

AUTOMATICALLY IMPROVING THE CODE QUALITY OF RUST VIA LLM

A Dissertation
Presented to
The Academic Faculty

By

Xiang Cheng

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Cybersecurity and Privacy
College of Computing

Georgia Institute of Technology

August 2025

© Xiang Cheng 2025

AUTOMATICALLY IMPROVING THE CODE QUALITY OF RUST VIA LLM

Thesis committee:

Dr. Taesoo Kim (Advisor)
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Dr. Xiaokuan Zhang
Department of Computer Science
George Mason University

Dr. Brendan D. Saltaformaggio
School of Cybersecurity and Privacy
Georgia Institute of Technology

Date approved: June 30, 2025

For my parents, Haoqi Cheng and Weihong Lei and my family

ACKNOWLEDGMENTS

I would thank lots of people for their endless support and help for this thesis and my Ph.D journey, below I only list some of them. First and foremost, I would thank Dr. Taesoo Kim and Dr. David Devecsery for advising me during the past few years. Dr. Devecsery taught me how to become a researcher and how to think about problems and ideas. I will always remember the time we worked together on our project and submissions. Dr. Kim taught me how to have a broader view of the research and balance the novelty against the practical usage. I enjoy working with all the SSLab members and won't forget the time we spend together.

I also want to thank to my closest collaborators: Dr. Xiaokuan Zhang, Dr. Sangdon Park, Dr. Yizhuo Zhai for their support in my research. Thanks for their guidance and help for all the submissions and discussions.

Second, I want to thank all my friends and lab members for their support. Thanks to instant messaging for helping me get connected with them even though we are far away. I always chat with Zhuoran Yu, Yibin Yang, Junxian Shen, Haoran Wang, Chuhong Yuan, Yuanbo Li, Xinyu Liu, Fang Wang, Yiqi Chen, Yuhong Wang, Zhufeng Xu and get support from them. And I also want to thank lab members: Fan Sang, Seulbae Kim, Yu-fu Yu, Mingyi Liu, Mingyu Guan, Hanqing Zhao, Fabian Fleischer, Andrew Chin, Sujin Park, Mansour Alharthi, Ammar Askar, Kevin Stevens, Gyejin Lee, Yonghwi Jin, Jalen Chuang, Woosun Song, Jiho Kim, HyungSeok Han, Soyeon Park, Yechan BaeBae, Dae R. Jeong, Cen Zhang, Hang Zhang, Jaewon Hur, Dongkwan Kim, Jaehyuk Lee, and my lab neighbors: Yupeng Yang, Zheng Yang, Bo Lu, Yichang Xiong. Thanks for all the discussions and collaborations we had together.

Third, I want to express my thanks to my girlfriend Xiaohua Shi for her understanding and support in the past few years. We also adopted two cute cats: Shaggy and Lulu, who give me much emotional support.

Finally, I want to give a big thank you to my parents Haoqi Cheng, Weihong Lei, and my family for their endless support and understanding. Without them, I could not finish this long journey, and I will always remember their help. Meanwhile, I also want to thank you—the reader who is reading my thesis. Thanks for reading my research, and hopefully you can find the things you want.

TABLE OF CONTENTS

Acknowledgments	iv
List of Tables	ix
List of Figures	xi
Summary	xiii
Chapter 1: Introduction	1
1.1 Problem Overview	1
1.2 Research Objectives	2
1.3 Thesis Contributions	3
Chapter 2: Background	4
2.1 Undefined Behaviors	4
2.2 Rust Language	5
2.3 Scope Definitions	8
2.4 Large Language Model	11
Chapter 3: RUG: Turbo LLM for Rust Unit Test Generation	13
3.1 Overview	13
3.2 Introduction	14

3.3	Challenges	17
3.4	RUG Design	20
3.5	Evaluation	28
3.6	Limitation	37
Chapter 4: Ruby: Machine Learning-Based Unsafe Detection in Rust Binaries		38
4.1	Overview	38
4.2	Introduction	39
4.3	Threat Model	42
4.4	Unsafe Binary Study	43
4.5	Implementation	50
4.6	Evaluation	58
4.7	Limitation	69
Chapter 5: To Be Safe or Unsafe: Finding Logically Unsafe Rust via LLM		71
5.1	Overview	71
5.2	Introduction	71
5.3	Threat Model	74
5.4	Empirical Study	75
5.5	Implementation	82
5.6	Evaluation	90
5.7	Limitation	99
Chapter 6: Conclusion		100

References 101

LIST OF TABLES

3.1	Coverage comparison between RUG, RustyUnit and SyRust. RUG outperforms RustyUnit by 20.14% and SyRust by 21.57%.	30
3.2	Code coverage evaluation for RUG on 17 popular Rust crates. 'P' indicates the pull request (PR) is still pending for response. The three dimensions for sensitivity tests are: GPT model versions, generation approaches and whether applying fuzzing. The newly API coverage denotes the APIs that baseline failed to test and only covered by RUG.	32
3.3	Robustness evaluation of rug on unlearned crates after the LLM training cut-off. The 'N/A' denotes the crates don't have tests. '*': the RUG with LS strategy achieves 48.98% excluding the no-test crates, close to the 50.64% coverage by developers.	36
4.1	Unsafe operations in Rustc [10] and their distributions in the binary (safe Rust takes the 76.28%). RUBY studies their unique binary representations in section 4.4.	42
4.2	Dataset statistics in the number of functions/records under x86_64 architecture. The RustSecU contains safe/unsafe labels and RustSecB contains bug labels.	52
4.3	The RUBY's final precision and recall for each label under PAC thresholding selection.	60
4.4	Comparison between RUBY and static binary analysis. The static analysis introduces more false-positive cases due to difficulty of identifying code regions from Rust and ELF framework.	60
4.5	Evaluation of RUBY's unsafe classification on large applications. RUBY achieves high AUPRC scores on all applications and even 0.907 on deno.	63

4.6	Angr analysis performance for Rust binaries in different architectures. T0 denotes for timeout and failed to find the bug and N/A denotes for failed to get the buggy binary. RUBY can save 57.95% of time to find the same bug compared with baseline in x64 and 61.4% for ARM binaries. Compared with the oracle baseline including source code, RUBY is only 2.48x compared with oracle method with source code in x64 and 1.19x in ARM.	67
4.7	RUBY provides guided targets for AFLGO. On average, RUBY can save 21.26% of the fuzzing time to find the crash.	68
5.1	The portion of each logically unsafe in COIN’s study.	76
5.2	Evaluation result on finding missed logically unsafe bugs and comparison with other baseline approaches. For the unsafe type, ‘PC’ means ‘Precondition Check’, ‘MI’ stands for ‘Memory Invariants’, ‘DM’ means ‘Device Memory mapping’, ‘FFI’ means ‘calling FFI functions’ and ‘DR’ means ‘data races’. For the status of the bug, ‘P’ indicates the bug has been patched by developers, ‘C’ means the bug is confirmed by developers and ‘W’ represents the report is still waiting for review.	91
5.3	Few-shot (N) examples evaluation for $N = 1, 3, 5$ on GPT-4o and Claude-3.7 with Best@1 settings. The COIN’s fine-tuning significantly improve the model’s ability for identifying logically unsafe Rust.	94
5.4	Best of K examples evaluation on GPT-4o and Claude-3.7 with $N=1$ shot example.	95
5.5	Precision/recall study of COIN. COIN achieves a precision of 57.9%, outperforming Rudra’s 53.3%, MirChecker’s 20.8% and ffi-checker’s 15.3%. ‘-’: recall not reported in the paper.	96

LIST OF FIGURES

2.1	Relationships between oracle undefined behaviors (UB, dashed), ideal unsafe operations (UO, dotted), practical <code>rustc</code> unsafe operations (RC, concrete), and all the program operations (the whole rectangle) in Rust. Note, the size of each area in the figure does <i>not</i> reflect the number of occurrences.	8
3.1	The general workflow and components of RUG. After building the type dependency graph from the input Rust project, RUG leverages bottom-up context building to handle the compilation challenges and fuzzing to resolve the coverage challenges.	20
3.2	RUG’s bottom-up context building example. The left side represents the baseline’s one-shot approach, requiring large language model (LLM) to generate the test with a long context, leading to buggy output. RUG automatically divides the task into subproblems and simplifies the task. The prompt of the RUG is in Figure 3.3 and Figure 3.4.	21
3.3	<Prompt 1>: template RUG used for each sub-problem. The optional paragraph is reusing the previous output to cut the context.	24
3.4	<Prompt 2>: Final test generation template RUG used to combine the sub-solution together and build the target unit test.	26
3.5	Coverage comparison between RUG and RULF, RPG. *: RPG encounters an unexpected crash while running for <code>http crate</code> .	30
4.1	The workflow of RUBY as an unsafe Rust classifier. RUBY takes a Rust binary program as input and outputs a list of unsafe functions, which can be prioritized for further vulnerability analysis.	50
4.2	Precision-recall(AUPRC) evaluation on x64 binaries in CrateU dataset. RUBY achieves relatively high scores on each label and 0.98 on the unsafe detection.	59

4.3	Precision-recall (AUPRC) evaluation on x64 binaries in RustSecU dataset, RUBY achieves 0.98 AUPRC score with a precision 95.21% and recall 91.09%, similar to the CrateU dataset result.	61
4.4	RUBY’s overall performance for assisting reverse engineers to narrow down search space. We fix target recall at 80 % and compare the candidate counts (lower is better).	62
4.5	Precision-recall (AUPRC) evaluation on different compiler versions of unsafe Rust classification. With the help of fine-tuning, RUBY improves the scores to 0.91 for 1.57.0 and 0.92 for 1.75.0.	64
4.6	Precision-recall (AUPRC) evaluation on ARM binaries in CrateU dataset. RUBY achieves high scores as x64, showing its robustness on different architectures.	66
5.1	COIN’s system overview. COIN includes a logically unsafe classifier trained on the existing logically unsafe examples collected from Rust community and a PoC generator trained on a manually created PoC dataset.	83
5.2	Precision-recall (AUPRC) evaluation of QWen2.5-1.5B model and Llama3.2-3B model. The COIN uses Llama3.2-3B model for evaluation.	93
5.3	Precision-recall (AUPRC) evaluation of different models on the logically unsafe classification task. COIN achieves 63.71% precision with 80.42% recall, out-performing other general models.	93

SUMMARY

Ensuring the quality and safety of Rust code is increasingly critical as the language is adopted for system-level and security-sensitive applications. The unique features of Rust, such as its ownership and borrowing system, present both opportunities and challenges for automated code quality improvement. This work addresses these challenges by leveraging LLMs in three key areas: automatic unit test generation, detection of unsafe operations in binaries, and identification of logically unsafe operations that escape compiler checks.

The research introduces a comprehensive framework that integrates semantic-aware static analysis with advanced machine learning techniques tailored for Rust’s complex type system. The first component, RUG, employs a bottom-up context construction strategy and coverage-guided fuzzing to generate high-quality unit tests, achieving coverage rates comparable to human developers. The second component, RUBY, applies machine learning to identify unsafe operations directly in Rust binaries, enabling security analysis even when source code is unavailable. The third component, COIN, uses LLM-based classification and proof-of-concept generation to uncover logically unsafe operations, revealing vulnerabilities that are not detected by the compiler.

Extensive evaluation across thousands of real-world Rust projects demonstrates the effectiveness of these approaches, with significant improvements in code coverage, precision, and recall over existing tools. The results highlight the potential of LLMs, when combined with domain-specific program analysis, to address the unique challenges of Rust and advance the state of automated code quality assurance.

CHAPTER 1

INTRODUCTION

1.1 Problem Overview

Rust is a rapidly growing systems programming language due to its focus on enhancing memory safety [1] and addressing memory safety vulnerabilities [2, 3]. For example, Rust has been employed in system software such as Mozilla Firefox, Google Chrome [4], Linux kernel modules [5], Windows kernel components [6], and various other device drivers [7]. As Rust continues to be adopted for low-level systems and critical software, ensuring memory safety in Rust programs becomes increasingly important.

To achieve memory safety while providing flexibility, Rust is divided into safe and unsafe code regions. Safe Rust is a strongly typed language that ensures memory safety through compile-time checks. However, its strict rules can limit the ability to implement certain features (e.g., resource sharing, low-level assembly) commonly needed in systems programming. To overcome this, unsafe Rust is employed, shifting the responsibility of memory safety checks from the compiler to the developers. In particular, unsafe Rust enables the execution of hazardous operations [8] such as dereferencing pointers or interfacing with external C libraries.

Despite the memory safety mechanism in Rust, more than 360 memory safety bugs have been found in Rust programs in the last five years [9]. The primary reason is that the strict rules of safe Rust often necessitate developers to engage with unsafe Rust, which can lead to memory safety bugs when developers fail to manually verify unsafe regions [10, 11]. For example, cyclic types (e.g., doubly-linked lists) cannot be implemented without resorting to unsafe Rust. Additionally, developers might inadvertently introduce unsafe Rust code when utilizing libraries that contain unsafe regions.

Therefore, the code quality of Rust project is important. However, Rust’s ownership and borrowing system create several code quality challenges that traditional approaches struggle to address. The language’s strict compiler checks, while beneficial for safety, make it difficult to generate valid code automatically. Additionally, Rust’s complex type system and trait relationships create intricate dependencies that are challenging for both static analysis tools and automated testing frameworks to navigate effectively. Because of these unique features and challenges brought by Rust, an automatic way of improving Rust code quality is important to ensure the memory safety and system security of Rust programs.

LLMs have demonstrated remarkable capabilities in understanding programming languages and completing various software engineering tasks. Their ability to process large amounts of code context and generate syntactically correct programs makes them attractive for automated code quality improvement. However, recent studies have shown that LLMs exhibit differential performance across programming languages, with languages like Rust showing significantly lower success rates compared to more commonly represented languages like Python [12]. Therefore, the research objectives of this work are tackling the problem of code quality in Rust through LLMs. Specifically, we show different approaches including prompt engineering, chain-of-thought and fine-tuning to improve LLMs’ performance on Rust program reasoning and leverage the smarter LLM to address potential problems in Rust development.

1.2 Research Objectives

In this thesis, the research objective is to define and resolve the challenges of leveraging LLM to automatically improve Rust’s code quality. The application of LLMs to Rust code quality improvement requires addressing fundamental challenges in three key areas: generating compilable code that satisfies Rust’s strict type system, detecting subtle safety violations that escape traditional analysis, and creating comprehensive test suites that achieve meaningful code coverage. These challenges necessitate novel approaches that combine LLMs with

program analysis techniques specifically designed for Rust’s unique characteristics.

1.3 Thesis Contributions

This thesis makes several key contributions to automatically improving Rust code quality through LLM-based approaches. First, we present RUG, a novel system for automatic unit test generation that addresses Rust’s compilation challenges through semantic-aware bottom-up context construction and achieves high code coverage through integration with coverage-guided fuzzing. Second, we introduce RUBY, a machine learning-based approach for detecting unsafe operations in Rust binaries, enabling security analysis of compiled programs where source code is unavailable. Third, we propose COIN, a system for identifying logically unsafe operations in Rust source code that combines empirical analysis with LLM-based classification and proof-of-concept generation.

Each contribution addresses a different aspect of Rust code quality: testing completeness, binary security analysis, and source-level safety verification. Together, these systems provide a comprehensive framework for leveraging LLMs to improve Rust code quality across the entire software development lifecycle.

CHAPTER 2

BACKGROUND

2.1 Undefined Behaviors

Undefined behaviors are the result of executing a program whose behavior is prescribed to be unpredictable, in the language specification to which the computer code adheres [13]. In Rust, common memory safety issues—including buffer overflows, use after free, and data races—are classified as undefined behavior [14]. The definition is as follows:

Definition 1 (Undefined Behaviors in Rust). *Rust considers the following behaviors as undefined: data races, producing an invalid value, accessing unaligned pointers, breaking aliasing rules, mutating immutable values, etc.*

Safe Rust is designed to prevent these undefined behaviors by adding necessary checks and handlers around the unsafe Rust, which contains the potential unsafe operations [8]. These unsafe operations are statically detectable by `rustc` and will be reported to developers. However, since there is not a formal model of Rust, the definition of Rust’s undefined behaviors is not complete [14].

Locations. Undefined behaviors can have two types of locations: the root cause location and the actual detection location. The root cause location is where the undefined behavior originates, while the actual detection location is where this behavior is observed by engineers. Typically, these locations coincide, as in cases like dereferencing an invalidated pointer. However, there are instances, such as a data race resulting in a dirty read, where these locations may be significantly separated in the binary program. According to unsafe Rust’s definition [15], it includes all operations that could initiate undefined behaviors, encompassing all potential root cause sites within undefined behaviors. Thus, by identifying regions of unsafe Rust, malicious users could pinpoint potential initiation sites for bugs in a

Rust binary, using this information for deeper analysis.

Memory Safety Errors. Memory safety the state of being protected from various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers [16]. The operations violate this protection is regarded as memory safety errors, defined as below:

Definition 2 (Memory safety errors). *Memory safety errors include: access errors, uninitialized variables, memory leak.*

These memory safety errors are closely related to the software security since most of these errors like use-after-free, buffer overflow are exploitable vulnerabilities. Among these errors, access errors, uninitialized variables, use-after-free can directly help malicious users hijack the program, leading to serious consequences. Therefore, searching for memory safety errors is important for reverse engineers to ensure the safety of the program.

On the other hand, these memory safety errors (excluding the memory leaks) are considered undefined behaviors in Rust [13], which are guaranteed to be encompassed by unsafe Rust, as unsafe Rust is defined as the superset of undefined behaviors. Rust consider almost all these memory safety errors except memory leaks as unsafe behaviors and is designed to eliminate these errors through compiler checks.

2.2 Rust Language

Rust is an emerging programming language for low-level and system development with memory safety guarantees. It has two parts; safe Rust is designed to achieve native performance in a memory-safe way guarded by the compiler, and unsafe Rust requires developers' help for memory safety. (Safe) Rust employs a robust type system that imposes strict disciplines, effectively mitigating security concerns and ensuring memory safety. In addition to its advanced type system, Rust supports traits to define the common behaviors and generic parameters to extend its usability. Next, we will describe some necessary language features

and concepts used in the latter context and highlight the challenges they present.

Type Checks. Rust is a statically typed language, all the variables are assigned a specific type at the compilation time. Rustc compiler will reason and check the type correctness and bounds for every variables in the program. As a result, the unit test generation tool needs to ensure the correctness of variable types and bounds.

Traits and bounds. To further empower the flexibility, Rust supports generic parameters to allow developers reuse the functions. Trait bounds are used as desired restrictions for the generic parameters. Therefore, when synthesising programs for generic functions, synthesizers are expected to find qualified candidate types as generic parameters, otherwise the Rustc will find the missing or mismatch of generic parameters.

The aforementioned features along with other compiler checks(e.g. mutability, lifetime, etc.) introduce new challenges for test statement construction or program synthesis in Rust. For instance, the recombination operator in a genetic algorithm takes the parent programs(e.g., program A, B) as inputs and 'breeds' them to generate child programs. In Java or C/C++ language, with the help of the type analysis, synthesizers generate a program with cross-usage of variables (a variable is created in A and used as a parameter in B). However, this step does not always hold in Rust because, except type correctness, the variable's ownership, mutability, and lifetime need to be correct as well. We argue that managing these requirements simultaneously is not trivial under the constraints of safe Rust. Furthermore, since Rust performs these checks during compile time, it is difficult to bypass these challenges.

2.2.1 Safe Rust

Rust language [17] provides a memory-safety guarantee during compile time, while allowing control over low-level access to resources. The claimed memory-safety guarantee is conditionally proved in [18, 19] under some assumptions on language semantics. In a nutshell, safe Rust achieves its memory safety guarantees by adhering to the following goal:

Definition 3 (Safe Rust Goal). *A safe Rust program should never cause undefined behaviors.*

Here, we describe the key concepts to achieve memory safety: *ownership* and *borrowing*. By applying these two concepts of static analysis, safe Rust guarantees that there are no undefined behaviors in compiled programs [18].

Ownership is a relation between a value and a variable; each value in Rust is owned by *only one* variable, and memory associated with the value is automatically freed if the variable goes out of scope. This simple memory management mechanism provides a compile-time memory safety without having a run-time garbage collector. In particular, since a value is automatically freed via `drop()`, added by a compiler, memory leakage due to programmers' mistakes is naturally avoided. Moreover, a value, which can be a pointer to heap memory, is owned by only one variable; thus, its associated memory is freed only via `drop()`, by which double-free bugs are avoided. While ownership serves as an important safeguard against memory-safety bugs, requiring each value to have only a single owner can be overly restrictive. Thus, borrowing is introduced to address this issue.

Borrowing allows having a reference to a variable without ownership. Specifically, Rust provides two types of borrowing: *immutable* and *mutable*. Each variable can have either multiple immutable references or only one mutable reference. Through this restriction, safe Rust ensures that no other party will have write access to the variable when it has been borrowed as a mutable reference.

2.2.2 Unsafe Rust

Although safe Rust provides a strong guarantee on memory safety and relatively flexible restrictions, there are many cases in which programmers need to maintain a shared mutable reference in system programming. For example, memory can be shared among multiple threads with well-defined synchronization. Also, reference counting is widely used under system memory management. To support such cases, unsafe Rust is introduced to escape from the `rustc`'s check inside safe regions and requires programmers to ensure the memory

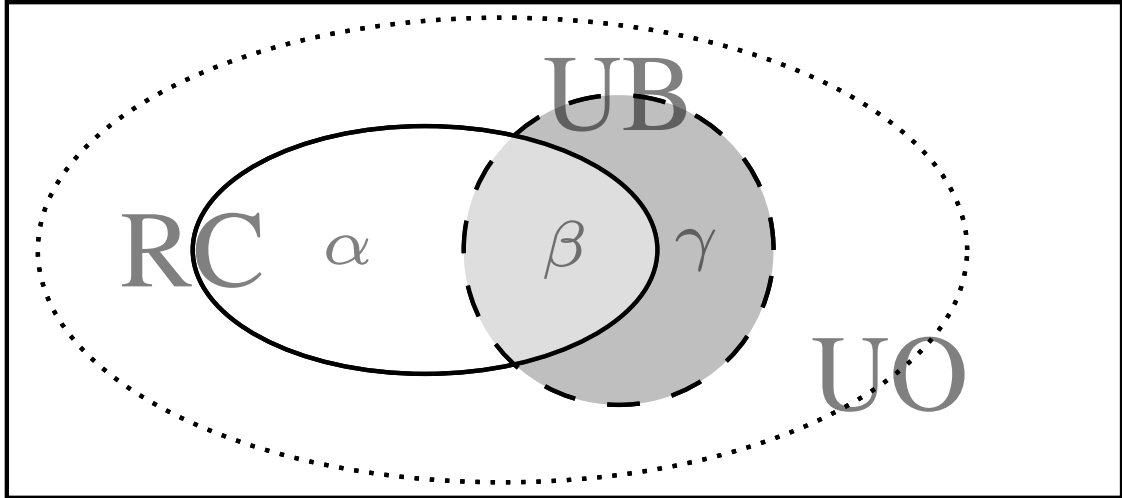


Figure 2.1: Relationships between oracle undefined behaviors (UB, dashed), ideal unsafe operations (UO, dotted), practical rustc unsafe operations (RC, concrete), and all the program operations (the whole rectangle) in Rust. Note, the size of each area in the figure does *not* reflect the number of occurrences.

safety inside the unsafe regions. In particular, rustc defines five operations as unsafe operations [8], which help programmers to identify unsafe regions and ensure memory safety.

Definition 4 (Unsafe Rust Operations). *Rust considers the following operations as unsafe: dereferencing a raw pointer, calling an unsafe function or method, accessing or modifying a mutable static variable, implementing an unsafe trait, or accessing fields of unions.*

As these memory-related operations within unsafe regions are not checked by the compiler for memory safety, they can cause memory-safety bugs.

2.3 Scope Definitions

Based on the above definitions of undefined behaviors, memory safety errors, safe Rust, and unsafe Rust, we demonstrate their relationships and scopes in Figure 2.1, with examples.

The whole rectangle represents all the program operations in the input binary. Namely, given a Rust program, we assume that all operations form a rectangle. Among these operations, those that cause undefined behaviors are in a subset of the whole program. We

```

1 static mut COUNTER: i32;
2 static mut CT_LOCK: libc::pthread_mutex;
3 fn read_counter() -> i32{
4     libc::pthread_mutex_lock(&CT_LOCK); // lock
5     answer = COUNTER; //access a global mutable
6     libc::pthread_mutex_unlock(&CT_LOCK);
7 }

```

Listing 2.1: Case α : accessing a mutable global variable via implicit lock protection. The operation is recognized as unsafe but within the proper lock protection no undefined behaviors will be triggered.

note that the relative size of the area does *not* represent the number of occurrences. For runtime events, we project them as the relevant program operations that trigger these events on the graph.

Based on Definition 3, ideally, Rust wants to define a precise set of unsafe operations as unsafe Rust, so that safe Rust never triggers undefined behaviors. The ideal set is exactly the same as UB shown in Figure 2.1, so that unsafe Rust includes all the unsafe operations while it does not introduce any false-positive cases. However, since undefined behaviors are runtime events, which are difficult to be statically reasoned (as hard as an NP problem [20]), it is difficult for `rustc` to precisely analyze them. Thus, the second-ideal case is to detect a superset of unsafe operations, shown as UO, which introduces some false-positive cases but ensures all the unsafe operations are included. But the practical implementation of `rustc`'s unsafe operations is shown as RC, which includes most undefined behaviors, but also introduces both false-positive and false-negative cases. In this work, we argue that this inconsistency between the ideal definition and practical implementation is the root cause of these logically unsafe operations, which are shown as area γ in the figure. Below, we illustrate each region and provide corresponding examples.

Area α . α represents the cases that are identified as UO by `rustc`, but do not trigger any undefined behaviors or memory safety errors (false positive cases of `rustc` analysis). Due to the runtime feature of UB, it is inevitable to have false positive cases when statically reasoning the program. One example is shown in Listing 2.1, where the lock protects all the

accesses to the mutable global variables. The lock ensures that there will be no data races, and during runtime, memory access will not trigger undefined behaviors.

Area β . In the ideal case, unsafe operations should include all the memory safety errors and undefined behaviors, indicating `rustc` can help developers check all the potential errors. The area β denotes this ideal case in which the `rustc` can help developers check the memory safety of their program. As the motivation of the Rust language, there are many cases, such as dereferencing a pointer and accessing global mutable variables, that will be identified by `rustc` and explicitly asked to have unsafe labels. Area β represents the benefits of using Rust in programming, including the common memory safety errors like invalid access, uninitialized memory read, etc.

```
1 static UNIT: &'static &'static () = &&();
2 fn foo<'a, 'b, T>(_:&'a &'b (), v:&'b T)
3     -> &'a T {v}
4 fn bad<'a, T>(x: &'a T) -> &'static T {
5     let f: fn(_, &'a T) -> &'static T = foo;
6     f(UNIT, x) // a static lifetime reference
7 }           // generated from local lifetime a,b
```

Listing 2.2: Case γ : Rust Issue-25860, a special logically unsafe that bypasses compiler check and triggers an undefined behavior.

Area γ . Area γ denotes the unsafe operations that can evade the compilation error detection of state-of-the-art `rustc`, which is also the target of our work. We name them logically unsafe operations, denoting the difficulties in understanding the logic of the program. Although it is not as common as normal unsafe operations, there are different types of such logically unsafe operations. The famous Rust Issue-25860 shown in Listing 2.2 also belongs to this category. This issue produces longer lifetime references by combining variance and implied bounds for nested references. Issue-25860 has persisted for approximately nine years, allowing malicious users to create seemingly safe crates that trigger undefined behaviors (e.g., a fake-static crate is deliberately crafted by developers to garner attention [21]). The root cause of this issue lies in the undecidability of lifetime analysis, and

resolving it is beyond our scope—we mention it here simply to illustrate the existence of γ area. Crucially, our approach identifies more general bugs that extend beyond this particular issue.

2.4 Large Language Model

LLMs in Software Engineering. Large Language Models have revolutionized software engineering by demonstrating remarkable capabilities in code understanding, generation, and analysis . Models such as CodeX and CodeT5+ have shown tremendous promise in achieving code intelligence across various tasks including program synthesis, bug detection, and test generation. These advances have opened new possibilities for automating software engineering tasks that were previously difficult or impossible to address with traditional approaches.

The success of LLMs in software engineering stems from their ability to process large amounts of code context and learn patterns from vast codebases . Unlike traditional rule-based or statistical approaches, LLMs can capture complex relationships between code elements and generate contextually appropriate solutions. This capability is particularly valuable for tasks that require understanding both syntactic and semantic aspects of code.

However, LLMs also exhibit limitations and biases that affect their performance in software engineering applications . The quality and quantity of training data significantly influence model performance, with languages like Rust being underrepresented compared to more popular languages like Python and JavaScript. This underrepresentation leads to differential performance across programming languages, with Rust showing significantly lower success rates in standard benchmarks.

LLMs in Program Analysis. The application of LLMs to program analysis and bug detection has shown promising results, but also revealed significant challenges . While LLMs can identify many types of bugs and generate test cases, their effectiveness varies significantly across different programming languages and problem domains. The non-

deterministic nature of LLM outputs and their tendency to generate plausible but incorrect solutions create additional challenges for safety-critical applications.

Test generation using LLMs has emerged as a particularly active area of research, with studies showing that LLMs can generate functional test cases that improve code coverage . However, the effectiveness of LLM-generated tests depends heavily on the quality of the prompt, the amount of context provided, and the specific characteristics of the target programming language. For languages with strict compilation requirements like Rust, generating valid test cases becomes significantly more challenging.

The integration of LLMs with traditional program analysis techniques has shown promise for addressing some of these limitations . By combining the contextual understanding capabilities of LLMs with the precision of static analysis tools, hybrid approaches can achieve better results than either technique alone. This integration is particularly important for Rust, where the complex type system and safety requirements necessitate a deep understanding of both syntactic and semantic program properties.

CHAPTER 3

RUG: TURBO LLM FOR RUST UNIT TEST GENERATION

3.1 Overview

Unit testing improves software quality by evaluating isolated sections of the program. This approach alleviates the need for comprehensive program-wide testing and confines the potential error scope within the software. However, unit test development is time-consuming, requiring developers to create appropriate test contexts and determine input values to cover different code regions. This problem is particularly pronounced in Rust due to its intricate type system, making traditional unit test generation tools ineffective in Rust projects. Recently, LLMs have demonstrated their proficiency in understanding programming language and completing software engineering tasks. However, merely prompting LLMs with a basic prompt like "generate unit test for the following source code" often results in code with compilation errors. In addition, LLM-generated unit tests often have limited test coverage.

To bridge this gap and harness the capabilities of LLM, we design and implement RUG, an end-to-end solution to automatically generate the unit test for Rust projects. To help LLM's generated test pass Rust strict compilation checks, RUG designs a semantic-aware bottom-up approach to divide the context construction problem into dependent sub-problems. It solves these sub-problems sequentially using an LLM and merges them to a complete context. To increase test coverage, RUG integrates coverage-guided fuzzing with LLM to prepare fuzzing harnesses. Applying RUG on 17 real-world Rust programs (average 24,937 LoC), we show that RUG can achieve a high code coverage, up to 71.37%, closely comparable to human effort (73.18%). We submitted 113 unit tests generated by RUG covering the new code: 53 of them have been accepted, 17 rejected, and 43 are pending for

review.

3.2 Introduction

Unit testing is essential to ensure program quality throughout the development process, with the aim of comprehensively testing a function in all possible branches and execution paths. It has become an integral part of the software development cycle, and many companies such as Google and Meta have a strict coverage requirement [22, 23]. A good unit test includes the calling context to successfully execute the function, the proper input triggering different paths, and clear assertions reflecting the correctness of the result. Consequently, it is widely recognized that the production of high-quality unit tests requires substantial human energy and effort. For example, in the United States, software testing labor is estimated to cost \$48 billion dollars per year [24].

To reduce developers' workload, several approaches have been proposed to automate test generation, including 1) traditional approaches like Search Based Software Testing (SBST) [25], fuzzing [26], program synthesis [27] and 2) latest Large Language Model (LLM)-based methods [28, 29]. Traditional approaches usually leverage program analysis to build the testing context and search for appropriate test input to trigger different paths. These tools achieve good testing coverage on programming languages such as Java (*e.g.*, EvoSuite[30]), Python (*e.g.*, Pynguin[31]) and C/C++ (*e.g.*, AFL[26]). Recently, many LLM-based tools such as CODAMOSA [32], ChatUniTest [29], TestGen-LLM [33] have been proposed to automate testing generation. Commercial products like Cody [34] and Copilot [35] are available for daily development.

Rust, an emerging system programming language that performs strict compilation checks, is gaining traction for its performance and memory safety advancements. It is adopted in crucial projects such as operating system kernels[5, 6], device drivers[7], web browsers[4], etc. Rust also has a plethora of over 130K packages to date. However, Rust packages are not well tested. We conducted a survey on crates.io, Rust's package repository, computing

the unit tests’ code coverage for the top 30K most downloaded crates¹. From our study, we found that many crates are rarely tested, and 47.41% of the crates have less than 30% test coverage, which underscores the serious problem *lack of unit testing* in Rust projects.

The difficulty of Rust testing is due to its complex type systems. For example, Rust’s ownership system defines the single owner of each memory value, and the borrow checker ensures that each value should only have one writable reference at a time. These language features impose extreme stringency in compiler checks, making it challenging even for experienced developers to craft valid code that passes compilation [36]. When applying existing approaches in Rust, traditional approaches such as SBST and fuzzing suffer from complex type dependencies and the potentially huge searching space, leading to limited test cases and test coverage. For LLM based approaches, Rust’s strict compiler checks and complex program conditions largely undermine the ability of LLMs to generate valid testing code with high code coverage. This underscores the necessity of developing an automated tool specifically for the Rust programming language and pinpoints two key challenges that must be addressed: **A) Passing compilation checks.** and **B) Expanding code coverage.**

We propose RUG², which leverages LLM to automatically generate compilable high coverage unit tests for Rust projects. To solve the challenge **A**, RUG proposes a bottom-up approach to divide the context construction problem into dependent sub-problems and iteratively interact with LLM to solve each sub-problem. Each subsolution will be verified and memorized from the bottom of the dependency graph to the top. The sub-solutions will be merged at the end to generate the final test context. Specifically, RUG first defines the context construction problem as a graph searching problem on the type dependency graph of the Rust project, then leverages static analysis to divide the graph into sub-graphs. For each sub-graph problem, RUG asks LLM for a potential solution and verifies the generated code through a compiler oracle. Based on the verified program, RUG dynamically simplifies the graph by replacing the sub-graph with a single node and reuse it as a hint for its dependent

¹In Rust, packages are called crates.

²RUG is available at <https://github.com/CXWorks/rug>.

problems. Through this iterative approach, RUG simplifies the graph using verified solutions and asks LLM to generate the final test based on the simplified context. For challenge **B**, RUG transforms the generated tests into fuzzing harnesses without breaking the test body and leverages fuzzing to improve the test coverage. Regarding the fuzzing corpus, RUG prompts LLM to prepare sample test data during context generation and reuses it as initial corpora for the fuzzing process.

We evaluated RUG on the 17 most frequently downloaded crates of Rust (average 24,937 LoC). The results show that RUG generates high-quality tests with better code coverage. Using the latest GPT-4 model, RUG achieves 71.37% code coverage, which is comparable to human practice (73.18%), and even achieves higher test coverage than human practice on 8 crates. We submitted 113 unit tests generated by RUG containing code regions that developers failed to test as PRs, and 53 of them have been merged into the project, taking 75.14% of all the tests reviewed.

Contributions. RUG makes the following contributions³:

- **A Bottom-up Context Building Algorithm.** We propose a semantic-aware bottom-up algorithm which simplifies the context construction problem for LLM and improves the correctness of generated code by 65.14%.
- **Fuzzing based Input Exploration.** We demonstrate that fuzzing tools can be used to compensate for LLM’s weakness in reasoning program conditions and expand testing coverage by 6.26% to 8.91%.
- **An automatic unit test generation tool for Rust.** We present RUG, an automatic unit test generation tool for Rust that leverages a deep combination of LLM and program analysis. RUG improves the code coverage of 13.21% to 29.68%, and 11.20% to 21.81% when compared to existing traditional approaches and existing LLM-based approaches, respectively.
- **Practical Usage.** We submitted the 113 generated unit tests to 12 different popular Rust

³RUG is available at <https://github.com/CXWorks/rug>.

projects, 53 of them are merged by maintainers with positive feedback, taking the 75.14% of all the reviewed unit tests.

3.3 Challenges

In this section, we use a motivating example in Listing 3.1 to showcase the challenges to automatically generate unit tests. Based on the root causes of these challenges, we categorize them into two classes and propose our solutions in section 3.4.

3.3.1 Motivating Example

The motivating example in Listing 3.1 shows a target testing function `encode` in line 1 and the related data structures in the `bincode` crate [37]. The function `encode` takes the first argument `self` as a `char` type, serializes it from memory into raw bytes and stores them in the specified location (specified in `encoder`). In order to test the target function, developers need to prepare two concrete variables: one is a simple variable with `char` type and the other is a complex instance of `E: Encoder` trait.

In this code snippet, `EncoderImpl` is a candidate implementation of `Encode`. However, `EncoderImpl` itself depends on two other traits: `W: Writer` and `C: Config`, representing any general `Writer` or `Config`. `Writer` controls the output location of the raw bytes, and `Config` controls the way to encode the memory object (*e.g.*, big endian or little endian). In lines 7-8, we can resolve a concrete instance of `Writer` named `IoWriter` or `SliceWriter`. As for the `Config` trait, instead of providing a direct definition, developers need to deduce that the 'proxy' definition of `Config` on line 10 implies that any type `T` satisfying the union of R_n traits can be an implementation of `Config`. Thus, RUG needs to search across the source codebase for the intersections of candidates implementing R_n trait, which is `Configuration`. After getting a valid `Writer` and `Config`, a valid testing context is built for the `encode` function and ready to test.

Generating a unit test for function `encode` highlights the two previous challenges. First,

Listing 3.1: Motivating examples for encode function. Details of struct/trait definitions are omitted. The challenge comes from lines 10-12; which provides an implicit definition of Config trait.

```
1 fn encode<E: Encoder> (&self :char, encoder: E)
2     -> Result<EncodeError> // target function
3 // impl for Encoder trait
4 impl<W:Writer, C:Config> Encoder for EncoderImpl
5 pub struct EncoderImpl<W: Writer, C: Config>
6 // impls for Writer Trait
7 impl Writer for SliceWriter
8 impl Writer for IoWriter
9 // proxy impls for Config Trait
10 impl<T> Config for T where T: R1 + R2 + R3
11 // def for Configuration, impls R1, R2, R3
12 pub struct Configuration<R1, R2, R3>
```

to pass the compiler check, the generation tool needs to infer the correct instance of the trait based on the function declaration, ensuring that the generated code adheres to the complex compiler rules. Through our evaluation on GPT-3.5 and GPT-4, even if all relevant source code, rustdoc, and sample code are presented in the prompt, it is still difficult for LLM to generate the correct tests. Second, to enhance the coverage of the code, the generated code should encompass various regions within the function. Even if the test generation step succeeds, both the LLM and SBST approaches face challenges in reasoning about the conditions, and thus cannot effectively improve code coverage. By analyzing the generated tests and their failure reasons, we identified the root causes of these two challenges and classified them by their sources as: the challenges caused by Rust and the challenges caused by the LLM models.

3.3.2 Challenges to Pass the Compiler Check

The Rust compiler enforces strict checks for all variables, making it challenging even for humans to write Rust code, let alone automatic tools. Two common mistakes that often require significant time for human engineers to diagnose and resolve are trait errors and path errors.

Traits and bounds. Trait is extensively used in Rust to define the shared behaviors across different structs, while trait bound serves as the restrictions of available generic parameters that can be used in the Rust library. For example, in Listing 3.1, the E: Encoder indicates the generic parameter E must satisfy the Encoder bound. Therefore, to generate a concrete test, it is vital that traits and trait bounds are filled with appropriate instances⁴. In the motivating example, the second parameter encoder needs to be an instance of Encoder trait, which further relies on the Writer trait and Config trait. Moreover, the Config trait only has a proxy definition shown in line 10 in Listing 3.1. Therefore, these complex compositions and proxies of trait & trait bounds become the burdens for LLM to correctly reason the proper candidates, which lead to compilation errors of type mismatch [38]. Besides, Rust’s separation of data definition and implementations makes it more difficult for LLMs to identify correct type information from the mixed source code.

Definition paths. To ensure successful compilation, all type and function definitions within the testing program must be explicitly imported into the context. This is hard for the purely LLM-based approach. In the motivating example, the target test function encode is part of the Encode trait, which is implemented for the char type as the self parameter. Hence, to trigger the encode function for char, the Encode trait must be explicitly imported into the testing context.

Cascading Errors. The unit test composition process involves several steps. First, we need to figure out the calling context, *e.g.*, reasonable input data for each variable. Second, we need to properly pass those contexts to the target function. Finally, we need to resolve the correct assertion statement. When prompting LLM through this process, it requires a complicated reasoning chain from the model. Even a minor error at any step could culminate in an inaccurate final prediction. Therefore, LLM must accurately navigate the challenges posed by the Rust language as mentioned previously.

⁴Although Rust provides *dyn* keyword to delay these bound checks until runtime, it’s rarely used in the Rust crates.

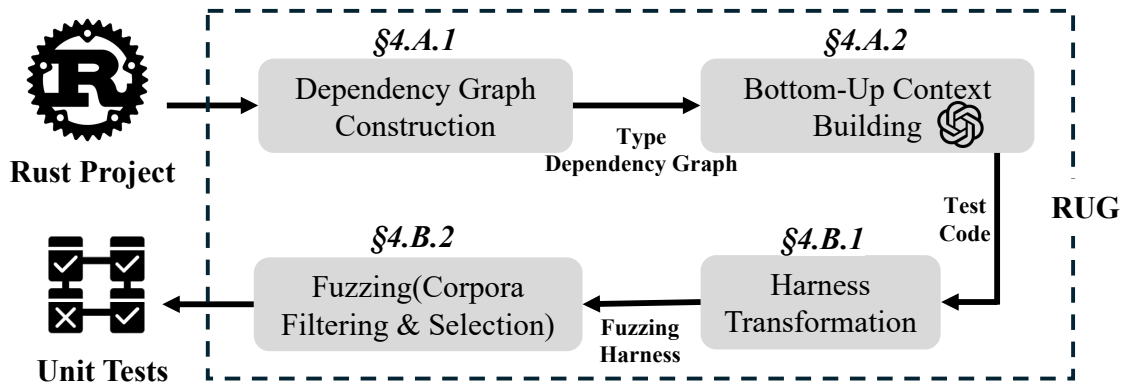


Figure 3.1: The general workflow and components of RUG. After building the type dependency graph from the input Rust project, RUG leverages bottom-up context building to handle the compilation challenges and fuzzing to resolve the coverage challenges.

3.3.3 Challenges of Low Code Coverage

Even after overcoming the challenge of generating unit tests that pass the compiler’s checks, low code coverage remains a significant issue when composing unit tests for Rust programs.

Difficulty in Path Exploration. In contrast to traditional approaches, LLM currently lacks the capability to examine the execution of the program. This limitation results in challenges in reasoning about branch conditions and exploring various code paths. Moreover, in the context of test assertions, the oracle is responsible for verifying the correctness of the function execution. While LLM is able to generate appropriate fields for verification, because of failure to statically reason the path that will be executed, they may yield incorrect values as oracles. Such a deficiency in accurately reasoning about data conditions and outcomes further complicates the LLM’s capability to produce valid and reliable test assertions, underlining the need for enhanced techniques.

3.4 RUG Design

To address the challenges of test generation and coverage, we propose a new tool called RUG, which uses a semantic-aware approach to guide the LLM building the test and

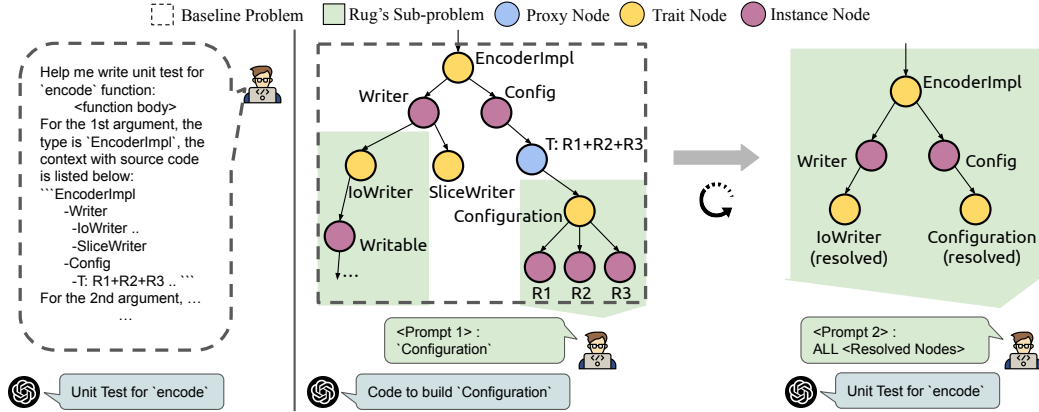


Figure 3.2: RUG’s bottom-up context building example. The left side represents the baseline’s one-shot approach, requiring LLM to generate the test with a long context, leading to buggy output. RUG automatically divides the task into subproblems and simplifies the task. The prompt of the RUG is in Figure 3.3 and Figure 3.4.

leverages fuzzing to expand testing coverage. The workflow of RUG is shown in Figure 4.1. Taking a Rust project as input, RUG first constructs a type dependency graph for the parameters of the target function, then it resolves the concrete types and the corresponding code for each node in the dependency graph from the bottom. After the root node is resolved, the unit test context is complete. Finally, RUG uses fuzzing to explore valid inputs and enhance code coverage.

In this section, we first introduce the context-building approach of RUG in subsection 3.4.1 and show the fuzzing process in subsection 3.4.2.

3.4.1 Testing Context Construction

To build the unit test context, RUG follows a two-step process for each target function: 1) RUG applies static analysis to construct a type dependency graph for the parameters of the target function; 2) Starting from the leaf node, RUG adopts the bottom-up approach to automatically build the concrete code with the help of LLM.

Constructing Type Dependency Graphs

To build the unit test context, RUG first statically builds the type dependency graph. To better explain our approach, we use $G = (V, E)$ to denote the type dependency graph, where G is a directed graph; V denotes the nodes in the graph and E denotes the edges. The details of the graph are defined as follows:

- $\forall v \in V, v \in \{v_{trait}, v_{instance}, v_{proxy}\}$, where v_{trait} denotes the vertices with trait bounds constraints; $v_{instance}$ represents the vertices with concrete types like struct or enum; v_{proxy} is a kind of special vertices for a proxy definition (line 10 in Listing 3.1), denoting the logical substitution of type compositions.
- $\forall e \in E, e$ is a directed edge indicating that the start vertex depends on the end vertex. For example, in Listing 3.1, line 4 can be represented as $EncoderImpl \rightarrow Writer, EncoderImpl$, and $Writer$ are vertices in the graph.

In this graph, the leaf vertices are defined as those with concrete types. For example, in Figure 3.2, which illustrates part of the type dependency graph for Listing 3.1, the vertex $Writer$ is considered an instance vertex, but since it does not have a concrete type, it is not a leaf vertex.

Bottom-Up Code Generation

Given the type dependency graph G , treat each node in the graph as a subproblem, and the unit test generation problem can be modeled as follows:

Given a type dependency graph G and parameters p_i of the target function, for each p_i , the context of the unit test to be generated is a subgraph $g \in G$, such that $\forall s \in S, s$ starts with p_i and ends with primitive type nodes, where S denotes all topological sequence permutations of g .

In Figure 3.2, the subgraph of $EncoderImpl$ is the scope of the target parameter to be solved. For existing LLM based auto-testing tools [29, 39, 38, 28], testing code generation

is finished in one shot like the left part in Figure 3.2: they collect the context and ask for the testing code, then check the output and apply automatic fixes or retries if necessary. This one-shot approach usually produces a long context for LLM to infer, increasing the difficulties of code generation due to the strict compiler checks and cascading errors in subsection 3.3.2.

RUG divides the generation of the test code into small steps as checkpoints, reducing the difficulties of each subproblem and guiding LLM to solve the final problem. Like the right part of Figure 3.2, RUG begins with nodes that can be directly instantiated, using LLM to construct the unit test context. Once the unit test code for a node is generated, the results are used to resolve its parent node in the graph. A node is ready to be resolved once all its dependent nodes have been addressed. For example, in Figure 3.2, after resolving the R_n types, RUG starts with the Configuration node, which can be resolved using the output of R_n type sub-problems. With the prompt shown in Figure 3.3, the LLM generates the concrete code for Configuration. To ensure the correctness of the generated code for each node, RUG uses an oracle compiler to verify that the code is compilable. Once validated, the results are used to generate the test code for the type T as a proxy node, denoting the composition of type R1+R2+R3.

When providing concrete types to the LLM, different strategies are employed for different types of nodes. For instance nodes, RUG resolves the correct definition paths for each type within the context and recursively gathers *relevant* items, including structure definitions, target function definitions, trait definitions, structure field types, implementation relationships. For trait nodes, RUG queries the intersection of bound constraints to identify all candidate types. If no valid candidate is found, RUG generates a prompt describing the trait and uses LLM to implement it. Finally, for proxy nodes, RUG applies its proxy definitions within the compilation context to find candidate types.

After resolving and validating all the unit test codes for each parameter, RUG leverages a test generation prompt (Figure 3.4) to ask LLM to generate the unit test code. The generated



Please help me fill in the sample code by creating an initialized local variable named {var-name} with type {var-type} using its constructor method or structural build in {crate-name} crate's {file-location} file. Fill in any sample data if necessary, ...

<Rust sample code to fill>

(Optional) For the {dependent-type}, please reuse below sample code to construct.

<Dependent code from previous round>

Figure 3.3: <Prompt 1>: template RUG used for each sub-problem. The optional paragraph is reusing the previous output to cut the context.

code is then validated using the compiler. The whole algorithm is shown in Algorithm 1: *getDependent* finds direct dependents (if any) of the input type for instance nodes and valid candidate types for trait & proxy nodes, *descriptionGen* generates the prompt description of the target type, $G[k]$ means accessing the value in the map G by the key k , and \bar{D} is a set of verified candidates for the input parameter type t .

Corner Cases

Although RUG's bottom-up building approach improves the quality of the generated code by minimizing the relevant context, there are several corner cases need to be handled. One of the corner cases is the loop in the subgraph g , indicating the presence of cyclical type dependencies. RUG will break the cycle by randomly determining orders and will remove the sample code in the prompt shown in Figure 3.4.

In addition, for trait/proxy nodes, sometimes there is no valid instance type in the compilation scope and RUG will prompt LLM to implement the traits. Meanwhile, for the trait/proxy nodes with multiple instances, RUG provides different candidate selection strategies according to the user's preferences. The occurrence of nodes with multiple candidates available takes around 17.48% and by default RUG will choose the candidate in

Algorithm 1 RUG’s context construction algorithm. t is the target parameter type, G is the type dependency graph. Corner cases are omitted for simplicity.

```
1: procedure BUILD_CONTEXT( $t, G$ )
2:    $\bar{D} \leftarrow \{\}$ 
3:    $\bar{S} \leftarrow getDependent(t, G)$ 
4:   for  $s \in \bar{S}$  do
5:     if not  $s$  is resolved then
6:        $build\_context(s, G)$ 
7:     end if
8:      $\bar{D}[s] \leftarrow G[s]$ 
9:   end for
10:  if  $t$  is Instance then
11:     $G[t] \leftarrow llmRequest(t, \bar{D})$ 
12:    if not  $compileVerify(t, G[t])$  then
13:       $G[t] = descriptionGen(t)$ 
14:    end if
15:  else
16:    if  $\bar{D}$  is Empty then
17:       $\bar{D} = llmRequest(t, descriptionGen(t))$ 
18:    end if
19:     $G[t] = \bar{D}$ 
20:  end if
21:  return  $G[t]$ 
22: end procedure
```



The target function is {fn-name} in {crate-name} crate's {file-location} file, its definition path is {def-path} and source code is like below:

<Target test function>

For n-th argument, {parameter-n-type} can be used, please use following sample code to construct it:

<Dependent code sample to reuse>

Please help me build unit test,

[Instructions to reuse the context]

Figure 3.4: <Prompt 2>: Final test generation template RUG used to combine the sub-solution together and build the target unit test.

the local crate.

Finally, due to the uncertainties of LLM, sometimes the model fails to give a correct answer, RUG will mark the subtask as unfinished and continue the next step with descriptions in natural language. In practice, we found, even without a sample code, that sometimes LLM can generate the correct code for the current sub-problem.

3.4.2 Fuzzing for Input Exploration

Although the divided context building approach helps LLM to write executable tests, the generation of useful test data is another burden for LLM. In subsection 3.3.3, we show that the root cause of this challenge is lack of the ability to inspect the state of the program during execution, so it is struggling for LLM to generate the corresponding data to trigger different conditions in the code path. In our motivating example, for the char type to encode, LLM usually prepares a valid ASCII symbol or one or two simple UTF-8 characters as test data. With the help of fuzzing, the test input is quickly scaled to complex UTF-8 characters and triggers the missed region. Thus, without fuzzing, it is challenging for LLM to prepare reasonable test data for high code coverage. In this section, we demonstrate how we prepare the fuzzing harness from generated tests and how we handle the redundant fuzzing corpora

as postprocessing.

Fuzzing Harness Transformation

Fuzzers need fuzzing harnesses to test with, where all input data is provided by the fuzzers as raw bytes. To convert existing test code into a fuzzing harness, RUG leverages program transformation to construct test data from raw inputs and preserve the semantics of the test body. During the transformation process, RUG first identifies all the primitive data in the original test code and replaces them with local variables of the same type built from the fuzzer input bytes. Meanwhile, RUG records these initial data as seeds and saves them as the initial corpus for the fuzzing process. Second, to ensure the graceful execution of fuzzer, RUG disabled all assertions to help fuzzer finish its execution. Although this step may miss some bugs, it ensures the graceful execution of the fuzzer and improves the code coverage. After fuzzing and postprocessing, RUG will review these assertions and try to replace the value based on the fuzzing result.

Fuzzing Corpora Selection

By applying fuzzers to an existing test program, we can efficiently expand our test coverage through the new input data found by the fuzzers. Because of its efficiency, a few seconds of fuzzing can execute the program thousands of times with hundreds of corpora generated, which is far beyond the requirement for the number of tests. To further manage these corpora, we develop a source code coverage-based, path weight guided corpora selection algorithm to filter and rank the corpora based on their coverage. The post-processing goes with two stages. First, RUG filters the corpora based on source code coverage, instead of fuzzers' bitmap coverage to remove the redundant inputs. Second, RUG uses *weight* to measure the importance of each code region[40] and the overall *weight* for each corpora to rank them. To calculate the weight, RUG assigns a default weight of **1** for each code region. For all the n corpora that touched this region, they will share the weight, gaining the score of $\frac{1}{n}$. The goal

of this weight sharing design is to find the corpus that covers more unique code regions, and the algorithm is shown in Algorithm 2: *src_cov* is a function that takes a specific corpus and returns a set of code regions \bar{R} that are covered by the given input. \bar{W} contains the weights for each input corpus. Finally, based on the weight score and the number of code regions covered, RUG ranks the corpora and selects them according to the given threshold.

Algorithm 2 RUG’s corpora ranking algorithm. \bar{I} is the input corpora as a set.

```

1: procedure RANK-CORPORA( $\bar{I}$ )
2:    $\bar{W}, \bar{C}, \bar{D} \leftarrow \{\}, \{\}, \{\}$ 
3:   for  $i \in \bar{I}$  do
4:      $\bar{R} \leftarrow \text{src\_cov}(i)$ 
5:      $\bar{D}[i] \leftarrow \bar{R}$ 
6:     for  $r \in \bar{R}$  do
7:        $\bar{C}[r] \leftarrow \text{getOrDefault}(\bar{C}, r, 0) + 1$ 
8:     end for
9:   end for
10:  for  $i \in \bar{I}$  do
11:     $\text{weights} \leftarrow 0$ 
12:    for  $r \in \bar{D}[i]$  do
13:       $\text{weights} \leftarrow \text{weights} + 1.0/\bar{C}[r]$ 
14:    end for
15:     $\bar{W}[i] \leftarrow \text{weights}$ 
16:  end for
17:  sort  $\bar{I}$  by  $\bar{W}[i], \forall i \in \bar{I}$  in descending order
18:  return  $\bar{I}$ 
19: end procedure

```

3.5 Evaluation

To demonstrate the effectiveness of RUG, we conducted a comprehensive evaluation motivated by the following research questions:

- **RQ1:** How does the test generation performance of RUG, in terms of coverage, compare to traditional tools?
- **RQ2:** Compared with other LLM-based tools, does RUG show any benefits?
- **RQ3:** How does each factor contribute to RUG’s testing coverage?

- **RQ4:** How is RUG’s usability for real world applications?
- **RQ5:** How robust is RUG in different scenarios? Specifically, how does RUG perform on crates that the LLM has not been trained on? Additionally, how do different type selection approaches impact the results?

3.5.1 RQ1: Comparison with Traditional Tools

To answer Q1, we compare RUG with four of the latest different test generation tools for Rust, including RustyUnit [25] as a SBST approach, SyRust[27] as a constraint solving program synthesizer, and RULF [41], RPG [42] as fuzzing-based tools.

Comparison with General Searching Based Tools. RustyUnit is a SBST approach that leverages the DynaMOSA algorithm to mutate the existing Rust codebase and generate the unit tests. And SyRust is a semantic-aware program synthesizer that uses a SAT solver to generate valid Rust programs. To ensure a fair comparison, we applied RUG to their original benchmarks separately and used LLVM source code coverage [40] to measure the code region coverage and function coverage. When calculating the code coverage, we considered only functional code, excluding testing code and compiler-generated code.

We conducted our experiments on servers equipped with two AMD EPYC 7452 CPUs and 256GB of memory, running Ubuntu 22.04. The Rust toolchain version used is nightly-2022-12-10, and all tools are assessed using their default configurations. Since RustyUnit provided different evolutionary algorithms, we selected the most powerful DynaMOSA algorithm, which achieves the highest code coverage, for comparison with RUG. For SyRust, we run the synthesizer with their default timeout of 10 hours for each project⁵.

The evaluation results are shown in Table 3.1. RUG achieves 54.84% coverage on RustyUnit’s and 52.28% on SyRust’s, outperforming all the works. RustyUnit’s seeded DynaMOSA algorithm, which is part of the genetic algorithm family, relies on existing code as ‘parents’ to mutate unit test statements. However, because the usage of each

⁵The crates are: bitvec, crossbeam, dashmap, imrc, ndarray, num-rational, slab for **data structure** and csv-core, encode-unicode, encoding-rs, hcid, sval, urlencoding, utf8-width for **encoding**.

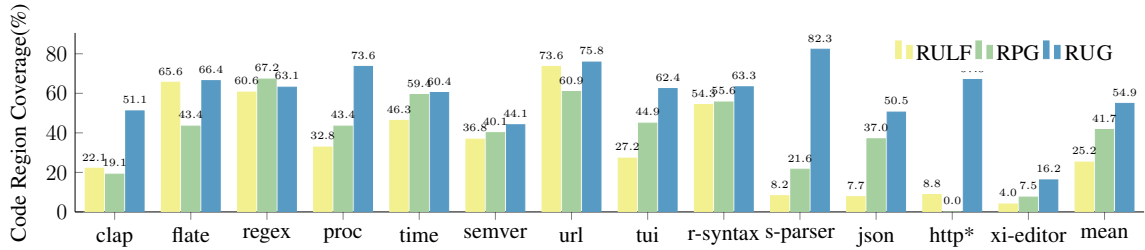


Figure 3.5: Coverage comparison between RUG and RULF, RPG. *: RPG encounters an unexpected crash while running for http crate.

Table 3.1: Coverage comparison between RUG, RustyUnit and SyRust. RUG out performs RustyUnit by 20.14% and SyRust by 21.57%.

Crate	Func	Region	Func	Region
	RustyUnit		RUG	
gamie	55.54%	30.79%	68.67%	72.24%
humantime	45.55%	26.67%	50.33%	64.92%
lsd	32.58%	40.23%	37.66%	43.98%
quick-xml	17.38%	24.61%	54.5%	62.76%
tight	24.70%	30.27%	32.24%	36.90%
time	75.26%	70.78%	68.13%	56.94%
mean	37.23%	34.70%	49.96%	54.84%
	SyRust		RUG	
data-structure	26.11%	31.19%	52.10%	56.03%
encoding	30.69%	28.51%	55.47%	48.54%
mean	28.40%	30.65%	53.79%	52.28%

function within a crate is often imbalanced, this approach struggles with less frequently used functions. SyRust, on the other hand, requires a manually crafted template of function arguments for synthesis, necessitating developer input and limiting the synthesis to the template input. Overall, RUG’s superior coverage is primarily due to its ability to construct calling contexts for a wider range of target functions.

Comparison with Fuzzing Tools. Besides comparing with SBST tool and program synthesis approach, RUG is evaluated against fuzzing-based tools: RULF [41] and RPG [42]. RULF constructs type dependency graphs to assist in generating fuzzing harnesses, producing a sequence of API calls through graph traversal. RPG extends RULF by adding support for generic parameters and prioritizes functions containing unsafe regions using a pool-based generator. We run all tools on RULF’s benchmarks under default configurations

to ensure a fair comparison. For each individual fuzzing harness, we set a timeout of 24 hours for RULF and 4 hours for RPG, since RPG generates much more fuzzing harnesses compared with RULF⁶. For tests generated by RUG, we conduct fuzzing for 60 seconds per function, which is a reasonable time for unit test generation. All fuzzing experiments were repeated three times, and the average values are reported.

The results are shown in Figure 3.5, where RUG achieves an average coverage of 54.9%, outperforming RULF’s 25.2% and RPG’s 41.7%. RULF’s harnesses generation process is an NP-Complete problem, using a heuristic threshold to limit the generation process. Besides, RULF does not implement static analysis for generic parameters, preventing it from triggering functions that use traits (e.g. `json crate`). RPG improves upon RULF by supporting generic parameters and utilizing a pool-based generator to produce more fuzzing harnesses. However, RPG encounters a similar search problem as RULF and prioritizes functions with unsafe regions during the generation process, which affects the total number of harnesses. In addition, if no valid candidate is found for the given generic parameters, the RPG may not locate a candidate outside the current compilation scope, leading to a missing fuzzing harness. In contrast, RUG leverages LLM to generate the testing code, avoiding the search problem and covering more testing functions.

3.5.2 RQ2: Comparison with LLM-based Tools

To answer the RQ2 and demonstrate RUG’s effectiveness in tackling the two challenges, we select 17 crates from the most downloaded on crates.io averaging 24,937 LoC per crate. Beyond their popularity, these crates represent a diverse spectrum of Rust applications in system development: *mio*, *crc32fast* for system call and hardware instructions; *json*, *toml*, *bincode* for data serialisations; *hashes*, *uuid* for crypto computations; and *num-traits*, *ryu* for numerical computations.

For the baseline approach, we integrate three LLM-based approaches: ChatUniTest [29]

⁶This setting outfits RPG’s evaluation settings of 4 to 6 hours total timeout for each project.

Table 3.2: Code coverage evaluation for RUG on 17 popular Rust crates. 'P' indicates the PR is still pending for response. The three dimensions for sensitivity tests are: GPT model versions, generation approaches and whether applying fuzzing. The newly API coverage denotes the APIs that baseline failed to test and only covered by RUG.

Crate Name (Downloads)	Tests ac/rej	GPT-3.5				GPT-4				Newly API Cov Rate	Human Test Coverage
		Base		RUG		Base		RUG			
		w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing		
bincode(49M)	4/0	1.57%	1.57%	22.92%	23.91%	16.63%	18.79%	44.67%	47.91%	74.11%	64.58%
chrono(128M)	22/13	37.88%	44.07%	47.2%	58.29%	54.04%	59.24%	56.90%	62.67%	73.05%	76.66%
hashes(266M)	P(7)	43.84%	43.84%	68.28%	68.28%	57.71%	57.71%	68.96%	85.16%	61.41%	85.17%
humantime(98M)	P(5)	63.09%	64.40%	67.02%	75.39%	74.08%	75.92%	74.61%	80.37%	40.00%	79.32%
itoa(221M)	1/0	26.00%	26.00%	82.00%	96.00%	96.00%	98.00%	100.00%	100.00%	83.33%	86.00%
json(203M)	-	28.10%	35.69%	44.60%	52.07%	62.26%	67.00%	70.25%	70.49%	47.33%	72.36%
mio(145M)	-	20.47%	20.47%	25.20%	25.20%	26.77%	26.77%	33.86%	33.86%	38.89%	24.19%
nom(114M)	6/1/P(14)	25.81%	25.81%	39.93%	40.04%	51.13%	51.17%	53.84%	53.87%	28.64%	76.20%
num-traits(185M)	-	36.02%	36.47%	43.20%	43.95%	46.94%	46.94%	47.23%	47.98%	90.36%	50.58%
demangle(93M)	P(14)	21.32%	21.62%	21.83%	65.99%	20.00%	74.82%	26.60%	76.55%	18.75%	72.25%
crc32fast(104M)	-	62.35%	64.71%	70.59%	71.76%	87.06%	88.24%	87.06%	88.24%	92.86%	68.24%
ryu(185M)	0/3	52.51%	95.28%	61.65%	97.64%	76.40%	99.42%	81.72%	99.42%	100.00%	87.85%
semver(168M)	18/0	61.40%	62.96%	62.54%	73.36%	72.36%	74.64%	74.22%	74.50%	95.24%	84.33%
textwrap(134M)	1/0	88.84%	92.56%	90.15%	94.31%	92.78%	94.75%	92.56%	94.97%	83.34%	87.53%
time(200M)	P(3)	33.08%	35.06%	48.98%	51.72%	55.34%	55.34%	79.89%	79.89%	66.06%	96.48%
toml(125M)	-	32.43%	37.06%	47.28%	49.02%	59.58%	64.40%	38.90%	38.90%	25.14%	70.81%
uuid(108M)	1/0	58.66%	64.44%	69.30%	77.20%	73.86%	75.08%	75.68%	76.60%	88.89%	61.40%
mean	-	40.79%	45.41%	53.69%	62.60%	60.17%	66.37%	65.11%	71.37%	65.14%	73.18%
Identifier	Ⓐ	Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓖ	Ⓙ	Ⓚ

for code generation, TestGen-LLM[33] for oracle compiler check, and RustAssistant [38] for code repair. The process is as follows: 1) We collect the relevant source code context of the target function and send it, along with the prompt, to the LLM to generate test cases. 2) The generated test code is then checked for correctness by an oracle compiler. 3) If the LLM output fails to pass the compiler checks, we collect the error message, line numbers, and reasons for the failure, then ask LLM to fix the problems within the same context. Since ChatUniTest and TestGen-LLM are not implemented for Rust, we reimplemented them specifically for Rust during the preparation of the baseline experiment. We treat this pipeline of three approaches as the baseline, representing the best existing LLM-based practice.

Compared to baseline, RUG does not have an additional error fixing stage and employs a bottom-up building algorithm to generate the compilable test code. In addition, RUG transforms the test code into a fuzzing harness and takes advantage of fuzzing as a final step.

The results across different experiment configurations are shown in Table 3.2. Compared with Ⓐ Ⓓ and Ⓔ Ⓕ, we show that RUG's overall approach improves the testing coverage by 21.81% and 11.20% under different LLMs. The improvement number of GPT-4 is

relatively smaller because the remaining untested code regions are limited, especially we only focus on the actual functional code and do not generate tests for derived functions⁷.

3.5.3 RQ3: Ablation Study

To demonstrate the effectiveness of RUG’s two techniques(bottom-up building and fuzzing) and its sensitivity to different LLM models, we conducted sensitivity experiments for each of them, showing their individual contributions to the final results shown in Table 3.2.

Bottom-up Building. To demonstrate the effectiveness of the bottom-up building approach of RUG, we evaluate the "newly covered APIs" shown in ①, representing target APIs that the baseline tools failed to generate the test context but can be resolved by the bottom-up approach of RUG. RUG successfully covers 65.14% of these APIs, highlighting its effectiveness. In addition, comparing between ① ③ and ② ④, we show that bottom-up building helps improve code coverage by 12.90% and 4.94%, which accounts for about half of the total coverage improvement.

Fuzzing Approach. To show the improvement contributed by fuzzing, we launch RUG’s fuzzing harness transformer on both generated tests. The result shows that the fuzzing component increases code coverage by 8.91% in ③ ④ and 6.26% in ⑤ ⑥. Even when the initial method achieved high coverage, such as 60.17% in ② and 65.11% in ⑤, fuzzing still provided an additional coverage boost of around 6%. This reinforces our insight that applying fuzzing can effectively enhance code coverage.

Large Language Model Versions. Finally, we test RUG’s sensitivity to different version of LLM models(GPT-3.5 and GPT-4). Clearly, GPT-4 is more intelligent than GPT-3.5 by showing about 20% improvement between ① and ②. The experiment result shows code coverage are all further improved by applying RUG for 21.81% in GPT-3.5 and 11.20% in GPT-4. Besides, in terms of testing coverage, applying RUG on GPT-3.5 ④ achieves a higher coverage number than vanilla GPT-4 approach ②.

⁷Rustc allows #[derive] to generate default implementations like clone, which will be counted as separate functions in LLVM source code coverage.

3.5.4 RQ4: Practical Usability Evaluation

In this section, we evaluate the usability of RUG by comparing its generated tests with human-written tests to answer RQ4. For the 17 crates in Table 3.2, we note that due to their popularity, these crates are actively maintained and well tested by the developers, achieving an average coverage of 73.18%. We refer to their coverage as the best that human developers can achieve in practice, and evaluate RUG’s generated tests against the human-written tests to demonstrate its practical usage.

Code Coverage. As shown in the column ①, the developers achieve an average of 73.18% code coverage on these 17 crates, and by leveraging the latest GPT-4, RUG can achieve 71.37% in ②, which is comparable to human tests. In addition, among the 17 crates, RUG and human developers both achieve higher coverage on 8 crates and get the similar testing coverage for hashes, indicating the potential usage of RUG in the software testing process. In addition, for crates like `chrono`, `nom` and `time`, RUG largely expands the testing coverage than developers’, showing the effectiveness of RUG.

For crates where RUG does not outperform (e.g. `bincode`, `toml`), the main reasons are as follows: 1) These crates are template libraries for (de)serialization, with few type implementations in the code base, making it difficult for RUG to find the valid candidates. 2) The code requires highly structured input to be fuzzed effectively, which is challenging for fuzzers without concrete structure definitions.

Readability. To evaluate RUG’s tests’ readability, we collect the test coverage of RUG and human developers and find ten missing tests for existing human-written tests. We directly leverage RUG’s generated tests, without changing test bodies and send them as PRs to the open source projects. To our surprise, the developers are happy to merge these machine generated tests. RUG generated a total of 248 unit tests, of which we submitted 113 to the corresponding crates based on their quality and priority. So far, 53 of these unit tests have been merged with positive feedback.

Developers chose not to merge 17 tests for two main reasons: first, the target functions are imported from external libraries(16), and the developers do not intend to include tests; second, the submitted tests are closed after a long pending period(1). We are still awaiting feedback on the remaining 43 unit tests. In general, 75.71% of the tests reviewed are merged by the developers.

Token Consumption. Although LLMs are getting more and more cheap nowadays⁸, automatic tools quickly consume large number of tokens by sending relevant source code as prompts. For example, under the GPT-4 model, the cost of the baseline approach is around \$1000. However, with the help of type-aware caching, RUG only takes 51.3% of the total tokens in the baseline approach and improves the coverage of the testing by 10.4%. RUG consumes less tokens because, for each method under the same structs with dependent types, RUG feeds each unique dependency to the LLM only once, caching and reusing the result without saving the prompt. This approach saves tokens for subsequent functions with the similar dependencies. In contrast, baseline approaches must feed the entire context, including transitive dependencies each time, leading to higher token usage.

3.5.5 RQ5: Robustness Evaluation

To evaluate RUG’s robustness, we conduct two experiments to measure RUG’s performance under unlearned crates and different candidate selection strategies for corner cases in section 3.4. For `gpt-3.5-turbo-0125`, its knowledge cut-off is September 2021 [43], and we carefully select ten crates that are **created** after January 2022 from crates.io as our benchmark: *coolssh*, *xelis-hash* for algorithm calculations, *behindthename*, *cbored*, *utapi* for FFI wrappers, *metrics-evaluation*, *europs-elects-csv* for string parsing, *osrm*, *mc-proto* for binary data parsing and *atomic-waitgroup* for concurrency.

Unlearned Crates. The evaluation result is shown in Table 3.3. Compared with the baseline LLM approach, RUG achieves an average improvement of 12.93%. When comparing

⁸As of March 2024, the cost of 1M tokens as inputs is \$30 for GPT-4 and \$0.5 for GPT-3.5

Table 3.3: Robustness evaluation of rug on unlearned crates after the LLM training cut-off. The 'N/A' denotes the crates don't have tests. '*': the RUG with **LS** strategy achieves **48.98%** excluding the no-test crates, close to the **50.64%** coverage by developers.

Crates	Base	Human	RUG		
			LS	RR	UF
metrics_evaluation	53.25%	70.45%	69.48%	71.43%	63.31%
coolssh	25.37%	N/A	36.76%	31.27%	35.66%
cbored	45.67%	23.51%	50.11%	44.53%	50.70%
rust_mc_proto	16.07%	N/A	44.60%	43.49%	45.98%
atomic-waitgroup	36.51%	80.95%	41.27%	36.51%	41.27%
osrm_client	5.29%	10.20%	9.02%	8.04%	9.41%
europa-electcs-csv	14.06%	N/A	20.31%	18.75%	25.00%
xelis_hash	49.51%	77.67%	80.58%	85.44%	77.18%
behindthename	21.71%	41.09%	43.41%	37.21%	43.41%
utapi-rs	14.24%	N/A	15.82%	15.19%	16.46%
mean	28.17%	50.64%	41.14%*	39.18%	40.84%

against human-written tests, excluding the four crates without human tests, RUG achieves 48.98% on the remaining crates, just 1.66% less than the developers' tests. In addition, RUG achieves a higher coverage rate on half of the crates compared to human developers, showing RUG's robustness on unlearned crates.

Candidate Selections. Regarding candidate selections, we compare three selection strategies for the same crates used in the data leakage analysis. Those selection strategies are widely used : local crate selection(LS), round robin(RR), and unsafe first(UF). The LS is the default strategy of RUG, trying to cover more local types, and only conducts random selection when there are multiple local candidate types. In addition, we further evaluated two more strategies: RR tries to fairly cover the candidate types while selection and UF leverages the program analysis to find the candidate type with highest number of unsafe regions, which is close to RPG's ranking algorithm. The result of different candidate selections are shown in the Table 3.3: compared with RR and UF, local crate selection(LS) prefers to use the instance from the current crate, increasing the chance of local code coverage. For RR and UF, they may select the outside instance as final candidate, leading to the potential drop in the local testing coverage.

3.6 Limitation

In this section, we discuss the potential limitations of RUG and possible future improvements. RUG relies on static type dependency analysis to divide the context for the following steps. Therefore, for other strong-typed languages such as Java, there is no fundamental burden to transplant RUG. For weak typed languages like Python, the accuracy and scope of the static analysis will affect the quality of the generated tests, and we argue that this is a general issue for these languages and other automatic testing tools have similar problems. In addition, RUG is bonded to the specific version of Rust and a general implementation can save time to accommodate changes.

To expand the code coverage of the generated tests, RUG uses fuzzing to explore the input space. In order to launch the fuzzer, RUG temporarily disables all assertions, which might miss some bugs while the fuzzing process. Apart from that, RUG's corpora selection algorithm may lose the code coverage based on the number limitation of generated tests. For the generated test, RUG does not consider the framework and coding style of the existing tests, which can be given as a sample prompt to LLM for better code quality.

CHAPTER 4

RUBY: MACHINE LEARNING-BASED UNSAFE DETECTION IN RUST BINARIES

4.1 Overview

Memory safety bugs are typically confined to the unsafe regions, which have been the primary focus of Rust bug-finding tools. However, such tools rely on the presence of the unsafe keyword in Rust source code; there are no tools available that can examine Rust binaries to pinpoint unsafe areas, since these binaries consist solely of assembly code without clear indicators for the unsafe regions. Therefore, identifying unsafe regions in Rust binaries is a crucial step toward finding memory safety bugs.

We propose RUBY, the first tool that finds unsafe regions in Rust binaries using machine learning. Our key insight is that we can automatically build enormous unsafe Rust functions and labels from open-source Rust projects for training purposes. This empowers RUBY to support numerous patterns of assembly code in the unsafe regions of the Rust binaries. We evaluated RUBY and demonstrated that RUBY successfully identified 91.75% of the total unsafe regions with a false positive rate of 6.16%, even on real-world applications like web browsers, language interpreters. To further illustrate RUBY’s practical usage, we applied RUBY to guide symbolic execution and fuzzing to find memory safety bugs in Rust programs, showing a speed-up of 57.95% and 21.26%, respectively. By applying RUBY for fuzzing Rust libraries in Android, we successfully identified and reported five zero-day bugs, which have been patched by developers.

4.2 Introduction

To find and fix memory-safe bugs in Rust programs, a rich line of prior work (*e.g.*, [44, 42, 45, 46, 47]) has focused on analyzing the unsafe regions in Rust source code. For example, Rudra [44] presented three important patterns of memory safety bugs in unsafe Rust and identified these bugs through static analysis of the unsafe regions; RPG [42] prioritized fuzzing unsafe regions in Rust libraries to more efficiently identify memory safety bugs; ERASAN [45] proposed a more efficient address sanitizer [48] for Rust programs, leveraging the characteristics of unsafe regions.

Our Focus: Rust Binary Analysis. Although source code analysis tools have made notable strides in detecting memory safety issues, a fundamental constraint exists: they need access to the program’s source code. This constraint becomes especially noticeable as Rust adoption grows in commercial and security-sensitive sectors where source code is frequently inaccessible. For example, in the examination of proprietary Rust-based firmware, embedded systems, or commercial device drivers like Ferrocene [49] and Windows device driver [6], security analysts and system administrators must depend solely on binary analysis. Indeed, even if source code is available, binary analysis remains essential as it can uncover vulnerabilities introduced during the compilation process, such as compiler optimizations that modify memory access patterns [50] or macro expansions that produce unforeseen unsafe operations [51], introducing security flaws not evident in the source code.

For Rust binary analysis, pinpointing unsafe regions is critical since these regions bypass the Rust compiler’s memory safety checks and are susceptible to vulnerabilities. However, as the unsafe keyword is absent from Rust binaries after compilation, identifying these unsafe regions becomes challenging. More specifically, these challenges stem from the following three main reasons:

- **Diverse unsafe Rust operations:** In practice, the rustc compiler checks 12 unsafe operations [10] outlined in Table 4.1. Each operation presents a distinct root cause, and

```
1 // a is &i32, dereferencing is safe
2 *a;           // => ; deref op is optimized out
3 // a is *i32, dereferencing is unsafe
4 unsafe {*a} // => mov eax,DWORD PTR [rbx]
```

Listing 4.1: Dereferencing a pointer and reference: reference can be optimized out but pointer is compiled into an explicit access. Code are omitted for simplicity and compiled assemblies are in Listing 4.4

addressing each unsafe operation constitutes a subproblem of the broader challenge of recovering unsafe Rust from binaries.

- **Diverse safe/unsafe Rust boundaries:** Some unsafe operations can closely resemble safe operations, making them challenging to identify. Examples include dereferencing pointers or references, traversing unions or structs, and accessing mutable or immutable static variables. These operations exhibit minimal differences in the compiled binary; *e.g.*, in Listing 4.1, there exists merely a one-instruction difference between the two functions.
- **Diverse architectures and compiler toolchains:** Unlike source code, binary programs are closely bound to the target architectures. Rust relies on LLVM as its backend to generate machine code, which does not preserve unsafe information. Therefore, different architectures and compiler toolchains can produce distinct binary programs from the same codebase.

Our Approach. Inspired by recent advances in the usage of machine learning (ML) to understand structures from binary programs, we propose RUBY, which leverages ML to address the diverse challenges mentioned above and identify unsafe regions within Rust binaries. Our insight in using ML is to *automatically* generate a large number of diverse datasets for training, enabling the ML model to identify patterns that distinguish unsafe regions. For diverse architectures and compiler toolchains, RUBY is trained based on x86 & Arm and fine-tuned for different compilers to enhance its adaptability.

To build and evaluate RUBY, we first construct CrateU dataset, which contains the unsafe labels on functions from the entire crates.io. Besides, RUBY collects the RustSecU

and RustSecB datasets containing real-world memory safety bugs from a five-year period, with each bug’s root cause manually labeled to evaluate RUBY’s performance. We trained RUBY on CrateU and evaluated it on RustSecU and RustSecB. Our evaluation demonstrates that RUBY can automatically identify unsafe regions from input Rust binaries, achieving a precision of 93.84% and a recall of 91.75%. Moreover, RUBY’s results can aid in the bug identification process by reducing the search area to merely 7.43% of the total binaries with just a 25% overhead. In comparison, a random search requires examining 41.38% of the binaries, demonstrating a significant enhancement.

To illustrate the real-world application of RUBY in the vulnerability hunting process, we incorporate RUBY into the program analysis workflow, using its results as guidance for static analysis by Angr [52] and directed fuzzing by AFLGo [53]. Utilizing RUBY, we can achieve a time reduction of 57.95% in symbolic execution by Angr and 21.26% in directed fuzzing efforts to identify bugs. By applying RUBY’s results with Android blackbox fuzzing, we identified five bugs in Android’s Rust libraries, which have been confirmed and patched by Google. In summary, RUBY’s **contributions** are as follows:

- We propose a new tool RUBY¹, which leverages machine learning techniques to address diverse challenges and identify unsafe regions in Rust binaries. To our knowledge, RUBY is the first tool specifically designed to locate unsafe regions in Rust binaries.
- We comprehensively study different unsafe operations and their binary assembly differences, collect binary program datasets CrateU, RustSecU and RustSecB for safe/unsafe Rust classification and memory safety bug detection. These datasets can be reused for training purposes and serve as evaluation metrics for binary analysis.
- We extensively evaluate RUBY, demonstrating that RUBY is **effective** (achieving high accuracy), **efficient** (capable of analyzing crates.io within a week) and **robust** (compatible with both x64 and ARM architectures, and different Rustc versions across the past four years).

¹RUBY’s artifact is available at <https://doi.org/10.5281/zenodo.14217066>.

Table 4.1: Unsafe operations in Rustc [10] and their distributions in the binary (safe Rust takes the 76.28%). RUBY studies their unique binary representations in section 4.4.

Label	Unsafe Operations	Description	Occurrence (%)
1	CallToUnsafeFunction	Call an unsafe function. This type has two subtypes: “internal” (1) and “external” (2)	17.61%
2			0.17%
3	UseOfInlineAssembly	Use a <code>asm!</code> macro with low-level assembly.	0.01%
4	InitializingTypeWith	Initialize a layout restricted type’s field with a value outside the valid range.	0.56%
5	CastPointerToInteger	Cast pointers to integers in constants.	0
6	UseOfMutableStatic	Access to a mutable static variable.	0.08%
7	UseOfExternStatic	Access to a mutable static variable from external.	0.01%
8	DerefOfRawPointer	Dereference a raw pointer.	2.59%
9	AccessToUnionField	Access to a union field.	1.69%
10	MutationOfLayoutConstrainedField	Change the layout of a constrained field.	0
11	BorrowOfLayoutConstrainedField	Borrow a layout constrained field.	0
12	CallToFunctionWith	Call to a function that requires special target features.	1.02%

4.3 Threat Model

We assume possession of a Rust binary compiled with the release profile, without debugging symbols. Furthermore, we are aware of all the function boundaries, which can be analyzed using reverse engineering tools such as IDA or Ghidra. Besides function boundaries, we do not require any other knowledge about the binary.

RUBY’s objective is to identify the code regions in the binary that are written using unsafe Rust to minimize manual efforts in exploitation. In particular, the Rust compiler defines five operations as unsafe operations, which help programmers identify unsafe regions and ensure memory safety [8]: dereference a raw pointer, call an unsafe function or method, access or modify a mutable static variable, implement an unsafe trait, and access fields of unions. As these memory-related operations within unsafe regions are not checked by a compiler for memory safety, they are likely to cause memory-safety bugs. In practice, the Rust compiler explicitly checks 12 unsafe operations, as shown in Table 4.1. These 12 explicit operations are derived from the five unsafe operations with detailed contexts. These unsafe Rust takes 24.6% of the codebase across all crates in the Rust community [10], and even if the target project does not have any unsafe code, the usage of third-party and the

standard libraries can introduce implicit unsafe regions in binary.

Once the unsafe regions are identified, we can focus on those areas and employ additional intensive analysis (e.g., symbolic execution, directed fuzzing) to uncover potential vulnerabilities (subsection 4.6.4).

4.4 Unsafe Binary Study

In this section, we discuss each of the unsafe operations in detail and show the unique binary representations² of each unsafe operation.

4.4.1 Calling Unsafe Functions

The first category of unsafe operations in Rust is calling unsafe functions. Based on the owner of the unsafe functions, calling unsafe functions has two cases: calling unsafe FFI (Foreign Function Interface) functions or calling unsafe Rust functions. An unsafe function indicates that there is an unsafe operation inside the function body, and by adding strict constraints, developers can convert an unsafe function into a safe function.

Calling unsafe Rust Function

Calling unsafe Rust functions (*i.e.*, label 1) represents that both the caller and callee are from Rust. It is a language-level definition of unsafe because calling an unsafe Rust function does not directly trigger undefined behaviors or memory safety errors, but it is the unsafe operations inside the unsafe functions that trigger these bugs.

To correctly infer such functions, there are two challenges:

- By adding certain restrictions, developers can wrap an unsafe function into a safe function. However, to detect unsafe Rust functions, RUBY is expected to reason these complex restrictions to check if they covered all cases, which is as difficult as the NP hard

²All the code samples are compiled and discussed with release profile with Rustc 1.67.0 and linked to executable as output.

problem [54].

- To audit the target function, all callee functions and their inner functions are expected to be added to the context, exceeding the token limitation for machine learning models. Therefore, directly detecting unsafe Rust is challenging both for static analysis and machine learning models.

Instead of directly identifying such cases, RUBY tries to identify the root unsafe operations inside the unsafe function and skip the unsafe calls. RUBY's solution is based on a property of calling unsafe Rust functions:

Theorem 1. *Calling unsafe Rust functions must have an inner unsafe operation other than itself.*

We separate this property into two individual problems:

Problem 1. The Rust unsafe calling function must have a different innermost unsafe operation.

Suppose we have an unsafe function F in Rust without root cause operation, which means that the Rustc compiler cannot detect any unsafe operations inside F . Then we can safely remove the `unsafe` keyword from F without causing compilation errors. Thus, for any unsafe Rust functions, there must be an inner unsafe operation inside the function body, referring to its root cause.

Problem 2. The innermost unsafe operation within the unsafe function is not calling to an unsafe Rust function.

Suppose that we have a function F_{n+1} calling an unsafe function F_n , and the caller F_{n+1} will have unsafe regions around the calling instructions. For the callee function F_n , its unsafe label can be either because it calls other unsafe Rust functions (*i.e.*, label 1) or it has other unsafe operations (*i.e.*, label 2-12) in Table 4.1. For the second case, we show that the root cause is different from the callee itself. For the first case, there must exist another unsafe Rust function F_{n-1} called F_n , so the root cause of F_n will be delegated to F_{n-1} . Due

to Problem 1’s proof, there must exist an innermost function F_0 that could not be further delegated, and the unsafe operation F_0 belongs to the second case.

Therefore, guided by Theorem 1, RUBY skips calling unsafe Rust function identification and attempts to locate other concrete unsafe operations as their root causes. Subsequently, utilizing the call graph and the root cause locations, automatic tools such as directed fuzzing will strive to identify a valid path to reach the target.

Calling unsafe FFI Function

FFI functions enable Rust to interact with existing C/C++ libraries and are inherently unsafe. These functions can originate from either statically or dynamically linked libraries. For dynamically linked functions, RUBY embeds the dynamic symbol table entries corresponding to the called functions as part of the input, aiding the model in recognizing such cases. In the case of statically linked functions, Rust functions often exhibit different stack prologues from their C/C++ counterparts. Due to ownership tracking and move semantics, Rust tends to allocate more local variables, resulting in larger stack frames compared to typical C/C++ functions. An example can be found at Godbolt. These differences can be captured by ML models and utilized to distinguish the function sources.

4.4.2 Using Inline Assembly

Rust, similar to C/C++, enables developers to integrate inline assembly for precise control over low-level behavior. These inline instructions are treated as `unsafe` since they bypass the Rust compiler’s type and memory safety checks. As shown in Listing 4.2, developer-written assembly is preserved verbatim in the final binary, whereas semantically equivalent Rust code is typically optimized away or transformed. Moreover, manually crafted assembly usually varies from compiler-generated code, and these discrepancies can be utilized to detect potential unsafe operations during binary analysis. However, we note that developers may write arbitrary inline assembly due to its flexibility, creating challenges

```

1 let mut x: u64 = 4;
2 asm!(
3     "mov_{tmp},_{x}",
4     "shl_{tmp},_1",
5     "shl_{x},_2",
6     "add_{x},_{tmp}",
7     x = inout(reg) x,
8     tmp = out(reg) _,
9 ); // Assembly preserved in binary
10 let mut y: u64 = 4;
11 let tmp = y << 1;
12 y = y << 2;
13 y = y + tmp; // Compiler optimizes computation

```

Listing 4.2: Example of using inline assembly in Rust. The hand-written assembly is preserved in the final binary whereas equivalent Rust code is optimized by compiler. Full example can be found at [Godbolt](#)

```

1 #[rustc_layout_scalar_valid_range_start(1)]
2 #[rustc_layout_scalar_valid_range_end(5)]
3 struct NonZeroI64(i64);
4 size_of:::<Option<NonZeroI64>> // 8
5 #[repr(transparent)]
6 struct PlainI64(i64);
7 size_of:::<Option<PlainI64>> // 16

```

Listing 4.3: Example of using `rustc_layout_scalar_valid_range` to enable the niche optimizations for Rustc, leading to different memory layout. The full example can be found in [Godbolt](#)

for RUBY to identify.

4.4.3 Initializing Type with Constraints

To further support compiler optimization, Rust provides `rustc_layout_scalar_valid_range` attributes that enable users to specify the valid range of a given variable. Any value outside of the valid range can lead to invalid values, which is considered unsafe in Rust. The compiler can then assume this valid range and perform niche optimizations accordingly. As shown in Listing 4.3, niche optimization [55] in Rust attempts to compress the actual memory size of enums with values by reusing invalid

values of the wrapped type. In the example, `NonZeroI64` is defined with a valid range from 1 to 5, making values outside this range—including 0—invalid. This enables the compiler to use 0 as a niche value to represent `None`, reducing the size of `Option<NonZeroI64>` to 8 bytes, the same as `i64`. In contrast, `PlainI64` does not provide such a range guarantee, thus `Option<PlainI64>` cannot reuse any value as a niche and requires 16 bytes to store both the value and the discriminant. By inspecting the memory access patterns, RUBY can analyze the memory size of the types and attempt to identify this type of unsafe operation.

4.4.4 Mutating Static Variables

Mutating global variables can lead to data races, which are considered unsafe in Rust. To mitigate this, Rustc explicitly checks two types of unsafe operations: (1) mutations to static variables defined in Rust code, and (2) accesses to static variables originating from FFI libraries (e.g., in C/C++).

To support detection in binary analysis, RUBY embeds information from static variable sections (i.e., `.data` and `.bss` in ELF binaries) into the corresponding functions, enabling the model to identify and reason about global variable usage.

For FFI static variables, Rustc cannot analyze their mutability, as their definitions reside outside of its analysis scope. Consequently, it conservatively assumes all FFI static variables are mutable and treats any access to them as unsafe. As discussed in subsection 4.4.1, RUBY can leverage characteristic differences between Rust and C/C++ binaries to infer the origins of these static variables and assess their safety.

4.4.5 Dereferencing Raw Pointers

Dereferencing raw pointers, a common operation in C/C++, can lead to severe memory safety issues, such as use-after-free and out-of-bounds access. To uphold memory safety, Rust enforces the use of references instead of raw pointers through its borrow checker, as illustrated in Listing 4.1.

```

1 <reference>:
2     ... ; omitted 13 instructions for simplicity
3     ; deref reference is optimized and removed
4     8eb4: add     rsp,0x40
5     8eb8: pop     rbx
6     8eb9: ret
7
8 <pointer>:
9     ... ; omitted 13 instructions for simplicity
10    8f33: mov     eax,DWORD PTR [rbx]; deref ptr
11    8f35: add     rsp,0x40
12    8f39: pop     rbx
13    8f3a: ret

```

Listing 4.4: Different binary instruction outputs for code shown in Listing 4.1 to access pointers/references. The safe guarantee of reference promotes its value to registers. The full example can be found at [Godbolt](#).

However, from the perspective of binary instructions, both references and raw pointers are represented as memory addresses, making them indistinguishable at the instruction level. This poses a key challenge in binary analysis: correctly identifying whether a given memory access corresponds to a safe reference or an unsafe raw pointer.

To address this, we observe that Rust references act as ‘checked pointers’ with guaranteed valid access. This property enables the compiler to optimize references more aggressively, frequently promoting them to registers and eliminating redundant memory loads. In contrast, raw pointer accesses lack these guarantees and are generally preserved explicitly in the binary. As demonstrated in Listing 4.4, the compiler optimizes reference-based access into register operations, whereas raw pointer dereferencing produces additional memory access instructions.

4.4.6 Accessing Union Fields

Unions are special types that allow multiple fields of different types to share the same memory location. While union types are powerful and flexible, improper initialization or access can lead to undefined behavior, such as reading uninitialized memory or interpreting bits with an incorrect type.

```

1 union MyUnion {
2     f1: u32,
3     f2: i64,
4 }
5 let mut x = MyUnion { f1: 1 };
6 x.f1 = x.f1 + 1; // mov DWORD PTR [rsp],0x2
7 x.f2 = x.f2 + 1; // inc QWORD PTR [rsp]

```

Listing 4.5: Example of accessing union fields in Rust. Each field access generates assembly with different operand sizes, reflecting their types. Full example can be found at [Godbolt](#)

```

1 #[target_feature(enable = "aes")]
2 ...
3 for &key in &keys[1..KEYS - 1] {
4     b = _mm_aesenc_si128(b, key);
5 } // aesenclast xmm0, XMMWORD PTR [rip+0x3d533]

```

Listing 4.6: Example of using AES-NI instructions in unsafe Rust, resulting in `aesenclast` instructions in assembly. Full example can be found at [Godbolt](#)

At the binary level, union accesses can be distinguished by the operand size of the generated instructions, which reflects the accessed field’s type. As shown in Listing 4.5, accessing the 32-bit field results in a `DWORD` instruction, while accessing the 64-bit field results in a `QWORD` instruction. RUBY leverages these differences in operand size to infer possible field types in union-based memory locations and to detect potentially unsafe or type-violating operations.

4.4.7 Calling Functions with Hardware Features

Rust provides the `std::arch` module to enable direct use of specialized hardware instructions, such as SIMD and AES-NI, for performance-critical operations. However, these instructions depend on specific CPU features and may not be supported across all hardware platforms. To manage compatibility, Rust uses the `target_feature` attribute at the function level to indicate the required hardware capabilities. Executing such instructions on unsupported hardware can result in undefined behavior. In Listing 4.6, the AES-NI instruction `_mm_aesenc_si128` is utilized to enhance the performance of AES encryption.

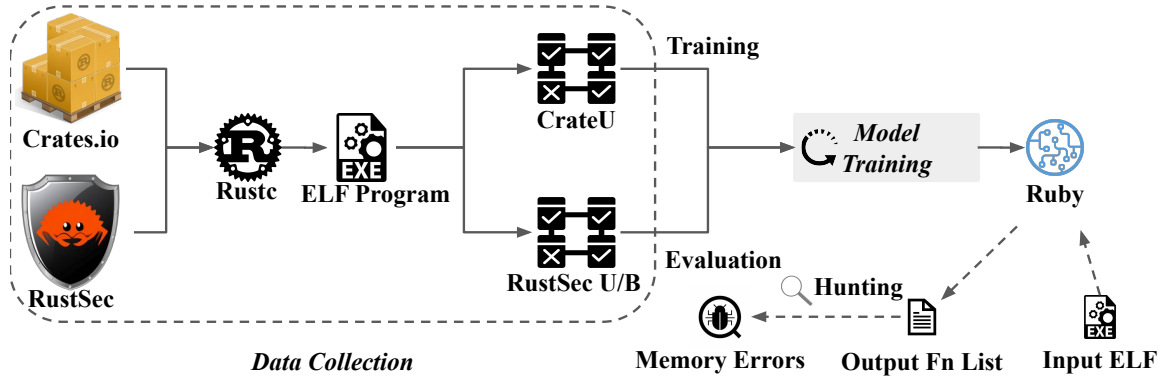


Figure 4.1: The workflow of RUBY as an unsafe Rust classifier. RUBY takes a Rust binary program as input and outputs a list of unsafe functions, which can be prioritized for further vulnerability analysis.

This leads to the inclusion of the `aesenc1ast` instruction within the binary, which is tailored for the `x86_64` architecture. These hardware-dependent instructions are preserved in the final binary and can be identified during analysis, allowing RUBY to detect functions relying on unsafe hardware features.

4.4.8 Phantom Unsafe Operations

Certain unsafe operations in Rust, such as `CastPointerToInteger`, `MutLayoutConstrainedField`, and `BorrowLayoutConstrainedField`, are considered **phantom**—they exist in the Rust compiler’s internal semantics (e.g., in the `rustc` implementation) but are virtually absent in real-world Rust code. Our dataset, which includes over 150K crates from crates.io, contains no observed instances of these operations. Due to their rarity and limited practical use in Rust development, RUBY is not trained or designed to detect these phantom unsafe operations in binary programs and excludes them from its analysis scope.

4.5 Implementation

Building on the insights of binary representations in section 3.4, we propose RUBY, a machine learning-based tool for detecting unsafe operations, aimed at identifying unsafe

constructs in Rust binaries. The workflow of RUBY is shown in Figure 4.1: Trained with an open-source dataset collected from the Rust community, RUBY accepts the raw Rust binary program as input and generates a prioritized list of unsafe functions for further analysis. In this section, we present the design choices and implementation details of RUBY.

4.5.1 ML-Based Detection

Although different binary representations exist for unsafe operations, recovering them from the Rust binary is not straightforward. The challenges are summarized below:

Heterogeneous Root Causes. For the different unsafe operations in Rust, there are distinct root causes: some are related to the type system, such as pointers, unions, and type layout; some arise from shared global variables; and others are tied to specific hardware features or instructions. Capturing different unsafe labels presents a challenge because it necessitates the development of various static analysis algorithms to identify each of them.

Ambiguous Safe/Unsafe Boundaries. The boundary between safe and unsafe operations is challenging to distinguish in compiled binary programs, presenting difficulties for traditional static analysis in identifying such patterns. This challenge arises for two primary reasons. First, Rust’s type system and information regarding unsafe operations are lost during early compilation stages (specifically when transforming into LLVM IR), obstructing the recovery of type information. Second, safe and unsafe operations are specific to the Rust language; once compiled into binary programs, a clear boundary becomes ambiguous. For instance, in Listing 4.1 and its optimized binary Listing 4.4, while dereferencing a raw pointer is deemed unsafe in Rust, dereferencing a reference is considered safe. Nonetheless, both pointers and references are represented as memory addresses within binaries, which complicates the static analysis to distinguish these subtle differences.

Diverse Architectures and Compilation Toolchains. According to the threat model in section 4.3, there is a lack of information regarding the binary’s compiler versions and architectures, as safe and unsafe Rust are language specifications independent of architectures and

Table 4.2: Dataset statistics in the number of functions/records under x86_64 architecture. The RustSecU contains safe/unsafe labels and RustSecB contains bug labels.

Label	CrateU	RustSec U/B
safe	689, 105, 165 (76.28%)	9, 001, 339 (79.16%)
unsafe	214, 246, 312 (23.72%)	2, 370, 234 (20.84%)
no-bug	-	1, 815, 230 (99.98%)
bug	-	302 (0.02%)

compilers. Nonetheless, varying architectures and compiler toolchains can lead to substantial differences in the binary. For instance, ARM and x86_64 architectures possess unique instruction sets; Rust updates its backend compiler’s LLVM version approximately every five releases. These differing instruction sets and version discrepancies present challenges in the detection and tracking of unsafe regions through conventional methods.

To address the challenges outlined, we propose employing machine learning techniques to analyze input assembly code, capturing subtle differences and classifying types of unsafe operations. Machine learning models have demonstrated their power and effectiveness in handling binary assemblies, rendering them particularly suitable for auditing assembly code and detecting minor discrepancies [56, 57, 58]. Regarding various architectures and compiler versions, models can be trained with diverse inputs tailored for specific architectures and fine-tuned to adapt to changes in compiler versions.

4.5.2 Dataset Collection

Table 4.2 summarizes the statistics of the datasets in the number of functions in binaries along with their labels.

Crate. The Crate dataset, denoted by *CrateU*, is a set of pairs consisting of a function in binary and the corresponding unsafe labels, *i.e.*, $\{(x_1, u_1), \dots, (x_m, u_m)\}$, where x_i is a function in binary, u_i is a set of safe or unsafe labels (*i.e.*, $u_i = \{0\}$ for “safe” and $u_i \subseteq \{1, \dots, 12\}$ for “unsafe” root causes in Table 4.1), m is the total number of function and label pairs. The dataset is generated from all the Rust crates from crates.io, encompassing a

total of 107,460 crates.

RustSec. To further demonstrate the connection between unsafe Rust and memory safety errors, we created a novel dataset that contains real cases of memory safety bugs from the RustSec Advisory Database [9]. The RustSec dataset is a set of the tuples of a function in assembly code, unsafe labels, and a bug label, *i.e.*, $\{(x_1, u_1, y_1), \dots, (x_n, u_n, y_n)\}$, where x_i is a function, u_i is a set of unsafe labels as in CrateU, y_i is a bug label, and n is the total number of labeled functions. The combination of labeled functions only with unsafe labels is denoted as RustSecU dataset and only with bug labels is denoted as RustSecB dataset. During the construction of these datasets, we exclude the relevant crates from CrateU to ensure that the model will not be trained with them.

We collected 360 bug reports and their associated GitHub issues, commits from the RustSec Advisory Database [9] in five years. For RustSecU, we download the corresponding 257 crates and exclude them from CrateU dataset, ensuring model doesn't use them for training. After the compilation, the CrateU has around 11M functions. Furthermore, we **manually** read these reports, label the memory safety errors and filter in 121 memory safety bugs with their precise buggy locations. This memory safety bug dataset is named as RustSecB. Compared with unsafe regions, the real buggy functions are rare among the dataset, demonstrating the difficulties of bug hunting process. Overall, the RustSecB dataset has 1.8M functions and only 302 are buggy, taking only less than 0.02%. This number also reflects the challenges for finding memory safety errors in general binary programs.

Generation. To generate the CrateU dataset for training purposes, RUBY utilizes two components: a custom Rust toolchain to record unsafe locations and a binary analyzer to map instructions back to the source. The process begins by modifying the Rustc to log the locations of all unsafe operations during the compilation process. Subsequently, the modified toolchains are applied to recompile the input crates and extract the unsafe location information from both the compiler and binary programs as compilation output. Additionally, the configuration files are modified to include debugging output for all compilation targets.

After obtaining the binary programs and compilation logs, binary analyzer can utilize the DWARF debugging information present in the binary programs to map the instructions back to the source code. By comparing the source code locations of unsafe regions, the binary analyzer outputs the unsafe region addresses and labels as part of the training dataset. Through this automated approach, RUBY builds the CrateU dataset for training and RustSecU evaluation.

4.5.3 Preprocessing

Embedding Metadata. As discussed in section 3.4, detecting certain unsafe operations related to global variables and FFI functions necessitates a comprehensive understanding of the binary program’s metadata and memory layout. Thus, RUBY incorporates the binary’s metadata information: (1) the static analysis of global variables is represented as a special token <GLB> when instructions access the global variables in the `.data` and `.bss`, and (2) external function calls are signified as <EXT> when the calling instruction attempts to invoke an external function.

RUBY embeds these two metadata as special tokens in the same line as the assembly instruction and utilizes machine learning models to capture subtle differences in assembly instructions, inferring correct unsafe operations.

Tokenizing. Before the training step, RUBY first trains a customized tokenizer based on the input architecture and assembly language. Since assembly is a specialized programming language, a customized tokenizer can effectively split the assembly instructions into meaningful tokens without compromising their integrity. RUBY by default applies its analysis at the function granularity; however, for functions longer than the token limitation of the model, RUBY will segment the instructions into pieces that conform to the token limitation and perform analysis on each piece independently. After analyzing all the pieces, RUBY will consolidate the results to produce the final output.

Sampling. Due to the large number of records in CrateU dataset and limited computing

resources, RUBY we cannot perform training and evaluation on the entire dataset. Consequently, we sampled 10 million records from the entire CrateU dataset (approximately 806 million in total) for training purposes. During the sampling process, RUBY we prioritized the minor labels by attempting to include all cases while maintaining a 1:1 ratio of safe to unsafe records (the biased distribution is preserved in the validation and test datasets). Specifically, based on the training dataset, RUBY we calculate the weights of each label to support a weighted loss function and mitigate bias during the training process.

4.5.4 Model Training

We define our problem as a multi-class, multi-label classification task, where the model input is a sequence of binary instructions in assembly language, and the expected output consists of labels for the input sequences that indicate their unsafe status and reasons. We note from Rust’s unsafe definition that the labels can overlap (e.g. accessing a global mutable union structure); hence, the expected output can be either `safe` label or `unsafe` label, with at least one reason representing the root cause of the unsafe regions.

Model Definition. The goal of unsafe classification is to design a classifier that predicts whether a given function embeds unsafe blocks. In particular, let $x \in \mathcal{X}$ be a sequence of instructions represented in assembly code,

let $\mathcal{U} := \{2, 3, 4, 6, 7, 8, 9, 12\}$ be a set of unsafe labels, where the corresponding unsafe notations are defined in Table 4.1,

$u \in 2^{\mathcal{U} \cup \{0\}}$ be a subset of safe or unsafe labels, where a safe label is denoted by 0, $\hat{s} : \mathcal{X} \times \mathcal{U} \cup \{0\} \rightarrow \mathbb{R}_{\geq 0}$ be an unsafe scoring function, and $\hat{u} : \mathcal{X} \rightarrow \{0, 1\}$ be the binary unsafe classifier.

Lastly, labeled functions from a distribution are split into train, validation, and test sets *i.e.*, $S := (S_{\text{train}}, S_{\text{val}}, S_{\text{test}})$.

We consider the following parameterized function of the binary unsafe classifier based

on \hat{s} :

$$\hat{u}(x) := \begin{cases} 1 & \text{if } 1 - \hat{s}(x, 0) \geq \hat{\tau} \\ 0 & \text{otherwise} \end{cases} .$$

Here, $\hat{s}(x, 0)$ is the safe score for a given function x and $\hat{\tau} \in \mathbb{R}_{\geq 0}$ is a threshold for unsafe classifier; thus, if the risk $1 - \hat{s}(x, 0)$ is greater than the threshold $\hat{\tau}$, we consider a function x to be unsafe.

Model Structure. We use RoBERTa-large [59] as the backbone of the unsafe classifier, which is the long-standing stable masked language model based on transformers [60] and its input is assembly code. On top of this backbone model, we attach a classification header for the unsafe classifier. The entire unsafe classifier is trained by minimizing the loss of cross entropy in the training set S_{train} for each unsafe and safe label. In addition, we weighted each label’s loss by considering the distribution in the training dataset to eliminate the effects of the unfair label distribution.

4.5.5 Trustworthy Thresholding

After the training process, we get the unsafe classifier, noted at \hat{u} . Now we describe how to pick the threshold $\hat{\tau}$ with a guarantee of correctness on the recall of \hat{u} . In particular, choosing a threshold for a classifier is a classic problem [61], where heuristic methods are mostly considered. We consider a rigorous thresholding approach that comes with a PAC guarantee based on conformal prediction [62, 63].

PAC Algorithm. We adopt the PAC conformal set algorithm [64, 65] for thresholding. Let $\bar{\theta}$ be the upper Clopper-Pearson (CP) bound [66], where the binomial parameter μ is included with high probability, *i.e.*, $\bar{\theta}(k; m, \delta) := \inf\{\theta \in [0, 1] \mid F(k; m, \theta) \leq \delta\} \cup \{1\}$, where $\mathbb{P}_{k \sim \text{Binomial}(m, \mu)} [\mu \leq \bar{\theta}(k; m, \delta)] \geq 1 - \delta$. Here, $F(k; m, \theta)$ is the cumulative distribution function of the binomial distribution with trials m and the probability of success θ . The

threshold $\hat{\tau}$ is obtained by solving the following optimization:

$$\hat{\tau} = \arg \max_{\tau \in \mathbb{R}_{\geq 0}} \tau \quad \text{subj. to} \quad \bar{\theta}(k; |S_{\text{cal}}|, \delta) \leq \varepsilon, \quad (4.1)$$

where S_{cal} is the set of unsafe functions in S_{val} , *i.e.*, $S_{\text{cal}} := \{(x, u) \in S_{\text{val}} \mid u \neq \{0\}\}$, and k is the number of unsafe functions that are missed by a threshold, *i.e.*, $k := \sum_{(x,u) \in S_{\text{cal}}} \mathbb{1}(1 - \hat{s}(x, 0) < \tau)$. Intuitively, the interval $[\hat{\tau}, \infty)$ contains the most unsafe scores $1 - \hat{s}(x, 0)$ for $x \in S_{\text{cal}}$. If an downstream analyzer wants to have 90% recall on unsafe functions, ε is set by 0.1; if the analyzer wants this desired recall level to be strictly satisfied, δ needs to be small, where we use $\delta = 10^{-3}$. See ?? in ??.

Theorem 2. *Let $\hat{\tau}$ be the solution of (Equation 5.1). For any \hat{s} , we have*

$$\mathbb{P}_{(x,u)} [1 - \hat{s}(x, 0) \geq \hat{\tau} \mid u \neq \{0\}] \geq 1 - \varepsilon$$

with probability at least $1 - \delta$.

Theory. The threshold $\hat{\tau}$ of (Equation 5.1) ensures a desired recall of unsafe functions across an unsafe function distribution; due to space constraint, we defer the proof to ??.

4.5.6 Fine-tuning for Different Toolchains

To resolve the challenge from different compilation toolchains, RUBY utilizes fine-tuning. In particular, we first train our unsafe scoring function model \hat{s} on a CrateU training set, built by rustc toolchain version 1.67.0. Then, given a RustSecU set, we hold out 20% of them and use them as a fine-tuning set built by rustc 1.57 and 1.75. This fine-tuned model is evaluated on the remaining RustSecU set built by all three toolchains (*i.e.*, 1.57, 1.67, and 1.75), demonstrating \hat{s} 's robustness across different compiler versions.

4.6 Evaluation

To demonstrate the effectiveness of RUBY, we conducted a comprehensive evaluation motivated by the following questions:

- **RQ1:** How effective is RUBY for unsafe Rust classification?
- **RQ2:** How does unsafe Rust help for reverse engineering?
- **RQ3:** How robust is RUBY when dealing with different architectures and different compiler toolchains?
- **RQ4:** How effective is RUBY’s guidance on practical vulnerability hunting process?

Hardware Settings. We launch our experiment on a machine with 128-core AMD EPYC 7452 processors and 8 NVIDIA RTX A6000 GPUs running under the Ubuntu 22.04 operating system. For ARM evaluation, we use an ARM Neoverse-N1 CPU with 80 cores.

4.6.1 RQ1: Unsafe Classifier Evaluation

To answer RQ1, we first evaluate RUBY in the context of the unsafe Rust classification task.

Dataset. We utilize the CrateU dataset for both training and evaluation. The dataset is partitioned into 60% for training, 20% for validation during training, and 20% for testing. We also conduct further evaluation RUBY on RustSecU, which are excluded from CrateU to ensure that RUBY never learned from those data.

Metrics. We use the precision recall curve to evaluate unsafe classification. In particular, the precision and recall of the unsafe classifier are computed as follows:

$$\begin{aligned} \text{(precision)} &:= \frac{\sum_{(x,u) \in A_{\text{test}}} \mathbb{1}(u = 1 \text{ and } \hat{u}(x) = 1)}{\sum_{(x,u) \in A_{\text{test}}} \mathbb{1}(\hat{u}(x) = 1)} \text{ and} \\ \text{(recall)} &:= \frac{\sum_{(x,u) \in A_{\text{test}}} \mathbb{1}(u = 1 \text{ and } \hat{u}(x) = 1)}{\sum_{(x,u) \in A_{\text{test}}} \mathbb{1}(u = 1)}, \end{aligned}$$

where A_{test} is a RustSecU test set. Note that by choosing a threshold via subsection 4.5.5,

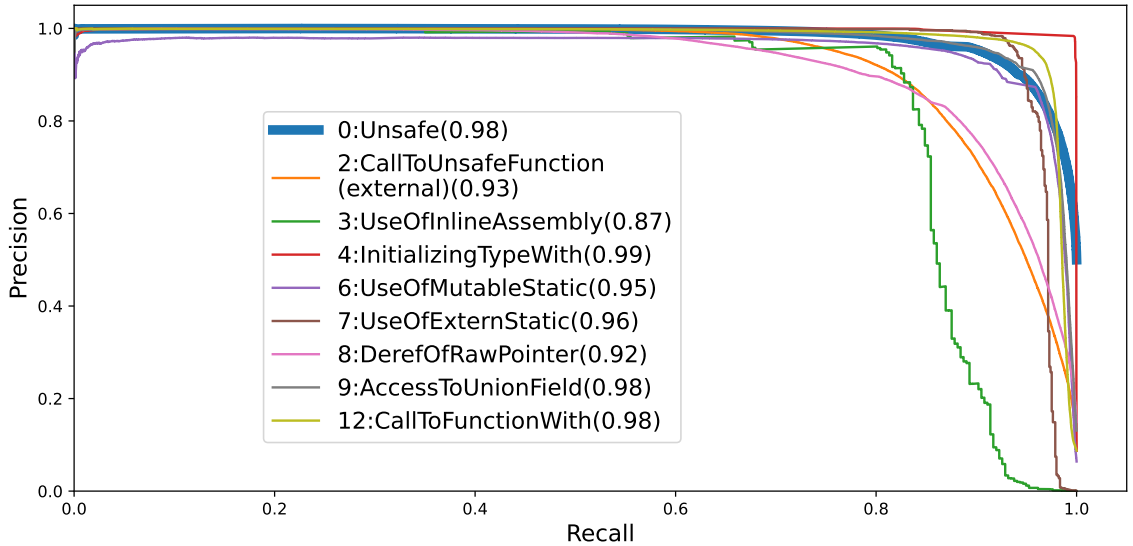


Figure 4.2: Precision-recall(AUPRC) evaluation on x64 binaries in CrateU dataset. RUBY achieves relatively high scores on each label and 0.98 on the unsafe detection.

one point in the precision-recall curve is chosen for the final evaluation.

Evaluation on CrateU

We first evaluate RUBY over CrateU for the unsafe classification task. Figure 4.2 presents the precision-recall curve in RustSecU for each type of unsafe.

Unsafe Classifier. The area under the precision recall curve (AUPRC) of an unsafe classifier is 0.98 for the overall safe/unsafe classification, demonstrating that unsafe blocks in the Rust binary are identifiable. Besides, RUBY applied weights to eliminate the biased distribution of unsafe labels, achieving higher scores in all unsafe Rust classification tasks.

Trustworthy Thresholding. The precision-recall curve shows the trend of precision and recall with a varying threshold $\hat{\tau}$; however, this threshold should be chosen in practice. We use the trusted thresholding algorithm proposed in (Equation 5.1) for $\hat{\tau}$, and a chosen threshold provides 91.75% recall of unsafe functions, which is larger than the desired recall of 90%, as expected. This suggests that reliable thresholding provides the desired guarantee of recall, controlled by ε . In practice, we desire to have a list of functions containing the

Table 4.3: The RUBY’s final precision and recall for each label under PAC thresholding selection.

ID	Name	Precision(%)	Recall(%)
0	Unsafe	93.84	91.75
2	CallUnsafeFn (external)	91.21	80.95
3	UseInlineASM	95.45	81.01
4	InitialType	98.36	99.70
6	UseMutStatic	92.34	91.32
7	UseExtStatic	98.25	92.67
8	DerefRawPtr	86.94	83.32
9	AccessUnion	94.55	92.23
12	CallFnWith	96.02	95.25

Table 4.4: Comparison between RUBY and static binary analysis. The static analysis introduces more false-positive cases due to difficulty of identifying code regions from Rust and ELF framework.

Unsafe Label	Static Analysis		RUBY	
	Precision	Recall	Precision	Recall
CallExtFn	0.374	0.877	0.912	0.811
UseMutStatic	0.428	0.893	0.923	0.913

desired rate of unsafe functions, so we empirically demonstrate that the proposed algorithm achieves this goal. For detailed precision/recall of each unsafe label, please check Table 4.3.

Evaluation on RustSecU

To further demonstrate the classification performance of RUBY, we evaluated RUBY in the RustSecU dataset. The crates in RustSecU are excluded from CrateU prior to training, so RUBY has never seen them before. The evaluation result is shown in Figure 4.3. In general, RUBY achieves the 0.98 AUPRC score with precision 95.21% and recall 95.09%, showing its ability to recover unsafe Rust operations from unknown assembly.

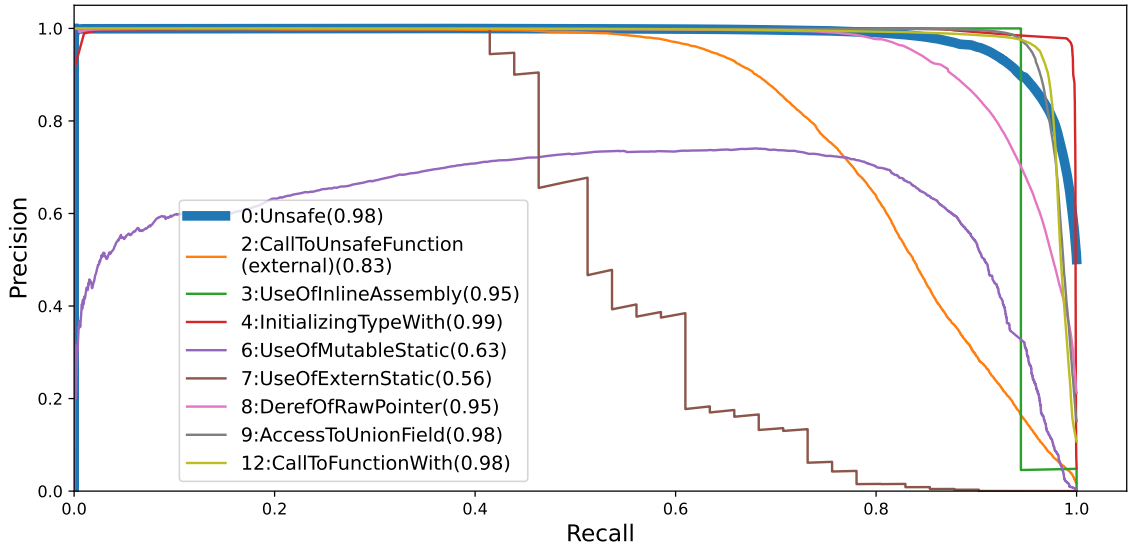


Figure 4.3: Precision-recall (AUPRC) evaluation on x64 binaries in RustSecU dataset, RUBY achieves 0.98 AUPRC score with a precision 95.21% and recall 91.09%, similar to the CrateU dataset result.

Comparison with Static Analysis

Unlike source code based tools, Rudra [44] and MIRChecker [46], RUBY does not rely on Rust source code to analyze unsafe regions. To demonstrate the difficulties of recovering unsafe Rust from optimized binaries, we selected two labels: `CallToExternalFn` and `UseOfMutableStatic` and compared RUBY’s performance against binary program analysis on the CrateU. Specifically, by analyzing the function tables and global variables, it is straightforward for static analysis to list suspected code regions. However, as the results show in Table 4.4, static analysis exhibits much lower precision compared to RUBY. The root cause of these false-positive cases is that most of them originate from ELF frameworks, which complicates the ability of traditional static analysis to accurately identify the code regions from frameworks or source code.

4.6.2 RQ2: Assistance in Reverse Engineering

The ultimate goal of RUBY is to accelerate the bug hunting process for reverse engineers; therefore, to further demonstrate the benefit of RUBY in RQ2, we evaluate RUBY in the

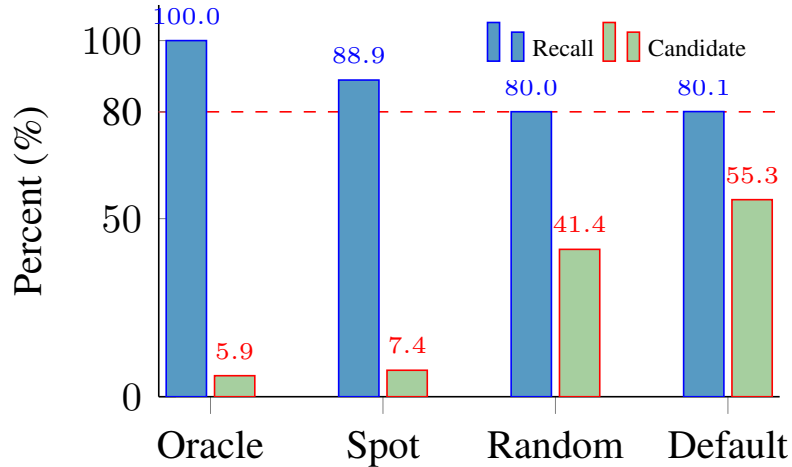


Figure 4.4: RUBY’s overall performance for assisting reverse engineers to narrow down search space. We fix target recall at 80 % and compare the candidate counts (lower is better).

RustSecB dataset to show its ability to accelerate the bug hunting process. Then we measure the analysis speed to show its practical applications.

Assisting Reverse Engineering

The ultimate objective of RUBY is to help reverse engineers narrow the potential search space. Therefore, to show RUBY’s efficiency, we compare RUBY’s guidance with following methods on the RustSecB dataset:

- **Oracle:** using source code to get all unsafe operations;
- **Random:** reverse engineers randomly pick functions;
- **Default:** reverse engineers uses the default ordering from ELF tools for analysis.

We set our target recall to be at least 80% and the result shows RUBY can minimize the searching space to only 7.43% and guarantees 91.75% of unsafe operations inside, details are in Figure 4.4. Among the four methods, RUBY performs close to the optimal case: with only 1.25x coverage overhead. Compared to randomly searching or prioritizing third-party crates, RUBY achieves 4-6x benefits.

Table 4.5: Evaluation of RUBY’s unsafe classification on large applications. RUBY achieves high AUPRC scores on all applications and even 0.907 on deno.

category	name (LOC)	AUPRC (\uparrow)
Web browser	servo(11.10M) [67]	0.890
Ruby interpreter	artichoke(1.93M) [68]	0.839
Python interpreter	rustpython(8.02M) [69]	0.832
JavaScript runtime	deno(9.36M) [69]	0.907

Performance Overhead

RUBY disassembles the input binary program and iterates over each functions to find the unsafe regions. In theory, its analyses time complexity is $O(n)$ where n is the size of the binary program. In practical, RUBY is affected by the bootstrapping process like decompiling stage and model loading stage. To show RUBY’s performance, we sampled 200 binaries based on their size and applies RUBY on them with single CPU core and single GPU card. It takes RUBY around 10 hours to finish the analyses. Considering CrateU has around 100K binary programs, it only take RUBY for around one week to analyze all the binary programs in crate.io with 32 single CPU and GPU processes.

Real World Applications

We evaluated RUBY in real-world applications to demonstrate its capability in analyzing large binary programs that exhibit complex logic. In particular, we choose three types of binaries: `servo` as a web browser, `artichoke`, `rustpython` as high-level language interpreters, Ruby and Python accordingly, `deno` as a JavaScript and TypeScript runtime³, where these binaries are not in our dataset, while related packages might be included (*e.g.*, `smallvec` for `servo`). We additionally count the lines of Rust code in the target repository to evaluate our model’s performance. Since Rust utilizes `cargo` to manage its dependencies, we employ `cargo vendor` to download all dependencies and subsequently count the code

³We use `servo` with commit 16da1c2, `artichoke` with commit 4c72aba, `RustPython` with commit 3b6db8e and `deno` with commit 8c2f1f5.

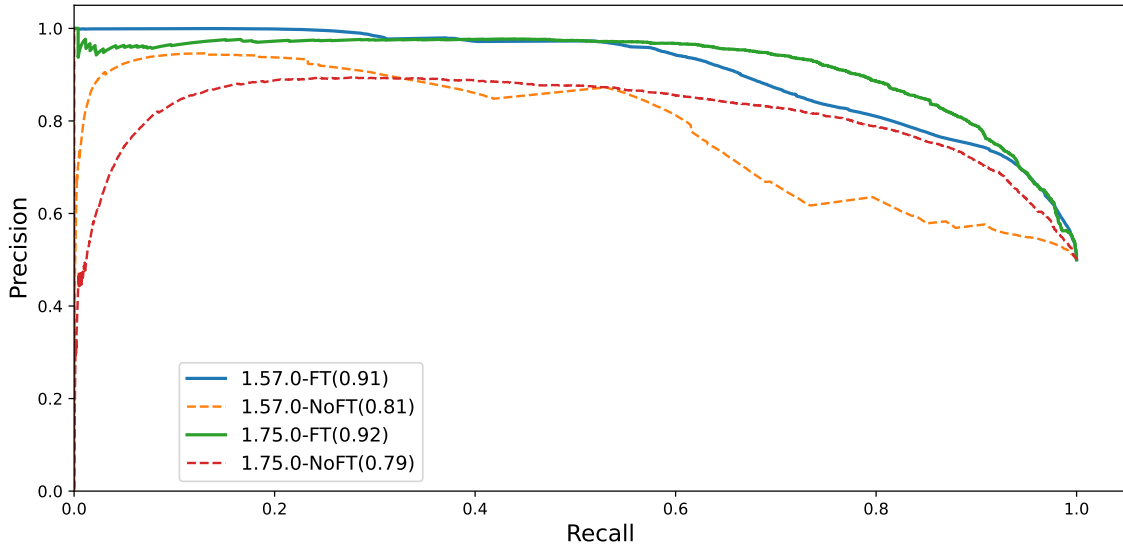


Figure 4.5: Precision-recall (AUPRC) evaluation on different compiler versions of unsafe Rust classification. With the help of fine-tuning, RUBY improves the scores to 0.91 for 1.57.0 and 0.92 for 1.75.0.

lines, which encompass both the project’s source code and all its dependencies.

Table 4.5 shows the evaluation results for each application and RUBY’s AUPRC score. Overall, RUBY performs well in AUPRC, achieving over 0.907 for deno and above 0.83 for the other applications. For language interpreters like artichoke and rustpython, RUBY’s performance is influenced by the various system calls and low-level APIs supported by the target language. We note that all of these applications contain more than 1M lines of Rust code, and RUBY can complete the analysis of these projects in three hours.

4.6.3 RQ3: Robustness Evaluation

By the definition of unsafe Rust in [??](#), it is a language-level definition that is expected to be independent of different compiler toolchains, architectures, etc. To show the robustness of RUBY, we evaluate RUBY across different compiler toolchains (Rustc 1.57.0, 1.67.0, and 1.75.0) and architectures(x64 and ARM).

Different Compiler Toolchains

We first demonstrate how fine-tuning aids RUBY in managing various toolchains of Rust and in recovering the unsafe regions.

Setup. The CrateU dataset is built on customized Rustc 1.67.0 [70], which uses LLVM-15 as the backend. To further demonstrate RUBY’s robustness, we collected the RustSecU dataset using Rustc 1.57.0 [71] with the LLVM-13 backend and Rustc 1.75.0 [72] with the LLVM-17 backend, fine-tuning and evaluating RUBY’s performance on both versions and comparing it with the 1.67.0 version. Since Rustc converts MIR into LLVM IR and leverages LLVM to optimize and generate instructions, different LLVM backend versions can produce varying outputs. We sampled only 20% of the data from both Rustc 1.57.0 and Rustc 1.75.0 and fine-tuned our model for two epochs, completing the process in less than an hour.

Result. The comparison of the unsafe classification task is presented in Figure 4.5. The results indicate that prior to fine-tuning, the model’s score diminishes by approximately 0.10 due to the compiler toolchain changes. With the help of the fine-tuning process, RUBY is able to capture minor changes across different compiler versions, achieving high scores among all compiler versions: 0.91 for 1.57.0 and 0.92 for 1.75.0.

Different Architectures

Besides the toolchains, we further explore RUBY’s performance under different architectures.

Setup. According to the definition of unsafe Rust, unsafe information is lost in the early stages of compilation when Rustc ports its MIR to the LLVM IR, indicating the architecture independence of unsafe betrayal. To validate this important property, we conduct a similar machine learning pipeline: dataset collection, preprocessing, model training, and evaluation on ARM architectures.

Result. The results are shown in Figure 4.6. RUBY demonstrate high performance on both

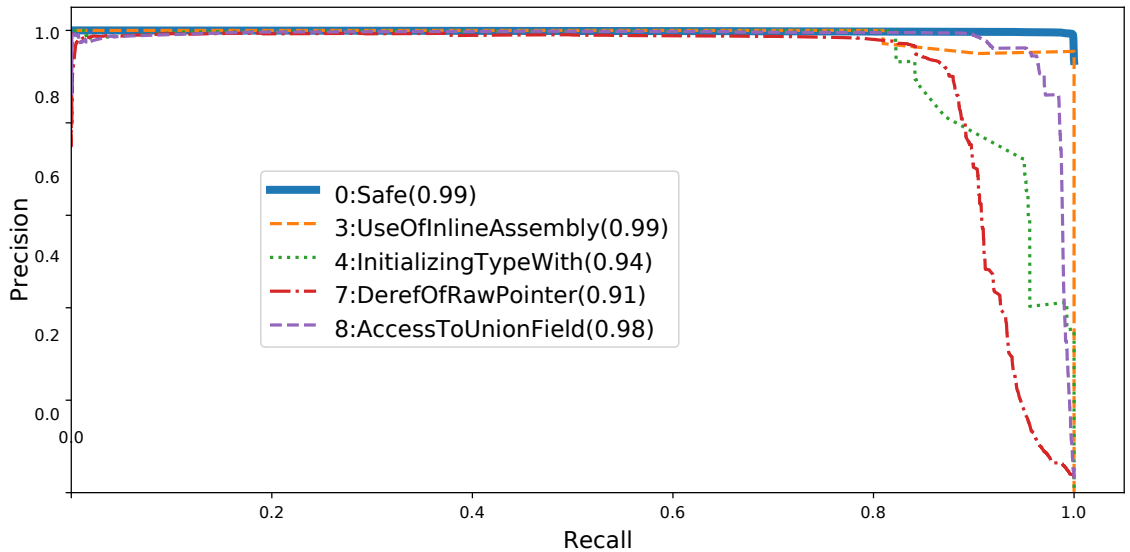


Figure 4.6: Precision-recall (AUPRC) evaluation on ARM binaries in CrateU dataset. RUBY achieves high scores as x64, showing its robustness on different architectures.

ARM and x64 architectures, indicating the architectural independence of unsafe Rust. Some improvements observed in certain labels can be attributed to ARM’s fixed-length instruction set, which offers a simpler assembly language relative to the variable-length instruction set of x64.

4.6.4 RQ4: Bug Hunting Applications

To demonstrate RUBY’s practical usage in the vulnerability hunting process, we design two evaluations to show that RUBY’s guidance can assist both extensive static and dynamic analyses with appropriate targets, thereby reducing the overall time required to identify the bug. Finally, we applied RUBY on the latest Pixel Android systems to assess its end-to-end effectiveness.

Guidance for Angr Static Analysis

Angr[52] is a powerful binary analysis tool that finds bugs through symbolic execution. However, due to the complexity of the programs, performing symbolic execution is time and resource-consuming as it involves the exploration of different paths in the program.

Table 4.6: Angr analysis performance for Rust binaries in different architectures. T0 denotes for timeout and failed to find the bug and N/A denotes for failed to get the buggy binary. RUBY can save 57.95% of time to find the same bug compared with baseline in x64 and 61.4% for ARM binaries. Compared with the oracle baseline including source code, RUBY is only 2.48x compared with oracle method with source code in x64 and 1.19x in ARM.

RUSTSEC	Name	x86_64			ARM		
		Baseline	Oracle	RUBY	Baseline	Oracle	RUBY
2021-0015	search_error	7060.73	660.14	1607.65	29028.36	3491.61	1465.16
	excel_to_csv	4668.81	343.36	424.62	2548.93	6750.60	343.76
2020-0043	bench-server	43200(TO)	2908.19	20565.21	22305.18	19151.26	6781.42
	external_shutdown	43200(TO)	18805.00	39182.24	N/A		
2021-0009	crosstalk	26816.39	244.69	5601.20	N/A		
2021-0088	worldbank	1641.82	2264.12	3269.44	3161.93	3725.85	2972.57
2021-0092	extension1	8672.45	263.82	506.95	3439.36	317.63	616.76
	extension2	10425.88	1351.28	24621.30	4495.87	1391.97	4837.85
	stream	6283.93	4.70	423.35	2385.22	887.84	484.90
2021-0094	predefined	6551.97	5050.39	4598.43	14804.18	495.10	17458.97
	file_watcher	27198.87	3982.78	11730.89	15422.56	3811.30	13806.29
2021-0090	texture	43200(TO)	8618.49	4396.34	N/A		
2021-0085	binjs_dump	11034.37	481.09	546.64	33386.22	771.72	589.74
	binjs_decode	43200(TO)	3043.78	1603.98	3368.33	2636.86	2404.86
average(seconds)		20225.37	3430.13	8505.59	12213.29	3948.34	4705.66

Therefore, RUBY can be integrated as a guiding framework for such heavy analyses, enabling the prioritization of suspect functions given the limited time and resources.

To demonstrate the efficiency, we collected the benchmark from the RustSec dataset and manually filtered out bugs that were not shown in the binary programs. We collected these memory safety bugs based on RustSecB and successfully built 14 vulnerable binaries with bugs. We set up our Angr analyses based on QueryX [73]’s memory safety analysis script, identifying heap/stack overflows, use-after-free, and access out of bounds errors, and added detection of uninitialized memory access. We established targets for each function in the binaries with a total limitation of 12 hours per binary and 5 minutes per function. We compared our approach with the baseline, which provided no guidance, and the unsafe oracle from the source code; the results are shown in Table 4.6.

Overall, RUBY is close to the unsafe oracle result, with only 1.48x overhead to reach the oracle case with additional source code information. Compared to the baseline approach,

Table 4.7: RUBY provides guided targets for AFLGO. On average, RUBY can save 21.26% of the fuzzing time to find the crash.

Issue	Target	Baseline	RUBY Guided	Ratio
🐛	rmpv	7.80	6.20	79.4%
🐛	asn	86.78	35.04	40.4%
🐛	tiff	158.44	132.58	83.7%
🐛	serde	5081.94	4931.55	97.0%
🐛	proc_macro2	10360.74	8048.14	77.7%
🐛	boa	27791.16	26474.50	95.3%
🐛	cpp_demangle	31128.52	30317.32	97.4%
	average	10659.34	9959.88	78.7%

RUBY saves 57.95% of time to find the same bugs, largely saving time and resources during the vulnerability hunting process. To demonstrate architecture independence, we also performed the same evaluation on the ARM binaries. Similarly to the x86_64 result, RUBY saves 61.4% on ARM architectures to compared with the baseline approach to find all the bugs and is closer to the oracle method, only 1.19x overhead.

Guidance for Directed Fuzzing

To demonstrate the effectiveness of RUBY for dynamic analysis, we apply the result of RUBY to a directed fuzzer as its target function to show the direction of RUBY for dynamic analysis. Directed fuzzers leverage control flow graph and static analysis to compute the instruction’s distance to target function and prioritize the corpus that reached closer places. Specifically, we use AFLGO [53] as our directed fuzzer and we use the trophy cases found by cargo fuzz as our benchmark. Among the 7 reproducible bugs with their harnesses, we first launch the normal AFL to fuzz them. then apply RUBY to these binaries and launch the AFLGO with the RUBY’s output as the target. The AFLGO is purely built on AFL without adding additional fuzzing optimizations except directed fuzzing. We count the wall time of the fuzzers that encounter the first crash as a result and repeat the fuzzing process three times to get the mean value. The result in shown in Table 4.7, on average, with the

help of RUBY, AFLGO can save 21.26% of time to find the same bug as normal AFL.

Android Rust Library Fuzzing

To enhance the memory safety of the Android system, Rust has been introduced and has demonstrated its effectiveness in the development of the AOSP. However, some third-party crates are also incorporated as dependency libraries in the Android release, which may contain vulnerabilities. To evaluate RUBY’s end-to-end effectiveness, we applied RUBY on the Android system and provide guidance for black-box fuzzing of Rust library binaries. Specifically, we consider the following steps: given a crate library in .so, (1) we feed all the functions into RUBY, (2) based on RUBY’s output sequence, we select the top 5 functions and manually develop the fuzzing harnesses, and (3) conduct black box fuzzing with AFL++ over the harness code. For each target, we run for a maximum of 24 hours.

Finally, with the help of RUBY, we successfully identified five bugs: two stemming from character boundary issues, one concerning an out-of-bounds access, one related to an unexpected unwrap in the library, and one resulting in a panic abort. We reported all the PoC code with corresponding inputs to Google, and all the bugs were confirmed by the maintainers and are currently awaiting patches at the time of writing. Due to ethical considerations, we omitted the details related to the five bugs.

4.7 Limitation

RUBY’s limitation can be categorized into two parts: the limitation inherited from the toolsets and methodologies used by RUBY, and the limitation related to our implementation.

4.7.1 Inherited Limitations

First, RUBY relies on Rustc to generate the corresponding dataset for training purposes. For unsafe operations that Rustc cannot detect, RUBY cannot identify them as well. Second, RUBY leverages machine learning to recover unsafe Rust, the loss during the training process

cannot be recovered by RUBY. RUBY experiences instances of false positives and false negatives compared to the oracle method.

4.7.2 Implementation Limitations

RUBY's implementation limitations are mainly from the data collection and model training steps. Currently RUBY leverages the specific Rustc to automatically generate the CrateU dataset. Because of the compiler differences and architecture requirements, CrateU dataset may miss several crates and the unsafe functions are missed by RUBY as well. Furthermore, constrained by hardware capacity, RUBY utilizes only 10M out of a 903M dataset for training. Finally, as a prototype, RUBY uses RoBERTa [59], which is a classic BERT model specialized for classification tasks. We leave the task of training a larger, specialized model on the full dataset for future work.

CHAPTER 5

TO BE SAFE OR UNSAFE: FINDING LOGICALLY UNSAFE RUST VIA LLM

5.1 Overview

Rust is an emerging system programming language that can guarantee memory safety while preserving high performance. However, to provide more flexibility, Rust introduces unsafe Rust for developers to perform certain unsafe operations, where the security guarantee does not hold. As a result, unsafe Rust code often leads to security issues. To better manage unsafe Rust code, developers are required to use the unsafe keyword when performing unsafe operations, and any unsafe operations without using unsafe will trigger a compilation error. Our study, however, has revealed that the Rust compiler fails to detect some unsafe operations, noted as *logically unsafe operations*, that could lead to severe problems such as memory safety breaches and system failures.

In this paper, we conduct the first systematic study of logically unsafe operations and identify their root causes. Given the challenges associated with their detection using traditional static analysis methods, we propose COIN, a LLM-based system designed to identify these operations within Rust source code. COIN consists of (1) a logically unsafe classifier that takes the source code as input to identify whether it contains logically unsafe operations, and (2) a proof-of-concept exploit generator that assists developers in generating exploitation for triggering and verifying the identified issues. We applied COIN across 10K crates within the Rust ecosystem, uncovering 25 logically unsafe operations—10 of which have received CVE assignments.

5.2 Introduction

Despite `rustc`'s intention to identify all mislabeled unsafe operations, we show that

```

1 pub fn set_len(&mut self: Vec,
2     new_len: usize){
3     debug_assert!(new_len <= self.capacity());
4     self.len = new_len; //change the logical len
5 }

```

Listing 5.1: Incorrect `set_len()` implementation: a logically ‘unsafe’ operation cannot be detected by `rustc`.

it is not flawless—it occasionally overlooks some mislabeled unsafe operations without generating compilation errors. For instance, the `set_len` function of `Vec` in Listing 5.1 modifies the `len` field, which controls the accessible part of the vector’s memory. If this field is set incorrectly, it might result in accessing uninitialized memory, which is undefined behaviors and should be marked unsafe. However, as `rustc` regards `len` as a regular integer, the modification is not deemed unsafe, hence no compilation error occurs in Listing 5.1. We refer to such cases as *logically unsafe operations*—operations that can evade `rustc`’s checks without adding the `unsafe` keyword, but are logically unsafe in nature. Like other unsafe code, they can cause serious issues, such as memory safety violations and system crashes. *We focus on studying logically unsafe operations with their root causes, and developing reliable solutions to detect them.*

Challenges. To investigate the root causes of these logically unsafe operations, we begin by collecting a dataset from the Rust community. By examining the fundamental causes of these logically unsafe operations and scrutinizing their code, we argue that accurately identifying such issues poses significant challenges for existing automated program analysis techniques, due to the following challenges in distinguishing logically unsafe operations from normal safe operations:

- **Accommodating diverse unsafe patterns:** Logically unsafe operations exhibit a wide range of patterns and originate from multiple root causes, which contrasts greatly with the bug types commonly identified by traditional rule-based bug detection tools [44, 46, 74].
- **Comprehending code semantics:** Precisely identifying these operations requires a deep

understanding of the code’s logic at the semantic level. For instance, to identify *set_len* as shown in Listing 5.1, the detector must grasp the variable’s name (*len*) and the program logic, a task that traditional static analysis approaches struggle with.

- **Verifying the security implications:** Once these issues are identified, it is crucial to determine whether they may lead to security vulnerabilities. This often involves crafting PoC exploits to demonstrate the bug. However, because of the diverse logically unsafe causes, building an automated exploit generator applicable for all possible cases is challenging, which is challenging for existing automated exploit generation tools [75].

Our Solution. As LLMs are good at capturing the semantics of source code [76, 77], we propose COIN, a LLM-based classifier to automatically detect potential logically unsafe operations from Rust source code and a PoC generator to automatically generate the exploit as guidance. In our context, LLMs are suitable for the following reasons: First, LLMs can address each pattern as a distinct sub-problem and autonomously learn to resolve them, thus significantly reducing the effort and resources needed to pinpoint each logically unsafe pattern through conventional methods. Second, LLMs possess a degree of understanding of source code semantics, such as interpreting variable names and usages to reason their logical functionalities. Third, LLMs are good at generation tasks, which can help in generating PoCs of logically unsafe operations, guiding developers to understand the root causes.

However, simply prompting LLM with few-shot examples for logically unsafe Rust cannot produce reliable answers. To tackle this, COIN uses LoRA to fine-tune LLM on existing logically unsafe operations. To help developers comprehend and verify the detected issues, we construct a fine-tuned PoC generator (based on a manually built PoC dataset for logically unsafe operations) to generate PoC code as guidance.

To demonstrate COIN’s effectiveness, we evaluated COIN on 10K popular crates from crates.io and COIN and successfully identified 25 logically unsafe bugs, with 10 CVEs assigned. With the help of the PoC generator, we successfully generated bug reports with PoC code samples for 20 logically unsafe functions and submitted them to the developers.

Among the 25 bugs submitted, 13 were confirmed by developers (5 of them have already been patched). Besides, through comprehensive evaluation against state-of-the-art LLM models (QWen3, Llama3.2 and GPT-4o, Claude-3.7) and traditional static analysis tools (Rudra, MirChecker and ffi-checker), we demonstrate that COIN out-performs these tools and achieves a reasonable precision (63.7%) and recall (80.4%) for logically unsafe Rust detection. In summary, this paper makes the following major contributions:

- We identified and defined logically unsafe operations, and we classified the root causes of such issues into four categories based on an empirical study of existing Rust issues. To our knowledge, we are the first to study logically unsafe operations in Rust.
- We designed and implemented COIN, a LLM-based detector for logically unsafe operations and a LLM-based PoC generator for verifying detected issues.
- We evaluated COIN on 10,000 popular Rust crates, successfully identified 25 logically unsafe bugs, and 10 of them have been assigned as CVEs. One of the bugs in `sqlite3-parser` (over 1.5M downloads and 14.6K GitHub stars) allows invalid UTF-8 strings to be propagated, leading to crashes and undefined behaviors in applications using this crate.

5.3 Threat Model

Based on the scope definition in Figure 2.1, the target of this work is to detect the area γ in Rust programs, which consists of logically unsafe operations that are missed by the `rustc`. These logically unsafe operations can trigger undefined behavior while using safe Rust,¹ thereby breaking the memory safety guarantee provided by Rust.

Challenges. Due to the difficulty of understanding the program’s logic and operations, we argue it is challenging for traditional static analysis and developers to detect these behaviors; otherwise, the `Rustc` compiler is expected to implement relative analysis. In addition, compared to other works related to Rust [78, 44, 79] focusing on explicit unsafe

¹Undefined behaviors caused by safe Rust are named as `unsound` issues.

regions in Rust, finding logically unsafe operations in safe Rust is more challenging because 1) logically unsafe operations are typically safe operations with logical meanings, which are hard to distinguish from the other safe operations and 2) compared to unsafe Rust, the search space for safe Rust is larger and more complex.

5.4 Empirical Study

We begin by examining existing logically unsafe operations to understand their root causes. We describe our study methodology and categorize the various types of logically unsafe operations with their underlying causes.

5.4.1 Method Overview

Although logically unsafe functions constitute only approximately 0.38% of all functions in the Rust ecosystem, the absolute number remains substantial, around 570K, exceeding human efforts for manual analysis. To address this, we leverage LLM to filter redundant instances, generated code, and assist in classifying the remaining 74.3K cases. The taxonomy of logically unsafe types was previously unknown. We thus adopt an iterative clustering approach using LLMs to categorize these cases into distinct classes.

We begin by employing a customized `rustc` to detect all existing logically unsafe instances. From this set, we sample 100 cases and manually define the initial classes, such as *memory invariant violations* (MI) and *foreign function interface issues* (FFI). We then prompt the LLM to classify additional cases into existing classes or new categories. Newly identified classes are incorporated into subsequent prompts. This iterative process is repeated twice, resulting in the discovery of three additional categories: *precondition checks* (PC), *data races* (DR), and *direct mapping* (DM). The distribution of each portion is shown in Table 5.1. Despite these efforts, a remaining fraction (13.8%) of cases remains unclassified. Upon manual inspection of 100 such instances, we find that all of them are developer-induced false positives or benign edge cases that serve primarily as warnings to

Table 5.1: The portion of each logically unsafe in COIN’s study.

Type	PC	MI	FFI	DR	DM	Unclassified
Portion	31.56%	14.67%	26.03%	8.98%	4.89%	13.87%

```

1 let mut vec = SvmVec::<i32>::with_capacity(5);
2 vec.set_len(4); // break invariant
3 vec.pop(); // uninitialized memory read

```

Listing 5.2: The PoC code of breaking memory invariants generated by COIN. The uninitialized memory is introduced into safe Rust.

users.

5.4.2 Breaking Logical Memory Invariants

The first class of logically unsafe operations involves breaking the logically memory invariants, especially concerning initialized and uninitialized memory. An example of this type is the *set_len* function in the *Vec* shown in Listing 5.1. Similarly, for any data structure containing uninitialized memory, modifying its logical length will allow users access to uninitialized regions, leading to the read of uninitialized memory error. The challenge of detecting such an unsafe logically operation is understanding related variables as logically memory invariants instead of normal integers. We argue that this type of error is difficult for traditional static analysis to find a fixed pattern to detect, but requires an understanding of program logic (i.e., the semantics).

To build the PoC for this type of error, we need to break the invariants and introduce uninitialized memory into safe Rust. For example, the PoC of a similar issue in *OpenCL3* crate detected by COIN is shown in Listing 5.2. The PoC first manipulates the usable memory size and introduces uninitialized memory into the *Vec*, then attempts to access the memory to trigger undefined behavior.

5.4.3 Missing Checks on Precondition

The second type of logically unsafe operations is the failure to validate the logically

```

1 unsafe fn from_utf8_unchecked(bytes: Vec<u8>)
2     -> String {
3     String { vec: bytes }
4 }

```

Listing 5.3: The utf-8 conversion function in Rust standard library. It requires an explicit unsafe to prevent the invalid values introduced into safe Rust.

```

1 struct A{}
2 impl A {
3     pub const fn as_bytes(&self) -> &[u8] {
4         // an invalid utf-8 encoding
5         [0xC0, 0x80].as_slice()
6     }
7 }
8 fn main() {
9     obfstr::obfstr!(A{});
10 }

```

Listing 5.4: The PoC code generated by COIN demonstrates how introducing invalid utf-8 encoding in Rust can trigger undefined behavior.

preconditions of input values, so that invalid values will be introduced to safe Rust and trigger undefined behavior. One example of this type of unsafe operation is the utf-8 encoding when transforming raw bytes into string in Rust, shown in Listing 5.3. In the utf-8 encoding, not all permutations constitute valid encodings; therefore, safe Rust expects developers to verify the encoding prior to conversion into a string unless it can be guaranteed that the input bytes are valid strings.

However, when developers forget to add checks before converting or miss the ways to bypass the designed input, this function can introduce invalid values into safe Rust. For instance, the PoC code in Listing 5.4 from a `obfstr` crate shows a case to bypass the designed way of using the APIs from the library and trigger this type of error. The developer allows users to pass their custom structs and convert them into strings in the program; however, when a malicious user builds a struct with invalid encodings, the library fails to perform the necessary encoding checks and consequently introduces the invalid value into safe Rust.

```

1 fn main() {
2     let s = "6.77777";
3     let sheap = s.to_string();
4     <f64 as mpv_client::Format>
5         ::from_ptr(sheap.as_ptr()
6             as * const c_void);
7 }

```

Listing 5.5: The PoC code generated by COIN. Here, the mpv crate forgets to check the integer value as pointer inputs and triggers undefined behavior.

The difficulty of detecting this type of logically unsafe error is the various preconditions in the program logic, as utf-8 is only one example. Another example is from the mpv crate and the PoC code is shown in Listing 5.5, where developers assume the input pointers to be an integer string but forget to check the value. We argue that due to the various conditions and program logics, it is challenging for static analysis to reason about the program preconditions from source code. However, LLMs can understand the program logic from both source code and contexts, such as crate descriptions and documents, thereby detecting such errors.

5.4.4 Violating Rust Constraints in FFI

The third type of logically unsafe operations arises from Foreign Function Interface (FFI) functions, as rustc is unable to extend its analysis to encompass existing libraries. Although FFI functions are unsafe in Rust, developers can create custom bindings to wrap the FFI into the safe function. However, when creating the safe binding for FFI, developers need to add additional check to ensure the safe Rust constraints and for FFI functions failed to follow the Rust constraints, developers are expected to keep the function as unsafe. For example, in png crate, the library creates a safe Rust bindings for libpng and allows users to call the safe API to load and read PNG files into Rust. However, unlike C, Rust's read function expects to have all the buffer initialized to prevent the uninitialized memory read, and this constraints are not implemented in the original C library, introducing the unsound issue

```

1 struct A{ f: File }
2 impl Read for A {
3     fn read(&mut self, buf: &mut [u8])
4         -> std::io::Result<usize> {
5         let _ = buf[0]; // uninitialized memory
6         return self.f.read(buf);
7     }
8 }
9 fn main(){
10     if let Ok(png) =File::open("test.png") {
11         let a = A{f:png};
12         spng::decode(a, spng::Format::Rgba8);
13     }
14 }

```

Listing 5.6: The PoC code of uninitialized memory read in the spng crate. The FFI function does not follow Rust’s constraint, and developers fail to enforce it.

```

1 #[doc = r"Writes raw bits to the field"]
2 pub fn bits(self, value: u8) -> &'a mut W {
3     // value can be changed to any status
4     self.w
5 }

```

Listing 5.7: The direct memory mapping function generated by svd2rust, allowing direct changes of register values in device.

shown in Listing 5.6.

Note that the FFI is a special case of logically unsafe Rust since detecting it requires a cross-language analysis between Rust and the library, which is beyond the scope of this work. Nevertheless, if the target source code or libraries are not available, COIN can still infer the library behaviors from the function signatures and developer comments, which assist developers in constructing safe bindings for FFI functions.

5.4.5 Mapping Invalid Value to Devices

Another category of logically unsafe operations is from the device drivers written in Rust, especially from svd files. As proposed as one of the open problems in [80], the boundary between safe Rust and unsafe Rust is not technically clear in embedding system

```

1 pub fn try_lock(&self) -> Option<SpinGuard<T>> {
2     // spin lock...
3 }
4 // unlock is logically unsafe
5 pub fn unlock(&self) {
6     self.locked.store(false,
7         Ordering::Release);
8 }

```

Listing 5.8: Unlock is a logically unsafe operation in Rust, the unsound code is identified by COIN from anode crate.

development. For example, the direct memory mapping is commonly used in device drivers allowing the system control the register values in device by manipulating normal memory regions. However, many devices use finite state machines to control their state changes, so not all values or transformations are valid. Therefore, when host system change the register to an invalid value through a simple integer manipulation, the device system can be invalid and trigger undefined behaviors. For example, with the help of `svd2rust` tool, developers can generate driver bindings through the hardware `svd` files. For each register in device, `svd2rust` will generate functions with memory mappings shown in Listing 5.7. Due to the difficulty of inferring valid states and their transformations, `svd2rust` only validates the correct range of the register value, without ensuring the correct status transformation when the function is called. Since the device’s internal status is far beyond `rustc`’s analysis scope, the mitigation of this type of risks is still under discussion in `svd2rust` [81].

5.4.6 Data Race

Another part of undefined behaviors in Rust is the potential data race in concurrent programs. To mitigate the potential data races, `rustc` explicitly checks access to mutable global variables and requires `unsafe` for these operations. However, in addition to accessing the global mutable variable, with the help of LLM, we identify more logically unsafe operations due to complex synchronizations. For example, in Rust’s lock API shown in Listing 5.8, the `unlock` function is logically unsafe without any `unsafe` operations inside

```

1 let mut s1 = Arc::new(SpinLock::new(5));
2 let mut s2 = s1.clone();
3 let h = std::thread::spawn(move || {
4     let mut guard = s2.lock();
5     *guard = 10; // thread 1 write
6 });
7 s1.unlock();
8 let guard = s1.lock();
9 *guard = 5; //thread 2 write
10 h.join().unwrap();

```

Listing 5.9: The PoC generated by COIN for logically unsafe unlock function in anode crate, the code triggers data races with safe Rust.

the function body. Rust leverages LockGuard to implicitly unlock the resource when it is dropped; thus, manually unlocking the resource before the LockGuard is dropped is considered unsafe since the LockGuard still holds a reference to the shared data. For the above unlock function, a PoC in Listing 5.9 generated by COIN can trigger data races in safe Rust as an unsound issue.

5.4.7 Unclassified types and False-Positives

The limitations of COIN’s study stem from the following two sources: Due to the high volume of logically unsafe operations and the unpredictability of LLM models, there exist unclassified types that the LLM has failed to categorize. Since the cases are derived from developers, there are instances of false positives that developers have conservatively marked as unsafe.

Unclassified logically Unsafe Types

For cases such as Listing 2.2, it is challenging to determine the root causes of these logically unsafe operations, resulting in some cases remaining unclassified. Furthermore, during our empirical study, we observed that many crates are not actively maintained and have limited program context, making it difficult for LLMs to infer the root causes in such scenarios.

```

1 pub unsafe fn with_mutable_key(self)
2     -> CursorMutKey<'a, K, V, A> {
3     self.inner // create a mut ref for the k, v
4 }

```

Listing 5.10: A false-positive unsafe case from Rust standard library. The function can lead to logic errors, but it will not trigger undefined behaviors in Rust.

False-positive Cases

According to Definition 4, only memory safety errors triggered by safe Rust are considered unsound issues, and other types of logically errors are not included. Therefore, if a function can only introduce logically errors without triggering memory safety bugs, Rust does not expect it to be unsafe. In practice, there are many program invariants that can lead to false-positive cases for developers to mark as unsafe. For example, in the Rust standard library, a mutable cursor on a `BTree` is explicitly marked as unsafe, because users can change the key values of the entries and break the order of the `BTree` shown in Listing 5.10. However, according to `BTree`'s document [82], breaking the order of a `BTree` will only result in a logic error, but will not result in undefined behavior. Conservatively, developers still mark this function as unsafe to warn users while using the API.

5.5 Implementation

Due to the various root causes of logically unsafe operations and the large number of similar yet safe operations, it is hard to build traditional static analysis tools to detect each unsafe operation. In this work, we propose COIN, a LLM-based logically unsafe Rust detector and an unsound PoC generator. The workflow of COIN is depicted in Figure 5.1. Based on the empirical study, we fine-tuned Llama3.2 [83] to classify Rust functions (with context) as logically unsafe. To help developers understand each issue, we manually wrote PoC examples per category and fine-tuned an LLM to automatically generate PoCs for unsound behaviors.

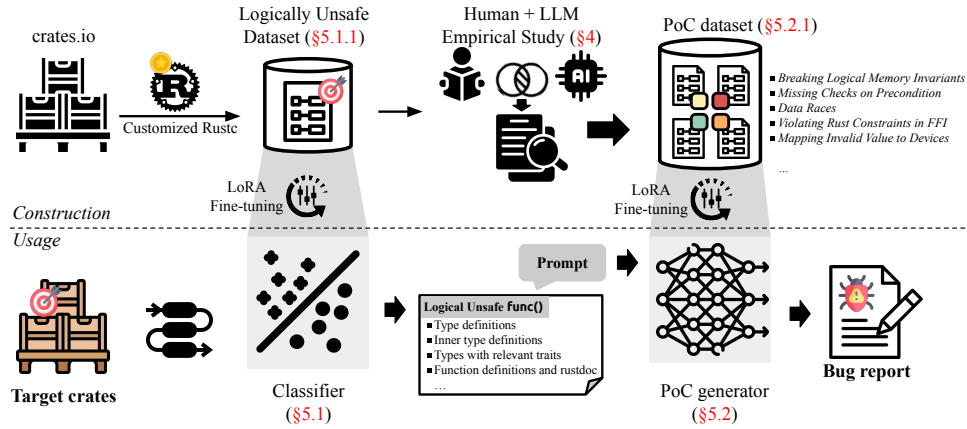


Figure 5.1: COIN’s system overview. COIN includes a logically unsafe classifier trained on the existing logically unsafe examples collected from Rust community and a PoC generator trained on a manually created PoC dataset.

5.5.1 Logically Unsafe Classifier

As shown in Figure 5.1, COIN’s first part is a logically unsafe classifier, which takes the target function and relevant contexts as input and outputs potential labels indicating whether the target function is logically unsafe.

Logically Unsafe Dataset

The dataset collected from Rust community, denoted as the logically unsafe dataset, is a set of pairs consisting of a Rust function and the corresponding logically unsafe labels, where x_i is a Rust function with its context, u_i is a set of safe or logically unsafe labels (*i.e.*, $u_i = 0$ for “safe” and $u_i = 1$ for “logically unsafe”), m is the total number of the function and label pairs. We note that during label collection, we filtered out unsafe functions that can be detected by `rustc` since they have already been marked. Thus, the remaining functions are only safe functions that may contain logically unsafe operations. The dataset is generated from all the Rust crates from `crates.io`.² Among the 150M functions we collected, only 570K functions may contain logically unsafe operations, taking less than 0.38%.

²We fixed the crate version to the latest before 11th December 2024

Context Collection

To assist model understand the function text, we build a static analyzer as a preprocessor to recursively collect the function context and remove irrelevant code. Specifically, for the target types from function signature, COIN recursively collect the relevant items including: (1) all the definitions of the types; (2) all the definitions of the inner types; (3) all the types that implement the relevant traits; and (4) all the function definitions and the rustdoc of the relevant types. COIN’s searching scope is limited to the target crate, and when a foreign trait or type outside of the current compilation scope is included, COIN stops searching for its relevant items. Meanwhile, after the context collection, COIN ranks the context priority by its depth and dynamically builds the prompt as input sequences given the token length limitation of the LLM.

Problem Definition

The goal of logically unsafe classification is to design a classifier that predicts whether a given function contains logically unsafe operations. In particular, let $x \in \mathcal{X}$ be a sequence of tokens representing normal Rust code and its relevant context, we define COIN’s unsafe classifier as a binary classification model \mathcal{M} that satisfies $\mathcal{M} : \mathcal{X} \rightarrow \{0, 1\}$, where ‘0’ represents the function being safe and ‘1’ signifies that the input function contains logically unsafe operations.

Model Architecture

We utilize the Llama3.2 model to perform this classification task by adding a customized classification header. The architecture of Llama3.2 is particularly suitable due to its advanced contextual understanding from long context length and sequence modeling capabilities. Before fine-tuning the model, a unified system prompt defining the problem and the input Rust function x are tokenized into a sequence capturing code structure and semantics. During the fine-tuning process, the token sequence is fed into the Llama3.2 model, which processes

the sequence and generates contextual embeddings for each token. Each token embedding encapsulates the syntactic and semantic information required for the classification task. We then apply a fully connected layer with a sigmoid activation on the final embedding to produce a probability score $\hat{u}(x) \in [0, 1]$, representing the likelihood that the function contains a logically unsafe operation.

Weighted Loss and Metric Calculation

To effectively tackle class imbalance between large number of safe functions and rare logically unsafe functions, we employ weighted classification. This approach ensures that the model assigns appropriate importance to the ‘unsafe’ class during training. For the loss function, we leverage binary cross-entropy loss with class weights: Given the true label $u \in \{0, 1\}$ and the predicted probability $\hat{u}(x)$ from the classification head, we assign different weights to the loss function for each class. Let w_0 and w_1 be the weights for the ‘0’ (safe) and ‘1’ (unsafe) classes respectively. The weighted binary cross-entropy loss is then computed as:

$$\mathcal{L}(u, \hat{u}(x)) = -[w_u (u \log(\hat{u}(x)) + (1 - u) \log(1 - \hat{u}(x)))]$$

where $w_u = w_0$ when $u = 0$ and $w_u = w_1$ when $u = 1$, calculated based on the training dataset’s statistics. As for evaluation metrics, standard binary classification metrics such as Precision, Recall, F1-Score, and Accuracy are computed based on the output of the classification head. These metrics provide an overall assessment of the classifier’s performance in distinguishing between safe and unsafe functions. To handle the imbalance, we focus on the precision of unsafe label classification during training and evaluation processes.

LoRA Fine-tuning

Fine-tuning large language models often requires substantial GPU memory. To reduce this requirement, we applied Low-Rank Adaptation (LoRA) to Llama3.2. LoRA freezes original weights and adds small-rank update matrices into selected linear submodules, enabling efficient adaptation with far fewer trainable parameters.

In our setup, we injected LoRA adapters into the following target modules: `q_proj`, `k_proj`, `v_proj`, and `o_proj` within each self-attention block, as well as `gate_proj`, `up_proj`, and `down_proj` in each feed-forward (MLP) block. Adapting `q_proj` and `k_proj` allows the model to refine how query and key vectors are computed for attention, thereby adjusting the attention patterns that distinguish safe versus unsafe code contexts. Updating `v_proj` modulates how value vectors carry content information across tokens, and fine-tuning `o_proj` refines how multi-head outputs are combined before passing to subsequent layers. In the MLP block, `gate_proj` and `up_proj` collaborate to expand hidden representations into a higher-dimensional space (often via a gated activation), while `down_proj` projects those intermediate features back to the original hidden dimension—together improving the model’s ability to capture task-specific patterns for classifying safe versus unsafe Rust functions.

During training, all original Llama3.2 weights remain frozen; only the low-rank adapter matrices inserted into the seven target modules are updated. We feed tokenized sequences of Rust functions with context into the model and use the final token’s representation to compute a binary “unsafe” probability via a task-specific head. We employ the AdamW optimizer with a linear learning-rate scheduler and incorporate class weights into a binary cross-entropy loss to address class imbalance. By updating only the LoRA adapters, we reduce VRAM usage by over 90% compared to full-model fine-tuning, enabling efficient experiments on limited hardware without sacrificing performance.

PAC Thresholding

The last step of a classifier is to pick a correct threshold for the inference. In particular, choosing a threshold for a classifier is a classic problem [61], where heuristic methods are mostly considered. We consider a rigorous thresholding approach that comes with a PAC guarantee based on conformal prediction [62, 63].

Algorithm. We adopt the PAC conformal set algorithm [64, 65] for thresholding. Let $\bar{\theta}$ be the upper Clopper-Pearson (CP) bound [66], where the binomial parameter μ is included with high probability, *i.e.*, $\bar{\theta}(k; m, \delta) := \inf\{\theta \in [0, 1] \mid F(k; m, \theta) \leq \delta\} \cup \{1\}$, where $\mathbb{P}_{k \sim \text{Binomial}(m, \mu)}[\mu \leq \bar{\theta}(k; m, \delta)] \geq 1 - \delta$. Here, $F(k; m, \theta)$ is the cumulative distribution function of the binomial distribution with trials m and the probability of success θ . The threshold $\hat{\tau}$ is obtained by solving the following optimization:

$$\hat{\tau} = \arg \max_{\tau \in \mathbb{R}_{\geq 0}} \tau \quad \text{subj. to} \quad \bar{\theta}(k; |S_{\text{cal}}|, \delta) \leq \varepsilon \quad (5.1)$$

where S_{cal} is the set of unsafe functions in S_{val} , *i.e.*, $S_{\text{cal}} := \{(x, u) \in S_{\text{val}} \mid u = 1\}$, and k is the number of unsafe functions that are missed by a threshold, *i.e.*, $k := \sum_{(x, u) \in S_{\text{cal}}} \mathbb{1}(\hat{u}(x) < \tau)$. Intuitively, the interval $[\hat{\tau}, \infty)$ contains the most unsafe probabilities $\hat{u}(x)$ for $x \in S_{\text{cal}}$. If an downstream analyzer wants to have 98% recall on unsafe functions, ε is set by 0.2; if the analyzer wants this desired recall level to be strictly satisfied, δ needs to be small, where we use $\delta = 10^{-3}$.

Theorem. The threshold $\hat{\tau}$ of Equation 5.1 ensures the desired recall of unsafe functions over an unsafe function distribution. Formally, this is specified by the following theorem due to the standard results in conformal prediction [84]:

Theorem 3. *Let $\hat{\tau}$ be the solution of (Equation 5.1). For any \hat{y} , we have*

$$\mathbb{P}_{(x, u)}[\hat{u}(x) \geq \hat{\tau} \mid u = 1] \geq 1 - \varepsilon.$$

5.5.2 LLM PoC Generator

Due to the various complex root causes of different logically unsafe operations, it is non-trivial for developers to understand and reason about the root causes of logically unsafe operations. Therefore, we propose an LLM-based PoC generator that takes the predicted logically unsafe function and relevant context as input and attempts to generate PoC code that demonstrates undefined behaviors.

PoC Dataset

Unlike the logically unsafe dataset, the PoC dataset cannot be automatically collected since it requires intricate construction of context to trigger the undefined behaviors. Therefore, we manually created a dataset containing pairs of existing logically unsafe functions and their corresponding PoC code. Each pair consists of a target function with a specific logically unsafe operation and its relevant context, noted as x , and we manually built a PoC code snippet as y that triggers undefined behavior. This dataset includes all the categories of logically unsafe operations and their PoCs we studied in section 5.4, ensuring comprehensive coverage of potential issues. For y , it only contains safe Rust operations and leverages the logically unsafe function to trigger the undefined behaviors. Due to the difficulty and engineering efforts of manually constructing PoC code and explanations, we only built about 20 cases per type of logically unsafe operation. We fine-tuned our PoC generator on this dataset with collected context and evaluated its performance on newfound bugs to ensure the model never learns the PoC code.

Problem Definition

The task of generating PoC code from logically unsafe functions can be formulated as a sequence-to-sequence (seq2seq) translation problem. Specifically, let x be an input sequence representing a logically unsafe function with its related context, and let y be the corresponding output sequence representing the PoC code snippet that triggers the undefined

behavior. The goal is to learn a mapping $f : X \rightarrow Y$, where X is the domain of all logically unsafe functions and Y is the domain of all PoC code snippets.

Formally, given an input sequence $x \in X$, the objective is to generate an output sequence $\hat{y} \in Y$ that faithfully reproduces the PoC code associated with the logically unsafe operations identified in x . This can be formulated as $\hat{y} = f_\theta(x)$, where f_θ represents the LLM-based PoC generator parameterized by θ . The model is trained to minimize the discrepancy between the generated sequence \hat{y} and the ground truth sequence y . This is typically achieved through a token-level cross-entropy loss, which measures the prediction error for each token in the generated sequence. The training objective is as follows:

$$\mathcal{L}(\theta) = - \sum_{(x,y) \in D} \sum_{t=1}^T \log p_\theta(y_t | y_{<t}, x)$$

where D is the PoC dataset, T is the length of the output sequence, and $p_\theta(y_t | y_{<t}, x)$ is the probability of generating the t -th token y_t given the previous tokens $y_{<t}$ and the input sequence x .

In addition to token-level cross-entropy loss, we use BLEU (Bilingual Evaluation Understudy), METEOR (Metric for Evaluation of Translation with Explicit ORdering), and token-level accuracy to quantify the similarity between the generated sequence \hat{y} and the ground truth sequence y . These metrics help ensure that the generated PoC code can illustrate the root causes of the logically unsafe operations.

Our goal with this seq2seq approach is to assist developers in identifying and understanding the root causes of logically unsafe operations, rather than producing semantically correct PoCs that exploit the crate. Due to the inherent uncertainties of LLM outputs and Rust’s strict compiler checks, generating fully compilable Rust code without human intervention remains challenging. Nonetheless, we believe that LLM-generated PoCs can guide developers in reasoning about these unsafe operations and resolving potential issues.

5.6 Evaluation

To evaluate COIN, we design our experiment guided by the following questions:

1. How effective is COIN in real-world logically unsafe bug hunting?
2. What’s the performance of COIN’s fine-tuned model compared with other prompt-based LLMs?
3. How is COIN’s precision and recall compared with static analysis tools?
4. Can COIN’s PoC generator help developers understand the logically unsafe root causes?
5. What are the practical impacts of logically unsafe bugs found by COIN?

Environment Setup. We conducted our evaluation on an Ubuntu 24.04 LTS server equipped with AMD EPYC 7452 CPU, 512GB of memory, and NVIDIA RTX A6000 GPU. We fixed the Rust toolchain version to be 1.83.0-dev.

5.6.1 Q1: Logically Unsafe Bug Hunting

To demonstrate COIN’s end-to-end effectiveness, we evaluated COIN on open-source Rust crates to identify logically unsafe operations that were overlooked by developers. Due to GPU resource constraints, we randomly sampled 10K popular crates from crates.io, ran COIN on their repositories, and attempted to locate logically unsafe operations. The overall results are summarized in Table 5.2. Crucially, COIN successfully discovered 25 *previously unknown* logically unsafe bugs across various crates, with 58.1% precision. We have reported all of them to corresponding maintainers following the responsible disclosure procedure. So far, 13 of these bugs have been confirmed (10 have been fixed), and we have obtained 10 CVE IDs³. These identified vulnerabilities span over multiple categories of

³We did not request CVE IDs for issues related to private APIs within crates and some of the CVE requests are still pending.

Table 5.2: Evaluation result on finding missed logically unsafe bugs and comparison with other baseline approaches. For the unsafe type, ‘PC’ means ‘Precondition Check’, ‘MI’ stands for ‘Memory Invariants’, ‘DM’ means ‘Device Memory mapping’, ‘FFI’ means ‘calling FFI functions’ and ‘DR’ means ‘data races’. For the status of the bug, ‘P’ indicates the bug has been patched by developers, ‘C’ means the bug is confirmed by developers and ‘W’ represents the report is still waiting for review.

Crate	Status	Unsafe Type	CVE	Downloads	COIN	Rudra	MirChecker	FFI-checker	GPT-4o	Claude-3.7	PoC
sqlite3-parser	P	PC	2025-4*	1.6M	✓	✗	✗	✗	✓	✓	✓
obfstr	P	PC	2024-5*	1.3M	✓	✗	✗	✗	✓	✓	✗
ruru	W	PC		47.8K	✓	✗	✗	✗	✓	✓	✓
mpv	P	PC		17.2K	✓	✗	✗	✗	✓	✗	✓
spiral-rs	W	PC	2025-4*	4.7K	✓	✗	✗	✗	✓	✗	✓
memory_pages	W	PC	2025-4*	1.3K	✓	✓	✓	✗	✗	✗	✓
trailer	W	PC	2025-4*	36K	✓	✗	✓	✗	✓	✓	✓
rustybuzz	P	MI		3.8M	✓	✗	✗	✗	✗	✗	✓
opencl3	P	MI		0.2M	✓	✗	✗	✗	✗	✓	✓
scsir	W	DM	2025-4*	2.0K	✓	✗	✗	✗	✗	✗	✓
efm32gg11b	W	DM		2.6K	✓	✗	✗	✗	✗	✗	N/A
rsl10-pac	W	DM		1.9K	✓	✗	✗	✗	✗	✗	N/A
rula	C	FFI		1.9M	✓	✗	✗	✗	✗	✓	✓
spng	C	FFI		31.7K	✓	✓	✓	✓	✓	✓	✗
libucl	W	FFI		10.8K	✓	✗	✗	✓	✓	✗	✓
libblkid-rs	P	FFI		0.2M	✓	✗	✗	✗	✗	✗	✓
winit	P	FFI		19.2M	✓	✗	✗	✗	✗	✗	✓
stivale	W	FFI		4.3K	✓	✗	✗	✓	✓	✗	✓
xrdb	P	FFI		3.5K	✓	✗	✗	✗	✗	✗	✓
anode	P	DR	2025-4*	1.3K	✓	✗	✗	✗	✓	✓	✓
process-sync	W	DR	2025-4*	7.2K	✓	✗	✗	✗	✓	✓	✓
process_lock	W	DR	2025-4*	2.1K	✓	✗	✗	✗	✗	✗	✓
trionphe	P	DR		36M	✓	✗	✓	✗	✗	✗	✗
xarc	W	DR		3.8K	✓	✗	✗	✗	✓	✓	✓
wgp	C	DR	2025-4*	3.7K	✓	✗	✗	✗	✓	✓	✓
Total	-	-	10	-	25	2	4	3	13	12	20

logically unsafe Rust issues, showing COIN’s effectiveness in detecting a wide range of different logically unsafe bug patterns.

5.6.2 Q2: Comparing with Vanilla LLMs

To further evaluate COIN’s effectiveness, we compare it with the latest LLM models, including both open-source ones (QWen3-4B [85], Llama3.1-8B [86] and Llama-3.2-3B [87]) and closed-source ones (OpenAI’s GPT-4o [88] and Anthropic’s Claude-3.7 [89]). We note that COIN’s classifier is fine-tuned based on the Llama3.2-3B model.

Open Source Models

We split the logically unsafe dataset into training, validation, and test subsets with the ratio of 6:2:2, and evaluate each model’s performance on the testing dataset. We use the precision-recall curve of unsafe label to demonstrate unsafe classification results, which represent the model’s ability to classify logically unsafe labels. In particular, we compute the *precision* and *recall* of the unsafe classifier defined as follows:

$$\begin{aligned} \text{precision} &:= \frac{\sum_{(x,u) \in A_{\text{test}}} \mathbb{1}(u = 1 \text{ and } \hat{u}(x) = 1)}{\sum_{(x,u) \in A_{\text{test}}} \mathbb{1}(\hat{u}(x) = 1)} \\ \text{recall} &:= \frac{\sum_{(x,u) \in A_{\text{test}}} \mathbb{1}(u = 1 \text{ and } \hat{u}(x) = 1)}{\sum_{(x,u) \in A_{\text{test}}} \mathbb{1}(u = 1)} \end{aligned}$$

The precision-recall curve is shown in Figure 5.3. Overall, COIN achieves satisfactory performance with an AUPRC of 0.82. For other general models, since they are not trained or fine-tuned specifically for logically unsafe Rust classification, their performance is limited.

Different Models. To further demonstrate the generality of COIN’s solution, we also trained logically unsafe classification on QWen2.5-1.5B model. The precision/recall evaluation of QWen2.5 model is shown in Figure 5.2, achieving comparable classification accuracy compared with Llama3.2-3B model.

PAC Thresholding. In practice, a threshold is picked based on the AUPRC graph to power

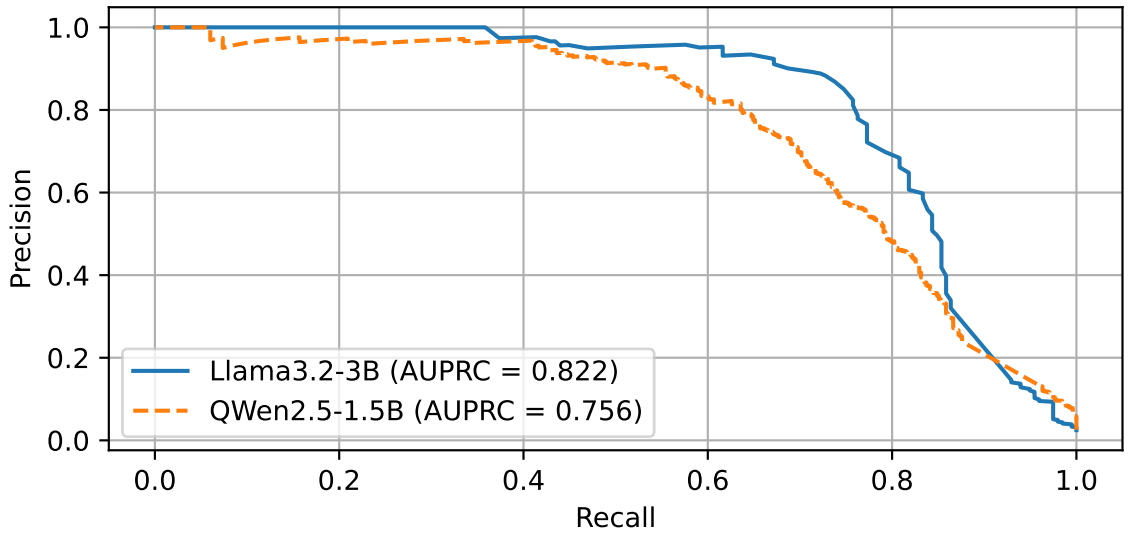


Figure 5.2: Precision-recall (AUPRC) evaluation of QWen2.5-1.5B model and Llama3.2-3B model. The COIN uses Llama3.2-3B model for evaluation.

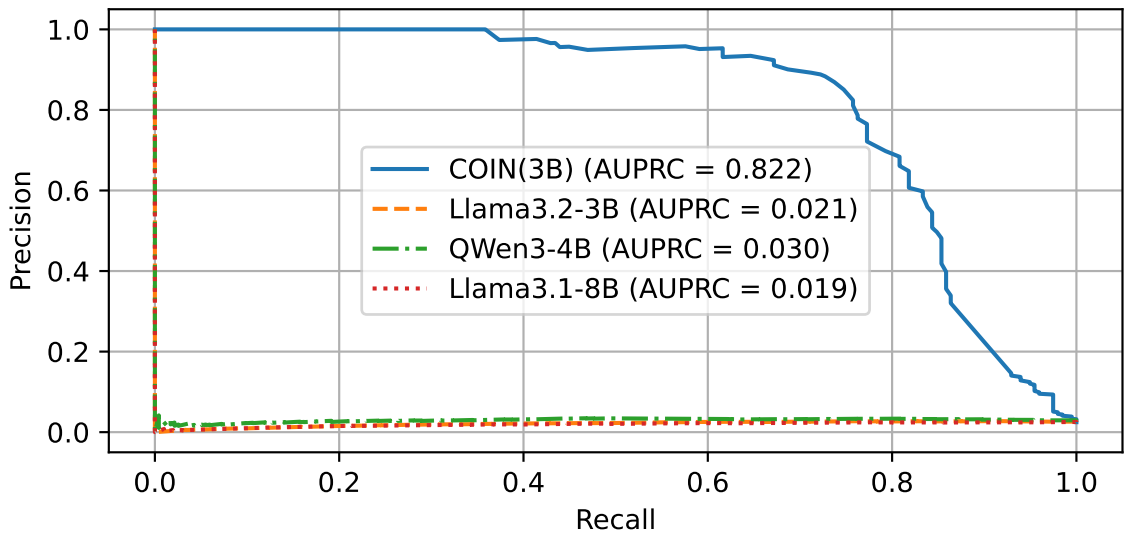


Figure 5.3: Precision-recall (AUPRC) evaluation of different models on the logically unsafe classification task. COIN achieves 63.71% precision with 80.42% recall, out-performing other general models.

Table 5.3: Few-shot (N) examples evaluation for $N = 1, 3, 5$ on GPT-4o and Claude-3.7 with Best@1 settings. The COIN’s fine-tuning significantly improve the model’s ability for identifying logically unsafe Rust.

Model (P/R)	$N=1$	$N=3$	$N=5$
GPT-4o	4.6%/21.4%	3.7%/21.4%	3.9%/21.4%
Claude-3.7	4.2%/7.1%	6.5%/11.9%	6.9%/11.9%
COIN	63.7%/80.4%		

the model in the inference step. We use the trusted thresholding algorithm proposed in (Equation 5.1) for $\hat{\tau}$, and a chosen threshold provides 80.42% recall of unsafe functions. This suggests that reliable thresholding provides the desired guarantee of recall, controlled by ε . Therefore, the COIN’s performance on the logically unsafe dataset with PAC thresholding is 63.71% precision and 80.42% recall.

Private Models

For closed-source models, we do not have access to their raw output logits; therefore, we cannot draw a precision–recall curve for comparison. Instead, we conducted both a few-shot evaluation and a best-of- K evaluation to measure each model’s precision and recall point for comparison with COIN.

Few-Shot Example. In the few-shot evaluation, we embedded N examples for each category into the LLM prompt, including the corresponding answers, to assess the LLM’s understanding of the target problem with context. The evaluation results for GPT-4o and Claude-3.7 are presented in Table 5.3. The result shows even with the relevant examples as context, it’s still challenging for state-of-the-art models(GPT-4o and Claude-3.7) to reason the logically unsafe operations and identify them. Through fine-tuning, COIN improves the precision and recall of detecting logically unsafe bugs.

Best of K . In the best-of- K evaluation, we queried the model K times and checked whether at least one of the K attempts correctly identified the answer. This evaluation demonstrates the potential of LLMs to solve our task, and the results are shown in Table 5.4. The results

Table 5.4: Best of K examples evaluation on GPT-4o and Claude-3.7 with $N=1$ shot example.

Model (P/R)	$K=1$	$K=3$	$K=5$
GPT-4o	4.6%/21.4%	5.4%/22.3%	6.2%/23.4%
Claude-3.7	4.2%/7.1%	4.2%/16.7%	6.8%/21.4%
COIN	63.7%/80.4%		

show that even with more chances, the general models are struggling to correctly resolve the problem, compared with COIN.

Bug Hunting. We tested both vanilla models on the logically unsafe bugs identified by COIN, and the results are shown in Table 5.2. Among the 25 bugs, GPT-4o identified 13, and Claude-3.7 identified 12. However, given their precision rates in earlier evaluations, these models are likely to introduce many false positives when applied to the entire dataset.

5.6.3 Q3: Comparing with static analysis tools

Besides LLM-based approaches, we further compare COIN with traditional static analysis tools, including Rudra [44], MirChecker [46], and ffi-checker [74]. In a nutshell, Rudra is a source code-level static analyzer focusing on unsafe Rust to detect unsound issues; MirChecker and ffi-checker work on the LLVM IR level to detect memory safety errors in Rust. We do not compare COIN with dynamic analysis tools like fuzzers, due to their inherent randomness and difficulties in identifying these unsafe operations and misuses. We run static analysis tools on logically unsafe bugs found by COIN, and conduct a precision/recall study of COIN comparing with them.

Bug Hunting. We first run static analysis tools on all the logically unsafe bugs found by COIN to check if the tools can detect such errors. As shown in Table 5.2, most of logically unsafe bugs cannot be detected by existing static analysis tools because of different bug patterns. We note that the reason is mainly because previous work mainly focuses on the explicit unsafe regions to analyze, which are enforced by `rustc`, but COIN focuses on the logically unsafe operations that `rustc` cannot infer.

Table 5.5: Precision/recall study of COIN. COIN achieves a precision of 57.9%, outperforming Rudra’s 53.3%, MirChecker’s 20.8% and ffi-checker’s 15.3%. ‘-’: recall not reported in the paper.

	COIN	Rudra	MirChecker	FFI-checker
Precision	57.9%	53.3%	20.8%	15.3%
Recall	76.6%	-	-	-

Precision/Recall. Besides the real-world bug hunting, we conducted a precision–recall study of COIN using a benchmark derived from Rust standard library and compared the results against other static analysis tools on their own dataset for a general comparison. The Rust standard library is maintained by the Rust core developers under the Rust Foundation and is frequently updated with new features and bug fixes. Due to the difficulty of manually identifying all logically unsafe operations, we used functions from the Rust standard library as ground truth to evaluate COIN’s precision and recall. We removed these functions from our logically unsafe dataset before fine-tuning. The evaluation results are shown in Table 5.5: COIN achieves 57.9% precision and 76.6% recall on standard library dataset. Compared to other static analysis tools, COIN demonstrates better precision and recall, making it well suited for Rust program analysis.

5.6.4 Q4: PoC Generator Evaluation

To evaluate the performance of COIN’s PoC generator, we applied the newly found logically unsafe operations as inputs for the PoC generator and check if COIN can generate a reasonable PoC facilitating the bug analysis, confirmation, and reproduction. We manually reviewed all the PoC generated by COIN and included succeeded PoCs in our bug reports to developers.

The PoC generation results are shown in the PoC column in Table 5.2. COIN successfully generates the PoC indicating the root causes for 20 out of 25 cases (✓), showing its practicality. COIN failed to generate PoC for limited complex cases like `obfstr` (✗), whose PoC requires constructing a customized struct to bypass the macros in Rust. Note that we

```

1 // Context omitted for simplicity
2 let mut v = b"SELECT_?_".to_vec();
3 v.extend_from_slice(&[0xC0, 0x80]);
4 let mut parser = Parser::new(&v);
5 ...

```

Listing 5.11: The PoC code generated by COIN for invalid utf-8 strings in `sqlite3-parser` crate, a crash will be triggered whenever the invalid string is accessed.

exclude the bug type of direct memory mapping (DM) from PoC generation (N/A) since the device driver requires a special environment to build.

5.6.5 Q5: Case Study

We use two cases to demonstrate the potential security impact of logically unsafe bugs in Rust community.

Missing Precondition Check in `sqlite3-parser`

The ‘`sqlite3-parser`’ is a popular crate in Rust to convert the sql string into a structured AST for further execution. It is widely used in sqlite development in Rust, with more than 1.5M downloads and even in commercial forks of sqlite with more than 14.6K stars on GitHub. While auditing the implementations, COIN finds a missing condition check for parsing raw bytes into a utf-8 encoded string without validating the encoding, and the raw bytes can be manipulated by users with arbitrary inputs. Besides, COIN’s PoC generator successfully generates the PoC code to demonstrate this issue, shown in Listing 5.11. The PoC introduced the invalid utf-8 encoding bytes as inputs, and `sqlite3-parser` converts it into the structured AST. This invalid string can crash the program whenever it is used. In Rust, producing an invalid value (invalid utf-8 string in this case) is considered undefined behavior. Because of its wide range of impact, a CVE ID is assigned for this issue.

```

1 // helper function
2 fn drop_slow(this: *mut _, old_ref: usize) {
3     match old_ref {
4         2 => (*this).waker.wake(),
5         1 => drop(Box::from_raw(this)),
6         _ => {}
7     }
8 }
9 // drop implementation, which is 'safe'
10 let old_ref= self.deref().refcount
11     .fetch_sub(1, Ordering::Release);
12 if old_ref > 2 {
13     return;
14 }
15 self.deref().refcount.load(Ordering::Acquire);
16 unsafe { drop_slow(self.0.as_ptr(), old_ref) }

```

Listing 5.12: A potential data race can be introduced between the last two threads trying to drop the reference and lead use-after-free error in wgp, the code details are omitted for simplicity

Use-after-free Data Races in wgp

The ‘wgp’ crate is a wait group implementation in Rust that provides concurrency capabilities for shared data. In its `drop` implementation shown in Listing 5.12, the crate aims to wake up the waiting thread on its second-to-last reference and free the resources in its last reference. However, the crate only ensures synchronization to restrict triggering `drop_slow` to the last two threads, but it neglects to add synchronization between these last two threads. Consequently, it is possible for the last thread to free the resources in line 5 first, while the second-to-last thread initiates the call to `waker` in line 4, as demonstrated in Listing 5.12. This potential error is identified by COIN while auditing the function from the wgp crate, and the PoC generator can create code that spawns two threads of the data and attempts to drop them. The thread sanitizer reports the error when executing the PoC code after minor modifications to resolve path and crate naming issues. Due to its complex data race conditions, a CVE number has been assigned for this issue.

5.7 Limitation

In this section, we discuss the potential limitations of COIN. The first limitation of COIN is the limited study of logically unsafe operations due to their large volume. In cases that are misclassified or fail to be classified by the LLM, they can introduce true-negative cases into COIN’s analysis. Additionally, another limitation arises from the PoC code, which serves as a verification sample for the unsound issue. However, failing to produce a PoC code does not necessarily indicate that the function is safe; there may be cases where constructing the PoC code is excessively challenging.

COIN’s implementation limitations are mainly from the following steps: 1. Because of the limited GPU resources, we only trained COIN on 3B models with LoRA fine-tuning. We argue that a larger model may improve the performance of COIN for both tasks. 2. We reduced the token limitation of COIN to 8192 tokens; a larger context window may help COIN better understand the program logic.

CHAPTER 6

CONCLUSION

We demonstrate that the integration of LLMs with program analysis techniques can substantially improve the quality and safety of Rust code across the software development lifecycle. The proposed framework addresses three critical aspects: generating comprehensive unit tests, detecting unsafe operations in binaries, and identifying logically unsafe code that escapes traditional analysis. RUG’s bottom-up context construction and fuzzing integration enable the generation of compilable unit tests with coverage rates on par with human-written tests, surpassing existing LLM-based and traditional approaches. RUBY’s machine learning-based binary analysis achieves high precision and recall in identifying unsafe operations, filling a crucial gap for security analysis in environments where source code is inaccessible. COIN’s LLM-based detection of logically unsafe operations uncovers previously unknown vulnerabilities, with several findings confirmed and assigned CVEs in real-world projects.

While the approaches presented here mark significant progress, ongoing challenges remain, such as adapting to rapid language evolution and improving the robustness of LLM-based tools. Future work should explore cross-language applications, more sophisticated program analysis techniques, and deeper integration with modern development workflows. As Rust’s adoption continues to grow, the importance of automated, reliable code quality assurance will only increase, and the methods developed in this work offer a path forward for both research and practice.

REFERENCES

- [1] T. R. Team. “Rust: A language empowering everyone to build reliable and efficient software”. <https://www.rust-lang.org/>. (2010).
- [2] Microsoft security engineers, *Microsoft: 70 percent of all security bugs are memory safety issues*, <https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues>, 2019.
- [3] Google engineers, *Chrome: 70% of all security bugs are memory safety issues*, <https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues>, 2020.
- [4] Google Security Blog. “Supporting Use of Rust in Chromium”. ().
- [5] Rust for Linux Contributors. “Rust-for-Linux/linux”, GitHub. (), (visited on 08/01/2023).
- [6] The Register. “Microsoft to explore Windows 11 code written in Rust”. ().
- [7] Rust-GPU Contributors. “Rust-GPU/Rust-CUDA”, GitHub. ().
- [8] T. R. Team. “Unsafe operations in rust”. ().
- [9] T. R. S. C. W. Group, *The rust security advisory database*, <https://rustsec.org/>.
- [10] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, “How do programmers use unsafe rust?”, *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.
- [11] A. N. Evans, B. Campbell, and M. L. Soffa, “Is rust used safely by software developers?”, in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, IEEE, 2020, pp. 246–257.
- [12] H. F. Eniser *et al.*, “Towards translating real-world code with llms: A study of translating to rust”, *arXiv preprint arXiv:2405.11514*, 2024.
- [13] Wikipedia contributors, *Undefined behavior*, Wikipedia, The Free Encyclopedia, Accessed: 2025.
- [14] T. R. Team, *Behavior considered undefined*, <https://doc.rust-lang.org/reference/behavior-considered-undefined.html>.
- [15] T. R. P. Language and Contributors, *Meet safe and unsafe*, <https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html>.

- [16] Wikipedia contributors, *Memory safety*, https://en.wikipedia.org/wiki/Memory_safety, Accessed: 2025.
- [17] N. D. Matsakis and F. S. Klock, “The rust language”, *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.
- [18] E. C. Reed, “Patina : A formalization of the rust programming language”, University of Washington, Tech. Rep., 2015.
- [19] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “Rustbelt: Securing the foundations of the rust programming language”, *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017.
- [20] W. Landi, “Undecidability of static analysis”, *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.
- [21] Crates.io, *Fake-static*, <https://crates.io/crates/fake-static>, [Online; accessed 2025-05-03], n.d.
- [22] F. Taufiqurrahman, S. Widowati, and M. J. Alibasa, “The impacts of test driven development on code coverage”, in *2022 1st International Conference on Software Engineering and Information Technology (ICoSEIT)*, IEEE, 2022, pp. 46–50.
- [23] M. Siniaalto and P. Abrahamsson, “A comparative case study on the impact of test-driven development on program design and test coverage”, in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, IEEE, 2007, pp. 275–284.
- [24] M. Davis, S. Choi, S. Estep, B. Myers, and J. Sunshine, “Nanofuzz: A usable tool for automatic test generation”, in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1114–1126.
- [25] V. Tymofyeyev and G. Fraser, “Search-based test suite generation for rust”, in *International Symposium on Search Based Software Engineering*, Springer, 2022, pp. 3–18.
- [26] *American fuzzy lop*.
- [27] Y. Takashima, R. Martins, L. Jia, and C. S. Păsăreanu, “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.

- [28] X. Wu, N. Cheriére, C. Zhang, and D. Narayanan, “Rustgen: An augmentation approach for generating compilable rust code with large language models”, 2023.
- [29] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, “Chatunitest: A chatgpt-based automated unit test generation tool”, *arXiv preprint arXiv:2305.04764*, 2023.
- [30] G. Fraser and A. Arcuri, “Evosuite: Automatic test suite generation for object-oriented software”, in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.
- [31] S. Lukaszcyk and G. Fraser, “Pynguin: Automated unit test generation for python”, in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.
- [32] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models”, in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 919–931.
- [33] N. Alshahwan *et al.*, “Automated unit test improvement using large language models at meta”, *arXiv preprint arXiv:2402.09171*, 2024.
- [34] Sourcegraph, *Cody*, <https://sourcegraph.com/cody>, [Online; accessed 2024-03-22], 2023.
- [35] GitHub, *Github copilot: Your ai pair programmer*, <https://github.com/features/copilot>, [Online; accessed 2024-03-22], 2023.
- [36] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, “Learning and programming challenges of rust: A mixed-methods study”, in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1269–1281.
- [37] bincod-org, *Bincod: A binary encoder/decoder in rust*, <https://github.com/bincod-org/bincod>, 2023.
- [38] P. Deligiannis, A. Lal, N. Mehrotra, and A. Rastogi, “Fixing rust compilation errors using llms”, *arXiv preprint arXiv:2308.05177*, 2023.
- [39] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation”, *IEEE Transactions on Software Engineering*, 2023.
- [40] C. LLVM. “Source-based code coverage - clang 11 documentation”. ().

- [41] J. Jiang, H. Xu, and Y. 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.
- [42] Z. Xu, B. Wu, C. Wen, B. Zhang, S. Qin, and M. He, “Rpg: Rust library fuzzing with pool-based fuzz target generation and generic support”, in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
- [43] *Openai platform models*.
- [44] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, “Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale”, in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.
- [45] J. Min, D. Yu, S. Jeong, D. Song, and Y. Jeon, “Erasan: Efficient rust address sanitizer”, in *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, 2024, pp. 239–239.
- [46] Z. Li, J. Wang, M. Sun, and J. C. Lui, “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.
- [47] X. Cheng, F. Sang, Y. Zhai, X. Zhang, and T. Kim, “RUG: Turbo LLM for Rust Unit Test Generation”, in *Proceedings of the 47th International Conference on Software Engineering (ICSE)*, Ottawa, Canada, Apr. 2025.
- [48] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “{Addresssanitizer}: A fast address sanity checker”, in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.
- [49] F. Developers, *Ferrocene: Safety-critical systems in rust*, <https://github.com/ferrocene/ferrocene>, Accessed: 2025-03-31, 2024.
- [50] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, “Undefined behavior: What happened to my code?”, in *Proceedings of the Asia-Pacific Workshop on Systems*, 2012, pp. 1–7.
- [51] CasualX, *Issue #60: Unsound issue while converting bytes to utf8 str*, <https://github.com/CasualX/obfstr/issues/60>, Accessed: 2023-10-10, 2023.
- [52] Y. Shoshitaishvili *et al.*, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”, in *IEEE Symposium on Security and Privacy*, 2016.

- [53] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing”, in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.
- [54] J. C. King, “Symbolic execution and program testing”, *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [55] Atticus, *Niche optimizations in rust*, https://www.0xatticus.com/posts/understanding_rust_niche/, Accessed: 2025-04-21, Jul. 2024.
- [56] H. Xue, S. Sun, G. Venkataramani, and T. Lan, “Machine learning-based analysis of program binaries: A comprehensive study”, *IEEE Access*, vol. 7, pp. 65 889–65 912, 2019.
- [57] B. Beckman and J. Haile, “Binary analysis with architecture and code section detection using supervised machine learning”, in *2020 IEEE Security and Privacy Workshops (SPW)*, IEEE, 2020, pp. 152–156.
- [58] J. Wang, M. Sharp, C. Wu, Q. Zeng, and L. Luo, “Can a deep learning model for one architecture be used for others? {retargeted-architecture} binary code analysis”, in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7339–7356.
- [59] Y. Liu *et al.*, “Roberta: A robustly optimized bert pretraining approach”, *arXiv preprint arXiv:1907.11692*, 2019.
- [60] A. Vaswani *et al.*, “Attention is all you need”, in *Advances in neural information processing systems*, 2017, pp. 5998–6008.
- [61] J. Platt *et al.*, “Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods”, *Advances in large margin classifiers*, 1999.
- [62] V. Vovk, “Conditional validity of inductive conformal predictors”, *Machine learning*, vol. 92, no. 2-3, pp. 349–376, 2013.
- [63] S. S. Wilks, “Determination of sample sizes for setting tolerance limits”, *The Annals of Mathematical Statistics*, vol. 12, no. 1, pp. 91–96, 1941.
- [64] S. Park, O. Bastani, N. Matni, and I. Lee, “Pac confidence sets for deep neural networks via calibrated prediction”, in *International Conference on Learning Representations*, 2020.
- [65] S. Park, E. Dobriban, I. Lee, and O. Bastani, “PAC prediction sets under covariate shift”, in *International Conference on Learning Representations*, 2022.

- [66] C. J. Clopper and E. S. Pearson, “The use of confidence or fiducial limits illustrated in the case of the binomial”, *Biometrika*, vol. 26, no. 4, pp. 404–413, 1934.
- [67] The Servo Project Developers, *The servo parallel browser engine project*, <https://github.com/servo/servo>.
- [68] Artichoke Ruby, *Build the next ruby for wasm with artichoke*, <https://github.com/artichoke/artichoke>.
- [69] Deno Land Inc, *Deno: A modern runtime for javascript and typescript*, <https://github.com/denoland/deno>.
- [70] Rust Blog. “Announcing rust 1.67.0”. Accessed: 2025-04-02. (Jan. 2023).
- [71] Rust Blog. “Announcing rust 1.57.0”. Accessed: 2025-04-02. (Dec. 2021).
- [72] Rust Blog. “Announcing rust 1.75.0”. Accessed: 2025-04-02. (Dec. 2023).
- [73] H. Han, J. Kyea, Y. Jin, J. Kang, B. Pak, and I. Yun, “Queryx: Symbolic query on decompiled code for finding bugs in cots binaries”, in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023, pp. 3279–312 795.
- [74] Z. Li, J. Wang, M. Sun, and J. C. Lui, “Detecting cross-language memory management issues in rust”, in *European Symposium on Research in Computer Security*, Springer, 2022, pp. 680–700.
- [75] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, “Automatic exploit generation”, *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.
- [76] Y. Zhang, “Detecting code comment inconsistencies using llm and program analysis”, in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 683–685.
- [77] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, “Using an llm to help with code understanding”, in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
- [78] M. Cui, S. Sun, H. Xu, and Y. Zhou, “Is unsafe an achilles’ heel? a comprehensive study of safety requirements in unsafe rust programming”, in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
- [79] M. Cui, C. Chen, H. Xu, and Y. Zhou, “Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis”, *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–21, 2023.

- [80] A. Sharma, S. Sharma, S. R. Tanksalkar, S. Torres-Arias, and A. Machiry, “Rust for embedded systems: Current state and open problems”, in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2296–2310.
- [81] svd2rust contributors, *Issue #714: Compatibility with new version of svdtools*, <https://github.com/rust-embedded/svd2rust/issues/714>, Accessed: 2025-05-05, 2024.
- [82] The Rust Project Developers, *BTreeMap in The Rust Standard Library (std)*, <https://doc.rust-lang.org/std/collections/struct.BTreeMap.html>, Accessed: 2025-05-05, 2024.
- [83] Z. Feng *et al.*, “Codebert: A pre-trained model for programming and natural languages”, *arXiv preprint arXiv:2002.08155*, 2020.
- [84] S. Park, X. Cheng, and T. Kim, “Unsafe’s betrayal: Abusing unsafe rust in binary reverse engineering via machine learning”, *arXiv preprint arXiv:2211.00111*, 2022.
- [85] QwenLM, *Qwen3: Think Deeper, Act Faster*, Blog post, <https://qwenlm.github.io/blog/qwen3/>, Accessed June 2025, 2025.
- [86] Meta AI, *Model Cards and Prompt Formats for Llama 3.1*, Documentation page, https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_1/, Accessed June 2025, 2024.
- [87] Meta AI, *Model Cards and Prompt Formats for Llama 3.2*, Documentation page, https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_2/, Accessed June 2025, 2024.
- [88] OpenAI, *GPT-4o system card*, arXiv preprint arXiv:2410.21276, Available at <https://arxiv.org/abs/2410.21276>, 2024.
- [89] Anthropic, *Claude Sonnet 3.7 system card*, Available at <https://www.anthropic.com/claude-3-7-sonnet-system-card>, Accessed June 2025, Feb. 2025.