

# REDUCING BARRIERS TO MEMORY SAFE CODE

by

DIANE BEIGHTOL STEPHENS

(Under the Direction of Dr. Mustakimur Rahman Khandaker)

## ABSTRACT

Secure software begins with safe memory management as over 65% of software vulnerabilities in modern code bases are the result of memory safety problems. Effective memory management is crucial to preventing vulnerabilities and ensuring robustness. Rust is a promising memory safe language, rapidly gaining traction as a replacement for C and C++ in systems software.

Despite Rust’s strengths, complete safety cannot be guaranteed, primarily due to the incorporation of `unsafe Rust`, which allows code to bypass the strict compile time checks. Recent research efforts have focused on statically identifying memory and thread safety bugs in Rust code. This work examines the scope and practicality of five analysis tools: three static analysis tools and two fuzzers. We analyzed the vulnerabilities reported from each static analysis tool. Using this information, we attempted to identify paths to the reported vulnerabilities to determine if we could generate exploits, confirming the presence of actual security-critical bugs. We present the challenges of finding exploits, recommend a system for prioritizing vulnerabilities, and propose a systematic tool for exploit generation using a fuzzer and a fuzzer assistant.

Rust’s novel approach to memory safety presents challenges. A 2023 Stack Overflow survey indicates that Rust’s complexity and steep learning curve are significant barriers to adoption. The Rust programming paradigm is focused on the principles of ownership and borrowing. These concepts effectively

achieve memory and thread safety but are enforced at compile-time, enabling runtime performance comparable to C but posing significant implementation challenges for programmers.

To reduce the barriers to Rust adoption, this work presents RustLIVE, an innovative visualization tool that clarifies Rust's most difficult concepts: ownership and borrowing. RustLIVE is an extension for VSCode, the IDE of choice for over 60% of Rust developers. Seamlessly integrated with the Rust compiler and its borrow checker, RustLIVE requires no code annotations. Instead, it extracts necessary information directly from the compiler to provide color-coded visual timelines that illustrate the ownership of memory resources and the liveness of borrows. This intuitive visualization is particularly valuable for understanding non-lexical lifetimes, which lack visual cues in the source code. RustLIVE is an independent learning tool that represents a step toward flattening Rust's learning curve and reducing barriers to adoption of Rust.

In summary, this dissertation presents two works that promote writing memory safe code in Rust, an analysis of Rust vulnerability detection tools to advance methods for generating exploits, and a novel visualization tool to depict Rust memory safety concepts.

INDEX WORDS: [Memory Safety, Secure Programming, Rust Programming Language]

REDUCING BARRIERS TO MEMORY SAFE CODE

by

DIANE BEIGHTOL STEPHENS

B.A., University of Tennessee, Knoxville, 1987

M.S., University of Alabama, Huntsville, 1992

A Dissertation Submitted to the Graduate Faculty of the  
University of Georgia in Partial Fulfillment of the Requirements for the Degree.

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2024

©2024

Diane Beightol Stephens

All Rights Reserved

REDUCING BARRIERS TO MEMORY SAFE CODE

by

DIANE BEIGHTOL STEPHENS

Major Professor: Dr. Mustakimur Rahman Khandaker

Committee: Dr. Lakshmish Ramaswamy

Dr. Roberto Perdisci

Dr. Kyu Hyung Lee

Electronic Version Approved:

Ron Walcott

Dean of the Graduate School

The University of Georgia

December 2024

# DEDICATION

This dissertation is dedicated to my family, for their unwavering support and encouragement. My quiver is full and I am blessed. I am especially indebted to Speedy, who first asked me to be his lab partner in CS318o Logic Design of Digital Systems. Thirty-something years later he still makes me laugh, reminds me to stop and smell the roses, supports me in all my endeavors, and remains my best friend. Thanks Babe.

## ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my major advisor Dr. Mustakimur Rahman Khandaker, for his knowledge and invaluable insight. I am also deeply indebted to the members of my committee: Dr. Lakshmish Ramaswamy for his thoughtful suggestions and continuous encouragement, Dr. Roberto Perdisci whose expertise and commitment to excellent research enriched the depth of this work, and Dr. Kyu Hyung Lee who generously gave his time and expertise to guide this work to completion.

Special thanks to Dr. Gagan Agrawal and Dr. Thiab Taha whose leadership and support made teaching and research possible.

This research would not have been possible without the financial support of the University System of Georgia's Tuition Assistance Program (TAP) and the University of Georgia's Faculty Development in Georgia assistantship.

Thank you.

# CONTENTS

<b>Acknowledgments</b>	<b>v</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Tables</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Thesis Statement . . . . .	4
1.2 Barriers to Memory Safety in Rust . . . . .	5
1.3 Modern Approaches to Systems Programming: Rust and C++ . . . . .	7
1.4 Contributions to Computer Science Education . . . . .	12
<b>2 Literature Review</b>	<b>13</b>
2.1 Visualization toward Usability . . . . .	14
2.2 Rust Compiler Development . . . . .	15
2.3 Rust Vulnerability Detection . . . . .	15
<b>3 Understanding Vulnerabilities in Rust Applications</b>	<b>21</b>
3.1 Background . . . . .	24
3.2 State-of-the-Art Bug Detector and Exploit Generator . . . . .	28
3.3 Research Questions . . . . .	31
3.4 Evaluation . . . . .	33



3.5	Future Work . . . . .	40
3.6	Conclusion . . . . .	41
<b>4</b>	<b>Reducing the Learning Barriers of Rust Through Visualization</b>	<b>43</b>
4.1	Background and Motivation . . . . .	44
4.2	Project Design and Compiler Integration . . . . .	46
4.3	Implementation . . . . .	56
4.4	Evaluation . . . . .	71
4.5	Discussion and Future Work on RustLIVE . . . . .	77
<b>5</b>	<b>Conclusion and Future Work</b>	<b>82</b>
	<b>Appendices</b>	<b>84</b>
	<b>A</b>	<b>84</b>
	<b>Bibliography</b>	<b>88</b>

# LIST OF FIGURES

1.1	Trait summarizable allows sharing the behavior of method summary. . . . .	9
1.2	Types may use the default behavior defined in the Summary trait or override with custom behavior. . . . .	10
1.3	Generic types in Point allow coordinates of same or different types, enabling code reuse.	11
2.1	Comparison chart of static and dynamic vulnerability detection tools. . . . .	20
3.1	An integer overflow report from Speedy Web Compiler (SWC) project reported by the authors. . . . .	26
3.2	A panic will cause an undefined behavior in unsafe Rust . . . . .	27
3.3	Rust vulnerability detectors categorized by their analysis approach. . . . .	28
3.4	Research questions and their connections . . . . .	31
3.5	Statistics on selected projects including dependent crates. . . . .	35
3.6	An incorrect send/sync trait bound for a generic type parameter in anyhow-1.0.57. However, it is a false positive report because the only use of boxed() is using a type that guarantees thread-safety. . . . .	37
3.7	Demonstrate the required symbolization to exploit a vulnerability reported in slide_deque crate. . . . .	42

4.1	Rust program that will not compile due to the enforcement of the ownership principle. The ownership of grades is transferred to <code>update_grades</code> yet ownership of <code>n_grade</code> remains in <code>main</code> . . . . .	44
4.2	Phases and intermediate representations of the Rust compile process . . . . .	47
4.3	Example Rust program and associated MIR . . . . .	49
4.4	Ownership Transfer and Drop . . . . .	51
4.5	Borrow Checker Example Program . . . . .	52
4.6	Borrow Checker Example with Region Variables . . . . .	54
4.7	Ownership transfer and drop depicted in RustLIVE . . . . .	55
4.8	RustLIVE depiction of non-lexical lifetime (NLL) . . . . .	56
4.9	Implementation of RustLIVE . . . . .	57
4.10	RustLIVE webview panel view: active code and timeline graphic side by side . . . . .	58
4.11	Representation of each Local in the webview frontend . . . . .	59
4.12	Representation of the graph as in the webview panel of RustLIVE . . . . .	60
4.13	Representation of the starting point of each local . . . . .	61
4.14	Representation of each liveness path . . . . .	61
4.15	RustLIVE webview panel. Owned locals <code>x</code> and <code>y</code> are solid points. Borrows <code>r</code> and <code>s</code> are hollow and originate at the referent. . . . .	62
4.16	Return from <code>get_body_with_borrowck_facts</code> includes the MIR intermediate representation in <code>body</code> as well as the input and output facts computed by the borrow checker . . . . .	65
4.17	Data to obtain for each Local in <code>body</code> . . . . .	66
4.18	MIR Body representation - one per function . . . . .	67
4.19	MIR Basic block representation . . . . .	67
4.20	The <code>VarDebugInfo</code> member of <code>Body</code> provides useful information for RustLIVE . . . . .	67
4.21	<code>region_map</code> is created to map each reference (region) to its live MIR locations . . . . .	69
4.22	<code>place_regions_live_ranges</code> maps each Local and Region to a set of live lines. . . . .	70

4.23	BorrowSet contains information for identifying the referent of references. . . . .	70
4.24	BorrowData of location_map contains borrowed_place . . . . .	70
4.25	User Responses to Question 2 - Effectiveness to understanding Ownership. 73% rank RustLIVE 8 or higher for usefulness to understanding ownership. . . . .	72
4.26	User responses to Question 3 - Effectiveness to understanding lifetime of borrows. . . .	73
4.27	User responses to Question 4 - Effectiveness to understanding Rust memory safety. 49 of 190 evaluators rated RustLIVE an 8, 48 rated 9 and 53 gave RustLIVE a 10 for under- standing Rust memory safety. . . . .	73
4.28	User responses to Question 5 - Effectiveness at developing Rust programming skills. 69 of 190 responses ranked RustLIVE a 10, the highest possible score, for skill development. . . .	74
4.29	Frequency of responses to Question 1: Rate your understanding of memory safety in Rust	74
4.30	RustLIVE evaluation results overall and by skill level. . . . .	76

# LIST OF TABLES

3.1	Selected trophy cases from MIRChecker, Rudra, FFIChecker, afl.rs, and cargo-fuzz . . .	34
3.2	Vulnerability reports from selected projects and their dependent crates. Note, some of the project reports are duplicates from crates. . . . .	36
3.3	Vulnerability report distribution over crates and exploits. . . . .	38
4.1	Example Rust Programs for Assessment (Part 1). . . . .	79
4.2	Example Rust Programs for Assessment (Part 2) . . . . .	80
4.3	Example Rust Programs for Assessment (Part 3) . . . . .	81

# CHAPTER I

## INTRODUCTION

Memory unsafety has been the Achilles heel of the software industry for over a decade. Memory safety errors consistently represent more than 65% of vulnerabilities in code bases. Research by Microsoft's Security Response Center and Google's Chromium browser project both confirm that memory safety issues - invalid memory access - account for over 65% of their serious security bugs. [75, 10]

Secure software begins with safe memory management. How a program manages memory is key to preventing vulnerabilities and ensuring robustness [48]. Memory management involves how and when memory is allocated and deallocated as well as who owns it and who has access to it. Reading from memory that has already been freed or writing beyond the bounds of allocated memory are memory management errors. Such errors result in program crashes and open doors for serious attacks [70].

A memory-safe language incorporates methods for preventing memory errors, protecting against vulnerabilities such as memory leaks, dangling pointers, and unchecked bounds. For instance, if a program deletes a list but later tries to read from it, what happens? A memory unsafe language may still allow access to the now-freed memory. In contrast, memory safe languages prevent this type of access, avoiding potential security breaches.

Memory safety is handled differently in different programming languages:

- **C** and **C++** allow manual memory management. The programmer is responsible for memory safety.

- **Java, Python, Go**, and most other ‘memory safe’ languages employ a garbage collector to manage memory automatically, abstracting away the allocation and freeing of memory.
- **Rust** enforces memory safety at compile time. Code that passes the Rust compiler checks is generally memory safe.

Most memory safe languages employ a garbage collector, which runs periodically to clean up memory allocations. Garbage collectors identify and reclaim memory that is no longer in use or no longer reachable. Garbage collectors can be effective at preventing memory errors such as memory leaks and use-after-free vulnerabilities, but they come at a cost; programs use more memory and run slower. So, while languages such as Java and Python are well suited for many applications, the runtime performance makes them less suitable for systems software and embedded systems development where performance and efficient memory management are critical.

Government leaders are promoting the use of memory safe language. In a 2023 report, NSA recommends using a memory safe language “when possible” instead of C and C++. “The highest leveraged method manufacturers can use to reduce memory safety vulnerabilities is to secure one of the building blocks of cyberspace: the programming language.” Using memory safe programming languages can eliminate most memory safety errors. “The overarching software community across the private sector, academia, and government have begun initiatives to drive the culture of software development towards utilizing memory safe languages.” [47]

Furthermore, a recent gathering sponsored by the United States government, the National Science Foundation (NSF) and the National Institute for Standards and Technology (NIST) brought together leaders from academia, industry, and government to identify ways to improve the security of the open-source software ecosystem . “We need to incentivize developers to write secure code and make security a part of professional training and university course curricula,” adding that “the Rust programming language, despite its initial learning curve, is particularly well-suited for safe system development. The transition to memory safe languages is several years away, but we can speed it up by increasing investments in ease-of-use, interoperability, and automation.” [32]

Eliminating memory vulnerabilities before they ever happen is the approach of the Rust language. Rust was designed from the ground up for safety. The author, Graydon Hoare, a developer for the Mozilla browser, set out to create a language that does not allow memory errors. Rust is a memory-safe language built for system software development [33]. It has emerged as a contender to replace C/C++ for system applications such as operating systems (Redox[76], Theseus[7], Tock[37]), web engines (Servo[63], Deno[15]), blockchains (lighthouse[41], OpenEthereum[50], electrs[18]), and the Firefox browser [22].

What is the difference between Rust and other memory safe languages? Rust achieves memory safety without compromising performance; memory checks are performed at compile time, so there is essentially zero cost to runtime performance. The Rust compiler enforces its memory safety guarantees.

The core of the Rust safety model is its principles of *ownership* and *lifetime* of references (*borrow*s).

The three tenants of Rust ownership are:

- Each value has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

In Rust, every resource has a single owner at a time. Ownership can be transferred, but the resource always has exactly one owner until it is no longer needed and then the resource is freed or *dropped*. This ownership model prevents data races as no two threads can simultaneously own the same resource. All resources are owned in Rust and dropped when no longer in use.

Rust allows references to a value, called *borrow*s, during the lifetime of the value. Borrowes allow for the sharing of resources without transferring ownership, adhering to the following rules:

- Multiple immutable borrows, aliases, are allowed.
- Only one mutable borrow at any time is allowed.
- References cannot outlive the owner variable.



## 1.1 Thesis Statement

Rust offers a promising alternative for systems software by combining both safety and speed. However, its adoption faces two significant barriers: vulnerabilities in Rust code and a steep learning curve for new developers.

Despite the strict safety policies, vulnerabilities exist in Rust code bases, particularly from the use of `unsafe` code. Existing analysis tools detect vulnerable patterns statically, but are there more efficient methods for vulnerability analysis and targeted exploitation. This research evaluates current vulnerability detection methods to determine a systematic prioritization for vulnerabilities and identifies the challenges in generating exploits of statically identified vulnerabilities. The work detailed in this dissertation demonstrates that static analysis methods, combined with localized symbolic analysis and fuzzing is a practical approach for targeted and efficient exploit generation.

In addition, Rust's concepts of ownership and lifetimes—central to its memory safety model—present a steep learning curve and a barrier to the adoption of Rust. Visualization is a proven aid for understanding complex concepts. Can visualization of ownership and lifetime of borrows reduce the learning curve of Rust? Furthermore, can the enforcement data be obtained directly from the compiler so the visualization can be generated automatically, functioning as an independent learning tool? Also, since programming in Rust requires understanding memory safety principles, is learning to program in Rust a viable approach to teach safe coding practices in Computer Science education? This dissertation presents RustLIVE, a tool that automatically gathers ownership and lifetime data from the Rust compiler and produces an intuitive visual timeline of these principles. RustLIVE enhances comprehension of memory safety principles, helping programmers develop safe coding skills and making learning Rust in undergraduate Computer Science courses viable.

## **1.2 Barriers to Memory Safety in Rust**

The research detailed in Chapter 3, Understanding the Challenges in Detecting Vulnerabilities of Rust Applications, assesses current methods of vulnerability detection and suggests methods for more targeted and efficient detection and exploitation. This research provided invaluable experience for examining how the Rust language is currently used, its memory safety, the use of `unsafe` code, and the barriers to adoption of Rust. Rust requires a different model of thinking; it forces the programmer to think about memory management. This is a major shift in program methodology which has recently trended toward non-procedural, human-friendly abstractions.

The study of Rust analysis tools revealed many of the hindrances to writing memory safe code in Rust. One study cites, “The biggest drawback of Rust is its learning curve. Specifically, the borrow checker and programming paradigms are the hardest to learn.” [23] This research inspired the second work, a tool to assist in the development of safe code in Rust. The tool, named RustLIVE and described in Chapter 4, provides a visualization of Rust memory safety principles.

Rust’s novel approach to memory safety guarantees that code written in safe Rust is free from memory safety issues and data races, while also requiring programmers to understand the principles of memory safety enforced by the language. This dissertation details a tool to help programmers develop safe coding practices and addresses issues in detecting vulnerabilities in Rust code. By learning how Rust achieves memory safety and recognizing potential vulnerabilities, programmers are better equipped to write memory-safe code not only in Rust but also in other languages.

### **1.2.1 Understanding the Challenges in Detecting Vulnerabilities of Rust Applications**

Rust’s strict compile time policies prohibit programmers from operations like dereferencing a raw pointer, inline assembly, making a C system call, or performing a bitwise copy. To allow such constructs for which the Rust compiler cannot guarantee safety, Rust has `unsafe` Rust. Code marked with `unsafe` in Rust

bypasses many of the safety checks of the Rust compiler. Studies [84, 52] show that `unsafe` Rust code is the primary culprit for Rust vulnerabilities. Recent research has focused on static analysis tools to detect vulnerabilities in existing Rust code bases.

The work detailed in *Understanding Vulnerabilities in Rust Applications*, examines five tools: three static analysis tools for detecting vulnerabilities in Rust code and two dynamic analysis tools - fuzzers. The static analysis tools are run on three selected Rust projects. With the vulnerability reports generated from static analysis of the three projects, reports are analyzed and target harness code is developed to attempt exploit of the vulnerabilities with fuzzing. Additionally, RULF to generate target harness code and symbolic engine KLEE[8] are employed. The obstacles to reaching and exploiting the vulnerable code are documented, and a priority categorization method for detected vulnerabilities is suggested. With the vast number of vulnerabilities reported and many of them in dependent crates, sound methodology for prioritizing resources is valuable. Finally, methods to automatically produce exploits with the set of tools is discussed.

### **1.2.2 Reducing the Learning Barriers of Rust through Visualization**

Rust introduces a novel programming paradigm to enforce memory safety. The core safety principles of Rust - ownership and borrowing - are conceptually clear and straightforward. Yet Rust is challenging due to its novel approach to memory safety that diverges significantly from approaches in other programming languages. A 2023 Stack Overflow survey confirms that the complexity of Rust is a major concern and the steep learning curve is the biggest barrier to adoption. [74] The principles of ownership and borrowing effectively achieve memory and thread safety but are enforced at compile-time, enabling runtime performance comparable to C but posing significant implementation challenges for programmers.

To support developers and students, the visualization of Rust's memory model has been discussed in Rust usability research and user forums. However, the requirement for code annotations in existing methods presents a barrier to practical use.

In *Reducing the Learning Barriers of Rust Through Visualization* of Chapter 4, we present RustLIVE, an extension for the popular IDE, VS Code. Seamlessly integrated with the Rust compiler and its borrow checker, RustLIVE requires no annotations from the user. Instead, it extracts necessary information directly from the compiler to provide visual timelines that illustrate the ownership of memory resources and the lifetimes of borrows.

## **1.3 Modern Approaches to Systems Programming: Rust and C++**

Despite its drawbacks, Rust offers memory safety without compromising speed, making it a viable language for systems software, a category of software that typically interacts with hardware, operating system or network. System software often requires high performance, low-level access and control over resources. The choice of programming language significantly impacts the development process and potentially the performance, safety and maintainability of the code base. Rust and C++ are both considered viable programming languages for most systems projects as has been noted earlier. Understanding their differences is important to making informed decisions about their use.

### **1.3.1 C/C++**

C/C++ is well established as the language of choice for operating systems, device drivers, file systems and embedded systems software. C++ provides the tools and low-level features of C which make for highly efficient code yet also comes with a mature ecosystem and a rich standard library. An object-oriented programming (OOP) language, C++ provides features such as classes, encapsulation, inheritance, and polymorphism. Existing code bases written primarily in C/C++ include Linux, Windows and MacOS operating systems, numerous device drivers, file systems, the GNU compilers, and the Chrome browser.

C was designed by Dennis Ritchie at Bell Laboratories in the early 1970's to write the Unix operating system. It was designed to allow direct memory management and produce really efficient code - a hallmark of systems software. Several years later, C++ was developed by Bjarne Stroustrup, also at Bell Laboratories,

to enhance C. Object-oriented programming was added in C++, introducing classes, objects, inheritance and polymorphism - tools for modeling complex systems and reusing code.

C++ offers improvements over C in memory safety. Smart pointers which employ the Resource Acquisition Is Initialization (RAII) concept that binds the life of a resource with the lifetime of the pointer represent a significant improvement over C raw pointers. However, C++ is a superset of C so the direct memory management techniques and raw pointers of C are valid in C++.

The memory safety issues in system software code bases stem from the use of system languages, C/C++ which put the burden on the programmer to perform the needed protections on memory resources. For example, if I create an array to hold 100 elements then later in my program realize that 100 is not enough, I can `malloc()` memory for a new array that holds 200 elements and then copy the data from the old array into the new one. However, if I don't `free()` the memory for the old array that memory remains intact but with no pointer to this memory. This is noted in [70] as a memory leak. The programmer is responsible, and managing memory in C is difficult and a primary reason that memory vulnerabilities exist and continue to be the source of most security vulnerabilities.

C++ is not and cannot be made a memory safe language. C++ contains features which are memory safe but features that are not memory safe cannot be eliminated without breaking existing code. Memory safe code can be written in C++. Memory errors in C/C++ can be detected by careful inspection, testing and analysis tools. The language and the compiler do not enforce safety guarantees. "So really the onus is on the programmers in order to develop code that does not have undefined behaviors and this is not a trivial thing. Better tools is one half of the equation but training programmers to avoid creating vulnerable code is the other half. " [61]

### **1.3.2 Rust in comparison**

Rust was designed as a systems programming language with safety and speed as top priorities. Rust places restrictions on the programs which can be frustrating but the restrictions come with safety guarantees.

```
pub trait Summarizable {
    fn summary(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Figure 1.1: Trait summarizable allows sharing the behavior of method summary.

The Rust safety policies ensure that a programmer cannot accidentally leave a dangling pointer in play, create a data race with multiple accesses to mutable data, or access data after it has been dropped.

While Rust is not object-oriented as traditionally defined, it supports key benefits of object-oriented language. The names may be different but the concepts of objects, encapsulation, and polymorphism have support in Rust.

In OOP, objects package data and methods that operate on the data. In Rust, structs and enums have data and `impl` blocks provide methods to operate on the data. In OOP, encapsulation protects internal state and methods of objects and allows interaction through its public interfaces. Rust promotes principles of encapsulation with modules and visibility modifiers. Rust allows fine-grained control over visibility with the `pub` keyword used to make modules, functions, and methods public, which by default are private.

Inheritance in OOP achieves code reuse and facilitates polymorphism. Rust does not support traditional class-based inheritance where child classes inherit the data and behavior (methods) of the parent class. Instead, Rust uses composition and traits to achieve code reuse and polymorphism.

With Rust composition you can define one struct that contains another struct as a field to facilitate data sharing. To share behavior (methods) among multiple structs, you can use traits. Traits allow you to define shared methods across different structs. By defining a trait, such as the `Summarizable` example in Figure 1.1 the default behavior of `summary` is shared with different types that implement `Summarizable`. [34]

Polymorphism allows for code that can run on different data types. To achieve polymorphism, Rust traits allow different types to share behavior, a similar concept to interfaces in other languages. Each

```

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summarizable for NewsArticle {}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summarizable for Tweet {
    fn summary(&self) -> String {
        format!("{: {}", self.username, self.content)
    }
}

```

Figure 1.2: Types may use the default behavior defined in the Summary trait or override with custom behavior.

type that implements the Summarizable trait may use the default behavior for the summary method or may override the default behavior. In the example depicted in Figure 1.2, both NewsArticle and Tweet implement the Summarizable trait. Polymorphism is achieved as summary exhibits different behavior for different types NewsArticle and Tweet. NewsArticle shares the default behavior and Tweet implements custom behavior.

Developers choose which fields and functionality to reuse which avoids the potential of sharing more code than is necessary as in inheritance.

Rust also achieves polymorphism through generics. In Rust, functions may use generics in the signature where the data types of the parameters and return would go. This saves on code duplication and provides more flexible code. Consider the Point struct in Figure 1.3 which uses generic data types in its

```

struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}

```

Figure 1.3: Generic types in Point allow coordinates of same or different types, enabling code reuse.

signature. This allows more flexible code that includes coordinates of same or different types, while not introducing code duplication.

### 1.3.3 Challenges to Secure Systems Software

Comparing C++ and Rust is more accurately about understanding the tradeoffs rather than picking a winner. In addition to memory safety concepts and object-oriented principles addressed above, both languages have strengths and the costs of moving existing C++ code bases to Rust may prove prohibitive as rewrites are not typically funded. [61]

Both C++ and Rust compile to machine code, making them suitable for performance-critical applications. Rust does not have the extensive standard library available with C++ but concurrency is free from data races in Rust.

Perhaps tooling to better secure existing C++ code is a practical approach for large projects. Some industry development teams are adopting an incremental strategy. Rather than rewrite existing code, project teams such as the Android Linux kernel group are developing new code in Rust.

Rust prioritizes security without sacrificing efficiency. Writing projects in Rust helps reduce costly maintenance issues like memory leaks and data races. While C++ remains widely used in established systems codebases, the choice between Rust and C++ depends on the specific needs and goals of a project.



However, with increased emphasis on memory safe language and growing government pressure, Rust is well-positioned for continued growth.

## **1.4 Contributions to Computer Science Education**

The struggle to teach secure coding practices is realized in Computer Science education and remains a challenge. Recent research confirms that Computer Science students lack the key fundamental skills to write secure programs[36]. “The first and foremost strategy for reducing security related coding flaws is to educate developers how to avoid creating vulnerable code.” [61]

The Rust compiler may be likened to a strict teacher that enforces safe memory management - a preemptive approach for teaching memory safety. Learning to program in Rust provides practice in writing memory safe code. But adding a new programming course is often not practical and changing the language of existing programming courses to Rust presents challenges as well.

RustLIVE, an innovative, independent learning tool, makes integrating Rust into an educational curriculum more practical. Rust can be added as part of an existing course, such as systems programming, providing students with hands-on experience in a language that enforces safe and robust systems development. Feedback confirms that RustLIVE will facilitate students learning safe coding principles.

# CHAPTER 2

## LITERATURE REVIEW

This research began with an curious graduate student intrigued by the possibility of a new systems language that achieved safety without compromising speed. Several early research efforts contributed to understanding and appreciating the benefits and drawbacks of the Rust language. Key works are highlighted here.

“Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs” [52] examined the safety issues in Rust programs with an empirical study of how `unsafe` is used in Rust code bases. This work was pivotal in understanding what mistakes programmers make and how the use of `unsafe` undermined the safety of Rust projects and lead to memory and thread safety issues. The study concluded that all memory-safety bugs involved `unsafe` code and that the scope of lifetimes in Rust were difficult and led to memory-safety issues. The study contributed that misunderstanding ownership and lifetime rules were the primary cause for memory bugs and suggested that a visualization of these rules would benefit Rust programmers in writing safe code.

The authors of “Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study” [23] conducted a survey of Rust developers to get their personal experience with using Rust. The work cited the documentation and improvement of secure programming skills as a benefit of Rust. The steep learning curve and limited library support were reported drawbacks of Rust. The survey revealed that `unsafe` blocks are common and few are reviewed for safety. A confirmation to conclusions from

other works was that Rust was hard to learn, presenting a steep learning curve with the borrow checker singled out as the most difficult aspect. However, the confidence of robust code and lack of debugging necessary once a program passed the compiler were benefits. A final key takeaway from the Rust developers interviewed was that learning and programming in Rust improves safe development in other languages.

## **2.1 Visualization toward Usability**

While the idea of a visualization tool for Rust safety concepts began with the works cited above, the potential benefit of this approach was supported in the following works which addressed the usability of Rust and visualization as a tool for understanding difficult concepts.

In [11] Crichton suggests that Rust’s ownership is difficult for even experienced developers and considers the complexity of Rust to be a major concern. Authors of [20] add that Rust programmers must learn new patterns of structuring code and need to abandon common familiar patterns. Both usability works suggest that visualization of Rust’s lifetimes could address the usability issues of the borrow checker, but acknowledge the difficulty of visualizing the large amount of information in a “succinct, non-intrusive, yet informative” manner.

### **2.1.1 Use of Visualization in Computer Science Education**

The use of visualization as a tool for understanding difficult concepts in Computer Science education is well documented in the works by Sorva. [65, 66, 67] Scholars in Computer Science education advocate for visualization as an appealing educational tool with considerable support, particularly to support beginner programmers, citing that a “visualization can make focal and explicit processes that are hidden and implicit.” [68] Visualization is a pedagogical technique that helps develop mental models of systems and processes. A particularly useful side affect of visualization is that it provides personal feedback to the user or student, facilitating independent learning.

### **2.1.2 Rust Visualization Tools**

RustViz [43] presents a tool for building visualizations of ownership and borrowing but does not generate these visualizations automatically. Instead, RustViz requires a Rust expert to annotate code to be visualized. This tool also provides tooltips with information gathered from the annotations.

VRLifetime [87] uses visualization in an IDE to help detect two common kinds of concurrency bugs: double lock and locks in conflicting order. Other works have tackled the issue of explaining compiler errors such as [6, 17] which depicts a directed graph representation of the lifetime constraints that lead to a lifetime error in Rust. Rust life follows the lifetime constraints violated in order to provide an informative explanation or a graphical explanation of the error. No evaluations on users were conducted but, worth noting, the error messages provided by the Rust compiler for errors depicted by the borrow checker have greatly improved.

## **2.2 Rust Compiler Development**

The success of RustLIVE hinges on the integration with the Rust compiler. By obtaining liveness and ownership calculated by the compiler ensures that although the compiler may change how liveness is determined, RustLIVE graphical depictions and annotations will match the computations of the compiler. While the Rust compiler remains a work in progress; and the nightly version which RustLIVE interacts with is particularly volatile, the Rust compiler team puts considerable effort in the documentation. The Rust Compiler Development Guide was the central resource for development of RustLIVE. [73]

## **2.3 Rust Vulnerability Detection**

*Understanding the Challenges in Detecting Vulnerabilities of Rust Applications* evaluates five popular Rust vulnerability detectors. These include three static analysis tools: Rudra [4], MirChecker [39], and FFIChecker [38], and two dynamic analysis tools: afl.rs and cargo-fuzz [9].

### 2.3.1 Static Analysis Tools

A significant contribution of the static analysis techniques is in identifying the patterns that Rust programmers make that lead to safety violations. While the reports from the static analysis tools are lengthy and contain many false positives, they provide useful information for generating effective harness code for dynamic analysis tools.

#### Rudra

Rudra found 263 bugs by identifying three bug patterns in the use of unsafe code: panic safety, higher-order safety invariant and propagating send/sync traits in generic types.

The panic safety bug pattern depends on an external function call that triggers a panic error; panic in Rust is for unrecoverable errors. When a panic happens, Rust unwinds the call stack invoking destructors for all the stack-allocated objects and transfers control flow to a panic handler. This is normally an orderly process of unwinding and releasing the associated memory. However, a panic triggered in unsafe code starts the unwinding process with objects potentially in an inconsistent state. For example, it is common in unsafe code to temporarily bypass the Rust ownership system with extended lifetimes or uninitialized variables and then fix them before returning. However, if a panic is triggered at this point in unsafe code, there are no guarantees that the memory can be safely released and a double free could result.

The higher-order safety invariant pattern occurs when unsafe code assumes certain properties of Rust generics that cannot be guaranteed to be true. A Rust function should execute safely for all safe inputs but sometimes the Rust type system only enforces the correctness of type signature. For example, an implementation could pass an uninitialized buffer to a caller-provided read implementation which is expected to read data from one source and write to a provided buffer. A higher order safety bug occurs if the callee reads the buffer instead. This leads to undefined behavior as the buffer may be uninitialized.

A generic type in Rust has a send/sync variance bug if it specifies an incorrect bound on inner types when implementing send/sync. Thread safety in Rust is determined by the send and sync traits. The send/sync bug pattern identifies thread safety vulnerabilities where generic types incorrectly bind their

inner type in implementing send and sync traits. The send trait indicates that a type can be sent to other threads and the sync trait indicates that a type can be referenced concurrently by multiple threads. Developers must manually implement send and sync on synchronization primitives such as locks. A developer who does not implement send/sync on a new API that is not thread-safe introduces a thread-safety bug.

Rudra is scalable, scanning the entire Rust package registry, crates.io, with over forty thousand packages in just over 6.5 hours. Rudra identified 263 previously unreported memory safety bugs. The bugs found included two in the standard library and one in the Rust compiler.

## **MIRChecker**

MIRChecker specifically tracks numerical values to identify integer-related bugs such as integer overflow and division by zero. MIRChecker also tracks symbolic values to follow memory management of those values. The symbolic values represent potential aliases of heap memory created in unsafe and then auto dropped in Rust. This symbolic information is useful for detecting potential use after free vulnerabilities.

MIRChecker uses the assertions inserted by the Rust compiler to check dynamic safety conditions such as buffer overflows and implements the checks in a static analyzer. Using the assertions generated by the compiler takes advantage of bug patterns unique to Rust and eliminates the need for manual annotations in the code. MIRchecker is implemented as an additional pass of the Rust compiler and operates on the Mid-level Intermediate Representation (MIR) generated by the compiler.

Strengths of MIRChecker are in detecting integer-related bugs and providing detailed diagnostic information. A drawback, however, is that it requires a harness test case similar to what is required by fuzzers to generate the compiler assertions for analyzing a crate. While the harness can target a specific vulnerability, it limits the code coverage. Another significant drawback of MIRChecker is the high false positives generated - 79.2%. Thus, the results generated by MIRChecker require significant manual analysis.

## **FFIChecker**

FFIChecker detects memory management issues across the Rust/C foreign function interface (FFI). FFIChecker targets the dynamic memory allocations of the C functions for potential corruption of the shared heap memory space. It uses a taint analysis approach to keep track of the states of the dynamic heap memory allocations, determines whether each allocation is borrowed or moved, if any heap memory is passed across the FFI boundary, and whether it is freed in the C code. The tool detects potential memory corruption issues that stem from situations such as C code deallocating memory allocated in Rust code or crashes in C code that could result in a memory leak of allocated resources. FFIChecker works on the LLVM intermediate representation for both the Rust and C code. Noted in the FFIChecker report is that 72% of packages on crates.io utilize a FFI. In total, FFIChecker evaluated 987 packages from the official package registry, crates.io, and detected 34 bugs in 12 packages.

Strengths of FFIChecker are its detection of memory safety issues caused by the integration with C functions. A drawback of FFIChecker is that it relies on a test harness so the code coverage is limited by the harness and potential bugs are missed.

### **2.3.2 Dynamic Analysis Tools**

Dynamic analysis tools provide the most precise bug-finding techniques because they detect memory and thread safety issues during execution of Rust applications. However, dynamic approaches rely on proper inputs to trigger bugs. Therefore, tools to generate proper inputs are key to meaningful dynamic analysis. Dynamic analysis produces false positives on Rust crates due to assertions set by the programmer.

#### **afl.rs**

afl.rs is based on the existing, popular American Fuzzy Lop ++ fuzzer. AFL++ instruments LLVM IR code for fuzzing which the Rust compiler generates for Rust code so afl.rs can generate similar instrumentation on Rust code. afl.rs provides a `fuzz!` macro to read bytes from `stdin` and provide the fuzz closure. The fuzz closure is a test harness directed to a specific area of a Rust crate. The `fuzz!` macro also

ensures that if a panic error occurs, the process aborts instead of attempting an orderly panic unwinding process that would bypass the crash reporting system. A strength of afl.rs is its use of the existing AFL fuzzer. An additional feature of afl.rs is its graphical user interface which allows you to monitor paths being generated, a good indication of a successful run. However, finding effective target harnesses that generate multiple paths is difficult .

### **cargo-fuzz**

Cargo-fuzz utilizes the Rust package manager, Cargo, to invoke an existing fuzzer, libfuzzer. LibFuzzer uses LLVM's sanitizer coverage instrumentation to gather code coverage information. Cargo-fuzz also uses a macro, fuzz\_target! for writing the test harness. LibFuzzer generates pseudo-random input to the harness until an error condition is triggered. A strength of cargo-fuzz is its ability to generate integer errors with proper harness code and appropriate seeds.

Because afl.rs and cargo-fuzz use different underlying fuzzers and utilize different harness techniques, the results are usually different. Both generate crashes with appropriate test harnesses but are dependent on effective harness code to steer the code coverage and generate multiple paths.



Tool	Analysis Type	# bugs	publication date	location of study	scalability	target
Rudra	static	264 new bug reports: 98 RustSec advisories, 74 CVEs	October, 2021	Georgia Institute of Technology	scans entire package registry (43k packages) in 6.5 hours	3 patterns: panic-safety, high-order invariant, send/sync,
MIRChecker	static	17 runtime panics and 16 memory-safety issues	November 2021	Chinese Univ. Hong Kong	~1000 crates on crates.io and GitHub limitation - requires harness code	integer-related bugs symbolic values in MIR
FFIChecker	static	34 bugs found in 12 packages	September 2022	Chinese Univ. Hong Kong	limitation - requires harness code 987 packages on crates.io	integration with C functions heap memory allocations
afl.rs	dynamic	25 bugs found in 3 selected projects	initial commit 2015	open source, multiple contributors	resource utilization limits concurrent runs - each run specific to project	determined by target harness code
cargo-fuzz	dynamic	23 bugs found in 3 selected projects	initial commit 2016	open source, multiple contributors	resource utilization limits concurrent runs - each run specific to project	determined by target harness code

Figure 2.1: Comparison chart of static and dynamic vulnerability detection tools.

# CHAPTER 3

## UNDERSTANDING VULNERABILITIES IN RUST APPLICATIONS

Memory and thread safety bugs remain a significant challenge for system application development. Every year more than 10,000 vulnerabilities are reported in critical system applications. For example, in the Linux kernel alone, in 2023, there are 192 official vulnerabilities reported, among which 41 are capable of privilege escalation[13]. Vulnerability detection is one approach for identifying potential bugs before they are exploited. However, in many cases, it is more feasible to enforce writing code that adheres to safe coding practices from the beginning. Several memory-safe languages, such as Rust[58] and Go[24], have been introduced in recent years to address this challenge. Due to its close to native performance, Rust has emerged as a strong contender to replace C/C++ for system applications such as operating systems (Redox[76], Theseus[7], and Tock[37]), web engines (Servo[63], and Deno[15]), and blockchains (lighthouse[41], OpenEthereum[50], and electrs[18]).

Unfortunately, even Rust with its strict compile time policies cannot guarantee complete safety. For example, writing a doubly-linked data structure or calling a foreign function cannot be achieved in safe Rust. To allow such constructs for which the Rust compiler cannot guarantee safety, Rust has `unsafe Rust`. Recent research [84, 52] shows `unsafe Rust` code is the primary culprit for Rust vulnerabilities. Despite the issues with the use of `unsafe` in Rust, the new language is gaining popularity as a system

programming language which has led to extensive crate (Rust libraries) development to support more functionality. New libraries introduce more `unsafe` Rust code. Also, some industry development teams are adopting a unique approach; rather than rewrite existing code in Rust, project teams such as the Android Linux kernel group are developing new coding projects in Rust. However, the new Rust code must interface with the existing C/C++ codebase, widening the Foreign Function Interface (FFI) footprint, and increasing the use of `unsafe` Rust.

In recent years, vulnerability detection in Rust has primarily focused on identifying memory and thread safety bugs in crates as 25% of `unsafe` code resides in crates. Issues in crates may have a widespread impact as a widely used crate can affect numerous other crates and applications running on multiple platforms. However, most bug detectors perform static analysis and focus on specific domains. For example, Rudra[4] detects three specific bug patterns (panic safety, higher-order invariant, send/sync variance) in `unsafe` Rust code. Another popular static analysis tool, MirChecker[39], leverages numerical and symbolic information in Rust’s Mid-level Intermediate Representation (MIR). Due to its design, most bugs detected by MirChecker are integer related e.g. integer-overflow, division-by-zero, etc. FFIChecker[38] is another static bug detector that detects memory management issues across the Rust/C FFI. There are also dynamic analysis-based bug detectors in Rust. For example, `afl.rs`[1] is a rustc wrapper of AFL++, and `cargo-fuzz`[9] leverages the Cargo package manager to invoke libfuzzer. Overall, dynamic analysis of Rust crates is a new research direction, which the insights gained from this study can provide direction toward prioritizing reported vulnerabilities and generating actual exploits.

This chapter represents our study of three static analysis tools: MirChecker, Rudra, and FFIChecker, and two dynamic analysis tools: `afl.rs` and `cargo-fuzz`. These tools identify different types of vulnerabilities, with minimal overlap between their respective detection domains. To verify our setup, we ran the selected tools against detections reported in their trophy cases [82, 80, 81, 79].

Next, we ran the tools against three popular Rust projects with more than twenty crate dependencies. We discovered new vulnerabilities in the projects and the crates. However, we observed that static analysis methods usually fail to report vulnerabilities detected by the fuzzers. This, unfortunately, indicates that the

static analysis tools, despite a large number of reports and vast false positives, are inadequate in identifying all valid vulnerabilities. We also discovered that fuzzers also report false positive vulnerabilities trapped by developers' instrumented assert and trigger panic! error. Overall, neither static nor dynamic vulnerability detector tools are entirely reliable.

The next steps were focused on generating exploits of the reported vulnerabilities. From this process, we identified challenges in finding exploits and assessed the criticality of the reported vulnerabilities. We assembled a comprehensive list of vulnerability reports and attempted to generate exploits via fuzzing. This proved difficult because most vulnerabilities reported were from the dependent crates. Fuzzers require a target harness to guide the fuzzer toward the target. Providing an appropriate target harness to reach a crate vulnerability was not trivial. We leveraged the RULF[29] tool to generate target harness code to activate various sequences of APIs. Then, we used the static analysis reports from crates as our guide to RULF. This process was beneficial to minimize the amount of human labor. Still, manual modification of the target harness code was necessary to achieve specific exploits, such as panic safety bugs of Rudra. The process was beneficial for identifying the challenges to generating exploits and organizing our efforts toward solutions.

Finally, we propose a method for categorizing vulnerability reports based on the criteria that exploitable vulnerabilities reachable from multiple APIs (paths) of the crates are the most critical. We found that, on average, 23% of vulnerabilities are critical while 64% of vulnerabilities are considered high impact (due to being in crates). We also determined that combining a localized symbolic analysis tool with a fuzzer and RULF is a practical approach to discover if a vulnerability can be exploited.

Overall, we make the following three key contributions:

- We identify key challenges to finding exploits for vulnerabilities reported by vulnerability detector tools. This is accomplished primarily with fuzzers aided by the RULF project for generating target harnesses. We found this method partially effective and additionally carefully analyzed the static analysis vulnerability reports as in our initial approach.

- We propose a priority categorization method for the reported Rust vulnerabilities. Our observation concludes that an exploitable vulnerability reachable from multiple APIs paths within a crate is the most critical. We believe our priority categorization will assist developers in deciding which vulnerabilities to resolve first.
- We have also demonstrated that combining fuzzing with RULF and the symbolic engine, KLEE[8], provides a practical approach for assessing exploitable vulnerabilities when an immediate automated solution is not available. Such a tool can quickly annotate the criticality of vulnerabilities, reducing the manual effort for exploit generation.

### 3.1 Background

Rust is a memory-safe language built for system application development [33]. Rust achieves memory management guarantees by enforcing safety policies during compilation, with additional checks performed at runtime depending on the build type.

**Compile-time Check.** Ownership, lifetime, and fearless concurrency are distinguishing features of the Rust language. Rust is strongly-typed; each value has an owner and there can only be one owner at a time. By default, values are immutable unless explicitly marked as mutable. Rust enforces strict rules regarding ownership and borrowing: only one mutable borrower or multiple immutable borrowers can access a value at a time, and the borrower’s lifetime cannot exceed that of the owner. The Rust compiler employs a borrow checker that assigns a lifetime to each borrow that is strongly connected to its owner variable. When the owner goes out of scope, the value is implicitly dropped. This compiler instrumentation avoids garbage collector dependency and guarantees no use-after-free/double-free bugs. By restricting multiple mutable references/owners simultaneously, the compiler also protects values against data races. Rust Send/Sync traits provide concurrency guarantees beyond simple data races. The Send/Sync traits force developers to guarantee specific features of their type before they can be allowed in safe multi-threaded communication. A Send trait type guarantees that the ownership of these values can be safely transferred

between threads. A `Sync` trait type guarantees that the values of this type can be referenced from multiple threads.

**Runtime Check** Many languages support exception handlers to manage a program in unexpected situations like divide-by-zero. The Rust feature is called error handling, subcategorized as recoverable and unrecoverable errors. The Rust compiler forces developers to handle success and fail cases for recoverable errors. By default, unrecoverable errors are handled by the Rust ecosystem. For example, if Rust detects a buffer overflow in the program execution, it triggers a panic error. The Rust ecosystem immediately takes control of the execution, unwinds the call stack and invokes destructors of the stack-allocated objects before safely exiting the process. However, detecting such errors at runtime is costly, so Rust offers both debug and release builds. For example, a debug binary is protected against integer overflow but the release binary is not. This leads to undiscovered integer overflows in the release builds, which may later lead to buffer overflow and cause a panic error in production. Although the Rust ecosystem safely terminates the program in such cases, persistent applications e.g., web hosting, can suffer from DoS attacks. Figure 3.1 shows an integer overflow reported by authors [27]. The overflow was detected while running `afl.rs` in the Deno project [15], a popular JavaScript, TypeScript, and WebAssembly runtime built on Rust. However, the underlying vulnerability is in its dependency crate, the `swc` project [69], which is also used by popular projects such as Parcel and Next.js.

### 3.1.1 Unsafe Rust

Rust code is generally safe due to the compile-time rules. However, such strict rules also prohibit features required for certain designs e.g. doubly-linked lists. So, Rust offers `unsafe` Rust that permits a developer to write code that ultimately violates the safe Rust policies. The compiler overlooks code marked `unsafe` and leaves the responsibility of safety on the developer. According to RUSTSEC, `unsafe` Rust code contributes to more than 80% of vulnerability reports on Rust. More importantly, `unsafe` Rust has primarily been used in crates, the Rust library, meaning the `unsafe` code is frequently reused. Figure 3.2 shows a recent report from `once_cell` where the developer assumes the `None` handle can

```

1  fn digits(value: u64, radix: u64) -> ... {
2      debug_assert!(radix > 0);
3
4      #[derive(Clone, Copy)]
5      struct Digits {
6          n: u64,
7          divisor: u64,
8      }
9
10     impl Digits {
11         fn new(n: u64, radix: u64) -> Self {
12             let mut divisor = 1;
13             while n >= divisor * radix { // integer overflow
14                 divisor *= radix;
15             }
16             ...
17         }
18     }
19     ...
20 }

```

Figure 3.1: An integer overflow report from Speedy Web Compiler (SWC) project reported by the authors.

not be reached [60]. Nevertheless, a panic from its caller triggers an undefined behavior because of the `unreachable_unchecked()`. The `unsafe` keyword is also required when calling non-Rust code, which is inherently unsafe as both safe and unsafe code share the process memory. For example, a Rust raw pointer can be freed by the external C code, leading to double-free when implicitly dropped by Rust [38].

### 3.1.2 Rust Vulnerability Detector

We have observed that a vulnerability can be anywhere (safe/unsafe Rust, FFI) in a Rust application. It also sometimes requires a specific usage of crates (e.g. panic from a user function). Overall, it is difficult to find a single vulnerability detector that can identify all types of security bugs.

```

1  pub fn force(this: &Lazy<T, F>) -> &T {
2      this.cell.get_or_init(|| unsafe {
3          match (*this.init.get()).take() {
4              Some(f) => f(),
5              //reaching below function is Undefined Behavior.
6              None => unreachable_unchecked(),
7          }
8      })
9  }
10 // thread 'main' panicked at 'explicit panic',
11 // [1] 3838533 illegal hardware instruction (core dumped)

```

Figure 3.2: A panic will cause an undefined behavior in unsafe Rust

Figure 3.3 shows most Rust vulnerability detectors are based on static analysis. The first research in this area is called Crust [78]. It focuses on functions containing `unsafe` Rust code and translates them to C code for further analysis by existing C/C++ vulnerability detectors. Similarly, other research [5, 3] also translates Rust code into a different language and leverages existing technology to detect vulnerabilities statically. However, recent research such as MIRChecker [39], SafeDrop [12], and FFIChecker [38] depends on data-flow analysis generated directly from Rust code. They identify different types of memory vulnerabilities. For example, MIRChecker detects integer-related bugs, FFIChecker detects security issues in FFI, and SafeDrop focuses on detecting temporal memory corruptions. Moreover, Qin et al. [52] developed an IDE tool to visualize blocking bugs in Rust concurrency statically. Rupair [26] is an automated system that detects potential buffer overflows and rectifies them. Rudra [4] is unique research because it performs scalable crate analysis detecting three specific bug patterns. RustBelt [30] introduces a formal verifier for Rust code based on the semantic model of the language. Later, multiple research [14, 46] endeavors extend RustBelt to cover more semantics, including concurrency, alongside other independent research [53, 31] in the arena.

Compared to static analysis, dynamic/concolic analysis of Rust code has yet to be extensively explored by researchers. However, industry partners have developed several effective dynamic analysis tools. For



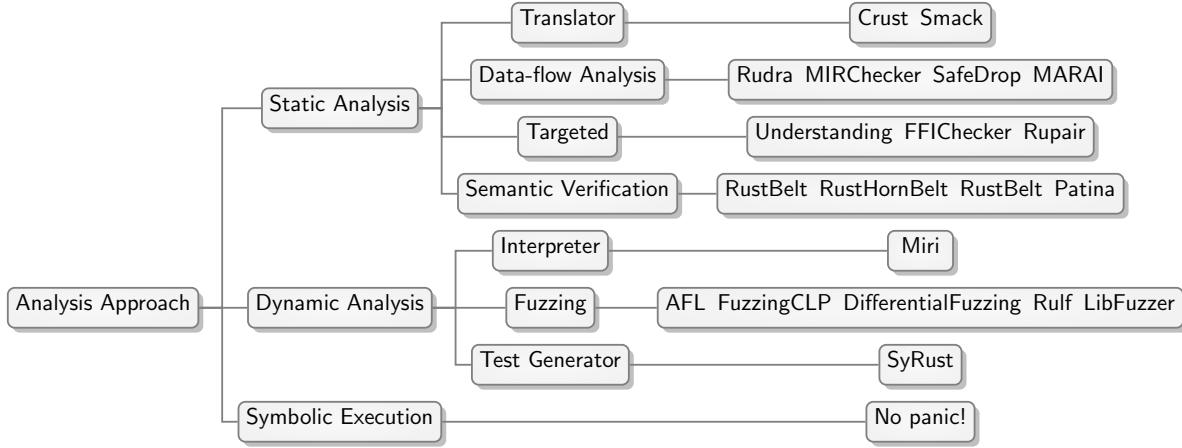


Figure 3.3: Rust vulnerability detectors categorized by their analysis approach.

example, Miri [49] interprets Rust MIR and detects potential undefined behaviors (e.g. memory leaks, alignment issues, etc.). Fuzzers based on existing tools, `afl.rs` [2] and `cargo-fuzz` [9] are two projects that have been adopted by crate developers. However, writing harness programs for fuzzers remains a critical challenge. Several research endeavors [29, 85, 71, 16] have been conducted to analyze the Rust target and automatically generate good harness programs for the fuzzer. So far, the only symbolic analysis research is a KLEE-based [35] generic contract-based verifier [42] of Rust.

## 3.2 State-of-the-Art Bug Detector and Exploit Generator

In this study, we chose MIRChecker and Rudra to statically detect bugs in Rust code. We also evaluated FFIChecker to identify FFI functions that may corrupt Rust memory. For exploit generation, we leverage `afl.rs` and `cargo-fuzz`. Additionally, we employ RULF to generate fuzz targets. In this section, we provide a brief explanation of methodology.

### 3.2.1 MIRChecker

MIRChecker [39] statically analyzes Rust MIR (Mid-level Intermediate Representation) to automatically detect potential runtime crashes and memory-safety errors. It uses a dedicated abstract domain that keeps track of numerical (for integer bounds) and symbolic values (as a memory model). The former helps identify integer-related bugs (e.g. integer overflow, division by zero, etc.), and the latter helps detect memory-related bugs (e.g. use-after-free, double-free, etc.). However, to analyze a crate, the user must write code similar to a test harness, as usually required for a fuzzer. As MIRChecker generates the abstract states based on the Control Flow Graph (CFG) of the harness code, it limits the code coverage analysis. Given MIRChecker’s significantly high false positive rate of 79.2%, it also prompts concerns regarding the potential false negatives in its report; the absence of a bug cannot be guaranteed.

### 3.2.2 Rudra

Rudra [4] is a bug pattern match-based static analysis tool. The three bug patterns target unsafe Rust code that may lead to memory corruption and synchronization errors. The panic safety bug depends on an external function call that, if a panic error is triggered, executes unsafe Rust code that may leave an unstabilized memory region to the panic handler during stack unwinding. The higher-order safety invariant bug is caused when unsafe Rust code wrongfully assumes certain properties (e.g. always returns the same values for the same input) on a generic, higher-order invariant, function. This bug may lead to undefined behavior due to inappropriate memory states. The third bug is a thread safety error where generic types incorrectly bind their inner type while implementing send/sync. Rudra uses an unsafe dataflow checker to match panic safety and higher-order safety invariant bugs and a send/sync variance checker to find generic types with incorrect send/sync implementation. However, crate analysis using Rudra is scalable because it does not require significant manual effort such as writing test harnesses. It also guarantees complete code coverage as it is purely static analysis. Unfortunately, in addition to being

limited to three bug patterns, the tool also exhibits high false positives in low-precision analysis and high false negatives in high-precision analysis.

### 3.2.3 FFIChecker

FFIChecker [38] targets a different format of unsafe Rust code, a foreign function interface (FFI). Rust crates inherently depend on existing C/C++ codebases to avoid rewriting existing code. FFIChecker uses static analysis and a bug detector to analyze the FFI of memory bugs from C/C++ code that corrupts safe Rust memory, especially heap memory. FFIChecker employs an approach similar to MIRChecker that augments the taint analysis algorithm to monitor the states of heap memory. It checks whether certain heap memory is either deallocated by C/C++ code originally allocated by Rust code, or if it leads to a memory leak because the C/C++ code crashes before returning to its owner, the Rust code. Like MIRChecker, FFIChecker is limited by its test harness CFG coverage, potentially missing bugs.

### 3.2.4 afl.rs

The `afl.rs` [2] is an AFL++-based fuzzer [21] for Rust code that can generate inputs to trigger a panic error/memory corruption. AFL++ has a modified clang compiler for the instrumentation that can be effectively used in LLVM IR. The Rust compiler also generates LLVM IR, so `afl.rs` leverages this capacity. The `afl.rs` additionally introduces a `fuzz!` macro that reads bytes from standard input and passes the bytes to the provided closure. In this case, the closure code is a test harness to analyze a Rust crate. In case of a panic error, the `fuzz!` macro also ensures it catches and forcefully triggers `process::abort` instead. Otherwise, it would not be reported to the fuzzer.

### 3.2.5 cargo-fuzz

The `cargo-fuzz` [9] supports libFuzzer [62] that uses LLVM’s Sanitizer Coverage instrumentation for the code coverage information. Similar to `afl.rs`, libFuzzer integrates to fuzz Rust code. The `cargo-fuzz` also leverages a macro `fuzz_target!` to write the test harness. The harness is repeatedly

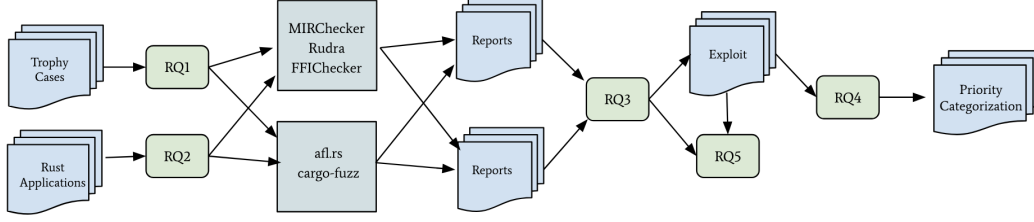


Figure 3.4: Research questions and their connections

executed with a pseudo-random input generated by the libFuzzer until the code hits an error condition (segfault, panic, etc). However, because `afl.rs` and `cargo-fuzz` depend on two different underlying fuzzers, their outcome is usually different.

### 3.2.6 RULF

RULF [29] is aimed to explore the Rust crate API and create fuzz targets based on API. The tool guarantees compilable and effective fuzz targets. RULF begins with a breadth-first-search (BFS) of API sequences. To keep the fuzz target small, a threshold is set. This may cause uncovered APIs, so RULF runs a backward search of the missed APIs. A combination of both may result in duplication, so RULF refines the sets to reduce to a subset of the fuzz targets. RULF has been tested against `afl.rs` fuzzer with multiple crates. We use RULF to generate fuzz targets for both `afl.rs` and `cargo-fuzz`.

## 3.3 Research Questions

Our primary focus is determining the types of security vulnerabilities that state-of-the-art vulnerability detectors can identify. The secondary focus is identifying the challenges in finding exploits of the reported vulnerabilities. We present the following five questions and expect our observations and evaluations gathered from both focus areas to provide valuable insights that will guide research on Rust vulnerability and exploit detection. Figure 3.4 depicts the relations between the artifacts and tools to each question.

**RQ1 (Base Truth): How well do the tools perform in identifying trophy cases?** Our selected projects have published their trophy vulnerability reports [82, 80, 81, 79]. Some of them are also reported in RustSec. We selected three reports from each project to verify our setup. Note that each tool can identify multiple types of vulnerabilities. Therefore, we select trophies that encompass a wide variety of cases. We carefully review each vulnerability report.

**RQ2: Do the tools identify new vulnerabilities?** Next, we evaluate our selected tools against three popular Rust applications [15, 41, 63] and their dependent crates. Our goal is twofold: 1) to determine if the static analysis tools identify new vulnerabilities in the crates, and 2) to assess whether a fuzzer on the application identifies vulnerabilities that were not reported by the static analysis tools. This comparison helps us understand why a static analysis tool may result in a false negative. Note, our investigation also includes reported vulnerabilities by independent parties. This analysis provides the ground truth for our **RQ3** that identifies challenges in finding exploits. Additionally, we compare our results with those from **RQ1** to identify failing scenarios.

**RQ3: What are the challenges in finding exploits of statically identified vulnerabilities?** We combine our reported vulnerabilities from **RQ1** and **RQ2**. Finding an exploit is essential to prove criticality. If a statically found vulnerability has already been found by our fuzzers, we consider it readily provable. If a similar vulnerability requires a target harness for the fuzzer, we first run the target against the RULF project to see if it can produce the required target harness. If RULF fails to generate the expected sequence of the APIs, we study the crates individually and write a target harness with manual effort. If all attempts fail to identify an exploit, we consider the vulnerability difficult to prove. We investigate these cases further to propose a potential solution using symbolic analysis (**RQ5**).

**RQ4: How can the vulnerabilities be categorized to assign priority?** The exploit of a vulnerability brings the vulnerable target to a critical edge. However, some exploits may cause more harm than others. We categorize the vulnerability reports by measuring three factors: 1) Is an exploit identified (**RQ3**) for the vulnerability? 2) How many different exploits are identified for the vulnerability? 3) Is the vulnerability in the application or a crate? If an exploit has not been identified, we classify as low priority. For

an exploitable vulnerability that is reachable from multiple paths, we classify as high priority. If such a vulnerability is in a crate, we assign critical priority. If an exploitable vulnerability is only available through a single path, we classify as medium priority. We propose an automated priority assignment based on our criteria.

**RQ5: Would a localized concolic analysis help automatically find exploits?** We create two sets of reported vulnerabilities. The first set is vulnerabilities with exploits requiring manual effort, and the second set is vulnerabilities for which we do not have exploits. We compare the sets and their attributes to understand how humans might anticipate exploits and determine if an automated tool can follow a similar approach. We evaluate a localized concolic analysis approach to address the challenge. Our first attempt was to employ the fuzzer and RULF to find a reachable path to a vulnerable function. At this point, we use our knowledge from **RQ3** to identify which attributes are essential to exploit the specific type of bug report. Later, the symbolic engine KLEE will locally evaluate the function to prove exploitability.

## 3.4 Evaluation

Our evaluation attempts to answer the five research questions. We present our insights and observations. Our evaluation signifies that exploit generation is equally essential to vulnerability detection, especially for determining vulnerability priority. Additional techniques, such as localized concolic analysis, can also reduce the required manual effort.

**Experimental setup.** We ran all the following experiments on a machine with an 8-core AMD Ryzen 3700 and 32 GB memory. We allocated one hour for each test study, so we ran the tools on average for 5 hours for each research question. However, we spent most of our evaluation time finding the working target harness and studying the tools for their success/failure.

### 3.4.1 Study on Trophy Cases

We selected three trophies from each of the tools. Table: 3.1 demonstrates the selected trophies by their vulnerability type. We ensure that the trophies selected encompass the full coverage of their tools and include any unique vulnerabilities to consider. For example, MIRChecker can detect integer-related and temporal memory safety bugs. However, its trophy cases include more integer-related than memory safety bug reports. Furthermore, in comparing MIRChecker trophies with those from Rudra and FFIChecker, MIRChecker stood alone at the top for robust numerical analysis. Rudra, on the other hand, detects bugs that could not be verified by fuzzing without specific knowledge of the details. We discuss this issue in **RQ3**.

Table 3.1: Selected trophy cases from MIRChecker, Rudra, FFIChecker, afl.rs, and cargo-fuzz

<b>Trophy Case</b>	<b>Reported By</b>	<b>Vulnerability Type</b>
bitvec-0.21.1	MIRChecker	division by zero
byte-unit-4.0.10	MIRChecker	integer overflow
gmath-0.1.0	MIRChecker	use after free
gls-layout-0.3.2	Rudra	PanicSafety
rdiff-0.1.2	Rudra	HigherOrderInvariant
va-ts-0.0.3	Rudra	SendSyncVariance
libtaos-0.4.0	FFIChecker	exception safety
triangle-rs-0.1.2	FFIChecker	undefined behavior
emd-0.1.1	FFIChecker	memory leak
clap-2.33.3	afl.rs	utf-8
image-0.23.4	afl.rs	index out-of-range
jpeg-decoder-0.1.4	afl.rs	unwrap
bincode-2.0.0	cargo-fuzz	panic
boa-0.10	cargo-fuzz	assertion fail
flac-vo.5.0	cargo-fuzz	index out of bounds

We experienced that Rudra is the most scalable tool, and the other tools require a target harness which is challenging to produce. We successfully reproduced the trophy cases with a minimum of false positive reports. Surprisingly, static analysis tools outperform the fuzzers in reporting fewer potential vulnerabilities. In general, it is expected that fuzzers do not report false positives; however, in Rust, our fuzzers occasionally hit panic errors due to program assertions. As they are intentionally set by the developer, we are not concerned with them. However, we have to confirm these cases manually.

Overall, after completing this step, we gain a better understanding of the strengths and weaknesses of the tools.

### 3.4.2 New Vulnerabilities Found

Our selected three Rust projects are the most popular projects in their area. Each project depends on (mostly) popular crates available in crates.io and their custom crates. Figure 3.5 refers to the statistics regarding the projects and their dependent crates.

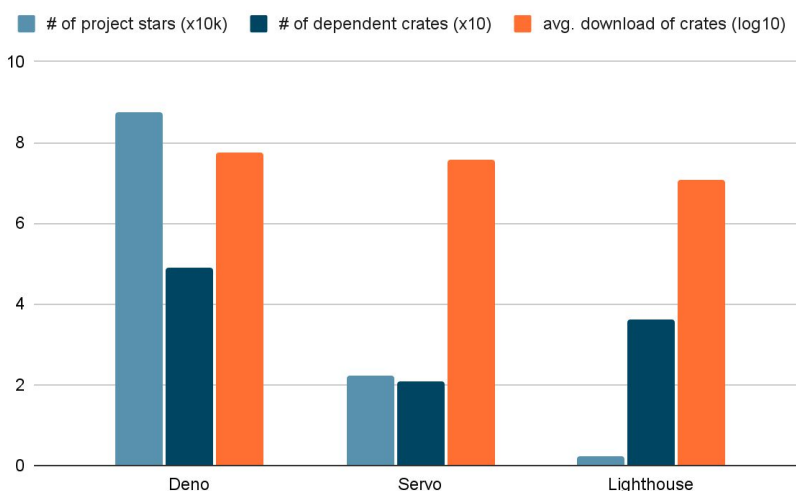


Figure 3.5: Statistics on selected projects including dependent crates.

We ran each tool for each project and also separately in their dependent crates. Table 3.2 depicts the true vulnerability reports we found during the experiments. Our observation demonstrates that fuzzers are more effective than static analysis for projects and crates. So, we revisited the trophy cases of static analysis to cross-check the crates with dependent crates of the selected projects. We found that most trophy cases were reporting vulnerabilities from unpopular crates. However, because most tools rely on a target harness, we do not claim to have a complete vulnerability analysis of any project or crate.

Figure 3.6 shows a send/sync variance vulnerability reported by Rudra while attempting Rudra on anyhow-1.0.57 crates. The crate was one of the dependencies of the Deno project. However, the vulnerable feature was never directly used by Deno because it is inside a private module. Our further investigation



Table 3.2: Vulnerability reports from selected projects and their dependent crates. Note, some of the project reports are duplicates from crates.

Tools	Deno		Servo		Lighthouse	
	Proj	Cra	Proj	Cra	Proj	Cra
MIRChecker	0	2	0	2	1	3
Rudra	1	3	1	1	0	2
FFIChecker	0	1	0	0	0	1
afl.rs	4	5	3	4	4	5
cargo-fuzz	3	4	3	2	5	6

reveals that the crate only defines the generic `T as ErrorImpl` where the inner type has implemented both the `Send` and `Sync` trait. So, it is a false positive report from Rudra due to the absence of context analysis.

Manual analysis requires costly human resources, so we are seeking a systematic solution that can help us reduce most false positive reports more quickly.

### 3.4.3 Challenges on Exploitation Generation

During the exploit generation phase, we began the process by finding common vulnerabilities reported by static analysis and fuzzer tools. We quickly found it difficult for fuzzers to reproduce vulnerabilities reported by static analysis tools, especially those reported by Rudra whose vulnerabilities depend on critical attributes.

For example, Rudra's panic safety bug requires executing a `panic!` error. We found that Rudra only reports panic safety in functions with closures as their argument. So, we made sure those closures would trigger a `panic!` error. Additionally, while the closure can be executed from any path, the vulnerability will usually only be triggered from one path. Not only that, the panic safety bug is either based on double free or use after free. So, we also must determine which memory is unstable in that path and explicitly cause a drop after the `panic!` error trigger. So, to achieve an exploit of the panic safety bug, we have to meet three challenges: 1) find the path where the `panic!` would end up in an unstable state of memory,

```

1  pub struct Own<T>
2  where
3      T: ?Sized,
4  {
5      pub ptr: NonNull<T>,
6  }
7
8  unsafe impl<T> Send for Own<T> where T: ?Sized {}
9
10 unsafe impl<T> Sync for Own<T> where T: ?Sized {}
11
12 impl<T> Own<T>
13 where
14     T: ?Sized,
15 {
16     pub fn new(ptr: Box<T>) -> Self {
17         Own {
18             ptr: unsafe { NonNull::new_unchecked(Box::into_raw(ptr)) },
19         }
20     }
21
22     ...
23
24     pub unsafe fn boxed(self) -> Box<T> {
25         Box::from_raw(self.ptr.as_ptr())
26     }
27 }
28
29 // Safety: requires layout of *e to match ErrorImpl<E>.
30 unsafe fn object_boxed<E>(e: Own<ErrorImpl>) -> Box<dyn StdError + Send + Sync + 'static>
31 where
32     E: StdError + Send + Sync + 'static,
33 {
34     // Attach ErrorImpl<E>'s native StdError vtable. The StdError impl is below.
35     e.cast::<ErrorImpl<E>>().boxed()
36 }

```

Figure 3.6: An incorrect send/sync trait bound for a generic type parameter in anyhow-1.0.57. However, it is a false positive report because the only use of boxed() is using a type that guarantees thread-safety.

2) find the memory which would be responsible for either double free or use-after-free, and 3) write a complex data structure for the memory representation that explicitly drops values.

We encountered similar challenges with higher-order safety invariant bugs and propagating send/sync in generic types bugs. However, integer-related bugs from MIRChecker were comfortably triggered, but their exploit was hindered due to their dependence on memory-related corruption.

### 3.4.4 Vulnerability Priority Categorization

Our collection of reported vulnerabilities from the diverse set of tools and projects is ideal for testing the effectiveness of our report prioritization methods. We found that most static analysis reports are false positives due to their context-insensitive manner of analysis. Figure 3.6 is an ideal case study. Regardless, their reports help us pinpoint targets in the crate and write an effective harness code. This process requires a heavy manual workload which is not scalable considering the pace which the Rust ecosystem is advancing. For example, a current false positive vulnerability report may later be exploitable because it is publicly reachable. Therefore, excluding them from future reports would be a mistake.

Table 3.3: Vulnerability report distribution over crates and exploits.

Projects	# of reports	# of reports from crates	# of expl. reports
Deno	19	14	4
Servo	13	9	3
Lighthouse	21	13	4

Table: 3.3 demonstrates the difficulty in finding an exploit of a vulnerability, especially those only reported by static analysis in crates. Overall, we found that `af1.rs` performs best at achieving exploit. Although we could not find multiple exploits for each bug reported by `af1.rs`, most are reachable through multiple paths. We understand that such a vulnerability in a crate is most dangerous due to the increased potential of reaching the vulnerable code in an exploitable manner. This study supports our metrics for prioritization.

Our comprehensive study on vulnerability prioritization suggests the following:

1. Static analysis tools are easy to configure and test the crates. Their reports also indicate potentially vulnerable functions.
2. The assistance of RULF makes it easier to find initial harness code that could reach potentially vulnerable functions. Note that in most cases, RULF fails to find a direct executable path to the potentially vulnerable function.

3. A fuzzer running with the RULF-produced code is usually effective in finding the correct data set leading to the exploitable function. It also turns out that multiple paths to the exploitable function are available.
4. Finally, to achieve exploit on those paths, which is one of the challenging tasks, we consider integrating a localized symbolic engine as a potential solution. This approach is discussed in the future section.

### 3.4.5 Concolic Analysis for Exploitation

We found that guided fuzzing with the assistance of RULF for harness code is effective. However, more work was needed to prove that exploit is achievable. The first challenge was finding the path inside the potentially vulnerable function. This is crucial because most Rust vulnerabilities are relevant to `unsafe` code execution and also require satisfying other essential conditions. For example, a panic safety bug has to trigger a `panic!` error between the bypass and fix-up of an inconsistent state. So, the second challenge was to symbolize the state variable to produce the expected memory state. Similarly, an integer-related bug requires producing the integers that would lead to a `panic!` error.

We employed a KLEE-based symbolic engine at the entry of potentially exploitable functions and symbolized their arguments. We kept the global memory as the fuzzer produced, but identified interesting function arguments to find the appropriate value for them. In Figure 3.7, we call `klee_panic(filter)` to guarantee it would trigger a `panic!` error if it executes. Additionally, to trigger the double free, the `v[i]` at line 24 should implement an `explicit drop()`. So, we call `klee_drop(v[i])` that diverts the execution to an `explicit drop`, hence triggering the exploit as the proof of concept. Our identification and instrumentation of symbolization currently requires human involvement.

## 3.5 Future Work

This research provides important insights to shape future research in the area of vulnerability prioritization and exploit generation. In the following sections, we briefly discuss our future research plan on the relevant topics.

### 3.5.1 Automatic Prioritization of Vulnerabilities

Implementing automatic vulnerability prioritization would effectively contribute to a safer Rust ecosystem in the future. In this research, vulnerability prioritization was conducted with human involvement. However, a set of static analysis tools and their reports could effectively guide RULF and fuzzers to produce near-perfect harness code, making an integrated system with minimal human intervention achievable.

Once this system builds the stack of reports, an analyzer could walk through them to determine their origin, exploitability, and other relevant factors. In the future, with even more tools available, such an integrated system could employ differential analysis to prioritize vulnerabilities with finer granularity. Additionally, integrating RustSec could help determine whether any project actively reported vulnerable code and if their proof of concept (PoC) is available.

### 3.5.2 Localized Concolic Analysis for Exploit Generation

RULF and `afl.rs` perform effectively together, but they are insufficient alone to prove the exploitability. Our human-involved concolic analysis proves the system's effectiveness. However, to automate this method, an additional analyzer is needed to determine the necessary instrumentation locations. Furthermore, KLEE requires direction to pinpoint which memory to symbolize; otherwise, it would blindly symbolize references instead of values. For example, if the argument is a mutable string, only the heap memory where the string literals are stored should be symbolized, and thus mutated. Therefore, we need to build a Rust type knowledge base that works with KLEE to leverage full path coverage in a function.

## 3.6 Conclusion

Rust represents a promising alternative to C/C++ for systems applications with its near-native performance and memory safety guarantees. The adoption of Rust in operating systems, web engines, and cloud technologies gives evidence of the promising future for software development in Rust. Research has proven the safety guarantees provided by the Rust compiler. However, Rust cannot guarantee complete safety, primarily because of the incorporation of `unsafe` Rust, which allows code to bypass the strict compile time checks. Recent research efforts have focused on identifying memory and thread safety bugs in Rust code, particularly within crates, due to their extensive use and widespread impact.

This paper examines three static analysis tools: MIRChecker, Rudra, and FFIChecker and two fuzzers: `af1.rs` and `cargo-fuzz`. RULF is employed to identify appropriate API sequences for our test harnesses. Research questions are formulated to assess the performance of each tool and the challenges of generating exploits. We started with establishing the base truth using the trophy cases, followed by a systematic reevaluation of the tools to identify new reports from three popular projects. We then use our collection of reports to compare and explain the challenges of establishing a systematic tool for exploit generation. Finally, we offer observations on how this study could improve developers' experiences by mitigating the need for manual vulnerability analysis.

```

1  pub struct DrainFilter<'a, T: 'a, F>
2  where
3      F: FnMut(&mut T) -> bool,
4  {
5      deq: &'a mut SliceDeque<T>,
6      idx: usize,
7      del: usize,
8      old_len: usize,
9      pred: F,
10 }
11
12 impl<'a, T, F> Iterator for DrainFilter<'a, T, F>
13 where
14     F: FnMut(&mut T) -> bool,
15 {
16     type Item = T;
17
18     fn next(&mut self) -> Option<T> {
19         unsafe {
20             while self.idx != self.old_len {
21                 let i = self.idx;
22                 self.idx += 1;
23                 ...
24                 if (self.pred)(&mut v[i]) { // klee_drop(v[i])
25                     ...
26                 }
27             }
28             None
29         }
30     }
31     ...
32 }
33
34 impl<T> SliceDeque<T> {
35     pub fn drain_filter<F>(&mut self, filter: F) -> DrainFilter<T, F>
36     where
37         F: FnMut(&mut T) -> bool,
38     {
39         ...
40         DrainFilter {
41             deq: self,
42             idx: 0,
43             del: 0,
44             old_len,
45             pred: filter, // klee_panic!(filter)
46         }
47     }
48 }

```

Figure 3.7: Demonstrate the required symbolization to exploit a vulnerability reported in `slide_deque` crate.

## CHAPTER 4

# REDUCING THE LEARNING BARRIERS OF RUST THROUGH VISUALIZATION

The core safety principles of Rust - ownership and borrowing - are conceptually clear and straightforward. Yet, their implementation can be counter-intuitive, diverge significantly from approaches in other programming languages, and lack visual clues in the Rust source code. Consequently, Rust presents a steep learning curve, described as “near vertical” by one programmer [23]. Rust programmers must adopt a fundamentally different way of thinking. To illustrate the difference, consider the code sample in Figure 4.1 which will not compile due to Rust’s ownership rules. Both variables, `grade` and `grades`, are initially owned by `main`. However, ownership of `grades` is transferred to `update_grades`, while `main` retains ownership of `grade`. The source code does not visually indicate these ownership dynamics.

This chapter introduces RustLIVE, an innovative visualization tool designed for Rust programmers. RustLIVE integrates seamlessly into the IDE, offering a visual timeline for each variable to clearly show ownership of resources and liveness of borrows. This tool helps Rust programmers retrain how they think - with memory safety in mind. While Rust is highly effective at mitigating serious coding errors, it introduces a paradigm that can be challenging for both experienced programmers and students. Rust strictly enforces the safety of owned and borrowed values, and non-compliance results in compilation



```

1  fn main() {
2      let mut n_grade = entered_grade;
3      let mut grades = vec![100, 90, 80, 70, 60];
4      update_grades(n_grade, grades);
5      println!("Grade calculated is: {}", n_grade);
6      println!("Final grades: {:?}", grades);
7  }
8  fn update_grades<T>(grade:T, grades:Vec<T>) {}

```

Figure 4.1: Rust program that will not compile due to the enforcement of the ownership principle. The ownership of `grades` is transferred to `update_grades` yet ownership of `n_grade` remains in `main`.

errors. To assist users in understanding these requirements, RustLIVE provides clear visualizations of resource ownership liveness of references.

## 4.1 Background and Motivation

Rust programmers must adhere to strict safety rules set by the compiler; programs that do not comply do not compile. Programmers must actively manage and track ownership of values and ensure that references are live. Although Rust’s ownership model, which moves heap variables and copies stack variables, is efficient, it is not visually intuitive and can be challenging to grasp. Perhaps most difficult to reason, programmers must track where references are live to understand where the current access method is.

A recent empirical study analyzing 100 Stack Overflow questions confirmed that programmers find Rust’s memory safety policies challenging to implement [88]. This study identified that understanding the liveness of references and reasoning about the movement of owned values are particularly difficult aspects of Rust. Furthermore, discussions among Rust developers reveal widespread complaints about the complexity of the borrow checker [86]. Experienced programmers, including those familiar with C++, have noted that the borrow checker requires a significant shift in thinking. Some describe it as an “alien concept,” [64] and it is common for programmers to discuss “fighting with the borrow checker.” [40]

Research also highlights that a common cause of blocking bugs in Rust is the misunderstanding of lifetime rules, suggesting that even seasoned developers struggle with these complex concepts [52]. The study proposes that tools to visualize borrow lifetimes during coding could greatly assist programmers in avoiding memory bugs. Industry professionals acknowledge that ‘Rust’s steep learning curve poses a significant barrier to its adoption, further emphasizing the challenges posed by these safety features [23].

#### **4.1.1 Visualization for Software Usability**

In his report on the usability of Rust, Will Crichton [11] suggested that “tools for visualizing static information such as types and lifetimes can help programmers build a mental model of how the type checker works and reason about why individual errors occur. Displaying this information in a succinct, non-intrusive, yet informative way is still an open problem.” A recent study on usability and HCI of Rust [20], suggests visualization of Rust’s lifetimes as a next step for usability of Rust.

The need for visualization has been a topic of discussion among Rust programmers in blogs and at RustConf since at least 2016. Several blog posts propose various visualizations. One depicts a vertical timeline similar to RustLIVE [54]. A different mock-up in another blog post shows a possible approach for visualizing Rust code in an editor [83]. On the Rust Internals Forum [19], a contributor (Nashenas88) started a thread to discuss ideas for visualizing the ownership and borrowing in an editor. The thread contains various ideas for such a visualization. One contributor was able to create a prototype in Atom. In all cases, the mockups were not substantially developed or evaluated.

Computer Science education scholars have long advocated for the use of visualization as an educational tool for understanding complex concepts. “Visualization can make focal and explicit processes that are hidden and implicit.” [68] Visualization enhances comprehension of inherently abstract software structures [25]. And to underscore the motivation of RustLIVE, visualization provides a conceptual model that aids learners in constructing programming knowledge [67]. This aligns with the sentiment that visualization facilitates the development of reasoning skills crucial in introductory programming education. The goal of software visualization is to improve our understanding of inherently invisible and

intangible software [25]. Some scholars suggest that visualization “can illuminate critical aspects of pivotal programming concepts, aiding learners in the formation of mental representations essential for mastering programming.” [66]

#### **4.1.2 Goal of RustLIVE**

RustLIVE aims to improve the usability of Rust by providing a visual timeline that clearly illustrates difficult concepts of Rust: the ownership of values and the liveness of references. The tool is designed to help programmers form a clear mental model of Rust’s memory safety principles and to enhance learning experiences. Furthermore, RustLIVE integrates with the Rust compiler, functioning as an independent learning tool. This feature is particularly useful as it provides personalized feedback to users.

An original objective of RustLIVE was to provide a timeline graphic adjacent to the source code that depicts the ownership of resources. An aspirational objective was to integrate with the borrow checker and depict the liveness of references. Both of these objectives were met along with useful additions: annotated source code and linking references with the referent. These features are described in the following sections.

### **4.2 Project Design and Compiler Integration**

The preferred development environment for Rust developers, as indicated by a recent survey [74], is VS Code. VS Code provides an integrated environment for language development tools, along with a webview panel API that facilitates a seamless environment for Rust developers. An innovative design, RustLIVE is integrated with the Rust compiler - the enforcer of Rust’s novel memory safety tenants. Utilizing a VS Code web panel to display the timeline graphic and annotated source code, RustLIVE extracts essential information from intermediate levels of the compile process, calculates the lifetimes of references and liveness of owned locals from the extracted information, and writes the json data. The VS Code extension

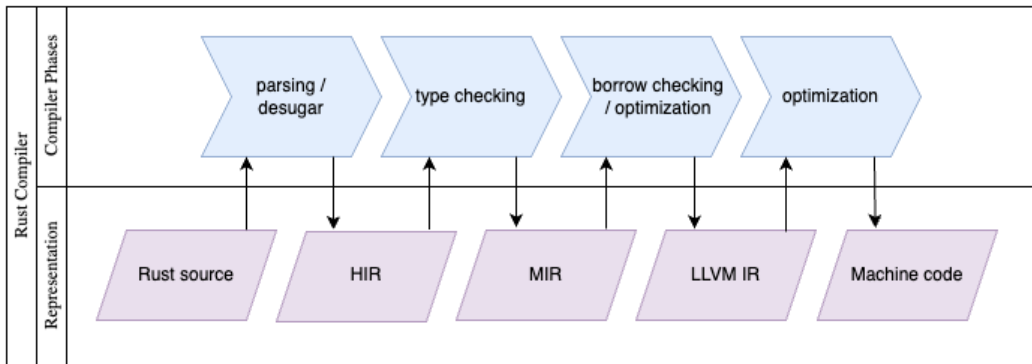


Figure 4.2: Phases and intermediate representations of the Rust compile process

opens the json file and generates visual timelines that illustrate the ownership of owned values and the lifetimes of references.

Rust memory safety is enforced by the compiler, so RustLIVE interacts directly with the compiler to gather data for visualization. Specifically, RustLIVE parses the Mid-level Intermediate Representation (MIR) to obtain data on owned values. RustLIVE obtains the computed facts from the borrow checker. The borrow checker is the component of the Rust compiler responsible for verifying lifetime constraints of references and enforcing non-lexical lifetimes. MIR and the borrow checker are central components in the design and functionality of RustLIVE.

#### 4.2.1 Compiler Phases and Intermediate Representations

The Rust compiler works incrementally, progressing through several phases to transform Rust source code into executable machine code. This process involves constructing intermediate representations at different stages, as illustrated in Figure 4.2.

## HIR

Rust source code is tokenized, forming a token stream that is parsed into an Abstract Syntax Tree (AST) [73]. The AST serves as the basis for generating the High-level Intermediate Representation (HIR), where source code is broken down into its constituent elements, macros are expanded and loops are desugared. The HIR represents the hierarchical syntax structure, grammar, of the source code, which is useful for parsing and static code analysis. HIR breaks the code down into hierarchical elements: functions, statements, expressions, etc. This representation is vital for type checking and trait solving, ensuring the program matches the grammar of the language, each `{` has a matching `}`, statements are terminated, etc. RustLIVE uses the `TyCtxt` from this phase. `TyCtxt` is the central data structure of the Rust compiler and the keeper of the results from the compiler queries. HIR is lowered to Mid-level Intermediate Representation (MIR).

## MIR

The Rust compiler underwent a grand transformation in 2016 with RFC 1211 [57] introducing a new intermediate, mid-level representation - MIR [72]. MIR represents a “vastly simpler” intermediate version of Rust between the HIR and LLVM. MIR facilitates many optimizations and Rust execution speeds increased with the incorporation of MIR because language-specific optimizations are possible in this representation. Without MIR, optimizations are reserved for the LLVM which is not unique to Rust. Several optimization passes are performed on the MIR but, importantly, MIR is used for borrow checking. MIR enables more flexible borrowing with better precision which allows more programs to compile and paves the way for the implementation of non-lexical lifetimes (NLL).

Internally to the compiler, MIR is represented in the compiler as a set of data structures that encode a control-flow graph (CFG). The control-flow graph is structured as a set of basic blocks, where each basic block contains a sequence of statements and ends in a terminator. Basic blocks execute sequentially without branching. For example, to represent an `if` statement, a different basic block is created to represent each possible path. The terminator of each basic block determines the control flow to the next block.

```

fn main() {
    let x = 42;
    println!("x is: {}", x);
}

fn main() -> () {
    let mut _0: ();
    let _1: i32;

    bb0: {
        StorageLive(_1);
        _1 = const 42i32;
        StorageLive(_2);
        _2 = _1;
        _3 = &_2;
        _4 = const "x is: {}";
        _5 = const core::fmt::ArgumentV1::new(_3, const core::fmt::Debug::fmt);
        _6 = &[_5] as &[const core::fmt::ArgumentV1];
        _7 = const core::fmt::Arguments::new_v1(&_4, _6);
        _8 = std::io::_print(_7) -> [return: bb1, unwind: bb2];
    }

    bb1: {
        StorageDead(_2);
        StorageDead(_1);
        return;
    }
}

```

Figure 4.3: Example Rust program and associated MIR

Memory locations, including local variables, parameters, and compiler temporaries are represented within MIR by `locals`, identified by an index and denoted with an underscore, as `_1` in Figure 4.3. In the example shown in Figure 4.3 `_0` is the return value and `_1` is the local for `x`. Basic blocks contain statements which form the control-flow graph. Execution follows each basic block where each statement is executed sequentially. Each `_x` represents a local, some temporary locals created by the compiler and others represent user variables. Assignments are common statements in MIR as values are moved to and from temporary and user storage. Storage is reserved for `x` at the first statement of basic block `0`. That storage is dropped in basic block `1` at the `StorageDead` statement.

MIR is converted to LLVM Intermediate Representation (LLVM IR). LLVM IR undergoes further optimization before ultimately being translated into machine code.

The incorporation of the MIR intermediate representation to the Rust compiler is essential to the RustLIVE project. The control flow graph representation is needed to create a timeline of where variables are live. While an abstract syntax tree representation is useful for parsing and syntax checking, it does not provide information about the flow of control during program execution. The control flow graph representation of the MIR captures the dynamic aspect of the code, how execution moves from one statement to the next or one block to the next. This is essential to be able to track points in the flow where variables are defined, moved, and dropped, where they are live or where they hold a value that might be used in the future. The MIR allows tracking variables as the program runs, enabling the depicting of the runtime liveness of variables.

#### **4.2.2 Query System**

The Rust compiler operates in the context of a query system. Each query in the compiler has an associated *provider* function that computes information about the source code input, caches and returns the results. This caching can significantly improve compilation times because the incremental process only needs to recompute parts that have changed and subsequent invocations of the same query can use cached results. Providers are functions that compute the result of the query. Also worth mentioning because it affects the implementation of RustLIVE, some query results are stolen, meaning their results are not efficient to implement copy so other parts of the process may take ownership instead of getting a copy from the cache. Although the entire Rust compile process does not yet employ the query system, the newer portion that generates MIR is implemented as a query process. [55]

#### **4.2.3 Transfer of Ownership**

In Rust every value has an owner and there can only be one owner at a time. This enforcement of the one-to-one relationship between owner to value sometimes requires ownership to be transferred rather than copied, especially when copying the resource is not feasible. This distinction between copying some values and transferring (or 'moving' as know in Rust) others differs from other programming languages.

```

fn main() {
    let n_grade = 100;
    let grades = vec![100, 90, 80, 70, 60];
    {
        let _new_grade = n_grade;
        let _new_grades = grades;
    }
    println!("Grade calculated is: {}", n_grade);
    // println!("Final grades: {:?}", grades);    <-- error
}

```

Figure 4.4: Ownership Transfer and Drop

Moreover, in Rust, there are no visual clues to indicate whether a value is being copied or moved, which can make reasoning about ownership challenging. Consider the example program in Figure 4.4. In this program, `n_grade` and `grades` are created and then assigned to new variables. We can print `n_grade` but get a compiler error if we try to print `grades`. For programmers new to Rust, this behavior is difficult to reason.

#### 4.2.4 Borrow Checker

In the Rust compile process, checking for violations of the borrow rules is the role of the borrow checker. The borrow checker operates on the Mid-level Intermediate Representation (MIR) as one of the final phases of the compilation process. In Rust 2018, a reformulation of the borrow checker took place, bringing with it nonlexical lifetimes (NLL), a big overhaul of how the borrow checker worked that allowed it to accept more programs. Prior to NLL, borrows were considered live until end of scope. While this approach was visually intuitive and relatively easy for programmers to reason, it prevented many safe programs from compiling. With NLL, however, a borrow is considered live only as long as it may be used again. Although NLL allows many more programs to compile, its lifetimes are less intuitive and pose challenges for programmers to reason.



```

1      fn main() {
2          let mut x = 22;
3          let y = &x;
4          x += 1;
5          println!("{}", y);
6      }

```

Figure 4.5: Borrow Checker Example Program

## Loans

To understand how the borrow checker detects violations of the borrow rules, consider a classic borrow checker error, depicted in Figure 4.5. In this sample code a local variable, `x`, is borrowed on line 3. The borrow creates a reference `y`. Borrow expressions, such as `&x` or `&mut x`, are referred to as *loans*. *Loans* have terms which include the local that was borrowed, `x` in this example, and whether it is a shared or mutable borrow. An error happens at a point in the program if an access to a local occurs that violates the terms of a loan and the loan is live at that point. An attempt to modify a local violates a shared loan. An access attempt violates a mutable loan. Applying the parameters of non-lexical lifetimes, NLL, the loan is live if it might be used later. In the sample code, the loan `&x`, created at line 3 is live at line 5 because it is used there. The loan is violated on line 4 when `x` is mutated. If no references to `x` were still in use, modifying `x` would not create a borrow violation.

Depicting the liveness of references is a goal of RustLIVE. The borrow checker computes the set of program points where a loan is live - the part of the program where the reference may be used. These points form the *lifetime* of the reference. The *lifetime* provides the parameters for detecting violations of the loan.

## Polonius formulation

The NLL redesign of the borrow checker originally did not accept all the cases as intended in the original NLL RFC [56] which led to a reformulation, named Polonius, and announced in a blog post by Niko

Matsakis [44]. Polonius changed the process of how lifetimes are calculated. Rather than tracking how long each reference is used, it tracks where each reference may have come from - its origin. RustLIVE utilizes the Polonius formulation of the borrow checker which uses *origins* to determine the liveness of borrows. [45]

The following terms from the borrow checker process are important to the implementation of RustLIVE:

- **Regions** - numbered identifiers representing each participant in the loan
- **Loans** - borrow expressions that associate a borrowed location and the mode with which it was borrowed (either shared or mutable)
- **Lifetimes** - the points in the program where the reference may be used again
- **Place** - location in memory
- **Rich Locations** - start and mid points for each MIR statement
- **Origins** - regions that represent where loans are issued.

The borrow checker works to accumulate the set of *regions*, *origins* and *loans* to create a set of facts, that together provide the information necessary to enforce the borrowing rules.

### Rich Locations

The borrow checker operates on the MIR but rather than simply a numbered statement in a basic block, (bb,stmt), the borrow checker defines points within the statement for finer granularity: a start point and a mid point of each MIR statement. These are referred to in the compiler as rich locations. In the program flow, the start is before the statement executes and the mid is where the statement is executing.

### Numbered Regions

The borrow checker creates numbered regions for every reference and borrow expression (*loan*). The regions are assigned unique numeric identifiers, referred to as *region variables*. The example in Figure 4.6

```

1      fn main() {
2          let mut x = 22;
3          let y: &'0 u32 = &'1x;
4          x += 1;
5          println!("{}", y);
6      }

```

Figure 4.6: Borrow Checker Example with Region Variables

illustrates numbered regions for the sample code. Region variables form the starting point for computing the liveness points for RustLIVE.

To compute lifetimes, the borrow checker must collect the points where regions are live. `region_0` must include all the points where `y` is live. In the example this includes the points from lines 4 and 5. `region_1` *flows into* `region_0` so `region_1` must outlive `region_0`. In other words, `x` has to live longer than `y`; the reference cannot outlive the value. `region_1` also includes the points from lines 4 and 5.

To determine the points where a loan is live, the borrow checker computes an *origin* for each numbered *region*. The origin for both `region_0` and `region_1` are the loan `&x`. Reference `y` originated with the loan created by `&x` so `y` includes the loan `&x` in its type. The borrow checker essentially creates a sub-typing dataflow relationship where the creating loan becomes a sub-type of the reference. With the origin of each reference, the borrow checker determines the liveness points.

At each rich location, the borrow checker considers all the live variables and determines what origins are in their types. A variable is live if its current value may be used. The borrow checker computes the set of liveness points for each origin. This set of liveness points is used to determine loan violations. At the statement in line 4, `x` is mutated which violates the terms of the loan `&x` and so the borrow checker must determine if the loan is live at this point. The variables that are live include `y` and `y` includes the loan `&x` therefore the loan is live.

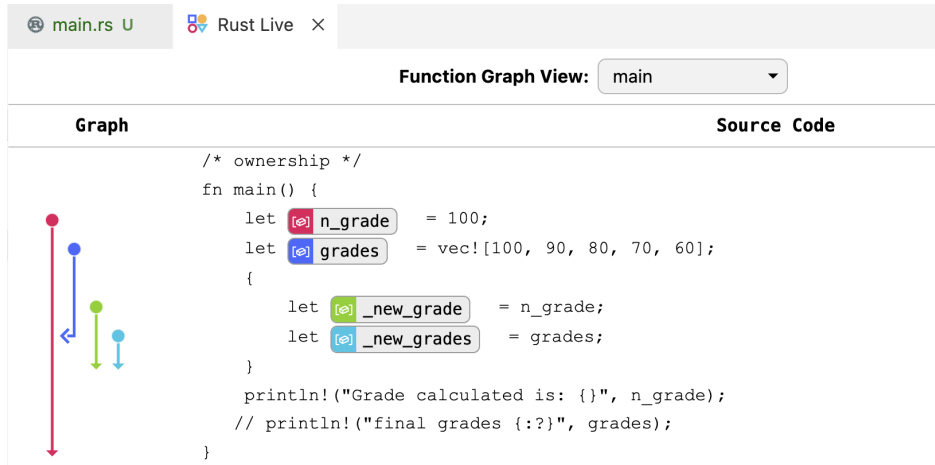


Figure 4.7: Ownership transfer and drop depicted in RustLIVE

## 4.2.5 RustLIVE Example Visualizations

### Ownership

The example program in Figure 4.4 gave no visual indication why `grades` could not be accessed in the `println!` macro. However, we can see in the RustLIVE visual in Figure 4.7 that the ownership of `grades` is moved in the inner block to enforce the only one owner principle; `grades` transferred ownership of the resource to `_new_grades`.

### Non-lexical lifetimes

In the RustLIVE graph of Figure 4.8, the `stringdata` is borrowed and the resulting reference is assigned to a variable `str`. `str` is passed to `capitalize`. According to Rust safety rules, the string can not be mutated while the reference is live. A programmer would likely reason that the lifetime of `str` extends to the end of scope, and prior to the incorporation of NLL, it did. However, with NLL, the lifetime of `str` ends at the call to `capitalize` which is depicted in the graph so mutating the data string with `data.push` is allowed.

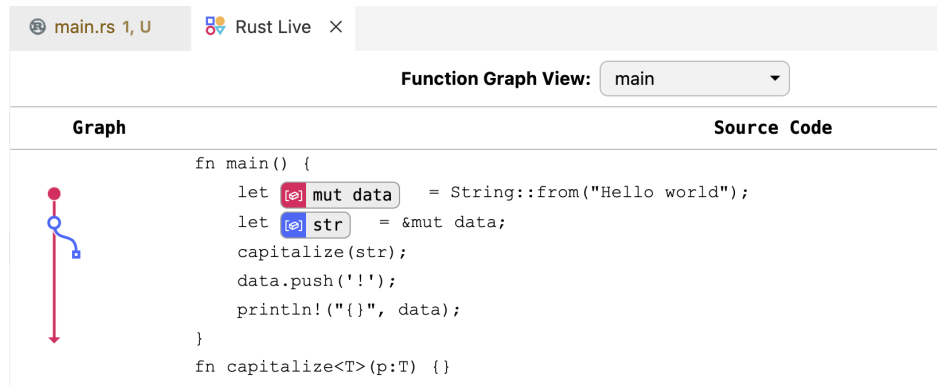


Figure 4.8: RustLIVE depiction of non-lexical lifetime (NLL)

## 4.3 Implementation

RustLIVE is invoked as a VS Code extension for Rust code development. When launched, it initiates a new compiler instance of the code in the active window but with a nightly version of the Rust compiler. It runs at multiple callback points during the compile process and gathers information about the locals of each function and writes the data gathered as a JSON file. The RustLIVE webview awaits the data write. On resolve of the asynchronous operation, the RustLIVE webview creates a visual graph representing the live timeline for each variable, owned and borrowed, side by side with the Rust source code. The visual depicts liveness of owned values and lifetimes of borrows, the most difficult safety rule for programmers to reason. The implementation of RustLIVE is depicted in Figure 4.9

### 4.3.1 Webview Extension

The user interface of RustLIVE is implemented as a webview panel in Visual Studio Code. Visual Studio Code is the most popular IDE for Rust programmers according to a developer survey. [28] Therefore, RustLIVE integrates well with the development workflow of most Rust programmers. The initial plan and prototype for RustLIVE included a timeline graph displayed alongside the working source code. However, without the ability to pinpoint locations within the active code window, the graph could not

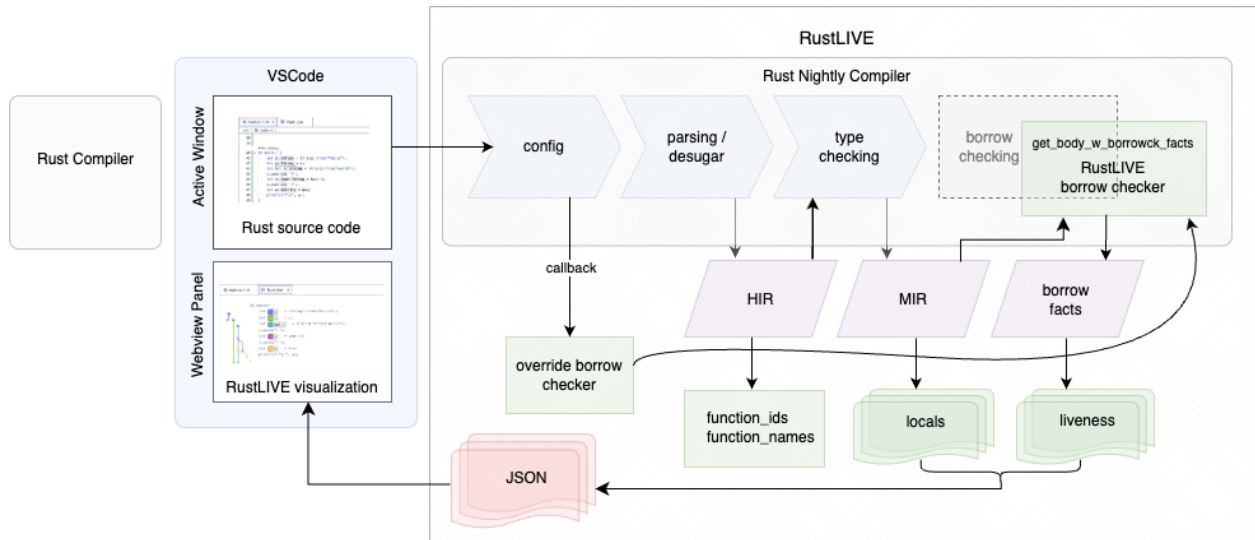


Figure 4.9: Implementation of RustLIVE

align accurately with the source code. This misalignment would render the visualization of vertices, moves, and liveness starts and stops ineffective.

To synchronize the graph with the source code, RustLIVE replicates the source code on the right side of the webview panel. An added benefit of displaying the source code in this manner is the ability to annotate locals in the source code so they visually correspond with their representation in the graph. RustLIVE outlines the definition of each local variable and matches its color with the depiction in the graph. This integration leverages VS Code’s capability to incorporate web-based visuals within the Rust development environment.

When a programmer executes the RustLIVE command from the RustLIVE tab at the bottom of the Visual Studio Code window or from the command palette drop-down menu, RustLIVE activates and opens the webview panel. Since RustLIVE relies on data gathered during the compilation process to generate its visualizations, it first initiates the compiler-query backend.

The webview frontend awaits the resolve of the compiler-query to compute the information for each function and each of its locals and writes the data. On resolve of the compiler-query, the webview

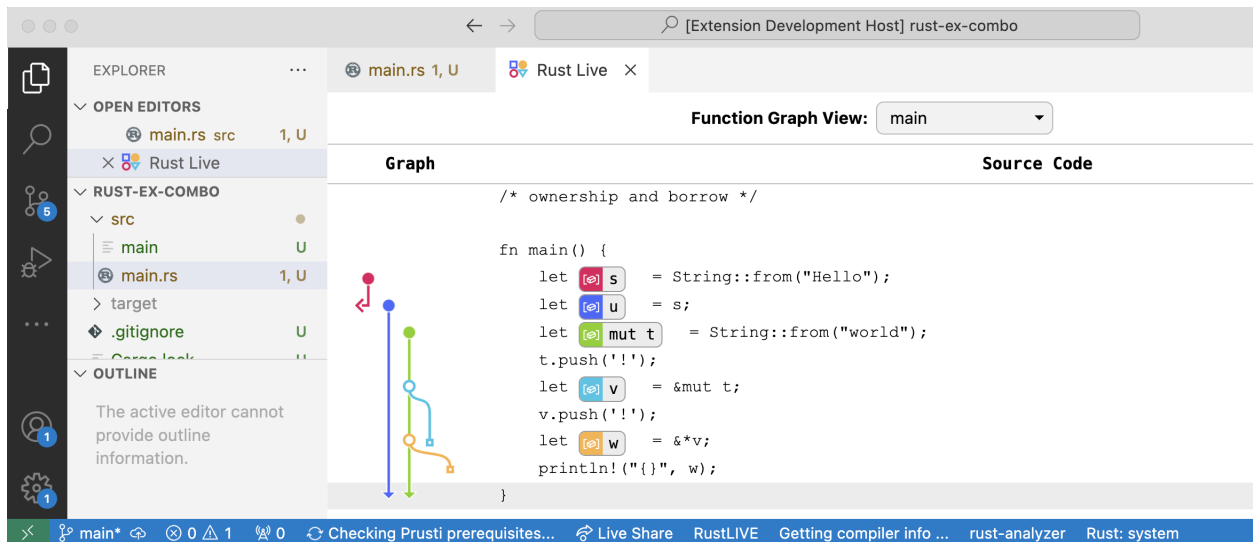


Figure 4.10: RustLIVE webview panel view: active code and timeline graphic side by side

subsequently generates the webview panel. The data is exported as JSON format, with each function represented by an object that contains key value pairs for each local and its associated data. The JSON data in Appendix A is collected and written by RustLIVE for the example program displayed in Figure 4.10.

The webview renders a table on the right, presenting annotated source code extracted from the active code window. On the left side, SVG graphics are dynamically created to represent the liveness of variables as a timeline graph. This dual presentation provides programmers a visual depiction of the dynamics of ownership and lifetime of borrows as they change in the flow of the source code. The webview panel is depicted in Figure 4.10

On launch of the RustLIVE VS Code extension, the `rust-live.view` command is registered via callback and the extension state is initialized. `getHtmlForWebview` retrieves the JSON data written by the compiler query and the source code from the active window, and constructs the html framework for the new webview panel. The html framework returned contains empty divs for the graph and table and returns the

```

interface Local {
  name: string; /* local name */
  local: number; /* local from mir representation */
  line: number; /* span line number */
  start: number; /* span start */
  len: number; /* span length */
  is_storage: boolean; /* storage local rather than reference & type */
  storage_live: number; /* line number of storage live */
  storage_dead: number; /* line number of storage dead */
  move_out: number; /* move of owned data */
  borrowed_local: number; /* storage local that this borrow refers to */
  live_regions: [number, number][]; // array of tuples
}

```

Figure 4.11: Representation of each Local in the webview frontend

html as a string. `main` is selected as the first function to depict by default. Once the new panel is launched, it communicates with the extension via messages.

## Create Graph

Before rendering the graph, colors, grid coordinates and offset intervals to control the spacing left to right of multiple timelines and ensure y coordinates match the vertical spacing of the source code are configured. A message is sent to the extension to request the set of `locals`. With the `locals`, a vertex for the starting point and a path to represent the liveness of each local is constructed. The set of `locals` is supplied from the JSON data written by `compiler_query` and depicted in Figure 4.11.

We chose SVG vector-based graphics, to construct our graph. SVG elements can be directly embedded in HTML pages and are scalable; they do not lose any quality if they are zoomed or resized. Our Graph class contains the elements depicted in Figure 4.12.

On the first pass through the set of `locals`, each storage local is assigned an x value. Storage locals are differentiated from references because their vertex will render at the next available x coordinate. The vertex for reference locals will render at the x coordinate of the referent. However, both types will utilize



```

class Graph {
    src_lines: string[] = [];
    locals: Local[] = [];
    vertices: Vertex[] = [];
    lines: Lifeline[] = [];
    colors: number[] = [];
    ....
}

```

Figure 4.12: Representation of the graph as in the webview panel of RustLIVE

a new lifetime path. The set of vertexes is constructed by iterating on the lines of source code. If a local or multiple locals are defined on the line, a new vertex is pushed with the y coordinate of the source code line and the x coordinate as determined by the first pass. Each vertex is represented with the elements listed in Figure 4.13.

With the set of starting points (vertices) collected, the path for each vertex is determined. Each path contains the elements in Figure 4.14. Each `Lifeline` is associated with its starting point - `Vertex`. If the vertex represents a storage local, the path is set to the same x coordinate as the vertex. Owned variables are represented by solid points at the beginning of their lifetimes, while borrowed variables are depicted with hollow points. Then, the next x path is updated and `endtype` is set based on how the path ends - a move or drop. If the path represents the lifetime of a reference, the set of liveness points is collected from the `locals` data and a set of `lines` to represent liveness is created, line number by line number, if the reference is live at the point. The set of lines is created one segment at a time and then optimized by combining the consecutive segments. With a set of vertices, lifelines and endpoints, the graph is rendered. See Figure 4.15.

```

class Vertex {
    y: number; // the source code line# - the y position
    local: Local;
    x: number;
    lifetimeX: number;
    onLifeline: Lifeline | null = null;
    .....
}

```

Figure 4.13: Representation of the starting point of each local

```

class Lifeline {
    color: number;
    lifetimeX: number;
    end: number = 0;
    endType: EndType;
    lines: Line[] = [];
    ...
}

```

Figure 4.14: Representation of each liveness path

## Create Table

The RustLIVE panel displays the source code in an HTML table on the right side, with each row containing one line of source code. The source code is taken from the active editing window and combined with information about the locations of local variables and regions, obtained from the source span element of the Rust compiler data.

The local variables are first sorted by line number and then by their position within the line. As each line of source code is processed, it is entered as text content into a new HTML table row. If a local variable or region is defined on the current line, a styled span tag and an SVG icon are inserted at the start position of the local variable, and a closing span tag is placed at the end of the local's definition.


Graph	Source Code
	<pre>fn main() {     let mut x = 22; // owned integer x     let mut v = vec![]; // owned vector v     let p = &amp;x; // x is borrowed here to create p     let r = &amp;mut v; // r is a mutable borrow     r.push(p); // p is stored into v, but through r     takev(v); // pass v and x to take }</pre>

Figure 4.15: RustLIVE webview panel. Owned locals x and y are solid points. Borrows r and s are hollow and originate at the referent.

Each processed row is then appended to the table, creating an annotated version of the source code. This method allows for the alignment of each table row with the y-coordinates of the graph, ensuring that the code representation is aligned with the timeline graph.

In the Visual Studio Code (VS Code) workspace, the programmer has the flexibility to compile using any version of the Rust compiler. However, the compiler-query initiated by RustLIVE specifically requires the rustc 1.75.0-nightly version. While a nightly version of the compiler is required to generate the necessary data for RustLIVE, the programmer's flexibility to compile with any version ensures that RustLIVE does not break due to compiler updates.

Moreover, the programmer can switch back and forth between the development window and the RustLIVE panel. Compiler-query does not execute automatically on changes made to the active source code but rather is triggered on user selection of the RustLIVE command. This design choice mitigates the performance impact of the Rust nightly compile process which is quite slow.

### 4.3.2 Backbone- Compiler Query

When the RustLIVE command is issued by the user, data structures are initialized in the IDE, the webview panel is created, the language of the code in the active window is verified to be valid Rust, and the resolve of the asynchronous return from *compiler-query* is awaited. *compiler-query* is executed with cargo run with the Rust nightly compiler 1.75.0 toolchain. The nightly compiler is required for obtaining information from the running compile process. This design feature allows the programmer to compile their code with any version of the Rust compiler. RustLIVE spawns a separate compile instance to gather information from a running compile process.

#### Compiler Callbacks

The RustLIVE compiler-query uses the `rustc_driver` interface to run the compiler. The `mir-opt-level=0` option is set for MIR bodies so that no optimizations are performed and we have the full data set to parse. `rustc_driver` provides callbacks that are executed as specific stages of the compilation process are completed. Four callbacks provided by `rustc_driver` are:

- `config`
- `after_crate_root_parsing`
- `after_expansion`
- `after_analysis`

For RustLIVE to depict the Rust safety tenants of ownership and the liveness of borrows, it needs access to the control flow graph representation known as the MIR (Mid-level Intermediate Representa-

tion) and the facts computed by the borrow checker. These elements are not computed until later in the compile process so can only be accessed by a compiler consumer at the final `after_analysis` callback.

## **Override Borrow Check**

In a normal compile process, the borrow checker runs late in the compile process pipeline, computes the borrow checker facts: regions, loans, loan constraints and liveness points and uses the set of facts calculated to detect violations of safety principles and then drops these facts. They are not normally available to a compiler consumer.

However, the development team behind the Prusti Rust language verifier [51] raised issue #86977 [77] requesting access to the facts computed by the borrow checker for Prusti. In response, the borrow checker development team made available via a Rust compiler consumer query, `get_body_with_borrowck_facts`. The `get_body_with_borrowck_facts` query provides the associated MIR bodies along with the facts computed in the borrow checker and serves as the key to unlocking access to the memory safety constraints calculated and enforced by the Rust compiler and its borrow checker.

However, a `get_body_with_borrowck_facts` query should not be performed in the normal compile process because `get_body_with_borrowck_facts` will panic if body was stolen and ownership was moved before it is invoked. To prevent panic, RustLIVE uses the early callback, `config`, to override the borrow checker query, `mir_borrowck`, with a custom `get_body_with_borrowck_facts` query that stashes the body and associated facts in thread local storage. Figure 4.16 details the provided information from the `get_body_with_borrowck_facts` query.

## **after\_analysis**

With the `after_analysis` callback of the nightly compiler, the set of MIR bodies is retrieved, one for each function, along with the borrow checker facts computed in the override of the borrow checker from the stashed thread local storage location. `body` is the function unit in the Rust MIR; each body contains the set of basic blocks for one function.

```

pub struct BodyWithBorrowckFacts<'tcx> {
    pub body: Body<'tcx>,
    pub promoted: IndexVec<Promoted, Body<'tcx>>,
    pub borrow_set: Rc<BorrowSet<'tcx>>,
    pub region_inference_context: Rc<RegionInferenceContext<'tcx>>,
    pub location_table: Option<LocationTable>,
    pub input_facts: Option<Box<PoloniusInput>>,
    pub output_facts: Option<Rc<PoloniusOutput>>,
}

```

Figure 4.16: Return from `get_body_with_borrowck_facts` includes the MIR intermediate representation in `body` as well as the input and output facts computed by the borrow checker

With the function IDs, `LocalDef` IDs, in the MIR bodies, the high-level intermediate representation, HIR, is queried to obtain the corresponding function name from the `items` of the abstract syntax tree. The function ID and function name are inserted into the `crate_fn_names` `HashMap` and analysis of each MIR function body begins. The task is to identify elements depicted in Figure 4.17 from the `locals` in each body:

## Traversing the MIR

Keys elements for RustLIVE while traversing the MIR are:

- **Body** - the MIR of one function
- **Basic blocks** - basic unit of the control flow graph
- **Statements** - elements of a basic block, executed sequentially
- **Locals** - memory locations, may be local variables, function arguments or compiler temporaries, represented as an underscore and number.

```

crate_locals: HashMap<LocalId, LocalInfo>

pub struct LocalId {
    pub fn_id: LocalDefId,
    pub local: Local,
}
pub struct LocalInfo{
    pub name: String,
    pub local: usize,
    pub line: usize,
    pub start: usize,
    pub len: usize,
    pub is_storage: bool,
    pub storage_live: usize,
    pub storage_dead: usize,
    pub move_out: usize,
    pub borrowed_local: u32,
    pub live_regions: Vec<(usize, usize)>,
}

```

Figure 4.17: Data to obtain for each Local in body

### For each local in the function

For each function, create the set of locals and local info as depicted in Figure 4.17. From the MIR body, obtain the set `local_decls` and iterate on each `local` and `local_decl` to exclude the compiler-generated temporaries and focus solely on tracking user variables. Add these to the set of `storage_locals`. For each `local` remaining, instantiate a `LocalInfo` struct for `name`, `local`, `line`, `start`, `len`, `is_storage`, `storage_live`, `storage_dead`, `move_out`, `borrowed_local` and a vector of `live_regions`.

Several pieces of useful information for RustLIVE are found in `var_debug_info`. See Figure 4.20. Destructuring yields the `source_info` for information about the source code origin. With the `span` element of `source_info`, perform a lookup on the `tcx` to obtain the line number, starting character position and ending character position for the source code location of the `local`. With this information, RustLIVE is able to align points in the graph with lines in the source code as well as annotate locals within the source code table. Both the variable name in the source code and its `local` representation in the MIR are retrieved from `local_decls`.

```
pub struct Body<'tcx> {
    pub basic_blocks: BasicBlocks<'tcx>,
    pub local_decls: IndexVec<Local, LocalDecl<'tcx>>,
    pub var_debug_info: Vec<VarDebugInfo<'tcx>>,
    pub span: Span,
    . . .
}
```

Figure 4.18: MIR Body representation - one per function

```
pub struct BasicBlockData<'tcx> {

    pub statements: Vec<Statement<'tcx>>,
    pub terminator: Option<Terminator<'tcx>>,
    pub is_cleanup: bool,
}
```

Figure 4.19: MIR Basic block representation

```
pub struct VarDebugInfo<'tcx> {
    pub name: Symbol,
    pub source_info: SourceInfo,
    pub composite: Option<Box<VarDebugInfoFragment<'tcx>>>,
    pub value: VarDebugInfoContents<'tcx>,
    pub argument_index: Option<u16>,
}
```

Figure 4.20: The VarDebugInfo member of Body provides useful information for RustLIVE



## MIR location translation

The MIR intermediate representation has a vastly different structure than Rust source code. RustLIVE must correlate MIR locations (see Figure 4.3) to lines in the Rust source code. A map of MIR location (basic block index, statement index) to Rust source code line is created for each function body (MIR), `mir_location_to_line`.

## Statement by statement

While iterating on the basic blocks and statements of the MIR, a match on `StatementKind` detects important timeline events for the timeline graphic. Moves are not detected as a statement type so `assign` statements must be examined to determine potential moves. To detect *moves* - transfer of ownership - `assign` statements are checked for the move of a location. In other words, check for an `Rvalue` of `Place` that matches `local` and an `Operand` of `move`. Matching on `StorageLive` and `StorageDead` can detect the live range of storage locals. `StorageLive` signifies memory is allocated for the local and `StorageDead` triggers freeing the memory. [59]

## Borrow checker

Important information for RustLIVE, the liveness of borrows, cannot be obtained from `mir::Body` but rather needs the input and output facts computed by the borrow checker. The borrow checker is where RustLIVE identifies which `locals` are references, which owned variables they might refer to, and where the references are live.

The borrow checker operates with an even finer location granularity within the MIR body, which is referred to as *rich locations*. Rich locations differentiate between the start and intermediate points of basic block statements. Each rich location is identified by an index. The `location_table` associates each location index to its corresponding rich location.

RustLIVE uses the set of regions constructed by the borrow checker and checks each `local` against this set to obtain which locals are references. Once obtained, this set of references, denoted by `region`

```
let mut region_map: HashMap<Region, Vec<(BasicBlock, Statement)>>;
```

Figure 4.21: `region_map` is created to map each reference (region) to its live MIR locations

number, form the keys that RustLIVE uses to create the `region_map` in Figure 4.21. `region_map` will map each reference, denoted by region number, to a set of MIR (bb, stmt) locations.

The borrow checker also computes a set of output facts. Among the output facts useful for RustLIVE is the liveness information of regions. For every rich location, the borrow checker accumulates all regions that are live at that point. The borrow checker also computes an origin for each reference. Each origin is represented by a numbered region variable[44]. Origins represent where *loans* are issued.

The vector of locations for `region_map` are constructed in RustLIVE by first extracting the set of origin liveness points from the output facts of the current function and sorting them according to the origin (region). For each origin, the `location_index` of the origin serves as the lookup key to translate from an index to a *rich location* with the `location_table` obtained from `get_body_with_borrowck_facts`. RustLIVE extracts the MIR (bb, stmt) location from the *rich location*. With a loaded `region_map`, RustLIVE has a map from each numbered region that represents a reference to a vector of its live MIR locations.

A Rust compiler method `load_place_regions` is employed to prepare a map of local to numbered region, `place_regions`, which RustLIVE uses to create the keys for the `place_regions_live_ranges` map. RustLIVE uses its `mir_location_to_line` lookup to translate the MIR locations in `region_map` to source code line numbers, filters out the macro expansion lines, then combines consecutive line numbers to create a set of ranges which are inserted as the associated values in `place_regions_live_ranges`. See Figure 4.22.

```
place_regions_live_ranges = HashMap<(Local, Region), Vec<(line_number, line_number) >>;
```

Figure 4.22: `place_regions_live_ranges` maps each `Local` and `Region` to a set of live lines.

```
pub struct BorrowSet<'tcx> {
    pub location_map: FxIndexMap<Location, BorrowData<'tcx>>,
    pub activation_map: FxIndexMap<Location, Vec<BorrowIndex>>,
    pub local_map: FxIndexMap<Local, FxIndexSet<BorrowIndex>>,
    pub locals_state_at_exit: LocalsStateAtExit,
}
```

Figure 4.23: `BorrowSet` contains information for identifying the referent of references.

## Referent of references

An additional feature, believed to enhance the effectiveness of RustLIVE, is to depict the place, referent, of the borrows that refer to owned locals. To obtain information on the referent, the data in the `borrow_set` of `BodyWithBorrowckFacts` contains a wealth of useful information. See Figures 4.23 and 4.16

The `BorrowData` in Figure 4.24 from the `location_map` contains the `borrowed_place`. RustLIVE creates a map with keys borrowed locals and values borrowed place. `BorrowData` (Figure 4.24) contains a region id and a borrowed place. With this information, RustLIVE can represent borrows graphically as

```
pub struct BorrowData<'tcx> {
    pub reserve_location: Location,
    pub activation_location: TwoPhaseActivation,
    pub kind: BorrowKind,
    pub region: RegionVid,
    pub borrowed_place: Place<'tcx>,
    pub assigned_place: Place<'tcx>,
}
```

Figure 4.24: `BorrowData` of `location_map` contains `borrowed_place`

an open circle positioned to start at the referent. This visual representation illustrates the specific location to which the borrow refers.

## 4.4 Evaluation

We assessed the effectiveness of RustLIVE among the following three groups:

- **[GRP<sub>1</sub>]** Rust Workshop Participants - Individuals who attended a Rust workshop at a security conference, and are either currently using Rust in their professional roles or plan to use it in the near future.
- **[GRP<sub>2</sub>]** Graduate Students - Computer Science Masters and PhD students who have completed a graduate-level secure programming course taught using Rust.
- **[GRP<sub>3</sub>]** Undergraduate Students - Third and fourth-year undergraduate students in Computer Science who have not received formal instruction in Rust, but have expressed an interest in secure coding and learning the Rust language.

Prior to the tool demonstration and evaluation, the undergraduates were required to watch a video explaining memory safety and the Rust model. Evaluators first examined sample Rust programs in the active code window of VS Code, and, then observed the code and visualization in the RustLIVE panel. The Rust programs used for evaluation are shown in Table 4.1.

We requested the evaluators to provide numerical scores in response to the following questions, using a scale of one to ten, where one represents the lowest and ten represents the highest rating.

- **Q<sub>1</sub>.** How would you rate your understanding of the Rust language and how it achieves memory safety?
- **Q<sub>2</sub>.** Is the visualization provided by the tool helpful to understanding how ownership is enforced in Rust?

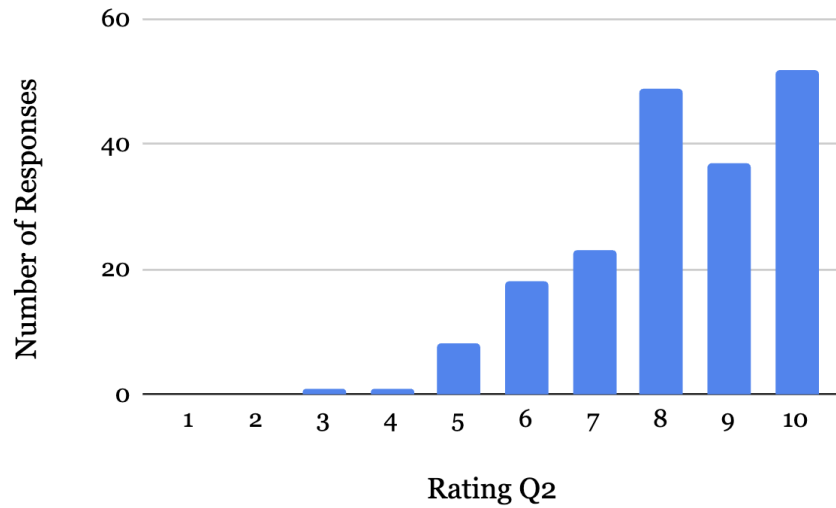


Figure 4.25: User Responses to Question 2 - Effectiveness to understanding Ownership. 73% rank RustLIVE 8 or higher for usefulness to understanding ownership.

- **Q3.** Is the visualization provided by the tool helpful in understanding the lifetime of borrows in Rust?
- **Q4.** Overall, is the tool useful in understanding memory safety in the Rust language?
- **Q5.** Would this tool be useful to developing your Rust programming skills?

We present the evaluation results to Questions 2-5 in Figures 4.25, 4.26, 4.27, and 4.28. The responses to these questions provide insight into how well the tool meets its intended objectives across various user skill levels. Overall, the tool and the usefulness of its visualizations received high ratings.

In order to better understand our pool of evaluators, we examine the responses to Question 1 where respondents were asked to rate their understanding of the Rust programming language and its approach to memory safety on a scale from 1 to 10. Figure 4.29 shows that the distribution of prior Rust knowledge is skewed towards higher ratings with most respondents rating their Rust knowledge between 6 and 9. We use these responses to categorize respondents into their self-assessed skill level in order to gain more meaningful insights from the data. The responses for question 1 were sorted, and quartiles were

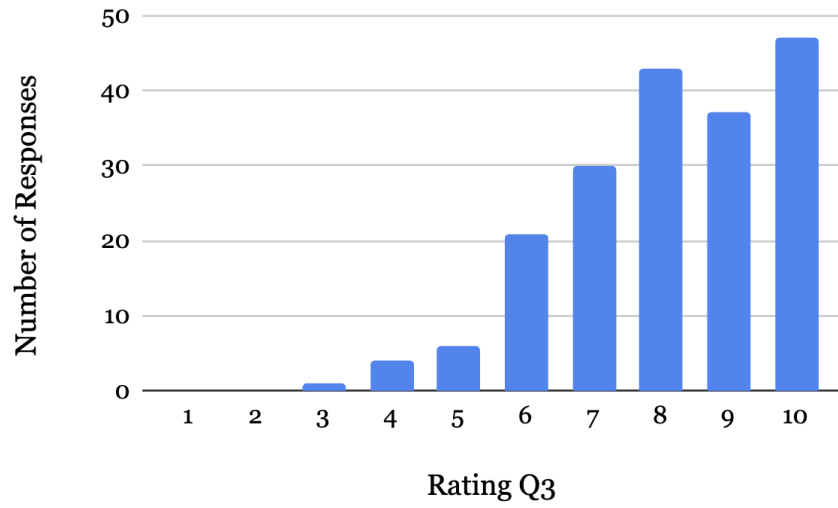


Figure 4.26: User responses to Question 3 - Effectiveness to understanding lifetime of borrows.

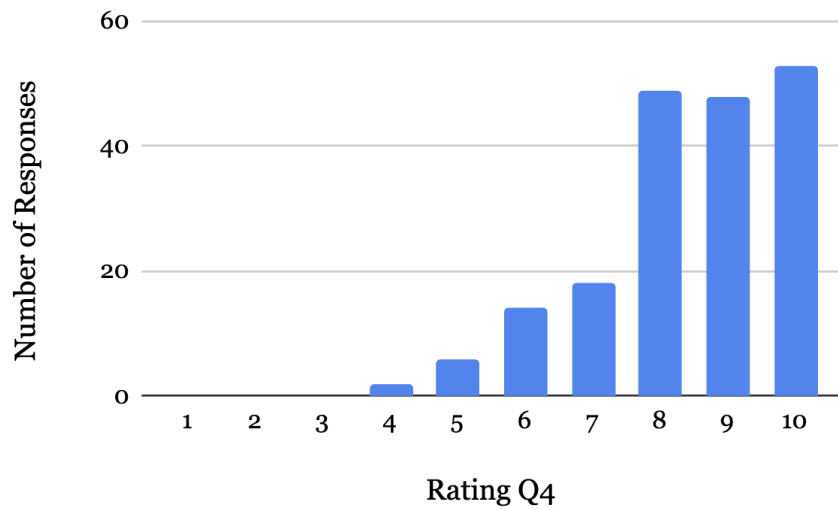


Figure 4.27: User responses to Question 4 - Effectiveness to understanding Rust memory safety. 49 of 190 evaluators rated RustLIVE an 8, 48 rated 9 and 53 gave RustLIVE a 10 for understanding Rust memory safety.

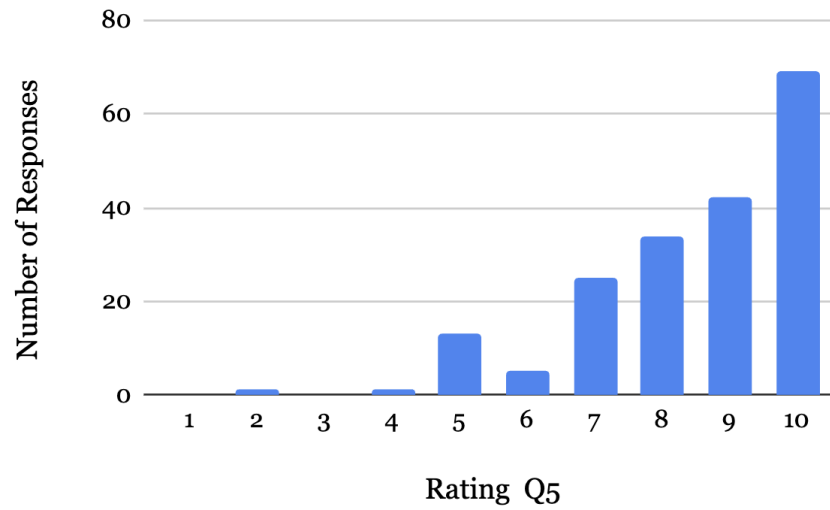


Figure 4.28: User responses to Question 5 - Effectiveness at developing Rust programming skills. 69 of 190 responses ranked RustLIVE a 10, the highest possible score, for skill development.

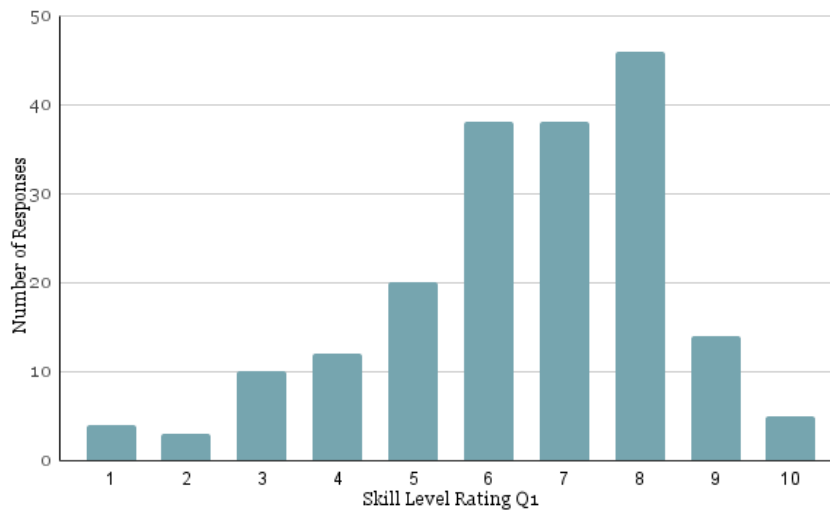


Figure 4.29: Frequency of responses to Question 1: Rate your understanding of memory safety in Rust

calculated to form four equal dataset groups, each representing 25% of the data. Based on the quartiles, the respondents were categorized into three skill levels:

- **Novice:** Respondents with ratings at or below the first quartile ( $Q_1$ ). In this study,  $Q_1$  was determined to be a rating of 5. Therefore, respondents with ratings of 5 or lower were categorized as Novice.
- **Intermediate:** Respondents with ratings between the first quartile ( $Q_1$ ) and the median ( $Q_2$ ). The median rating was 7. Thus, respondents with ratings of 6 or 7 were categorized as Intermediate.
- **Expert:** Respondents with ratings at or above the third quartile ( $Q_3$ ). The third quartile was determined to be a rating of 8. Consequently, respondents with ratings of 8 or higher were categorized as Expert.

By categorizing the respondents into skill level, it is possible to analyze how different levels of understanding influence perceptions of the tool's usefulness. This segmentation provides valuable insights into the effectiveness of the tool across skill levels.

Table 4.30 provides further analysis which reveals the perception and consistency of the tool's effectiveness in helping users understand Rust ownership, borrow lifetimes, memory safety, and skill development. The table details the overall results of each question as well as the results by skill level.

Some key observations from the overall results include:

- **Consistency in responses:** The low standard deviations across most questions reveal consistent opinions among respondents.
- **Visualizations of key concepts ownership and lifetime of references are highly valued:** The visualizations for understanding ownership and borrow lifetimes received particularly high ratings, highlighting their effectiveness. Visualizing ownership and borrowing were goals of the RustLIVE project. These ratings indicate the goals were achieved.



Figure 4.30: RustLIVE evaluation results overall and by skill level.

- **Effective for learning:** The tool is considered useful both for understanding memory safety and for developing Rust programming skills, underscoring its value to teaching memory safety and the Rust language in Computer Science education.

Interesting findings emerged when comparing the scores provided by participants based on their skill level in Figure 4.30. Key findings include:

- **Experts** consistently rate the usefulness of the tool higher compared to the other groups and with lower variability.
- **Intermediates** generally rate the usefulness of the tool positively but slightly lower than experts.
- **Novices** find the tool helpful, but with more variability in their ratings, suggesting a more mixed perception.

Further analysis of feedback from the most experienced users revealed positive comments such as, “I think this tool will really help with learning as well as debugging. What a great tool!”, “I absolutely think this tool does a great job of visualizing a difficult concept in an intuitive way”, and “I believe it will greatly help new devs in the future.” Feedback from inexperienced users was less persuasive, with comments such as, “Rust seems to have a steep learning curve and a visual tool is the perfect way to help overcome that steep curve. My understanding is limited but overall the visual aid helped greatly” and “I don’t understand rust well, but I think this tool does a great job of visualizing ownership and borrows”. Overall, the feedback suggests that individuals with prior Rust coding experience perceive the tool’s value more than those without such experience. The results also indicate that these concepts remain challenging even for experienced Rust programmers. Nonetheless, even participants with no prior Rust experience rated the tool’s usefulness above 7 on average.

With the primary objectives of RustLIVE to aid programmers in developing a mental model of the Rust ownership and lifetime principles, evaluation results confirmed the tool’s effectiveness.

## 4.5 Discussion and Future Work on RustLIVE


RustLIVE, with its innovative design, aims to clarify Rust’s challenging concepts, particularly ownership and borrowing. By utilizing RustLIVE, we anticipate users will develop a correct mental model of Rust’s memory safety principles. Feedback suggests that RustLIVE will facilitate learning and lower the barriers to writing Rust. RustLIVE does not visualize code that does not compile because RustLIVE depends on data that is generated late in the compile process. This is not considered a limitation since the Rust compiler provides descriptive error messages that detail why errors occur.

The initial objective of RustLIVE was to extract information from the Rust compiler to depict the ownership of resources. Additionally, integrating with the borrow checker to illustrate the liveness of references was an aspirational goal. Both were met, along with additional features: annotating resources in the source code table alongside the timeline graph, and references visually originating at their referent with a hollow circle. These features enhance the effectiveness of RustLIVE by correlating the source code with the timeline to connect actions in the source code to their effects on memory resources. Furthermore, RustLIVE successfully obtains compiler computations used to enforce memory safety in Rust. With this data, many additional visual or textual depictions are possible. Careful consideration and collaboration by future developers will ensure that depictions yield maximum benefit to Rust programmers. Some suggestions are included below.

In order to maintain simplicity in the graphic timeline, a visual distinction between immutable and mutable owned variables and borrows was not implemented. This design decision was made with knowledge that the resource would be highlighted in the source code. The `mut` keyword must be included in the definition and thus part of the annotation for any mutable resource in Rust. Future versions could easily incorporate this with a clear design for differentiation. Another idea for future development is to provide tooltips with descriptive information when hovering over points in the graph.

RustLIVE generates a graphic display of one function at a time in the webview panel, defaulting to the `main` function, although other functions can also be depicted. A drop-down menu provides the full list of functions which the user can select from. One evaluator's feedback suggested expanding the visualization to show transitions of ownership across multiple functions. Also, in the current implementation, moves of ownership are depicted with an arrow pointing left; a reviewer suggested that the arrow point to the new owner. However, ownership is often moved to another function.

Table 4.1: Example Rust Programs for Assessment (Part 1).

Ownership example in VS Code	
<pre> src &gt; main.rs &gt; ... 1  /* ownership */    ▶ Run   Debug 2  fn main() { 3      let n_grade: i32 = 100; 4      let grades: Vec&lt;i32&gt; = vec![100, 90, 80, 70, 60]; 5      { 6          let _new_grade: i32 = n_grade; 7          let _new_grades: Vec&lt;i32&gt; = grades; 8      } 9      println!("Grade calculated is: {}", n_grade); 10     // println!("final grades {:?}", grades); 11 } 12 </pre>	
Ownership example in RustLIVE	
Function Graph View: <span>main</span>	
Graph	Source Code
	<pre> /* ownership */ fn main() {     let [n_grade] = 100;     let [grades] = vec![100, 90, 80, 70, 60];     {         let [_new_grade] = n_grade;         let [_new_grades] = grades;     }     println!("Grade calculated is: {}", n_grade);     // println!("final grades {:?}", grades); } </pre>

The timeline in the RustLIVE graph depicts that ownership of `grades` moves to `_new_grades` on the assignment, making the `println` of `grades` invalid.

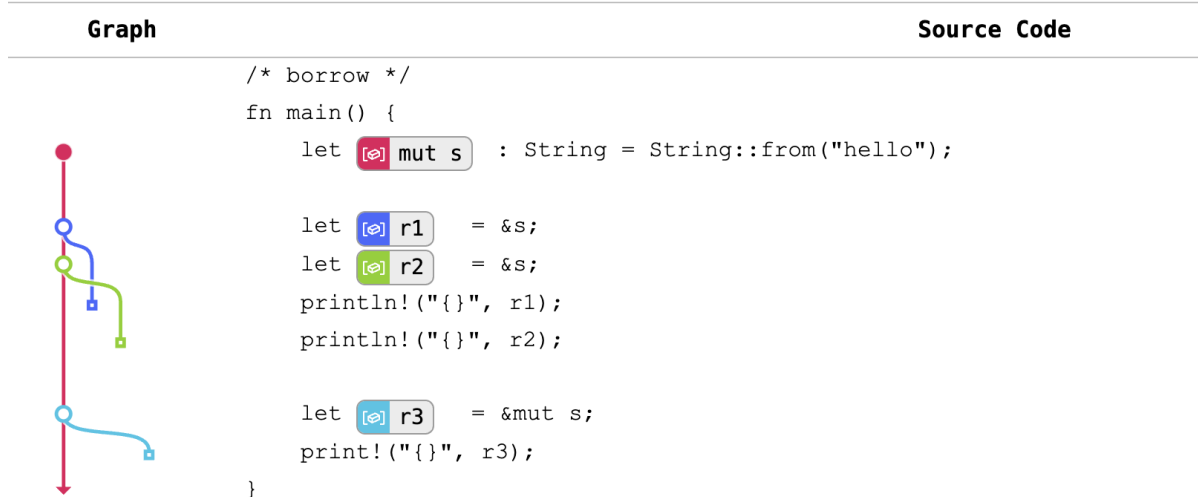
Table 4.2: Example Rust Programs for Assessment (Part 2)

### Lifetime example in VS Code

```
src > main.rs > ...
1  /* borrow */
   ▶ Run | Debug
2  fn main() {
3      let mut s: String = String::from("hello");
4
5      let r1: &String = &s;
6      let r2: &String = &s;
7      println!("{}", r1);
8      println!("{}", r2);
9
10     let r3: &mut String = &mut s;
11     print!("{}", r3);
12 }
```

### Lifetime example in RustLIVE

Function Graph View: main



The timeline in the RustLIVE graph shows two simultaneous immutable borrows of `s` allowed. Their lifetimes end at last use, making the exclusive mutable borrow possible, `r3`. The lifetimes do not outlive the data they refer to, `s`.

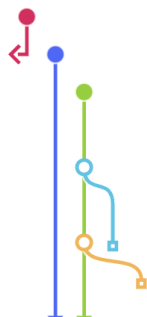
Table 4.3: Example Rust Programs for Assessment (Part 3)

### Memory Safety example in VS Code

```
src > main.rs > ...
1  /* ownership and borrow */
2
3  ▶ Run | Debug
4  fn main() {
5      let s: String = String::from("Hello");
6      let u: String = s;
7      let mut t: String = String::from("world");
8      t.push(ch: '!');
9      let v: &mut String = &mut t;
10     v.push(ch: '!');
11     let w: &String = &*v;
12     println!("{}", w);
13 }
```

### Memory Safety example in RustLIVE

Function Graph View: main

Graph	Source Code
	<pre>fn main() {     let s = String::from("Hello");     let u = s;     let mut t = String::from("world");     t.push('!');     let v = &amp;mut t;     v.push('!');     let w = &amp;*v;     println!("{}", w); }</pre>

This RustLIVE graph illustrates ownership of `s` transferred to `u` on the assignment. `t` is created and mutated before it is mutably borrowed by `v`, mutated exclusively through `v` with non-lexical lifetime ending at last use. The borrow checker then allows the immutable reborrow `w` whose lifetime ends on the `println!` before the value is dropped.

## CHAPTER 5

# CONCLUSION AND FUTURE WORK

This research was sparked by the intrigue surrounding a new programming language that promised safety without compromising speed. As a former systems software developer, the potential of a memory safe language as efficient as C/C++ was the motivating factor to learn more about the Rust language design. As I studied works such as [84] and [52], I realized that the approach was novel but similar to proven protections for shared resources in multi-threaded environments. The works examined the safety issues in Rust programs, how `unsafe` was used, and how the use of `unsafe` undermined the safety of Rust projects and lead to memory safety issues. `unsafe` is necessary in certain situations but often not used securely.

The research presented in *Understanding Vulnerabilities in Rust Applications* of Chapter 3, along with other studies on Rust and usability, indicates that misunderstanding ownership and lifetime rules are the primary cause for memory bugs in Rust code. Several studies have suggested that a visualization of these rules could help Rust programmers write safe code. This insight led to the development of RustLIVE, a tool designed to help users develop a correct mental model of ownership of resources and lifetime of borrows.

Rust represents a promising memory safe language with the speed, safety and fearless concurrency that make it a suitable language for many systems software projects. Writing in Rust forces programmers to learn safe memory practices. The learning curve is steep, but RustLIVE helps to flatten the curve and

the tradeoff is that programmers learn to write code that only has one owner at a time, does not allow a reference to live longer than its value, and automatically drops the value when the owner goes out of scope. At a 2023 Keynote address on the future of C and C++, Robert Seacord concluded, “the first and foremost strategy for reducing security related coding flaws is to educate developers how to avoid creating vulnerable code.” [61] Learning how Rust achieves memory safety and how safe code can be written in Rust teaches programmers how to write code in any language that is memory safe.

Many Rust users and researchers have pointed to the need for visualization of Rust ownership and borrowing. Several attempts are documented in Chapter 2 but none had successfully obtained the actual calculations from the enforcer of memory safety in Rust - the compiler. RustLIVE obtains this information at multiple points in the compile process without causing a panic. The information is presented in a simple, color-coded, intuitive visualization that matches the control flow of the program source code. Evaluations confer the usefulness of RustLIVE.

RustLIVE makes available the entire control-flow graph representation of every function and the entire collection of facts computed by the borrow checker. With this information, any violation of ownership and lifetimes in safe Rust code can be detected and visualized. The difficulty lies in visualizing the large amount of information in a “succinct, non-intrusive, yet informative” manner [11]. The evaluations included some suggestions for future features. My hope is that a select group of stakeholders will carefully evaluate and guide the future development of this tool. Rust brings considerable promise to securing our code bases and RustLIVE can play an important role in the usability of Rust.



# APPENDIX A

Sample Rust code and the JSON data calculated, written, read and visualized by RustLIVE.

```
fn main() {  
    let s = String::from("Hello");  
    let u = s;  
    let mut t = String::from("world");  
    t.push('!');  
    let v = &mut t;  
    v.push('!');  
    let w = &*v;  
    println!("{}", w);  
}  
  
{  
    "main": [  
        {  
            "name": "w",  
            "local": 9, // _9 in MIR
```

```

    "line": 8,                    // source code definition: line #
    "start": 8,                  // character position
    "len": 1,
    "is_storage": false,
    "storage_live": 8,           // line # in source code
    "storage_dead": 10,
    "move_out": 0,
    "borrowed_local": 6,         // referrent local _6
    "live_regions": [[8, 9]]     // regions (line #s) of liveness
},
{
    "name": "v",
    "local": 6,
    "line": 6,
    "start": 8,
    "len": 1,
    "is_storage": false,
    "storage_live": 6,
    "storage_dead": 10,
    "move_out": 0,
    "borrowed_local": 3,
    "live_regions": [[6, 8]]
},
{
    "name": "s",
    "local": 1,

```

```

    "line": 2,
    "start": 8,
    "len": 1,
    "is_storage": true,
    "storage_live": 2,
    "storage_dead": 10,
    "move_out": 3,
    "borrowed_local": 0,
    "live_regions": []
  },
  {
    "name": "u",
    "local": 2,
    "line": 3,
    "start": 8,
    "len": 1,
    "is_storage": true,
    "storage_live": 3,
    "storage_dead": 10,
    "move_out": 0,
    "borrowed_local": 0,
    "live_regions": []
  },
  {
    "name": "t",
    "local": 3,

```

```
"line": 4,  
"start": 8,  
"len": 5,  
"is_storage": true,  
"storage_live": 4,  
"storage_dead": 10,  
"move_out": 0,  
"borrowed_local": 0,  
"live_regions": []  
}  
]  
}
```

# BIBLIOGRAPHY

- [1] *afl*. <https://github.com/rust-fuzz/afl.rs>. 2022.
- [2] *afl.rs: Fuzzing Rust code with American Fuzzy Lop*. <https://github.com/rust-fuzz/afl.rs>. 2022.
- [3] Vytautas Astrauskas et al. “Leveraging Rust types for modular specification and verification”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.
- [4] Yechan Bae et al. “RUDRA: finding memory safety bugs in Rust at the ecosystem scale”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 84–99.
- [5] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. “Verifying Rust programs with SMACK”. In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2018, pp. 528–535.
- [6] David Blaser. “Simple explanation of complex lifetime errors in Rust”. In: *Bachelor Thesis* (2019).
- [7] Kevin Boos et al. “Theseus: an experiment in operating system structure and state management”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 1–19.
- [8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.

- [9] *cargo-fuzz: Command line helpers for fuzzing*. <https://github.com/rust-fuzz/cargo-fuzz>. 2022.
- [10] Catalin Cimpanu. *Microsoft: 70 Percent of All Security Bugs Are Memory Safety Issues*. 2019. URL: <https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues>.
- [11] Will Crichton. “The usability of ownership”. In: *arXiv preprint arXiv:2011.06171* (2020).
- [12] Mohan Cui et al. “SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis”. In: *arXiv preprint arXiv:2103.15420* (2021).
- [13] *CVE-Linux*. [https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor\\_id=33](https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33). 2023.
- [14] Hoang-Hai Dang et al. “RustBelt meets relaxed memory”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–29.
- [15] *Deno: Deploy JavaScript Globally*. <https://deno.com/deploy>. 2022.
- [16] Kyle Dewey, Jared Roesch, and Ben Hardekopf. “Fuzzing the Rust typechecker using CLP (T)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 482–493.
- [17] Dietler Dominik. “Visualization of lifetime constraints in Rust”. In: *Bachelor. Thesis* (2018).
- [18] *Electrum Server in Rust*. <https://github.com/romanz/electrs>. 2022.
- [19] Paul Daniel Faria. “Borrow visualizer for the Rust language service”. In: *Retrieved September 13* (2019), p. 2019.
- [20] Kasra Ferdowsi. “The usability of advanced type systems: Rust as a case study”. In: *arXiv preprint arXiv:2301.02308* (2023).
- [21] Andrea Fioraldi et al. “{AFL++}: Combining Incremental Steps of Fuzzing Research”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. 2020.

- [22] Mozilla Foundation. *Mozilla Firefox*. Accessed: 2024-07-07. 2024. URL: <https://www.mozilla.org/en-US/firefox/new/>.
- [23] Kelsey R Fulton et al. “Benefits and drawbacks of adopting a secure programming language: rust as a case study”. In: *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. 2021, pp. 597–616.
- [24] *Go at Google*. <https://go.dev/talks/2012/splash.article>. 2012.
- [25] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. “Software visualization”. In: *Innovations in Systems and Software Engineering* 1.2 (2005), pp. 221–230.
- [26] Baojian Hua et al. “Rupair: towards automatic buffer overflow detection and rectification for Rust”. In: *Annual Computer Security Applications Conference*. 2021, pp. 812–823.
- [27] *Integer overflow @ swc/ecmascript/parser/src/lexer/number.rs*. <https://github.com/swc-project/swc/issues/1803>. 2022.
- [28] JetBrains. *Developer Ecosystem Survey 2023*. <https://www.jetbrains.com/lp/devecosystem-2023/>. Accessed: 2024-05-13. 2023.
- [29] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. “RULF: Rust library fuzzing via API dependency graph traversal”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 581–592.
- [30] Ralf Jung et al. “RustBelt: Securing the foundations of the Rust programming language”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34.
- [31] Ralf Jung et al. “Stacked borrows: an aliasing model for Rust”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–32.
- [32] Angelo D. Keromytis. *Recommendations from the Workshop on Open-source Software Security Initiative*. <https://cpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/a/2878/files/2022/10/OSSI-Final-Report.pdf>. 2022.

- [33] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.
- [34] Steve Klabnik and Carol Nichols. *The Rust Programming Language, 2nd Edition*. 2018. URL: [https://web.mit.edu/rust-lang\\_v1.25/arch/amd64\\_ubuntu1404/share/doc/rust/html/book/second-edition/ch17-01-what-is-oo.html](https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/second-edition/ch17-01-what-is-oo.html).
- [35] *klee-rs: A safe KLEE API for Rust*. <https://crates.io/crates/klee-rs>. 2022.
- [36] Jessica Lam et al. “Identifying gaps in the secure programming knowledge and skills of students”. In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 2022, pp. 703–709.
- [37] Amit Levy et al. “The tock embedded operating system”. In: *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 2017, pp. 1–2.
- [38] Zhuohua Li et al. “Detecting Cross-language Memory Management Issues in Rust”. In: *European Symposium on Research in Computer Security*. Springer. 2022, pp. 680–700.
- [39] Zhuohua Li et al. “MirChecker: detecting bugs in Rust programs via static analysis”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2183–2196.
- [40] *Lifetimes*. [https://web.mit.edu/rust-lang\\_v1.25/arch/amd64\\_ubuntu1404/share/doc/rust/html/book/first-edition/lifetimes.html](https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/lifetimes.html).
- [41] *lighthouse: Ethereum consensus client in Rust*. <https://github.com/sigp/lighthouse>. 2022.
- [42] Marcus Lindner, Jorge Aparicius, and Per Lindgren. “No panic! Verification of Rust programs by symbolic execution”. In: *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE. 2018, pp. 108–114.
- [43] Gongming Luo et al. “RustViz: Interactively Visualizing Ownership and Borrowing”. In: *arXiv preprint arXiv:2011.09012* (2020).



- [44] Niko Matsakis. *An Alias-Based Formulation of the Borrow Checker*. <https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/>. Accessed: 2024-05-13. Apr. 2018.
- [45] Niko Matsakis. *Polonius: Either Borrower or Lender Be, but Responsibly*. Conference Presentation, Rust Belt Rust Conference. 2019. URL: [https://www.youtube.com/watch?v=\\_agDeiWek8w&t=1864s](https://www.youtube.com/watch?v=_agDeiWek8w&t=1864s).
- [46] Yusuke Matsushita et al. “RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 841–856.
- [47] *National Security Agency | Cybersecurity Information Sheet*. [https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/o/CSI\\_SOFTWARE\\_MEMORY\\_SAFETY.PDF](https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/o/CSI_SOFTWARE_MEMORY_SAFETY.PDF).
- [48] *NSA Releases Guidance on How to Protect Against Software Memory Safety Issues*. <https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/>.
- [49] S Olson. *Miri: an interpreter for Rust’s mid-level intermediate representation*. Tech. rep. Technical report, 2016.
- [50] *OpenEthereum*. <https://github.com/openethereum/openethereum>. 2022.
- [51] Prusti Contributors. *Prusti: Rust verifier*. <https://github.com/viperproject/prusti-dev>. 2021.
- [52] Boqin Qin et al. “Understanding memory and thread safety practices and issues in real-world Rust programs”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 763–779.
- [53] Eric Reed. “Patina: A formalization of the Rust programming language”. In: *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015), p. 264.

- [54] Phil Ruffwind. “Graphical depiction of ownership and borrowing in Rust”. In: *Retrieved September 13 (2017)*, p. 2020.
- [55] *Rust Compiler Development Guide*. <https://rustc-dev-guide.rust-lang.org/query.html#queries-demand-driven-compilation>.
- [56] *Rust Language NLL RFC*. <https://rust-lang.github.io/rfcs/2094-nll.html>.
- [57] *Rust Language RFC 1211*. <https://github.com/rust-lang/rfcs/blob/master/text/1211-mir.md>.
- [58] *Rust Programming Language*. <https://www.rust-lang.org>. 2022.
- [59] *RustCNightlyStatementKind*. [https://doc.rust-lang.org/nightly/nightly-rustc/rustc\\_middle/mir/syntax/enum.StatementKind.html](https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/syntax/enum.StatementKind.html).
- [60] *RUSTSEC-2019-0017: Panic during initialization of Lazy might trigger undefined behavior*. <https://rustsec.org/advisories/RUSTSEC-2019-0017.html>. 2022.
- [61] Robert Seacord. *Keynote: Safety and Security: The Future of C and C++*. Keynote address. 2023. URL: <https://ndcotechtown.com/speakers/robert-seacord>.
- [62] Kostya Serebryany. “libFuzzer—a library for coverage-guided fuzz testing”. In: *LLVM project (2015)*.
- [63] *Servo: The Servo Browser Engine*. <https://github.com/servo/servo>. 2022.
- [64] Nischal Shrestha et al. “Here we go again: Why is it difficult for developers to learn another programming language?” In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 691–701.
- [65] Juha Sorva. “Students’ understandings of storing objects”. In: *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*. 2007, pp. 127–135.

- [66] Juha Sorva. “The same but different students’ understandings of primitive and object variables”. In: *Proceedings of the 8th International Conference on Computing Education Research*. 2008, pp. 5–15.
- [67] Juha Sorva et al. *Visual program simulation in introductory programming education*. Aalto University, 2012.
- [68] Juha Sorva, Ville Karavirta, and Lauri Malmi. “A review of generic program visualization systems for introductory programming education”. In: *ACM Transactions on Computing Education (TOCE)* 13.4 (2013), pp. 1–64.
- [69] *SWC: Rust-based platform for the Web*. <https://crates.io/crates/swc>. 2022.
- [70] Laszlo Szekeres et al. “Sok: Eternal war in memory”. In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 48–62.
- [71] Yoshiki Takashima et al. “Syrust: automatic testing of rust libraries with semantic-aware program synthesis”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 899–913.
- [72] The Rust Team. *Introducing MIR*. <https://blog.rust-lang.org/2016/04/19/MIR.html>. Accessed: 2024-05-13. Apr. 2016.
- [73] The Rust Compiler Team. *Rust Compiler Developer Guide*. Accessed: 2024-07-07. 2024. URL: <https://rustc-dev-guide.rust-lang.org/>.
- [74] The Rust Survey Team. *2023 Rust Annual Survey Results*. <https://blog.rust-lang.org/2024/02/19/2023-Rust-Annual-Survey-2023-results.html>. Accessed: 2024-05-13. Feb. 2024.
- [75] The Chromium Projects. *Memory Safety*. n.d. URL: <https://www.chromium.org/Home/chromium-security/memory-safety>.
- [76] *The Redox Operating System*. <https://www.redox-os.org/>. 2019.

- [77] The Rust Project Developers. *GitHub Pull Request: Fix for issue #86977*. <https://github.com/rust-lang/rust/pull/86977>. Accessed: 2024-06-10. 2021.
- [78] John Toman, Stuart Pernsteiner, and Emina Torlak. “Crust: a bounded verifier for rust (N)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 75–80.
- [79] *Trophy case for afl.rs and cargo-fuzz*. <https://github.com/rust-fuzz/trophy-case>. 2022.
- [80] *Trophy case for FFIChecker*. <https://github.com/lizhuohua/rust-ffi-checker/blob/master/trophy-case/README.md>. 2022.
- [81] *Trophy case for MIRChecker*. <https://github.com/lizhuohua/rust-mir-checker/blob/master/trophy-case/README.md>. 2022.
- [82] *Trophy case for Rudra bugs*. <https://github.com/sslab-gatech/Rudra-PoC>. 2022.
- [83] Jeff Walker. *Rust lifetime visualization ideas*. 2019. URL: <https://blog.adamant-lang.org/2019/rust-lifetime-visualization-ideas/>.
- [84] Hui Xu et al. “Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.1 (2021), pp. 1–25.
- [85] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. “Finding consensus bugs in ethereum via multi-transaction differential fuzzing”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 2021, pp. 349–365.
- [86] Anna Zeng and Will Crichton. “Identifying barriers to adoption for Rust through online discourse”. In: *arXiv preprint arXiv:1901.01001* (2019).
- [87] Ziyi Zhang et al. “VRLifeTime—An IDE Tool to Avoid Concurrency and Memory Bugs in Rust”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 2085–2087.

- [88] Shuofei Zhu et al. “Learning and Programming Challenges of Rust: A Mixed-Methods Study”.  
In: *Proceedings of the 44th International Conference on Software Engineering*. 2022.