The Illusion of Rust Safety: Detecting Modular Unsafe Functions with LLMs

Xiang Cheng
Georgia Institute of Technology
Atlanta, GA, USA
cxworks@gatech.edu

Fan Sang
Georgia Institute of Technology
Atlanta, GA, USA
fsang@gatech.edu

Yibin Yang*
University of Toronto
Toronto, Canada
yibiny@ece.utoronto.ca

Hang Zhang
Indiana University
Bloomington, IN, USA
hz64@iu.edu

Sangdon Park
Pohang University of Science and Technology
Pohang, South Korea
sangdon@postech.ac.kr

Xiaokuan Zhang
George Mason University
Fairfax, VA, USA
xiaokuan@gmu.edu

Taesoo Kim†
Georgia Institute of Technology
Atlanta, GA, USA
taesoo@gatech.edu

Abstract

Memory safety vulnerabilities account for approximately 70% of reported security bugs, prompting the adoption of Rust, a systems programming language designed to prevent such issues through compile-time checks. While Rust categorizes code into safe and unsafe regions, we identify a previously understudied class of vulnerabilities: modular unsafe functions (`mufs`) in Rust code. These functions appear safe to the compiler but are, in fact, unsafe within the whole module's APIs and can cause undefined behavior without using the `unsafe` keyword, leading to a misleading sense of safety and unsound issues in Rust.

We present the first systematic study of `mufs`, defining five root cause categories from empirical analysis. We develop Coin, a detection system using fine-tuned large language models for vulnerability identification and proof-of-concept generation. Coin achieves 63.7% precision and 80.4% recall on labeled evaluation set, outperforming existing tools. On evaluation of 10,000 randomly sampled Rust crates, Coin identified 26 unknown `mufs` with 14 developer confirmations and 10 CVE assignments.

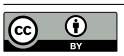
CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Rust, LLM, Undefined Behavior

*Also with NTT Research.
†Also with Microsoft.



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832709

ACM Reference Format:
Xiang Cheng, Fan Sang, Yibin Yang, Hang Zhang, Sangdon Park, Xiaokuan Zhang, Taesoo Kim. 2026. The Illusion of Rust Safety: Detecting Modular Unsafe Functions with LLMs. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832709

1 Introduction

In recent decades, memory security has drawn significant attention from the security community. Approximately 70% of the numerous reported bugs are identified annually as memory safety issues, as highlighted by Microsoft [32] and Google [17]. These vulnerabilities allow attackers to manipulate memory, resulting in critical security risks, such as privilege escalation in the Linux kernel [19]. Despite thorough investigations into different bug categories (*e.g.,* use-after-free, double-free, and buffer overflow), the emergence of new memory safety bugs persists each year [22]. Rust is a system programming language aimed at memory safety [51], and is extensively utilized in OS kernels [46, 54], browser engines [18], and device drivers [47]. In six years of usage in the Android, Rust has reduced memory safety bugs to under 24%, significantly lower than the industry average of 70% [50, 59].

Rust ensures memory safety by categorizing code into safe and unsafe regions. `Safe` Rust, a strictly-typed language, guarantees memory safety through compile-time checks based on Rust-specific features such as *ownership* and *borrowing*. Although safe Rust provides strong guarantees for memory safety, it imposes certain limitations that may not be suitable when developers need to perform low-level operations (*e.g.,* handling raw pointers). To overcome these limitations, `unsafe` Rust is used to temporarily relax the restrictions, shifting the responsibility for memory safety from the compiler to the developers. While unsafe Rust can offer certain advantages—as its name suggests—it also introduces security risks. Developer errors may lead to memory safety issues [3, 14]. The compiler (`rustc`) mandates the `unsafe` keyword for such operations,

and it generates compilation errors when the `unsafe` keyword is missing. This enforcement mechanism creates a false sense of security, as developers often assume code without the `unsafe` keyword is inherently safe.

Modular-level Safety Markers. We have observed Rust code patterns in popular Rust crates where the `unsafe` keyword is not required for compilation but is intentionally used by the developers as a safety marker (see an example in Listing 1 in Section 6.5). Removing `unsafe` keywords still allows code to pass `rustc`'s compilation checks, but can introduce undefined behavior [53] to the whole module[1] through carefully crafted API sequences, this can violate the overall module-level safety and soundness.

We define these as *modular unsafe functions (`mufs`)*: functions written in Safe Rust that i) are accepted by the compiler without the `unsafe` keyword, yet ii) are fundamentally unsafe because they enable undefined behavior when composed with other APIs. The **root cause** of `mufs` lies in the dependency of unsafe blocks on state established by safe code. This separation creates a non-local safety coupling, where an ostensibly safe function can invalidate the invariants required by a separate unsafe region. These `mufs` functions undermine compiler-enforced safety and can cause memory safety violations or system failures within safe Rust code. While prior work has focused on memory safety in `unsafe` Rust [6, 8, 24, 25, 27, 29, 58], we are the first to study modular unsafe functions in safe Rust and their security implications.

Research Questions. This work aims to bridge this gap by answering the following research questions:

- RQ1: What types of modular unsafe functions (`mufs`) exist, and how to categorize them?
- RQ2: How to systematically detect `mufs`?
- RQ3: What are the real-world security impacts of `mufs`?

Challenges. Accurately identifying `mufs` is difficult for existing automated program analysis techniques, due to the following challenges in distinguishing modular unsafe functions from normal safe functions:

- *Accommodating diverse patterns:* Modular unsafe functions manifest in various forms with distinct root causes that exceed the scope of pattern-based detection tools [6, 8, 24, 25]. The wide range of possible patterns makes it impractical to enumerate them exhaustively and develop comprehensive static analysis tools.
- *Understanding interprocedural semantics:* Detecting these functions requires understanding the logic and intent of the modular-level code. For example, the detection of Listing 1 requires grasping the meaning of the variable's name (*e.g.,* , `len` often signify length), the program logic (length denotes the modular usable memory in `vec`), and other modular APIs (`Vec.get` relies on `length` to perform boundary checks), which is challenging for traditional static analysis.
- *Constructing concrete exploits:* Assessing the security impact of these vulnerabilities requires constructing proof-of-concept (PoC) exploits [5]. A key challenge of module-level Rust unsoundness is that we only observe the exposed APIs, without any concrete misuse instances. As a result, we must synthesize PoCs without any prior knowledge by composing and exercising these API sequences.

Our Solution. As Large Language Models (LLMs) are good at capturing the semantics of source code [34, 64] with a long context window, we propose leveraging LLMs to detect modular unsafe functions in safe Rust code. Our key insight is that LLMs' semantic understanding capabilities can overcome the limitations of traditional static analysis tools: 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 modular unsafe pattern through conventional methods. Second, LLMs possess a degree of understanding of interprocedural code semantics, such as interpreting variable names and usages to reason about their logical functionalities and global effects. Third, LLMs excel at generation tasks, which can assist in generating PoCs of modular unsafe operations, guiding developers to understand the root causes.

To answer the three research questions and overcome the challenges, first, we begin by employing a customized `rustc` to find existing candidate instances: for each `unsafe` function, we try to remove the `unsafe` keyword and check if the program can be compiled; if so, we keep the function without the `unsafe` keyword in our dataset. we start with a small seed dataset with manually labeled `muf` categories, then gradually construct the *first* `muf` dataset (containing over 74k `mufs`) with new categories using LLMs and human-in-the-loop. Second, we have tried simply prompting LLM with few-shot `muf` examples, but it cannot produce reliable answers. To tackle this, we use Low-Rank Adaptation (LoRA) to fine-tune foundation models (`Llama3.2`) utilizing the `muf` dataset to build the classifier for identifying `muf` from Rust crates. Third, to help developers comprehend and verify the detected issues, we construct a fine-tuned PoC generator (based on a manually built PoC dataset for `mufs`) to generate PoC code as guidance.

We demonstrate Coin's effectiveness through comprehensive evaluation against state-of-the-art LLM models (QWen3, Llama3.2 and GPT-4o, Claude-3.7), static analysis tools (Rudra [6], MirChecker [24] and ffi-checker [25]) and dynamic analysis tools (RPG [63], RUG [10]). Coin out-performs these tools and achieves a reasonable precision (63.7%) and recall (80.4%) for `muf` detection. We also performed an in-the-wild experiment and evaluated Coin on 10K sampled crates from crates.io. Coin successfully identified 26 modular unsafe functions without the `unsafe` keyword. With the help of the PoC generator, we successfully generated bug reports with PoC code samples for 20 cases and submitted them to the developers. Among the 26 bugs submitted, 14 were confirmed by developers (5 of them have already been patched), with 10 CVEs assigned. 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 downstream applications.

In summary, this paper makes the following contributions:

- We identified and defined modular unsafe functions (`mufs`), and we classified the root causes of such issues into five categories based on an empirical study assisted by LLM.

[1] A module can be a `struct` or `mod` in a Rust crate, which contains multiple public APIs.

- We designed and implemented Coin, a LLM-based detector for `mufs` and a LLM-based PoC generator for verifying detected issues.
- We compared Coin with state-of-the-art static, dynamic and LLM-based tools. We also evaluated Coin on 10K sampled Rust crates and successfully identified 26 `mufs` that were previously unknown (with 10 CVEs assigned).

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 [61]. In Rust, common memory safety issues—including buffer overflows, use after free, and data races—are classified as undefined behavior [53]. 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 [52]. 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 [53].

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 [23], it includes all operations that could initiate undefined behaviors, encompassing all potential root cause sites within undefined behaviors.

2.2 Rust Programming Language

Safe Rust. Rust language [28] 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 [21, 45] under some assumptions on language semantics. In a nutshell, safe Rust tries to adhere to the following goal:

Definition 2 (Safe Rust Goal). *Safe Rust ideally should be sound—never cause undefined behavior, even with misuses.*

Safe Rust tries to ensure the memory safety of a program with a 'soundness' guarantee. Here, we describe the key concepts to achieve memory safety: *ownership* and *borrowing*. 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 compile-time memory safety without having a run-time garbage collector. 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 parties have write access to the variable when it is borrowed as a mutable reference.

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. 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 safety inside the unsafe regions. In particular, `rustc` defines five operations as unsafe operations [52], which help programmers to identify unsafe regions and ensure memory safety.

Definition 3 (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 Soundness and Unsound Issues

The soundness [57] of safe Rust can be defined as:

Definition 4 (Safe Rust Soundness). *Soundness is a statement about whether all possible uses of a library or language feature uphold the intended invariants.*

In other words, soundness describes the functionality of safe Rust APIs that cannot be misused, neither by mistake nor maliciously. With this strong argument and its unique high-order invariants, Rust transforms the misuse detections from users to the library/API owners. However, due to the complexity of program logics, it's challenging to fully achieve this target for all libraries in the Rust community. An unsound issue is introduced to denote such violations: users trigger undefined behaviors using pure safe library functions.

3 Motivating Example

Rust distinguishes itself from other systems languages through its strict memory safety guarantees. However, the interaction between *Safe Rust* and *Unsafe Rust* is subtle. While the `unsafe` keyword demarcates regions where the compiler relaxes checks, the preservation of memory safety is not strictly local to these blocks. In this section, we utilize the `Vec` example to demonstrate that safety in Rust is at *modular* level rather than function-local.

3.1 Two Sound Implementations of Vec

Consider a simplified implementation of a vector, a fundamental collection in Rust. The correctness of a vector relies on specific

invariants regarding its raw pointer, length, and capacity. Listing 1 presents a naive definition.

```
1  ... // Definition details are omitted
2  impl<T> Vec<T> {
3      /// Option 1: Marking function as unsafe
4      pub unsafe fn set_len(&mut self, new_len: usize) {
5          // No unsafe operations occur here
6          self.len = new_len;
7      }
8      /// Option 2: Making function private from users
9      fn set_len(&mut self, new_len: usize) {
10         // No unsafe operations occur here
11         self.len = new_len;
12     }
13     ...
14     pub fn push(&mut self, elem: T) {
15         if self.len == self.cap { self.reallocate(); }
16         unsafe {
17             // Relies on invariant: len < cap
18             ptr::write(self.ptr.add(self.len), elem);
19             self.set_len(self.len + 1);
20         }
21     }
22 }
```

Listing 1: A simplified, sound implementation of `Vec` with two different options: the `set_len` function has to be public but unsafe or private. Full example is at Appendix D

The `set_len` function in Listing 1 presents a paradox. Syntactically, the function body contains only safe operations: it assigns an integer to a field, doesn't belong to any of the unsafe operations in Definition 3. However, if `set_len` is exposed as a `public` interface, it must be marked `unsafe`. Omitting this marker allows external callers to leverage the function in conjunction with other APIs to construct a safe Rust sequence that triggers undefined behavior, as illustrated in Listing 6.

We characterize this distinct class of scenarios as *modular unsafety*: the `unsafe` marker is required not due to locally unsafe operations within the function body but because the function permits state transitions that violate module-level invariants (`vec`'s usable memory bounds), thereby compromising the soundness of the broader abstraction. In Listing 1, the unsafe regions are from line 16 to line 20, but the soundness of this unsafe region depends on the precondition checks in line 15 using safe Rust, which is further affected by the `set_len`.

In practice, developers address this dichotomy through either *encapsulation* or *delegation*. The first approach restricts `set_len` to private visibility, preventing external misuse. The second exposes the function as `public unsafe`, explicitly transferring the burden of invariant preservation to the caller. We note that both strategies constitute sound implementations, and there are no rigid guidelines dictating the choice.

3.2 Challenges

Modular unsafety arises because the soundness of an `unsafe` block frequently depends on invariants established by prior "safe" operations [56]. In the `push` example, the unsafe block's validity is predicated on a capacity check performed in safe code (line 15).

Consequently, soundness becomes a modular rather than a local property; an unsafe operation can be compromised by a safe mutation in a disparate function. Detecting such cases mandates global, interprocedural analysis to track state across boundaries. Furthermore, distinguishing relevant invariant-breaking operations from the noise of benign safe code requires precise semantic understanding. Finally, detection is complicated by the fact that these vulnerabilities are often latent—representing a potential for misuse by external clients rather than an active crash in the existing codebase.

3.3 Workflow

Given the inherent complexity of inferring high-level semantic logic and the necessity of synthesizing non-existing API sequences to trigger undefined behaviors, we contend that traditional static analysis is insufficient for addressing modular unsafety. In this work, we propose Coin, an LLM-based detector and a PoC generator for `muf`. The workflow of Coin is depicted in Figure 1: we first collect existing modular unsafe functions across the Rust community and build the dataset in Section 4.1. Then we conduct an empirical study with the help of LLM to classify the root causes Section 4.2. Based on the empirical study results and dataset, in Section 5.1, we fine-tuned Llama3.2 [15] to classify Rust functions (with context) as modular unsafe. Finally, 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 in Section 5.2.

4 Empirical Study of `mufs`

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

4.1 Dataset Collection

Since the modular unsafe operations are not detectable by `rustc`, we begin by employing a customized `rustc` to find all existing candidate instances: for each 'unsafe' region or function, we try to remove the 'unsafe' keyword and check if the program can be compiled. Our assumption is that these seemingly unnecessary 'unsafe' are intentionally added by developers because of the potential modular-level unsound issues. Through this approach, we successfully collected around 570K functions across the crates.io, taking approximately 0.38% of all functions in the Rust ecosystem. Then we filter out redundant instances, generated code from templates, and unimplemented functions. The final dataset, named as `modular unsafe dataset`, D_{muf}, remains 74.3K cases.

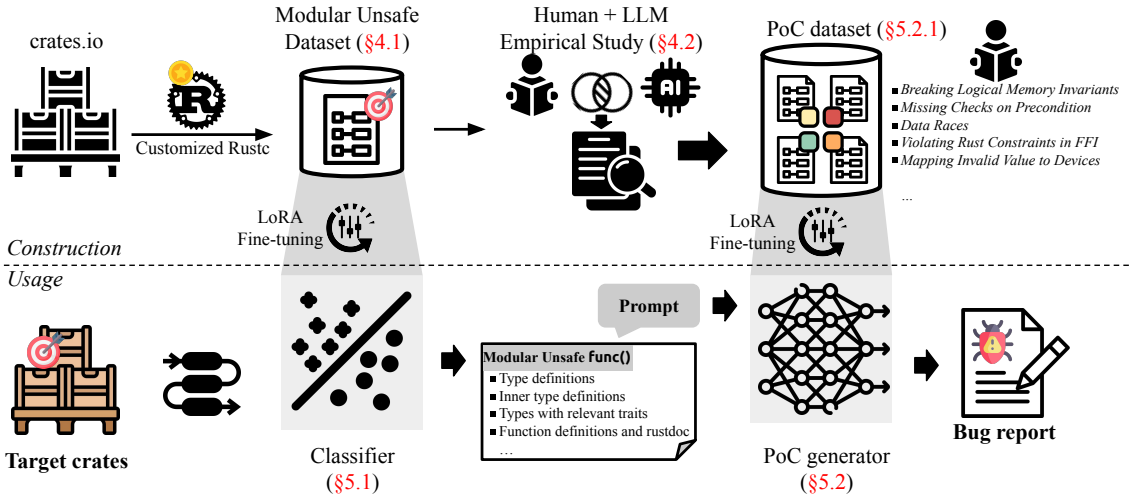


Figure 1: Coin's system overview. Coin includes a modular unsafe classifier trained on the existing modular unsafe examples collected from Rust community and a PoC generator trained on a manually created PoC dataset.

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

Table 1: The portion of each modular unsafe in Coin's study. 'PC' means 'Precondition Check', 'MI' stands for 'Memory Invariants', 'DM' means 'Device Memory mapping', 'FFI' means 'calling FFI functions' and 'DR' means 'data races'.

```rust
unsafe fn from_utf8_unchecked(bytes: Vec<u8>)
    -> String {
    String { vec: bytes }
}
```

Listing 2: The `utf-8` conversion function in Rust standard library. It requires an explicit unsafe to prevent the invalid values introduced into safe Rust.

4.2 Method Overview

Although modular unsafe functions constitute only a small portion in Rust, the absolute number (74.3K) remains substantial, exceeding human efforts for manual analysis. Besides, the taxonomy of modular unsafe types was previously unknown. We thus adopt an iterative clustering approach using LLMs to categorize these cases into distinct classes.

From this set, we sample 1000 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). 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 users. Detailed results are shown in Table 1. The overall classification study takes 165 person-hours for two engineers in total. Preparing the proof-of-concept exploits required an additional 90 person-hours for one engineer. Triaging each model flag during the crate evaluation takes approximately 5–15 minutes per candidate.

4.3 Classification and Root Cause Analysis

4.3.1 Categories of `mufs`. With the help of LLM, we follow the human-in-the-loop iterations described above and managed to identify following categories for `mufs`.

Breaking Logical Memory Invariants. The first class of modular unsafe operations involves breaking the logical memory invariants, especially concerning initialized and uninitialized memory. An example of this type is the *set_len* function in the `Vec` shown in **??**. Similarly, for any data structure containing uninitialized memory, modifying its logical length will allow users to access uninitialized regions, leading to errors in reading uninitialized memory.

Challenge. The challenge of detecting such a modular unsafe operation is understanding related variables as modular memory invariants rather than normal integers. We argue that this type of error is difficult for traditional static analysis to find a fixed pattern for detection, but it requires an understanding of program logic (i.e., the semantics).

Missing Checks on Precondition. The second type of modular unsafe operations is the failure to validate the modular 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 strings in Rust, shown in Listing 2. In the `utf-8` encoding, not all permutations constitute valid encodings; therefore, safe

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

Listing 3: The direct memory mapping function generated by `svd2rust`, allowing to change register values in device.

```rust
/// Unlocks this mutex.
/// This method may only be called
/// if the mutex is held in the current context
unsafe fn unlock(&self);
```

Listing 4: Unlock is a modular unsafe in Rust, as shown in the lock API from parking_lot crate.

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.

Challenge. The difficulty of detecting this type of modular unsafe error is the various preconditions in the program logic, as `utf-8` is only one example. Another example is from the `mpv` crate, where developers assume the input pointers to be an integer string but forget to validate. 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.

Violating Rust Constraints in FFI. The third type of modular 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 checks to ensure the safe Rust constraints, and for FFI functions that fail to follow the Rust constraints, developers are expected to keep the function as unsafe. For example, in `spng` crate, the library creates 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 all the buffers to be initialized to prevent uninitialized memory reads, and this constraint is not implemented in the C library.

Challenge. The difficulty of analyzing the FFI function is the challenge of a cross-language analysis between Rust and the library, which is beyond the scope of this work [29]. 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.

Mapping Invalid Value to Devices. Another category of modular unsafe operations is from the device drivers written in Rust, especially from `svd` files. As proposed as one of the open problems in [48], the boundary between safe Rust and unsafe Rust is not technically clear in embedding system development. For example, the direct memory mapping is commonly used in device drivers allowing the system to control the register values in the 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 the host system changes 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 3. 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 risk is still under discussion in `svd2rust` [49].

Challenge. The challenge of detecting such unsafe code is to reason about device status from the manual, which is far beyond the traditional program analysis's scope. But with the help of LLM, Coin tries to extend its reasoning to include relevant comments, Rustdoc and shorten this gap.

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 modular unsafe operations due to complex synchronizations. For example, in Rust's lock API shown in Listing 4, the `unlock` function is modular unsafe without any unsafe operations inside the function body. As explained in the comment, Rust requires the caller to hold the lock before calling the `unlock` method, which is difficult to analyze and reason about `rustc`.

Challenge. Detecting potential data races remains a significant challenge in program analysis. This problem is exacerbated in Rust, where the language's soundness guarantees can be undermined by subtle API misuses. For instance, a developer might prematurely release a lock, leading to a data race that is difficult to detect through conventional static analysis relying solely on API signatures. To address this, Coin leverages an LLM to infer likely API misuse patterns, thereby enabling the detection of such unsafe behaviors.

4.3.2 Unclassified Types and False-Positives. The limitations of Coin's study stem from the following two sources: Due to the high volume of modular 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 are conservatively marked as unsafe.

Unclassified modular Unsafe Types. For cases with limited context, it is challenging to determine the root causes of these modular unsafe operations, resulting in some cases remaining unclassified. Furthermore, during our empirical study, we observed that many

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

Listing 5: 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.

crates are not actively maintained and have limited program context, making it difficult for LLMs to infer the root causes in such scenarios.

False-positive Cases. According to Definition 4, only memory safety errors triggered by safe Rust are considered unsound issues, and other types of modular errors are not included. Therefore, if a function can only introduce modular 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. However, according to `BTree`'s document [55], 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 Proposed Solution

5.1 Modular Unsafe Classifier

As shown in Figure 1, Coin's first part is a modular unsafe classifier, which takes the target function and relevant contexts as input and outputs potential labels indicating whether the target function is modular unsafe.

5.1.1 Modular Unsafe Dataset (D_{muf}). We define the D_{muf} from Section 4.1 as a set of pairs consisting of a Rust function and the corresponding modular unsafe labels, where x_i is a Rust function with its context, u_i is a set of safe or modular unsafe labels (*i.e.,* $u_i = 0$ for "safe" and $u_i = 1$ for "modular unsafe"), m is the total number of the function and label pairs. We note that during training and testing steps, we **removed** all the 'unsafe' keywords from the dataset to ensure the model can not use them to predict.

5.1.2 Context Collection for Interprocedural Analysis. To assist the model in understanding the function context, we build a static analyzer as a preprocessor to recursively collect the function context and remove irrelevant code. Specifically, for the target types from the function signature, Coin recursively collects 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.

5.1.3 Problem Definition. The goal of modular unsafe classification is to design a classifier that predicts whether a given function contains modular 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 $\mathcal{M} : \mathcal{X} \to \{0, 1\}$, where '0' represents the function being safe and '1' signifies that the input function contains modular unsafe operations.

5.1.4 Model Architecture. We utilize the Llama3.2 model to perform this classification task by adding a customized classification header. 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. 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 modular unsafe operation. The full classifier prompt is provided in Appendix B.

5.1.5 Weighted Loss and Metric Calculation. To effectively tackle class imbalance between a large number of safe functions and rare modular 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. To handle the imbalance, we focus on the precision of unsafe label classification during training and evaluation processes.

5.1.6 LoRA Fine-tuning. Fine-tuning large language models often requires substantial GPU memory. To reduce this requirement, we applied LoRA to Llama3.2. LoRA freezes original weights and adds small-rank update matrices into selected linear layers, enabling efficient adaptation with 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

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

Listing 6: The PoC code of breaking memory invariants. The uninitilized memory is introduced into safe Rust.

capture task-specific patterns for classifying safe versus unsafe Rust functions.

5.1.7 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 [42]. We consider a rigorous thresholding approach that comes with a PAC guarantee based on conformal prediction [60, 62].

Algorithm. We adopt the PAC conformal set algorithm [38, 39] for thresholding. Let $\bar{\theta}$ be the upper Clopper-Pearson (CP) bound [12], where the binomial parameter μ is included with high probability, *i.e.,* $\bar{\theta}(k; m, \delta) \coloneqq \inf\{\theta \in [0,1] \mid F(k; m, \theta) \leq \delta\} \cup \{1\}$, 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:

$$\hat{\tau} = \arg\max_{\tau\in\mathbb{R}_{\geq 0}} \tau \qquad \text{subj. to} \qquad \bar{\theta}(k; |S_{\text{cal}}|, \delta) \leq \varepsilon \tag{1}$$

where S_{cal} is the set of unsafe functions in S_{val}, *i.e.,* $S_{\text{cal}} \coloneqq \{(x, u) \in S_{\text{val}} \mid u = 1\}$, and k is the number of unsafe functions that are missed by a threshold, *i.e.,* $k \coloneqq \sum_{(x,u)\in S_{\text{cal}}} \mathbb{1}(\hat{u}(x) < \tau)$. We applied a 80% recall expectation for Coin during the thresholding step.

5.2 PoC Generator

Due to the various complex root causes of different modular unsafe operations, it is non-trivial for developers to understand and reason about the root causes of modular unsafe operations. Therefore, we propose an LLM-based PoC generator that attempts to generate PoC code that demonstrates root causes.

5.2.1 PoC Dataset (D_{poc}). Unlike the D_{muf}, the PoC dataset, noted as D_{poc} 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 modular unsafe functions and their corresponding PoC code. D_{poc} dataset includes all the categories of modular unsafe operations and their PoCs we studied in Section 4. Due to the difficulty and engineering efforts of manually constructing PoC code and explanations, we only built at most 20 cases per type of `muf`, following state-of-the-art [37]. We fine-tuned our PoC generator on D_{poc} with the collected context and evaluated its performance on newfound bugs to ensure the model never learns the PoC code. Below we demonstrate several PoCs for each type of modular unsafe functions.

Breaking Logical Memory Invariants. 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 6. The PoC first manipulates the usable memory size and introduces uninitialized memory into the `Vec`, then attempts to access the uninitialized memory.

```
struct A{}
impl A {
    pub const fn as_bytes(&self) -> &[u8] {
        // an invalid utf-8 encoding
        [0xC0, 0x80].as_slice()
    }
}
fn main() {
    obfstr::obfstr!(A{});
}
```

Listing 7: The PoC code demonstrates how introducing invalid `utf-8` encoding in Rust can trigger undefined behavior.

```
struct A{ f: File }
impl Read for A {
    fn read(&mut self, buf: &mut [u8])
        -> std::io::Result<usize> {
        let _ = buf[0]; // uninitialized memory
        return self.f.read(buf);
    }
}
fn main(){
    if let Ok(png) =File::open("test.png") {
        let a = A{f:png};
        spng::decode(a, spng::Format::Rgba8);
    }
}
```

Listing 8: 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.

Missing Checks on Precondition. The PoC code in Listing 7 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.

Violating Rust Constraints in FFI. The POC of uninitialized memory read error in `spng` crate is shown in Listing 8. To trigger the error in the C part, we need to prepare the input from the Rust side and call the buggy FFI functions to trigger the bug. The Coin is expected to reason about the FFI function behaviors from the input document and function names. To prepare the input data, Coin is expected to collect information from the input context and properly build the input.

Mapping Invalid Value to Devices. For this type of error, the POC requires a hardware device to work with, so Coin isn't expected to generate the POC. The expected inputs are usually documented in the comments or specifications.

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

Listing 9: The PoC generated by Coin for modular unsafe unlock function in anode crate, the code triggers data races with safe Rust.

Data Race. The POC of incorrect implementation of `unlock` is shown in Listing 9: after releasing the lock, the thread still holds a shareable reference and triggers data races. Besides the `unlock` error, there are different potential data races and Coin is expected to reason about the APIs from the input context and tries to build the POC.

5.2.2 Problem Definition. The task of generating PoC code from modular unsafe functions can be formulated as a sequence-to-sequence (seq2seq) translation problem. 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 modular 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. The training objective is as follows:

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

where 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, METEOR, 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 modular unsafe operations. The full PoC generation prompt is provided in ??.

Our goal with this seq2seq approach is to assist developers in identifying and understanding the root causes of modular 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.

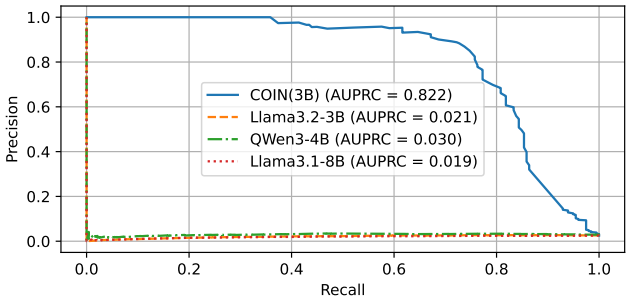


Figure 2: Precision-recall (AUPRC) evaluation of different models on the modular unsafe classification task. Coin achieves 63.71% precision with 80.42% recall, out-performing other general models.

6 Evaluation

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

- What's the performance of Coin's fine-tuned model compared with other prompt-based LLMs?
- How effective is Coin in real-world unsound bug hunting?
- How is Coin compared with other static/dynamic analysis tools?
- Can Coin's PoC generator help developers understand the modular unsafe root causes?
- What are the practical impacts of modular unsafe bugs found by Coin?

Environment Setup and Performance Overhead. We fixed the Rust toolchain version to be 1.83.0-dev. 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. The Coin's performance overhead includes two parts: the fine-tuning process takes a one-time 100 GPU-hours, and the inference of 12K crates takes about 48 GPU-hours.

6.1 Comparing with Vanilla LLMs

We compare Coin with the latest LLM models, including both open-source ones (QWen3-4B [44], Llama3.1-8B [30] and Llama-3.2-3B [31]) and closed-source ones (OpenAI's GPT-4o [35] and Anthropic's Claude-3.7 [1]). We note that Coin's classifier is fine-tuned based on Llama3.2-3B model[2].

6.1.1 Open Source Models. We split the modular unsafe dataset (D_{muf}) 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 the unsafe label to demonstrate unsafe classification results, which represent the model's ability to classify modular unsafe labels. As shown in Figure 2, Coin achieves satisfactory performance with an AUPRC of 0.82. For other general models, since they are not trained or fine-tuned specifically for modular unsafe Rust classification, their performance is limited.

[2]To demonstrate Coin's generality, we also trained a model based on `QWen2.5-1.5B`, details are shown in Appendix E.

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%		

Table 2: 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 improves the model's performance.

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%		

Table 3: Best of K examples evaluation on GPT-4o and Claude-3.7 with N=1 shot example.

PAC Thresholding. In practice, a threshold is picked based on the AUPRC graph to power the model in the inference step. We use the trusted thresholding algorithm proposed in (1) for $\hat{\tau}$, and a chosen threshold provides 80% 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 modular unsafe dataset with PAC thresholding is 63.71% precision and 80.42% recall.

6.1.2 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 2. 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 modular unsafe operations and identify them. Through fine-tuning, Coin improves the precision and recall of detecting modular 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 3. The results show that even with more chances, the general models are struggling to correctly resolve the problem compared with Coin.

6.2 Modular Unsafe Bug Hunting

To demonstrate Coin's end-to-end effectiveness, we evaluated Coin on open-source Rust crates to identify modular unsafe operations that were overlooked by developers. Due to GPU resource constraints, we evaluated based on the crates' popularity: we first uniformly sampled 10K crates from crates.io[3], ran Coin on their repositories, and attempted to locate modular unsafe operations. To better evaluate the impact of `muf`, we exhaustively evaluated the top 2k downloaded crates as additional evaluation. The overall results are summarized in Table 4.

Crucially, Coin successfully discovered 26 *previously unknown* modular unsafe bugs across various crates , with 58.1% precision, higher than existing Rust analyzers shown in Table 6. We have reported all of them to corresponding maintainers following the responsible disclosure procedure. So far, 14 of these bugs have been confirmed (10 have been fixed), and we have obtained 10 CVE IDs. The identified vulnerabilities span all categories of modular unsafe Rust issues, showing Coin's effectiveness in detecting different modular unsafe bug patterns.

False-positive Analysis. While Coin maintains decent precision in detecting modular unsafe functions, some false-positives occur for the following reasons: 1. The restricted analysis scope results in false-positives from FFI functions, as Coin does not analyze beyond its language. 2. Certain crates suffer from poor code quality, which includes cumbersome functions, making soundness verification difficult.

Comparing with Vanilla LLMs. We tested two vanilla models, GPT-4o and Claude-3.7 on the modular unsafe bugs identified by Coin, and the results are shown in Table 4. Among the 26 bugs, GPT-4o identified 13, and Claude-3.7 identified 12. However, given their precision rates, these models introduce many false positives when applied to the entire dataset.

Comparing with Static Analysis. Besides LLM-based approaches, we further compare Coin with traditional static analysis tools, including Rudra [6], MirChecker [24], and ffi-checker [25]. 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 run static analysis tools on all the modular unsafe bugs found by Coin to check if the tools can detect such errors. As shown in Table 4, none of the three tools detected all types of `muf`, for structurally distinct reasons. Rudra operates exclusively on `unsafe-marked` code blocks. MirChecker and ffi-checker instrument explicit unsafe memory operations at the LLVM IR level; `muf` contains no such operations to instrument.

For *the logical requirement* `muf` involving array-bounds checks, a range-tracking analysis could, in principle, detect whether an index is validated before use without requiring semantic understanding. However, Coin addresses `muf` categories where the safety contract exists only in documentation or API conventions — not in types — and where detection requires cross-module compositional reasoning that local dataflow analyzes cannot express.

Comparing with Dynamic Analysis. For dynamic analysis, specifically fuzzers, we compare Coin with two fuzzers: RPG [63] and RUG [10]. RPG constructs type dependency graphs to assist in generating fuzzing harnesses, producing a sequence of API calls through graph traversal. It also supports generic parameters and prioritizes functions containing unsafe regions using a pool-based generator. RUG is an LLM based testing generator that leverages

[3]We used the data dump on January 15, 2025, at which time the crates.io contained approximately 164K crates.

Crate	Status	Unsafe Type	CVE	Downloads / Version	Coin	Rudra	Mir Checker	FFI Checker	RPG	RUG	GPT -4o	Claude -3.7	PoC
sqlite3-parser	P	PC	✓	1.6M / 0.13.0	✓	✗	✗	✗	N/A	N/A	✓	✓	✓
obfstr	P	PC	✓	1.3M / 0.4.4	✓	✗	✗	✗	✗/5	✗/28	✓	✓	✗
ruru	W	PC		47.8K / 0.9.3	✓	✗	✗	✗	✓/300	✗/199	✓	✓	✓
mpv-client	P	PC		17.2K / 1.0.1	✓	✗	✗	✗	✗/22	✓/48	✓	✗	✓
spiral-rs	W	PC	✓	4.7K / 0.2.1	✓	✗	✗	✗	N/A	✓/194	✓	✗	✓
memory_pages	W	PC	✓	1.3K / 0.1.0	✓	✓	✓	✗	✗/300	✓/30	✗	✗	✓
trailer	W	PC	✓	36K / 0.1.2	✓	✗	✓	✗	✓/14	✓/6	✓	✓	✓
triomphe	C	PC		36M / 0.1.14	✓	✗	✗	✗	✗/179	✗/113	✗	✗	✓
rustybuzz	P	MI		3.8M / 0.20.1	✓	✗	✗	✗	✗/300	N/A	✗	✗	✓
opencl3	P	MI		0.2M / 0.10.0	✓	✗	✗	✗	N/A	N/A	✗	✓	✓
scsir	W	DM	✓	2.0K / 0.2.0	✓	✗	✗	✗	N/A	N/A	✗	✗	N/A
efm32gg11b	W	DM		2.6K / 0.1.0	✓	✗	✗	✗	N/A	N/A	✗	✗	N/A
rsl10-pac	W	DM		1.9K / 0.0.2	✓	✗	✗	✗	N/A	N/A	✗	✗	N/A
rula	C	FFI		1.9M / 0.0.1	✓	✗	✗	✗	N/A	N/A	✗	✓	✓
spng	C	FFI		31.7K / 0.2.0	✓	✓	✓	✓	N/A	N/A	✓	✓	✗
libucl	W	FFI		10.8K / 0.2.3	✓	✗	✗	✓	✗/1300	✓/39	✓	✗	✓
libblkid-rs	P	FFI		0.2M / 0.3.2	✓	✗	✗	✗	N/A	N/A	✗	✗	✓
winit	P	FFI		19.2M / 0.30.8	✓	✗	✗	✗	N/A	N/A	✗	✗	✓
stivale	W	FFI		4.3K / 0.2.2	✓	✗	✗	✓	✓/300	✓/42	✓	✗	✓
xrdb	P	FFI		3.5K / 0.1.1	✓	✗	✗	✗	✓/40	✗/6	✗	✗	✓
anode	P	DR	✓	1.3K / 0.1.0	✓	✗	✗	✗	✓/300	✓/103	✓	✓	✓
process-sync	W	DR	✓	7.2K / 0.2.2	✓	✗	✗	✗	✗/15	✗/19	✓	✓	✓
process_lock	W	DR	✓	2.1K / 0.1.0	✓	✗	✗	✗	✗/8	✗/19	✗	✗	✓
triomphe	P	DR		36M / 0.1.14	✓	✗	✓	✗	✗/179	✗/113	✗	✗	✗
xarc	W	DR		3.8K / 0.3.0	✓	✗	✗	✗	✗/27	✗/4	✓	✓	✓
wgp	C	DR	✓	3.7K / 0.2.0	✓	✗	✗	✗	✗/19	✗/11	✓	✓	✓
Total	-	-	10	-	26	2	4	3	5	7	13	12	20

Table 4: Evaluation result on finding missed modular 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. Due to rust toolchain versions, dynamic fuzzers failed to compile on some crates, noted as 'N/A'

LLM with chain-of-thought to build the testing context and fuzzers to explore different paths. We evaluated both baseline tools against the dataset of modular unsafe bugs identified by Coin. For each target crate, we employed the tools to automatically synthesize fuzzing harnesses; the total counts are detailed in Table 4. Each fuzzing campaign ran for a maximum of 24 hours or until the first crash occurred. We then manually triaged all reported crashes to verify whether they reproduced the specific `muf` issues.

The evaluation results for the fuzzing baselines are detailed in Table 4. Among the 15 compilable target crates, RPG successfully triggered 5 crashes, while RUG identified 7 issues. Compared to Coin, both fuzzers exhibited lower detection rates. This disparity is primarily due to the latent nature of modular unsafety; because these vulnerabilities lack internal triggers within the library's codebase, automated harness generators struggle to synthesize the specific sequence of API calls required to violate invariants and expose undefined behavior.

Distribution breakdown. We present the CDF distribution of bugs discovered by Coin against the download-rank of their respective crates. As summarized in Table 5, `muf` bugs are not confined to the long tail of the registry; they also appear in widely adopted crates that undergo active code review. Among the 10,000 randomly sampled crates, Coin identified bugs in 6 crates ranked within the top-10K by download count: triomphe (rank 803), winit (1,143), rustybuzz (2,044), sqlite3-parser (4,594), obfstr (5,456), and opencl3 (9,520).

Top # by downloads	# evaluated	Percentage	Bugs	Percentage
2,000	2000	16.7%	3	11.5%
5,000	2,311	19.3%	4	15.4%
10,000	2,612	21.8%	6	23.1%
50,000	5,039	42.0%	12	46.2%
150,000	11,147	92.9%	26	100.0%
164,000 (Total)	11,976	100.0%	26	100.0%

Table 5: The distribution of identified `muf` by Coin among sampled crates evaluation and cumulative percentage of the whole evaluation dataset, indicating the `muf` exists in both actively maintained crates and newly developed crates.

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

Table 6: Precision 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.

```
// Context omitted for simplicity
let mut v = b"SELECT␣?␣".to_vec();
v.extend_from_slice(&[0xC0, 0x80]);
let mut parser = Parser::new(&v);
...
```

Listing 10: The PoC code generated by Coin for invalid utf-8 strings in sqlite3-parser crate, a crash will be triggered whenever the invaid string is accessed.

6.3 Precision/Recall Study

We conducted a precision–recall study of Coin using a benchmark derived from the 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 modular 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 modular unsafe dataset before fine-tuning.

The evaluation results are shown in Table 6: Coin achieves 57.9% precision and 76.6% recall on the standard library dataset. The Coin's false-positive cases are mainly from developers' preventive markings described in Section 4.3.2 and false-negative cases are mainly from different hardware-specific intrinsics. Compared to static analysis tools, Coin demonstrates better precision/recall, making it suited for Rust program analysis.

6.4 PoC Generator Evaluation

To evaluate the performance of Coin's PoC generator, we applied the newly found modular unsafe operations as inputs for the PoC generator and checked 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 PoCs in our bug reports to developers.

The PoC generation results are shown in the `PoC` column in Table 4. Coin successfully generates the PoC indicating the root causes for 19 out of 22 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 exclude the bug type of DM from PoC generation (N/A) since the device driver requires a special environment to build.

6.5 Case Study

```
// helper function
fn drop_slow(this: *mut _, old_ref: usize) {
 match old_ref {
      2 => (*this).waker.wake(),
      1 => drop(Box::from_raw(this)),
      _ => {}
 }
}
// drop implementation, which is 'safe'
let old_ref= self.deref().refcount
 .fetch_sub(1, Ordering::Release);
if old_ref > 2 {
 return;
}
self.deref().refcount.load(Ordering::Acquire);
unsafe { drop_slow(self.0.as_ptr(), old_ref) }
```

Listing 11: 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 details are omitted for simplicity

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 10. The PoC introduced the invalid utf-8 encoding bytes, 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.

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 11, 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 11. 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.

7 Limitation

In this section, we discuss the potential limitations of Coin. The first limitation of Coin is the limited study of modular 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.

8 Related Work

Understanding unsafe Rust. As the soundness of unsafe is essential to Rust's safety guarantee, several attempts have been made to understand its uses and bug patterns from existing Rust projects and their CVEs [13, 21], . In Coin, we take this one step further to proactively discover modular unsafe operations and automate their detection at a large scale. The unprecedented number of new memory safety bugs Coin found changes the perspective of these empirical studies; it is much more subtle and error-prone to write completely sound unsafe code, and it is even difficult for Rust experts and language designers.

Unsafe Rust and its usage. Although software engineers try to avoid `unsafe` regions in their programs to prevent potential memory safety problems [16], many Rust crates are using "unsafe" more frequently, leading to implicit usage and widespread unsafe blocks in Rust binaries [9, 14]. In addition, studies [3, 43] show that these unsafe usages are typically justified by valid or unavoidable reasons, which are often difficult to eliminate. While `unsafe` catches the attention of programmers on memory safety, it can also be utilized by reverse engineers to identify the weaknesses in the given Rust binaries.

Formal methods and verification. Rust, being a new programming language, has seen a lot of community effort into building formal foundations (i.e., type system and operational semantics) with various design goals [13, 20, 21], e.g., proving the correctness of encapsulated unsafe code with an extensible semantic typing [21] and an aliasing model to validate raw pointers [20]. Being an early-stage language, much of the existing verification work for Rust focuses on transpiling Rust code or IR to existing verification frameworks: C [11, 58], Viper IR [4, 33], and LLVM IR [7]. These are promising directions in verifying memory safety or correctness at various layers, but, unlike Coin, it is fundamentally difficult to apply them to the entire ecosystem. scalability is limited by design: lack of generic type awareness, limited performance, or reliance on manual annotations.

Program analysis. Existing Rust static analyzers — Rudra [6], MirChecker [24], and ffi-checker [25] — target explicitly `unsafe`-marked code and specific bug classes (lifetime violations, FFI type mismatches). They are structurally ill-suited to `muf` detection for reasons specific to some categories. For *logical memory controls* (e.g., the invariant that a `set_len` argument not exceed buffer capacity) and *hardware feature* categories (SIMD preconditions), the safety contract exists only as a prose convention in documentation; no static rule can express it without manual annotation. For *FFI* categories, ffi-checker requires foreign functions to carry explicit safety annotations; undocumented C-side preconditions — pointer aliasing requirements, lifetime contracts — have no annotation to check against.

Coin provides a single unified detector trained across the full taxonomy, performing well even on categories — documentation-only contracts, cross-module semantic invariants — where rule-based approaches have no foothold.

LLM for Security. LLMs have been widely applied to software security, including vulnerability detection [26, 40], zero-shot repair [41], and exploit generation. IRIS [26] combines LLMs inference with CodeQL-based static analysis, showing that LLMs and static analyzers are complementary.

Recent frontier models have further raised this capability ceiling. Claude Mythos [2], Anthropic's latest general-purpose model, exhibits unprecedented autonomous security reasoning: it uncovered decades-old zero-days in OpenBSD and FFmpeg, built a 20-gadget ROP chain for a FreeBSD NFS exploit, and saturated standard vulnerability benchmarks, all from general advances in code understanding and reasoning rather than security-specific training. OpenAI has similarly launched dedicated security initiatives [36].

Yet our evaluation shows that these general-purpose models perform poorly on MUF detection: GPT-4o attains 4.6% precision / 21.4% recall and Claude-3.7 4.2% / 7.1%, versus Coin's 63.7% / 80.4%. Coin's fine-tuned Llama 3.2 3B, outperforming much larger frontier models on this focused task, aligns with prior work showing that targeted fine-tuning on domain-specific corpora outperforms general prompting for specialized semantic properties [26].

9 Conclusion

In this work, we propose Coin, a LLM-based source code level static analysis framework to detect modular unsafe operations that may be overlooked by developers and the Rust compiler. These modular unsafe operations can cause unsound issues in Rust and introduce security risks for the entire system. We first conduct an empirical study with the help of LLM to categorize the root causes into five classes. Then we propose a modular unsafe classifier to detect potential modular unsafe operations and a PoC generator to assist developers in understanding the unsound issues. Our evaluation shows Coin can help developers find missed modular unsafe operations (26 newly found bugs and 10 CVE ids) that existing static analysis tools failed to detect and generate reasonable PoC code for further investigation. Besides, by comparing with existing static/dynamic bug hunting tools for Rust, Coin demonstrates its uniqueness in finding `mufs` errors.

10 Acknowledgment

We thank our anonymous reviewers for their valuable reviews and our shepherd for suggestions and detailed feedback. We also thank our anonymous artifact evaluation reviewers for their patience and suggestions. Finally, we thank our previous reviewer A from USENIX Security'26 cycle 1, for providing guidance on the problem insight and definition of `mufs`. This research was supported, in part, by the ONR under grant N00014-23-1-2095, IITP grant (No. RS-2024-00509258 and No. RS-2024-00469482), NRF grant (RS-2025-00560062), and gifts from Meta, Mozilla, Intel, VMware and Google.

A Ethical Considerations

We automatically collect Rust packages from crates.io. All data collected is publicly available, and we only use this data for research purposes. All the new unsound issues found by Coin are reported to the corresponding maintainers and strictly follow the 90-day disclosure policy. All the CVEs have been reported to the corresponding maintainers.

B Open Science

To promote transparency and reproducibility in our research, the data artifacts of this paper is available at https://github.com/CXWorks/coin-ae.

C Generative AI Usage

AI tools (e.g., ChatGPT) have been used to check grammar, polish the sentences, and revise the paper for scientific presentations.

References

[1] Anthropic. Claude Sonnet 3.7 system card. Available at https://www.anthropic.com/claude-3-7-sonnet-system-card, Feb 2025. Accessed Dec 2025.
[2] Anthropic. Claude mythos preview, 2026.
[3] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. How do programmers use unsafe rust?, 2020.
[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.
[5] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.
[6] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, October 2021.
[7] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. Verifying rust programs with smack. In *Automated Technology for Verification and Analysis: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings 16*, pages 528–535. Springer, 2018.
[8] Hung-Mao Chen, Xu He, Shu Wang, Xiaokuan Zhang, and Kun Sun. TYPEPULSE: Detecting Type Confusion Bugs in Rust Programs. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.
[9] Xiang Cheng, Sangdon Park, HyungSeok Han, Xiaokuan Zhang, and Taesoo Kim. Ruby: Unmasking unsafe rust in stripped binaries via machine learning. In *Proceedings of the 56th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2026.
[10] Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang, and Taesoo Kim. Rug: Turbo llm for rust unit test generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 634–634. IEEE Computer Society, 2025.
[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*, pages 168–176. Springer, 2004.
[12] Charles J Clopper and Egon S Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.
[13] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2019.
[14] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers?, 2020.
[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
[16] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. Benefits and drawbacks of adopting a secure programming language: rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 597–616, 2021.
[17] 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.
[18] Google Security Blog. Supporting Use of Rust in Chromium. Blog post, 2024.
[19] https://cve.mitre.org. CVE-2017-6074. https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues, 2017.
[20] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for rust. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.
[21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language, dec 2017.
[22] Taesoo Kim. Are we done yet? our journey to fight against memory-safety bugs, 2021.
[23] The Rust Programming Language and Contributors. Meet safe and unsafe. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html, Accessed: 2025.
[24] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 2183–2196, 2021.
[25] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Detecting cross-language memory management issues in rust. In *European Symposium on Research in Computer Security*, pages 680–700. Springer, 2022.
[26] Ziyang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities. In *International Conference on Learning Representations*, volume 2025, pages 35735–35758, 2025.
[27] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with xrust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 234–245, New York, NY, USA, 2020. Association for Computing Machinery.
[28] Nicholas D Matsakis and Felix S Klock. The rust language, 2014.
[29] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *NDSS*, pages 1–18, 2022.
[30] 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/, 2024. Accessed Dec 2025.
[31] 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/, 2024. Accessed Dec 2025.
[32] 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.
[33] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016.
[34] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
[35] OpenAI. GPT-4o system card. arXiv preprint arXiv:2410.21276, 2024. Available at https://arxiv.org/abs/2410.21276.
[36] OpenAI. Openai security initiatives. https://openai.com/news/security/, 2025. Accessed: May 2026.
[37] OpenAI. Supervised fine-tuning. https://platform.openai.com/docs/guides/supervised-fine-tuning, 2025. Accessed: 2025-12-22.
[38] Sangdon Park, Osbert Bastani, Nikolai Matni, and Insup Lee. Pac confidence sets for deep neural networks via calibrated prediction. In *International Conference on Learning Representations*, 2020.
[39] Sangdon Park, Edgar Dobriban, Insup Lee, and Osbert Bastani. PAC prediction sets under covariate shift. In *International Conference on Learning Representations*, 2022.
[40] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Communications of the ACM*, 68(2):96–105, 2025.

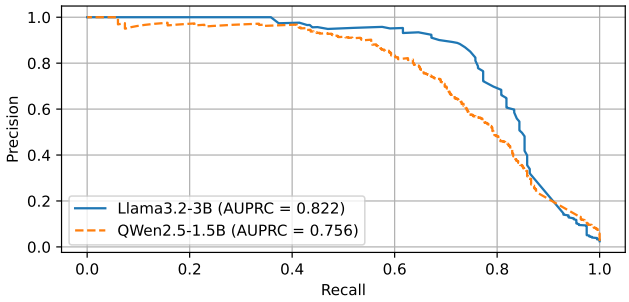


Figure 3: Precision-recall (AUPRC) evaluation of QWen2.5-1.5B model and Llama3.2-3B model. The Coin uses Llama3.2-3B model for evaluation in the paper.

[41] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE symposium on security and privacy (SP)*, pages 2339–2356. IEEE, 2023.
[42] John Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 1999.
[43] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 763–779, 2020.
[44] QwenLM. Qwen3: Think Deeper, Act Faster. Blog post, https://qwenlm.github.io/blog/qwen3/, 2025. Accessed Dec 2025.
[45] Eric C. Reed. Patina : A formalization of the rust programming language, 2015.
[46] Rust for Linux Contributors. Rust-for-Linux/linux, 2023.
[47] Rust-GPU Contributors. Rust-GPU/Rust-CUDA, 2024.
[48] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. Rust for embedded systems: Current state and open problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2296–2310, 2024.
[49] svd2rust contributors. Issue #714: Compatibility with new version of svdtools. https://github.com/rust-embedded/svd2rust/issues/714, 2024. Accessed: 2025-12-05.
[50] Android Security Team. Eliminating memory safety vulnerabilities in android. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html, September 2024. [Online; accessed 2025-12-03].
[51] The Rust Team. Rust: A language empowering everyone to build reliable and efficient software, 2010.
[52] The Rust Team. Unsafe operations in rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#unsafe-superpowers, 2018.
[53] The Rust Team. Behavior considered undefined. https://doc.rust-lang.org/reference/behavior-considered-undefined.html, 2024.
[54] The Register. Microsoft to explore Windows 11 code written in Rust. Online article, 2023.
[55] The Rust Project Developers. BTreeMap in The Rust Standard Library (std). https://doc.rust-lang.org/std/collections/struct.BTreeMap.html, 2024. Accessed: 2025-05-05.
[56] The Rust Project Developers. The rustonomicon: Working with unsafe, 2024. Accessed: 2025-12-20.
[57] David Tolnay. Soundness bugs in rust libraries: can't live with 'em, can't live without 'em, 2019.
[58] John Toman, Stuart Pernsteiner, and Emina Torlak. Crust: a bounded verifier for rust (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 75–80. IEEE, 2015.
[59] Jeff Vander Stoep. Rust in Android: move fast and fix things. Google Online Security Blog, November 2025. Accessed: 2026-01-15.
[60] Vladimir Vovk. Conditional validity of inductive conformal predictors. *Machine learning*, 92(2-3):349–376, 2013.
[61] Wikipedia contributors. Undefined behavior. Wikipedia, The Free Encyclopedia, Accessed: 2025.
[62] Samuel S Wilks. Determination of sample sizes for setting tolerance limits. *The Annals of Mathematical Statistics*, 12(1):91–96, 1941.
[63] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda 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*, pages 1–13, 2024.
[64] Yichi 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*, pages 683–685, 2024.

D Vector Implementation Details

In this section, we demonstrate how a modular unsafe function can be utilized to bypass `rustc` analysis and break safe Rust's memory safety guaranty. We start with a safe implementation of `Vec` in Rust; the code is shown in https://godbolt.org/z/Y1zqfWTas.

However, modular unsafe functions can bypass `rustc` analysis. Therefore, a malicious `Vec` from attackers can be implemented as https://godbolt.org/z/xPYnhqP14. We note that the malicious `Vec` can pass the `rustc` analysis and provide seemingly 'safe' APIs for developers. Nevertheless, these APIs can trigger undefined behaviors and break the memory safety guaranty provided by safe Rust.

E Coin's Training on QWen2.5-1.5B

To further demonstrate the generality of Coin's solution, we also trained modular unsafe classification on `QWen2.5-1.5B` model. The precision/recall evaluation of QWen2.5 model is shown in Figure 3, achieving comparable classification accuracy compared with `Llama3.2-3B` model.