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 chỉ dẫn chuyên môn mà còn động viên và hỗ trợ tôi vượt qua khó khăn để thực hiện đề tài tốt nghiệp một cách tốt nhất.

Tôi cũng xin cảm ơn thầy Trần Mạnh Cường và thầy Vũ Trọng Thanh cũng như toàn thể anh chị, và các bạn trong phòng thí nghiệm iSE (Khoa Công nghệ thông tin, Trường Đại học Công nghệ, ĐHQGHN) đã ủng hộ, động viên tôi trong quá trình hoàn thành khóa luận. Sự hỗ trợ nhiệt tình của mọi người đã giúp tôi có thêm tự tin trong việc mở rộng kiến thức và kỹ năng thực hành.

Chúc mọi người 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-C-C 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 20 tháng 12 năm 2024

Sinh viên

Tóm tắt

Tóm tắt: Rust đang trở thành một ngôn ngữ lập trình nổi bật và được sử dụng rộng rãi trong những năm gần đây. Nhờ vào cơ chế đảm bảo an toàn bộ nhớ và hiệu suất vượt trội, Rust được rất nhiều dự án lớn lựa chọn làm ngôn ngữ kế thừa và nối tiếp C/C++. 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. Khi các dự án lớn chuyển sang sử dụng Rust, việc đảm bảo mã nguồn an toàn, tránh lỗ hổng bảo mật rất được đề cao. Đã có những nghiên cứu về đảm bảo chất lượng mã nguồn được thực hiện và cho thấy tiềm năng, kết quả thực tiễn nhất định. Các phương pháp đa dạng trong cách tiếp cận từ kiểm chứng, kiểm thử động và phân tích tĩnh. Điểm chung của các phương pháp là đều sử dụng một loại đầu vào riêng biệt của ngôn ngữ Rust. Rust là ngôn ngữ nối tiếp của C/C++, do vậy Rust có khả năng tương thích với C/C++. Các nghiên cứu đã có về đảm bảo chất lượng mã nguồn cho C/C++ hoàn toàn có thể áp dụng được cho Rust. Tuy nhiên, việc sử dụng đầu vào riêng biệt của Rust khiến cho các nghiên cứu đã có phải sửa đổi đáng kể thì mới có thể phù hợp. Điều này làm chậm quá trình mở rộng, chuyển tiếp giữa Rust và C/C++ trong các dự án thực tế. Khóa luận phát triển một công cụ phân tích mã nguồn cho ngôn ngữ lập trình Rust với mục tiêu khắc phục được các hạn chế kế trên. Khóa luận lựa chọn đồ thị thuộc tính mã nguồn làm kiểu biểu diễn trung gian cho mã nguồn Rust. Điều này giúp giải pháp trong khóa luận tương thích với nhiều công cụ và nghiên cứu đã có từ trước. Với bản chất là phân tích tĩnh, đồ thị thuộc tính mã nguồn cho ngôn ngữ Rust có thể áp dụng vào các dự án lớn và tốn ít chi phí. Công cụ thực hiện phân tích ngôn ngữ Rust ở mức độ mã nguồn, thay vì các cấp độ trung gian thấp hơn để không bị mất mát thông tin về các cơ chế đảm bảo an toàn bộ nhớ. Với đầu ra là đồ thị thuộc tính mã nguồn, các thao tác truy vấn và phân tích tự động có thể được thực hiện nhằm phục vụ cho các mục đích khác nhau. Ngoài ra, đồ thị thuộc tính mã nguồn có thể được áp dụng cho các lớp bài toán học máy, học tăng cường để phát hiện lỗ hổng bảo mật cho mã nguồn Rust.

Từ khóa: *Phân tích mã nguồn, đồ thị thuộc tính mã nguồn, ngôn ngữ lập trình Rust*

Abstract

Abstract: Rust has become a widely used programming language in recent years. Thanks to its memory safety features and outstanding performance, Rust has been adopted by many large projects as a successor to C/C++. These projects show that Rust is not just a trend but also a language with long-term potential. As major projects transition to Rust, ensuring safe source code and avoiding security vulnerabilities have become high priorities. Research on Rust source code quality assurance has been conducted, showing potential and delivering some practical results. Various methods, such as formal verification, static analysis, and dynamic testing, are being used. A common feature of these methods is their reliance on Rust-specific input formats. Since Rust is a successor to C/C++, it is compatible with C/C++. Existing research on source code quality assurance for C/C++ can be applied to Rust. However, Rust-specific input requirements mean these studies need significant adjustments to work well, which slows down the transition and integration of Rust with C/C++ in real-world projects. This thesis develops a source code analysis tool for the Rust programming language to address these limitations. The thesis uses code property graphs as an intermediate representation for Rust source code. This approach ensures compatibility with many existing tools and studies. It can be applied to large-scale projects at low cost due to the static analysis nature of code property graph. The tool performs analysis at the source level rather than lower levels to keep all information about Rust's memory safety mechanisms. With code property graphs as the output, automated querying and analysis can be performed for different purposes. Additionally, code property graphs can be applied to machine learning and reinforcement learning tasks to detect security vulnerabilities in Rust source code.

Keywords: *Source code analysis, code property graph, 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 đoạn mã **vii**

Danh mục từ viết tắt **viii**

Chương 1 Đặt vấn đề **1**

Chương 2 Kiến thức cơ sở **5**

2.1 Ngôn ngữ lập trình Rust 5

2.1.1 Cơ chế an toàn của Rust 5

2.1.2 Tính hướng hàm 8

2.2 Các cách biểu diễn mã nguồn 10

2.2.1 Cây cú pháp trừu tượng 10

2.2.2 Đồ thị luồng điều khiển 11

2.2.3 Đồ thị phụ thuộc chương trình 13

2.2.4 Đồ thị thuộc tính mã nguồn 14

2.3 Công cụ Joern 14

2.3.1 Đặc tả đồ thị thuộc tính mã nguồn của Joern 14

2.3.2 Luồng hoạt động của công cụ Joern 16

2.3.3 Bộ công cụ của Joern 18

Chương 3 Thiết kế và cài đặt công cụ **19**
3.1 Luồng hoạt động và cài đặt công cụ 19
3.2 Kiến trúc công cụ 21
3.3 Chuyển đổi các cú pháp Rust sang đồ thị CPG 23
3.3.1 Cú pháp if let 23
3.3.2 Cú pháp while let 26
3.3.3 Cú pháp match 27
3.3.4 Cú pháp lifetime 29

Chương 4 Ứng dụng thực nghiệm và đánh giá **34**
4.1 Ứng dụng phân tích mã nguồn có lỗ hổng bảo mật 34
4.1.1 RUSTSEC-2021-0086 34
4.1.2 RUSTSEC-2022-0028 36
4.1.3 RUSTSEC-2020-0044 37
4.1.4 RUSTSEC-2021-0130 39
4.2 Ứng dụng bài toán học máy trên đồ thị CPG 41
4.3 Hạn chế 43
4.3.1 Chưa hỗ trợ cú pháp Macro 43
4.3.2 Chưa hỗ trợ cơ chế Module 44

Kết luận **45**

Tài liệu tham khảo **46**

Danh sách hình vẽ

2.1 Ví dụ về cây AST cho mã nguồn Rust. 11
2.2 Các thành phần cơ bản trong đồ thị CFG. 12
2.3 Các cấu trúc điều khiển phổ biến trong ngôn ngữ lập trình. 12
2.4 Ví dụ về đồ thị PDG. 13
2.5 Minh họa đồ thị CPG của một đoạn mã nguồn C. 14
2.6 Luồng hoạt động của công cụ Joern. 16
2.7 Các công cụ xung quanh Joern. 18

3.1 Quy trình phân tích mã nguồn Rust. 19
3.2 Kiến trúc công cụ. 21
3.3 Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp if let 3.2. 25
3.4 Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp while let 3.4. . . 27
3.5 Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp match 3.5. 29
3.6 Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp lifetime 3.6. . . . 31
3.7 Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp where 3.7. 32

4.1 Minh họa đồ thị CPG cho đoạn mã nguồn RUSTSEC-2021-0086 4.1. 35
4.2 Minh họa đồ thị CPG cho đoạn mã nguồn RUSTSEC-2022-0028 4.2. 37
4.3 Minh họa đồ thị CPG cho đoạn mã nguồn RUSTSEC-2020-0044 4.3. 38
4.4 Minh họa đồ thị CPG cho đoạn mã nguồn RUSTSEC-2021-0130 4.4. 40

Danh sách đoạn mã

2.1 Ví dụ các khái niệm an toàn trong Rust: (1) ownership, (2) borrowing, (3) lifetime và (4) thread safe. 7
2.2 Ví dụ về biểu thức if else trong Rust. 8
2.3 Ví dụ về cú pháp pattern matching trong Rust. 8
2.4 Ví dụ về mẫu thiết kế Monad trong Rust. 9
2.5 Ví dụ về kiểu dữ liệu đại số trong Rust. 10

3.1 Mã giả cho cú pháp tổng quát của if let. 24
3.2 Ví dụ đoạn mã nguồn cho cú pháp if let. 24
3.3 Ví dụ đoạn mã nguồn cho cú pháp if let tương đương trong C++. . . 25
3.4 Ví dụ đoạn mã nguồn cho cú pháp while let. 26
3.5 Ví dụ đoạn mã nguồn cho cú pháp match. 28
3.6 Ví dụ đoạn mã nguồn cho cú pháp lifetime. 30
3.7 Ví dụ đoạn mã nguồn cho cú pháp lifetime kết hợp cú pháp where. . 32

4.1 Ví dụ đoạn mã nguồn cho RUSTSEC-2021-0086. 35
4.2 Ví dụ đoạn mã nguồn cho RUSTSEC-2022-0028. 36
4.3 Ví dụ đoạn mã nguồn cho RUSTSEC-2020-0044. 37
4.4 Ví dụ đoạn mã nguồn cho RUSTSEC-2021-0130. 39

Danh mục 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 CFG
PDG	Program Dependence Graph	Đồ thị phụ thuộc chương trình
CPG	Code Property Graph	Đồ thị thuộc tính mã nguồn
JSON	JavaScript Object Notation	Ký hiệu đối tượng JavaScript
XML	Extensible Markup Language	Ngôn ngữ đánh dấu mở rộng
YAML	YAML Ain't Markup Language	Ngôn ngữ đánh dấu YAML
OOP	Object-Oriented Programming	Lập trình hướng đối tượng
FP	Functional Programming	Lập trình hướng hàm
DSL	Domain-Specific Language	Ngôn ngữ miền chuyên biệt

Chương 1

Đặt vấn đề

Rust được phát triển bởi Mozilla Foundation và giới thiệu lần đầu vào năm 2006. Phiên bản 1.0 của Rust được công bố vào năm 2015 [39], đánh dấu phiên bản ổn định đầu tiên được đem vào sử dụng. Kể từ đó đến nay, Rust liên tục cải tiến và ngày càng được nhiều dự án sử dụng, đặc biệt là những dự án ở cấp độ hệ thống. Rust được thiết kế nhằm giải quyết các vấn đề về an toàn bộ nhớ, an toàn đa luồng mà ngôn ngữ C/C++ mắc phải và đảm bảo tốc độ, hiệu năng trong việc phát triển phần mềm hệ thống [10]. Rust là một ngôn ngữ lập trình an toàn về bộ nhớ, cung cấp nhiều tính năng mới như ownership (tính sở hữu), borrowing (mượn giá trị), lifetime (vòng đời). Các tính năng trên giúp tránh được một lớp lớn các lỗi điển hình trong ngôn ngữ C/C++ như tràn bộ đệm, sử dụng sau khi giải phóng, giải phóng hai lần, con trỏ chỉ tới địa chỉ null, con trỏ treo. Những tính năng đảm bảo an toàn bộ nhớ trên được áp dụng ngay trong quá trình phát triển, cụ thể là thông qua trình biên dịch. Rust có thể ngăn chặn được các lỗi trên ở thời điểm biên dịch, giúp cho mã nguồn Rust ít lỗi hơn so với mã nguồn C/C++. Với ưu điểm vượt trội, Rust hiện tại đã được tích hợp vào mã nguồn của nhân Linux [16] hay trong phát triển hệ điều hành Android của Google [41]. Với sức ảnh hưởng của Rust, Nhà Trắng đã có một bản báo cáo yêu cầu các phần mềm trong tương lai phải được phát triển bằng một ngôn ngữ an toàn về bộ nhớ [46].

Mặc dù cung cấp các tính năng bảo mật như trên, mã nguồn Rust vẫn tồn tại những nguy cơ tiềm ẩn. Mã nguồn Rust được chia thành hai phần, bao gồm mã an toàn và mã không an toàn. Mã an toàn trong Rust là những đoạn mã sử dụng các tính năng, hàm an toàn mà Rust cung cấp và được trình biên dịch kiểm tra thông qua Borrow Checker (bộ kiểm tra mượn giá trị) [27] và các quy tắc an toàn khác. Tuy nhiên, mã an toàn không đảm bảo an toàn tuyệt đối. Tồn tại những trường hợp phức tạp mà trình biên dịch không phát hiện ra hoặc do các chỉ dẫn không chính xác từ lập trình viên khiến cho trình biên dịch bị đánh lừa. Điển hình như việc sử dụng tính năng Interior Mutability [25] trong Rust cũng có thể dẫn đến các lỗi về tương tranh dữ liệu. Phần thứ hai của Rust là mã không an toàn. Đôi

khi việc sử dụng các cơ chế đảm bảo an toàn bộ nhớ của Rust là quá hạn chế đối với một số loại chương trình, do đó Rust cung cấp một giải pháp là mã không an toàn. Mã không an toàn là những đoạn mã mà trình biên dịch không kiểm tra tính an toàn về bộ nhớ, do đó lập trình viên phải tự chịu trách nhiệm kiểm tra độ an toàn của đoạn mã này. Khi không có sự kiểm tra từ trình biên dịch, mã Rust sẽ trở nên không an toàn như mã C/C++. Rust là một ngôn ngữ cấp độ hệ thống, và phần lớn các dự án hiện tại ở cấp hệ thống đều sử dụng C/C++, do đó yêu cầu Rust phải có khả năng tương tác với mã C/C++ đã tồn tại từ trước đó [40]. Các thao tác với ngôn ngữ ngoài Rust đều được coi là không an toàn, và C/C++ cũng không có cơ chế đảm bảo, do đó việc này càng trở nên nguy hiểm. Mặc dù số lượng mã không an toàn trung bình chỉ chiếm một phần rất nhỏ trong tổng khối lượng mã của cả dự án [51], nhưng mã không an toàn, hay thậm chí là mã an toàn, vẫn tiềm ẩn rất nhiều rủi ro về bảo mật.

Rust là ngôn ngữ mới nổi dạo gần đây nhưng cũng đã có một số lượng nghiên cứu về đảm bảo chất lượng mã nguồn cho Rust, một số cái tên nổi bật có thể kể đến như RustBelt [15], Miri [23], Rudra [1], Yuga [24]. RustBelt sử dụng kiểm chứng để chứng minh tính đúng đắn của đoạn mã nguồn Rust. Rustbelt chỉ ra một đoạn mã nguồn Rust nhất định cần đảm bảo điều kiện gì thì đoạn mã đó sẽ được coi là an toàn. Tuy nhiên, theo nghiên cứu của Yechan Bae và cộng sự [15], hướng tiếp cận của RustBelt không thể sử dụng ở phạm vi lớn, dành cho nhiều dự án mã nguồn khác nhau do hạn chế của việc sử dụng kiểm chứng. RustBelt có hiệu năng thấp và phụ thuộc vào chỉ dẫn thủ công của chuyên gia thì mới có thể sinh ra các điều kiện kiểm chứng chính xác.

Miri là một phương pháp sử dụng kiểm thử động thay vì kiểm chứng như RustBelt. Đầu tiên, Miri là một trình thông dịch dành cho ngôn ngữ Rust MIR (Rust Mid-Level Intermediate Representation), một ngôn ngữ trung gian được trình biên dịch sử dụng khi dịch mã nguồn Rust thành mã máy. Miri thực thi từng đoạn mã Rust ở dưới dạng MIR và sử dụng mô hình Stacked Borrow [14] để lý giải cho những hành vi borrowing mà không quy định lifetime tường minh. Như đã đề cập, Miri là một công cụ kiểm thử động và nó phát hiện ra lỗi với giá trị thực sự. Vì vậy, Miri chỉ tìm được lỗi khi chạy chương trình, không phải vào khoảng thời gian biên dịch khi mà đa số tính năng liên quan đến an toàn bộ nhớ của Rust được thực hiện. Vì Miri sử dụng kiểm thử động và kiểm thử mờ [17] nên cũng có các hạn chế

nhất định. Thứ nhất, đoạn mã được kiểm thử phải có yếu tố có thể khai thác thì mới có thể phát hiện được lỗi khi chạy chương trình. Để đạt hiệu quả thì phải có ca kiểm thử được viết thủ công hay đoạn mã được kiểm thử phải dễ xuất hiện lỗi, nhưng việc này rất khó đạt được trong dự án thực tế ở cấp độ hệ thống. Thứ hai, kiểm thử động và kiểm thử mờ sẽ tốn rất nhiều tài nguyên máy tính và thời gian, do vậy không thể áp dụng cho nhiều dự án mã nguồn lớn.

Rudra và Yuga là hai công cụ phát hiện lỗ hổng trong mã nguồn Rust dựa trên phân tích tĩnh. Rudra chuyên tìm kiếm các lỗi về bộ nhớ và an toàn luồng, Yuga tìm kiếm một loại lỗi hiếm gặp khi thực hiện sai tính năng Lifetime Annotation [24]. Hiện tại, đây là hai công cụ cho thấy kết quả tốt nhất so với các phương pháp kiểm chứng hay kiểm thử động đã đề cập ở trên. Rudra thực hiện kiểm tra trên 43 nghìn dự án mã nguồn Rust và phát hiện ra 264 lỗi chưa từng được phát hiện trước đó, trong khi Yuga thực hiện trên 21 đoạn mã có lỗi và phát hiện được 16 lỗi Lifetime Annotation. Rudra và Yuga sử dụng HIR (High-level Intermediate Representation) [29] và MIR (Mid-level Intermediate Representation) [37], thực hiện các thuật toán riêng biệt trên hai loại dữ liệu này để phát hiện lỗ hổng trong mã nguồn Rust. HIR là ngôn ngữ trung gian được sinh ra từ AST và vẫn giữ được cấu trúc của mã nguồn. MIR là ngôn ngữ trung gian bậc thấp hơn của HIR, tập trung vào các thông tin ngữ cảnh. Với mục tiêu là tìm ra các lỗi nhất định dựa trên thuật toán, Rudra và Yuga không được áp dụng cho các loại lỗi phổ biến. Điểm chung của các giải pháp RustBelt, Miri, Rudra và Yuga là đều khai thác các đầu vào riêng biệt của Rust, như HIR và MIR. Với đặc điểm trên, các nghiên cứu đã được phát triển từ trước cho các ngôn ngữ tương thích với Rust như C/C++ không thể tái sử dụng ngay lập tức. Cần những sự thay đổi đáng kể từ các nghiên cứu đã có để có thể áp dụng được cho Rust. Điều này cản trở tốc độ phát triển cũng như quá trình chuyển tiếp giữa Rust và C/C++ trong các dự án thực tế. Do vậy cần có một cấu trúc dữ liệu thống nhất cho việc phân tích mã nguồn Rust.

Ngoài ra, cũng đã có nghiên cứu thực hiện phân tích mã nguồn áp dụng cho Rust và sử dụng đồ thị CPG để biểu diễn. Đồ thị CPG là một dạng biểu diễn mã nguồn dưới dạng đồ thị thuộc tính chứa nhiều thông tin về phân tích mã nguồn [47]. Đồ thị CPG đã được sử dụng cho nhiều ngôn ngữ khác nhau như Java, C/C++, Python, JavaScript, v.v và đã chứng minh hiệu quả trong việc phân tích mã nguồn. Đồ thị CPG có thể coi là một cấu trúc dữ liệu, nền tảng chung cho nhiều ngôn

ngữ, có khả năng tái sử dụng và mở rộng cao. Hiện tại, chưa xuất hiện đồ thị CPG cho Rust ở mức độ mã nguồn mà mới chỉ có công cụ hỗ trợ gián tiếp ở mức độ trung gian, cụ thể là LLVM-IR [18]. Ưu điểm khi sử dụng LLVM-IR [19] là không chỉ áp dụng được cho Rust mà còn dùng cho các ngôn ngữ khác cũng sử dụng LLVM-IR làm ngôn ngữ trung gian như Clang C/C++, Swift, Zig, v.v. Tuy nhiên, chính việc sử dụng LLVM-IR lại mang lại những bất lợi khiến nó không phù hợp với Rust. LLVM-IR sử dụng hệ thống định kiểu đơn giản và không có kiểu tổng quát, trái ngược với Rust sử dụng hệ thống kiểu phức tạp và có kiểu tổng quát, do vậy LLVM-IR sẽ mất đi thông tin về kiểu tổng quát so với mã nguồn Rust gốc. Tiếp theo, LLVM-IR không giữ được thông tin về ownership, borrowing, lifetime đây là các tính năng mà trình biên dịch Rust xây dựng để đảm bảo an toàn về bộ nhớ và đa luồng. Đây là điểm đặc trưng của Rust, không có trong các ngôn ngữ khác, đặc biệt là ngôn ngữ trung gian bậc thấp như LLVM-IR. Do vậy LLVM-IR không phù hợp để phân tích mã nguồn cho Rust.

Để xử lý hạn chế của các giải pháp đi trước, khóa luận đưa ra một giải pháp cho việc phân tích mã nguồn Rust, đó là sử dụng đồ thị CPG để biểu diễn. Việc sử dụng đồ thị CPG cho ngôn ngữ Rust không chỉ giới hạn ở việc khai thác để phát hiện một số lớp lỗi nhất định mà có thể sử dụng cho nhiều tập lỗi khác nhau. Với nền tảng chung như vậy, các nghiên cứu đi trước cho việc khai thác đồ thị CPG có thể được áp dụng lại cho Rust mà không cần xử lý việc không tương thích cấu trúc dữ liệu riêng biệt của từng ngôn ngữ, ví dụ như HIR và MIR của Rust. Mã nguồn Rust có thể được chuyển thành LLVM-IR, và sử dụng nghiên cứu đồ thị CPG cho LLVM-IR đã có sẵn. Tuy nhiên, LLVM-IR không thể biểu diễn hết được các tính năng đặc trưng của Rust, vì vậy khóa luận sẽ phân tích mã nguồn Rust ngay tại cấp độ mã nguồn, không phải ở cấp độ trung gian như LLVM-IR.

Phần còn lại của khóa luận sẽ được trình bày với cấu trúc như sau. Chương 2 thảo luận một số kiến thức cơ sở liên quan đến chủ đề phân tích mã nguồn cho ngôn ngữ lập trình Rust. Chương 3 trình bày về quy trình hoạt động, kiến trúc và cài đặt của công cụ phân tích mã nguồn Rust. Ngoài ra, chương này cũng sẽ mô tả cách mã nguồn Rust được ánh xạ thành đồ thị CPG trên các cú pháp đặc trưng. Chương 4 bao gồm các thực nghiệm nhằm chứng tỏ được tiềm năng khai thác của đồ thị CPG trong việc đảm bảo chất lượng mã nguồn Rust. Cuối cùng sẽ là kết luận và kinh nghiệm rút ra sau quá trình phát triển công cụ.

Chương 2

Kiến thức cơ sở

Chương 2 sẽ trình bày các kiến thức cơ sở về ngôn ngữ lập trình Rust, bao gồm cơ chế quản lý bộ nhớ hiệu quả và tính hướng hàm đặc trưng. Chương này cũng sẽ trình bày về các dạng biểu diễn của mã nguồn bao gồm cây AST, đồ thị CFG, đồ thị PDG và đồ thị CPG. Joern, một bộ công cụ phân tích mã nguồn, và những ưu điểm mà nó mang lại sẽ được nhắc tới ở cuối chương.

2.1 Ngôn ngữ lập trình Rust

2.1.1 Cơ chế an toàn của Rust

Rust là một ngôn ngữ an toàn về bộ nhớ, được thiết kế cho phát triển phần mềm mức hệ thống. Trình biên dịch Rust kiểm tra chương trình với các luật để đảm bảo an toàn bộ nhớ và chống tương tranh dữ liệu. Các cơ chế an toàn bao gồm các khái niệm cơ bản: ownership, borrowing, lifetime và thread safe (an toàn luồng) 2.1.

Ownership. Cơ chế ownership giúp Rust có sự kiểm soát vừa đủ với việc quản lý bộ nhớ, không cần sử dụng cơ chế thu dọn bộ nhớ tự động như Java hoặc để người dùng tự xử lý như C/C++. Theo cơ chế ownership của Rust, một giá trị (có địa chỉ nhất định trong bộ nhớ) trong một lúc chỉ có một biến sở hữu độc quyền. 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 một biến khác dẫn đến chuyển ownership. Khi một biến mất quyền sở hữu một giá trị, biến đó sẽ không còn sử dụng được nữa. Trình biên dịch Rust theo dõi sự tồn tại 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 thiết kế RAII (Resource Acquisition Is Initialization) [3] thường được sử dụng trong ngôn ngữ C++.

Borrowing. Để có thể chia sẻ giá trị mà không chuyển ownership, cơ chế borrowing cho phép việc tạo một tham chiếu đến giá trị và biến mới sẽ sử dụng tham chiếu đó. Với cơ chế borrowing, một giá trị có thể được đọc hoặc cập nhật mà không thay đổi ownership của giá trị. Có hai kiểu borrowing là shared read-borrowing

(mượn chia sẻ để đọc) và exclusive mutable-borrowing (mượn độc quyền để ghi). Trình biên dịch Rust đảm bảo rằng 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, từ đó loại bỏ khả năng xảy ra tương tranh dữ liệu, bế tắc hay các lỗi lập trình khác.

Lifetime. Lifetime đại diện cho phạm vi mà tham chiếu có hiệu lực, hay việc sử dụng tham chiếu đó để lấy giá trị là an toàn về bộ nhớ. Tính năng lifetime trong Rust là một loại kiểu tổng quát, các lifetime tổng quát thể hiện mối quan hệ ràng buộc giữa thời gian hợp lệ của tham chiếu và thời gian sống của giá trị mà chúng tham chiếu tới. Cụ thể, để xác định khi nào các tham chiếu ra khỏi phạm vi, trình biên dịch liên kết mỗi tham chiếu với một lifetime và theo dõi các ràng buộc giữa các lifetime. Hệ thống lifetime kết hợp với borrowing của Rust đảm bảo rằng các vấn đề an toàn bộ nhớ truyền thống như sử dụng sau khi giải phóng, con trỏ chỉ tới địa chỉ null, con trỏ treo là không thể xảy ra bằng cách không cho phép các tham chiếu tồn tại lâu hơn biến thực sự sở hữu giá trị.

Thread safe. Lập trình đa luồng trong Rust được đảm bảo an toàn nhờ mô hình ownership và borrowing, từ đó ngăn chặn tương tranh dữ liệu, bế tắc ngay từ khi biên dịch. Rust sử dụng thuộc tính Send (Gửi) và Sync (Đồng bộ) để đảm bảo an toàn đa luồng ở cấp độ kiểu dữ liệu. Một kiểu có thuộc tính Send nếu nó có thể được chuyển quyền sở hữu một cách an toàn từ luồng này sang luồng khác, và có thuộc tính Sync nếu nó có thể được truy cập 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. Để quản lý trạng thái có thể thay đổi được trong môi trường đa luồng, Rust cung cấp các cơ chế đồng bộ hóa như Mutex (Mutual exclusion) (Ghi độc quyền) và Arc (Atomic reference count) (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 tương tranh dữ liệu. Arc được sử dụng để đếm số lượng tham chiếu an toàn giữa các luồng, cho phép nhiều luồng chia sẻ quyền sở hữu dữ liệu, hoạt động tương tự cơ chế dọn rác bộ nhớ trong Java. Một nguyên tắc quan trọng trong lập trình đa luồng Rust là chuyển quyền sở hữu dữ liệu từ luồng cũ vào luồng mới. 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 tương tranh dữ liệu.

```rust
fn ownership() {
  let s1 = String::from("hello"); // s1 sở hữu giá trị "hello"
  let s2 = s1; // Chuyển quyền sở hữu từ s1 sang s2
  // println!("{}", s1); // s1 không còn sử dụng được vì mất quyền sở hữu
  println!("{}", s2); // s2 có thể sử dụng bình thường
}

fn borrowing() {
    let mut vec = vec![1, 2, 3];
    // Mượn độc quyền để sửa
    let mut_ref = &mut vec;
    mut_ref.push(4);
    // Mượn chia sẻ để đọc
    let shared_ref1 = &vec;
    let shared_ref2 = &vec;
    println!("{}", shared_ref1[0]);
    println!("{}", shared_ref2[0]);
}

fn lifetime() {
    let r;
    {
        let x = 5;
        r = &x; // r mượn x nhưng x tồn tại chỉ trong phạm vi này
    }
    // println!("r: {}", r); // Lỗi: r trỏ đến x đã bị giải phóng
}

// Data chứa kiểu dữ liệu i32 và i32 tự động là Send và Sync
struct Data(i32)

fn thread_safe() {
    let data = Data(42);
    let handles: Vec<_> = (0..10).map(|_| {
        let data = data; // Di chuyển quyền sở hữu data vào luồng mới
        thread::spawn(move || {
            println!("Thread data: {:?}", data);
        })
    }).collect();
}
```

Đoạn mã 2.1: Ví dụ các khái niệm an toàn trong Rust: (1) ownership, (2) borrowing, (3) lifetime và (4) thread safe.

2.1.2 Tính hướng hàm

Rust được lấy cảm hứng từ nhiều ngôn ngữ khác nhau, trong đó có các ngôn ngữ lập trình hướng hàm [9]. Rust là sự kết hợp giữa lập trình hướng đối tượng và lập trình hướng hàm. Do có sự pha trộn của tính hướng hàm nên một số tính năng, cú pháp của Rust sẽ khác biệt so với ngôn ngữ tiêu biểu như C/C++.

Expression over statement (Sử dụng biểu thức hơn mệnh đề). Trong Rust, hầu hết mọi thứ đều là biểu thức và có thể trả về một giá trị. Điều này tạo ra sự khác biệt so với các ngôn ngữ cổ điển như C/C++. Đoạn mã 2.2 mô tả cú pháp điều kiện if else trong Rust là một biểu thức. Cú pháp if else có thể trả về giá trị và gán giá trị đó cho một biến khác hoặc truyền vào hàm khác.

```
fn main() {
  let x = 5;

  let value = if x < 0 {
      -1
  } else {
      1
  }
}
```

Đoạn mã 2.2: Ví dụ về biểu thức if else trong Rust.

Pattern matching (Khớp mẫu). Pattern matching làm cho mã nguồn ngắn gọn và rõ ràng hơn. Pattern matching trong Rust tương tự như switch case trong C/C++ nhưng mạnh mẽ hơn, có thể phân tách cấu trúc dữ liệu phức tạp như `tuple`, `enum`, `struct`. Đoạn mã 2.3 minh họa cơ chế pattern matching trong Rust.

```
fn match_example(value: Option<i32>) {
  match value {
      Some(x) => println!("Received a value: {}", x),
      None => println!("Received None"),
  }
}
```

Đoạn mã 2.3: Ví dụ về cú pháp pattern matching trong Rust.

Monad design pattern (Mẫu thiết kế Monad). Monad là một mẫu thiết kế cho phép nối chuỗi các tính toán một cách tuần tự, đồng thời cung cấp ngữ cảnh cho các tính toán đó [7]. Monad sẽ ở dạng một lớp cha bọc lấy dữ liệu con bên trong, thường được ví như "hộp" chứa giá trị, và hộp này có những quy tắc đặc biệt về cách lấy giá trị bên trong. Đoạn mã 2.4 lấy ví dụ về hai loại dữ liệu `Option` và `Result` trong Rust, một dạng Maybe Monad [21], dùng để xử lý trường hợp không có giá trị và ngoại lệ một cách an toàn.

```rust
enum Option<T> {
    Some(T),
    None,
}

fn safe_divide(divided: i32, divisor: i32) -> Option<i32> {
  if divisor != 0 {
      Some(divided / divisor)
  } else {
      None
  }
}

fn main() {
  let result = safe_divide(10, 2);

  match result {
      Some(value) => println!("Result: {}", value),
      None => println!("Cannot divide by zero"),
  }
}
```

Đoạn mã 2.4: Ví dụ về mẫu thiết kế Monad trong Rust.

Algebraic Data Types (Kiểu dữ liệu đại số). Hệ thống kiểu của Rust theo xu hướng kiểu dữ liệu đại số có trong lập trình hướng hàm. Nhờ kiểu dữ liệu đại số, Rust cung cấp cách thức linh hoạt để mô hình hóa các vấn đề trong lập trình. Kiểu dữ liệu đại số trong Rust được thể hiện thông qua cú pháp khai báo kiểu `enum`. Đoạn mã 2.5 minh họa cách kiểu `enum` được sử dụng cùng với pattern matching để biểu diễn và xử lý các trường hợp khác nhau của dữ liệu một cách rõ ràng.

```rust
enum GameResult {
    Win { score: u32 },
    Lose { reason: String },
}

fn main() {
    let result = GameResult::Win { score: 100 };

    match result {
        GameResult::Win { score } => {
            println!("You win with a score of {}!", score);
        }
        GameResult::Lose { reason } => {
            println!("You lose because: {}", reason);
        }
    }
}
```

Đoạn mã 2.5: Ví dụ về kiểu dữ liệu đại số trong Rust.

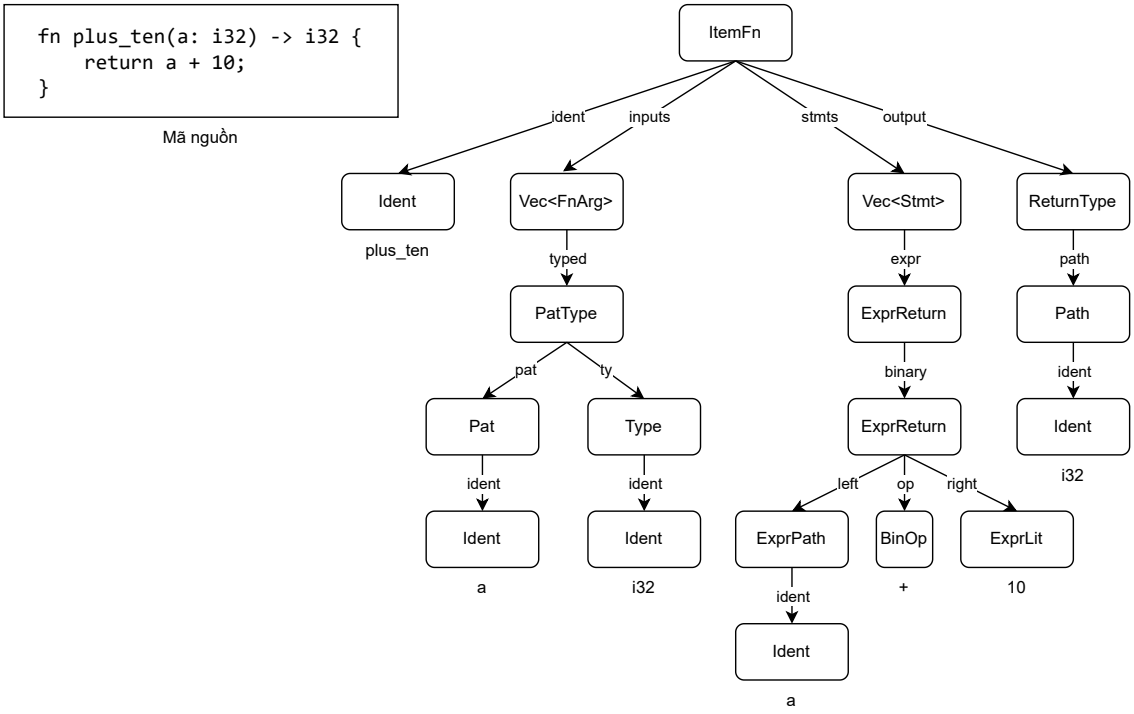
2.2 Các cách biểu diễn mã nguồn

Có nhiều cách biểu diễn mã nguồn khác nhau đã được phát triển và sử dụng rộng rãi trong lĩnh vực phân tích chương trình và thiết kế trình biên dịch. Mục đích chung của các cách biểu diễn là giải thích các thuộc tính của chương trình. Các kiểu biểu diễn chủ yếu ở dưới dạng cấu trúc dữ liệu cây và đồ thị. Khóa luận sẽ chỉ tập trung vào đồ thị CPG, trong đó đồ thị CPG là dạng đồ thị được hợp thành từ cây AST, đồ thị CFG và đồ thị PDG.

2.2.1 Cây cú pháp trừu tượng

Cây cú pháp trừu tượng (Abstract Syntax Tree) [50] là dạng biểu diễn đầu tiên của mã nguồn, và là cơ sở cho các dạng biểu diễn tiếp theo. AST không chứa tất cả cú pháp chi tiết nhưng vẫn thể hiện được quan hệ của các biểu thức, mệnh đề trong mã nguồn. AST 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. AST là một cây cấu trúc phân cấp, nút trong gọi là toán tử bao gồm biểu thức và mệnh

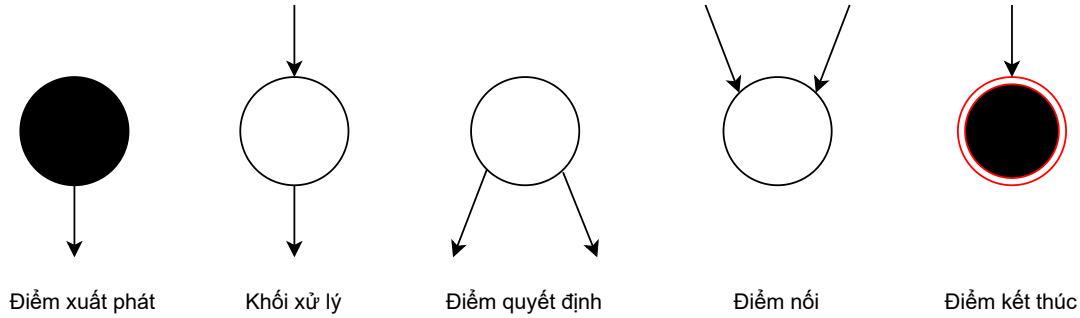
đề, các nút lá gọi là toán hạng bao gồm các biến và ký tự. AST được áp dụng cho phân tích cấu trúc, biến đổi cấu trúc hoặc phát hiện các đoạn mã cấu trúc giống nhau [48]. Tuy nhiên, AST không sử dụng được cho các phân tích chuyên sâu hơn bởi vì AST không chứa thông tin về luồng điều khiển hoặc phụ thuộc dữ liệu của chương trình. Hình 2.1 biểu diễn một cây AST tương ứng với một đoạn mã nguồn trong Rust.



Hình 2.1: Ví dụ về cây AST cho mã nguồn Rust.

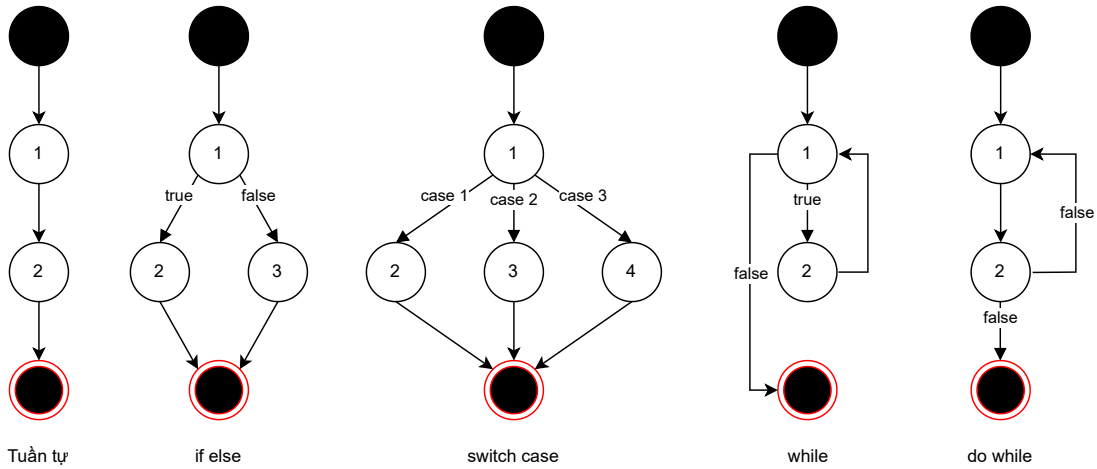
2.2.2 Đồ thị luồng điều khiển

Đồ thị luồng điều khiển (Control Flow Graph) [49] là đồ thị có hướng, mô tả thứ tự thực thi của mệnh đề và điều kiện để một mệnh đề được thực thi. Các nút là các mệnh đề hoặc mệnh đề điều kiện, nối với nhau bằng các cạnh có hướng, thể hiện thứ tự điều khiển giữa các nút. Một nút là mệnh đề thì có một cạnh ra. Nếu một nút là mệnh đề điều kiện thì sẽ có hai cạnh ra, bao gồm một cạnh điều khiển khi điều kiện đúng và một cạnh điều khiển khi điều kiện sai. CFG được sử dụng cho nhiều ứng dụng phân tích ngữ cảnh, phân tích mã độc hại, định hướng cho công cụ kiểm thử mờ [6]. Tuy nhiên, CFG không chứa thông tin về luồng dữ liệu, do vậy không đủ toàn diện để ứng dụng phát hiện lỗ hổng bảo mật trong mã nguồn.



Hình 2.2: Các thành phần cơ bản trong đồ thị CFG.

Đồ thị luồng điều khiển bao gồm các thành phần chính là điểm xuất phát, khối xử lý, điểm quyết định, điểm nối và điểm kết thúc. 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 có 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 hai cạnh đi ra. **Điểm nối** biểu thị các câu lệnh thực hiện ngay sau các lệnh rẽ nhánh, có hai cạnh nối đến, được thể hiện bằng hình tròn rỗng.

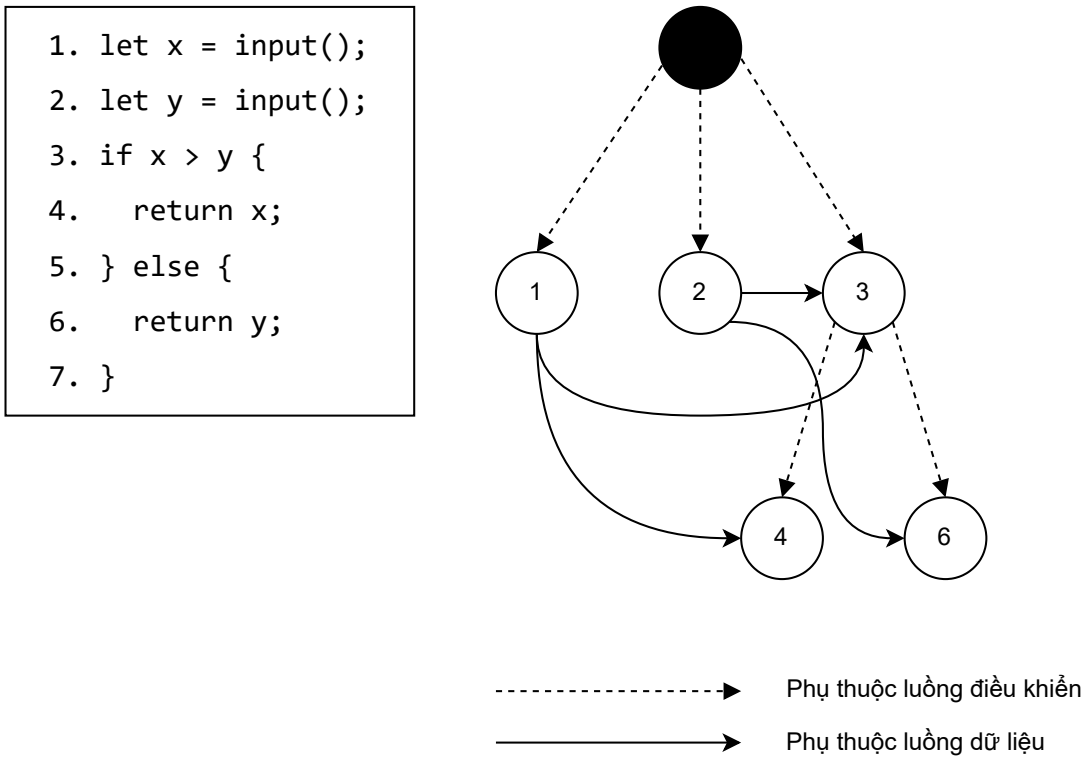


Hình 2.3: Các cấu trúc điều khiển phổ biến trong ngôn ngữ lập trình.

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) [5] là đồ thị có hướng thể hiện hai khía cạnh của chương trình, phụ thuộc điều khiển và phụ thuộc dữ liệu. Một nút đại diện cho các mệnh đề hoặc mệnh đề điều kiện, một cạnh thể hiện mối quan hệ phụ thuộc điều khiển hoặc phụ thuộc dữ liệu giữa các nút. Mệnh đề mà một nút đại diện có được thực thi hay không phụ thuộc vào các cạnh điều kiện điều khiển trỏ tới nút, giá trị của các biến mà mệnh đề sử dụng phụ thuộc vào các cạnh phụ thuộc dữ liệu trỏ tới nút đó. Lưu ý rằng cạnh phụ thuộc điều khiển không giống như cạnh luồng điều khiển của đồ thị CFG. Cạnh phụ thuộc điều khiển chỉ thể hiện điều kiện để mệnh đề của một nút được thực thi, không thể hiện thứ tự thực thi của mệnh đề giữa các nút.

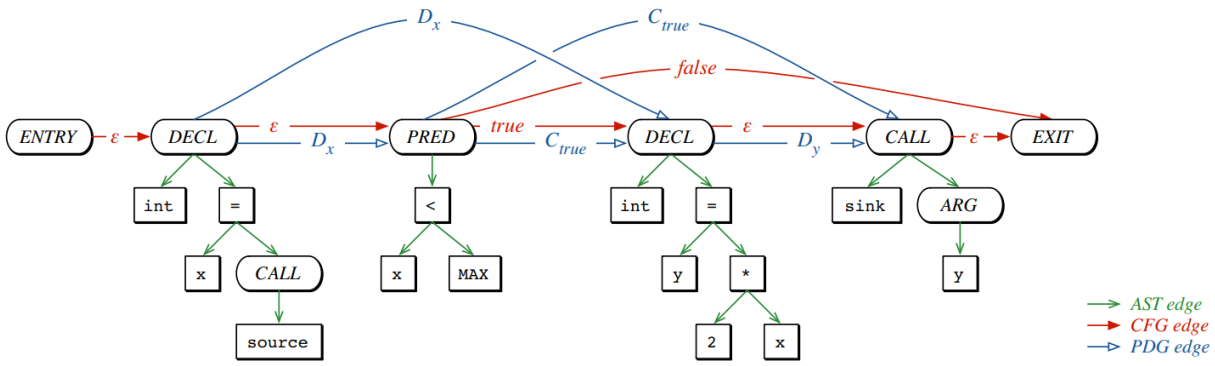


Hình 2.4: Ví dụ về đồ thị PDG.

Hình 2.4 biểu diễn ví dụ về một đồ thị PDG 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) [47] là một dạng đồ thị biểu diễn mã nguồn hợp thành từ cây AST, đồ thị CFG và đồ thị PDG. Đồ thị chứa các thông tin về cấu trúc cú pháp, luồng điều khiển và phụ thuộc dữ liệu trong chương trình Đồ thị CPG tạo ra một lớp biểu diễn trung gian cho mã nguồn mà không bị phụ thuộc vào ngôn ngữ lập trình cụ thể. Các nút đại diện cho các thành phần như hàm, biến, lớp 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. Mỗi nút, cạnh đều có các thuộc tính, mỗi thuộc tính có giá trị riêng. Đồ thị CPG được ứng dụng để tìm kiếm lỗ hổng trong mã nguồn bằng học máy, học tăng cường [52]. Hình 2.5 minh họa đồ thị CPG cho một đoạn mã nguồn C [47].



Hình 2.5: Minh họa đồ thị CPG của một đoạn mã nguồn C.

2.3 Công cụ Joern

2.3.1 Đặc tả đồ thị thuộc tính mã nguồn của Joern

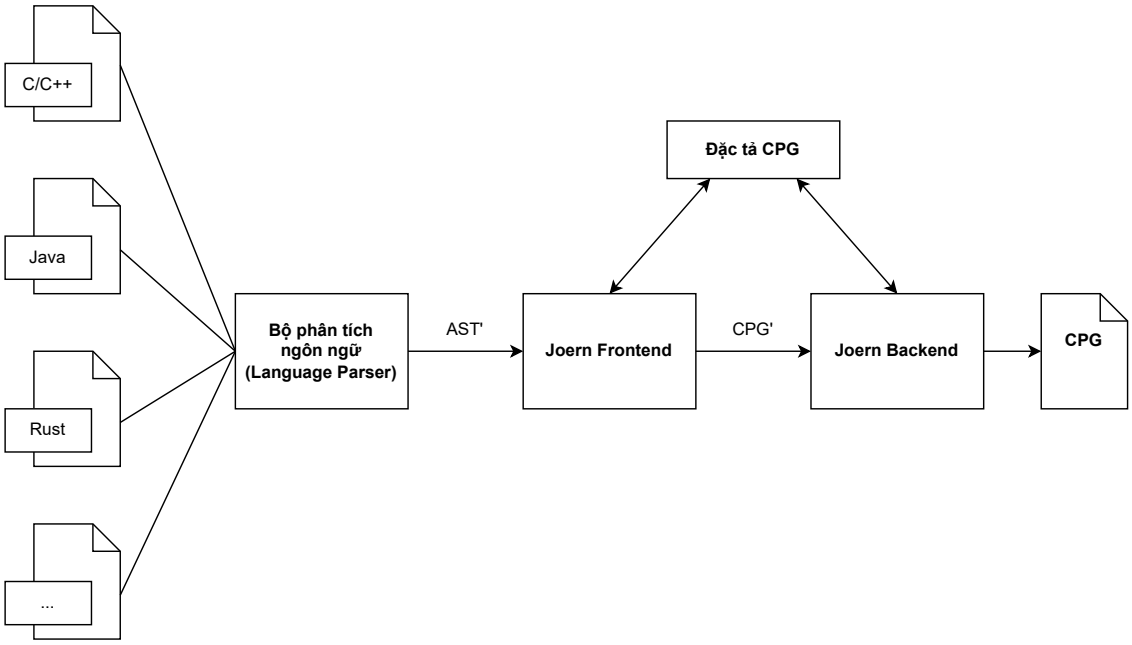
Đồ thị CPG đã được nghiên cứu rộng rãi và có rất nhiều phiên bản cài đặt được xây dựng dành cho các mục đích khác nhau [18, 45]. Tuy nhiên, có một phiên bản được do chính tác giả của khái niệm đồ thị CPG, Fabian Yamaguchi, đích thân phát triển và mã nguồn mở mang tên Joern [11]. Các thành phần trong đồ thị CPG theo đặc tả CPG của Joern bao gồm nút, cạnh, và thuộc tính. Các nút đại diện cho các thành phần cấu trúc của chương trình như phương thức, biến, và cấu trúc điều khiển. Mỗi nút có một loại riêng, 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 thể hiện quan hệ giữa các thành phần chương trình, cạnh này có hướng và có nhãn. 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 phương thức đến nút biến cục bộ. Bằng cách sử dụng các cạnh có nhãn, có thể biểu diễn nhiều loại quan hệ khác nhau trong cùng một đồ thị. Hơn nữa, các cạnh có hướng giúp biểu thị mối quan hệ có chiều, ví dụ như phương thức chứa biến cục bộ nhưng không xảy ra điều ngược lại. Giữa hai nút có thể tồn tại nhiều cạnh. Các nút, cạnh có các thuộc tính, tồn tại dưới dạng khóa và giá trị, trong đó các khóa phụ thuộc vào loại nút, nhãn cạnh riêng biệt. Đồ thị CPG được lưu trữ trong một cơ sở dữ liệu đồ thị và khai thác thông qua một ngôn ngữ truy vấn đặc thù dựa trên Scala.

Một phiên bản cài đặt cụ thể của đồ thị CPG sẽ có một bản đặc tả, gọi là đặc tả CPG. Bản đặc tả này định nghĩa các loại nút, loại cạnh, thuộc tính và quan hệ giữa chúng, ngoài ra có thể bao gồm các mở rộng khác. Định nghĩa về cạnh, nút có thể được chuyên biệt cho một ngôn ngữ cụ thể hoặc tổng quát cho nhiều ngôn ngữ. Đặc tả CPG của Joern được thiết kế chủ yếu cho C/C++ và Java, và cũng có hỗ trợ cú pháp của ngôn ngữ lập trình khác như Python, Go, TypeScript, v.v. Tuy nhiên, đặc tả CPG hiện thời của Joern chưa hỗ trợ cho ngôn ngữ Rust.

Đặc tả CPG của Joern được thiết kết chủ yếu cho ngôn ngữ C/C++ và Java, đưa ra định nghĩa cho các cấu trúc chung như if else, while, for, v.v mà nhiều ngôn ngữ khác có cấu trúc tương đồng. Dù vậy, chuẩn chung này không thể đáp ứng được đặc thù của tất cả các ngôn ngữ. Joern tập trung cho hai ngôn ngữ lớn là C/C++ và Java do vậy đặc tả hiện tại sẽ phù hợp với các ngôn ngữ có cú pháp C-like và lập trình hướng đối tượng. Trong khi đó Rust là một ngôn ngữ lập trình mới, có cú pháp dựa trên C-Like nhưng vẫn có sự khác biệt vì đây là ngôn ngữ hiện đại, hỗ trợ đan xen cả hướng đối tượng và hướng hàm. Đặc biệt với tính hướng hàm, Rust có nhiều cú pháp mới, hay những biểu thức, mệnh đề trong C/C++ hay Java là không hợp lệ nhưng ngược lại đối với Rust. Nhìn chung bản đặc tả CPG của Joern cung cấp một cơ sở bao phủ các tính năng phổ biến xuất hiện trong nhiều ngôn ngữ nhưng không thể đáp ứng được tất cả. Do vậy đối với từng ngôn ngữ riêng biệt vẫn phải bổ sung thêm các định nghĩa mới sao cho phù hợp với từng ngôn ngữ. Đặc biệt đối với Rust, cơ chế quản lý bộ nhớ an toàn thể hiện qua các tính năng ownership, borrowing, lifetime là thứ đặc tả CPG của Joern chưa có.

2.3.2 Luồng hoạt động của công cụ Joern



Hình 2.6: Luồng hoạt động của công cụ Joern.

Ngoài phần đặc tả CPG dùng chung cho nhiều ngôn ngữ, Joern còn cung cấp một kiến trúc cài đặt có tính mở rộng cao. Joern hiện tại hỗ trợ rất nhiều ngôn ngữ C/C++, Java, Python, Go, TypeScript, v.v. Để đáp ứng được nhiều ngôn ngữ và đồng thời mở rộng cho các ngôn ngữ khác trong tương lai, kiến trúc của Joern bao gồm hai thành phần chính là Joern Frontend và Joern Backend. Đây là hai thành phần có tính tái sử dụng cao, không phụ thuộc vào ngôn ngữ đầu vào.

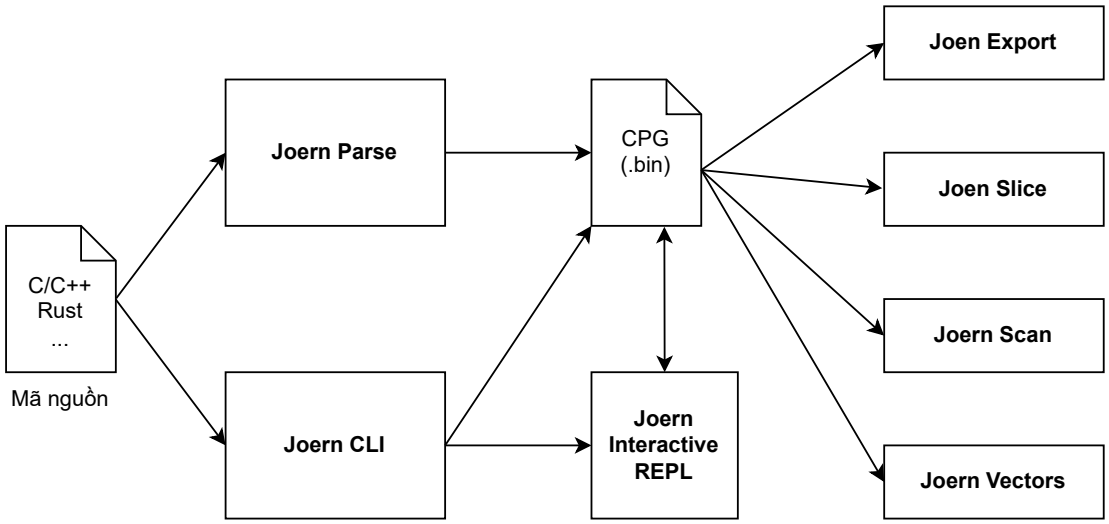
Hình 2.6 mô tả luồng hoạt động dựa trên kiến trúc Joern. Mỗi ngôn ngữ có tập cú pháp khác nhau, tương ứng cũng sẽ có định nghĩa về cây AST khác nhau. Đây là phần người dùng muốn mở rộng một ngôn ngữ mới cho Joern cần phải tự đảm nhiệm. Bước đầu, mã nguồn của một ngôn ngữ được bộ phân tích ngôn ngữ chuyển thành cây AST'. Cây AST' ở đây không nhất thiết chỉ dừng ở đúng mức độ thông tin của một cây AST thông thường mà có thể bổ sung thêm thông tin khác như kiểu dữ liệu, vị trí nút tương ứng trong mã nguồn, v.v nhưng đảm bảo tối thiểu đủ thông tin của một cây AST. Ví dụ như các ngôn ngữ có sự phát triển lâu dài như C/C++ sử CDT [44] để làm bộ phân tích ngôn ngữ. CDT là công cụ lớn, được dùng trong các IDE nên số lượng thông tin khai thác được rất đáng kể. Còn những

ngôn ngữ hiện đại như Rust, thì các công cụ như này chưa được phát triển toàn diện nên thông tin của cây AST' chưa được chi tiết. Tiếp theo, cây AST' sẽ được biến đổi thành đồ thị CPG theo định nghĩa trong đặc tả CPG. Joern Frontend sẽ đọc cây AST' và chuyển đổi thành đồ thị CPG theo đặc tả CPG. Joern Frontend đối với từng ngôn ngữ cũng là khác nhau, công việc cần làm là chuyển đổi cấu trúc dữ liệu của cây AST sang cấu trúc dữ liệu tương ứng theo đặc tả CPG. Joern Frontend đóng vai trò chuyển đổi cây AST' sang đồ thị CPG'.

Sau khi được Joern Frontend xử lý, ta sẽ có được đồ thị CPG' nhưng đồ thị CPG' này là đồ thị CPG không hoàn chỉnh. Để hoàn thiện được đồ thị CPG' thì ta cần phải sử dụng đến Joern Backend. Ở Joern Frontend, ta đã thực hiện bước chuyển đổi một nút AST thành một nút CPG tương ứng, tạo thêm các cạnh để thể hiện các mối quan hệ giữa các nút. Tuy nhiên, các nút và các cạnh này chưa thể hiện được đầy đủ đồ thị CPG bao gồm cây AST, đồ thị CFG, đồ thị PDG. Với cấu trúc dữ liệu CPG của Joern, ta chỉ cần định nghĩa một phần số cạnh, nút cần thiết của đồ thị CPG, phần còn lại sẽ được Joern Backend thực hiện các thuật toán để có thể tự động suy diễn các nút, cạnh còn lại. Ví dụ trong cây AST có nút thể hiện vòng lặp `for`, ta chuyển thành nút `CONTROL STRUCTURE` của CPG kèm thêm một số thông tin như mệnh đề điều kiện, biến chỉ số vòng lặp. Joern Backend sử dụng thông tin đã có thể tự động suy diễn ra các mối quan hệ như `REACHING DEFINITION`, `DOMINATOR`, `POST DOMINATOR`, `CONTROL DEPENDENCY`, `DATA DEPENDENCY`, v.v từ đó tạo ra đồ thị CPG hoàn chỉnh.

Trong Joern, thông tin của ba thành phần cây AST, đồ thị CFG, đồ thị PDG được ánh xạ thành các lớp trong kiến trúc, tương ứng ba lớp. Một đồ thị CPG được cấu tạo từ nhiều lớp chồng lên nhau, có thể sinh ra đồ thị CPG chỉ có lớp AST, hoặc AST và CFG, hoặc AST, CFG và PDG. Hoàn toàn có thể mở rộng, viết thêm các lớp thông tin khác nếu cần thiết. Ví dụ như bổ sung một lớp thông tin về an toàn bộ nhớ, cụ thể trong Rust là lớp chứa thông tin về ownership, borrowing và lifetime. Sau khi toàn bộ các lớp thông tin của CPG được hoàn chỉnh, dữ liệu được xuất ra thành tệp nhị phân dưới dạng cấu trúc dữ liệu đồ thị. Dữ liệu này có thể được sử dụng để truy vấn thông qua các cơ sở dữ liệu đồ thị như Neo4j [22] hoặc có thể được sử dụng để phân tích, tìm kiếm thông qua các công cụ khác của Joern.

2.3.3 Bộ công cụ của Joern



Hình 2.7: Các công cụ xung quanh Joern.

Không chỉ cung cấp kiến trúc có tính mở rộng và tái sử dụng cao, Joern còn cung cấp một loạt các công cụ để khai thác thông tin từ đồ thị CPG được sinh ra 2.7. Các công cụ này bao gồm:

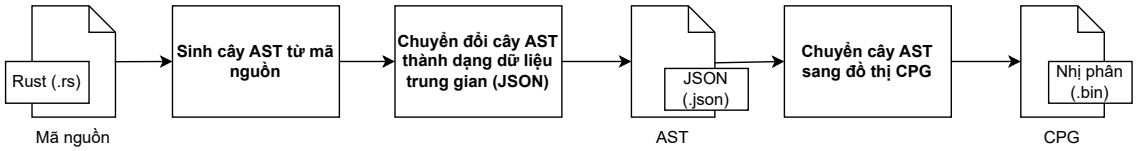
- **Joern Export.** Dùng để chuyển đổi đồ thị CPG từ dạng nhị phân sang dạng dữ liệu tương thích với các cơ sở dữ liệu đồ thị bao gồm neo4jcsv, graphml, graphson, graphviz dot. Từ các định dạng tương thích với cơ sở dữ liệu đồ thị, các ứng dụng về học máy, học tăng cường để phát hiện lỗ hổng trên đồ thị CPG có thể được áp dụng.
- **Joern Slice.** Dùng để chia nhỏ đồ thị CPG của các mã nguồn lớn thành từng phần nhỏ, từ đó dễ dàng khai thác tập trung vào một khía cạnh cụ thể của mã, chẳng hạn như luồng dữ liệu xung quanh một biến cụ thể.
- **Joern Scan.** Thực hiện quét toàn bộ đồ thị CPG để tìm ra lỗ hổng bảo mật. Joern QueryDB [12] cung cấp cho Joern Scan một tập các truy vấn được định nghĩa sẵn để tìm kiếm các mẫu cụ thể trong mã. Các truy vấn trong QueryDB có thể áp dụng riêng cho một hoặc sử dụng lại chung cho nhiều ngôn ngữ.
- **Joern Vectors.** Trích xuất các biểu diễn vectơ từ đồ thị CPG.

Chương 3

Thiết kế và cài đặt công cụ

Chương 3 trình bày về quy trình phân tích mã nguồn cho ngôn ngữ Rust bao gồm việc xây dựng cây AST và ánh xạ từ cây AST sang đồ thị CPG. Chương cũng sẽ đi sâu vào kiến trúc, các thành phần và cài đặt của công cụ. Ngoài ra, các thể loại cú pháp Rust sẽ được phân tích để minh họa cách ánh xạ từ cây AST sang đồ thị CPG một cách hiệu quả.

3.1 Luồng hoạt động và cài đặt công cụ



Hình 3.1: Quy trình phân tích mã nguồn Rust.

Mục tiêu của công cụ là phân tích mã nguồn Rust và xây dựng đồ thị CPG biểu diễn mã nguồn đó. Đầu vào của công cụ là các tệp mã nguồn Rust và đầu ra là đồ thị CPG được lưu dưới dạng nhị phân để dễ dàng xử lý và lưu trữ. Đồ thị CPG được phục vụ cho việc thực hiện các câu lệnh truy vấn trên đồ thị hoặc quét đồ thị để tìm lỗ hổng trong mã nguồn. Với yêu cầu đầu vào và đầu ra như trên, luồng hoạt động của công cụ được thiết kế thành các bước liên kết chặt chẽ và được thể hiện trong Hình 3.1. 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 `.rs`).

2. Với mỗi tệp mã nguồn, sử dụng thư viện *syn* [42] để sinh cây AST từ nội dung mã nguồn của tệp đó. *syn* là một thư viện phân tích mã nguồn thành cây AST được sử dụng rộng rãi với nhiều mục đích, trong đó bao gồm việc cài đặt tính năng Procedural Macro [26] của Rust. Tính tới thời điểm hiện tại Rust không có đặc tả ngôn ngữ chính thức, do đó cộng đồng sử dụng Rust Reference [38]

coi như phiên bản sát nhất so với một đặc tả ngôn ngữ. Thư viện syn xây dựng định nghĩa các nút của cây AST tuân theo Rust Reference.

3. Thực hiện chuyển đổi cây AST từ ngôn ngữ Rust sang định dạng JSON. Joern có thể được sử dụng cho nhiều ngôn ngữ và Joern Frontend không phụ thuộc vào bộ phân tích ngôn ngữ của một ngôn ngữ nhất định, do vậy cần có định dạng dữ liệu trung gian để chuyển đổi cây AST của bộ phân tích ngôn ngữ sang ngôn ngữ Scala của Joern Frontend. Có 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.

4. Cây AST dưới định dạng JSON được đọc ngược lại bằng mã nguồn Scala của Joern Frontend. Từ đây ta sẽ thực hiện chuyển đổi cây AST sang đồ thị CPG, từng loại nút trong cây AST sẽ có ánh xạ tương ứng với một loại nút trong đồ thị CPG. Các thông tin trong cây AST sẽ được khai thác để xây dựng nút CPG phù hợp, thông tin giữa các nút AST được sử dụng để xây dựng các cạnh, thuộc tính cho cạnh và nút trong đồ thị CPG. Quá trình xây dựng đồ thị CPG sẽ bao gồm vai trò của Joern Frontend và Joern Backend, quá trình này sẽ được mô tả chi tiết ở phần 3.2.

5. Cuối cùng, đồ thị CPG được lưu dưới dạng tệp nhị phân và đây là đầu ra kì vọng của công cụ.

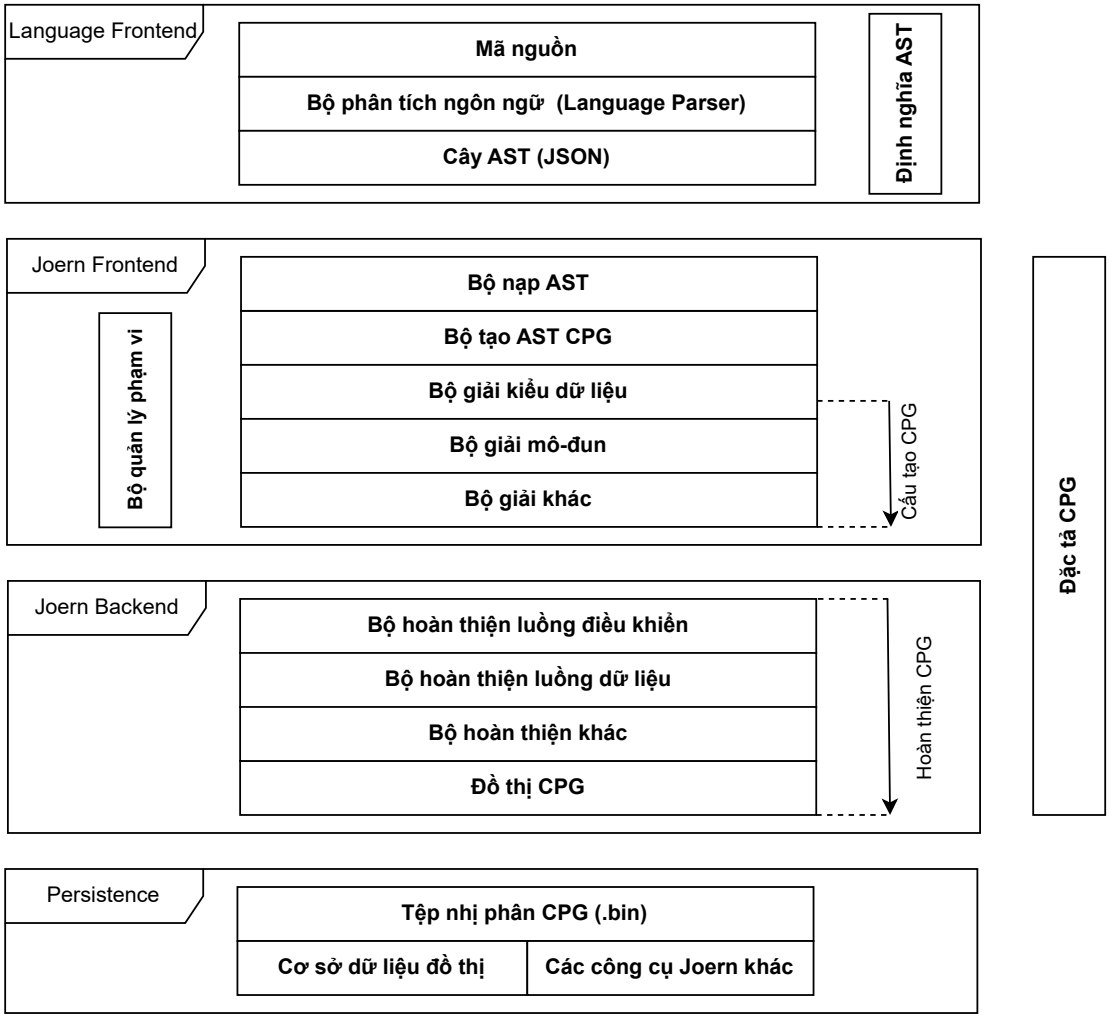
Về cài đặt, thư viện syn phiên bản v2.0.87 có định nghĩa của 162 `struct` tương ứng với 162 loại nút AST và 33 `enum` tương ứng với 33 loại nút AST đa hình. Hiện tại, công cụ đã ánh xạ 162 loại nút AST và 33 loại nút AST đa hình ở trên thành loại nút CPG tương ứng, tức là **100% định nghĩa các loại nút AST đã được ánh xạ sang nút CPG tại phiên bản hiện thời**. Bảng tổng hợp ánh xạ các loại nút AST sang loại nút CPG tương ứng được lưu tại địa chỉ rust-parser/mapping.

Công cụ được thực hiện kiểm thử trên 151 tệp mã nguồn bao gồm đa dạng các thể loại cú pháp, thu thập từ trang Rust By Example [28]. Ngoài ra, việc chuyển đổi từ AST sang CPG được kiểm tra trên 20 dự án lớn nằm trong 100 dự án Rust có lượng sao lớn nhất trên Github [8]. Mã nguồn của công cụ được lưu trữ tại địa chỉ rust-parser, syn-serde, joern, codepropertygraph.

3.2 Kiến trúc công cụ

Kiến trúc của công cụ về cơ bản sẽ là kiến trúc của Joern và mở rộng thêm. Phần này sẽ đi chi tiết hơn so với phần 2.3.2 đã trình bày ở phía trên. Hình 3.2 mô tả các thành phần kiến trúc của công cụ, bao gồm:

Language Frontend. Đây là phần riêng của mỗi ngôn ngữ, người dùng cần tự đảm nhiệm phần này khi muốn mở rộng ngôn ngữ mới cho công cụ Joern. Nhiệm vụ của Language Frontend là chuyển đổi mã nguồn thành cây AST dựa trên đặc tả ngôn ngữ và Language Frontend có thể được cài đặt bằng bất cứ ngôn ngữ nào. Do đó để có thể chuyển tiếp dữ liệu cho quá trình tiếp theo, Language Frontend cần xuất cây AST ra một loại dữ liệu trung gian, ở đây là JSON.



Hình 3.2: Kiến trúc công cụ.

Joern Frontend. Là khối chính thực hiện công việc chuyển đổi cây AST của một ngôn ngữ lập trình thành đồ thị CPG theo đặc tả CPG [13]. Đặc tả CPG sẽ được sử dụng xuyên suốt bởi Joern Frontend và Joern Backend. Joern Frontend sẽ nhận dữ liệu từ Language Frontend dưới dạng JSON bằng bộ nạp AST. Sau khi được nạp vào Joern Frontend dưới ngôn ngữ Scala, cây AST được chuyển đổi thành cây CPG bằng bộ tạo AST CPG. Bộ tạo AST CPG thực hiện ánh xạ mỗi một loại nút AST của ngôn ngữ sang một loại nút CPG tương ứng. Bộ tạo AST CPG còn tạo các cạnh giữa các nút, giữa các nút có thể có nhiều cạnh, mỗi cạnh thể hiện một mối quan hệ giữa các nút. Mặc định giữa các nút có mối quan hệ là cạnh `AST`. Tùy vào thể loại nút thì sẽ có các cạnh thể hiện mối quan hệ khác, ví dụ nút `CONTROL STRUCTURE` sẽ có cạnh `CONDITION` nối với nút thể hiện biểu thức điều kiện.

Bộ quản lý phạm vi. Được sử dụng trong Joern Frontend để thực hiện quản lý phạm vi của các biến, hàm, khai báo. Trong chương trình, một đơn vị thành phần sẽ có định danh riêng và hợp lệ trong một giới hạn nhất định. Trong quá trình duyệt cây AST, bộ quản lý phạm vi sẽ kiểm soát thông tin về các định danh và phạm vi hoạt động của chúng. Khi sử dụng một định danh hoặc khai báo một định danh mới, bộ quản lý phạm vi sẽ kiểm tra xem khai báo đó có hợp lệ trong phạm vi hay không. Với bộ quản lý phạm vi ta có thể xác định được quan hệ giữa việc khai báo và sử dụng một biến, hàm hay đơn vị cấu trúc khác. Từ đó có thể xác định được các cạnh giữa các nút trong cây CPG như cạnh `REF`, `REACHING DEFINITION`. Các cạnh này sau sẽ được sử dụng để xây dựng đồ thị CFG và đồ thị PDG.

Bộ giải. Cây CPG đã được xây dựng bằng bộ tạo AST CPG kết hợp với bộ quản lý phạm vi. Cây CPG sẽ tiếp tục được làm giàu thông tin bằng cách đi qua các bộ giải và trở thành đồ thị CPG chưa hoàn chỉnh. Mỗi bộ giải sẽ bổ sung một lớp thông tin riêng biệt. Các lớp thông tin này phụ thuộc vào ngữ cảnh của ngôn ngữ, do vậy số lượng bộ giải sẽ không cố định. Các bộ giải thao tác trên cây CPG nên có thể dùng chung cho nhiều ngôn ngữ, nhưng cũng có các bộ giải được xây dựng riêng cho một ngôn ngữ cụ thể. Một bộ giải có thể phụ thuộc vào kết quả của bộ giải trước đó hoặc chạy độc lập nên thứ tự chạy các bộ giải có ảnh hưởng. Ở đây, công cụ chỉ sử dụng hai bộ giải là bộ giải kiểu dữ liệu và bộ giải mô-đun để xử lý hệ thống kiểu và hệ thống module của Rust, các bộ giải khác sẽ được xây dựng trong tương lai. Sau khi chạy qua các bộ giải, cây CPG đã được bổ sung các lớp thông tin nhất định và đây là công đoạn cuối cùng của Joern Frontend.

Joern Backend. Khi chuyển đổi một ngôn ngữ sang đồ thị CPG, người dùng phải tự thực hiện công đoạn Language Frontend và Joern Frontend để xây dựng đồ thị CPG. Các thông tin có được từ hai bước trên sẽ được Joern Backend và các bộ hoàn thiện bên trong tận dụng để tự động xây dựng đồ thị CPG hoàn chỉnh. Các nút, cạnh mới về luồng điều khiển, luồng dữ liệu sẽ được thêm vào và kết nối với các cạnh, nút đã tồn tại. Không chỉ cung cấp lớp thông tin về luồng điều khiển và luồng dữ liệu, Joern Backend còn bổ sung các lớp thông tin riêng của đặc tả CPG của Joern như `FileSystem`, `CallGraph`, `Shortcuts`, `TagsAndLocation`, và `Annotation`. Những lớp thông tin này giúp tăng cường khả năng truy vấn, phân tích, và phát hiện các vấn đề tiềm ẩn trong mã nguồn một cách hiệu quả hơn. Kết quả cuối cùng của Joern Backend là một đồ thị CPG hoàn chỉnh, sẵn sàng phục vụ cho các bài toán như phân tích mã nguồn, phân tích bảo mật, hoặc trích xuất thông tin chuyên sâu. Điều này cho phép người dùng tận dụng tối đa tiềm năng của Joern trong việc hiểu và cải thiện chất lượng mã nguồn.

Persistence. Đồ thị CPG có thể được lưu trữ bền vững dưới dạng tệp nhị phân. Tệp này có thể tiếp tục được chuyển đổi thành kiểu dữ liệu tương thích với các cơ sở dữ liệu đồ thị như Neo4j để thực hiện các truy vấn phức tạp, hoặc sử dụng với các công cụ khác của Joern.

3.3 Chuyển đổi các cú pháp Rust sang đồ thị CPG

Phần này sẽ trình bày một số cú pháp của Rust khác biệt so với ngôn ngữ C/C++ và cách mà công cụ đã chuyển đổi sang đồ thị CPG cho các cú pháp này. Các cú pháp bao gồm: if let, while let, match, lifetime. Những đoạn mã nguồn và hình ảnh mô tả đồ thị CPG được sử dụng từ giờ đến cuối khóa luận đã được đơn giản hóa để dễ dàng thể hiện và minh họa. Các cạnh và các nút không phải trọng tâm của đồ thị CPG đã được loại bỏ để tập trung vào các tính năng cần trình bày.

3.3.1 Cú pháp if let

If else là cấu trúc điều khiển có mặt trong tất cả các loại ngôn ngữ phổ biến. Cấu trúc câu lệnh if else sẽ bao gồm một điều kiện và hai khối mã. Nếu điều kiện đúng, khối mã trong if sẽ được thực thi, ngược lại khối mã trong else sẽ được thực thi. Thông thường khối điều kiện sẽ là biểu thức trả về kết quả đúng hoặc sai của biểu

thức đó. Trong ngôn ngữ như C/C++ thì chỉ chấp nhận biểu thức điều kiện, việc sử dụng mệnh đề (có dấu hai chấm để kết thúc câu lệnh) là không hợp lệ. Với phương châm "Expression over statement" và nhằm mục đích tạo sự ngắn gọn, Rust cho phép thực hiện phép khai báo biến và gán giá trị cho biến trong cùng một câu lệnh bằng điều kiện if let. Cú pháp tổng quát của câu lệnh if let như sau:

```
if let <pattern> = <expression> {
    <block>
} else {
    <block>
}
```

Đoạn mã 3.1: Mã giả cho cú pháp tổng quát của if let.

Cú pháp 3.1 sẽ thực hiện hai công việc. Thứ nhất là kiểm tra xem `<expression>` có khớp với `<pattern>` hay không, nếu không khớp thì trả về *false*, nếu khớp thì trả về *true*. Thứ hai là khai báo các biến mới từ `<pattern>` nếu điều kiện thành công, các biến sẽ có phạm vi tồn tại trong khối lệnh điều kiện thành công.

```
let number: Option<i32> = None;

if let Some(i) = number {
    println!("Matched number {:?}!", i);
} else {
    // ...
}
```

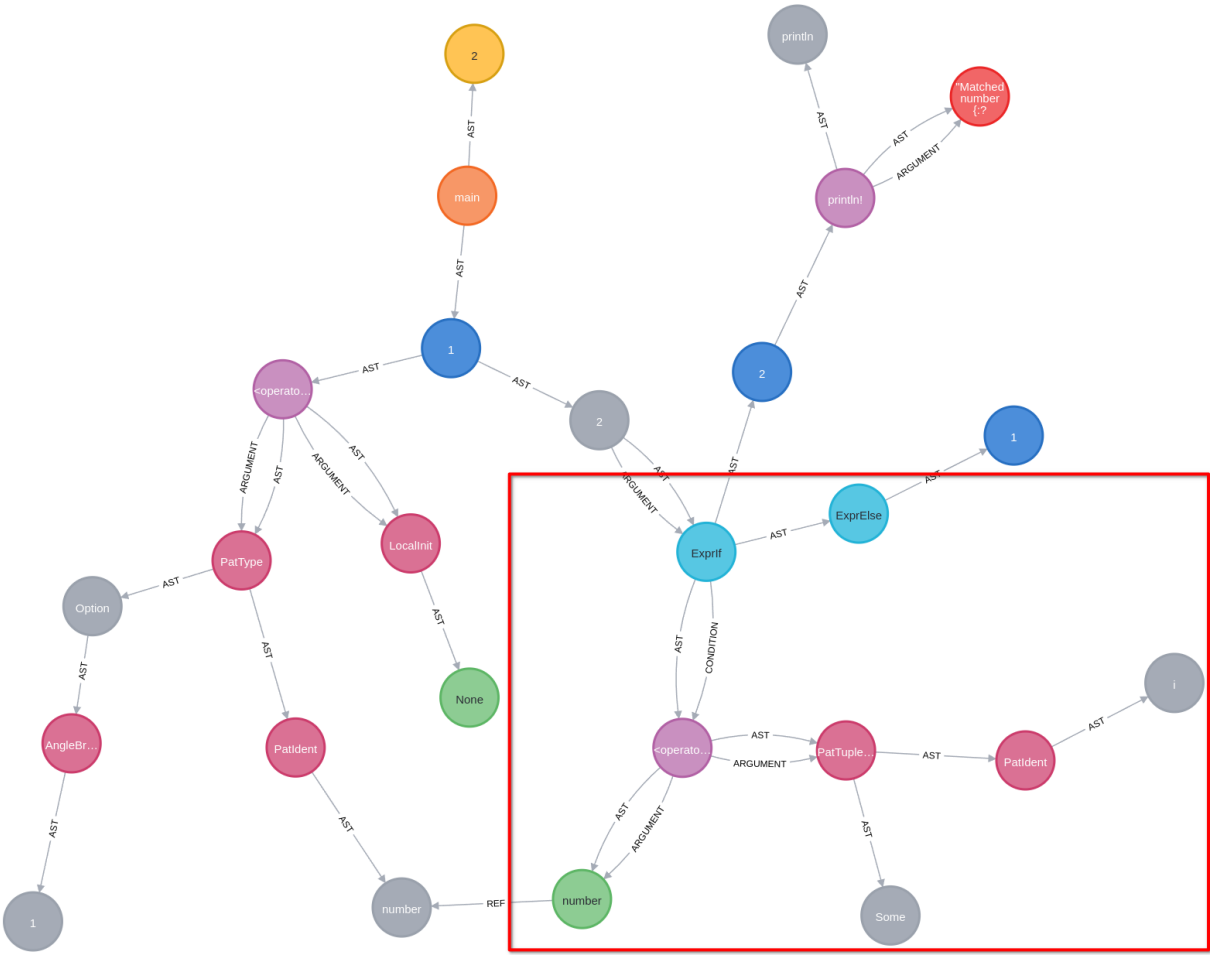
Đoạn mã 3.2: Ví dụ đoạn mã nguồn cho cú pháp if let.

Đoạn mã 3.2 sử dụng cú pháp if let với điều kiện là kiểm tra biến `number` có giá trị bên trong hay không, nếu có thì khai báo một biến `i` mới và gán giá trị cho biến `i`. Tiếp theo sẽ thực thi khối lệnh bên trong if. Biến `i` sẽ chỉ tồn tại trong khối lệnh if, và được sử dụng trong câu lệnh `println!`. Do thực hiện hai công việc trong cùng một câu lệnh nên khi quy đổi sang câu lệnh tương tự trong ngôn ngữ C/C++, cú pháp if let sẽ tương đương hai câu lệnh kiểm tra điều kiện và gán biến 3.3.

```
Object* obj = inputObj();

if (obj != nullptr) { // một câu lệnh kiểm tra điều kiện
    int number = *static_cast<int*>(obj); // một câu lệnh gán biến
    std::cout << "Matched " << number << "!" << std::endl;
}
```

Đoạn mã 3.3: Ví dụ đoạn mã nguồn cho cú pháp if let tương đương trong C++.



Hình 3.3: Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp if let 3.2.

Trong Hình 3.3, cạnh `CONDITION` của nút `EXPR IF` trỏ tới nút `ASSIGNMENT` và đồng thời khai báo biến mới với tên `i`. Mặc định, Joern không cho phép cạnh `CONDITION` trỏ tới nút `ASSIGNMENT` bởi vì trong ngôn ngữ C/C++ không thể lấy một phép gán giá trị làm điều kiện. Tuy nhiên, khóa luận đã thực hiện chỉnh sửa đặc tả CPG của Joern để cho phép cạnh `CONDITION` trỏ tới nút `ASSIGNMENT` trong trường hợp này.

Các mệnh đề trong khối điều kiện đúng được thực thi, nếu có sử dụng tới biến `i` thì sẽ tham chiếu tới biến `i` vừa được khai báo thông qua cạnh `REF`.

3.3.2 Cú pháp while let

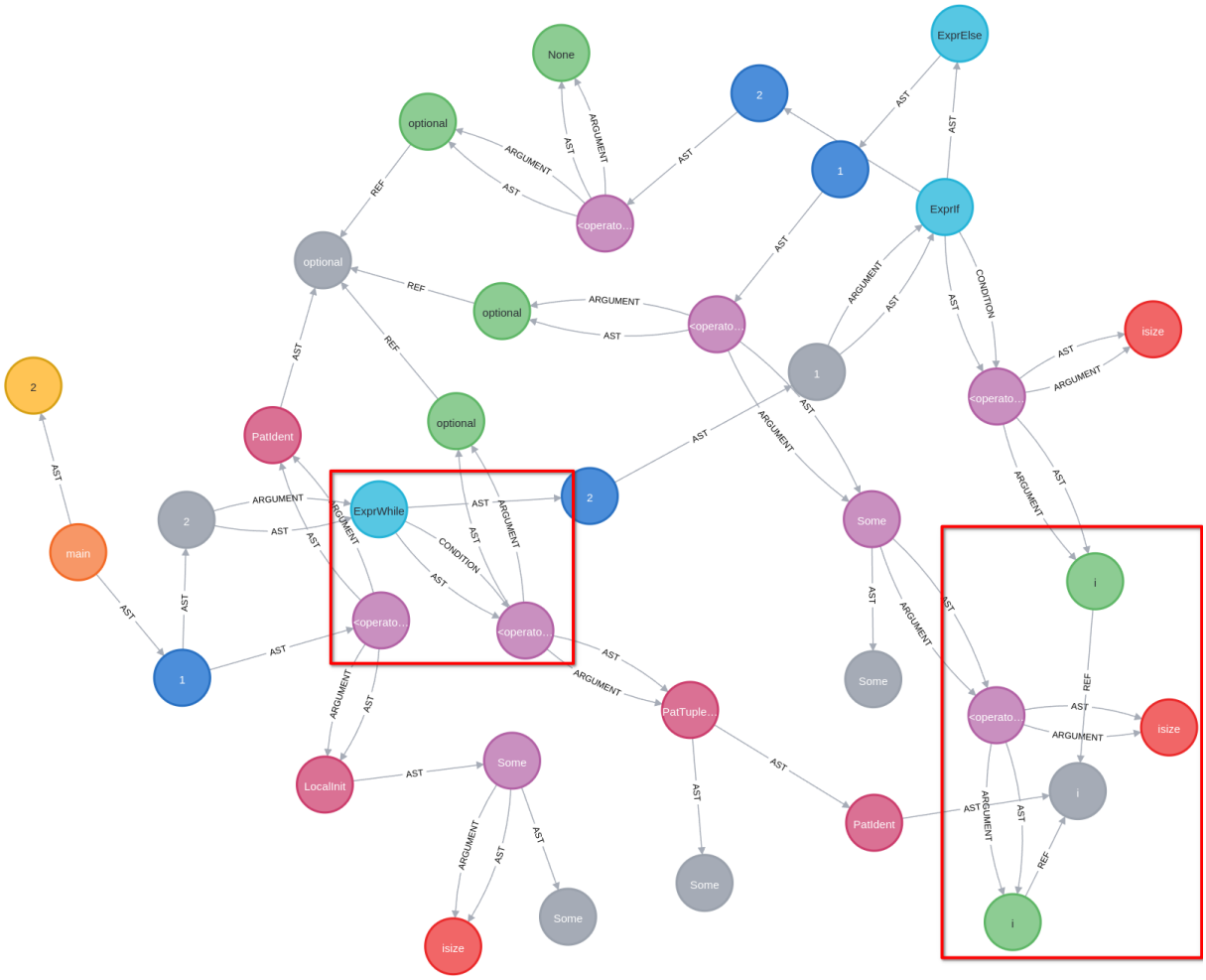
```
let mut optional = Some(0);

while let Some(i) = optional {
    if i > 9 {
        optional = None;
    } else {
        optional = Some(i + 1);
    }
}
```

Đoạn mã 3.4: Ví dụ đoạn mã nguồn cho cú pháp while let.

Tương tự với cú pháp if let ở trên, Rust cũng hỗ trợ việc khai báo biến làm điều kiện cho vòng lặp while. Đoạn mã 3.4 được hiểu là nếu biến `optional` có giá trị bên trong thì thực hiện vòng lặp, nếu không thì kết thúc vòng lặp. Đồng thời với mỗi lần lặp, sẽ có một biến `i` mới được khởi tạo, giá trị của `i` bằng giá trị của số nguyên bên trong biến `optional`. Nếu các mệnh đề trong khối mã điều kiện đúng có sử dụng biến `i` vừa được khai báo, chúng sẽ tham chiếu tới thông qua cạnh `REF`. Không chỉ vậy, vế phải của điều kiện có thể được gán lại liên tục trong quá trình lặp, đảm bảo vòng lặp kiểm tra trạng thái mới của biến `optional` sau mỗi lần lặp. Cú pháp if nằm trong khối mã điều kiện đúng được sử dụng với mục đích thể hiện các thay đổi động của biến `optional` trong suốt quá trình thực thi vòng lặp. Cú pháp if này là cú pháp kiểm tra điều kiện thông thường, khác với cú pháp if let được nhắc tới ở phần 3.3.1. Khi việc kiểm tra giá trị số nguyên bên trong biến `optional` thất bại, vòng lặp sẽ kết thúc.

Hình 3.4 mô tả đồ thị CPG cho đoạn mã nguồn while let ở trên. Nút `CONTROL STRUCTURE` thể hiện vòng lặp while, trong khi nút `ASSIGNMENT` biểu diễn điều kiện của vòng lặp và được nối với nút `CONTROL STRUCTURE` thông qua cạnh `CONDITION` Ngoài ra, biến `i` được khai báo riêng tại nút `LOCAL` và tham chiếu thông qua cạnh `REF` bởi các nút tương ứng với các mệnh đề trong khối mã.



Hình 3.4: Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp while let 3.4.

3.3.3 Cú pháp match

Ngoài việc sử dụng mệnh đề gán biến thành biểu thức điều kiện, tính hướng hàm của Rust còn thể hiện ở sự kết hợp giữa cơ chế pattern matching và kiểu dữ liệu đại số. Đoạn mã 3.5 cho thấy cấu trúc match không chỉ kiểm tra giá trị mà còn kết hợp với các mẫu phức tạp, bao gồm kiểm tra điều kiện, kiểm tra các kiểu dữ liệu khác nhau và so sánh. Điều này mang lại cho Rust tính linh hoạt cao hơn so với switch trong C/C++ khi chỉ so sánh giá trị nguyên thủy. Một điểm khác biệt quan trọng giữa match và switch là tính toàn diện của match. Rust yêu cầu các mẫu trong match phải bao quát tất cả các khả năng có thể xảy ra, nếu không trình biên dịch sẽ báo lỗi. Điều này giúp đảm bảo rằng không có tình huống nào bị bỏ qua, tăng cường độ an toàn của mã nguồn. Trong khi đó, switch trong C/C++ không yêu cầu bao quát tất cả các trường hợp, và việc bỏ sót một trường hợp có

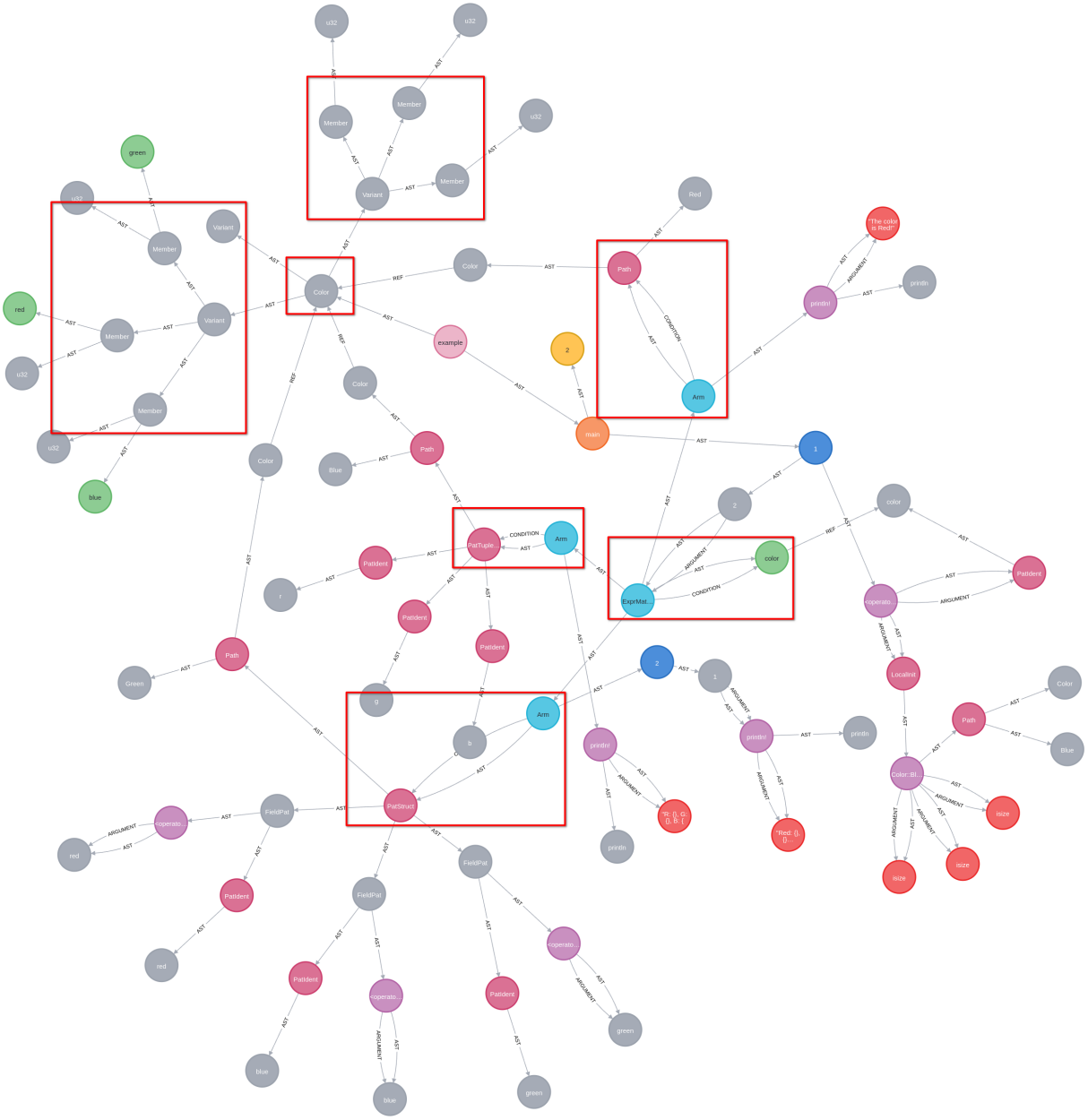
thể dẫn đến lỗi hoặc hành vi không mong muốn. Thêm vào đó, match trong Rust cho phép trích xuất và xử lý các thành phần của cấu trúc dữ liệu phức tạp ngay trong quá trình đối chiếu mẫu. Cú pháp match có thể thực hiện trên `tuple`, `enum`, `struct`, trong khi switch của C/C++ chỉ giới hạn cho các giá trị nguyên thủy.

```
enum Color {
    Red,
    Blue(u32, u32, u32),
    Green {
        red: u32,
        green: u32,
        blue: u32
    },
}

fn main() {
    let color = Color::Blue(0, 0, 255);
    match color {
        Color::Red =>
            println!("The color is Red!"),
        Color::Blue(r, g, b) =>
            println!("R: {}, G: {}, B: {}!", r, g, b),
        Color::Green { red, green, blue } => {
            println!("Red: {}, Green: {}, Blue: {}!", red, green, blue)
        }
    }
}
```

Đoạn mã 3.5: Ví dụ đoạn mã nguồn cho cú pháp match.

Trong Hình 3.5, các biến thể của `enum Color` được thể hiện bằng nút `VARIANT`. Biến thể `Red` do không có giá trị nên không có nút con nào khác. Biến thể `Blue` có giá trị là ba số nguyên tính theo chỉ mục nên sẽ có ba nút con `MEMBER` tương ứng, nhưng mỗi nút `MEMBER` này không có nút con `IDENTIFIER` chỉ ra tên thành phần. Ngược lại mỗi nút `MEMBER` của `Green` sẽ có nút con `IDENTIFIER` thể hiện tên của thành phần. Còn cú pháp match được thể hiện bằng nút `CONTROL STRUCTURE` và cạnh `CONDITION` nối với nút `IDENTITER` chỉ tới biến `color`. Mỗi trường hợp của match sẽ được thể hiện bằng nút `ARM` và mỗi nút `ARM` cũng có sẽ cạnh `CONDITION` nối tới nút thể hiện biểu thức điều kiện.



Hình 3.5: Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp match 3.5.

3.3.4 Cú pháp lifetime

Lifetime là cơ chế được sử dụng trong Rust để quản lý vòng đời của các biến tham chiếu. Quản lý lifetime một cách chặt chẽ đảm bảo rằng các biến tham chiếu không trỏ tới vùng nhớ không còn tồn tại. Lifetime có vai trò tương tự như kiểu tổng quát, nhưng thay vì định kiểu cho biến thì lifetime sẽ xác định thời gian hợp lệ cho biến tham chiếu. Để biểu diễn được kiểu dữ liệu tổng quát, đặc tả CPG của Joern đã có các nút như `TYPE`, `TYPE PARAMETER`, `TYPE ARGUMENT`. Do vậy để biểu

diễn được tính năng lifetime trên đồ thị CPG, ba loại nút mới đã được thêm vào đặc tả CPG bao gồm `LIFETIME`, `LIFETIME PARAMETER`, `LIFETIME ARGUMENT`. Để chỉ ra quan hệ ràng buộc giữa biến và lifetime hay quan hệ giữa các lifetime với nhau, ta sẽ thêm cạnh `OUT LIVE`. Cạnh `OUT LIVE` mang ý nghĩa nút nguồn phải có thời gian hợp lệ lớn hơn nút đích. Quy định về thời gian hợp lệ của một đơn vị thành phần so với đơn vị thành phần khác được tham khảo từ Rust Reference.

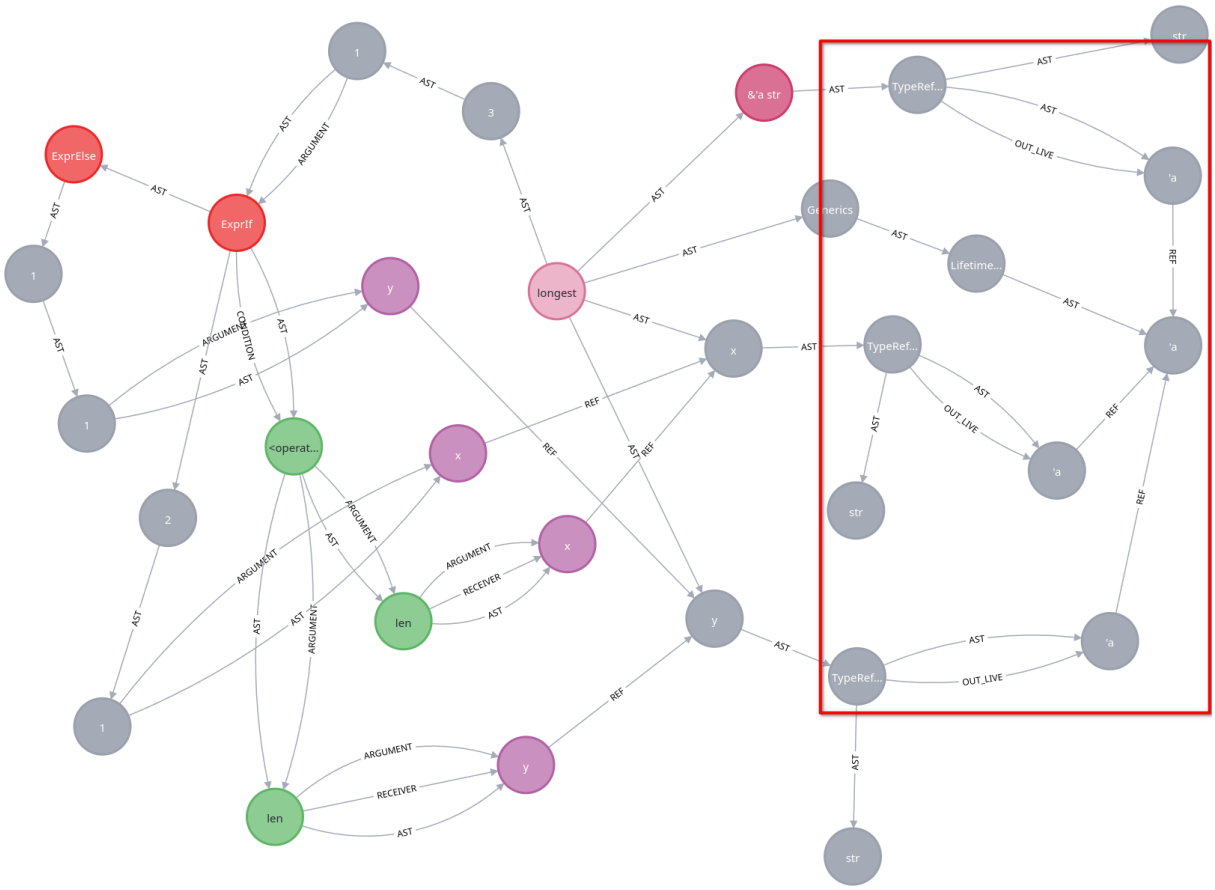
Nút `LIFETIME PARAMETER` và `LIFETIME ARGUMENT` được sử dụng cho các `struct`, `enum`, `trait` để chỉ ra lifetime tổng quát cho các biến tham chiếu. Loại nút `LIFETIME` sẽ thể hiện vòng đời thực sự của biến tham chiếu. Các biến tham chiếu sẽ được gán lifetime thông qua việc sử dụng dấu `"'"` đi trước tên lifetime, ví dụ như `'a`. Nếu biến đánh dấu lifetime `'a` thì sẽ có cạnh `OUT LIVE` trỏ từ nút `IDENTIFIER` của biến tới nút `LIFETIME` đại diện cho `'a` tương ứng. Nếu lifetime `'a` được giới hạn bởi lifetime `'b` thì sẽ có cạnh `OUT LIVE` từ nút `LIFETIME 'a` tới nút `LIFETIME 'b`.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Đoạn mã 3.6: Ví dụ đoạn mã nguồn cho cú pháp lifetime.

Các biến tham chiếu luôn có một lifetime tương ứng với nó, có thể được chỉ ra một cách tường minh hoặc không tường minh. Đoạn mã 3.6 mô tả một ví dụ bắt buộc phải khai báo lifetime một cách tường minh để chỉ rõ vòng đời của biến tham chiếu. Hàm `longest` nhận vào hai biến tham chiếu `x` và `y` cùng có lifetime `'a` và giá trị trả về cũng có lifetime `'a`. Trình biên dịch Rust có thể tự động suy diễn lifetime không tường minh cho hai biến tham chiếu đầu vào `x` và `y`, nhưng không thể suy diễn được lifetime cho giá trị tham chiếu trả về. Hơn nữa, giá trị trả về của hàm `longest` là một trong hai giá trị tham chiếu đầu vào. Trình biên dịch không thể biết được trong khi thực thi hàm `longest` thì giá trị trả về sẽ trỏ tới vùng nhớ của `x` hay `y`, do đó cần phải chỉ rõ lifetime cho giá trị trả về. Biến `x` và `y` cùng có lifetime `'a` tức là `x` và `y` sẽ có một khoảng thời gian mà `x` và `y` cùng tồn tại và đặt

tên cho khoảng thời gian đó là `'a`. Giá trị đầu ra tồn tại trong khoảng thời gian `'a` này thì luôn đảm bảo không trỏ tới vùng nhớ không còn tồn tại. Khi sử dụng hàm, nếu đối số đầu vào hay biến nhận giá trị trả về không thỏa mãn ràng buộc lifetime `'a` thì trình biên dịch Rust sẽ báo lỗi.



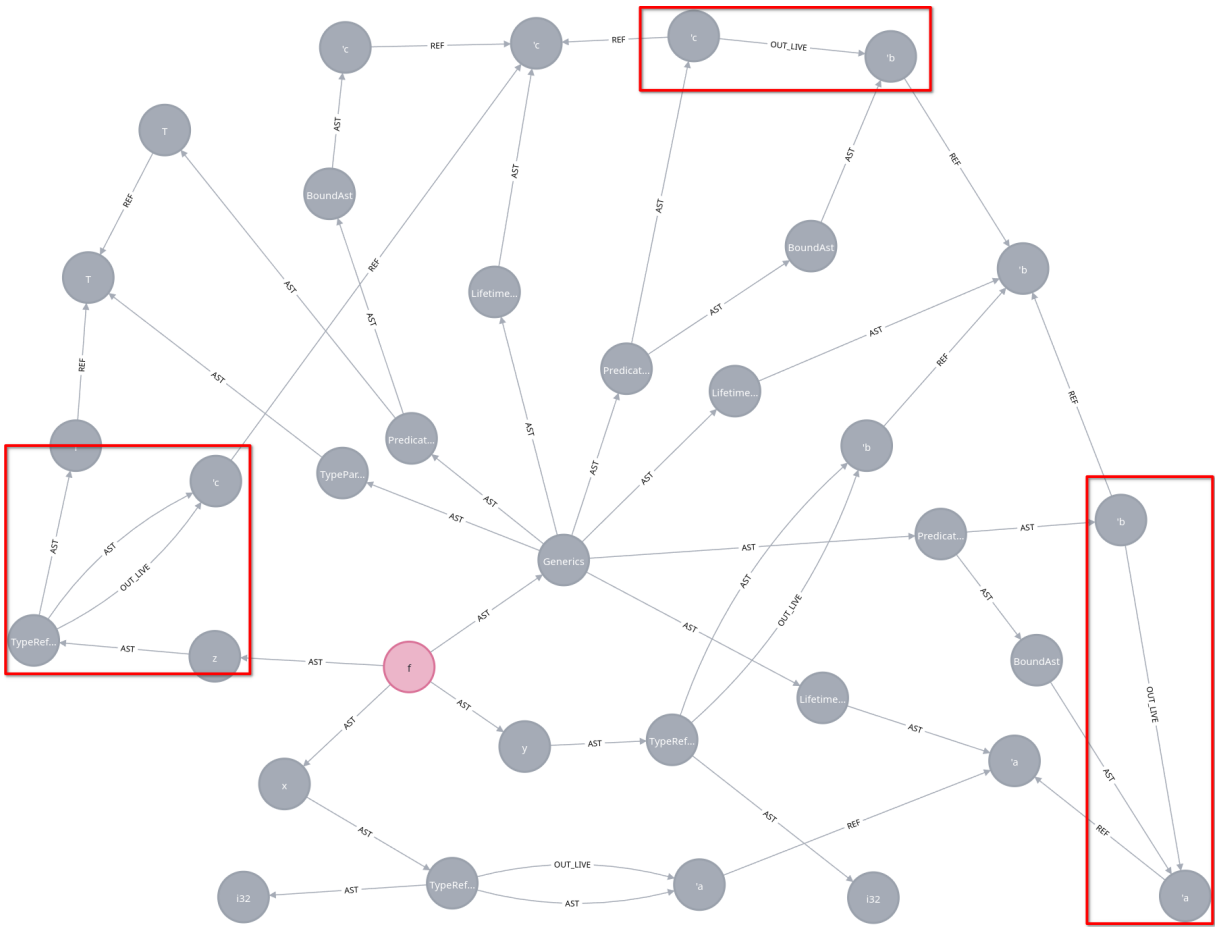
Hình 3.6: Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp lifetime 3.6.

Hình 3.6 thể hiện mối quan hệ lifetime giữa các biến tham chiếu và giá trị trả về của hàm. Hàm `longest` được thể hiện qua nút `METHOD`. Trong đó, nút `METHOD` có cạnh `AST` trỏ tới nút `GENERICS`, đại diện cho nút cha bọc lấy nhiều nút con thể hiện kiểu tổng quát và lifetime tổng quát. Nút `LIFETIME PARAMETER` thể hiện lifetime `'a` của hàm `longest`, và cũng là lifetime của các biến tham chiếu và giá trị trả về. Các biến tham chiếu `x` và `y` có cùng lifetime `'a`. Mối quan hệ này được thể hiện qua cạnh `OUT LIVE` từ nút `TYPE REF` tương ứng của biến `x` và `y` tới nút `LIFETIME 'a`. Nút `LIFETIME 'a` chỉ là đại diện tạm thời, nút này sẽ chiếu tới lifetime `'a` thực sự là nút `LIFETIME PARAMETER 'a` của hàm `longest` thông qua cạnh `REF`. Tương

tự, giá trị trả về của hàm cũng có lifetime `'a` và được thể hiện qua nút `TYPE REF` xuất phát từ nút `METHOD RETURN`, nối tới nút `LIFETIME PARAMETER 'a` bằng cạnh `OUT LIVE` và `REF`. Mối quan hệ lifetime giữa các biến tham chiếu và giá trị trả về đều được thể hiện qua các nút `LIFETIME` và cạnh `OUT LIVE` trên đồ thị CPG.

```
fn f<'a, 'b, 'c, T>(x: &'a i32, mut y: &'b i32, z: &'c T)
where
    'b: 'a,
    'c: 'b,
    T: 'c,
{
    // ...
}
```

Đoạn mã 3.7: Ví dụ đoạn mã nguồn cho cú pháp lifetime kết hợp cú pháp where.



Hình 3.7: Minh họa đồ thị CPG cho đoạn mã nguồn cú pháp where 3.7.

Trong Rust, kiểu tổng quát và ràng buộc của nó có thể được chỉ rõ thông qua cú pháp `where`. Cú pháp `where` cho phép thể hiện ràng buộc giữa kiểu dữ liệu tổng quát và kiểu lifetime tổng quát. Đoạn mã 3.7 minh họa về một hàm có nhiều lifetime, nhiều kiểu tổng quát và tham chiếu đầu vào kết hợp với cú pháp `where` để thể hiện mối quan hệ ràng buộc giữa chúng. Hàm `f` nhận vào ba biến tham chiếu `x`, `y`, `z` có lifetime tương ứng là `'a`, `'b`, `'c`. Cú pháp `'b: 'a` thể hiện ràng buộc lifetime `'b` của biến `y` phải lớn hơn hoặc bằng lifetime `'a` của biến `x`. Hình 3.7 mô tả cú pháp trên bằng đồ thị CPG có cạnh `OUT LIVE` từ nút `LIFETIME 'b` tới nút `LIFETIME 'a`. Điều này tương tự với việc lifetime `'c` của biến `z` phải lớn hơn hoặc bằng lifetime `'b` của biến `y`, cạnh `OUT LIVE` từ nút `LIFETIME 'c` tới nút `LIFETIME 'b` được thể hiện rõ ràng trên đồ thị CPG.

Không chỉ ràng buộc giữa các lifetime với nhau, hoàn toàn có thể ràng buộc lifetime cho một kiểu dữ liệu tổng quát. Kiểu dữ liệu `T` của biến `z` phải có lifetime lớn hơn `'c` thông qua cú pháp `T: 'c`. Cú pháp này có ý nghĩa trong trường hợp khi T là một kiểu `struct` tự định nghĩa, thì tất cả các thuộc tính tham chiếu của T cũng phải có lifetime lớn hơn hoặc bằng `'c`. Tồn tại cạnh `OUT LIVE` từ nút `TYPE REF` của kiểu `T` tới nút `LIFETIME 'c` để thể hiện mối quan hệ ràng buộc giữa chúng. Rust cung cấp hai cách để thể hiện ràng buộc cho kiểu dữ liệu và kiểu lifetime tổng quát. Một là khai báo ngay tại phần khai báo hàm, hai là sử dụng cú pháp `where` để thể hiện ràng buộc. Lưu ý rằng khai báo ngay tại hàm và cú pháp `where` không thay thế cho nhau mà cùng tồn tại để có thể kết hợp với nhau. Xuất hiện ngay tại hàm mang ý nghĩa khai báo nên sẽ sử dụng các nút `TYPE PARAMETER` và `LIFETIME PARAMETER` để thể hiện. Còn cú pháp `where` mang ý nghĩa ràng buộc nên sẽ sử dụng nút `LIFETIME`, cạnh `OUT LIVE`, cạnh `REF`.

Ownership, borrowing và lifetime là ba tính năng làm nên cơ chế an toàn về bộ nhớ trong Rust. Đây là bộ ba tính năng quan trọng làm cho Rust trở nên khác biệt so với các ngôn ngữ lập trình khác. Đồ thị CPG cần phải thể hiện được ba tính năng trên và cung cấp thông tin thông qua các nút, cạnh và thuộc tính phù hợp. Đặc biệt là tính năng lifetime dùng để thể hiện thời gian tồn tại của vùng nhớ mà một biến tham chiếu trỏ đến. Do đó việc thể hiện đúng quan hệ giữa lifetime và biến, lifetime với lifetime, biến với biến là rất quan trọng. Từ đó có thể khai thác thông tin để kiểm tra sự hợp lệ của biến tham chiếu, giúp phát hiện được các lỗi về bộ nhớ gây ra khi đánh dấu lifetime không chính xác.

Chương 4

Ứng dụng thực nghiệm và đánh giá

Chương 4 trình bày về các ứng dụng phân tích mã nguồn Rust bằng đồ thị CPG bao gồm phân tích mã nguồn có lỗ hổng và kỹ thuật học máy. Công cụ được sử dụng để phân tích mã nguồn của một số đoạn mã có lỗ hổng bảo mật được công bố trên RUSTSEC Database [4]. Chương này cũng sẽ ứng dụng đồ thị CPG cho bài toán học máy để phân loại mã nguồn Rust có lỗ hổng, thực hiện so sánh với mô hình học máy khác và chứng minh tiềm năng của đồ thị CPG dành cho ngôn ngữ Rust. Phần cuối cùng sẽ thảo luận các hạn chế hiện thời của công cụ.

4.1 Ứng dụng phân tích mã nguồn có lỗ hổng bảo mật

Tính đến năm 2023, tổng cộng có 17 thể loại lỗi được báo cáo về RUSTSEC Database [51]. Trong đó, lỗi về an toàn bộ nhớ và đa luồng chiếm tới gần hai phần ba tổng số loại lỗi. Mặc dù Rust có cơ chế để khắc phục những lỗi này, nhưng trong dự án thực tế vẫn tồn tại một số lỗ hổng, nhất là khi sử dụng tính năng `unsafe` (mã không an toàn). Để minh họa cho việc áp dụng đồ thị CPG vào thực tế, khóa luận sẽ trình bày ứng dụng của đồ thị CPG trên bốn lỗi nằm trong RUSTSEC Database. Bốn lỗi được minh họa có gán nhãn thể loại là `memory-exposure`, `memory-corruption` và `thread-safety`. Việc phân tích đoạn mã có lỗ hổng của các lỗi nêu trên sẽ cho thấy khả năng khai thác của đồ thị CPG cho ngôn ngữ Rust, đặc biệt đối với thể loại lỗi phổ biến nhất là an toàn bộ nhớ và đa luồng.

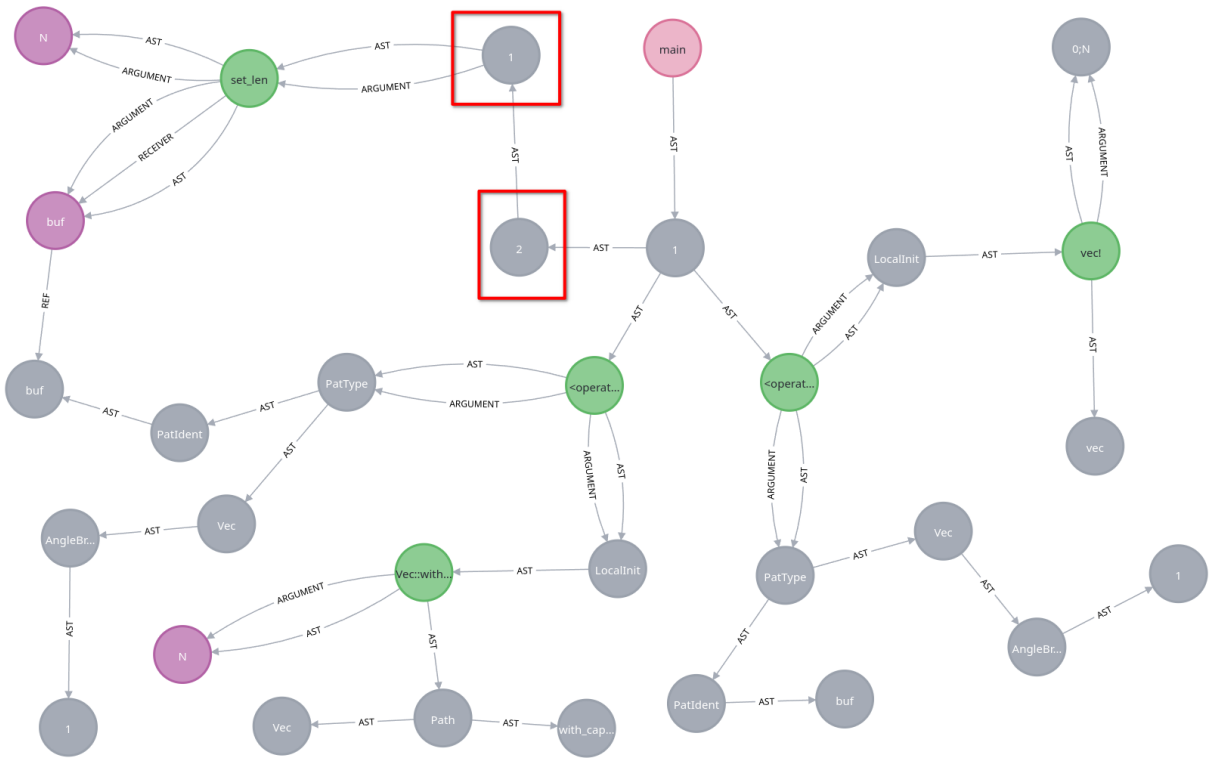
4.1.1 RUSTSEC-2021-0086

Đoạn mã 4.1 minh họa một lỗ hổng bảo mật phổ biến liên quan đến việc khởi tạo bộ nhớ không an toàn trong Rust, được gán nhãn `memory-exposure`. Lỗi xảy ra khi sử dụng hàm `Vec::with_capacity` để tạo một vectơ có dung lượng được định sẵn là N, sau đó tùy chỉnh độ dài của vectơ bằng hàm `set_len`. Tuy nhiên hàm `set_len`, một hàm được coi là `unsafe`, chỉ thay đổi biến thể hiện chiều dài của vectơ mà không khởi tạo giá trị cho các phần tử mới được thêm vào. Điều này dẫn

đến việc các phần tử chứa giá trị không xác định, tiềm ẩn nhiều nguy cơ như lỗi tràn bộ nhớ, truy cập trái phép vào vùng nhớ. Để sửa lỗi, ta có thể dùng macro `vec!` để khởi tạo vectơ với độ dài và dung lượng là N, giá trị mặc định của mỗi phần tử là 0. Trong Hình 4.1, nút `BLOCK` với thuộc tính `unsafe` đánh dấu bắt đầu khối các lệnh không an toàn. Các nút con của nút `BLOCK` này biểu diễn các mệnh đề, lời gọi hàm có nguy cơ gây ra lỗi. Ngoài ra, ngay sau nút `BLOCK` không phải là các mệnh đề không an toàn mà là một nút `EXPR RETURN`. Trong Rust, mỗi mệnh đề đều trả về một giá trị, nhưng trong trường hợp này hàm `set_len` không trả về giá trị nào. Do hạn chế của giao diện Neo4j, nút `BLOCK` với thuộc tính unsafe không thể được nhìn thấy một cách dễ dàng ngay trên Hình 4.1.

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

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



Hình 4.1: Minh họa đồ thị CPG cho đoạn mã nguồn RUSTSEC-2021-0086 4.1.

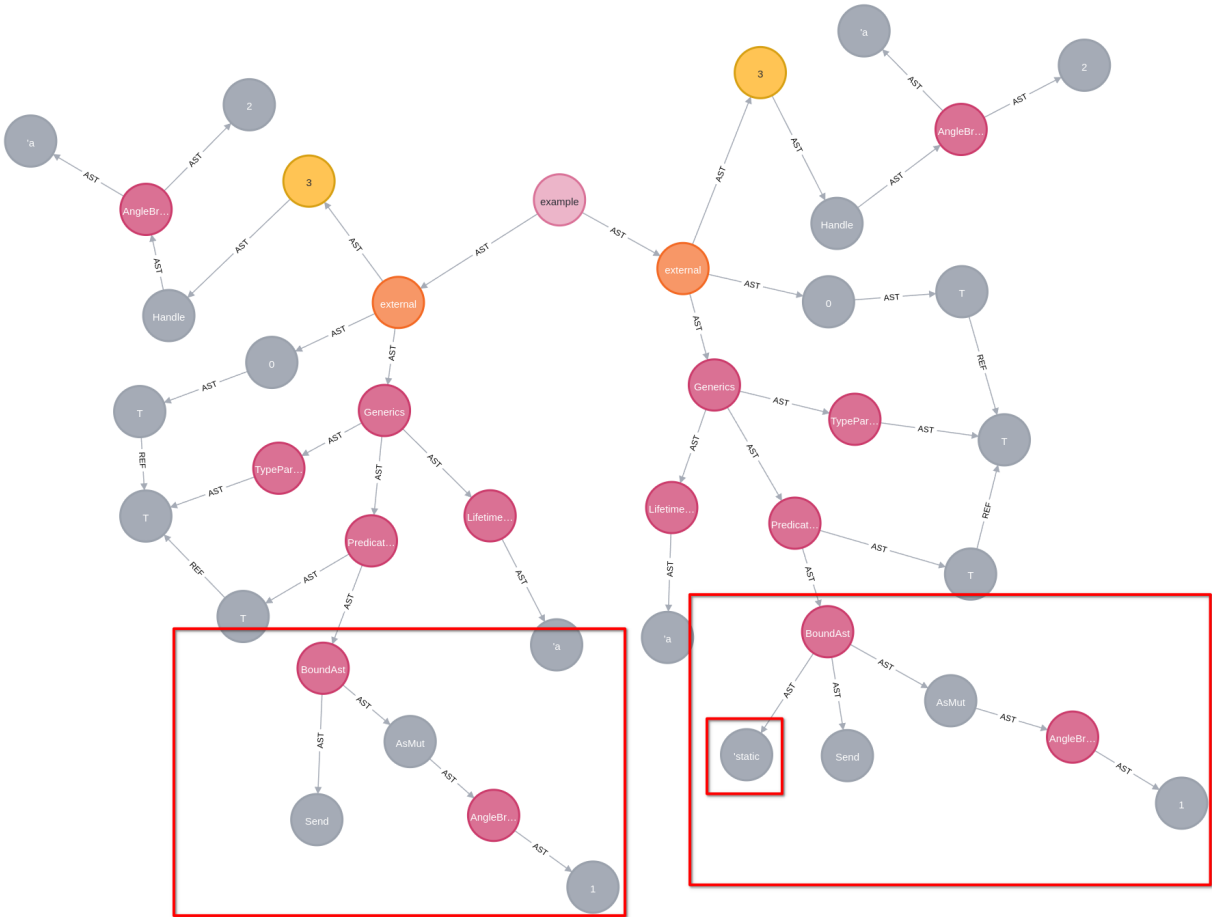
4.1.2 RUSTSEC-2022-0028

```rust
// Before fix
pub fn external<'a, T>(data: T) -> Handle<'a, Self>
where
    T: AsMut<[u8]> + Send,
{
    // ...
}

// After fix
pub fn external<'a, T>(data: T) -> Handle<'a, Self>
where
    T: AsMut<[u8]> + Send + 'static,
{
    // ...
}
```

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

Đoạn mã 4.2 mô tả lỗi xảy ra khi sử dụng hàm `external` mà không xác định đúng ràng buộc lifetime cho kiểu tổng quát `T`, được gán nhãn `memory-corruption` và `memory-exposure`. Trong bối cảnh này, `external` là hàm được dùng để tạo ra dữ liệu giao tiếp chung giữa Rust và JavaScript thông qua Web Assembly Binding [43]. Việc kiểu tổng quát `T` không được xác định đúng lifetime cho phép tạo ra một vùng dữ liệu có thể bị giải phóng trong khi chúng vẫn được tham chiếu bởi mã nguồn JavaScript. Để hai ngôn ngữ có thể giao tiếp được thì dữ liệu đó phải tồn tại trong suốt thời gian chạy của cả hai ngôn ngữ. Để sửa lỗi, cần thêm ràng buộc `T: 'static` để đảm bảo rằng dữ liệu được tham chiếu sẽ không bị giải phóng trong suốt thời gian tồn tại của chương trình. RUSTSEC-2022-0028 là một lỗi thực sự rất khó để phát hiện tự động vì nó liên quan đến bối cảnh mã nguồn, ở đây là dữ liệu giao tiếp giữa hai ngôn ngữ Rust và JavaScript thông qua Web Assembly Binding. Nếu chỉ sử dụng các mẫu khai phá tự động thông thường trên đồ thị CPG để phân tích thì chưa chắc phát hiện ra lỗi. Do cần kiến thức về bối cảnh bài toán nên sự can thiệp của con người là cần thiết trong trường hợp này. Trong Hình 4.2, các câu truy vấn trên đồ thị CPG kết hợp với kiến thức miền của các chuyên gia cần được xây dựng thủ công để có thể phát hiện ra lỗ hổng.



Hình 4.2: Minh họa đồ thị CPG cho đoạn mã nguồn RUSTSEC-2022-0028 4.2.

4.1.3 RUSTSEC-2020-0044

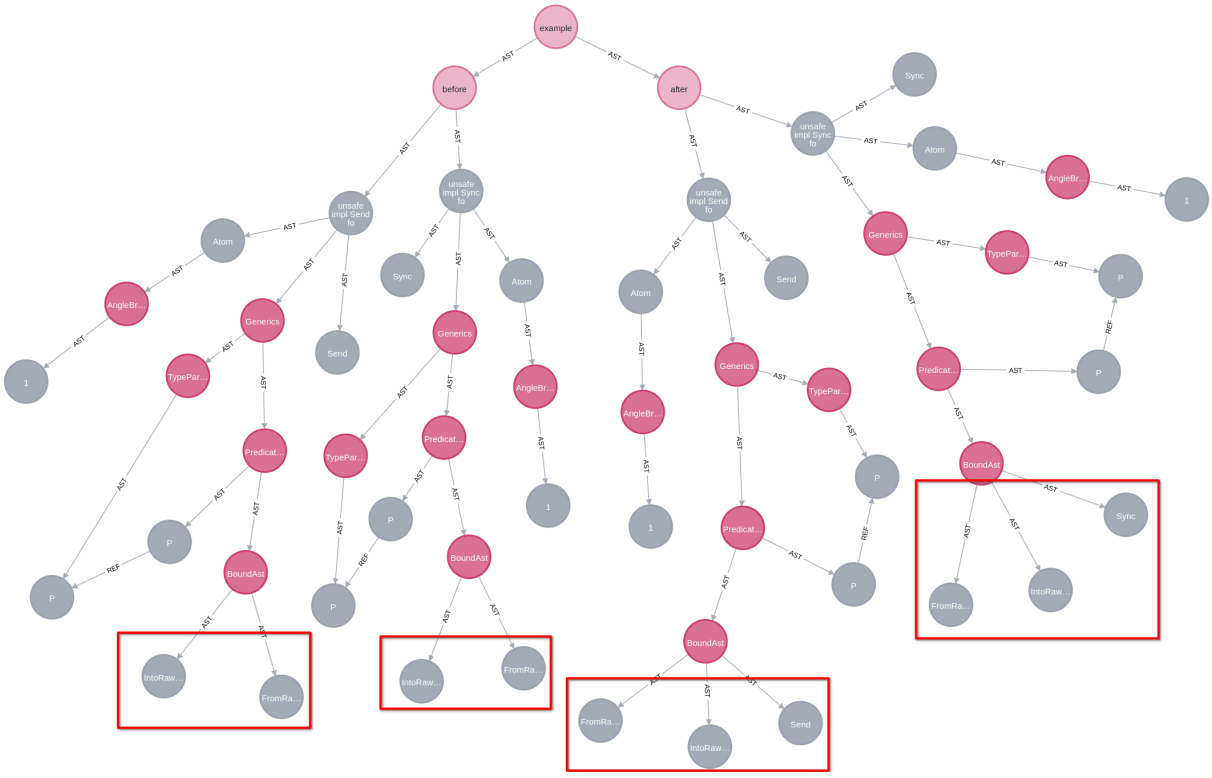
```
// Before fix
unsafe impl<P> Send for Atom<P> where P: IntoRawPtr + FromRawPtr {}
unsafe impl<P> Sync for Atom<P> where P: IntoRawPtr + FromRawPtr {}

// After fix
unsafe impl<P> Send for Atom<P> where P: IntoRawPtr + FromRawPtr + Send {}
unsafe impl<P> Sync for Atom<P> where P: IntoRawPtr + FromRawPtr + Sync {}
```

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

Đoạn mã 4.3 mô tả lỗi liên quan đến việc không triển khai thuộc tính `Send` và `Sync` cho kiểu dữ liệu con bên trong kiểu cha tên `Atom`, được gán nhãn `thread-safety`. Trong bối cảnh này, `Atom` đóng vai trò như một hộp chứa cho kiểu dữ liệu `P` bên

trong, giúp quản lý việc sử dụng bộ nhớ an toàn hơn. Tuy nhiên, việc chỉ mình kiểu cha `Atom` được cài đặt thuộc tính `Send` và `Sync` không đảm bảo rằng kiểu dữ liệu con `P` cũng phải cài đặt thuộc tính `Send` và `Sync`. Điều này có nghĩa việc truy cập đến `Atom` có thể an toàn nhưng khi truy cập vào dữ liệu kiểu `P` bên trong thì không an toàn, có thể dẫn đến các vấn đề như sử dụng bộ nhớ sau khi đã giải phóng hoặc tương tranh dữ liệu cho kiểu dữ liệu `P` bên trong. Để sửa lỗi này, cần đảm bảo rằng kiểu `P` cũng phải được đánh dấu thuộc tính `Send` và `Sync`.



Hình 4.3: Minh họa đồ thị CPG cho đoạn mã nguồn RUSTSEC-2020-0044 4.3.

Lỗi RUSTSEC-2020-0044 là một lỗi xảy ra thường xuyên khi lập trình đa luồng trong Rust. Kiểu cha muốn đảm bảo an toàn về đa luồng thì kiểu con cũng phải đảm bảo an toàn về đa luồng. Trong bối cảnh của Rust, điều này có nghĩa là nếu kiểu cha muốn được gửi và truy cập giữa các luồng một cách an toàn bằng thuộc tính `Send` và `Sync` thì kiểu con cũng phải có khả năng này. Khi được biểu diễn trực quan trên đồ thị 4.3, ta hoàn toàn có thể thấy một mẫu duyệt đồ thị để phát hiện lỗi này. Một mẫu khai phá đồ thị phổ biến có thể được xây dựng và sử dụng cho nhiều đoạn mã khác nhau. Các truy vấn hoặc phân tích dựa trên mẫu này có thể được áp dụng trên đồ thị CPG để phát hiện các lỗi tương tự.

4.1.4 RUSTSEC-2021-0130

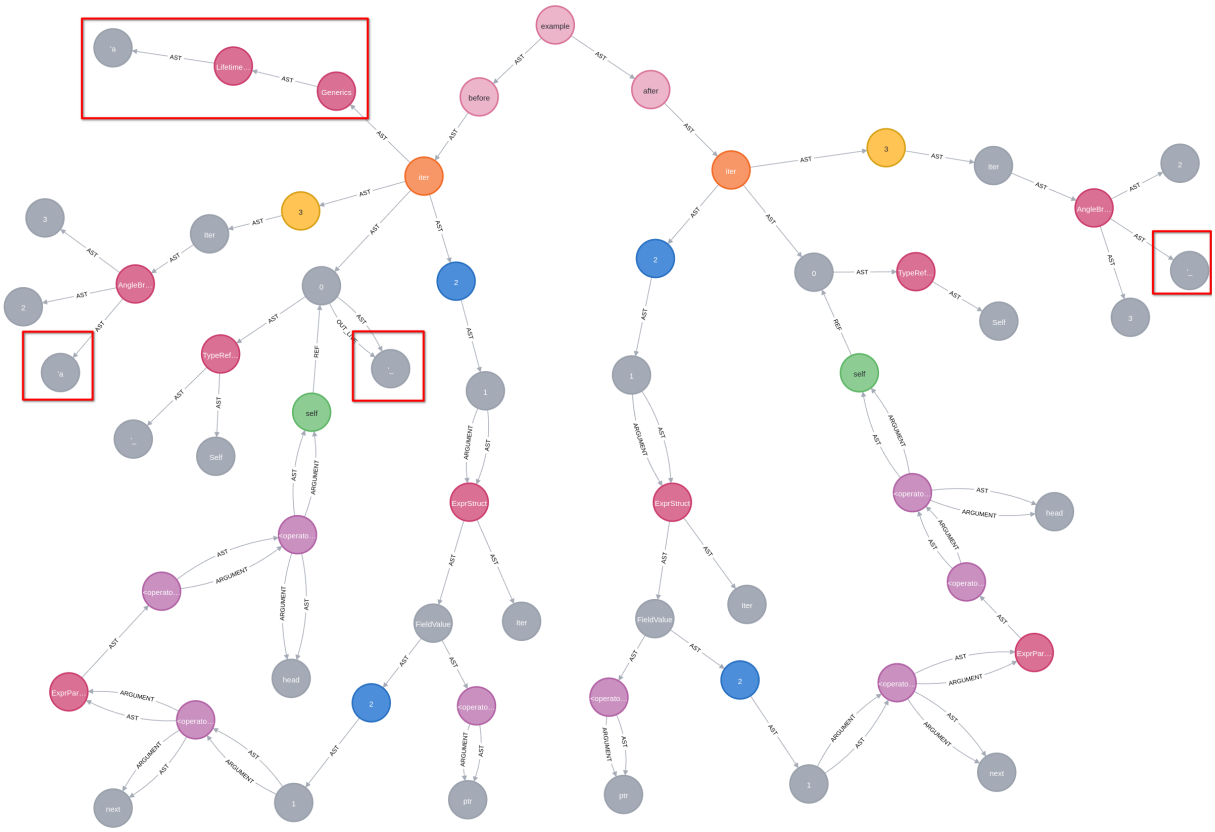
```
// Before fix
pub fn iter<'a>(&'_ self) -> Iter<'a, K, V> {
    Iter { ptr: unsafe { (*self.head).next }, }
}
// After fix
pub fn iter(&self) -> Iter<'_, K, V> {
    Iter { ptr: unsafe { (*self.head).next }, }
}
```

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

Đoạn mã 4.4 mô tả một lỗi sử dụng bộ nhớ sau khi đã giải phóng, được gán nhãn `memory-corruption`. `Iter` là một `struct` được sử dụng để duyệt qua các phần tử trong một danh sách liên kết. Khi `Iter` được trả về từ hàm `iter`, nó sẽ trỏ tới `self.head`, nhưng `self.head` có thể bị giải phóng trước khi `Iter` được sử dụng, dẫn đến việc sử dụng bộ nhớ sau khi đã giải phóng. Nguyên nhân là do quan hệ lifetime giữa biến `self` và giá trị trả về kiểu `Iter` không đồng bộ. Biến `self` được chỉ định lifetime là `'_`, tức là để trình biên dịch tự động gán một giá trị lifetime mặc định, trong khi đó kiểu trả về `Iter` được chỉ định lifetime là `'a`. Trong trường hợp này, `'_` và `'a` không có liên hệ với nhau do đó lifetime của `Iter` có thể lớn hơn lifetime của `self`, dẫn đến việc truy cập đến `self.head` sau khi `self` đã bị giải phóng. Để sửa lỗi, cần sử dụng chung một lifetime cho `self` và `Iter` thông qua việc sử dụng ba quy tắc lược bỏ lifetime [31]. Nếu không chỉ định lifetime một cách tường minh, ba quy tắc lược bỏ lifetime sẽ được trình biên dịch sử dụng để tự động xác định lifetime của các biến dựa trên cấu trúc của hàm, bao gồm:

1. Mỗi tham số kiểu tham chiếu của hàm mặc định có một lifetime riêng.
2. Khi hàm có một tham số kiểu tham chiếu duy nhất, và kiểu trả về cũng là kiểu tham chiếu thì lifetime của kiểu trả về sẽ là lifetime của tham số tham số kiểu tham chiếu duy nhất đó.
3. Nếu hàm có nhiều tham số kiểu tham chiếu nhưng có một tham số là `self` (`self` tương đương với `this` trong C++ và Java), thì lifetime của `self` sẽ được gán cho kiểu tham số trả về.

Trong trường hợp này, ta sửa lỗi bằng cách không chỉ định lifetime cho `self` mà để trình biên dịch tự xác định, lifetime của `Iter` sẽ để là `'_`. Ký hiệu `'_` tức là tự động suy diễn và áp dụng quy tắc số ba thì lifetime của `Iter` sẽ được gán bằng lifetime của `self`. Từ đó lifetime của `Iter` sẽ không lớn hơn lifetime của `self` và việc truy cập đến `self.head` sau khi `self` đã bị giải phóng sẽ không xảy ra.



Hình 4.4: Minh họa đồ thị CPG cho đoạn mã nguồn RUSTSEC-2021-0130 4.4.

Đánh dấu lifetime một cách tường minh cần được thực hiện cẩn thận để tránh được các lỗi liên quan đến bộ nhớ. Việc đánh dấu lifetime không chính xác có thể đánh lừa trình biên dịch và dẫn đến việc sử dụng bộ nhớ sau khi đã giải phóng, con trỏ treo. Lifetime của kiểu trả về phải đảm bảo không lớn hơn lifetime của tham số truyền vào nếu kiểu trả về có truy cập đến bộ nhớ của tham số truyền vào. Có thể sử dụng ba quy tắc lược bỏ lifetime để suy diễn việc xác định lifetime giữa tham số và tham số, giữa tham số và kiểu trả về. Kết hợp với đồ thị CPG được minh hoạ bằng Hình 4.4, các luật có thể được đề ra để phát hiện sự không đồng bộ về lifetime giữa các biến dựa vào các nút `LIFETIME`, `LIFETIME PARAMETER` và các cạnh thể hiện mối quan hệ giữa chúng như `REF`, `OUT LIVE`.

4.2 Ứng dụng bài toán học máy trên đồ thị CPG

Để chứng minh tiềm năng sử dụng, khóa luận xây dựng một thực nghiệm sử dụng kỹ thuật học máy để phát hiện lỗ hổng bảo mật trong mã nguồn Rust dựa trên đồ thị CPG. Bài toán đặt ra là phân loại tệp mã nguồn Rust có lỗ hổng bảo mật và không có lỗ hổng bảo mật. Thực nghiệm sẽ tiến hành so sánh độ hiệu quả giữa mô hình học máy sử dụng đồ thị CPG và mô hình học máy học sử dụng mã nguồn Rust truyền thống. Mô hình học máy chỉ sử dụng dữ liệu mã nguồn Rust, tạm gọi là Baseline, sẽ sử dụng phương pháp Word2Vec [2] để biểu diễn mã nguồn và Logistic Regression [20] để phân loại. Đối với bài toán học máy để phát hiện lỗ hổng bảo mật trên đồ thị CPG, trước đây đã có phương pháp Devign, một phương pháp kinh điển cho lớp bài toán này [52]. Devign đã sử dụng đồ thị CPG cho ngôn ngữ C/C++, do đó hoàn toàn có thể áp dụng phương pháp Devign với đồ thị CPG cho ngôn ngữ Rust trong khóa luận.

Bộ dữ liệu được sử dụng bao gồm 788 tệp mã nguồn, trong đó 381 tệp có lỗ hổng bảo mật và 407 tệp không có lỗ hổng bảo mật. Đối với 381 tệp có lỗ hổng bảo mật, các tệp này được trích xuất từ bộ dữ liệu trong nghiên cứu của David Lo và cộng sự [51]. Còn 407 tệp không có lỗ bảo mật được thu thập từ 100 dự án Rust mã nguồn mở nhiều sao nhất trên Github [8]. Tập dữ liệu được chia thành ba phần, 80% dữ liệu được sử dụng cho quá trình huấn luyện, 10% dữ liệu được sử dụng cho kiểm chứng, 10% dữ liệu được sử dụng cho kiểm thử. Để đánh giá, mỗi phương pháp được chạy 10 lần đối với tập dữ liệu như trên. Kết quả của các chỉ số là giá trị trung bình của 10 lần chạy. Kết quả thực nghiệm thể hiện ở bảng phía dưới.

Với bài toán phân loại, thang đo đánh giá hiệu suất của phương pháp sẽ bao gồm các chỉ số Accuracy, Precision, Recall và F1-score. Accuracy biểu thị tỷ lệ phần trăm các dự đoán đúng trên tổng số các dự đoán, thể hiện độ chính xác tổng thể của mô hình, nhưng có thể không phản ánh đúng với dữ liệu mất cân bằng. Precision đo độ chính xác trong việc phân loại đúng một lớp cụ thể, hữu ích khi cần giảm thiểu dự đoán dương tính sai. Recall, hay độ nhạy, đánh giá khả năng phát hiện đầy đủ các mẫu thuộc một lớp, đặc biệt quan trọng trong việc giảm thiểu bỏ sót các trường hợp dương tính. F1-score là chỉ số thể hiện sự cân bằng giữa Precision và Recall, giúp đánh giá hiệu suất chung của cả hai chỉ số quan trọng này.

Bảng 4.1: Bảng so sánh phương pháp Baseline và phương pháp Devign (Rust).

	Baseline	**Devign (Rust)**
Accuracy	0.73	**0.76**
Precision	0.73	**0.76**
Recall	0.70	**0.99**
F1-score	0.72	**0.86**

Trong Bảng 4.1, phương pháp Devign đã cho kết quả tốt hơn so với phương pháp Baseline ở tất cả các chỉ số. Ta thấy được Accuracy hơn 0.03, Precision hơn 0.03, Recall hơn 0.29, F1-Score hơn 0.14. Điều này chứng tỏ rằng đồ thị CPG cho ngôn ngữ Rust có tiềm năng áp dụng cho bài toán phát hiện lỗ hổng bảo mật kết hợp kỹ thuật học máy. Mã nguồn và bộ dữ liệu sử dụng trong thực nghiệm được lưu trữ lần lượt tại địa chỉ devign, rust-ecosystem.

Chỉ số Accuracy của phương pháp Devign cho ngôn ngữ Rust mới chỉ đạt được ở mức 0.76 bởi một vài nguyên nhân. Thứ nhất là kích cỡ của bộ dữ liệu cho ngôn ngữ Rust. Hiện tại, các mã nguồn có lỗ hổng được xác nhận đều lấy từ RUSTSEC Database. Rust là ngôn ngữ mới phát triển gần đây, hệ sinh thái chưa lớn mạnh nên số lượng mã nguồn có lỗ hổng được báo cáo không nhiều. Thứ hai là đồ thị CPG của Rust vẫn chưa đầy đủ hoàn toàn. Tồn tại các tính năng của Rust như marco, module chưa thể phân tích ra cây AST hay lấy được các lớp thông tin cần thiết. Các hạn chế này sẽ được trình bày chi tiết ở phần 4.3. Thứ ba là giới hạn trong cài đặt của phương pháp Devign. Hiện tại, phiên bản cài đặt của Devign là mô phỏng lại từ bài báo khoa học gốc. Devign mới chỉ có thể sử dụng lớp thông tin về AST và CFG mà không có PDG, do vậy không thể khai thác được hết toàn bộ thông tin từ đồ thị CPG. Nếu có thể khắc phục được các hạn chế kể trên, phương pháp Devign có thể đạt được kết quả tốt hơn nữa.

Dù tồn tại những hạn chế, kết quả thực nghiệm vẫn cho thấy tiềm năng khai thác to lớn của đồ thị CPG cho ngôn ngữ Rust. Đồ thị CPG có thể được sử dụng cho các lớp bài toán cần đến phân tích mã nguồn như phát hiện lỗ hổng bảo mật, phân loại mã nguồn hay các ứng dụng khác trong tương lai. Ngoài ra, việc cải thiện và hoàn thiện đồ thị CPG cho ngôn ngữ Rust sẽ mở ra nhiều hướng nghiên cứu và ứng dụng mới. Điều này không chỉ giúp nâng cao hiệu quả của các phương pháp hiện tại mà còn thúc đẩy sự phát triển của hệ sinh thái ngôn ngữ Rust.

4.3 Hạn chế

4.3.1 Chưa hỗ trợ cú pháp Macro

Trong ngôn ngữ lập trình C/C++ và Rust có khái niệm macro, là cách viết mã nguồn để sinh ra mã nguồn khác [35]. Ngôn ngữ C/C++ có bộ tiền xử lý trước khi cho mã nguồn vào trình biên dịch, do đó sau khi được tiền xử lý thì mã nguồn C/C++ đã được mở rộng toàn bộ macro. Tuy nhiên, Rust không giống C/C++, macro của Rust không được xử lý trước khi sinh cây AST mà sẽ được xử lý sau khi sinh cây AST nhưng trước khi đi vào pha phân tích ngữ cảnh của trình biên dịch [36]. Như đã đề cập ở phần các trước, công cụ hiện tại sử dụng thư viện syn để sinh cây AST cho mã nguồn và syn không hỗ trợ xử lý macro. Do đó tất cả các mã lệnh nằm bên trong macro sẽ không được xử lý, dẫn đến việc không thể sinh cây AST cho đoạn mã lệnh sử dụng macro. Tất cả các đoạn lệnh nằm trong một lời gọi macro hiện tại được xem như một chuỗi ký tự. Không chỉ vậy macro trong Rust sử dụng DSL (Domain-Specific Language) riêng. DSL này gần với ngôn ngữ Rust nhưng có sự mở rộng, biến đổi để phù hợp với tính năng macro, do đó gây khó khăn cho việc sinh cây AST.

Để xử lý trường hợp trên, một bước tiền xử lý mã nguồn có thể được thêm vào để mở rộng macro như C/C++. Công cụ có thể sử dụng đến tính năng mở rộng marco của trình biên dịch Rust [34]. Tính năng này có tác dụng đưa đoạn mã macro mà lập trình viên nhìn thấy thành đoạn mã mà trình biên dịch nhìn thấy. Mã nguồn sau khi được mở rộng sẽ thay thế các lời gọi macro bằng định nghĩa gốc của macro đó. Các macro được nạp sẵn của ngôn ngữ Rust như `println!`, `vec!` hay kể cả macro do người dùng định nghĩa cũng sẽ được xử lý. Không chỉ vậy, mã nguồn sau khi được mở rộng sẽ có được các thông tin bị ẩn đi như các mặc định của Rust bao gồm các hàm, các biến được nạp sẵn trong ngôn ngữ. Tuy nhiên, việc mở rộng macro trước khi sinh cây AST sẽ làm cho mã nguồn bị biến đổi so với mã nguồn gốc, đồng thời tăng kích cỡ và độ lớn của mã nguồn. Việc thêm các thông tin ẩn mà lập trình viên không nhìn thấy ở mã nguồn gốc có thể gây nhầm lẫn. Điều này cũng đồng nghĩa với việc quá trình sinh cây AST cho mã nguồn sau khi xử lý macro sẽ phức tạp và tốn nhiều thời gian hơn.

4.3.2 Chưa hỗ trợ cơ chế Module

Cơ chế module trong Rust tương tự namespace trong C++ hay package trong Java [32]. Module chia nhỏ mã nguồn thành các phần nhỏ hơn, tổ chức, sắp xếp và quản lý mã nguồn. Module giúp tái sử dụng mã nguồn, tránh xung đột tên biến, tên hàm giữa các module khác nhau. Mỗi module trong Rust có thể là một tệp riêng biệt hoặc nằm chung trong một tệp mã nguồn. Các module có thể được tổ chức thành một hệ thống phân cấp, với các module con được khai báo bên trong các module cha. Ngoài ra, có một số tệp được coi là tệp đặc biệt trong cấu trúc mã nguồn Rust. Ví dụ như tệp với tên `mod.rs`, đây là tệp module gốc của thư mục chứa nó. Tất cả các module con trong thư mục đó sẽ không được phép truy cập trực tiếp từ bên ngoài mà phải thông qua tệp `mod.rs`. Còn có tệp tên `main.rs` hoặc `lib.rs` để đánh dấu điểm đầu vào của chương trình và xác định xem dự án là một thư viện hay một ứng dụng. Rust còn cung cấp cơ chế workspace, cho phép quản lý nhiều dự án nhỏ trong cùng một dự án lớn, mỗi dự án là một thư mục con trong thư mục gốc lớn nhất [30]. Để kiểm soát khả năng truy cập, Rust sử dụng cơ chế visibility [33]. Mặc định các thành phần trong module chỉ có phạm vi truy cập giới hạn trong module nó được định nghĩa. Để làm cho một đơn vị có thể truy cập được từ các module khác, ta sử dụng từ khóa `pub` để đánh dấu cho đơn vị cần mở rộng quyền truy cập. Rust còn có cơ chế đường dẫn để định danh một thành phần trong mã nguồn. Ta có thể truy cập đến thành phần của module khác bằng cách chỉ ra đường dẫn định danh hợp lệ. Một đường dẫn có thể là đường dẫn tuyệt đối hoặc đường dẫn tương đối. Ví dụ từ khóa `self` dùng để chỉ tới module hiện tại, từ khóa `super` để chỉ tới module cha của module hiện tại, từ khóa `crate` để chỉ tới module gốc của dự án. Hệ thống module phức tạp của Rust làm tăng đáng kể độ khó cho việc xử lý quan hệ giữa các module trong cây AST và phân tích ngữ cảnh. Các khái niệm như workspace, module, visibility, cơ chế đường dẫn và những tệp tin đặc biệt tạo nên sự phức tạp cho việc phân tích ngữ cảnh của mã nguồn. Hiện tại, công cụ chưa xử lý được các tính năng xung quanh các khái niệm này.

Kết luận

Đảm bảo chất lượng mã nguồn là một trong những vấn đề quan trọng trong quy trình phát triển phần mềm. Việc sử dụng các công cụ đảm bảo chất lượng mã nguồn giúp giảm thiểu lỗ hổng bảo mật, chi phí bảo trì và tăng cường hiệu suất. Đối với Rust, một ngôn ngữ lập trình mới nổi nhưng có tốc độ phát triển vượt trội, nhu cầu đảm bảo chất lượng mã nguồn là vô cùng cấp thiết. Rust có vai trò kế thừa và nối tiếp C/C++, do vậy Rust có khả năng tương thích mạnh mẽ với C/C++. Các nghiên cứu đã có về đảm bảo chất lượng mã nguồn cho C/C++ hoàn toàn có thể áp dụng được cho Rust. Đã có các nghiên cứu được đề ra, từ kiểm chứng, kiểm thử động và phân tích tĩnh. Các giải pháp này sử dụng các đầu vào riêng biệt của Rust, do vậy còn tồn tại hạn chế về khả năng tương thích với các công cụ và nghiên cứu đã có từ trước. Nhu cầu về sự mở rộng và chuyển tiếp nhanh chóng giữa Rust và C/C++ trong các dự án thực tế vẫn chưa thực sự được đáp ứng.

Khóa luận phát triển thành công một công cụ phân tích mã nguồn cho ngôn ngữ lập trình Rust với mục tiêu khắc phục điểm yếu kể trên. Khóa luận lựa chọn đồ thị thuộc tính mã nguồn làm kiểu biểu diễn trung gian cho mã nguồn Rust. Việc này làm cho công cụ có thể tương thích, mở rộng với nhiều nghiên cứu đã có, đồng thời tái sử dụng được các công cụ khác nhau. Với đặc điểm phân tích tĩnh, đồ thị thuộc tính mã nguồn cho ngôn ngữ Rust có thể áp dụng vào các dự án lớn và tốn ít tài nguyên, chi phí. Công cụ thực hiện xây dựng đồ thị thuộc tính mã nguồn cho Rust ở mức độ mã nguồn thay vì các cấp độ thấp hơn để đảm bảo không bị mất mát thông tin về các cơ chế đảm bảo an toàn bộ nhớ.

Công cụ phát triển dựa trên Joern, kế thừa kiến trúc mở rộng và tái sử dụng. Công cụ được thực nghiệm trên các đoạn mã nguồn có lỗ hổng trong thực tế và ứng dụng vào bài toán học máy phân loại mã nguồn Rust, cho thấy khả năng khai thác to lớn. Mặc dù công cụ đã đạt được kết quả nhất định, các hạn chế vẫn còn tồn tại. Hiện tại, công cụ chỉ có thể phân tích từng tệp mã nguồn Rust riêng lẻ, chưa xử lý được liên kết bằng module giữa các tệp mã nguồn. Ngoài ra, các macro trong Rust vẫn là khó khăn lớn do tính chất không được xử lý trước khi xây dựng cây AST. Nhìn chung, bên cạnh các hạn chế liệt kê ở trên, phương pháp đã cho thấy tiềm năng khi áp dụng phân tích mã nguồn Rust thực tế. Phương pháp giải quyết được hạn chế gặp phải của các nghiên cứu trước đó.

Tài liệu tham khảo

[1] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 84–99, 2021.

[2] Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.

[3] cppreference. RAII - cppreference.com — en.cppreference.com. `https://en.cppreference.com/w/cpp/language/raii`. [Accessed 18-11-2024].

[4] Rust Project Developers. About RustSec; RustSec Advisory Database — rustsec.org. `https://rustsec.org/`. [Accessed 24-11-2024].

[5] 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.

[6] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54, 2013.

[7] Andy Gill, Neil Sculthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton. The remote monad design pattern. *ACM SIGPLAN Notices*, 50(12):59–70, 2015.

[8] githubtop100rust. Github-Ranking/Top100/Rust.md at master · EvanLi/Github-Ranking — github.com. `https://github.com/EvanLi/Github-Ranking/blob/master/Top100/Rust.md`. [Accessed 03-12-2024].

[9] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[10] O JE and T CT. Why scientists are turning to rust. *Nature*, 588:185, 2020.

[11] joern. Joern - The Bug Hunter's Workbench — joern.io. https://joern.io/. [Accessed 19-11-2024].

[12] joern. Joern Query Database | Joern Query Database — queries.joern.io. https://queries.joern.io/. [Accessed 23-11-2024].

[13] joernCPG. Code Property Graph Specification Website | Code Property Graph Specification Website — cpg.joern.io. https://cpg.joern.io. [Accessed 24-11-2024].

[14] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for rust. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

[15] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[16] kernel. Rust 2014; The Linux Kernel documentation — kernel.org. https://www.kernel.org/doc/html/next/rust/index.html. [Accessed 22-11-2024].

[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[18] Alexander Küchler and Christian Banse. Representing llvm-ir in a code property graph. In *International Conference on Information Security*, pages 360–380. Springer, 2022.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[20] Michael P LaValley. Logistic regression. *Circulation*, 117(18):2395–2399, 2008.

[21] maybeMonad. Haskell/Understanding monads/Maybe - Wikibooks, open books for an open world — en.wikibooks.org. https://en.wikibooks.org/wiki/Haskell/Understanding_monads/Maybe. [Accessed 04-12-2024].

[22] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, volume 2324, pages 141–147, 2013.

[23] miri. GitHub - rust-lang/miri: An interpreter for Rust's mid-level intermediate representation — github.com. https://github.com/rust-lang/miri. [Accessed 22-11-2024].

[24] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. Yuga: Automatically detecting lifetime annotation bugs in the rust language. *IEEE Transactions on Software Engineering*, 2024.

[25] Federico Poli, Xavier Denis, Peter Müller, and Alexander J Summers. Reasoning about interior mutability in rust using library-defined capabilities. *arXiv preprint arXiv:2405.08372*, 2024.

[26] ProceduralMacros. Procedural Macros - The Rust Reference — doc.rust-lang.org. https://doc.rust-lang.org/reference/procedural-macros.html. [Accessed 19-11-2024].

[27] rustBorrowChecker. Borrowing - Rust By Example — doc.rust-lang.org. https://doc.rust-lang.org/beta/rust-by-example/scope/borrow.html. [Accessed 04-12-2024].

[28] rustByExample. Introduction - Rust By Example — doc.rust-lang.org. https://doc.rust-lang.org/stable/rust-by-example/. [Accessed 11-12-2024].

[29] rustHir. The HIR (High-level IR) - Rust Compiler Development Guide — rustc-dev-guide.rust-lang.org. https://rustc-dev-guide.rust-lang.org/hir.html. [Accessed 22-11-2024].

[30] rustlangCargoWorkspaces. Cargo Workspaces - The Rust Programming Language — doc.rust-lang.org. https://doc.rust-lang.org/book/ch14-03-cargo-workspaces.html. [Accessed 04-12-2024].

[31] rustlangLifetimeElision. Lifetime elision - The Rust Reference — doc.rust-lang.org. https://doc.rust-lang.org/reference/lifetime-elision.html. [Accessed 05-12-2024].

[32] rustlangManagingGrowing. Managing Growing Projects with Packages, Crates, and Modules - The Rust Programming Language — doc.rust-lang.org. https://doc.rust-lang.org/book/ch07-00-managing-growing-projects-with-packages-crates-and-modules.html. [Accessed 04-12-2024].

[33] rustlangVisibilityPrivacy. Visibility and privacy - The Rust Reference — doc.rust-lang.org. https://doc.rust-lang.org/reference/visibility-and-privacy.html. [Accessed 04-12-2024].

[34] rustMacroExpansion. Macro expansion - Rust Compiler Development Guide — rustc-dev-guide.rust-lang.org. https://rustc-dev-guide.rust-lang.org/macro-expansion.html. [Accessed 04-12-2024].

[35] rustMarco. Macros - The Rust Programming Language — doc.rust-lang.org. https://doc.rust-lang.org/book/ch19-06-macros.html. [Accessed 23-11-2024].

[36] rustMarcoAst. Source Analysis - The Little Book of Rust Macros — veykril.github.io. https://veykril.github.io/tlborm/syntax-extensions/source-analysis.html. [Accessed 04-12-2024].

[37] rustMir. The MIR (Mid-level IR) - Rust Compiler Development Guide — rustc-dev-guide.rust-lang.org. https://rustc-dev-guide.rust-lang.org/mir/index.html. [Accessed 22-11-2024].

[38] rustReference. Introduction - The Rust Reference — doc.rust-lang.org. https://doc.rust-lang.org/reference/. [Accessed 19-11-2024].

[39] Lukas Seidel and Julian Beier. Bringing rust to safety-critical systems in space. *arXiv preprint arXiv:2405.18135*, 2024.

[40] 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.

[41] sourceandroid. Android Rust introduction | Android Open Source Project — source.android.com. https://source.android.com/docs/setup/build/rust/building-rust-modules/overview. [Accessed 22-11-2024].

[42] syn. syn - Rust — docs.rs. https://docs.rs/syn/latest/syn/index.html. [Accessed 19-11-2024].

[43] wasm bindgen. GitHub - rustwasm/wasm-bindgen: Facilitating high-level interactions between Wasm modules and JavaScript — github.com. https://github.com/rustwasm/wasm-bindgen. [Accessed 24-11-2024].

[44] EclipseWeb Web. Eclipse CDT (C/C++ Development Tooling) — projects.eclipse.org. https://projects.eclipse.org/projects/tools.cdt. [Accessed 23-11-2024].

[45] Konrad Weiss and Christian Banse. A language-independent analysis platform for source code. *arXiv preprint arXiv:2203.08424*, 2022.

[46] whitehouse. Press Release: Future Software Should Be Memory Safe | ONCD | The White House — whitehouse.gov. https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/. [Accessed 22-11-2024].

[47] 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.

[48] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th annual computer security applications conference*, pages 359–368, 2012.

[49] 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.

[50] 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.

[51] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. A closer look at the security risks in the rust ecosystem. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–30, 2023.

[52] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.