let mut sum = 0;
3   let mut i = 1;
4   while i <= n {
5     sum += i;
6     i += 1;
7   }
8   return sum;
9 }

```

Đoạn mã 2.2: Mã nguồn đầy đủ cho đồ thị thuộc tính mã nguồn hình 2.5.

2.3 Công cụ Joern

2.3.1 Đặc tả đồ thị thuộc tính mã nguồn của Joern

NOTE: Thêm ý đặc tả CPG Joern ban đầu được viết cho ngôn ngữ C, sau đó mở rộng cho các ngôn ngữ khác như Java, PHP, ... Tuy nhiên đa phần các ngôn ngữ này đều là ngôn ngữ có cú pháp C-like và chủ yếu được code dưới OOP. Do vậy đối với ngôn ngữ có sự đan xen của các tính năng Functional Programming, thì bản đặc tả này tỏ ra có sự hạn chế. Vẽ biểu đồ luồng hoạt động của Joern Frontend, Joern Backend, Joern Backend cung cấp lợi thế gì (tính tái sử dụng)

Đồ thị thuộc tính mã nguồn đã được nghiên cứu rộng rãi, có rất nhiều phiên bản cài đặt được xây dựng dành cho các mục đích khác nhau [1, 2, 9, 11, 12, 13, 21, 22, 23, 24]. Tuy nhiên có một phiên bản mã nguồn mở được do chính tác giả của đồ thị thuộc tính mã nguồn, Fabian Yamaguchi, đích thân phát triển và duy trì có tên Joern [9].

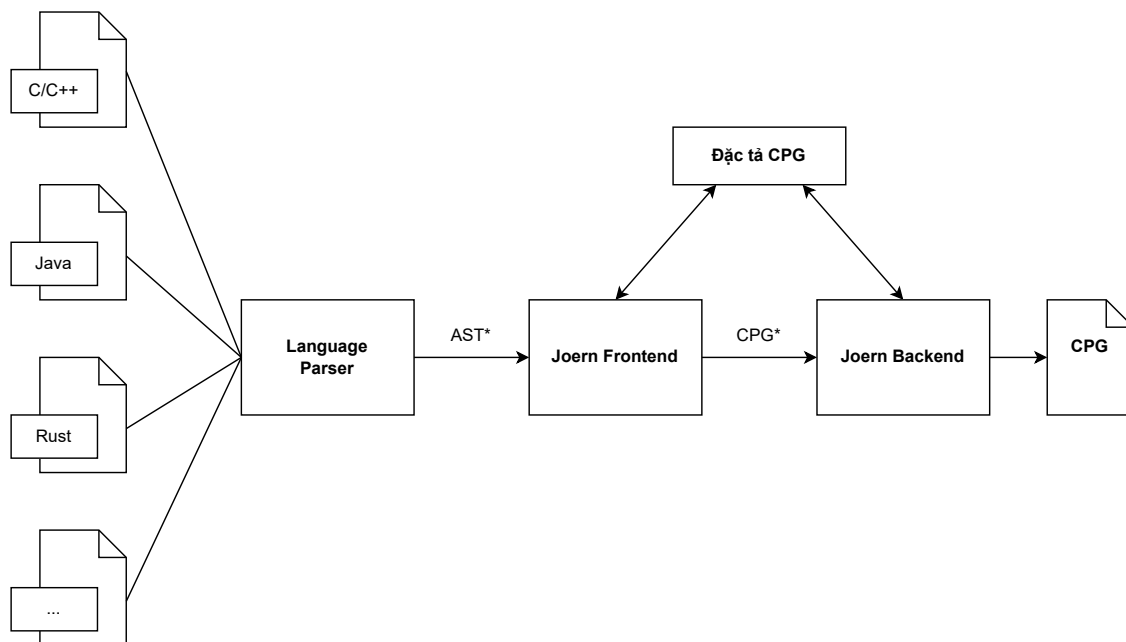
Dự án CPG [6, 8] cho phép biểu diễn mã nguồn của các ngôn ngữ lập trình khác nhau dưới dạng đồ thị. Cho đến nay, trọng tâm là Java và C/C++ nhưng hỗ trợ thử nghiệm cho Python, Go và TypeScript cũng có sẵn. Mục tiêu của dự án là cung cấp một cách biểu diễn mã nguồn không phụ thuộc vào ngôn ngữ. Điều này cho phép chuyên gia bảo mật xác định các lỗ hổng hoặc lỗi. Hơn nữa, thư viện CPG bao gồm một cách để lưu trữ đồ thị trong neo4j2 và làm cho đồ thị có thể truy cập qua giao diện dòng lệnh. Trong một số trường hợp, thư viện cũng có thể đánh giá giá trị mà một nút có thể giữ. Tất cả những điều này cho phép chuyên gia bảo mật viết các truy vấn tùy chỉnh cho cơ sở dữ liệu đồ thị hoặc cách biểu diễn CPG trong bộ nhớ. Thư viện CPG được thiết kế để cho phép tái sử dụng các truy vấn này giữa tất cả các ngôn ngữ lập trình được hỗ trợ. Để đạt được mục tiêu này, thư viện CPG thực hiện một hệ thống phân cấp lớp đầy đủ, bao gồm các loại câu lệnh và biểu thức khác nhau. CPG mã hóa thông tin như hệ thống phân cấp lớp của mã đang được phân tích, đồ thị luồng điều khiển và đồ thị cuộc gọi trong một đồ thị duy nhất. Thiết kế hiện tại chủ yếu nhằm vào các ngôn ngữ lập trình hướng đối tượng. Để đối phó với khả năng thiếu một số đoạn mã hoặc lỗi trong mã, thư viện có khả năng chịu lỗi với mã không đầy đủ, không biên dịch được và thậm chí ở mức độ nào đó còn không chính xác.

Các thành phần cấu thành đồ thị thuộc tính mã nguồn bao gồm:

- **Các nút và loại của chúng:** Các nút đại diện cho các thành phần của chương trình. Điều này bao gồm các cấu trúc ngôn ngữ cấp thấp như phương thức, biến, và cấu trúc điều khiển, cũng như các cấu trúc cấp cao hơn như điểm cuối HTTP hoặc các kết quả phân tích. Mỗi nút có một loại, loại này chỉ ra loại thành phần chương trình mà nút đó đại diện, ví dụ, một nút với loại METHOD đại diện cho một phương thức, trong khi một nút với loại LOCAL đại diện cho khai báo của một biến cục bộ.
- **Cạnh có nhãn:** Quan hệ giữa các thành phần chương trình được biểu diễn thông qua các cạnh giữa các nút tương ứng của chúng. Ví dụ, để biểu thị rằng một phương thức chứa một biến cục bộ, chúng ta có thể tạo một cạnh với nhãn CONTAINS từ nút của phương thức đến nút của biến cục bộ. Bằng cách sử dụng các cạnh có nhãn, chúng ta có thể biểu diễn nhiều loại quan hệ khác nhau trong cùng một đồ thị. Hơn nữa, các cạnh có hướng để biểu thị, ví dụ, rằng phương thức chứa biến cục bộ nhưng không phải ngược lại. Nhiều cạnh có thể tồn tại giữa cùng hai nút.
- **Cặp khóa-giá trị:** Các nút mang các cặp khóa-giá trị (thuộc tính), trong đó các khóa hợp lệ phụ thuộc vào loại nút. Ví dụ, một phương thức có ít nhất tên và chữ ký, trong khi một khai báo biến cục bộ có ít nhất tên và loại của biến được khai báo.

Tóm lại, đồ thị thuộc tính mã nguồn là các đồ thị có hướng, được gán nhãn cạnh, và chứa các thuộc tính, và chúng tôi khẳng định rằng mỗi nút mang ít nhất một thuộc tính chỉ ra loại của nó. Điều này giúp cho việc phân tích mã nguồn trở nên dễ dàng và hiệu quả hơn, đồng thời mở ra nhiều khả năng cho việc tìm kiếm và phát hiện các mẫu lặp trình trong các cơ sở mã nguồn lớn.

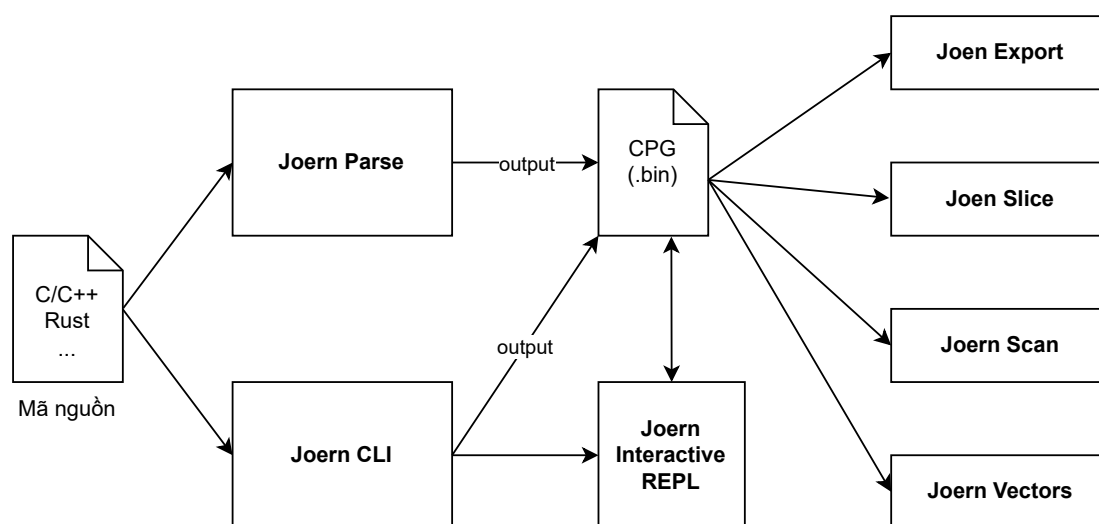
Joern Backend, Joern Frontend



Hình 2.6: Cách hoạt động của công cụ Joern.

2.3.2 Bộ công cụ của Joern

NOTE: Nhấn mạnh Joern hiện tại chưa hỗ trợ Rust, chủ yếu hỗ trợ cho C/C++, Java là cực mạnh. Vẽ biểu đồ đầu vào cpg.bin cho các công cụ khác của joern như



Hình 2.7: Các công cụ xung quanh Joern.

Joern là một nền tảng mạnh mẽ dành cho việc phân tích mã nguồn, bytecode và

mã nhị phân. Công cụ này tạo ra các đồ thị thuộc tính mã nguồn (code property graphs), một cách biểu diễn đồ thị của mã giúp cho việc phân tích mã nguồn đa ngôn ngữ trở nên dễ dàng hơn. Các đồ thị thuộc tính mã nguồn được lưu trữ trong một cơ sở dữ liệu đồ thị tùy chỉnh, cho phép khai thác mã nguồn thông qua các truy vấn tìm kiếm được xây dựng trong một ngôn ngữ truy vấn đặc thù dựa trên Scala. Joern được phát triển với mục tiêu cung cấp một công cụ hữu ích cho việc khám phá lỗ hổng bảo mật và nghiên cứu phân tích chương trình tĩnh.

Joern hỗ trợ các nhà phát triển và nhà nghiên cứu trong việc tìm kiếm và xác định các điểm yếu tiềm ẩn trong mã nguồn, giúp nâng cao chất lượng và bảo mật của phần mềm. Bên cạnh đó, Joern còn có khả năng phân tích mã nguồn đa ngôn ngữ, giúp các nhóm phát triển có thể làm việc với nhiều ngôn ngữ lập trình khác nhau mà không gặp trở ngại về công cụ. Với khả năng truy vấn mạnh mẽ và linh hoạt, Joern đã trở thành một công cụ quan trọng trong việc phân tích mã nguồn và phát hiện các lỗ hổng bảo mật. Bạn có thể tìm hiểu thêm về Joern tại địa chỉ <https://joern.io/>.

Joern hỗ trợ nhiều ngôn ngữ lập trình khác nhau. Các ngôn ngữ được hỗ trợ bao gồm C/C++, Java, JavaScript, Python, x86/x64, JVM Bytecode, Kotlin, PHP, Rust, Swift, Ruby và C#. Điều này cho thấy Joern có khả năng phân tích mã nguồn đa ngôn ngữ, giúp các nhà phát triển và nhà nghiên cứu có thể làm việc với nhiều ngôn ngữ lập trình khác nhau mà không gặp trở ngại về công cụ. Tuy nhiên, Joern chưa hỗ trợ cho ngôn ngữ Rust.

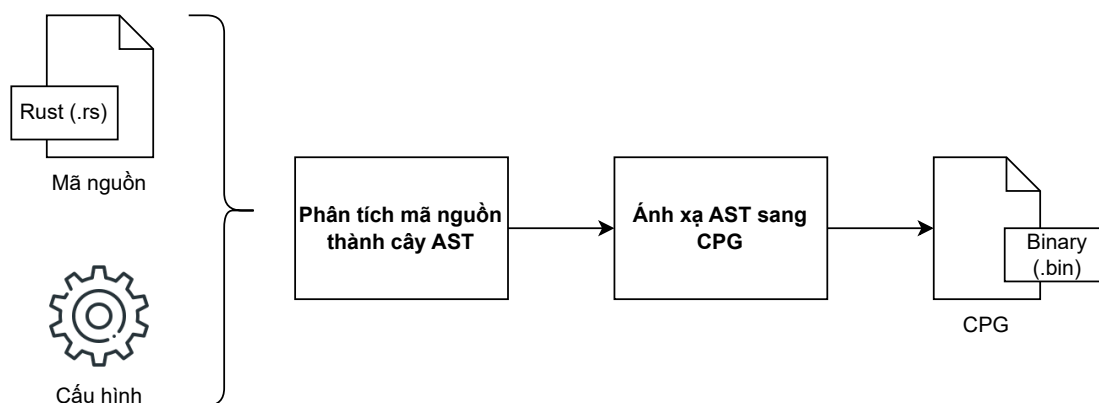
Chương 3

Phân tích mã nguồn Rust

Chương này sẽ trình bày quy trình phân tích mã nguồn cho ngôn ngữ lập trình Rust từ việc xây dựng cây cú pháp trừu tượng, ánh xạ từ cây cú pháp trừu tượng sang đồ thị thuộc tính mã nguồn. Chương này cũng sẽ nêu ra những điểm đã và chưa thực hiện được, khó khăn gặp phải và hướng cải thiện tiếp theo.

3.1 Quy trình tổng quan

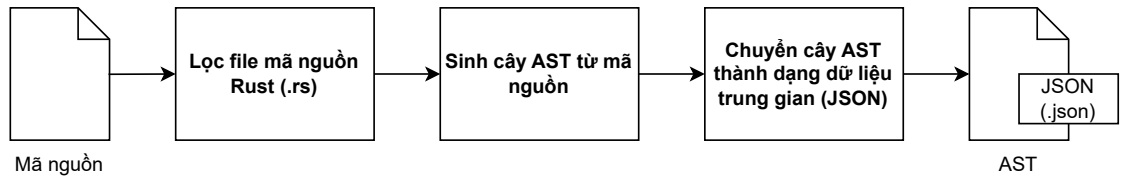
Mục tiêu của công cụ là phân tích mã nguồn Rust và xây dựng đồ thị thuộc tính mã nguồn (CPG) biểu diễn mã nguồn đó. Hình 3.1 thể hiện các bước xây dựng cây CPG từ các tệp mã nguồn Rust.



Hình 3.1: Quy trình phân tích mã nguồn Rust.

Mã nguồn Rust sẽ được trình phân tích và sinh ra cây AST tương ứng với từng tệp mã nguồn. Sau đó, cây cú pháp trừu tượng sẽ được ánh xạ sang đồ thị thuộc tính mã nguồn. Cuối cùng, đồ thị CPG sẽ được lưu dưới dạng cơ sở dữ liệu đồ thị phục vụ mục đích của người dùng.

3.2 Xây dựng cây cú pháp trừu tượng

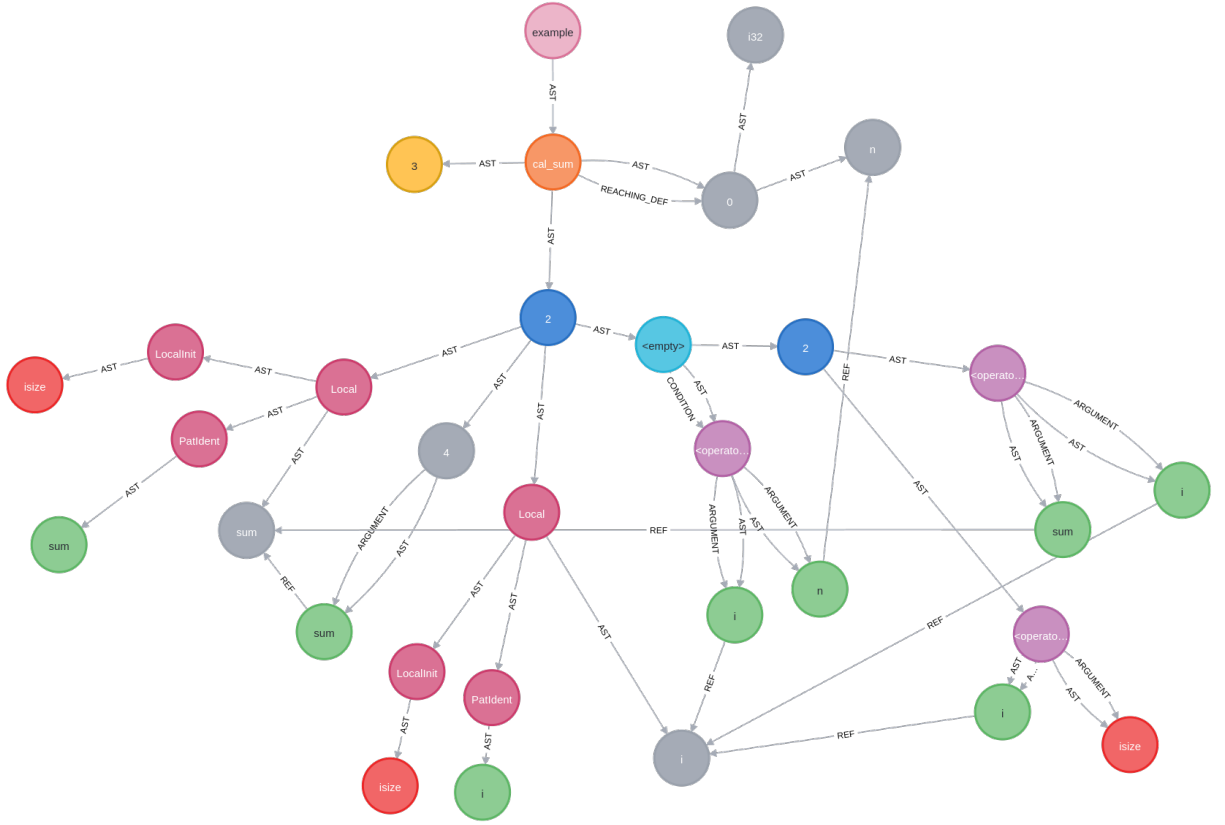


Hình 3.2: Quy trình xây dựng cây cú pháp trừu tượng.

Quy trình xây dựng cây cú pháp trừu tượng bao gồm ba bước được thể hiện trong Hình 3.2, chi tiết các bước được tiến hành như sau.

1. Từ thư mục của dự án, lọc lấy các tệp mã nguồn Rust (các tệp có đuôi là `.rs`).
2. Với mỗi tệp mã nguồn, sử dụng thư viện *syn* [20] để phân tích thành cây cú pháp trừu tượng (AST) ứng với tệp đó. *syn* là 1 thư viện phân tích mã nguồn thành cây cú pháp được sử dụng trong cộng đồng Rust và kể cả tính năng *procedural macro* [14] của Rust cũng cần thư viện này để cài đặt. Tính tới thời điểm hiện tại Rust không có đặc tả ngôn ngữ chính thức, cộng đồng sang sử dụng *Rust Reference* [16] coi như phiên bản sát nhất so với một đặc tả ngữ chính thức. Thư viện *syn* xây dựng các cây cú pháp trừu tượng dựa trên *Rust Reference*.
3. Chuyển đổi cây cú pháp trừu tượng từ ngôn ngữ Rust sang định dạng JSON. Do Joern có thể được sử dụng cho nhiều ngôn ngữ và Joern Frontend không phụ thuộc vào một công cụ Language Parser của một ngôn ngữ nhất định, do vậy cần có định dạng dữ liệu trung gian để chuyển đổi cây AST của Language Parser sang ngôn ngữ Scala của Joern Frontend. Các kiểu dữ liệu trung gian phổ biến như JSON, XML, YAML, trong đó JSON được lựa chọn bởi tính đơn giản, dễ chuyển đổi.

Trong quá trình xây dựng cây cú pháp trừu tượng, các thành phần mã nguồn trong Rust sẽ được phân tích thành dạng cây tương ứng. Dưới đây là một số hình ảnh ví dụ về cây cú pháp trừu tượng ứng với các thành phần mã nguồn này.



Hình 3.4: Ví dụ đồ thị thuộc tính mã nguồn.

Ví dụ về biểu diễn một kiểu cấu trúc được thể hiện trong hình 3.4. Nút TypeSpecification đại diện cho định nghĩa kiểu. Nút Identifier là nút định danh biểu diễn tên kiểu. Nút StructType biểu diễn kiểu cấu trúc. Nút FieldList có các nút con là Field biểu diễn các thuộc tính của kiểu này.

Các nút trong cây cú pháp mã nguồn của Go được liệt kê trong bảng 3.1 dưới đây.

STT	Tên nút	Mô tả
1	Abi	The binary interface of a function: 'extern "C" '.
2	AngleBracketed GenericArguments	Angle bracketed arguments of a path segment: the '<K, V>' in HashMap<K, V>.
3	Arm	One arm of a 'match' expression: '0..=10 => return true; '.
4	AssocConst	An equality constraint on an associated constant: 'the PANIC = false in Trait<PANIC = false>'.
5	AssocType	A binding (equality constraint) on an associated type: 'the Item = u8 in Iterator<Item = u8>'.
6	Attribute	An attribute, like '#[repr(transparent)]'.
7	BareFnArg	An argument in a function type: 'the usize in fn(usize) -> bool'.

8	BareVariadic	The variadic argument of a function pointer like <code>fn(usize, ...)</code> .
9	Block	A braced block containing Rust statements.
10	BoundLifetimes	A set of bound lifetimes: <code>'for<'a, 'b, 'c>'</code> .
11	ConstParam	A const generic parameter: <code>'const LENGTH: usize'</code> .
12	Constraint	An associated type bound: <code>'Iterator<Item: Display>'</code> .
13	DataEnum	An enum input to a <code>'proc_macro_derive'</code> macro.
14	DataStruct	A struct input to a <code>'proc_macro_derive'</code> macro.
15	DataUnion	An untagged union input to a <code>'proc_macro_derive'</code> macro.
16	DeriveInput	Data structure sent to a <code>'proc_macro_derive'</code> macro.
17	Error	Error returned when a Syn parser cannot parse the input tokens.
18	ExprArray	A slice literal expression: <code>'[a, b, c, d]'</code> .
19	ExprAssign	An assignment expression: <code>'a = compute()'</code> .
20	ExprAsync	An async block: <code>'async ... '</code> .
21	ExprAwait	An await expression: <code>'fut.await'</code> .
22	ExprBinary	A binary operation: <code>'a + b, a += b'</code> .
23	ExprBlock	A blocked scope: <code>'... '</code> .
24	ExprBreak	A <code>'break'</code> , with an optional label to break and an optional expression.
25	ExprCall	A function call expression: <code>'invoke(a, b)'</code> .
26	ExprCast	A cast expression: <code>'foo as f64'</code> .
27	ExprClosure	A closure expression: <code>'[a, b] + a + b'</code> .
28	ExprConst	A const block: <code>'const ... '</code> .
29	ExprContinue	A <code>'continue'</code> , with an optional label.
30	ExprField	Access of a named struct field <code>'obj.k'</code> or unnamed tuple struct field <code>'obj.0'</code> .
31	ExprForLoop	A for loop: <code>'for pat in expr ... '</code> .
32	ExprGroup	An expression contained within invisible delimiters.
33	ExprIf	An if expression with an optional else block: <code>if expr ... else ...</code> .
34	ExprIndex	A square bracketed indexing expression: <code>vector[2]</code> .
35	ExprInfer	The inferred value of a const generic argument, denoted <code>_</code> .
36	ExprLet	A let guard: <code>let Some(x) = opt.</code>
37	ExprLit	A literal in place of an expression: <code>1, "foo".</code>
38	ExprLoop	Conditionless loop: <code>loop ...</code> .
39	ExprMacro	A macro invocation expression: <code>format!("{}", q).</code>
40	ExprMatch	A match expression: <code>match n Some(n) => , None => .</code>
41	ExprMethodCall	A method call expression: <code>x.foo::<T>(a, b).</code>

42	ExprParen	A parenthesized expression: (a + b).
43	ExprPath	A path like std::mem::replace possibly containing generic parameters and a qualified self-type.
44	ExprRange	A range expression: 1..2, 1.., ..2, 1..=2, ..=2.
45	ExprRawAddr	Address-of operation: &raw const place or &raw mut place.
46	ExprReference	A referencing operation: &a or &mut a.
47	ExprRepeat	An array literal constructed from one repeated element: [0u8; N].
48	ExprReturn	A return, with an optional value to be returned.
49	ExprStruct	A struct literal expression: Point { x: 1, y: 1 }.
50	ExprTry	A try-expression: expr?.
51	ExprTryBlock	A try block: try { ... }.
52	ExprTuple	A tuple expression: (a, b, c, d).
53	ExprUnary	A unary operation: !x, x.
54	ExprUnsafe	An unsafe block: unsafe { ... }.
55	ExprWhile	A while loop: while expr { ... }.
56	ExprYield	A yield expression: yield expr.
57	Field	A field of a struct or enum variant.
58	FieldPat	A single field in a struct pattern.
59	FieldValue	A field-value pair in a struct literal.
60	FieldsNamed	Named fields of a struct or struct variant such as Point { x: f64, y: f64 }.
61	FieldsUnnamed	Unnamed fields of a tuple struct or tuple variant such as Some(T).
62	File	A complete file of Rust source code.
63	ForeignItemFn	A foreign function in an extern block.
64	ForeignItemMacro	A macro invocation within an extern block.
65	ForeignItemStatic	A foreign static item in an extern block: static ext: u8.
66	ForeignItemType	A foreign type in an extern block: type void.
67	Generics	Lifetimes and type parameters attached to a declaration of a function, enum, trait, etc.
68	Ident	A word of Rust code, which may be a keyword or legal variable name.
69	ImplGenerics	Returned by Generics::split_for_impl.
70	ImplItemConst	An associated constant within an impl block.
71	ImplItemFn	An associated function within an impl block.
72	ImplItemMacro	A macro invocation within an impl block.
73	ImplItemType	An associated type within an impl block.
74	Index	The index of an unnamed tuple struct field.

75	ItemConst	A constant item: <code>‘const MAX: u16 = 65535’</code> .
76	ItemEnum	An enum definition: <code>‘enum Foo<A, B> A(A), B(B) ‘</code> .
77	ItemExternCrate	An extern crate item: <code>‘extern crate serde’</code> .
78	ItemFn	A free-standing function: <code>‘fn process(n: usize) -> Result<(), ... ‘</code> .
79	ItemForeignMod	A block of foreign items: <code>‘extern "C" ... ‘</code> .
80	ItemImpl	An impl block providing trait or associated items: <code>‘impl<A> Trait for Data<A> ... ‘</code> .
81	ItemMacro	A macro invocation, which includes <code>‘macro_rules!’</code> definitions.
82	ItemMod	A module or module declaration: <code>‘mod m or mod m ... ‘</code> .
83	ItemStatic	A static item: <code>‘static BIKE: Shed = Shed(42)’</code> .
84	ItemStruct	A struct definition: <code>‘struct Foo<A> x: A ‘</code> .
85	ItemTrait	A trait definition: <code>‘pub trait Iterator ... ‘</code> .
86	ItemTraitAlias	A trait alias: <code>‘pub trait SharableIterator = Iterator + Sync’</code> .
87	ItemType	A type alias: <code>‘type Result<T> = std::result::Result<T, MyError>’</code> .
88	ItemUnion	A union definition: <code>‘union Foo<A, B> x: A, y: B ‘</code> .
89	ItemUse	A use declaration: <code>‘use std::collections::HashMap’</code> .
90	Label	A lifetime labeling a for, while, or loop.
91	Lifetime	A Rust lifetime: <code>‘a’</code> .
92	LifetimeParam	A lifetime definition: <code>‘a: ’b + ’c + ’d’</code> .
93	LitBool	A boolean literal: <code>‘true or false’</code> .
94	LitByte	A byte literal: <code>‘b’f’</code> .
95	LitByteStr	A byte string literal: <code>‘b"foo"’</code> .
96	LitCStr	A null-terminated C-string literal: <code>‘c"foo"’</code> .
97	LitChar	A character literal: <code>‘a’</code> .
98	LitFloat	A floating point literal: <code>‘1f64 or 1.0e10f64’</code> .
99	LitInt	An integer literal: <code>‘1 or 1u16’</code> .
100	LitStr	A UTF-8 string literal: <code>“foo”</code> .
101	Local	A local let binding: <code>‘let x: u64 = s.parse()?’</code> .
102	LocalInit	The expression assigned in a local let binding, including optional diverging else block.
103	Macro	A macro invocation: <code>‘println!("{}", mac)’</code> .
104	MetaList	A structured list within an attribute, like <code>‘derive(Copy, Clone)’</code> .
105	MetaNameValue	A name-value pair within an attribute, like <code>‘feature = "nightly”’</code> .

106	Parenthesized GenericArguments	Arguments of a function path segment: the (A, B) -> C in 'Fn(A, B) -> C'.
133	PatConst	A pattern that contains a constant.
134	PatIdent	A pattern that matches an identifier or wildcard: 'mut var @ pat'.
135	PatLit	A pattern that matches a literal value.
136	PatMacro	A pattern that comes from an inline macro.
137	PatOr	A pattern that matches any one of a list of patterns.
138	PatParen	A pattern enclosed in parentheses.
139	PatPath	A pattern that matches a path to a constant.
140	PatRange	A pattern that matches a range of values.
141	PatReference	A pattern that matches a reference.
142	PatRest	A pattern that matches the rest pattern: '..'.
143	PatSlice	A pattern that matches a slice of values.
144	PatStruct	A pattern that matches a struct.
145	PatTuple	A pattern that matches a tuple.
146	PatTupleStruct	A pattern that matches a tuple struct.
147	PatType	A pattern that matches a type.
148	PatWild	A wildcard pattern: '_'.
149	Path	A path like 'std::mem::replace', possibly containing generic parameters and a self-type.
150	PathSegment	A segment of a path: 'std', 'std::mem', 'std::mem::replace'.
151	PreciseCapture	Specifies precise capture behavior for closures.
152	PredicateLifetime	A lifetime predicate in a where clause.
153	PredicateType	A type predicate in a where clause.
154	QSelf	The portion of a path before the '::', if the path is qualified.
155	Receiver	The self argument of an associated method: '&self' or '&mut self'.
156	Signature	A function signature in a trait or impl block.
157	StmtMacro	A macro invocation as a statement: 'println!("Hello, world!")'.
158	TraitBound	A trait bound on a type parameter or associated type.
159	TraitItemConst	An associated constant in a trait.
160	TraitItemFn	An associated function in a trait.
161	TraitItemMacro	A macro invocation in a trait.
162	TraitItemType	An associated type in a trait.
163	TurboFish	A turbo-fish, e.g., 'collect::<Vec<_>>()'.
164	TypeArray	An array type: '[T; n]'.
165	TypeBareFn	A bare function type: 'fn(usize) -> bool'.

166	TypeGenerics	A generic type parameter in a type definition.
167	TypeGroup	A group of types.
168	TypeImplTrait	An ‘impl Trait’ type in return position.
169	TypeInfer	An inferred type: ‘_’.
170	TypeMacro	A macro in the type position.
171	TypeNever	The never type: !.
172	TypeParam	A generic type parameter: T: Into<String>.
173	TypeParen	A parenthesized type equivalent to the inner type.
174	TypePath	A path like std::slice::Iter, optionally qualified with a self-type as in <Vec<T> as SomeTrait>::Associated.
175	TypePtr	A raw pointer type: const T or mut T.
176	TypeReference	A reference type: &’a T or &’a mut T.
177	TypeSlice	A dynamically sized slice type: [T].
178	TypeTraitObject	A trait object type dyn Bound1 Bound2 + Bound3 where Bound is a trait or a lifetime.
179	TypeTuple	A tuple type: (A, B, C, String).
180	UseGlob	A glob import in a use item: .
181	UseGroup	A braced group of imports in a use item: A, B, C.
182	UseName	An identifier imported by a use item: HashMap.
183	UsePath	A path prefix of imports in a use item: std::..
184	UseRename	An renamed identifier imported by a use item: HashMap as Map.
185	Variadic	The variadic argument of a foreign function.
186	Variant	An enum variant.
187	VisRestricted	A visibility level restricted to some path: pub (self) or pub (super) or pub (crate) or pub (in some::module).
188	WhereClause	A where clause in a definition: where T: Deserialize<’de>, D: ’static.

Bảng 3.1: Các nút trong cú pháp mã nguồn của Rust.

3.3 Xây dựng đồ thị thuộc tính mã nguồn (CPG)

Mục này sẽ trình bày quá trình công cụ phân tích ánh xạ cây cú pháp trừu tượng thành cây đồ thị thuộc tính mã nguồn.

3.3.1 Các loại đỉnh và cạnh của đồ thị thuộc tính mã nguồn CPG

Đồ thị thuộc tính mã nguồn là dạng cấu trúc dữ liệu được kết hợp giữa cây cú pháp trừu tượng, đồ thị luồng điều khiển và đồ thị phụ thuộc chương trình. Đồ thị này biểu diễn các đỉnh là các nút trong cây cú pháp trừu tượng và các cạnh của đồ thị biểu diễn mối quan hệ giữa chúng. Đồ thị này giúp theo dõi các luồng điều khiển, các phụ thuộc trong mã nguồn.

Bảng 3.2 dưới đây mô tả các đỉnh trong đồ thị thuộc tính mã nguồn

STT	Tên nút	Mô tả
1	META_DATA	Nút này chứa siêu dữ liệu (metadata) của đồ thị. Một đồ thị CPG phải có và chỉ có duy nhất một nút META_DATA.
2	FILE	Nút biểu diễn một tệp mã nguồn. Với mỗi tệp tin trong mã nguồn, đồ thị sẽ có chính xác một nút FILE để biểu diễn tệp mã nguồn đó.
3	NAMESPACE	Nút này biểu diễn một không gian tên (namespace) hoặc một gói (package), đối với Go, nút này biểu diễn các gói trong mã nguồn.
4	NAMESPACE_BLOCK	Nút này biểu diễn một tham chiếu đến không gian tên (namespace) tương ứng. Nó chứa các khối mã nguồn được định nghĩa dưới không gian tên (namespace) tương ứng.
5	METHOD	Biểu diễn các hàm, phương thức hoặc biểu thức lambda.
6	PARAMETER_IN	Nút đại diện cho một tham số đầu vào của hàm.
7	PARAMETER_OUT	Nút đại diện cho một tham số đầu ra của hàm.
8	METHOD_RETURN	Nút này biểu diễn kiểu trả về của một hàm.
9	MEMBER	Biểu diễn thuộc tính của một kiểu cấu trúc (struct).
10	TYPE	Nút đại diện cho một thể hiện (instance) của một kiểu dữ liệu.
11	TYPE_ARGUMENT	Biểu diễn đối số của một kiểu dữ liệu, có thể hình dung giống như các kiểu dữ liệu generic trong JAVA.
12	TYPE_DECL	Biểu diễn một khai báo kiểu dữ liệu ví dụ như khai báo một kiểu cấu trúc (struct) hay khai báo một giao diện (interface).
13	BLOCK	Biểu diễn một khối mã nguồn.
14	CALL	Biểu diễn một lời gọi hàm.
15	CONTROL_STRUCTURE	Biểu diễn một cấu trúc điều khiển ví dụ như câu lệnh if else, vòng lặp for, cấu trúc switch case.
16	IDENTIFIER	Nút định danh biểu diễn tham chiếu đến một biến.
17	JUMP_LABEL	Biểu diễn các cấu trúc nhảy như BREAK, CONTINUE, GOTO.
18	LITERAL	Biểu diễn một hằng số như số nguyên hoặc chuỗi.
19	LOCAL	Biểu diễn một biến cục bộ.

20	METHOD_REF	Biểu diễn một tham chiếu đến hàm khi hàm đó được dùng làm tham số cho một lời gọi.
21	MODIFIER	Biểu diễn một bộ chỉnh sửa (modifier) như static, public hay private.
22	RETURN	Biểu diễn một chỉ thị trả về (return instruction).
23	TYPE_REF	Biểu diễn tham chiếu đến một kiểu dữ liệu.

Bảng 3.2: Các đỉnh trong đồ thị thuộc tính mã nguồn (CPG)

Bảng 3.3 liệt kê các quan hệ trong đồ thị thuộc tính mã nguồn (CPG)

STT	Tên nút	Mô tả
1	SOURCE_FILE	Quan hệ biểu diễn một nút là tệp gốc của một nút khác. Quan hệ cha con (biểu diễn quan hệ trong cây cú pháp mã nguồn (AST)).
2	AST	Cây cú pháp trừu tượng (Abstract Syntax Tree).
3	CALL	Quan hệ gọi, dùng để biểu diễn quan hệ giữa hàm/phương thức với nút gọi hàm/phương thức đó.
4	CONTAINS	Quan hệ bao gồm.
5	CONDITION	Quan hệ điều kiện, dùng để biểu diễn quan hệ giữa khối điều khiển như if else, switch hay for với biểu thức điều kiện của chúng.
6	ARGUMENT	Quan hệ đối số kết nối các điểm gọi (kiểu nút CALL) với các đối số của chúng, cũng như các nút RETURN với các biểu thức trả về.
7	RECEIVER	Quan hệ kết nối các điểm gọi (CALL) tới đối số nhận (receiver argument) của chúng. Một đối số nhận là một đối tượng mà phương thức đó thuộc về (có thể hiểu đối số nhận ở đây là con trỏ 'this' trong Java).
8	CFG	Quan hệ này biểu diễn luồng điều khiển từ nút nguồn đến nút đích.
9	DOMINATE	Quan hệ biểu diễn tính kiểm soát (dominate) ngay lập tức từ nút nguồn đến nút đích.
10	POST_DOMINATE	Quan hệ biểu diễn nút nguồn kiểm soát (dominate) ngay lập tức sau nút đích.
11	CDG	Quan hệ biểu diễn nút đích phụ thuộc điều khiển vào nút nguồn.
12	REACHING_DEF	Quan hệ biểu diễn một biến được tạo ra từ nút nguồn và không bị gán lại giá trị trên đường đi tới nút đích.
13	CONTAINS	Quan hệ này kết nối một nút tới hàm/phương thức chứa nó.

14	EVAL_TYPE	Quan hệ kết nối một nút tới nút kiểu dữ liệu của nút đó.
15	PARAMETER_LINK	Quan hệ kết nối một nút tham số đầu vào với nút tham số đầu ra tương ứng của hàm/phương thức đó.

Bảng 3.3: Các cạnh trong đồ thị thuộc tính mã nguồn (CPG).

3.3.2 Chuyển hóa cây cú pháp trừu tượng sang đồ thị thuộc tính mã nguồn

Với các thông tin có được về cây cú pháp mã nguồn được lưu trong các tệp JSON, đồ thị thuộc tính mã nguồn sẽ được dựng lên từ những thông tin đó. Sau đây tôi sẽ mô tả việc ánh xạ các thành phần mã nguồn chính của Rust từ cây cú pháp mã nguồn sang đồ thị thuộc tính mã nguồn.

Ánh xạ tệp (file) và gói (package)

Hình 3.5 biểu diễn ánh xạ từ mã nguồn sang cây cú pháp trừu tượng và ánh xạ sang đồ thị thuộc tính mã nguồn. Cây cú pháp trừu tượng của Rust không có nút biểu diễn gói, thông tin về gói sẽ được lưu trong nút tệp định nghĩa gói đó. Khi ánh xạ nút tệp sang đồ thị thuộc tính mã nguồn sẽ biểu diễn được nút FILE và NAMESPACE tương ứng. Nút NAMESPACE_BLOCK biểu diễn thân của gói, nút này đóng vai trò làm nút cha của các định nghĩa bên trong thân của gói.

3.4 Thực nghiệm trên các tính năng đã hỗ trợ

3.4.1 if let

```

1 fn basic() {
2     let number: Option<i32> = None;
3     let i_like_letters = false;
4
5     if let Some(i) = number {
6         println!("Matched {:?}!", i);
7     } else if i_like_letters {
8         println!("Didn't match a number. It's a letter!");
9     } else {
10        println!("I don't like numbers and letters");
11    }
12 }
```

Đoạn mã 3.1: Ví dụ mã nguồn cho

3.4.2 while let

```
1 fn main() {
2     let mut optional = Some(0);
3
4     while let Some(i) = optional {
5         if i > 9 {
6             println!("Greater than 9, quit!");
7             optional = None;
8         } else {
9             println!("`i` is `{:?}`. Try again.", i);
10            optional = Some(i + 1);
11        }
12    }
13 }
```

Đoạn mã 3.2: Ví dụ mã nguồn cho

3.4.3 match

```
1  enum Color {
2      Red,
3      Blue(u32, u32, u32),
4      Green {
5          red: u32,
6          green: u32,
7          blue: u32,
8      },
9  }
10
11 fn main() {
12     let color = Color::Blue(0, 0, 255);
13
14     match color {
15         Color::Red =>
16             println!("The color is Red!")
17         Color::Blue(r, g, b) =>
18             println!("Red: {}, green: {}, and blue: {}!", r, g, b)
19         Color::Green {red, green, blue} =>
20             println!(
21                 "Red: {}, green: {}, and blue: {}!",
22                 red, green, blue
23             ),
24     }
25 }
```

Đoạn mã 3.3: Ví dụ mã nguồn cho

3.4.4 lifetime

```
1 fn f<'a, 'b, 'c, 'd: 'c>(x: &'a i32, mut y: &'b i32, z: &'c i32)
2 where
3     'a: 'b,
4 {
5     // ...
6 }
7
8 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
9     if x.len() > y.len() {
10         x
11     } else {
12         y
13     }
14 }
```

Đoạn mã 3.4: Ví dụ mã nguồn cho

3.5 Hạn chế

3.5.1 Path

3.5.2 Macro

3.5.3 Module

3.5.4 Type Argument match Type Parameter

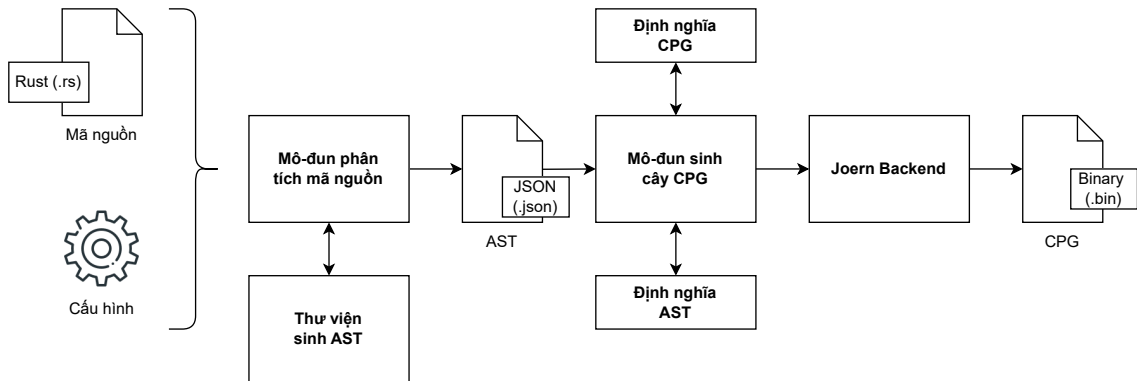
Chương 4

Cài đặt công cụ và thực nghiệm

Chương này sẽ tập trung trình bày về quá trình thực nghiệm và đánh giá phương pháp đánh giá điểm thưởng dựa trên thiết kế đã mô tả chi tiết trong phần trước. Phần thực nghiệm sẽ đi sâu vào việc áp dụng phương pháp để cải thiện hiệu quả công cụ ARAT-RL và so sánh hiệu suất của phương pháp mới với phương pháp đánh giá điểm thưởng mặc định của ARAT-RL. Dựa trên kết quả thu được, tôi sẽ đưa ra các nhận xét và kết luận về giải pháp đề xuất.

4.1 Cài đặt công cụ

Công cụ phân tích mã nguồn Rust được phát triển từ mã nguồn của công cụ Joern. Công cụ được cài đặt bằng ngôn ngữ Rust và Scala. Kiến trúc tổng quan của công cụ được mô tả ở Hình 4.1



Hình 4.1: Kiến trúc công cụ.

Công cụ gồm hai mô-đun là *Rust Parser* và *Joern* trong đó *Joern* là mô-đun chính. Mô-đun GoParse được cài đặt bằng ngôn ngữ Rust. Mô-đun này sẽ nhận đầu vào là đường dẫn đến thư mục dự án chứa mã nguồn Rust và thực hiện việc phân tích mã nguồn thành cú pháp mã nguồn, xử lý kiểu dữ liệu và lưu cây cú pháp mã nguồn vào các tệp JSON. *Joern* được cài đặt bằng ngôn ngữ Scala và được phát triển từ mã nguồn của công cụ *Joern*. Mô-đun thực hiện nhận mã nguồn đầu vào, gọi mô-đun *Rust Parser* để dựng cây cú pháp mã nguồn và sau đó xây dựng đồ thị thuộc tính mã nguồn từ các thông tin trong cây cú pháp mã nguồn. Đầu ra của mô-đun này cũng là đồ thị thuộc tính mã nguồn được lưu dưới dạng tệp nhị phân (.bin). Đây cũng là đầu ra của cả công cụ. Chúng ta có thể dùng một số công cụ được *Joern* cung cấp sẵn để thao tác với đồ thị này như xuất đồ thị dưới nhiều định dạng khác nhau như neo4jcsv, graphml, graphson,

dot bằng công cụ *Joern Export*, thực hiện các câu lệnh truy vấn trên đồ thị hoặc quét đồ thị để tìm lỗ hổng trong mã nguồn bằng công cụ *Joern Scan*.

4.1.1 Mô-đun xây dựng cây cú pháp trừu tượng Rust Parser

Chức năng chính của *Rust Parser* là nhận thông tin đường dẫn đến mã nguồn, lọc các tệp mã nguồn Rust, sinh cây cú pháp trừu tượng, xử lý kiểu dữ liệu cho các nút định danh (identifier) và lưu kết quả vào các tệp JSON. Các thành phần trong mô-đun này bao gồm cli, parser, resolver, ast và util. Chức năng chi tiết của từng thành phần được mô tả như sau:

- Thành phần cli làm nhiệm vụ xử lý và cung cấp giao diện dòng lệnh (CLI), nhận đầu vào là đường dẫn đến dự án Rust, nhận các cấu hình (ví dụ như đường dẫn lưu các tệp đầu ra hoặc danh sách các tệp mã nguồn cần bỏ qua) và gọi các chức năng phân tích.
- Thành phần parser thực hiện nhận thông tin từ thành phần cli, từ đó đọc các tệp mã nguồn của Rust có đuôi là ".rs" từ dự án đầu vào, phân tích các tệp tin này và sinh cây cú pháp trừu tượng.
- Thành phần resolver cung cấp các API phục vụ việc xử lý kiểu dữ liệu.
- Thành phần ast chứa các cấu trúc định nghĩa các loại nút và thuộc tính của từng loại nút của cây cú pháp trừu tượng.
- Thành phần util cung cấp các tiện ích chung bao gồm xử lý tệp tin và xử lý nhật ký (log).

Hình 4.2 biểu diễn các thành phần của mô-đun *Rust Parser*, các mũi tên biểu mối quan hệ sử dụng giữa các thành phần này.

4.1.2 Mô-đun xây dựng đồ thị thuộc tính mã nguồn Joern

Mô-đun *Joern* thực hiện nhiệm vụ xây dựng đồ thị thuộc tính mã nguồn (CPG) từ cây cú pháp trừu tượng AST. Mô-đun sẽ thực hiện đọc thông tin từ các tệp JSON được tạo ra từ mô-đun *Rust Parser* và xây dựng đồ thị thuộc tính mã nguồn (CPG). Mô-đun này được phát triển từ mô-đun *Joern* của công cụ *Joern* nên sử dụng lại một số thành phần có sẵn của *Joern* bao gồm các thành phần console, semanticcpg, codepropertygraph và x2cpg. Dưới đây tôi sẽ mô tả chức năng của từng thành phần:

- Thành phần console thực hiện chức năng tương tự như thành phần cli của mô-đun *Rust Parser*.
- Thành phần rust-language-frontend đảm nhận nhiệm vụ nhận đường dẫn đến mã nguồn từ thành phần console, gọi mô-đun *Rust Parser* để sinh cây cú pháp trừu tượng sau đó đọc các tệp JSON được sinh ra và xây dựng thành đồ thị thuộc tính mã nguồn.
- Thành phần codepropertygraph chứa các lớp định nghĩa các đỉnh và cạnh của đồ thị thuộc tính mã nguồn.
- Thành phần x2cpg cung cấp các tiện ích để ánh xạ các thuộc tính từ các nút của cây cú pháp trừu tượng sang thuộc tính của các đỉnh trong đồ thị thuộc tính mã nguồn.

- Thành phần `semanticcp` chứa các tiện ích cho phép vắn đồ thị thuộc tính mã nguồn. Thành phần này được sử dụng để thực hiện các tác vụ hậu xử lý sau khi đồ thị thuộc tính mã nguồn được xây dựng (ví dụ như bổ sung thông tin hoặc tạo các đỉnh liên kết).

Hình 4.3 biểu diễn các thành phần của mô-đun *Rust Parser*, các mũi tên biểu diễn mối quan hệ sử dụng giữa các thành phần này.

4.2 Tính áp dụng vào thực tế của đồ thị CPG dành cho Rust

4.2.1 RUSTSEC-2021-0086

```
1  const N: usize = 255;
2
3  // Before fix
4  let mut buf: Vec<u8> = Vec::with_capacity(N);
5  unsafe { buf.set_len(N) };
6
7  // After fix
8  let mut buf: Vec<u8> = vec![0; N];
```

Đoạn mã 4.1: Ví dụ mã nguồn cho RUSTSEC-2021-0086

4.2.2 RUSTSEC-2022-0028

```
1 // Before fix
2 pub fn external<'a, C, T>(cx: &mut C, data: T) -> Handle<'a, Self>
3 where
4     C: Context<'a>,
5     T: AsMut<[u8]> + Send,
6 {
7     // ...
8 }
9
10 // After fix
11 pub fn external<'a, C, T>(cx: &mut C, data: T) -> Handle<'a, Self>
12 where
13     C: Context<'a>,
14     T: AsMut<[u8]> + Send + 'static,
15 {
16     // ...
17 }
```

Đoạn mã 4.2: Ví dụ mã nguồn cho RUSTSEC-2022-0028

4.2.3 RUSTSEC-2020-0044

```
1 // Before fix
2 unsafe impl<P> Send for Atom<P> where P: IntoRawPtr + FromRawPtr {}
3 unsafe impl<P> Sync for Atom<P> where P: IntoRawPtr + FromRawPtr {}
4
5 // After fix
6 unsafe impl<P> Send for Atom<P> where P: IntoRawPtr + FromRawPtr + Send {}
7 unsafe impl<P> Sync for Atom<P> where P: IntoRawPtr + FromRawPtr + Send {}
```

Đoạn mã 4.3: Ví dụ mã nguồn cho RUSTSEC-2020-0044

4.2.4 RUSTSEC-2021-0130

```
1 // Before fix
2 pub fn iter<'a>(&'_ self) -> Iter<'a, K, V> {
3     Iter {
4         ptr: unsafe { (*self.head).next },
5     }
6 }
7
8 // After fix
9 pub fn iter(&self) -> Iter<'_, K, V> {
10     Iter {
11         ptr: unsafe { (*self.head).next },
12     }
13 }
```

Đoạn mã 4.4: Ví dụ mã nguồn cho RUSTSEC-2021-0130

Kết luận

Phân tích mã nguồn là một bước quan trọng trong quy trình phát triển phần mềm, giúp phát hiện lỗi, lỗ hổng bảo mật và các vấn đề tiềm ẩn khác trong mã nguồn. Quá trình này cũng hỗ trợ lập trình viên phát hiện lỗi logic, tối ưu mã nguồn và tuân thủ các quy tắc và tiêu chuẩn lập trình, từ đó đảm bảo tính ổn định, khả năng quản lý và bảo trì mã nguồn về lâu dài.

Hiện có khoảng 20 ngôn ngữ lập trình thông dụng trên thế giới, kèm theo đó là hàng trăm công cụ phân tích mã nguồn khác nhau. Khóa luận này đã phát triển thành công một công cụ phân tích mã nguồn cho ngôn ngữ lập trình Rust. Công cụ này phân tích mã nguồn thành đồ thị thuộc tính mã nguồn thay vì chỉ phân tích thành cây cú pháp trừu tượng, vì đồ thị này tổng hợp thông tin từ cây cú pháp trừu tượng, đồ thị luồng điều khiển và đồ thị thuộc tính mã nguồn. Điều này không chỉ cung cấp thông tin về cú pháp mà còn về luồng điều khiển và phụ thuộc dữ liệu trong chương trình.

Công cụ được phát triển dựa trên Joern, kế thừa các tiện ích mạnh mẽ giúp khai thác mã nguồn dựa trên đồ thị thuộc tính mã nguồn trở nên dễ dàng và chuyên sâu hơn. Qua thử nghiệm với các dự án mã nguồn Rust phổ biến, công cụ đã đạt được các mục tiêu đề ra, xử lý thông tin về kiểu dữ liệu, phân tích hầu hết các cú pháp Rust và đưa ra kết quả chi tiết hơn so với Joern. Tuy nhiên, công cụ vẫn chưa xử lý được tất cả các cú pháp của Rust và còn tốn nhiều thời gian phân tích, đặc biệt với các dự án sử dụng nhiều thư viện ngoài.

Trong tương lai, công cụ sẽ được hoàn thiện hơn để xử lý toàn bộ cú pháp Rust. Việc xử lý kiểu dữ liệu sẽ được tối ưu bằng chiến thuật khác, không cần phân tích tất cả các thư viện đi kèm dự án, giúp rút ngắn thời gian phân tích. Công cụ cũng sẽ hỗ trợ các hệ điều hành phổ biến như Windows và MacOS, thay vì chỉ Ubuntu. Với độ chi tiết của đồ thị đầu ra và các tiện ích truy vấn mạnh mẽ, công cụ sẽ là nền tảng mạnh mẽ để phát triển các công cụ phân tích mã nguồn khác, như tìm kiếm lỗ hổng, gợi ý mã nguồn và phát hiện lỗi cú pháp. Đầu ra của công cụ cũng có thể được sử dụng cho các bài toán học máy hoặc học sâu.

Tài liệu tham khảo

- [1] Fraunhofer AISEC. Home - Code Property Graph — fraunhofer-aisec.github.io. <https://fraunhofer-aisec.github.io/cpg/>. [Accessed 19-11-2024].
- [2] Christian Banse, Immanuel Kunz, Angelika Schneider, and Konrad Weiss. Cloud property graph: Connecting cloud security assessments with static code analysis. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 13–19. IEEE, 2021.
- [3] cppreference. RAII - cppreference.com — en.cppreference.com. <https://en.cppreference.com/w/cpp/language/raii>. [Accessed 18-11-2024].
- [4] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.
- [5] googleblog. Memory Safe Languages in Android 13 — security.googleblog.com. <https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html>. [Accessed 18-11-2024].
- [6] Jiaxuan Han, Cheng Huang, Siqi Sun, Zhonglin Liu, and Jiayong Liu. bxnnet: an improved bug localization model based on code property graph and attention mechanism. *Automated Software Engineering*, 30(1):12, 2023.
- [7] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, pages 392–411, 1992.
- [8] O JE and T CT. Why scientists are turning to rust. *Nature*, 588:185, 2020.
- [9] joern. Joern - The Bug Hunter's Workbench — joern.io. <https://joern.io/>. [Accessed 19-11-2024].
- [10] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in rust. *Communications of the ACM*, 64(4):144–152, 2021.
- [11] Wim Keirsgaeter and Willem Visser. Graft: Static analysis of java bytecode with graph databases. In *Conference of the South African Institute of Computer Scientists and Information Technologists 2020*, pages 217–226, 2020.
- [12] Alexander Kuchler and Christian Banse. Representing llvm-ir in a code property graph. In *International Conference on Information Security*, pages 360–380. Springer, 2022.
- [13] plume oss/plume. GitHub - plume-oss/plume: Plume is a code representation benchmarking library with options to extract the AST from Java bytecode and store the result in various graph databases. — github.com. <https://github.com/plume-oss/plume>. [Accessed 19-11-2024].

- [14] ProceduralMacros. Procedural Macros - The Rust Reference — doc.rust-lang.org. <https://doc.rust-lang.org/reference/procedural-macros.html>. [Accessed 19-11-2024].
- [15] rustlang. The Rust Programming Language - The Rust Programming Language — doc.rust-lang.org. <https://doc.rust-lang.org/stable/book/>. [Accessed 18-11-2024].
- [16] rustReference. Introduction - The Rust Reference — doc.rust-lang.org. <https://doc.rust-lang.org/reference/>. [Accessed 19-11-2024].
- [17] Lukas Seidel and Julian Beier. Bringing rust to safety-critical systems in space. *arXiv preprint arXiv:2405.18135*, 2024.
- [18] Ayushi Sharma, Shashank Sharma, Santiago Torres-Arias, and Aravind Machiry. Rust for embedded systems: Current state, challenges and open problems. *arXiv preprint arXiv:2311.05063*, 2023.
- [19] stackoverflow. Stack Overflow Developer Survey 2023 — survey.stackoverflow.co. <https://survey.stackoverflow.co/2023/>. [Accessed 18-11-2024].
- [20] syn. syn - Rust — docs.rs. <https://docs.rs/syn/latest/syn/index.html>. [Accessed 19-11-2024].
- [21] Konrad Weiss and Christian Banse. A language-independent analysis platform for source code. *arXiv preprint arXiv:2203.08424*, 2022.
- [22] wimkeir/graft. GitHub - wimkeir/graft: A static analysis tool for Java programs, based on the theory of code property graphs. — github.com. <https://github.com/wimkeir/graft>. [Accessed 19-11-2024].
- [23] Wang Xiaomeng, Zhang Tao, Wu Runpu, Xin Wei, and Hou Changyu. Cpgva: code property graph based vulnerability analysis by deep learning. In *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*, pages 184–188. IEEE, 2018.
- [24] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.
- [25] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 52–63. IEEE, 2019.
- [26] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.